IOCost: Block Input–Output Control for Containers in Datacenters

Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, and Chunqiang Tang, *Meta Inc., Menlo Park, CA, 94025, USA*

Dimitrios Skarlatos, *Carnegie Mellon University, Pittsburgh, PA, 15213, USA*

Resource isolation is a requirement in datacenter environments. However, our production experience in Meta's large-scale datacenters shows that existing input–output (IO) control mechanisms for block storage are inadequate in containerized environments. This article presents IOCost, an IO control solution designed for containerized environments that 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 comprising 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 input–output (IO) control mechanisms [e.g., Budget Fair Scheduling (BFQ[1])] for block storage are inadequate for datacenter workloads. There are several challenges with providing robust IO control for containers. First, IO control needs to account for hardware heterogeneity in datacenters. Multiple generations of solid-state drives (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 overexert 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 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 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.

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3277783
Date of publication 25 May 2023; date of current version 29 June 2023.

Furthermore, application developers often cannot effectively estimate IO needs in terms of metrics like the internetworking operating system (IOP) 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 virtual machine (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 (OS), the interactions of IO with the memory subsystem, and heavily stacked deployments. State-of-the-art solutions in the Linux kernel rely on either BFQ[1] or limits based on a maximum bandwidth usage through the IOP 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 the IOP 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 the IOP or bytes.

IOCost works by estimating device occupancy of each IO request using a device-specific model. For example, a 4-KB read would have a different cost on a high-end SSD than on a spinning disk. With a model of occupancy and additional quality-of-service (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 that 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 the IOP 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.

The eight types of SSDs (A–H) show distinctive characteristics. Specifically, SSD H achieves high IOP at a low latency, SSD G offers low IOP and a relatively low latency, and SSD A provides moderate IOP with a higher latency. Each device usually represents less than 14% of the total fleet. Roughly 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 versus writes and random versus sequential bytes. Workloads like "Web A" and "Web B" are most typical of Meta workloads, with a moderate amount of reads and writes mixed approximately equally in terms of random and sequential operations. Serverless workloads at Meta are highly overcommitted and exhibit a mixed

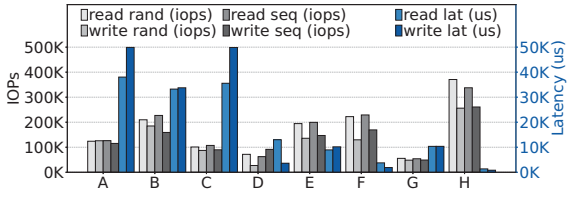

FIGURE 1. Device heterogeneity across Meta's fleet.

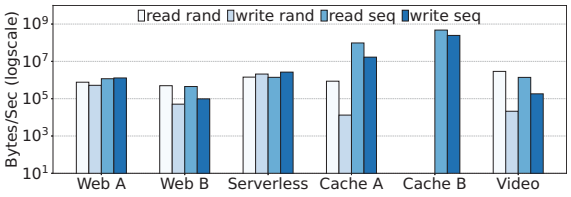

FIGURE 2. IO workload heterogeneity.

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.

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, IOP, or bytes per second), 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 considers 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 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 IOP.

Figure 3 provides an overview of IOCost's architecture. IOCost is logically separated into the "*Issue Path*," which operates on a microsecond timescale for each IO operation (called a `bio` in Linux), and the "*Planning Path*," which 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 20 ms 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 to which the cgroup is entitled.

