Article Type: Top Picks

IOCost: Block IO Control for Containers in Datacenters

Tejun Heo, *Meta Inc, Menlo Park, CA, 94025, USA*
Dan Schatzberg, *Meta Inc, Menlo Park, CA, 94025, USA*
Andrew Newell, *Meta Inc, Menlo Park, CA, 94025, USA*
Song Liu, *Meta Inc, Menlo Park, CA, 94025, USA*
Saravanan Dhakshinamurthy, *Meta Inc, Menlo Park, CA, 94025, USA*
Iyswarya Narayanan, *Meta Inc, Menlo Park, CA, 94025, USA*
Josef Bacik, *Meta Inc, Menlo Park, CA, 94025, USA*
Chris Mason, *Meta Inc, Menlo Park, CA, 94025, USA*
Chunqiang Tang, *Meta Inc, Menlo Park, CA, 94025, USA*
Dimitrios Skarlatos, *Carnegie Mellon University, Pittsburgh, PA, 15213, USA*

Abstract—Resource isolation is a 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. 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 Meta's datacenters comprised of millions of machines, upstreamed IOCost to the Linux kernel, and open-sourced our device-profiling tools.

Containers are swiftly evolving into one of the primary mechanisms for virtualizing capacity in modern datacenters. 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 [1]) for block storage are inadequate for datacenter workloads. 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 [2], [3].

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

Digital Object Identifier 10.1109/XXX.0000.0000000

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 [4], [5], [6], [7]. 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 [1] or limits based on a max bandwidth usage through IOPS or bytes. 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 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 datacenters comprising millions of machines, upstreamed IOCost to the Linux kernel, and open-sourced our device-profiling and benchmarking tools.

Hardware and Workload Heterogeneity

SSD Device Heterogeneity

Incremental hardware refresh and supply chain diversity lead to heterogeneous SSDs in datacenters. Figure 1 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* to measure the sustainable peak performance for each device.

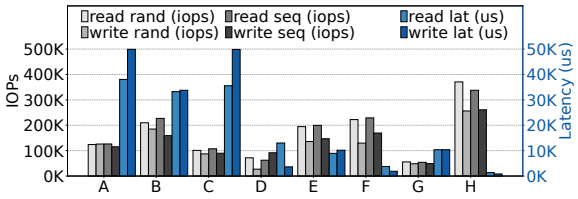

FIGURE 1. Device heterogeneity across Meta's fleet.

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. About 20% of the SSD capacity is spread over 18 devices not shown in the figure but their characteristics are captured by the devices shown.

Workload Heterogeneity

Applications at Meta exhibit a large diversity in their IO behavior. Figure 2 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.

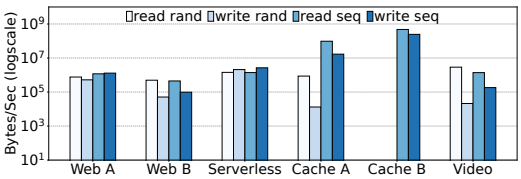

FIGURE 2. IO workload heterogeneity.

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.

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.

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

IOCost, first receives each `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.

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.

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 based on queuing or latency metrics measured since the last planning phase. Next, in step ⑦, IOCost's donation algorithm efficiently donates excess budget to other cgroups to achieve work conservation.

Offline in step ⑧, IOCost leverages profiling, benchmarking, and training across the deployed devices to build cost models and QoS parameters per device.

