USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Come Hell or Still Water: Alleviating Tail Latency in Cloud Block Store

Chaolei Hu, *Tsinghua University and Alibaba Cloud;* Kun Qian, Erci Xu, Yifan Shen, Haoran Zhang, Xue Li, Yuesheng Gu, and Lingjun Zhu, *Alibaba Cloud;* Fengyuan Ren, *Tsinghua University;* Ennan Zhai, *Alibaba Cloud*

https://www.usenix.org/conference/nsdi26/presentation/hu-chaolei

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Come Hell or Still Water: Alleviating Tail Latency in Cloud Block Store

Chaolei Hu[1,2], Kun Qian[2], Erci Xu[2†], Yifan Shen[2], Haoran Zhang[2],
Xue Li[2], Yuesheng Gu[2], Shu Ma[2], Lingjun Zhu[2], Fengyuan Ren[1†], Ennan Zhai[2]

[1]*Tsinghua University* [2]*Alibaba Cloud*

Abstract

Maintaining low tail latency is crucial for cloud storage services. In ALIBABA CLOUD, our Elastic Block Storage (EBS), like many others, adopts layers of load balancing to avoid hot-spot I/Os, a dominant contributor to tail latency.

However, in the field, EBS has still been suffering from tail latency spikes. Through extensive analysis of production workloads, we have identified the root cause: the workload bursts caused by a small group of Virtual Disks (VDs), which fundamentally influence the tail latency of the entire cluster. We hence propose a lightweight dual-bucket throttling mechanism to effectively mitigate the issue while maintaining fairness. In addition, we discover that, even under underloaded scenarios, the tail latency remains suboptimal due to the event-loop thread model. We propose a priority-based scheduling mechanism to separate I/O-related tasks from I/O-unrelated ones. Our evaluation shows that the proposed mechanisms can reduce the tail latency by up to 97% in burst and 43% in underloaded scenarios. Our mechanisms have been deployed across dozens of clusters for more than three months, and have served hundreds of trillions of I/O requests. They reduce the P99999 tail latency of steady segments by 59.7% under burst scenarios and of all I/Os by 22% in underloaded scenarios.

1 Introduction

Elastic Block Storage (EBS) plays an essential role in today's cloud ecosystem. Formatted as Virtual Disks (VDs) to Virtual Machines (VMs), EBS has been supporting workloads from single-server databases to large-scale data analytics.

The high-performance nature of these applications has placed demanding requirements on EBS. Hence, many cloud vendors, including Azure [5], AWS [2], and us at ALIBABA CLOUD, choose the compute-to-storage disaggregation architecture for building EBS. Under this setup, the compute cluster—which accommodates the nodes for hosting the VMs—does not store data. Rather, it interconnects through the datacenter network to a backend storage cluster which typically includes two sets of servers, proxy servers (for address translation) and storage servers (for data persistence). This architecture allows a VD to be served across nodes in the storage backend, providing a highly scalable throughput. Moreover, this disaggregated setup also allows the compute and storage clusters to evolve independently, enabling quick adoption of new technologies (e.g., RDMA [16]).

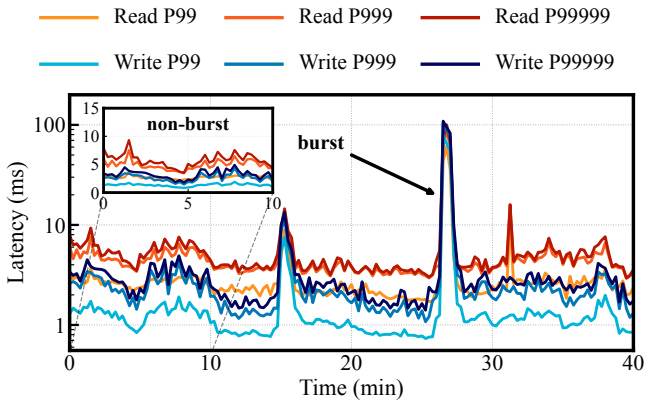


Figure 1: Tail latency in a Cluster.

Still, a performance challenge remains: the tail latency. Figure 1 shows that in a representative cluster, the tail latency can reach up to 109ms at the P99999 level. One major and common contributor to tail latency is the queued-up I/O requests caused by hot-spot accessing. In the field, we have constantly observed similar patterns in EBS where only a small proportion (0.1% to 11.6%) of the VDs in an EBS cluster are frequently accessed. To mitigate this issue, we equip EBS with multiple layers of load balancing strategies at VD, compute node and storage node levels.

Unfortunately, simply balancing the load is not enough. In the field, we have been witnessing tail latency spikes (defined as I/O latency exceeding 10ms, and multiple spikes occurring during a 10-minute window are treated as one). For example, over one week of monitoring on around 100 randomly selected EBS clusters has reported a total of 649 tail latency spikes, and the most severely hit EBS cluster has experienced 289 spikes. The situation will further deteriorate during representative online events like shopping festivals where nearly all clusters used by the promoters endure constant tail latency spikes during peak hours.

Interestingly, detailed profiling reveals that the culprit is not the ineffective load balancing. Instead, it is the sheer volume of I/O requests from a few VDs within a short period that overwhelms the entire cluster. Due to the load balancing, such a burst ironically floods almost all storage nodes within the cluster and subsequently causes tail latency spikes. This overly concentrated I/O access pattern driven by VDs owned by a small group of users (a.k.a., major clients) is fairly com-

†Corresponding authors

mon. For example, more than 51% of VDs are possessed by the top 3 major clients in each cluster. More importantly, while it is acceptable for the major clients to experience a longer latency since they initiated the burst, the minor clients that are co-located with the major ones, should not be but in fact are severely impacted by worsening tail latency.

Initially, we intended to address this issue by strictly capping the throughput on VDs owned by the major clients. However, this approach, while effective for reducing bursts, backfires as it severely impacts the bandwidth utilization during underloaded periods (i.e., the majority of the time). Statistically, by throttling the throughput of each major client to 60 MBps, the EBS would suffer 27% utilization deterioration.

By revisiting the EBS stack and burst traces, we discover that the underlying storage layer can tolerate temporary I/O spikes. Moreover, the bursts driven by major clients are usually short-lived (i.e., mostly tens of milliseconds) and only target a small portion of the Logical Block Address (LBA). Therefore, we should, instead of capping all major clients' traffic, identify and throttle hot segments (i.e., partitions of the VDs) in a burst. The remaining "steady" segments can proceed as usual or even *preempt* bandwidth resources from the hot ones to offset impacts from the burst.

Based on this key idea, we use a theoretical model to accurately characterize the available bandwidth for "steady" segments under various load conditions. Based on this estimation, we propose a dual-bucket throttling mechanism, to reduce the tail latency of steady segments during bursts. At a high level, the mechanism works by setting up two token buckets in each proxy server and refilling the buckets at a fixed rate on an interval basis (every 10ms in our case). The segment can issue an I/O after obtaining a token from the buckets. The *main bucket* is up for grabs for all segments, while the *cheque bucket* is only accessible to steady ones. The goal of the cheque bucket is to allow temporary I/O bursts (i.e., at most one interval) and preserve the bandwidth for steady segments.

Although the overall latency in Figure 1 remains relatively low under underloaded scenarios, our breakdown analysis revealed that the proxy layer still dominates the tail latency, contributing over 49.2% to the end-to-end latency — far exceeding the expected lightweight role of a proxy server. We found that the event-based scheduling framework (i.e., libevent, a common library for event-driven programming) can significantly postpone the processing of I/O requests due to delayed reception of new I/O requests and inefficient scheduling of tasks on the critical path.

We hence develop a priority-based task scheduler to further categorize the tasks into I/O-related and I/O-unrelated tasks, and allocate them to corresponding queues. The I/O-related tasks are always processed first. In contrast, the processing of I/O-unrelated tasks is delayed until all I/O-related ones are processed, and will be interrupted if the overall execution time exceeds a timeout (10us by default).

We extensively evaluate our proposed solutions by combining comprehensive microbenchmarks, real-world traces, and large-scale deployments across diverse setups. In production, dual-bucket throttling and the priority-based task scheduler reduce P99999 tail latency by 59.7% in burst scenarios and 22% in underloaded scenarios.

Our contributions are as follows:

- We have identified the simultaneous overload by major clients as the main contributor to the tail latency spikes in EBS. We have proposed a dual-bucket throttling mechanism to alleviate the tail latency spikes for minor clients.
- We have revealed that the existing event-based scheduling framework, even with thread splitting, can still suffer from suboptimal tail latency under underloaded scenarios. We further developed a priority-based task scheduler to schedule I/O-related and I/O-unrelated tasks separately.
- We have deployed both mechanisms at scale across dozens of clusters, serving hundreds of trillions of I/O requests, where they cut steady-segment P99999 tail latency by 59.7% in bursts and of all I/Os by 22% in underloaded scenarios.
- We have released production I/O traces of typical burst scenarios collected from ALIBABA CLOUD EBS at https://tianchi.aliyun.com/dataset/219284 to benefit future research and optimizations on related topics.