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

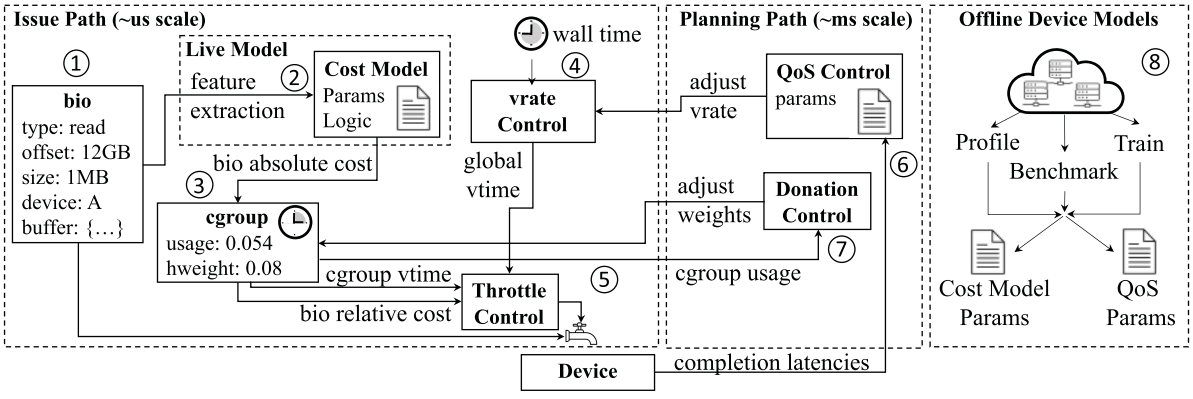

FIGURE 3. Overview of IOCost's architecture for (a) throttling decisions and (b) the offline cost model. Params: parameters.

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 the *vrate* and, consequently, the total IO issued in response to device feedback. Modeling may over- or underestimate true device occupancy, and this *vrate* adjustment ensures that 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.

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 that 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 because 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 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's behavior and adjusts how much IO can be issued across all cgroups by adjusting the *vrate* to control how fast or slow the global *vtime* runs compared to the wall clock. For example, if the *vrate* is at 150%, the global *vtime* runs at 1.5 times the speed of the wall clock and generates 1.5-times-more IO budget than the device's cost model specifies. The conditions and range of the *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 extended Berkeley Packet Filter (eBPF) program. Cost models can be derived by issuing saturating workloads (e.g., as many 4 K 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 30 different storage devices found in Meta datacenters. We have made our modeling tools available in the Linux source tree.

QoS

To handle inaccuracies in cost modeling, IOCost dynamically modifies the *vrate* that controls the overall IO issue rate. If the system could issue more IO and the device is not saturated, the *vrate* is adjusted upwards. If the device is saturated, the *vrate* is adjusted downwards. A system administrator can also bound the *vrate* explicitly or implicitly through latency targets, which provides a way to trade off 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 the *vrate* is lowered too much, ResourceControlBench's memory footprint is limited by the available IO for paging operations. Yet, if the *vrate* is configured too high, ResourceControlBench's performance is impacted by a co-located "aggressor" job, which attempts to saturate the device. The *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 the `vrate`, but found that local adjustment of weight was the only solution that met all of the following requirements: 1) the issue path remains low overhead, 2) the total amount of IO issued never exceeds what the `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 $1/3$ and $2/3$, 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 application programming interface 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 1-s service level objective (SLO) for read and write operations that makes it difficult to co-locate with other services.

We analyzed the behavior of this service in a scenario with 12 ensembles. Specifically, we consider 11 well-behaved ensembles that are co-located with a 12th 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

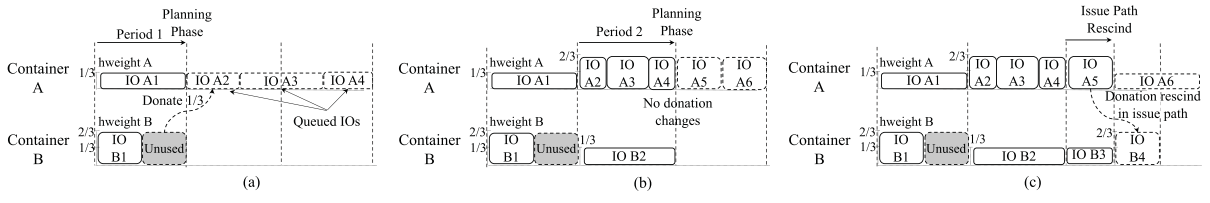

FIGURE 4. (a) Budget donation example at the planning phase, (b) after the planning phase, and (c) during issue path.

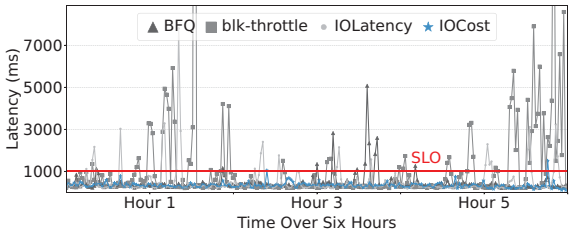

FIGURE 5. Impact of different IO control methods on ZooKeeper latency SLO violations.

