IOCost: Block IO Control for Containers in Datacenters

Tejun Heo
Meta Inc.
Menlo Park, USA
htejun@fb.com

Dan Schatzberg
Meta Inc.
Menlo Park, USA
dschatzberg@fb.com

Andrew Newell
Meta Inc.
Menlo Park, USA
newella@fb.com

Song Liu
Meta Inc.
Menlo Park, USA
songliubraving@fb.com

Saravanan Dhakshinamurthy
Meta Inc.
Menlo Park, USA
saravanand@fb.com

Iyswarya Narayanan
Meta Inc.
Menlo Park, USA
inarayanan@fb.com

Josef Bacik
Meta Inc.
Menlo Park, USA
jbacik@fb.com

Chris Mason
Meta Inc.
Menlo Park, USA
clm@fb.com

Chunqiang Tang
Meta Inc.
Menlo Park, USA
tang@fb.com

Dimitrios Skarlatos
Carnegie Mellon University
Pittsburgh, USA
dskarlat@cs.cmu.edu

ABSTRACT

Resource isolation is a fundamental requirement in datacenter environments. However, our production experience in Meta's large-scale datacenters shows that existing IO control mechanisms for block storage are inadequate in containerized environments. IO control needs to provide proportional resources to containers while taking into account the hardware heterogeneity of storage devices and the idiosyncrasies of the workloads deployed in datacenters. The speed of modern SSDs requires IO control to execute with low-overheads. Furthermore, IO control should strive for work conservation, take into account the interactions with the memory management subsystem, and avoid priority inversions that lead to isolation failures.

To address these challenges, this paper presents IOCost, an IO control solution that is designed for containerized environments and provides scalable, work-conserving, and low-overhead IO control for heterogeneous storage devices and diverse workloads in datacenters. IOCost performs offline profiling to build a device model and uses it to estimate device occupancy of each IO request. To minimize runtime overhead, it separates IO control into a fast per-IO issue path and a slower periodic planning path. A novel work-conserving budget donation algorithm enables containers to dynamically share unused budget. We have deployed IOCost across the entirety of Meta's datacenters comprised of millions of machines, upstreamed IOCost to the Linux kernel, and open-sourced our device-profiling tools. IOCost has been running in production for two years, providing IO control for Meta's fleet. We describe the design of IOCost and share our experience deploying it at scale.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507727

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; **Input / output**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Datacenters, Operating Systems, I/O, Containers

ACM Reference Format:
Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, Dimitrios Skarlatos. 2022. 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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507727

1 INTRODUCTION

Containers are swiftly evolving into one of the primary mechanisms for virtualizing capacity in modern datacenters. Containers virtualize resources at the operating system level and provide a lightweight and consistent environment that can be deployed and run anywhere with ease. Numerous container solutions are available today by all major cloud providers such as Amazon's AWS [34], Google's Cloud [9], and Microsoft's Azure [3]. Containers have also been taking over private datacenters, with Facebook's complete fleet running solely on containers [39]. As containers enable higher levels of application consolidation, it is important to build effective control and isolation mechanisms.

Resource isolation for compute, memory and network have been the focus of a large body of research with many improvements

landing in Linux. However, our production experience in Meta's large-scale datacenters shows that existing IO control mechanisms (e.g., BFQ [41]) for block storage are inadequate in containerized environments.

There are several challenges in providing robust IO control for containers. First, IO control needs to account for hardware heterogeneity in datacenters. Multiple generations of SSDs, spinning disks, local/remote storage, and novel storage technologies may all be available in a single datacenter. Hardware heterogeneity is further amplified by their vastly different performance characteristics in terms of latency and throughput, not only across different types of devices such as SSDs and hard drives, but also within a type. Effective control further needs to take into consideration SSD idiosyncrasies that may over-exert their performance in short bursts and then slow down drastically, adversely affecting a stacked environment [5–7, 14, 19, 28, 44, 45].

Second, IO control needs to cater to the constraints of a wide variety of applications. For instance, some applications are latency-sensitive while others benefit primarily from increased throughput, while yet others might perform sequential or random accesses, in bursts or continuously. Unfortunately, identifying a balance point between latency and throughput is particularly challenging when device heterogeneity and application diversity are combined at the datacenter scale.

Third, IO isolation needs to provide a set of properties required in datacenters. Work conservation is desirable in order to deliver high utilization and avoid idle resources. In addition, some IO control mechanisms rely on strict prioritization, which fails to provide fairness when equal priority applications share a machine. Furthermore, application developers often cannot effectively estimate IO needs in terms of metrics like IOPS on a per-application and per-device basis. IO control mechanisms should be easy for application developers to reason about and configure. Finally, IO isolation has interactions with memory management operations such as page reclaim and swap. IO control must be aware of these interactions to avoid priority inversions and other isolation failures.

Previous work in IO control has mostly focused on VM-based virtualized environments with various proposals that aim to enhance the hypervisor [16, 17, 22, 36]. These approaches do not take into account the intricacies of containers such as a single shared operating system, the interactions of IO with the memory subsystem, and heavily stacked deployments. The state-of-the-art solutions in the Linux kernel rely on either BFQ [41] or limits based on a max bandwidth usage through IOPS or bytes [29]. However, these fail to be sufficiently work-conserving, lack integration with the memory subsystem or add excessive performance overheads for fast storage devices.

In this work we introduce *IOCost*, a complete IO control solution that holistically addresses the challenges of heterogeneous hardware devices and applications while satisfying the IO isolation needs of containers at the datacenter scale, and taking into consideration interactions with memory management. The primary insight behind IOCost is that the major challenge in IO control is the lack of understanding of device occupancy. It becomes apparent when we compare existing IO control with CPU scheduling. CPU scheduling relies on techniques such as weighted fair queuing [4, 13] to proportionally distribute CPU occupancy by measuring CPU time consumption. In contrast, metrics like IOPS or bytes are poor measures for occupancy, particularly given the wide diversity of block devices. Modern block devices rely heavily on internal buffering and complicated deferred operations such as garbage collection, which cause issues for techniques reliant on device time sharing or ensuring fairness primarily based on IOPS or bytes.

IOCost works by estimating device occupancy of each IO request using a device-specific model. For example, a 4KB read would have a different cost on a high-end SSD than on a spinning disk. With a model of occupancy and additional QoS parameters which account for modeling inaccuracies and determine how heavily to load the device, IOCost distributes occupancy fairly among containers. System administrators or container management systems configure weights along the container hierarchy to ensure individual containers or groups of containers receive a certain proportion of IO service. IOCost further introduces a novel work-conserving budget donation algorithm that allows containers to efficiently transfer their spare IO budget to other containers.

We have deployed IOCost across Meta's fleet. Our evaluation demonstrates that IOCost outperforms other solutions to provide proportional, work-conserving, and memory-management-aware IO control with minimal overhead. Specifically, we show that IOCost successfully isolates IO operations in a stacked ZooKeeper [23] deployment, whereas previous solutions fail to provide a workable solution. To demonstrate the broad applicability of IOCost, we successfully validated it in public cloud VMs using remote storage such as AWS Elastic Block Store and Google Cloud Persistent Disk.

The contributions of this paper are as follows:

- IOCost introduces a container-aware, scalable, work-conserving, and low-overhead IO control for modern storage devices.
- We describe a modeling technique to estimate IO device occupancy for heterogeneous applications and devices. To compensate for model inaccuracy, IOCost makes runtime adjustments to IO control based on realtime statistics about cgroup usage and IO completion latency.
- We present a working-conserving algorithm that enables containers that do not fully utilize their IO budget to donate it proportionally to other containers in a cgroup hierarchy.
- To minimize runtime overhead, we separate IO control into a fast per-IO issue path and a slower periodic planning path.
- We present a detailed evaluation of IOCost and demonstrate how existing IO control mechanisms cannot match IOCost's set of features and performance.
- We have deployed IOCost across the entirety of Meta's datacenters comprising millions of machines, upstreamed IOCost to the Linux kernel, and open-sourced our device-profiling and benchmarking tools[1].

2 BACKGROUND

In this section, we first provide a brief overview of cgroup [20] for configuring per-container resource allocations. Next, we describe the Linux block layer and existing IO control solutions. Finally, we provide the context of modern datacenters with heterogeneous block devices and workloads.

