Hey Hey, My My, Skewness Is Here to Stay: Challenges and Opportunities in Cloud Block Store Traffic

Haonan Wu[1], Erci Xu[1*], Ligang Wang[2], Yuandong Hong[2], Changsheng Niu[2], Bo Shi[2], Lingjun Zhu[2], Jinnian He[2], Dong Wu[2], Weidong Zhang[2], Qiuping Wang[2], Changhong Wang[2], Xinqi Chen[1], Guangtao Xue[1, 3], Yi-Chao Chen[1], Dian Ding[1]

[1] Shanghai Jiao Tong University, Shanghai, China
[2] Alibaba Cloud, Hangzhou, China
[3] Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3, Shanghai, China
{haonanwu,chenxq66,yichao,gt_xue}@sjtu.edu.cn,jostep90@gmail.com
{leo.wlg,yuandong.hyd,changsheng.niu,shibo.sb,lingjun.zlj}@alibaba-inc.com
{jinnian.hjn,haoyu.wd,zhangweidong.zwd,wangqiuping.wqp,wangchanghong.wch}@alibaba-inc.com

Abstract

Elastic Block Storage (EBS) has a pivotal role in modern data center infrastructure, providing reliable, high-performance and flexible block storage service to users. In Alibaba Cloud, EBS is the most widely used service and has been supporting the operation of millions of virtual disks. However, even with layers of load balancing and caching, we still observe significant traffic skewness across the EBS stack. This motivates us to comprehensively investigate symptoms and root causes behind the traffic patterns and, more importantly, explore the fixes for the identified issues.

In this paper, we collect 310 million IO traces from approximately 60k virtual machines and 140k virtual disks deployed in our EBS. Based on extensive statistical analysis, we examine the traffic skewness across multiple components of the EBS stack. We identify four typical symptoms related to the IO virtualization framework, traffic throttle, storage cluster management and cache. For each symptom, we further explore the potential solutions along with the challenges.

CCS Concepts: • **General and reference** → *Measurement*; • **Computer systems organization** → **Cloud computing**.

Keywords: Compute-Storage Disaggregation, Elastic Block Storage, Load Balancing, Cache

*Corresponding author: Erci Xu (jostep90@gmail.com).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696068

ACM Reference Format:
Haonan Wu[1], Erci Xu[1*], Ligang Wang[2], Yuandong Hong[2], Changsheng Niu[2], Bo Shi[2], Lingjun Zhu[2], Jinnian He[2], Dong Wu[2], Weidong Zhang[2], Qiuping Wang[2], Changhong Wang[2], Xinqi Chen[1], Guangtao Xue[1, 3], Yi-Chao Chen[1], Dian Ding[1]. 2025. Hey Hey, My My, Skewness Is Here to Stay: Challenges and Opportunities in Cloud Block Store Traffic. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696068

1 Introduction

Elastic Block Storage (EBS) [1, 2, 4, 5, 12] is a fundamental component in cloud infrastructure. EBS provides reliable, scalable and high-performance storage in the form of virtual block device (a.k.a., virtual disk, VD). A prevalent architectural design in modern EBS systems is "compute-to-storage disaggregation", allowing independent design, deployment, and scaling of compute and storage resources.

In Alibaba Cloud, the EBS system also follows the compute-storage disaggregation design. Specifically, when a user issues an IO request to a VD, the request first reaches the hypervisor of the compute node that hosts the VM. The hypervisor then forwards the request to the storage cluster using a Remote Procedure Call (RPC) framework on the intra-DC network. Inside the storage cluster, an incoming request would go through two layers: the block server in the forwarding layer for address translation (i.e., from VD's LBA to offset in a file) and the chunk server in the distributed file system layer for finally persisting the data to the SSDs.

Along the path, EBS has employed multiple levels of load balancing and caching for high performance and availability. For example, the hypervisor attaches IO requests to worker threads in a round-robin manner. As an another example, EBS splits the address space of VD into fixed size segments and allocate them to different block servers to avoid hotspots. Unfortunately, in the field, we have still been observing severe workload skewness. For example, the hottest worker

of the hypervisor experiences 2.6 times the workload of the coldest worker on average.

While there are a spate of works studying the traffic of EBS-like systems, their findings can be limited or inconclusive for two reasons. First, their scale is small or specific as they only studied up to 6k VDs [33, 35] or focused on workloads driven by certain applications (e.g., virtual desktop in [33] and high-performance computing in [31]). Second, the analysis are not comprehensive, either focusing on the compute cluster [33] or the forwarding layer [34, 37].