The rest of the paper is organized as follows. We introduce the background of EBS in §2. Then, we move on to analyze the symptoms, root causes and the (unsuccessful) simple fixes of latency spikes under both burst and underloaded scenarios in §3. We present the design and implementation of our proposed dual-bucket throttling and priority-based task scheduler in §4 and extensively evaluate both strategies in §5. We end this paper with a discussion on related work and a short conclusion.

2 Background

2.1 Elastic Block Storage (EBS) Architecture

Overview. Similar to block store services by other vendors (e.g., Azure and AWS [11, 2]), our EBS service at ALIBABA CLOUD also follows the compute-storage disaggregation, to allow flexible and scalable deployment and management. Figure 2 highlights the three-layer architecture of EBS, including the compute, proxy, and storage layer.

Compute Layer. In EBS, the compute side consists of multiple compute clusters. Each cluster can have tens to hundreds of compute nodes*. Each compute node hosts one or more virtual machines (VMs) and each VM can be attached to one or more virtual disks (VDs). Inside a VD, its Logical

*To avoid ambiguity, throughout this paper, we use the term *node* (e.g., compute and storage node) to refer to the physical machine, and *server* to the software process (e.g., proxy server) running on the node.

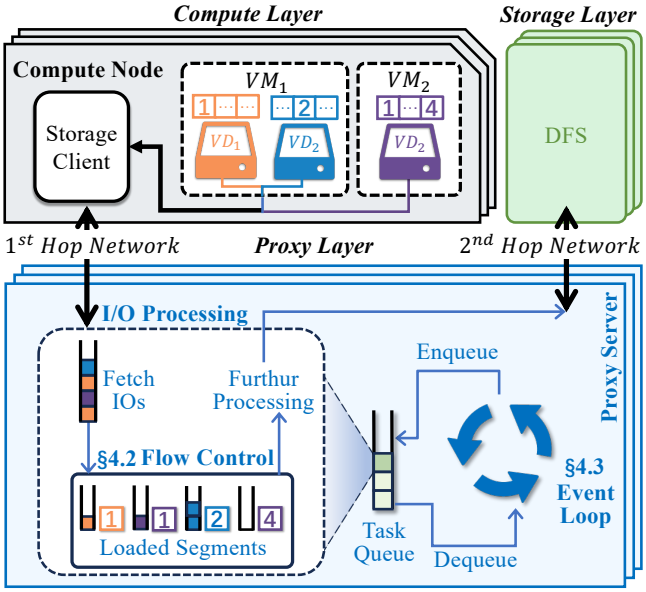


Figure 2: EBS Architecture Overview.

Block Address (LBA) space is split into a series of fixed-size (32 GiB by default) segments, namely the numbered blocks in Figure 2.

Under the disaggregation setup, when a VM issues I/Os to a VD, all requests will be first processed by the storage client running inside the hypervisor. The storage client adds the packet headers to the requests and further forwards them to the backend via the network (a.k.a., the 1st hop).

Proxy Layer. The storage backend includes two layers, the proxy and storage. In our EBS, the storage layer is essentially an append-only distributed file system. The key functionality of the proxy layer is to translate the block-based semantics of VD's requests to the file semantics of the underlying storage layer. Additionally, the proxy servers in this layer also perform garbage collection, indexing for reads, data scrubbing, and many other necessary tasks.

To efficiently orchestrate the various tasks listed above, the proxy servers adopt an event-driven thread model to reduce context switching and scheduling overhead. Under this model, each proxy server runs an infinite event loop, and adds the pending tasks (e.g., incoming I/Os and scheduled garbage collection) to a single queue. The tasks are then executed in a FIFO fashion, a common practice in large-scale systems, such as Demikernel [30], IX [8] and mTCP [19].

Storage Layer. At a high level, the underlying distributed file system (DFS) is similar to HDFS. It provides data persistence and retrieval service with the *append-only file* abstraction. In addition, it offers reliability support via data redundancy including either three-way replication or erasure coding. Note that, our distributed file system also has a set of metadata servers (like namenode in HDFS) and multiple data servers (i.e., datanode in HDFS). While each (physical) storage node hosts one proxy server and one data server,

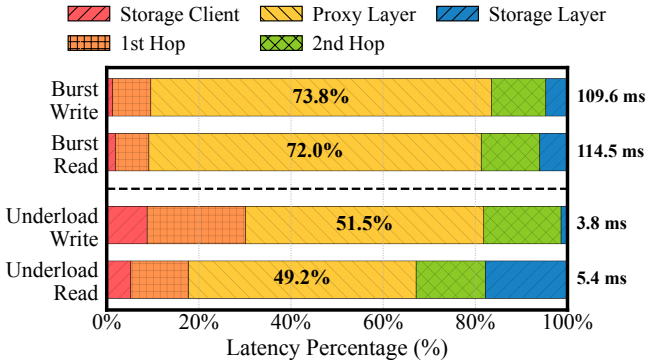


Figure 3: Breakdown of the 99.999th percentile tail latency under burst and underloaded workloads.

we do not enforce the strong locality in transferring data between the two processes. Instead, a proxy server only communicates with the data server via an intra-cluster network (named as the 2nd hop, see Figure 2).

2.2 Load Balancing in EBS

Achieving a balanced load distribution is crucial for improving the overall performance and, especially, the tail latency. To reach this goal, first, the Storage Client (on the compute node) is equipped with multiple hyper threads (e.g., 4-8) to handle VDs' requests in a round-robin fashion, avoiding the storage client being drained by a single VD. Furthermore, along the stack, it is obvious that the requests from VDs are shuffled among the proxy servers, thereby making the proxy server an ideal place for performing load balancing. Currently, at the proxy layer, EBS mainly employs three strategies including segmentation, allocation and migration. In particular, VDs of each client are evenly distributed across clusters within an availability zone during allocation, preventing over-concentration on any single cluster.

Segmentation. Recall that a VD's LBA is divided into a series of fixed-size segments, and the proxy server operates at the segment level. Thus, a VD is handled by different proxy servers. However, such granularity is not enough for balancing the load. One major shortcoming is that the burst I/Os often tend to concentrate on a short range of LBA (i.e., several MiBs) which is much smaller than the default segment size (i.e., 32 GiB). If we adjust the segment size to more closely align with the MiB-level of typical hot spots, the overhead of metadata can increase by three orders of magnitude or more, severely stressing the metadata servers.

Therefore, we choose to first divide the entire LBA space into 2-MiB data blocks and then place them into the 32-GiB segments in a round-robin manner. For example, if a VD's capacity is 320 GiB (i.e., 10 segments), then the first segment will contain the 2-MiB data blocks from LBA 0-2 MiB, 20-22 MiB, and so on. This way, the hot spots are much more likely to be distributed across different proxy servers, thus improving the load balancing.

Allocation. EBS further integrates a series of strategies to

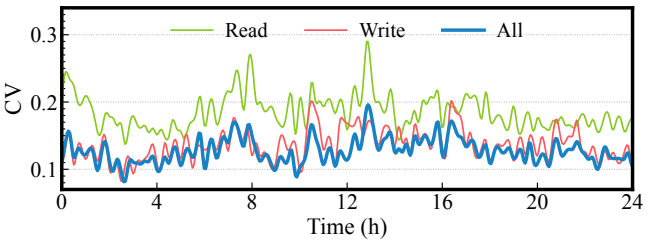


Figure 4: Intra-cluster request arrival CV for read, write, and total traffic.

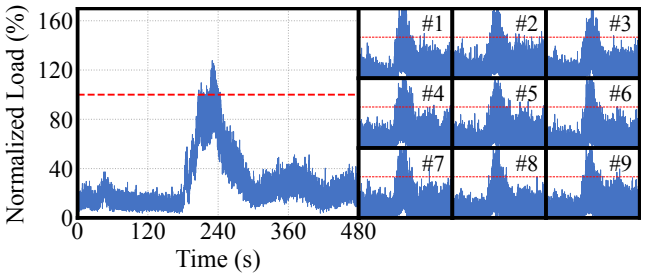


Figure 5: Cluster- and node-level views of a workload burst.

optimize the resource allocation. First, a user can subscribe multiple VDs (for different VMs) and they may tend to have similar access patterns. Therefore, we will scatter such VDs across different EBS clusters to alleviate the potential load imbalance. Second, similarly, on the storage end, EBS enforces the segments from a VD to be distributed across at least 4 different proxy servers.

Migration. Besides static measures like segmentation and allocation, EBS also supports dynamic load balancing via intra-/inter-cluster segment migrations. While both types of migrations follow similar procedures (i.e., stop I/Os on the source server, copy the metadata, and resume I/Os on the destination one), the inter-cluster migration incurs additional overheads due to the cross-cluster traffic and longer transferring time (e.g., 20 minutes for inter-cluster migration). Therefore, we only migrate segments to a different cluster when the load imbalance persists on the existing one for a certain period of time (i.e., 10 minutes at the moment).

3 Motivation

3.1 What Went Wrong?

Due to the consequential impacts on Service Level Objectives (i.e., user experiences), handling high tail latency has always been a priority of maintaining cloud services [6]. Previous work has shown a major cause of high tail latency to be I/O bursts [7]. With different levels of load balancing mechanisms in EBS enabled (see §2), one question inevitably arises: do the layered load balancing mechanisms successfully defend against the long tail latency in EBS?