[1] https://github.com/facebookexperimental/resctl-demo

2.1 Resource Control with cgroups

Container runtimes rely on control groups (cgroups) for resource control and isolation. Today, cgroups is the primary mechanism for containers to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner.

There are two major conceptual parts. First, individual cgroups form a hierarchy and processes belong to one cgroup. A cgroup can contain a large collection of processes or just a single process. Second, cgroup controllers distribute specific system resources such as CPU, memory and IO along the tree according to the configuration.

A common way of configuring cgroup controllers are *weights*, in which a resource is distributed by adding up the weights of all sibling cgroups and giving each the ratio of its weight against the sum.

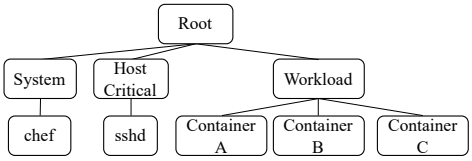

Figure 1: Meta's production cgroup hierarchy.

Figure 1 shows an example hierarchy used at Meta. The hierarchy is partitioned into system, host critical, and workload cgroups. *System* contains all auxiliary services, e.g., `chef`, that typically perform periodic actions to keep a host up to date. *Host Critical* contains the processes required to keep the host running, e.g., `sshd` and the container management agent. *Workload* holds all application processes, which is further partitioned into child cgroups in order to appropriately isolate different containers from one another.

2.2 Block Layer and IO Control

Applications and filesystems use the block layer to access block devices. Figure 2 shows the Linux block layer and other components that interact with it. Starting from the top, userspace interacts with the kernel through system calls. Read and write operations to filesystems propagate to the block layer as filesystem IO (FS IO). In addition, userspace can reach the block layer through memory operations that result in page faults, dirty page writebacks, or swap outs. The cgroup subsystem is responsible for resource accounting and passes control information along all the components based on the cgroup hierarchy.

The block layer uses the *bio* data structure to carry information such as the request type (e.g., read or write), the size, the target device, the sector offset of the device, the issuing cgroup, and the memory that data will be copied from or to. Before a request is submitted to the device driver, the block layer control and scheduling logic can choose to throttle the `bio`, merge it with other requests, etc. The Linux kernel has a number of different IO schedulers that can be enabled. We refer to those schedulers that integrate with the cgroup subsystem as "controllers" to distinguish from IO schedulers which simply ensure respectable machine-wide performance.

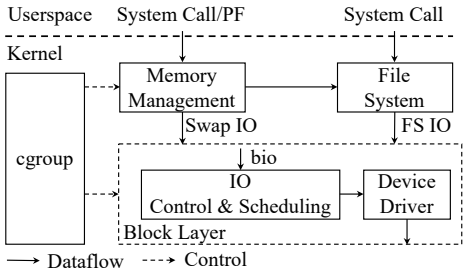

Figure 2: IO and the block layer.

Table 1 enumerates the features of various Linux IO control mechanisms. §4 provides a holistic comparison of these mechanisms through experiments. There are three main options for IO scheduling without cgroup control: *no scheduler*, *mq-deadline*, and *kyber*. They do not guarantee IO resources to containers, but instead ensure general performance properties, e.g., preventing asynchronous writes from dominating synchronous reads.

blk-throttle [29] allows setting IO limits in the form of read/write IOPS or bytes per second. These limits, however, are not work-conserving and difficult to configure for heterogeneous devices and diverse applications in datacenters.

BFQ [41] provides a work-conserving interface for proportional control of IO, but ignores interactions with memory management, which could lead to isolation failures. Moreover, as shown in §4.1, it has high per-request overheads and wide latency swings. Finally, BFQ does round-robin scheduling according to sectors read/written per container, which is ineffective on modern devices with complex internal operations.

In addition to IOCost, we also developed the *IOLatency* controller, which allows setting IO latency targets for individual cgroups. Specifically, it defines the maximum latency that a cgroup's IOs may take before other cgroups get throttled. For example, one cgroup with a 10 ms latency target would get throttled if the IOs of another cgroup with a 5 ms target start to take longer than 5 ms. We have upstreamed IOLatency to the Linux kernel.

We identified several limitations of IOLatency through our production deployment. First, the latency-based interface is suitable only for strict prioritization, i.e., to prevent low priority workloads from interfering with high priority workloads, but the lack of proportional control makes it unsuitable for ensuring fairness among equal-priority workloads. Second, although IOLatency is technically work-conserving, finding a configuration that is both isolating and work-conserving for diverse devices and workloads is nearly unmanageable.

2.3 Hardware and Workload Heterogeneity

Hardware Heterogeneity. Incremental hardware refresh and supply chain diversity lead to heterogeneous SSDs in datacenters. Figure 3 shows the device performance characteristics of various SSDs across Meta's fleet. The left y-axis shows IOPS for random and sequential reads and writes. The right y-axis shows latency for reads and writes. We use *fio* [2] to measure the sustainable peak performance for each device.

Table 1: Linux IO control mechanisms and features.

Mechanism	Low Overhead	Work Conserving	Memory Management-aware	Proportional Fairness	cgroup Control
kyber, mq-deadline	✓	✓	✗	✗	✗
blk-throttle	~	✗	✗	✗	✓
BFQ	✗	✓	✗	✓	✓
IOLatency	✓	~	✓	✗	✓
IOCost	✓	✓	✓	✓	✓

The eight types of SSDs (A-H) show distinctive characteristics. Specifically, SSD H achieves high IOPS at a low latency, SSD G offers low IOPS and a relatively low latency, and SSD A provides moderate IOPS with a higher latency. Each device usually represents less than 14% of the total fleet. An exception is device F with 19%. About 20% of the SSD capacity is spread over 18 devices not shown in the figure for readability but their characteristics are captured by the devices shown.

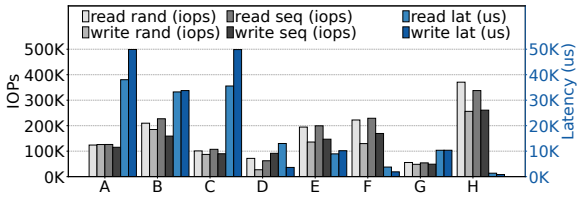

Figure 3: Device heterogeneity across Meta's fleet.

Workload Heterogeneity. Applications at Meta exhibit a large diversity in their IO workloads. Figure 4 displays the IO demand of several workloads at Meta. We measure the P50 over a week of production data, and show per-second reads vs. writes and random vs. sequential bytes. Workloads like Web A and Web B are most typical of Meta workloads, with a moderate amount of reads and writes mixed about equally in terms of random and sequential operations. Serverless workloads at Meta are highly overcommitted and exhibit a mixed amount of reads and writes. Cache A and Cache B are in-memory caching services that use fast block devices as a backing store for in-memory cache. Both exhibit high amounts of sequential IOs. Furthermore, non-storage services at Meta do relatively little explicit IO. Their IOs are often generated by paging and periodic software updates.

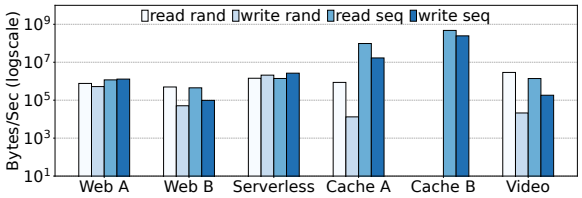

Figure 4: IO workload heterogeneity.

Overall, a major challenge of effective IO control is to be robust against heterogeneous hardware and diverse workloads, without requiring per-workload configuration (e.g., latency, IOPS, or bytes per second) that is often too brittle and intractable to be used in production at scale. An IO control mechanism needs to cater to the compound requirements of workloads while avoiding configuration explosion.

3 IOCOST DESIGN

IOCost's goal is to perform IO control that takes into account heterogeneous hardware devices and diverse workload requirements while providing proportional resources and strong isolation across containers.

3.1 Overview

IOCost explicitly decouples device and workload configurations. For each device, IOCost introduces a cost model and a set of quality-of-service (QoS) parameters that define and regulate device behaviors. For workloads, IOCost leverages cgroup weights for proportional configuration. This allows workload configuration to be made independently of device intricacies and improves the ease and robustness of large-scale configuration in heterogeneous environments.