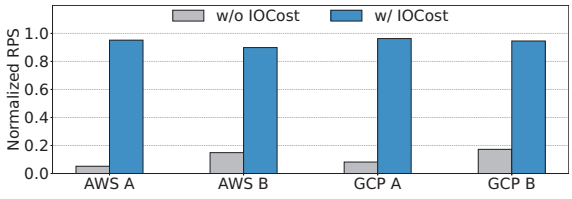

FIGURE 6. Requests per second (RPS) of a latency-sensitive workload when stacked with a memory-leak workload in Amazon Web Services EBS, and Google Cloud Persistent Storage.

repeatedly violate their 1-s SLO throughout the 6-h experiment. Specifically, `blk-throttle` shows 78 violations, with some lasting tens of seconds. `BFQ` shows 13 violations, each lasting 2–5 s. `IOLatency` cannot be configured for proportional control and also shows poor behavior of 31 violations, with the longest being 7.8 s. With `IOCost`, the effects of the noisy neighbor ensemble and snapshots were appropriately isolated, resulting in only two marginal violations of 1.5 and 1.04 s.

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, which is co-located with a high-speed memory-leak program running in a low-priority cgroup. We then report the drop in ResourceControlBench's requests per second 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 Amazon Web Services Elastic Block Store (gp3-3000iops, io2-64000iops) and two Google Cloud Persistent Disk configurations (balanced, SSD). Although 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 QoS 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 a compute express link (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 will 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, which provides a suite of basic system management blocks for the Linux system. With the integration of IOCost into systemd, most of the 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 in a multitenant 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 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's 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 further improve IO control and workload co-location efficiency. IOCost further provides insights into how future SSD architectural designs can consider 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.

REFERENCES

1. P. Valente and F. Checconi, "High throughput disk scheduling with fair bandwidth distribution," *IEEE Trans. Comput.*, vol. 59, no. 9, pp. 1172–1186, Sep. 2010, doi: 10.1109/TC.2010.105.
2. J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The unwritten contract of solid state drives," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA: Association for Computing Machinery, 2017, pp. 127–144, doi: 10.1145/3064176.3064187.
3. F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst. (SIGMETRICS)*, New York, NY, USA: Association for Computing Machinery, 2009, pp. 181–192, doi: 10.1145/2492101.1555371.
4. A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *Proc. 7th USENIX Conf. File Storage Technol. (FAST)*, San Francisco, CA, USA: USENIX Association, Feb. 2009, pp. 85–98.
5. A. Gulati, A. Merchant, and P. J. Varman, "mclock: Handling throughput variability for hypervisor IO scheduling," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, Vancouver, BC, Canada: USENIX Association, Oct. 2010, pp. 437–450.
6. L. Huang, G. Peng, and T.-C. Chiueh, "Multi-dimensional storage virtualization," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 14–24, Jun. 2004, doi: 10.1145/1012888.1005692.
7. A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2008, pp. 1–12, doi: 10.1109/SC.2008.5222625.
8. J. Weiner et al., "TMO: Transparent memory offloading in datacenters," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 609–621, doi: 10.1145/3503222.3507731.
9. "Configure the weight-based throttling feature of blk-iocost." Alibaba Cloud. Accessed: 2023. [Online]. Available: https://www.alibabacloud.com/help/en/elastic-compute-service/latest/configure-the-weight-based-throttling-feature-of-blk-iocost

TEJUN HEO is a kernel engineer at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at htejun@meta.com.

DAN SCHATZBERG is a research scientist at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at dschatzberg@meta.com.

ANDREW NEWELL was a research scientist at Meta Inc., Menlo Park, CA, 94025, USA during this work. He is currently with Tesla Contact him at andyjnewell@gmail.com.

SONG LIU is a kernel engineer at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at songliubraving@meta.com.

SARAVANAN DHAKSHINAMURTHY is a production engineer at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at saravanand@meta.com.

IYSWARYA NARAYANAN is a performance and capacity engineer at Meta Inc., Menlo Park, CA, 94025, USA. Contact her at inarayanan@meta.com.

JOSEF BACIK is a kernel engineer at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at jbacik@meta.com.

CHRIS MASON is an engineering director at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at clm@meta.com.

CHUNQIANG TANG is a senior director at Meta Inc., Menlo Park, CA, 94025, USA. Contact him at tang@meta.com.

DIMITRIOS SKARLATOS is an assistant professor of computer science at Carnegie Mellon University, Pittsburgh, PA, 15213, USA. He is a Member of IEEE. Contact him at dskarlat@cs.cmu.com.