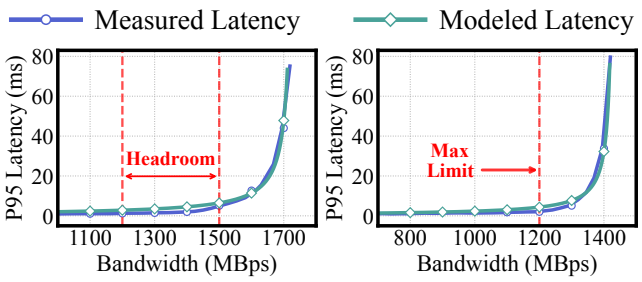
The short answer, unfortunately, is no. In production, we have still been experiencing frequent occurrences of tail latency spikes. In Figure 1, we demonstrate various levels of tail (from P99 to P99999)† I/O latency in an EBS cluster for nearly an hour. We can see that both read and write have suffered multiple tail latency spikes where P99999 metrics can reach up to 109 milliseconds. Our monitoring further shows that such spikes not only frequently occur but also can be widely spread across EBS clusters. We collected the latency statistics of I/O requests from around 100 clusters in a recent week, and we recorded 649 tail latency spikes. 289 of them even occurred in the same cluster. Furthermore, during shopping festivals and other special events, nearly all clusters serving these events constantly endure I/O bursts during peak hours.

To pinpoint the culprit, we start by profiling the distribution of the latency spikes by EBS components. In EBS, an I/O has to go through the storage client, the 1st hop network, the proxy server, the 2nd hop network, and finally the storage server (see Figure 2). In Figure 3, we present the breakdown under P99999 tail latency, where the network latency encompasses both the wire transmission delay and the time the packet waits in the NIC receive buffer before being consumed by the application. We discover that, for both read and write, the proxy server is the dominant contributor. To understand the root cause, we further cross check the software/hardware logs and the traffic monitoring data. The reason, instead of malfunctioning software/hardware components, remains to be the workload bursts.

Initially, we hypothesize that the load balancing mechanisms in EBS are suboptimal. To verify this, we examine the distribution of workload across proxy servers in EBS clusters. The results, however, do not support this hypothesis. We use the coefficient of variation (CV) – defined as the standard deviation divided by the mean of the request arrival rate among proxy servers in a cluster – as our metric for evaluating load balance. As shown in Figure 4, the CV averages 0.12 and oscillates between 0.08 and 0.18 over time, with similar results observed across other clusters. These results indicate that the load is well balanced, and are also comparable to prior reports from large-scale production systems [14, 12]. This very effectiveness rules out imbalance as the main culprit of the latency spikes; instead, the more plausible explanation is that the workload bursts are so severe that they overwhelm the layers of load balancing, causing requests to accumulate and spikes in latency.

Symptom. Figure 5 gives both a macroscopic and microscopic view of a burst scenario in an EBS cluster. On the left, we can see the cluster-wide normalized load during the burst, and on the right, we present the load patterns on 9 randomly selected servers (out of nearly 100) within the same cluster. Figure 6 shows, without imposing a bandwidth cap on proxy servers, the end-to-end latency increases dramatically. The root cause is that the storage layer has limited serving capacity. Issuing excessive requests can introduce super-linear waiting time. Based on evaluations and field experiences,

†P99999 tail latency is a standard metric closely monitored by major cloud services [3, 31].



(a) GC Rate = 300 MBps **(b)** GC Rate = 600 MBps

Figure 6: With the gradual increase of load in a proxy server, the end-to-end latency increases super-linearly, making throttling necessary.

we set the maximum limit to 1.2GBps (100% load in Figure 5.) The figures clearly show that all servers follow the very pattern the cluster experiences, indicating that the load balancing mechanisms are functioning as intended. It is simply that the burst workload has driven servers in the cluster to be overloaded (i.e., across the 100% load line), rendering long queueing delays (i.e., tail latency spikes).

Root causes. Surprisingly, this phenomenon is not caused by many users collectively, but by just a few major clients—sometimes even one. Field statistics show that, among the randomly sampled 20 EBS clusters, the top user can contribute up to 95% of the total traffic in a cluster and the top three, on average, account for 63%. This is inherent to multi-tenant clouds, where a small number of major clients inevitably account for the majority of resources.

Such dominance prevails not just in the burst scenario but widely in the production environment. Major clients (i.e., top three) own 51% of the total VDs in the cluster. This situation easily allows the traffic patterns of major clients to impact those of the cluster. And, with the load balancing mechanisms in place, the overwhelming traffic from the major clients' VDs can lead to an all-server overload, causing the simultaneous spikes shown in Figure 5.

Note that, even within major clients' VDs, the skewness is considerable. We compute the CV of all segments' load from the top 3 major clients in different clusters. The CV could be up to 553 and the average value is 47.7, showing great load variance of segments in different VDs.

Impact and Simple Fixes. Now, with a series of research questions (RQs), we intend to explore the rationale, the consequences, and the potential solutions.

RQ1: Why is this phenomenon bound to happen? To avoid over-concentration of traffic, we have already distributed VDs from major clients across different servers and clusters. Yet, the sheer volume of requests nevertheless drives all related servers (and the cluster) to be overloaded. Note that, enforcing a more stringent distribution policy, say setting the maximum number of VDs a client can own in a cluster, can be challenging due to the limited number of clusters in an availability zone. In addition, while certain "bursty" traffic is foreseeable before events like shopping festivals, the individual suddenness remains. Therefore, it is difficult to predict and (re)allocate resources in advance. *RQ2: Do the impacts only affect the major clients?* A naïve opinion is that the latency spikes are acceptable since it is the major clients who flooded the clusters in the first place. Unfortunately, the reality is that the latency spikes are not limited to the major clients. In fact, while the major clients are certain to take severe hits, the latency spikes also exist among the minors, with a 20× tail latency increase for I/Os of the "innocent" users.

RQ3: Why not limit the traffic from compute ends? One straightforward solution is to directly limit the traffic inside the storage client. Obviously, placing a limit on the bandwidth of major clients can alleviate the latency spikes. However, setting an appropriate threshold is challenging. On the one hand, a fixed limit can lead to either resource underutilization (for being too strict) or ineffectiveness (for being too generous). For example, if we set the bandwidth limit of a major client to 60 MBps, the proxy server, on average, leaves 27% headroom left unused. On the other hand, if we dynamically adjust the threshold, it would require a near real-time monitoring and feedback mechanism given that bursts are short-lived (typically last ten to hundreds of milliseconds, though some extend to tens of seconds). This is undesirable in practice as accurately adjusting dynamic thresholds requires frequent interactions among all (tens of thousands) VMs employing the same proxy server cluster. Similarly, imposing stricter per-client VD caps would directly reduce the amount of sellable storage capacity, negatively impacting revenue—an unacceptable trade-off for a commercial cloud service.

RQ4: Why not migrating the hot traffic on storage ends? Alternatively, one might suggest adopting a reactive approach by migrating the hot traffic to other clusters. The rationale is that, while a burst may easily flood the current cluster, it is unlikely such a workload would push all the clusters in an availability zone to their limits. Therefore, migrating the hot traffic to *cold* clusters seems promising. Yet, in practice, the bursts are both short-lived and unpredictable, thereby making the migration—which usually takes 20 minutes—infeasible.

Key Takeaways. *In shared storage systems, load balancing spreads localized bursts cluster-wide, amplifying rather than mitigating tail-latency degradation. Static rate caps can curb bursts but waste capacity under normal load. This calls for a mechanism that protects minor clients' tail latency during overload without sacrificing cluster utilization in the common case.*

3.2 Underloaded Scenario

Symptom. Now, with the burst scenario exposed, a follow-up question ensues: do I/Os during normal (i.e., under-

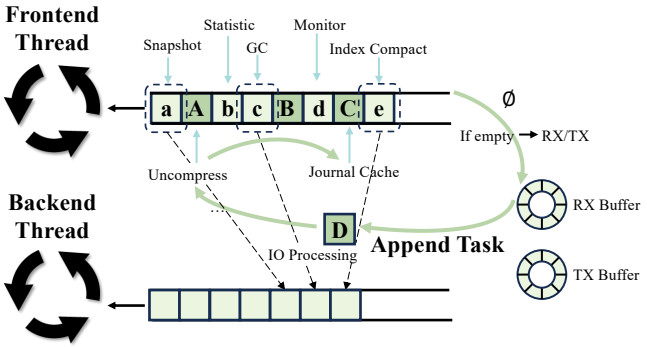


Figure 7: Event-based thread model and a thread-splitting solution.

loaded) periods also suffer abnormal tail latency? This question motivates a closer look at Figure 1. At first glance, the real-world statistics suggest not. Indeed, in stark contrast to the bursts, the tail latency metrics under normal I/O pressures are relatively low, only 1.34ms and 3.7ms on average for P99 and P99999, respectively.

However, a closer look at the breakdown of the tail latency indicates otherwise. The breakdown of the P99999 latency for read and write under normal I/O pressures in Figure 3 shows that the proxy server processing contributes significantly to the tail latency. In other words, simply because having an overall latency lower than that under the burst does not mean that the tail latency is well tamed.