IOCost adopts the concept of hierarchical weighted fair scheduling from multicore CPUs [30]. IOCost uses per-IO cost modeling to estimate the occupancy of an individual IO operation and then uses this occupancy estimate to make scheduling decisions according to the assigned weight for each cgroup. Our novel design separates out the low-latency issue path from a periodic planning path which allows IOCost to scale to SSDs that can reach millions of IOPS.

Figure 5 provides an overview of IOCost's architecture. IOCost is logically separated into the *Issue Path* that operates on a microsecond timescales for each `bio`, and the *Planning Path* that operates periodically at millisecond timescales. Additionally, offline work is done to derive device cost models and QoS parameters.

Let's briefly discuss the lifetime of a `bio` and its interaction with IOCost. First, IOCost receives a `bio` in step ① describing the IO operation. In the next steps IOCost calculates the *cost* of the `bio`, and then performs throttling decisions.

In step ②, IOCost extracts features from the `bio` and calculates the *absolute cost* using the cost model parameters. Cost is represented in units of time, but the cost of an IO is an occupancy metric, not latency. A cost of 20ms indicates that the device can process 50 such requests every second but does not say anything about how long each operation will take. We further describe feature selection and the cost model in §3.2.

Next, in step ③, the absolute IO cost is divided by the issuing cgroup's hierarchical weight (*hweight*) to derive the relative IO cost. *hweight* is calculated by compounding the cgroup's share of weight among its siblings while walking up the cgroup hierarchy. *hweight* represents the ultimate share of the IO device the cgroup is entitled to. For example, a cgroup with *hweight* of 0.2 has 20% share of the device, and the relative cost of an IO is $\frac{absolute\ cost}{hweight}$.

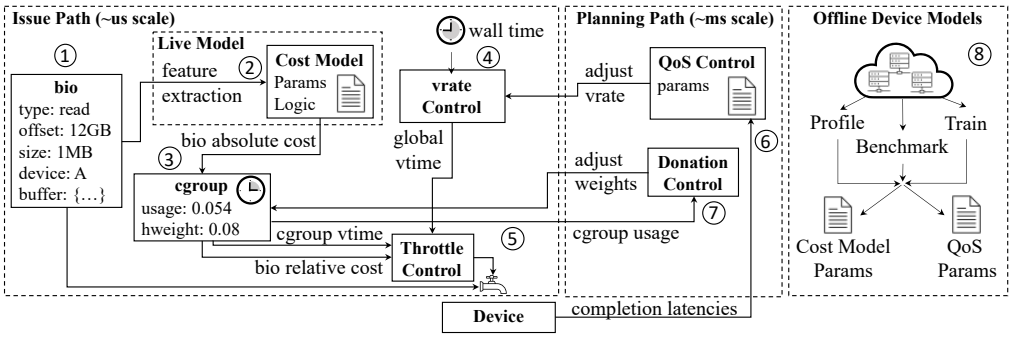

Figure 5: Overview of IOCost's architecture of how a bio's cost is assessed for throttling decisions on the left and offline cost model and logic generation on the right.

Step ④ shows the global *vtime* clock which progresses along with the wall clock at a rate specified by the virtual time rate (*vrate*). Each cgroup tracks its local *vtime* which advances on each IO by the IO's relative cost. Next, step ⑤ represents the throttling decision based on how far the local *vtime* is behind the global *vtime*. This gap represents a cgroup's current IO budget. If the budget is equal to or larger than an IO's relative cost, the IO is issued right away. Otherwise, the IO has to wait until the global *vtime* progresses far enough.

In the planning path, IOCost collects cgroup usage and completion latency, and makes periodic adjustments to IO control. In step ⑥, IOCost globally adjusts *vrate* and consequently the total IO issued in response to device feedback. Modeling may over- or under-estimate true device occupancy and this *vrate* adjustment ensures the device is well-utilized. We further discuss *vrate* adjustment and QoS in §3.3. Next, in step ⑦, IOCost's donation algorithm efficiently donates excess budget to other cgroups to achieve work conservation. §3.6 presents the algorithm in detail.

Offline in step ⑧, IOCost leverages profiling, benchmarking, and training across the deployed devices to build cost models and QoS parameters per device model that are later used during the production deployment.

3.1.1 Issue Path. The issue path determines the cost of an IO, the *hweight*, the available budget based on the local and global *vtimes*, and makes throttling decisions.

The absolute cost of a `bio` is calculated by applying the cost model to the features of the `bio`. Each cgroup is also assigned a weight, which represents the proportion of IO occupancy the cgroup is guaranteed among its siblings. To avoid repeating recursive operations on the hot path, the weights are compounded and flattened into *hweight* which is cached and recalculated only when the weights change.

A cgroup which does not issue IO and therefore does not consume its budget will leave the device underutilized. To address this, IOCost distinguishes *active* cgroups. A cgroup becomes active when it issues an IO and inactive after a full planning period passes without any IO. An inactive cgroup is ignored during *hweight* calculation. This low-overhead mechanism keeps device utilization high since idle cgroups implicitly donate their budget to the active cgroups. As a cgroup becomes active or inactive, it increments a weight tree generation number to indicate that weights have been adjusted. Subsequent cgroups executing through the issue path will notice this and recalculate their *hweight*.

3.1.2 Planning Path. The planning path is responsible for global orchestration so that each cgroup operates efficiently with only local knowledge and can converge on the desired hierarchically weighted fair IO distribution. It runs periodically based on a multiple of the latency targets in order to contain a sufficient number of IOs while allowing granular control.

The planning path tallies how much IO each cgroup is using to determine how much of their weight can be donated, and adjusts the weights accordingly. Through budget donations IOCost achieves work conservation while keeping the issue-path operations strictly local to the cgroup. The only donation-related issue-path operation is reducing or canceling donation if its budget runs low, which is also a local operation.

The planning path also monitors the device behavior and adjusts how much IO can be issued across all cgroups by adjusting *vrate* to control how fast or slow the global *vtime* runs compared to the wall clock. For example, if *vrate* is at 150%, the global *vtime* runs at 1.5x speed of the wall clock and generates 1.5x more IO budget than the device cost model specifies. The conditions and range of *vrate* adjustment are configured by a system administrator through the QoS parameters.

3.2 Device Cost Modeling

IOCost decouples device cost modeling from runtime IO control. Cost models are generated offline for each device before deployment. For maximum flexibility, IOCost allows a cost model to be expressed as an arbitrary eBPF program. In addition, IOCost natively supports a linear model, which works as follows. IOCost extracts the following features from a `bio` request: 1) read or write, 2) random or sequential relative to the cgroup's last IO, 3) request size. The IO cost is calculated as follows:

$$io\ cost = base\ cost + size\ cost\ rate * bio\ size \quad (1)$$

One of four *base costs* is selected by the combination of read/write and random/sequential. The *size cost rate* is selected by read or write. Thus, the linear model is composed of six parameters: the four *base costs* and two *size cost rates*.

For convenience, the configuration takes the six parameters in a different format - read/write byte per second (bps), and 4kB sequential and random IO per second (IOPS) for reads and writes -

and translates them internally to base_cost and size_rate_costs as:

$$size\_cost\_rate = \frac{1\ sec}{Bps} \quad (2)$$

$$base\_cost = \frac{1\ sec}{IOPS_{4kB}} - size\_cost\_rate * 4kB \quad (3)$$

```
rbps=488636629 rseqiops=8932 rrandiops=8518
wbps=427891549 wseqiops=28755 wrandiops=21940
```

Figure 6: Example IOCost configuration.

Figure 6 shows an example configuration. For reads, this translates to 2.05ns/B of size_rate, sequential base cost of 104us and random base cost of 109us. Accordingly, a random read `bio` of 32KB would cost $109us + 32 * 4096 * 2.05ns = 352us$ and the device would be able to service 2840 of them every second.

Our tools use `fio` and saturating workloads to infer the linear model's parameters for a device, e.g., issuing as many 4KB random reads as possible to determine the *base cost* for random reads. Systematically modeling devices in this way is practical even with the roughly thirty different storage devices found in Meta datacenters. We have made our modeling tools available in the Linux source tree.