In this paper, we examine the IO traffic of EBS at scale from an end-to-end perspective. With traffic from approximately 60k VMs and 140k VDs monitored, we have gathered two main datasets for the study: *trace* and *metric*. The trace dataset is a per-IO record that tracks the latency of VD IO across various EBS components. Besides, the trace includes block layer-related information, such as opcode, IO size, and offset. Due to the large volume of the trace, we randomly sampled the IO requests at a rate of 1/3200. The second dataset, called metric, is a statistical aggregation of the IO traces. It calculates the sum/average of throughput, IOPS and IO sizes of all the requests (i.e., no downsampling) at different levels of EBS (e.g., block server of forwarding layer) on a second-level granularity.

The two datasets enable us to conduct a thorough and in-depth revisit on the EBS traffic patterns and subsequently discuss the shortcomings of the existing load balancing and cache design. For each identified issue, we qualitatively present the symptoms, analyze the root causes, and propose potential solutions. More importantly, we discuss the challenges as why simple fixes and known techniques may not work or only partially address the problem. We have made the dataset publicly available[1]. We hope this work can motivate future research on these open-ended topics. The findings of this paper include the followings:

- The baseline statistics (§3) reveal that both spatial and temporal skewness are prevalent across the EBS stack. Moreover, contrary to prior work [33], skewness is much worse than previous assumptions and exhibits stronger read skewness. This motivates us to rethink and discuss the design of load balancer on both the compute and storage ends.
- We discover the load balancer in the hypervisor (§4) is inefficient as traffic tends to concentrate on a small number of VM's IO queues. The root cause is the round-robin based binding between IO queue and worker thread causes a few worker threads to handles the majority of the traffic. We further evaluate the limitations of the proposed rebind-based balancer in our scenario and discuss the potentials of a hardware-offloading solution.
- We study the hypervisor's throttling mechanism upon workload bursts (§5) and why the existing approach on limiting the VD traffic with a hard threshold is suboptimal to bandwidth utilization. We then propose a more flexible throttle model to better harvest VM's bandwidth during bursts.
- We observe the migration and reallocation of VD's traffic at the forwarding layer of the storage cluster can be unnecessarily frequent (§6). The root cause is the heuristic balancer design to level the workload of block servers. We then discuss and explore a more accurate heuristic based on the traffic prediction.
- We find that even with page cache in the VM, there are still significant LBA hotspots (§7). Based on this, we move on to explore alternative opportunities by deploying cache along the EBS stack, and discuss the goals and insights for better cache algorithms.

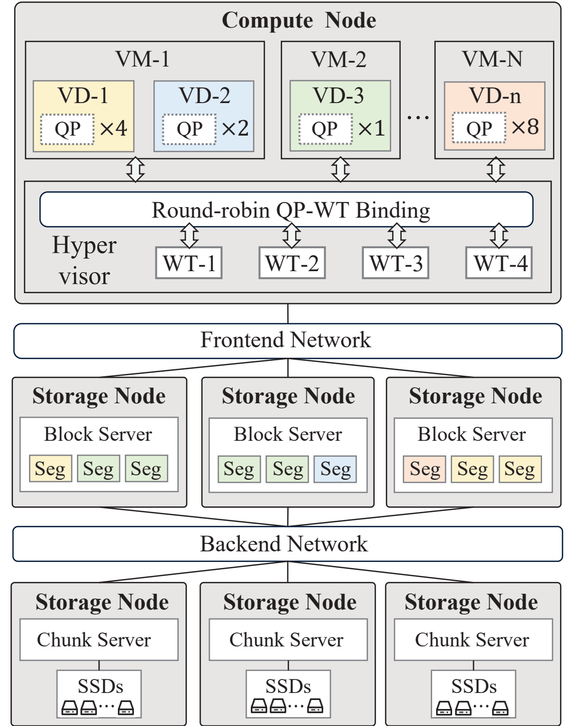


Figure 1. Architecture of our Elastic Block Storage. VM: *Virtual Machine*; **VD**: *Virtual Disk*; **QP**: *IO Queue Pair of the virtual disk*; **WT**: *Worker Thread of the hypervisor*; **Seg**: *Segment.*

The rest of this paper is organized as follows: §2 introduces our architecture and dataset; §3 presents the basic statistics of the dataset; §4 to §7 discuss the challenges and opportunities posed by skewed traffic on hypervisor load balancing and throttle model, storage cluster load balancing, and EBS cache; §8 discusses the limitation of our study; §9 discusses related work, and §10 concludes the paper.

[1]We have release our dataset at https://tianchi.aliyun.com/dataset/185310.

2 Architecture & Dataset

2.1 Elastic Block Store (EBS) Architecture

One archetypical characteristic of today's cloud storage—including block store, object store and table store from various vendors (e.g., Alibaba Cloud [1], Amazon AWS [2] and Microsoft Azure [4])—is the compute-storage disaggregation. In a nutshell, we can view this architecture as two sets of clusters—namely compute and storage—connected by a high-speed intra-DC network. Our EBS follows this philosophy.