Root Causes. The per-I/O tracing leads us to conclude that the fundamental event-based thread model is the culprit for the high tail latency. As shown in Figure 7, in the proxy server, each thread runs in the run-to-completion mode, and each processing is enveloped in an individual task and assigned into a task queue for FIFO scheduling. Although the proxy server is designed to serve I/O requests from clients, it also needs a large number of assistant tasks to keep the entire proxy server running healthily. For example, monitoring and statistic tasks are necessary for the operation and troubleshooting; the index compact task is necessary for keeping the block offset table concise, optimizing memory usage and processing performance. Therefore, even if there is no excessive amount of requests, such a model will postpone the processing of I/O requests owing to the execution of other I/O-unrelated tasks at the front of the task queue.

Limitations of intuitive and existing solutions. One "band-aid" solution is to assign additional computing resources, dedicating separate cores for task processing. This allocation initially showed promise by improving task processing efficiency. By diverting unrelated tasks (i.e., the tasks enclosed by the dashed box in Figure 7) from the I/O thread to other processing units, the I/O thread could complete its current loop processing more swiftly, facilitating quicker transitions to the next loop for efficient I/O handling. However, this approach has its limitations. While the majority of tasks could be offloaded successfully, some tasks remained inherently tied to the I/O processing logic, sharing critical data structures. Attempting to separate these tasks led to data contention issues, resulting in performance degradation rather than improvement.

Additionally, existing work primarily focuses on optimizing the user-level thread model [20, 26, 25], which requires significant architectural changes to the code, making it difficult to quickly address our issues. On the other hand, user-level thread scheduling introduces additional overhead compared to event-based thread models, and may also require specialized hardware support [25], making them hard to deploy at scale.

Key Takeaways. *Even in the underloaded scenario, tail latency remains problematic. In event-loop architectures, I/O-related and I/O-unrelated tasks inevitably share the same thread and are processed in batches. This batched execution delays newly arrived I/O requests even when overall utilization is low. This calls for a scheduling mechanism that minimizes the latency impact on I/O-related tasks while ensuring I/O-unrelated tasks still execute reliably.*

4 Design

4.1 Design Goals

Goal-1: practical and lightweight. The previous analysis shows that the tail latency is not well optimized under both burst and underloaded scenarios. Our earlier latency breakdowns have suggested that the hardware has also contributed significantly to the tail latency. While we notice that recent advancements in hardware (e.g., 2×200Gbps RDMA and ZNS NVMe) have offered better guarantees on tail latency, we, as a cloud vendor, can hardly depend on hardware upgrades to alleviate the tail latency. Apart from being unfavorable to the CapEx due to the large number of deployed servers, this can also incur huge engineering effort since a complete overhaul of certain parts or the entire software stack is often needed for fully utilizing the new hardware. Therefore, we aim to propose solutions that do not require extensive restructuring of the system architecture and only touch the necessary parts of the data path.

Goal-2: effective and fair. Obviously, our solutions need to be effective in reducing the tail latency in both burst and underloaded scenarios. Our discussion on the simple fixes has shown that certain methods can indeed reduce the tail latency but at the cost of introducing side effects. Again, as a cloud vendor, we emphasize the fairness and thus expect that our approaches, if not beneficial to all clients, would at least not worsen the experiences (i.e., Service Level Objectives, SLOs).

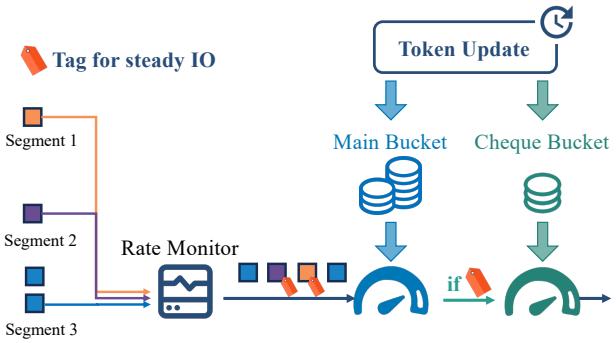


Figure 8: Burst-aware rate limiter overview.

4.2 Handling Burst Scenario

Observation. In § 3, we have shown that limiting I/O rates on major clients is suboptimal in that it would backfire on the utilization of the proxy server on average. This motivates us to revisit the field traces and leads to the following observations.

- *I/O carries different service guarantees in EBS.* When a virtual disk is created, users are provisioned with a baseline level of throughput/IOPS, which defines the SLA-bound base I/O that must be satisfied in terms of available bandwidth. Demand beyond this baseline is treated as burst I/O and is served on a best-effort basis without explicit guarantees. Under high load, our goal is to sustain throughput while protecting SLA-bound base I/O from burst interference.
- *Not all segments of the major clients under bursts are hot.* In fact, most of them are not. As aforementioned, the average CV of segments in the same VD is 47.7, revealing a weak correlation among those segments. Therefore, instead of throttling the entire client, we should operate at a finer granularity, i.e., segment-level.
- *The underlying DFS can tolerate I/O bursts, at least for a short period of time*. Recall that the bandwidth of a proxy server is limited to 1.2 GBps to avoid overloading the DFS. This strict limit is set to reserve enough resources for handling extremely high background loads (e.g., GC). However, temporarily exceeding this limit during low background loads would not cause a severe performance penalty.
- *The bursts are short-lived.* We have profiled the burst durations and found that the majority of the bursts last less than 100ms with a few exceptional cases persisting over a few seconds.

Key idea. The observations above motivate a key idea: instead of throttling hot segments from major clients during bursts, we let steady segments *overdraw* tokens, which simultaneously preserves overall resource utilization and ensures that minor clients' I/O demands are not disrupted. Moreover, by estimating the actual available resources, we can compensate the steady segments with dedicated tokens to issue more requests. Furthermore, we only perform overdrawing during the first time window of the burst. Now, our challenges boil down to three parts, as illustrated in Figure 8. First, build a segment-level monitor to identify hot segments. Second, estimate the available bandwidth to determine how much can be overdrawn. Finally, use the available bandwidth to allow steady segments to overdraw tokens, while maintaining sufficient resources for the DFS and the hot segments.

Segment-level I/O rate monitoring. To achieve a finer granularity of rate limiting, we first need to accurately monitor the I/O rate on a segment level. To achieve this, we calculate the I/O request rate of each segment under a fixed interval T (10ms in production). If the volume of requests served by a segment in the current interval exceeds the predefined threshold R, this segment is labeled as a "hot segment". Otherwise, this segment is categorized as a "steady segment". To mitigate the impact of network jitter and ensure the stability of segment categorization, the system tracks the last time a segment was marked as a "hot segment". A segment is designated as steady only if its request rate remains below R for at least N intervals ($N \times T$).

Available bandwidth estimation. Since the background workload is dynamic, we estimate the available bandwidth by modeling a proxy server as an *M/M/1* queue [21], where requests arrive at rate λ and the server processes them at rate μ. The expected I/O latency at percentile p is (full derivation in Appendix A):

$$w_p = \alpha \frac{\ln(1-p)}{\mu - \lambda} \tag{1}$$

where α is a correction factor (α=8000 in production) to account for the deviation from ideal exponential distributions. In practice, directly measuring P99999 latency requires minutes of samples, so we instead monitor P95 latency over short intervals. Since Equation (1) establishes a deterministic mapping between any percentile latency and the system load ($\mu - \lambda$), observing w_{P95} over a short window suffices to infer the current load and predict the P99999 tail latency. As shown in Figure 6, with μ=2130Mbps and different background loads (GC=300/600Mbps), Equation (1) precisely matches actual measurements. By rearranging Equation (1), the residual serving capacity can be obtained from the monitored w_p. Subtracting a safety headroom h_p yields the estimated available bandwidth:

$$\delta_p = \alpha \frac{\ln(1-p)}{w_p} - h_p \tag{2}$$

where h_p=330Mbps is a headroom that absorbs both model error and prevents unacceptably high latency.

During runtime, by monitoring the P95 latency of recent I/Os, we can directly calculate the available bandwidth δ_p for overdrawing with Equation (2).

Dual-bucket throttling. To enable the steady segments to overdraw tokens, we introduce a dual-bucket mechanism. The first bucket, called the *main bucket*, functions as usual and is shared by all segments. The size of the main bucket is equal to the total tokens issued in an interval. The second bucket, called the *cheque bucket*, is only accessible to the steady segments. The size of the cheque token bucket is dynamically set based on the detected bandwidth over a recent period.

In practice, the initialization process fills both buckets with tokens to full capacity. Under normal pressure, all segments are "steady" and consume tokens from the main bucket. Every interval, the proxy server first refills the main bucket. Then, if the cheque bucket is not full, the main bucket will give enough tokens to fulfill the cheque bucket before serving the real I/O requests.

When a burst occurs, the monitor would identify and mark the hot segments. And, the main bucket would be (presumably) quickly drained by the hot segments. Now, the steady segments can consume tokens from the cheque bucket and hot ones are thereby throttled. After the periodic interval T, the proxy server would refill the cheque bucket with $\delta_p T$, and fill the main bucket with the remaining tokens.