3.3 QoS and Dynamic Vrate Adjustment

Simple linear modeling cannot capture the subtleties of modern SSDs. Devices have complicated layers of caching, request reordering, garbage collection, and can perform in unexpected ways with different IO mixtures. Prior research has focused on the difficulty of accurately modeling SSD behavior [26, 48]. IOCost accommodates device performance variation by dynamically adjusting `vrate`.

`vrate` adjustment acts on two signals: IO budget deficiency and device saturation. The former indicates that the kernel could issue more IO but the global budget as determined by *vtime* prevents it. The latter indicates that the device cannot handle more IO. If the system could issue more IO and the device is not saturated, *vrate* is adjusted upwards. If the device is saturated, *vrate* is adjusted downwards.

IOCost tracks request depletion and latency target violations to identify device saturation. Request depletion occurs when there are too many in-flight IOs, depleting the available IO slots and causing a long queue at the device layer. Latency targets are set via QoS parameters. For example, a system administrator can configure the device to be considered saturated if the 90th percentile read completion latency is above 10 ms.

By constraining how much total IO is issued to the device, IOCost can achieve consistent latency even on devices that show bursty behaviors, or other behaviors poorly captured by the device model. IOCost considers latency as a device-level property. It uses QoS parameters to regulate the device behavior and then distributes the resulting IO occupancy. This separation simplifies workload configuration and is necessary to uphold the QoS targets. Hypothetically, if we loosen device throttling for a batch workload, we might lose control over the device and fail to meet the QoS targets when a latency-sensitive workload becomes active.

3.4 Tuning QoS Parameters with ResourceControlBench

The QoS parameters determine the overall throttling of the device, which is an important trade-off of device utilization for consistent latency. Ultimately, how to make this tradeoff is dependent on the storage use-cases. At Meta the predominant consideration is ensuring reasonable IO latencies under contention with raw throughput as a secondary consideration.

In order to ensure that devices are throttled sufficiently, we developed a systematic approach to determining QoS parameters for each device in the Meta fleet. While a full description is outside the scope of this paper, we present a simplified description.

We developed *ResourceControlBench*, a highly configurable synthetic workload imitating the behavior of latency-sensitive services at Meta. We leverage ResourceControlBench for QoS tuning by observing its behavior across *vrate* ranges. We execute ResourceControlBench in two scenarios.

First, ResourceControlBench runs alone on a machine and adjusts its working set size until the throughput available for paging and swap operations begins to limit ResourceControlBench's performance. As *vrate* is lowered, the working set size drops. Second, ResourceControlBench runs alongside a memory-leak in another container. As *vrate* is lowered, IO control improves until ResourceControlBench's latency is sufficiently protected from thrashing caused by the memory leak.

These two scenarios identify two points on the *vrate* range below which no further IO control improvements are needed and above which throughput gains result in no meaningful advantages for memory overcommit. We bound *vrate* between these two points for each device. These QoS parameters are deployed for each device in the fleet, resulting in consistent latency control with minimal throughput loss for all applications. ResourceControlBench and the scenario generating tools are made available as open source software.

3.5 Handling Priority Inversion

Consider two cgroups A and B sharing a machine with equal weights. A is leaking memory continuously. Eventually when the machine is low on memory, B attempts to allocate memory and enters memory reclaim which identifies some of A's memory for swapping out. The swap out `bio` can only reasonably be charged to A, the owner of the memory being swapped out. If charged to B, then B gets penalized for A's excessive memory usage, violating resource isolation.

This swap out must complete synchronously for B to complete its memory allocation. If A is over-budget, throttling would create a priority-inversion where again B is penalized for A's excessive memory usage. Instead, IOCost simply allows A to go into debt and issues the IO without throttling. A's future `bios` are throttled proportionally until the debt is paid off with its future budget.

However, if A leaks memory and issues no IO that can be throttled, A gets an unfair amount of "free" swap-out IO, and will never pay back its debt. To address this, IOCost adds a check before each return to userspace. If the accumulated debt exceeds a threshold, the thread is blocked momentarily before returning to userspace to throttle the generation of "free" IO. As a result, memory activities

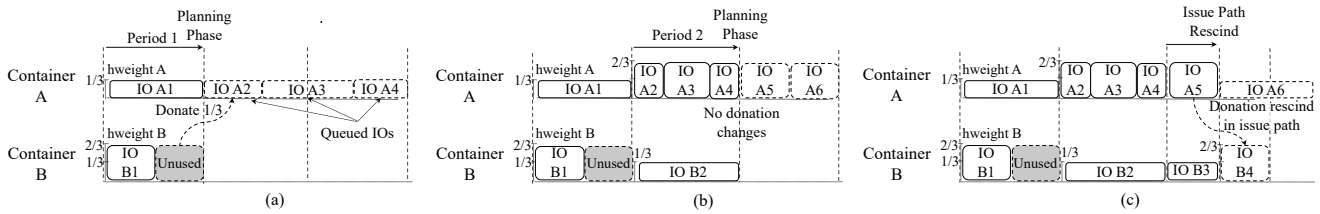

Figure 7: Budget donation example at the planning phase (a), after the planning phase (b), and during issue path (c).

generating swap outs are throttled without causing priority inversions. The same mechanism is used for shared filesystem operations like journaling.

3.6 Budget Donation

Individual cgroups do not always issue IOs that saturate their *hweight*. IOCost ensures work conservation by allowing other cgroups to utilize the device by dynamically lowering the weights of the donor cgroups. We explored multiple options including temporarily accelerating `vrate`, but found that local adjustment of weight was the only solution that met all the following requirements: 1) the issue path remains low overhead, 2) the total amount of IO issued never exceeds what `vrate` dictates, and 3) donors can cheaply rescind anytime.

Each planning phase identifies the donors and calculates how much of their *hweight* can be given away. It then calculates their lowered weights that compound to the after-donation *hweights*. The weight calculation process is structured in a way that parent weight adjustments are derived solely from child weight adjustments.

As donation happens through weight adjustments, the IO issue path does not change and there is no interaction with device-level behaviors, satisfying 1) and 2). A donor can rescind by updating its weight and propagating the update upwards in the issue path without any global operation, satisfying the final requirement. This increments the weight tree generation number so subsequent issuers will recalculate their *hweight*.

High-level Donation Example. In Figure 7(a), the *hweights* of containers A and B are $\frac{1}{3}$ and $\frac{2}{3}$, respectively. During the planning phase, it detects that B has not used half of its budget. To avoid leaving the device underutilized, it transfers half of B's original budget to A. Figure 7(b) shows how this affects the second period. With *hweight* increased, A's IOs have lower relative costs and can be issued more frequently, while B saturates its new lowered budget. At the end of the period, there is no need for further adjustments. Figure 7(c) shows that in the middle of the third period, B attempts to issue additional IOs and rescinds its donation in the issue path, without waiting for the next planning phase. Note that a container could also rescind only a portion of its original donation.

Weight Tree Update Algorithm. Let w be the weight, s be the summed weight of siblings, h be the *hweight*, and d be total *hweight* of all donating leaves for the subtree. Let subscript p denote a parent node, and an apostrophe denote the value after donation.

Figure 8 shows an overview of what occurs during budget donation. Here, leaf nodes B and H have active usage that is in total 0.25 less than their configured *hweights*. This excess is donated to other cgroups that can potentially use more IO proportionally to their *hweights*. Most importantly, only local updates are needed as w values are only decreased along paths from B and H to the root, and then all other nodes can correctly calculate their new *hweight* lazily in the issue path.

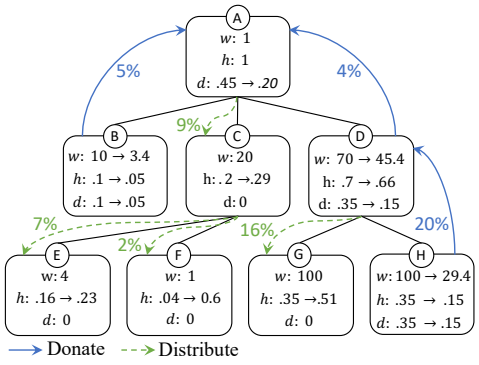

Figure 8: B and H donate portions of their budget.