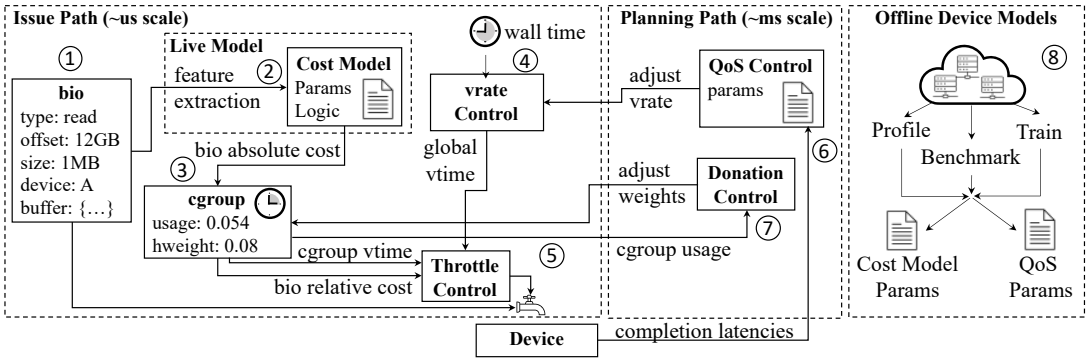

FIGURE 3. Overview of IOCost's architecture for throttling decisions on the left and offline cost model on the right.

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*.

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.

Device Cost Modeling

IOCost decouples device cost modeling from runtime IO control. Cost models are generated offline for each device before deployment. IOCost natively supports a linear model based on bio request properties such as request type, access pattern, and size. For maximum flexibility, IOCost allows a cost model to be expressed as an arbitrary eBPF program. Cost models can be derived by issuing saturating workloads (e.g. as many 4K random reads as possible) to determine the device occupancy consumed of a particular IO operation.

Our tools use `fio` and saturating workloads to infer the linear model's parameters for a device. 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.

Quality of Service

To handle inaccuracies in cost modeling, IOCost dynamically modifies the *vrate* which controls the overall IO issue rate. If the system could issue more IO and

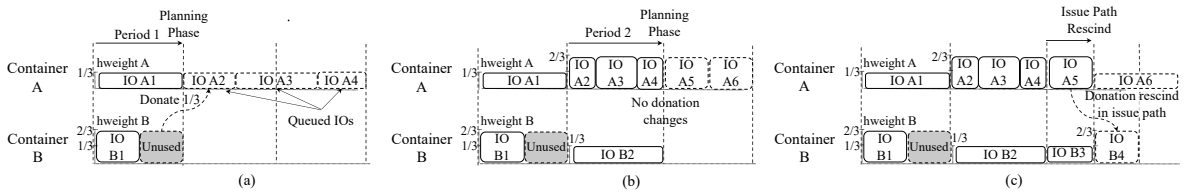

FIGURE 4. Budget donation example at the planning phase (a), after the planning phase (b), and during issue path (c).

the device is not saturated, *vrate* is adjusted upwards. If the device is saturated, *vrate* is adjusted downwards. A system administrator can also bound *vrate* explicitly or implicitly through latency targets which provides a way to tradeoff throughput and latency as workload needs dictate.

We developed a systematic approach to determine QoS parameters for each device in the Meta fleet. Specifically, 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 observe that as *vrate* is lowered too much, ResourceControlBench's memory footprint is limited by the available I/O for paging operations. Yet if *vrate* is configured too high, ResourceControlBench's performance is impacted by a co-located "aggressor" job which attempts to saturate the device. *vrate* is limited between these points to ensure an appropriate tradeoff between throughput and latency.

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 *hweight*s. 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.

High-level Donation Example.
In Figure 4(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 4(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 4(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.

IO Control Across Meta's Fleet

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.

Stacked Latency-Sensitive Workloads

One important production use of IOCost ensures that multiple containers receive their fair proportion of IO service. At Meta, we run a workload similar to Zookeeper which provides a strongly-consistent API for configuration, metadata and coordination primitives. The service triggers a snapshot of the in-memory database which results in momentary write spikes even under nominal loads. The production service has a one second SLO for read and write operations that makes it difficult to collocate with other services.