Now, if the burst diminishes, the hot segments would be marked as steady in the next interval. And, all things go back to normal as the steady segments can grab 100% of the tokens. Both buckets would be full again in the next interval(s). If the burst persists, the remaining hot segments would continue to consume tokens from the main bucket while the steady can draw tokens from both.

The key of dual-bucket throttling is to reduce the impact of bursts on the steady segments. Therefore, the cheque bucket allows the steady segments to get tokens even when the main bucket is depleted. Yet, such an overdraw mechanism does not last as the underlying DFS would eventually be over-saturated. Therefore, we only allow overdrawing for an interval. This mechanism helps to throttle the hot segments' I/O rate by forcing them to only consume tokens from the main bucket.

Configuration parameters. The success of dual-bucket throttling depends on two key parameters: the span of the interval and the size of the cheque bucket. First, the span of the interval should be long enough to endure most burst scenarios while short enough to avoid over-saturation of the DFS. Based on the fact that the average duration of bursts is hundreds of milliseconds, we set the interval T to be 10ms to timely react to the occurrence of bursts.

Second, the size of the cheque bucket should be set to a value that can allow the steady segments to overdraw enough tokens to avoid most tail latency spikes. Meanwhile, having an overly large cheque bucket would backfire by worsening the tail latency for the hot segments. Based on field statistics, we set the size as 20% which can also be adjusted on-the-fly.

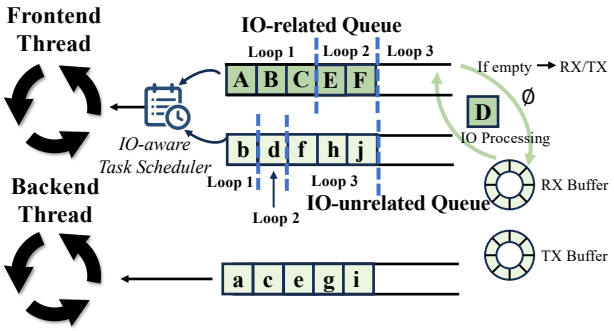


Figure 9: Priority-based task scheduling.

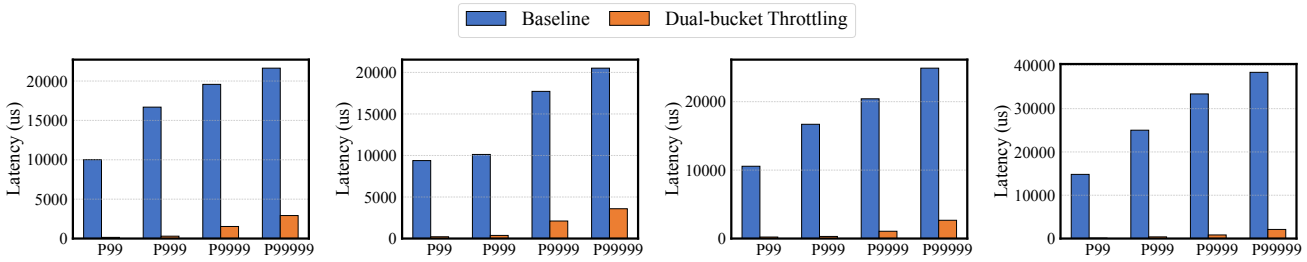
4.3 Handling Underloaded Scenario

Root causes. The latency breakdown indicates that simply shifting some tasks (e.g., GC, snapshot and index compact) has limited effectiveness. The reasons are two-fold. First, the proxy server runs the event-loop model, similar to common practices [30], in the *idle* mode and thus the pending requests in the rx buffer are not handled in a timely fashion. Second, while all client I/Os are processed in the frontend as tasks, not all tasks are directly related to I/O processing. In fact, the I/O-unrelated tasks (including monitoring, statistics and dump) can delay I/O processing. Hence, our target is to prioritize the I/O-related tasks and minimize the rx buffer waiting time.

Priority-based task scheduling. Figure 9 shows that, for the frontend event loop, we further enforce the following three policies in the task scheduling:

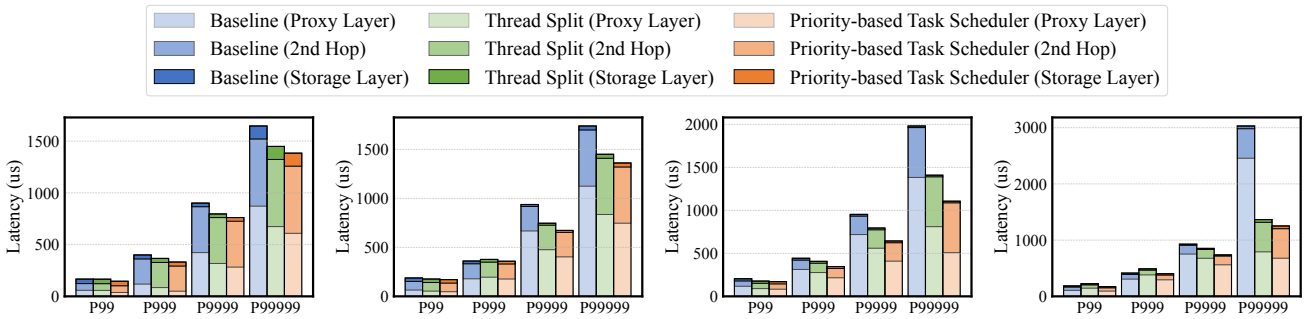
- *High priority first.* We have set up two queues for I/O-related and I/O-unrelated tasks. The I/O-related task queue includes journal cache lookup, compressing the write requests and decompressing messages in processing read requests. The I/O-unrelated task queue includes I/O thread monitoring, resource statistics, dump operations and others. During each loop, the proxy server now always starts with the execution of I/O-related tasks and will not process the I/O-unrelated tasks until all enqueued I/O-related tasks are done.
- *Fixed span for I/O-unrelated tasks.* To prevent new I/O-related tasks (pending in the rx buffer) from waiting too long, we introduce a loop timeout LT. This timeout caps only the execution of I/O-unrelated tasks: if I/O-related tasks exceed LT, their execution is not interrupted; the timeout only prevents subsequent I/O-unrelated tasks from running in the current loop.
- *Frequent rx buffer check.* To promptly receive and enqueue the I/O-related tasks for timely execution, we check the rx buffer at the beginning of every loop as long as the I/O-related queue is empty.

Preventing I/O-unrelated tasks from starving. Directly serving I/O-related tasks in a strict priority way may persistently delay the execution of I/O-unrelated tasks, caus-



(a) 100% write, 0% read. **(b)** 75% write, 25% read. **(c)** 50% write, 50% read. **(d)** 25% write, 75% read.

Figure 10: Tail latency of innocent I/O requests under burst background workload with various 4KB read/write requests.



(a) 100% write, 0% read. **(b)** 75% write, 25% read. **(c)** 50% write, 50% read. **(d)** 25% write, 75% read.

Figure 11: Tail latency of write requests under 200MBps background workload with various 4KB read/write requests.

ing starvation in high workload scenarios. Although this starvation does not interfere with I/O processing, it would lead to critical system crash risks (e.g., eternal delay of the health check would trigger the proxy service master to force this proxy server to shut down). To solve this problem, we further optimize the static LT to a dynamic value. $DLT = max\{\alpha T_{IO}, LT\}$, where T_{IO} is the processing time consumed by all I/O-related tasks in the current loop, and α is a parameter ($\alpha > 1$).

Configuration parameters. The setting of LT is a trade-off between the performance and cost. Lower LT allows I/O-related tasks to be executed more promptly, but introduces higher looping overhead at the same time. We set $LT = 10us$ in practice to achieve a sweet spot.

Through comprehensive experiments, we set $\alpha = 1.2$ in practice, which provides about 20% CPU cycles for I/O-unrelated tasks in each loop. This mechanism makes sure there are enough time slots for the processing of I/O-unrelated tasks, eliminating starvation risks. Through long-term observation in the testing cluster, no starvation of I/O-unrelated tasks is observed.

An example. Figure 9 illustrates the task processing in the frontend event loop, where upper case letters denote I/O-related tasks and lower case letters denote I/O-unrelated tasks. The trailing number indicates the processing time in microseconds. (e.g., A3 means an I/O-related task A takes 3us to finish).

At initialization, assume there are three tasks in both the I/O-related queue (A3, B3 and C3) and the I/O-unrelated queue (a3, b2 and c2). In the first loop, the proxy server would not poll the rx buffer as the I/O-related queue is not empty. Instead, the proxy server directly processes A3, B3 and C3 (total 9us). As 9us $<$ DLT=$max\{1.2\times9, 10\}$=10.8us, the server continues to process a3 before timing out at 12us. b2 and c2 are deferred to subsequent loops.