The values of d' are propagated up the tree and provided as inputs to the budget donation algorithm. Only w', h' need to be calculated down the tree along paths to B, D, and H. We describe two invariants which we maintain to ensure non-donating nodes do not need to be updated which lead to derivations of h', s', w'.

The first invariant enforces that the proportion of a parent's non-donating weight does not change with budget donation:

$$\frac{h - d}{h_p - d_p} = \frac{h' - d'}{h'_p - d'_p} \tag{4}$$

The second invariant enforces that the total summed weight w of siblings that are not donating does not change with budget donation:

$$s * \left(\frac{h_p - d_p}{h_p}\right) = s' * \left(\frac{h'_p - d'_p}{h'_p}\right) \tag{5}$$

The following steps determine new w' values for paths from the root to donating children. In our example, this is performed for nodes B and D first, then the results for D are used to perform the same at node H:

(1) Using the invariant of Equation (4) a new *hweight* is calculated from the parent's h' value: $h' := \left(\frac{h-d}{h_p-d_p}\right) * (h'_p - d'_p) + d'$
(2) The new sibling weight is calculated based on the invariant of Equation (5): $s' := s * \left(\frac{h_p - d_p}{h_p}\right) * \left(\frac{h'_p}{h'_p - d'_p}\right)$
(3) The final weight then is derived from the calculated h' and s': $w' := s' * \left(\frac{h'}{h'_p}\right)$

The remaining w', s', h' values are shown for completeness on other nodes, but they are not needed as part of budget donation. Note that w' does not change for other nodes. This efficiency is important for large cgroup hierarchies. Just by updating w' along paths from donating leaves to the root, all other node's new h' based on these w' updates will receive the correct value. In the example, 0.25 *hweight* is freed up by B and H, which is split among E, F, and G according to the ratio of their original *hweight*'s 0.16:0.04:0.35, resulting in a donation of 0.07, 0.02, and 0.16 to E, F, and G, respectively.

4 EVALUATION

This section demonstrates that IOCost provides IO control that is low overhead, work-conserving, memory-management aware, and allows for proportional cgroup configuration. We compare IOCost against the state-of-the-art Linux IO control mechanisms and our previous solution IOLatency as described in §2.2. We demonstrate that none of these mechanisms match IOCost's set of features and performance.

In all experiments, except where otherwise specified, we use a single-socket, 64 GB server with one of three different SSDs: 1) An older generation commercial SSD, 2) A newer generation commercial SSD, 3) A high-end enterprise-grade SSD. We install the 5.6 Linux kernel, patched with the latest IOCost changes from 5.15. Model parameters are determined using `fio` saturating workloads as described in section 3.2. QoS parameters are determined using ResourceControlBench as described in section 3.4.

4.1 Low Overhead

Controlling IO for fast SSDs in datacenters requires the controller to impose a minimal overhead. This experiment uses an SSD with maximum read IOPS of 750K. We use `fio` to generate as many 4KB random reads as the IO subsystem can support.

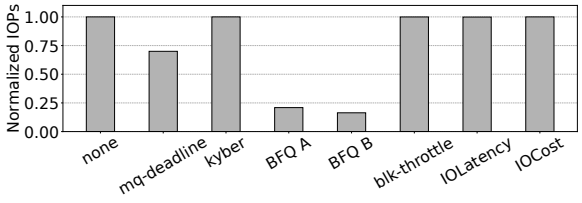

Figure 9: IO Control Overhead.

Figure 9 measures the maximum achievable IOPS with IO control enabled using a number of different mechanisms. The controllers or schedulers are not configured to perform any actual throttling so that we can measure the induced overhead on the fast-path of issuing IO. We use the enterprise-grade SSD for this experiment to demonstrate overheads on one of our fastest storage devices. We disable QoS settings for all controllers in order to simply measure their baseline overheads when not throttling the device.

The `none` column is with no software scheduler or controller running and therefore simply shows the achievable throughput of the block layer on this device. `mq-deadline` is the default Linux scheduler and has moderate overhead. `kyber`'s performance is indistinguishable from no scheduler. Neither of these IO schedulers provides cgroup control, as they merely provide system-wide scheduling. `bfq` has severe software overheads. Despite significant tuning we were unable to find a configuration with reasonable performance. The remaining columns illustrate that other IO controllers add no significant overhead. Despite having a much more complex throttling logic than others, IOCost is able to ensure no noticeable overhead due to its split between the issue path and the planning path.

4.2 Proportional Control and Work Conservation

Work-conserving IO control is important for ensuring that storage device performance is well-utilized in the event that some consumers are idle. Without work-conserving IO control we would need to over-provision IO for infrequent activities such as OS software updates.

To evaluate these properties, we perform two related experiments where two synthetic workloads run concurrently. In the first experiment, we run two instances of a latency-sensitive workload continuously issuing random 4KB reads so long as the observed p50 latency is below 200 us. These workloads simulate online services which may load-shed if request latencies climb too high. We configure the high-priority workload to be granted double the IO of the low-priority workload. We perform this experiment on our older-generation SSD, which due to its relatively lower latency, has higher demands in terms of IO control.

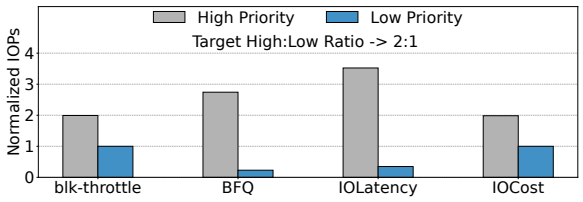

Figure 10: Proportional control. The target ratio of IOPS received by the high-priority and low-priority workloads is 2.

Figure 10 shows the results of this first experiment. We focus only on the cgroup-aware IO control mechanisms. `bfq` is configured with weights for the desired 2:1 distribution. However, the high-priority workload dominates at a ratio of more than 10:1. This is because the low-priority workload suffers from poor latency and continuously lowers its IO issue rate to remain below the 200 us target, which in turn allows the high-priority workload to dominate and receive much more than its fair share. `blk-throttle` is configured to limit each workload to preserve the 2:1 ratio. It matches IOCost's observed latencies and behaves as expected. `IOLatency` does not provide any way to configure such a distribution. Instead we tuned per-cgroup latency targets in an attempt to achieve the desired distribution, but the best configuration (shown in the figure) still results in a roughly 10:1 distribution. Finally, IOCost is configured with weights just as with `bfq` and able to precisely match the expected 2:1 ratio.

The second experiment preserves the same configuration, but replaces the high-priority workload with one that sequentially issues random 4KB reads with 100 us think time, i.e., a new IO is

issued after 100 us has passed since the last IO's completion. The throughput achieved is a function of the latency of the reads and substantially lower than in the previous experiment. Accordingly, the throughput available to the low-priority workload is dependent on the work-conserving properties of the IO controller. We expect to see the low-priority workload saturate the remaining available IO.

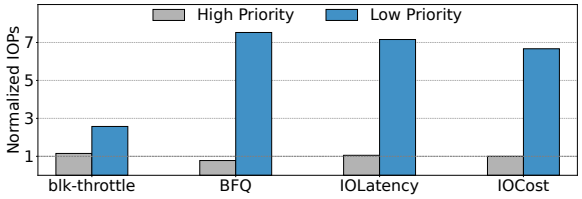

Figure 11: Work conserving. The low-priority workload should use up all available capacity.

Figure 11 shows the result of this second experiment. `bfq`'s work-conserving properties lead to the low-priority workload completing a substantial amount of IO. `bfq`'s ability to outperform the other mechanisms in this respect are due to its weak latency control, which leads to the high-priority workload performing noticeably worse. The high-priority workload sees an average of 250 us latency with a standard deviation of nearly 1 ms, whereas all others hold latency below 200 us on average with a standard deviation around 200 us. This experiment also demonstrates the major downsides of non-work-conserving approaches such as `blk-throttle`, which controls latency well, but does not allow the low-priority workload to consume more IO than in the previous experiment. IOLatency and IOCost perform comparably, controlling latency for the high-priority workload and allowing the low-priority workload to consume otherwise available IO. Together, these two experiments demonstrate that IOCost uniquely provides proportional and work-conserving IO control.