We analyzed the behavior of this service in a scenario with twelve ensembles. Specifically, we consider

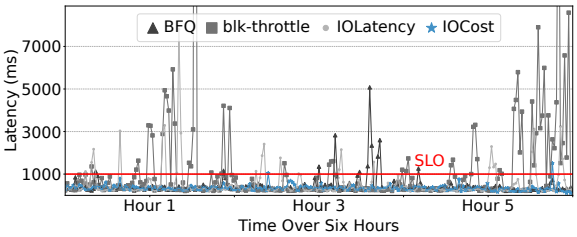

FIGURE 5. Impact of different IO control methods on ZooKeeper latency SLO violations.

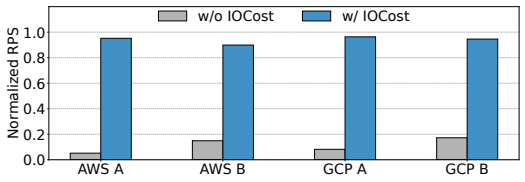

FIGURE 6. Requests per second (RPS) of a latency-sensitive workload when stacked with a memory-leak workload in AWS EBS, and Google Cloud Persistent Storage.

eleven well-behaved ensembles that are collocated with a twelfth ensemble that behaves as a noisy neighbor. Figure 5 shows the P99 latency. 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. `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.

Remote Storage and Cloud 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 replace the production web server at Meta with ResourceControlBench that is collocated with a high-speed memory-leak program running in a low priority cgroup. We then report the drop in ResourceControlBench's RPS as a measurement of how well IOCost protects the workload from interference.

We run the two workloads in a public cloud's VM whose guest OS is configured with IOCost. Figure 6 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 is robust and can be successfully applied to environments outside Meta.

Adoption in Production and Future Research Directions

IOCost is a holistic solution that introduces for the first time effective IO control for containerized environments and heterogeneous IO devices in datacenters.

A Holistic Resource Control Solution for Datacenters

IOCost introduces an end-to-end approach for IO control for containers that addresses the need of datacenter environments. Specifically, IOCost presents a robust methodology to accurately estimate device-specific occupancy models and a set of Quality-of-Service parameters offline. Furthermore, it separates scheduling work in the kernel between a fast and a slow path, leverages cgroup semantics, device-specific models, and online device behavior. The approach and methodology of IOCost can enable future research into resource control as the memory subsystem becomes increasingly heterogeneous with multiple memory tiers and upcoming interconnects and devices enabled by CXL.

Integration with Memory Offloading

IOCost's design enabled other use cases within Meta. Specifically, IOCost enabled efficient memory offloading to secondary storage to save memory through paging and swap [8]. Regularly depending on secondary storage through the memory subsystem has placed particularly high demands on the block layer to ensure fairness, low latency and high utilization. Advances to storage media are likely to lead to this model of memory offloading becoming even more prevalent and further the importance of IOCost to work well on a wide range of devices.

Linux Upstreaming and Open Source Toolsets

IOCost is upstreamed into the Linux kernel and the device-profiling and benchmarking tools have been made open source. Our future work is to create a unified database of per-device profiles that can be automatically pulled and deployed by various Linux distributions. The benchmarking and profiling tools of IOCost will help future work calibrate and optimize experimental setups to closely match production scenarios.

IOCost's Integration with Systemd

IOCost is currently being integrated into systemd that provides a suite of basic system management blocks for the Linux system. With the integration of IOCost into systemd most Linux deployments will be able to automatically take advantage of the IOCost benefits within and outside the datacenter. Dynamically tuning IOCost in the kernel and device occupancy models across environments, such as mobile, provides rich opportunities for performance and power optimizations.

Adoption in The Cloud

With IOCost available in the upstream Linux kernel, organizations other than Meta have begun to deploy it with success. *DigitalOcean*, a cloud hosting provider, offers Infrastructure-as-a-Service (IaaS) in a multi-tenant environment. In such environments, tenants tend to consume shared resources such as IO which may result in noisy neighbor challenges where one user's load negatively impacts other users. *DigitalOcean* is actively using IOCost to eliminate such issues and ensure that the overall performance of IO resources are functioning at optimum levels. They have successfully deployed IOCost across their entire fleet of servers with automation to tune QoS settings to eliminate the worst noisy neighbor cases. Additionally, *Alibaba* cloud provides a detailed documentation on how to configure IOCost on their cloud infrastructure [9].