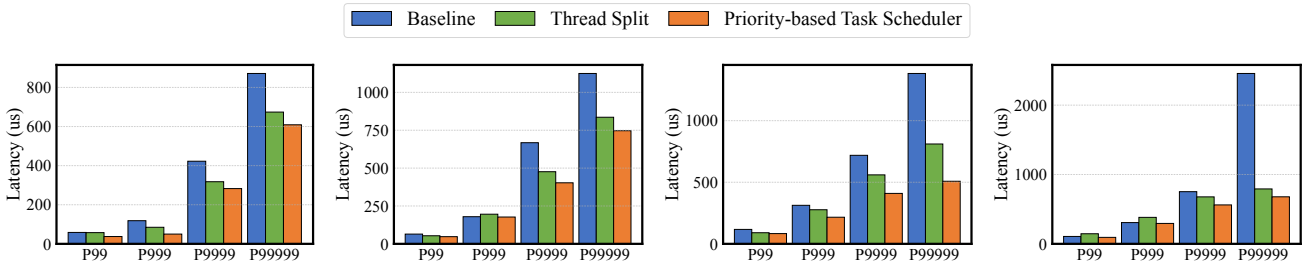
In the second loop, the I/O-related queue is empty, so the server first polls the rx buffer (D5, taking 5us), generating new I/O-related tasks E3, F3 and I/O-unrelated tasks d3, e3. After processing E3 and F3, the total execution time reaches 11us (5+3+3), exceeding the static LT. If directly employing this static LT, all I/O-unrelated tasks would miss this loop. However, with starvation prevention, DLT=$max\{1.2\times11, 10\}$=13.2us, so b2 is still executed in this loop. c2, d3 and e3 are deferred.

In the third loop, with no pending I/O tasks or buffered requests, the server processes the remaining c2, d3 and e3.

5 Evaluation

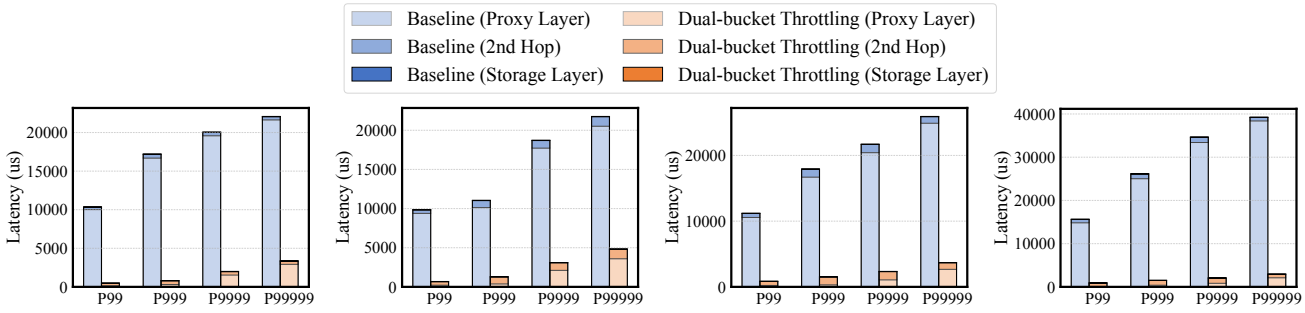
5.1 Setup

We use the same setup of our production environment. In total, we use 8 compute nodes and 16 storage nodes (running 12 proxy servers, 3 proxy metadata servers and 1 administrator node). They all are of the same hardware configuration including a 32-core Intel Xeon 2.30 GHz CPU, 192 GB memory and 2×25 Gbps NIC, except each storage node has



(a) 100% write, 0% read. **(b)** 75% write, 25% read. **(c)** 50% write, 50% read. **(d)** 25% write, 75% read.

Figure 12: Tail latency of write requests under 200MBps background workload with various 4KB read/write requests.



(a) 100% write, 0% read. **(b)** 75% write, 25% read. **(c)** 50% write, 50% read. **(d)** 25% write, 75% read.

Figure 13: Tail latency of innocent I/O requests under burst background workload with various 4KB read/write requests.

12 PCI-e Gen3 8TB enterprise-level NVMe SSDs. Moreover, each storage node runs a proxy server process. Both the 1st and 2nd hop network are 2×25 Gbps Ethernet. We use FIO to generate I/Os in the microbenchmarks and replay the traces from the production environment in the macrobenchmarks.

5.2 Microbenchmark

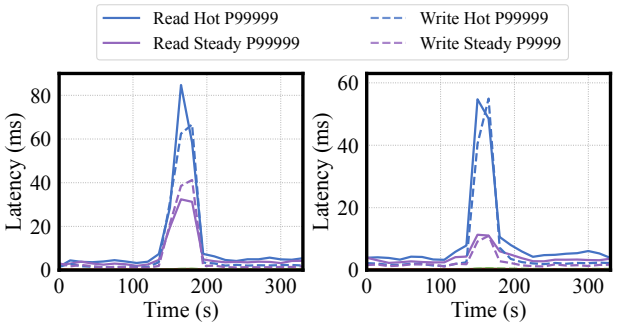
Burst scenario. We first generate a set of continuous burst scenarios. All compute nodes would generate I/O requests saturating the entire proxy layer. The I/O size and read/write component ratio differ in different experiments. Additionally, we set 20% of segments as "hot" and they contribute 80% of the overall load, simulating the typical patterns of major clients. The remaining 80% of segments are always steady and they contribute 20% of the load.

We first fix the I/O size to 4KB (the dominant I/O size in production) and vary the percentage of write/read requests (100%/0%, 75%/25%, 50%/50% and 25%/75%) to construct synthetic workloads. As shown in Figure 10, for steady segments, the P99 tail latency of the proxy server decreases by up to 96% compared with the baseline. P99999 tail latency decreases by up to 83%, indicating that dual-bucket throttling overcomes the key issue in the burst scenario. Besides, our measurements on hot segments demonstrate the tail latency of hot segments does not endure a significant increase. Figures are not included in this paper owing to the space limit.

We further test its performance under larger I/O sizes (varying from 8KB to 64KB) with mixed read/write ratio (50%/50%). Results are shown in Figure 18 in Appendix B. With dual-bucket throttling, the P99999 tail latency decreases by 92% to 97% under different I/O sizes. Notice that, with the increase in I/O request size, the performance of dual-bucket throttling is better. The reason is that larger I/O requests are prone to introduce severe bursts. With the increase in I/O size, the performance improvements introduced by the dual-bucket throttling become much more significant. This is because the impacts grow larger with increasing I/O sizes. Dual-bucket throttling can significantly prevent hot segments from influencing steady segments, resulting in better tail latency reduction.

Underloaded scenario. In this microbenchmark, all compute nodes generate steady traffic keeping the load of each proxy server at 200MBps, a typical underloaded scenario (i.e., 16.7% of traffic threshold). The workloads are composed of 4KB I/O requests with various read/write ratios. As aforementioned, a non-negligible proportion of the read I/O requests tail latency in the underloaded scenario is inevitable due to intrinsic hardware processing delay. Therefore, we focus on demonstrating the performance of write I/O requests. Figure 12 illustrates the results. In various read/write ratios, the tail latency of write requests is reduced by up to 63% and 37%, compared with the baseline and the thread split mechanism, respectively.

Similarly, we further conduct this microbenchmark with



(a) Baseline. **(b)** Dual-bucket throttling.

Figure 14: Tail latency of requests under burst scenarios in production.

various I/O sizes (8KB-64KB) with 50% to 50% read/write ratio. With priority-based task scheduling, the tail latency is reduced by up to 55% compared with only employing the thread split design, as shown in Figure 19 in Appendix B.

5.3 Storage Cluster Performance

We further evaluate the improvements under both mechanisms from the storage cluster perspective. The experiment settings in this subsection are the same as in §5.2.

Burst scenario. The performance of 4KB I/O requests is shown in Figure 13. The tail latency introduced by the entire storage cluster is split into three parts: (1) the proxy layer processing latency, (2) the second hop network latency and (3) the storage layer latency. In baseline, the proxy layer latency contributes 98% to the entire P99999 latency. The dual-bucket throttling decreases the overall P99999 latency by 85% compared with baseline.

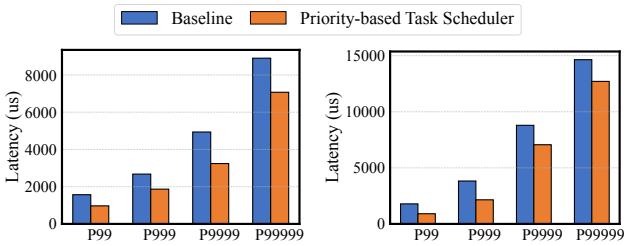
Similarly, we conduct further experiments to evaluate the performance of different I/O sizes (varying from 8KB to 64KB) with mixed read/write ratio (50%/50%). Figure 20 in Appendix B represents tail latency results under various I/O request sizes. Tail latency is reduced by 77% to 86% compared with baseline.

Underloaded scenario. The tail latency of write requests under 4KB underloaded scenario is shown in Figure 11. Priority-based task scheduler decreases the overall P99999 latency by 43% and 22% compared with baseline and the thread split mechanism, respectively. Under various I/O request sizes (varying from 8KB to 64KB), as shown in Figure 21 in Appendix B, tail latency is reduced by up to 66% and 37% compared with different alternatives, respectively.