4.3 Spinning Disk Modeling

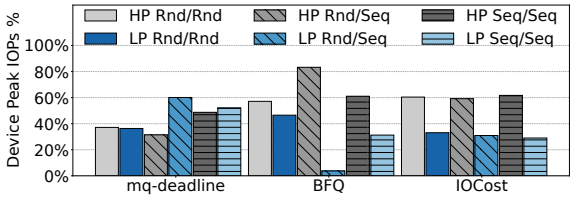

Figure 12: Fairness with random and sequential workloads on a spinning disk

While SSDs make up the vast majority of the Meta fleet, IOCost can be used with spinning disks. In contrast to SSDs, spinning disks have high seek latencies which means random IOs have a lower throughput (or higher occupancy cost) than sequential IOs. We run an experiment where two workloads issue either random 4KB reads or sequential 4KB reads. One workload (high weight) is configured with double the weight of the other workload (low weight). We compare mq-deadline, bfq, and IOCost under three scenarios: both workloads issuing random reads (rand/rand), the high priority issue random reads while the low priority issues sequential reads (rand/seq), and where both are issuing sequential reads (seq/seq).

Figure 12 shows the results of this experiment. To visualize the differences clearly, we normalize the throughput on the random and sequential workloads to the peak throughput the device can handle of each type of workload, respectively. The results show that mq-deadline cannot provide fairness at the 2:1 ratio across any of the workloads as it is simply a global scheduler. BFQ performs well when both workloads issue sequential IO, maintaining the desired 2:1 ratio but struggles when both workloads issue random IO and substantially over-allocates device occupancy to the random read workload when mixed with a sequential workload. In comparison, IOCost maintains the desired 2:1 ratio in all scenarios by modeling the cost of random vs sequential IO and ensuring fairness in terms of device occupancy. This leads to proper isolation where workloads receive the same service from the disk regardless of the nature of their neighbor's disk accesses.

4.4 QoS and Vrate Adjustment

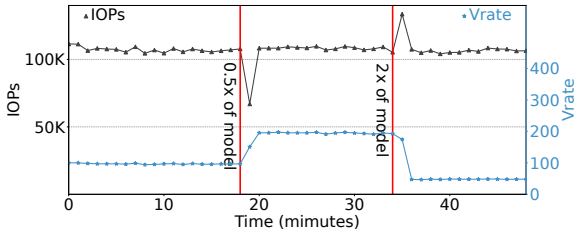

Figure 13: Vrate adjustment due to model inaccuracy

As discussed in section 3.3, modern SSDs have complexities which make simple modeling approaches inaccurate and could lead to IOCost under or over saturating a device. *vrate* compensates for modeling inaccuracy by dynamically adjusting the overall issue rate.

In Figure 13, we show the results of an experiment on our newer-generation commercial SSD where a workload is attempting to saturate the device with 4KB random reads and QoS settings are configured so that IOCost will maintain a p90 read latency of 250 microseconds. Initially, the vrate remains around 100, indicating the model parameters are appropriate for maintaining such QoS. At the first indicated time, we issue an online update of the model parameters which cuts their values in half (effectively claiming that the device has half the amount of occupancy as before). In response, the read rate drops. However, vrate quickly climbs to roughly double the issue rate while maintaining QoS as we expect. Finally, at the second indicated time, we issue an online update of the model parameters to set them to double the original value (effectively claiming that the device has twice the amount of occupancy as before). Initially, issue rate over-saturates the device, leading to a spike in latency which subsides as vrate drops to roughly half the initial value in order to maintain QoS. This experiment demonstrates that the dynamic vrate adjustment functionality of IOCost can handle modeling inaccuracies and still preserve QoS.

4.5 Memory-Management Awareness

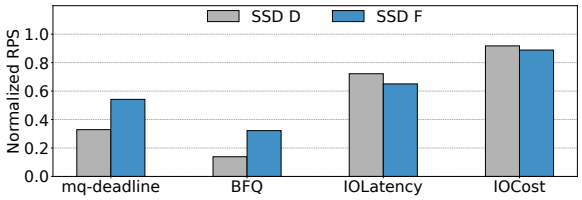

Figure 14: Requests per second (RPS) of a latency sensitive workload when stacked with a memory-leak workload.

Overcommitting resources is a common approach to increasing utilization in datacenters. One method is to deploy a high-priority workload with guaranteed resources and allow a low-priority workload to consume the remaining resources on the machine in a best-effort fashion. Memory management integration is crucial to ensure that resources are appropriately reclaimed.

We show the results from production web servers at Meta on both our older and newer generation commercial SSDs. We launch a memory leaking process in the system slice (see Figure 1 for the cgroup hierarchy), which is eventually killed by the out-of-memory (OOM) killer. Figure 14 shows that the web-server throughput is reduced due to thrashing. With ideal resource control, the web server should mostly maintain its throughput. `mq-deadline` isolates poorly as it lacks any cgroup integration, but fares slightly better with the higher end SSD simply due to having more bandwidth. Despite `BFQ`'s proportional control, it performs worst, resulting in an almost total loss of throughput due to lack of latency control and memory management integration. IOLatency performs moderately well. Finally, IOCost outperforms all the other IO control mechanisms, and the web server drops to no lower than 80% of its normal throughput.

In order to evaluate the particulars of the memory management integration, we created an experiment where ResourceControlBench is collocated with `stress`, a synthetic memory consumer which constantly touches its configured working set. We configure a PID controller to slowly add load to ResourceControlBench from 40% of its peak compute load to 80% while keeping p95 latency under 75 ms. As ResourceControlBench's load increases, its memory access frequency increases, pushing up its demand for resident memory. In turn, the synthetic memory consumer's memory must be paged out to ensure sufficient memory for the high-priority ResourceControlBench. We measure the time it takes for ResourceControlBench to scale from 40% of its peak load to 80%.

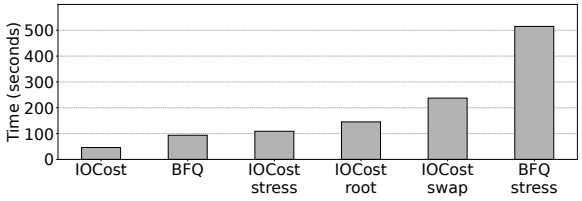

Figure 15: Ramp-up time in an overcommitted environment.

Figure 15 displays the results of this experiment. The two baseline configurations without `stress` show that IOCost ramps up in about half the time as `BFQ`. With `stress` consuming memory, the IOCost configuration is able to complete scaling up about 5x faster than `BFQ`. We additionally run modified versions of IOCost. In the first configuration, all swap out IO is charged to the root cgroup and thus never throttled. `stress` runs freely regardless of how much swap IO is consumed. In the second configuration, we instead throttle swap IO according to the originating cgroup, creating a priority inversion where ResourceControlBench may be throttled when swapping out `stress`'s memory. These two configurations perform worse than the production version of IOCost and show how IOCost's debt mechanism (§3.5) avoids a priority inversion while maintaining good IO control.

4.6 Stacked Latency-Sensitive Workloads

One production use of IOCost ensures that multiple containers receive their fair proportion of IO service. At Meta, we run a workload similar to Zookeeper [23] which provides a strongly-consistent API for configuration, metadata and coordination primitives like watches, locks and semaphores. A single operation is replicated to several participants in an ensemble providing fault tolerance. The service triggers a snapshot of the in-memory database after 500000 txns which results in momentary write spikes even under nominal loads. The production service has a one second SLO for read and write operations. This SLO makes the service a difficult candidate to collocate with other services as slow down experienced by a participant of the ensemble may result in an entire operation slowing down. This service runs on machines with our enterprise-grade SSDs

We analyzed the behavior of this service in a scenario where twelve ensembles (of five participants each) were distributed over five machines. No two participants of the same ensemble share a host. This configuration allows for multiple low-traffic ensembles to share machines with reasonable aggregate utilization.