Budget Donation Beyond IO

IOCost's novel budget donation algorithm for IO can be adopted for efficient resource control for other resources beyond IO. Managing deep cgroup hierarchies of containerized workloads efficiently is a major challenge for datacenter environments due to the high cost of traversing the cgroup hierarchy. IOCost's approach enables budget donations with only local updates along their path in the hierarchy to the root. This approach can enable efficient budget donations for other critical resources such as CPU and memory.

Integrating IO control

Integrating IO control and SSD devices into the system stack introduces several challenges. IOCost's approach highlights the need for building high-fidelity device occupancy models that will help to further improve IO control and workload collocation efficiency. IOCost further provides insights into how future SSD architectural designs can take into account datacenter requirements. Future research in SSD hardware and interfaces can help eliminate unpredictable device behavior due to hardware-specific operations such as garbage collection and caching.

Tejun Heo Tejun is a kernel engineer at Meta and can be reached at htejun@meta.com.

Dan Schatzberg Dan is a research scientist at Meta and can be reached at dschatzberg@meta.com.

Andrew Newell Andrew was a research scientist at Meta during this work. Now he is with Tesla and can be reached at andyjnewell@gmail.com.

Song Liu Song is a kernel engineer at Meta and can be reached at songliubraving@meta.com.

Saravanan Dhakshinamurthy Saravanan is a production engineer at Meta and can be reached at saravanand@meta.com.

Iyswarya Narayanan Iyswarya is a performance and capacity engineer at Meta and can be reached at inarayanan@meta.com.

Josef Bacik Josef is a kernel engineer at Meta and can be reached at jbacik@meta.com.

Chris Mason Chris is an engineering director at Meta and can be reached at clm@meta.com.

Chunqiang Tang Chunqiang is a senior director at Meta and can be reached at tang@meta.com.

Dimitrios Skarlatos Dimitrios is an assistant professor of computer science at Carnegie Mellon University and can be reached at dskarlat@cs.cmu.com.

REFERENCES

1. P. Valente and F. Checconi, "High throughput disk scheduling with fair bandwidth distribution," *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1172–1186, 2010.

2. J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “The unwritten contract of solid state drives,” in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys ’17, (New York, NY, USA), p. 127–144, Association for Computing Machinery, 2017.
3. F. Chen, D. A. Koufaty, and X. Zhang, “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*, SIGMETRICS ’09, (New York, NY, USA), p. 181–192, Association for Computing Machinery, 2009.
4. A. Gulati, I. Ahmad, and C. A. Waldspurger, “PARDA: Proportional allocation of resources for distributed storage access,” in *7th USENIX Conference on File and Storage Technologies (FAST 09)*, (San Francisco, CA), USENIX Association, Feb. 2009.
5. A. Gulati, A. Merchant, and P. J. Varman, “mclock: Handling throughput variability for hypervisor IO scheduling,” in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, (Vancouver, BC), USENIX Association, Oct. 2010.
6. L. Huang, G. Peng, and T.-c. Chiueh, “Multi-dimensional storage virtualization,” *SIGMETRICS Perform. Eval. Rev.*, vol. 32, p. 14–24, June 2004.
7. A. Singh, M. Korupolu, and D. Mohapatra, “Server-storage virtualization: Integration and load balancing in data centers,” in *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12, 2008.
8. J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, “Tmo: Transparent memory offloading in datacenters,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, (New York, NY, USA), p. 609–621, Association for Computing Machinery, 2022.
9. Alibaba Cloud. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/configure-the-weight-based-throttling-feature-of-blk-iocost.