5.4 Production Trace Performance

We record traces from in-production storage clusters, including both burst and underloaded scenarios. By replaying these traces at the per-I/O granularity, we can precisely evaluate the performance of our solution under real scenarios.

Burst scenario. We select one typical trace representing typical burst scenarios we encountered in production. The



(a) 200MBps workload trace. **(b)** 300MBps workload trace.

Figure 15: Tail latency of write requests under different underloaded scenarios in production.

trace is recorded in the cluster where major clients possess I/O-intensive tasks, introducing frequent traffic bursts in the entire cluster (called the I/O-intensive trace). As shown in Figure 14a and Figure 14b, dual-bucket throttling decreases the P99999 tail latency of innocent I/O requests by 71%.

Underloaded scenario. To validate the generality of our mechanism's effectiveness, we select two traces representing various levels of underloaded scenarios from real production. Specifically, two traces cover the average load of 200MBps and 300MBps, respectively. The results are shown in Figure 15. With priority-based task scheduling, the tail latency of write requests decreases by up to 14% and 13% compared with baseline, respectively.

5.5 Benefit of Deployment

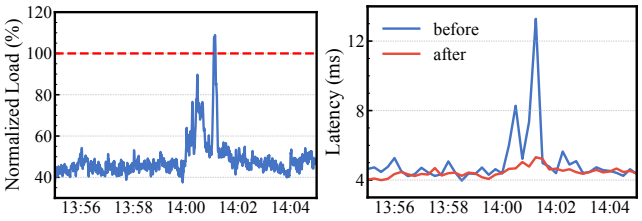
Both mechanisms in this paper have been deployed in our production, covering dozens of clusters across multiple availability zones, spanning hundreds of proxy servers with diverse hardware configurations, and have served hundreds of trillions of I/O requests. We are rolling out the full deployment of both mechanisms.

Burst scenario. Figure 16a shows a representative burst traffic pattern around 14:00 on the same weekday, where a sudden burst consistently appeared across multiple weeks. Figure 16b represents steady segment latency at this time point, showing that enabling our mechanism reduces tail latency by 59.7% for innocent I/O requests.

Underloaded scenario. We conducted an A/B test within a single cluster. Specifically, we randomly selected two proxy nodes and monitored tail latency. Figure 17 shows a week-long time series of P99999 latency. At the beginning of Thursday, we enabled our priority-based task scheduler on one node while leaving the other unchanged. Prior to activation, both nodes exhibited nearly identical latency trends; once the scheduler was enabled, a 22% reduction in P99999 latency was observed, confirming the effectiveness of the mechanism in underloaded conditions.

6 Related Work

Building cloud block store. As a cornerstone for cloud computing, there is a spate of work discussing the design



(a) Traffic pattern. **(b)** Latency.

Figure 16: Representative traffic pattern and the corresponding P99999 latency differences.

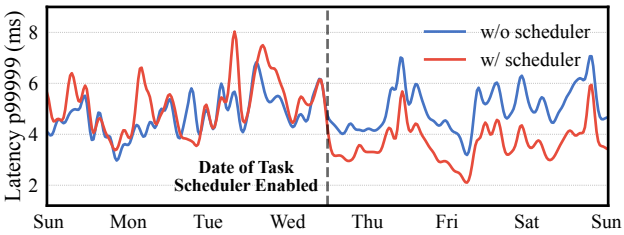


Figure 17: Week-long A/B test in an underloaded cluster. Two nodes from the same cluster were monitored for end-to-end latency; at the dashed line, the priority-based scheduler was enabled on one node, while the other remained unchanged.

and implementation of cloud block store from both industry, such as Microsoft Azure [5], Amazon EBS [2], Google Persistent Disk [4], Alibaba Elastic Block Storage [1] and academia, including Salus [29], Ursa [22], and LSVD [17]. Our paper differs from the above as we emphasize improving the tail latency of EBS, instead of designing a full-stack cloud block store. Therefore, we focus on analyzing the root causes behind the tail latency (especially under layers of load balancing), and propose corresponding solutions to alleviate the issue in a lightweight, effective and fair manner.

Improving tail latency. Tail latency under underloaded scenarios has been extensively studied across clusters [10, 13, 34] and datacenter networks [32] to cloud services [28]. Among these, the most relevant body of work lies in single-node designs [33, 18, 20, 26, 15, 25, 27, 9]. Shenango [25] addresses tail latency across applications through load balancing and dynamic core reallocation. Shinjuku [20] leverages Dune virtualization to realize fine-grained preemptive CPU scheduling at microsecond timescales. Zygos [26] reduces short-task tail latency by approximating a centralized queue, albeit at the cost of higher overhead. However, these systems rely on intrusive runtimes, virtualization, or heavy synchronization, which limit deployability. Our work in the underload scenario focuses on mitigating tail latency induced by intra-application task interleaving, while remaining simple and deployable.

In contrast, little prior work has systematically addressed tail latency under workload bursts from major clients. Our work fills this gap by analyzing burst-induced tail latency and proposing a lightweight mechanism specifically tailored for this scenario.

7 Potential Limitations

Unique workload patterns. Our work is based on the analysis of the production workloads of ALIBABA CLOUD. Unfortunately, while there is a small set of block store datasets available publicly [24, 23], we are unable to verify the existence of major clients' bursts because we are unable to correlate the workload pressure with the client's identity. However, we believe this phenomenon is not unique. First, based on the large number of users (i.e., millions of VDs deployed) and widespread major clients' bursts, we are convinced that this phenomenon is indeed representative. In addition, the root causes of such bursts are often correlated with certain events such as shopping festivals (e.g., Black Friday shopping) or new product releases where both are common in various cloud environments. As mentioned in § 1, we have released production I/O traces of classic burst scenarios to motivate future work in this direction.

Solutions are only applicable to ALIBABA CLOUD stack. In this paper, for high practicality, we intentionally build our solutions based on our EBS stack and refrain from fundamentally refactoring our stack. A possible outcome is that both the dual-bucket throttling and priority-based scheduling mechanism may not be directly applicable to other open-source or proprietary systems. However, we argue that our solutions reflect the general principles in overcoming the tail latency caused by workload bursts and inefficient handling of I/O tasks. In addition, due to the lightweight nature of our solutions, we believe that they can be easily adapted to other systems with minor engineering efforts.

8 Conclusion

In this paper, we systematically revisited a pressing issue in our cloud block store system, high tail latency. We first identify the root causes of suboptimal tail latency in production—despite the presence of multiple layers of load balancing—to be workload bursts caused by major clients and the inefficient event-loop thread model. We then propose dual-bucket throttling and priority-based scheduling, to mitigate the issue. Results in production show that the proposed mechanisms, while simple and lightweight, can significantly reduce the tail latency in varying scenarios.

Acknowledgments

We thank our shepherd, Murat Demirbas, and the anonymous reviewers for their valuable feedback. We also thank our colleagues at ALIBABA CLOUD for their helpful discussions and support. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant Nos. 62132007 and 62221003 and the Alibaba Research Intern Program. Erci Xu and Fengyuan Ren are the co-corresponding authors.

References

[1] Alibaba cloud elastic block storage devices. http://www.alibabacloud.com/help/en/ecs/user-guide/elastic-block-storage-devices.

[2] Amazon elastic block store. http://docs.aws.amazon.com/AWSEC2 /latest/UserGuide/AmazonEBS.html.

[3] Aws re:invent 2022 - keynote with peter desantis. https://www.youtube.com/watch?v=R11YgBEZzqE.

[4] Google persistent disk. https://cloud.google.com/persistent-disk.

[5] Introduction to azure managed disks. http://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview.

[6] Amazon found every 100ms of latency cost them 1 https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2023.

[7] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in Log-Structured merge Key-Value stores. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 753–766, Renton, WA, July 2019. USENIX Association.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[10] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 143–157, 2011.

[12] Li-Der Chou, Yao-Tsung Yang, Yuan-Mao Hong, Jhih-Kai Hu, and Bill Jean. A genetic-based load balancing algorithm in openflow network. In Yueh-Min Huang, Han-Chieh Chao, Der-Jiunn Deng, and James J. (Jong Hyuk) Park, editors, Advanced Technologies, Embedded and Multimedia for Human-centric Computing, pages 411–417, Dordrecht, 2014. Springer Netherlands.

[13] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 79–94, Boston, MA, February 2019. USENIX Association.

[14] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In Nsdi, volume 16, pages 523–535, 2016.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 281–297. USENIX Association, November 2020.

[16] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 519–533. USENIX Association, April 2021.

[17] Mohammad Hossein Hajkazemi, Vojtech Aschenbrenner, Mania Abdi, Emine Ugur Kaynar, Amin Mossayebzadeh, Orran Krieger, and Peter Desnoyers. Beating the i/o bottleneck: a case for log-structured virtual disks. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 628–643, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Jun He, Duy Nguyen, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Reducing file system tail latencies with chopper. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 119–133, Santa Clara, CA, February 2015. USENIX Association.

[19] EunYoung Jeong, Shinae Wood, Muhammad Jamshed,

Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 489–502, Seattle, WA, April 2014. USENIX Association.

[20] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazieres, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345–360, Boston, MA, February 2019. USENIX Association.