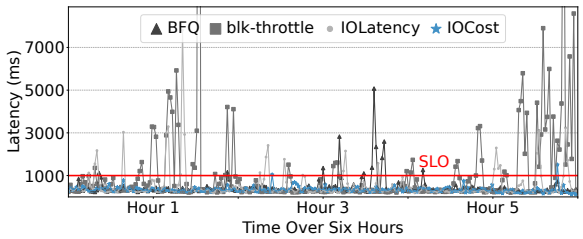

Figure 16: Impact of different IO control methods on ZooKeeper latency SLO violations.

The twelve ensembles receive a moderate amount of traffic, 3000 reads per second and 100 writes per second. Eleven of the twelve ensembles average 100 KB payloads while the twelfth ensemble behaves as a noisy neighbor with 300 KB payloads. Figure 16 shows the P99 latency of the eleven well-behaved ensembles. SLO violations are characterized by their frequency and magnitude. With `blk-throttle`, `BFQ` and `IOLatency`, the ensembles repeatedly violate their one-second SLO throughout the six-hour experiment.

Specifically, `blk-throttle` shows 78 violations with some lasting tens of seconds. `BFQ` shows 13 violations each lasting 2-5 seconds. Notably, while not shown in the figure, we had to run this experiment several times due to the severity of `BFQ` throttling leading to total system unresponsiveness. `IOLatency` cannot be configured for proportional control and also shows poor behavior of 31 violations with the longest being 7.8 seconds. With `IOCost`, the effects of the noisy neighbor ensemble and snapshots were appropriately isolated, resulting in only two marginal violations of 1.5 seconds and 1.04 seconds.

4.7 Remote Storage and VM Environments

Beyond local storage, IOCost is also useful for providing IO control for remote block storage environments like those found in public clouds. To evaluate the broad applicability of IOCost, we repeat the experiment from Figure 14, replacing the production web server at Meta with ResourceControlBench that is collocated with a high-speed memory-leak program running in a low priority cgroup. As before, we report the drop in ResourceControlBench's RPS as a measurement of how well IOCost protects the workload from interference.

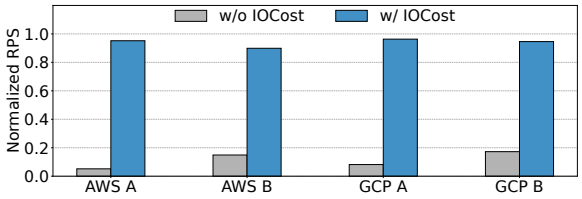

Figure 17: Requests per second (RPS) of a latency-sensitive workload when stacked with a memory-leak workload in AWS EBS, and Google Cloud Persistent Storage.

We run the two workloads in a public cloud's VM whose guest OS is configured with IOCost. Figure 17 shows the resulting protection ratios of the four configurations—two AWS Elastic Block Store (gp3-3000iops, io2-64000iops), and two Google Cloud Persistent Disk configurations (balanced, SSD). While there are variances from the different latency profiles, the experiment clearly shows that IOCost can effectively isolate IO for all configurations whether local or remotely attached. This experiment demonstrates that IOCost's approach to modeling and QoS parameterization is robust and can be successfully applied to environments outside Meta.

4.8 Package Fetching and Container Cleanup

A major feature of IOCost in comparison to IOLatency is that the proportional control capability allows us to ensure fair share of IO for system services and workloads instead of enforcing strict priority ordering. Furthermore, IOCost successfully protects services and workloads even under extreme circumstances, when the resources on a server are fully utilized and heavily contended.

Package fetching failures. A common operation at Meta is package fetching for containers. This works through a hostcritical service (the container agent) asking a system service to fetch packages. We often see instances where the communication between the two services fails due to the system service getting starved for IO. Package fetching failures lead to container update failures which often result in the machine being taken out of production entirely.

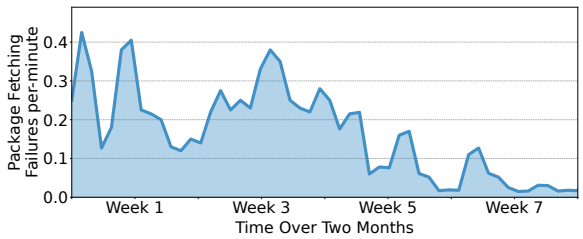

Figure 18: Package fetching failures reduction as a region migrates from our previous solution IOLatency, to IOCost.

Figure 18 shows the effect of IOCost, as a region of hundreds of thousands of servers migrates from IOLatency to IOCost over a period of two months. As IOCost is enabled the rate of package fetching errors drops in the region, resulting in roughly 10x fewer errors.

Container cleanup failures. A regular operation at Meta's datacenters is cleanup of old containers. We rely on btrfs and its copy-on-write semantics so this is a usually cheap operation but we still see cases where it can take several seconds. These cases are often caused by the main workload starving the container agent of I/O. Cleaning up old containers is often required to ensure sufficient disk space for subsequent containers and failures to clean can result in a machine being functionally unusable.

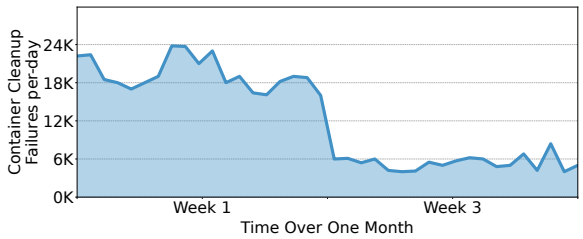

Figure 19: Container cleanup failures reduction as a region migrates from our previous solution IOLatency, to IOCost.

Figure 19 shows the failure reduction of container cleanups, those that take longer than 5 seconds, as the region migrates to IOCost. The effect of IOCost is immediate. Specifically, we see that IOCost achieves a 3x reduction, a major reduction of stalls. This again shows the impact IOCost has on the ability for the container orchestration system to successfully manage hosts.

5 LESSONS LEARNED

Meta has one the world's largest deployment of IO control. One initial motivation was to address isolation failures from system service memory leaks. Memory control alone was insufficient as memory limits still resulted in reclaim which interfered with latency-sensitive applications through IO. We could achieve comprehensive isolation only by doing both memory and IO controls together.

We experimented with the existing IO control mechanisms and found them ineffective for heterogeneous devices and applications

at Meta. Assigning IO limits for each application via `blk-throttle` was inefficient, brittle and ultimately intractable. `BFQ` showed substantial overheads and wide latency swings while failing to isolate in real-world scenarios.

We first developed IOLatency and it revealed isolation failures from priority inversions in memory management and filesystem operations. After addressing the priority inversions, we could achieve comprehensive isolation with tuned latency targets. However, production configuration was difficult because the latency target is a complex function of both the heterogeneous-device properties and dynamic-application properties. A configuration tuned for one scenario was often ineffective for others. Moreover, it could not arbitrate IO across multiple competing applications of equal priorities.

We then developed IOCost to address the limitations of IOLatency. IOCost is easier to configure as the device configuration can be approached systematically by first modeling device performance using `fio` (§3.2) and then tuning QoS parameters using *ResourceControlBench* (§3.3). With an IO cost model per device, effective IO control can be achieved for diverse applications with simple proportional weights without requiring per-application offline profiling or per-application configuration of IOPS, bytes, or latency that are often too brittle and intractable to be used in production at scale. Overall, IOCost has been running robustly in production for two years, handling our fleet's heterogeneous devices and diverse applications.

Preference towards SSDs with consistent performance. At Meta's datacenters, we repeatedly experienced unpredictable SSD behaviors and found it unrealistic to cater to the behaviors of specific devices. As diverse applications migrate across the heterogeneous fleet, it was impractical for us to tune each application to the oddities of the specific SSD that it happened to encounter. We abandoned our first-generation solution IOLatency primarily because it required fragile per-application tuning. Our current solution leverages IOCost's QoS feature to throttle SSDs to achieve acceptable latency and consistency for diverse applications.

Overall, our experience indicates that SSDs with more consistent behaviors, rather than ones with high but temporary and unpredictable peak performance, could be effectively utilized in highly scaled and complex environments. As such, we recommend that SSDs striving for steady throughput and latency are better suited for datacenters.

6 RELATED WORK

Consistent with our findings, [18] provided a large scale study of production storage devices and found a significant amount of performance variability across devices. In addition, FLIN [40] found that workload IO request patterns played a significant role in unfairness among concurrently-executing applications.

ReFlex [27] adopts a modeling approach to account for read/write interference in accessing remote flash devices. SSDcheck [26] constructs a performance model for modern SSDs in order to predict per-request latency and in-turn schedule based on expected request latencies. Similarly, [48] discusses the needs and challenges of modeling SSD performance and advocates for reverse-engineering of devices as opposed to black-box modeling.

The literature on virtual machine monitors has worked to address IO fairness across several different works. PARDA [16] and mClock [17] both explore designs for providing coarse-grained fairness for VMs accessing network-attached storage. Both VMWare [42] and NetApp [32] present IO solutions which allow for VMs to get a configured amount of IOPS. IOCost deals with the additional challenges of how the I/O subsystem interacts with the memory subsystem and uniquely approaches IO fairness by modeling device occupancy, rather than measuring and controlling in terms of IOPS or latency. We believe that modeling device occupancy could be a fruitful approach for virtual machine monitors to explore.

Cello [35], Argon [43], and Redline [47] all present approaches for controlling IO in the era of slow, rotational drives with relatively low concurrency and high seek latencies. More recently, WDT [1] describes a cgroup-aware IO scheduler configured via weights targeting high-speed SSDs which, in contrast to IOCost, distributes IO bandwidth not occupancy. FlashBlox [21] partitions SSD channels which allows for hardware-enforced isolation albeit at the cost of flexibility in the number of tenants.

In [46], the authors identify the need to account for information across different layers of the IO stack for scheduling. IOCost identifies swap and journaling IO sources and introduces IO control on memory management and filesystem journaling operations without priority inversions.

Several works have focused on resource management [8, 10–12, 15, 24, 25, 31, 33]. These solutions aim to partition system resources among co-located applications without violating individual SLOs. Other works [37, 38] have proposed architectural and OS extensions for containers. Overall, they operate at a layer above or below IOCost and can leverage its robust IO control to further push the co-location capabilities in datacenter environments.

7 CONCLUSION

We have identified the need for IO control in containerized environments. We presented IOCost, an IO control solution that is designed for containerized environments and provides scalable, work-conserving, and low-overhead IO control for heterogeneous storage devices and diverse workloads in datacenters. Our approach estimates device occupancy through an offline generated device cost model. Furthermore, IOCost's design separates IO control into a lightweight per-IO issue path and a periodic IO planning path. A novel cgroup tree hierarchy weight update algorithm ensures that containers dynamically share unused IO budget with minimal overhead. Finally, we shared our experience with IOCost and potential future hardware directions.

ACKNOWLEDGMENTS

Dimitrios Skarlatos at CMU is funded by NSF under grant CNS 2107307 and a Meta Faculty Award.

REFERENCES

[1] Sungyong Ahn, Kwanghyun La, and Jihong Kim. 2016. 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)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ahn

[2] Jens Axboe. 2021. Flexible I/O Tester. https://github.com/axboe/fio.

[3] Microsoft Azure. 2021. Container Instances. https://azure.microsoft.com/en-us/services/container-instances.

[4] J.C.R. Bennett and Hui Zhang. 1996. WF/sup 2/Q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*, Vol. 1. 120–128 vol.1. https://doi.org/10.1109/INFCOM.1996.497885

[5] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. 2012. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 94–101. https://doi.org/10.1109/ICCD.2012.6378623

[6] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, USA) *(SIGMETRICS '09)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/1555349.1555371

[7] Renhai Chen, Yi Wang, Duo Liu, Zili Shao, and Song Jiang. 2017. Heating Dispersal for Self-Healing NAND Flash Memory. *IEEE Trans. Comput.* 66, 2 (2017), 361–367. https://doi.org/10.1109/TC.2016.2595572

[8] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. New York, NY, USA, 107–120.

[9] Google Cloud. 2021. Containers at Google. https://cloud.google.com/containers.

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2451116.2451125

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[13] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.* 19, 4 (Aug. 1989), 1–12. https://doi.org/10.1145/75247.75248

[14] Peter Desnoyers. 2014. Analytic Models of SSD Write Performance. *ACM Trans. Storage* 10, 2, Article 8 (March 2014), 25 pages. https://doi.org/10.1145/2577384

[15] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, USA, 25–38.

[16] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. 2009. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*. USENIX Association, San Francisco, CA.

[17] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2010. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC.

[18] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. 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 16)*. USENIX Association, Santa Clara, CA, 263–276. https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao

[19] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/3064176.3064187

[20] Tejun Heo. 2015. Control Group V2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[21] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 375–390.

[22] Lan Huang, Gang Peng, and Tzi-cker Chiueh. 2004. Multi-Dimensional Storage Virtualization. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (June 2004), 14–24. https://doi.org/10.1145/1012888.1005692

[23] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[24] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 598–610. https://doi.org/10.1145/2830772.2830797

[25] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 729–742. https://doi.org/10.1145/2541940.2541944

[26] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. 2018. SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 455–468. https://doi.org/10.1109/MICRO.2018.00044

[27] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA.

[28] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. 2013. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 2 (2013), 247–260. https://doi.org/10.1109/TCAD.2012.2227479

[29] Shaohua Li. 2016. block-throttle: proportional throttle. https://lwn.net/Articles/676823/.

[30] Tong Li, Dan Baumberger, and Scott Hahn. 2009. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. 65–74.

[31] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. (2016), 33 pages.

[32] NetApp. 2021. Guarantee throughput with QoS overview. https://docs.netapp.com/us-en/ontap/performance-admin/guarantee-throughput-qos-task.html#about-throughput-ceilings-qos-max.

[33] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. 2021. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3477132.3483578

[34] Amazon Web Services. 2021. Containers on AWS. https://aws.amazon.com/containers.

[35] Prashant J. Shenoy and Harrick M. Vin. 1998. Cello: A Disk Scheduling Framework for next Generation Operating Systems. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Madison, Wisconsin, USA) *(SIGMETRICS '98/PERFORMANCE '98)*. Association for Computing Machinery, New York, NY, USA, 44–55. https://doi.org/10.1145/277851.277871

[36] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. 2008. Server-storage virtualization: Integration and load balancing in data centers. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5222625

[37] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and Operating System Support for System Call Security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 42–57. https://doi.org/10.1109/MICRO50266.2020.00017

[38] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. 2021. BabelFish: Fusing Address Translations for Containers. *IEEE Micro* 41, 3 (2021), 57–62. https://doi.org/10.1109/MM.2021.3073194

[39] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 787–803. https://www.usenix.org/conference/osdi20/presentation/tang

[40] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in

Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 397–410. https://doi.org/10.1109/ISCA.2018.00041

[41] Paolo Valente and Fabio Checconi. 2010. High throughput disk scheduling with fair bandwidth distribution. *IEEE Trans. Comput.* 59, 9 (2010), 1172–1186.

[42] VMWare. 2021. Storage I/O Control Resource Shares and Limits. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-D964A753-0844-4343-A96F-27A4C769F92D.html.

[43] Matthew Wachs and Michael Abd-El-Malek. 2007. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-07/argon-performance-insulation-shared-storage-servers

[44] Qi Wu, Guiqiang Dong, and Tong Zhang. 2011. Exploiting Heat-Accelerated Flash Memory Wear-Out Recovery to Enable Self-Healing SSDs. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/hotstorage11/exploiting-heat-accelerated-flash-memory-wear-out-recovery-enable-self

[45] Chengen Yang, Hsing-Min Chen, Trevor Mudge, and Chaitali Chakrabarti. 2014. Improving the Reliability of MLC NAND Flash Memories Through Adaptive Data Refresh and Error Control Coding. *Journal of Signal Processing Systems* 76 (09 2014), 225–234. https://doi.org/10.1007/s11265-014-0880-5

[46] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. *Split-Level I/O Scheduling*. Association for Computing Machinery, New York, NY, USA, 474–489. https://doi.org/10.1145/2815400.2815421

[47] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 73–86.

[48] Aviad Zuck, Philipp Gühring, Tao Zhang, Donald E. Porter, and Dan Tsafrir. 2019. Why and How to Increase SSD Performance Transparency. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 192–200. https://doi.org/10.1145/3317550.3321430