[21] Leonard Kleinrock. Queueing Systems, Volume 1: Theory. John Wiley & Sons, 1975.

[22] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In 2020 IEEE International Symposium on Workload Characterization (IISWC), pages 37–47, 2020.

[24] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical power management for enterprise storage. In 6th USENIX Conference on File and Storage Technologies (FAST 08), San Jose, CA, February 2008. USENIX Association.

[25] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 361–378, Boston, MA, February 2019. USENIX Association.

[26] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[28] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 513–527, Oakland, CA, May 2015. USENIX Association.

[29] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the salus scalable block store. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 357–370, Lombard, IL, April 2013. USENIX Association.

[30] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. What's the story in EBS glory: Evolutions and lessons in building cloud block store. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 277–291, Santa Clara, CA, February 2024. USENIX Association.

[32] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. Scalable tail latency estimation for data center networks. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 685–702, Boston, MA, April 2023. USENIX Association.

[33] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event scheduling for optimizing tail latency in a traditional OS kernel. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 759–772. USENIX Association, July 2021.

[34] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 1225–1240. USENIX Association, November 2020.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A M/M/1 Model Derivation

The process of a proxy server can be abstracted as a single-server queueing model [21]. In the *M/M/1* model, requests arrive at rate λ (following a Poisson process) and the server processes them at rate μ (exponentially distributed). The total I/O latency W (including both queuing time and service time) follows an Exponential distribution. The *M/M/1* queue has the following characteristics:

- The probability density function (PDF) of service time (i.e., the processing latency of a single I/O in Proxy Layer and Storage Layer) is:

$$f_{\mathrm{s}}(t) = \mu e^{-\mu t}, \quad t \geq 0 \tag{3}$$

- The PDF of inter-arrival time of two adjacent I/Os is:

$$f_{\mathrm{a}}(t) = \lambda e^{-\lambda t}, \quad t \geq 0 \tag{4}$$

The overall waiting time (i.e., I/O latency) w in the system is the time from I/O arrival to completion, including both waiting in the queue and service time. The PDF for the I/O latency $f_W(w)$ can be calculated as:

$$f_W(w) = \lambda e^{-\lambda w}\left(1 - \frac{\lambda}{\mu}\right) e^{-(\mu-\lambda)w}, \quad w \geq 0 \tag{5}$$

The cumulative distribution function (CDF) $F_W(w)$ of the I/O latency is:

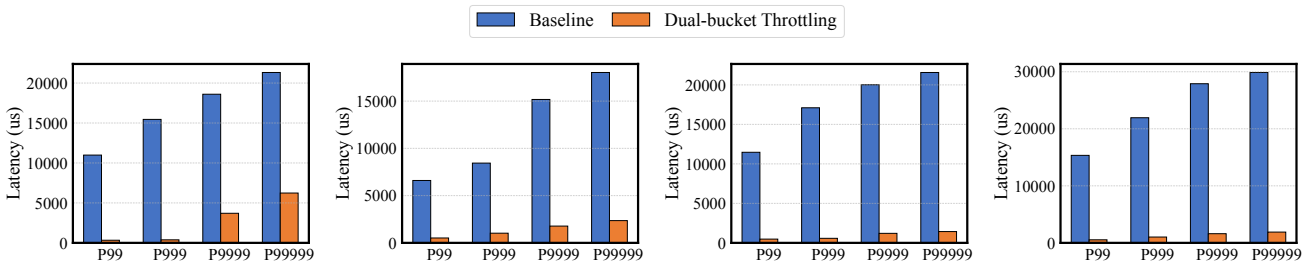
$$F_W(w) = 1 - e^{-(\mu-\lambda)w} \tag{6}$$

As a result, the expected I/O latency for any percentile (p) can be calculated as:

$$w_p = \alpha \frac{\ln(1-p)}{\mu - \lambda} \tag{7}$$

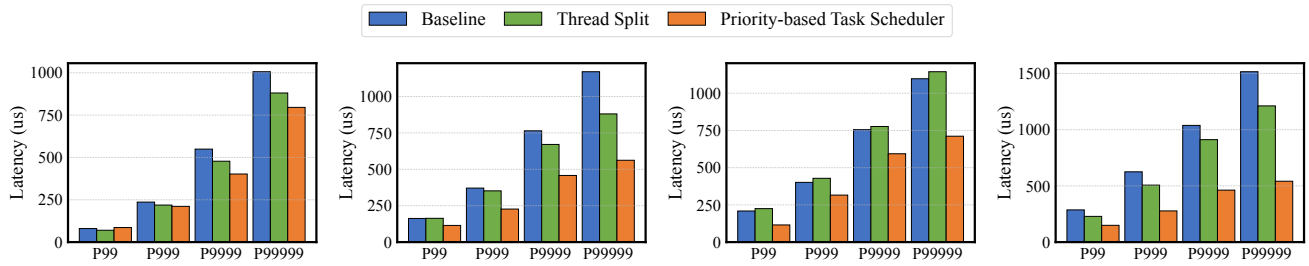
where α is a correction factor to account for the deviation from ideal exponential distributions in practice.

B Supplementary evaluations



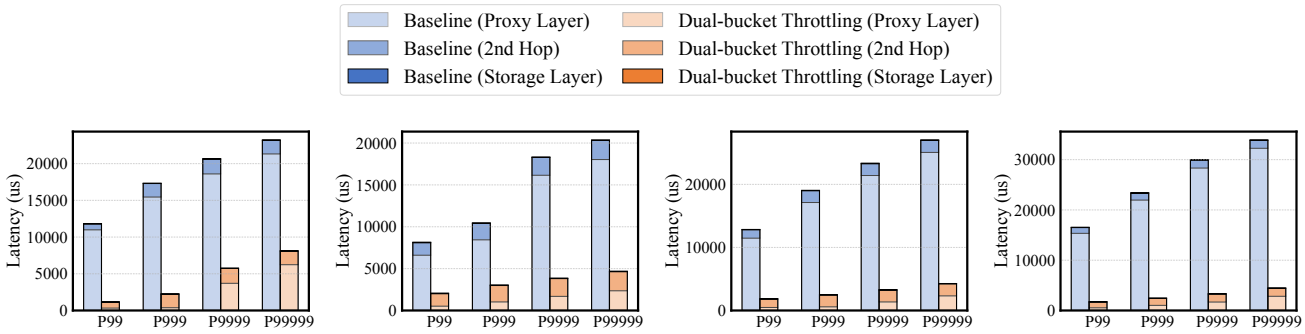
(a) 8KB 50% write, 50% read. **(b)** 16KB 50% write, 50% read. **(c)** 32KB 50% write, 50% read. **(d)** 64KB 50% write, 50% read.

Figure 18: Tail latency of innocent I/O requests under burst background workload with various 8-64KB read/write requests.



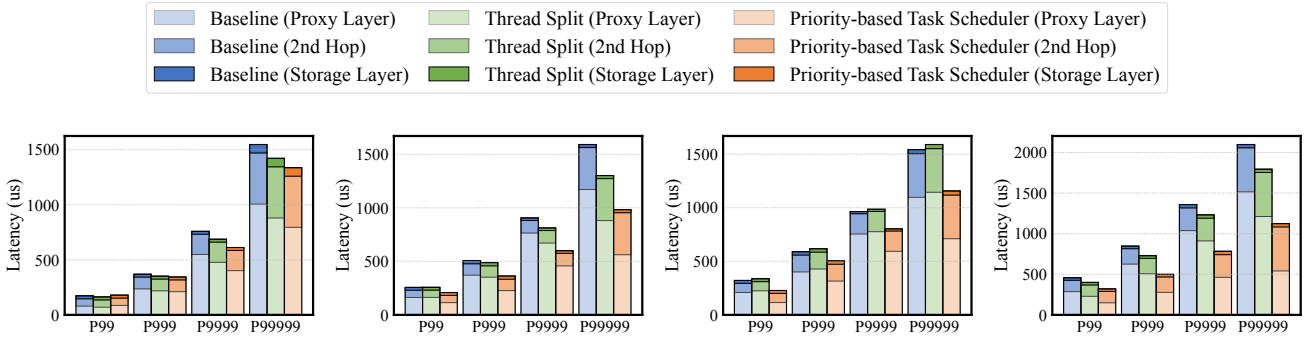
(a) 8KB 50% write, 50% read. **(b)** 16KB 50% write, 50% read. **(c)** 32KB 50% write, 50% read. **(d)** 64KB 50% write, 50% read.

Figure 19: Tail latency of write requests under 200MBps background workload with various 8-64KB read/write requests.



(a) 8KB 50% write, 50% read. **(b)** 16KB 50% write, 50% read. **(c)** 32KB 50% write, 50% read. **(d)** 64KB 50% write, 50% read.

Figure 20: Tail latency of innocent I/O requests under burst background workload with various 8-64KB read/write requests.



(a) 8KB 50% write, 50% read. **(b)** 16KB 50% write, 50% read. **(c)** 32KB 50% write, 50% read. **(d)** 64KB 50% write, 50% read.

Figure 21: Tail latency of write requests under 200MBps background workload with various 8-64KB read/write requests.