Figure 1 illustrates the high-level architecture. In the compute cluster, a Compute Node (CN) can host multiple Virtual Machines (VMs) where each can further mount several Virtual Disks (VDs). A VD, determined by user's subscription levels, can have one or multiple (up to 8) IO Queue Pairs (QPs). The QP of VD is conceptually identical to the IO queue pair in NVMe SSD [16], except the QP is virtualized by the hypervisor. The QP of VD offers IO isolation and parallelism to the VM, which is supported by NVMe virtualization frameworks (e.g., SPDK [9] and QEMU [17]).

The hypervisor runs multiple Worker Threads (WTs) to process the IO requests from the QP. Each WT is bound to a specific CPU core and operates in a polling-mode for ultra-low latency. The IO request issued by the VM is first delivered to the VD's QP. Then, the WT encapsulates the IO into a RPC request (Remote Process Call) and forwards it to the storage cluster via the frontend network. Here, a round-robin based load balancer is used for the inter-WT balancing.

Our storage cluster follows the design as [55]. Each storage node runs a process called BlockServer (BS). BS works as a proxy that translates the semantics of block IO into file APIs and further persists/retrieves data to/from the underlying HDFS-like file system. Specifically, in EBS, a VD's Logical Block Address (LBA) is partitioned into 32 GiB-size segments (Segs) to be managed by multiple BSs. Segment is actually a kind of striping for load balancing and fault tolerance, which is also utilized by other distributed storage systems like HDFS [10], Ceph [50], and Amazon S3 [6].

Each BS stores the segment as a file via the node-level storage engine, called ChunkServer (CS). The CS is equipped with multiple SSDs for persistence. Due to the append-only nature, BS also needs to periodically perform garbage collection for space reclaiming. While BS and CS can be co-located on the same storage node (SN), we do not enforce locality-based data placement. Instead, the BS and CS communicate through the backend network using RDMA [25].

2.2 Load Balancing and Cache in the EBS

The EBS stack includes multiple components such as the end-host (i.e., compute and storage node) and the fabric (i.e., frontend and backend network). As shown in Figure 1, VD IO goes through several software and hardware components before being persisted to the disks. We refer to these components collectively as the EBS stack. Along this path, we have designed several mechanisms for load balancing and caching including:

Inter-WT load balancer. Our hypervisor shares the same design philosophy of the SPDK-vHost [54], a framework designed for high-performance NVMe virtualization. Specifically, VD's queue pair (QP) is statically bound to only one worker thread (WT), which is called the *single-WT hosting*. Single-WT hosting can avoid the multi-thread lock when accessing the same QP and reduce the CPU cache miss. Under single-WT hosting, round-robin emerges as a common choice for load balancing and multi-tenant fairness. Note that the WT works in polling-mode and processes the IO request of the bound QPs in turn.

Inter-BS load balancer. The inter-BS (BlockServer) load balancer is the key to ensure the performance and reliability of the storage cluster. Our balancer shares the same design philosophy of Ceph's metadata server balancer [50, 51] and HDFS Balancer [10]. Specifically, it operates in a periodical fashion. For each period, the balancer checks whether there exists BSs whose traffic significantly exceeds the average level of the cluster (i.e., exporters), and triggers the migration accordingly. For each exporter, several segments are migrated to the BS with the lowest traffic in the current period (i.e., importer). Besides, our balancer performs migration based solely on the write traffic considering the write-dominance in the EBS system [33–35, 60]. We enclose the pseudo-code of the balancer in Appendix A.

Cache. First, the native page cache in the VM's operating system can cache part of the IO requests. Second, we have designed a data prefetching to cache the read requests. Specifically, the BS detects whether there exists continuous large block reads on a per-segment basis, and if so, the BS will load the subsequent data from the CS (ChunkServer) into the local memory. Data prefetching can only accelerate read requests and saves the latency of accessing the CS.

2.3 Datasets

To monitor the EBS operation, we have developed a tracing tool called *DiTing*. *DiTing*, similar to Google's Dapper [43], traces IO information across various EBS components. With the help of *DiTing*, we collect three datasets for this study: *trace data*, *metric data* and *specification data*.

Trace data. Similar to Dapper [43], *DiTing* collects the IO traces [2] at a sampling rate of 1/3200. Each IO is identified by a unique *TraceID* and contains the following three types of information. First, the trace records the IO-related information, including op-code (R/W), IO size and LBA offset. Second, the trace records the EBS stack-related information, including the compute node, VM, VD, WT, storage node, and

[2] A trace refers to a block IO issued by a VD. In this paper, we use trace and IO interchangeably.

Table 1. Format of the metric data. The two values separated by a slash ('/') refer to statistical value for read and write traffic, respectively.

Domain	Physical Information			User Information			Record Unit		Metric	
Compute	**Timestamp**	**Cluster**	**Node**	**UserID**	**VM**	**VD**	**WT**	**QP**	**Throughput**	**IOPS**
	17:23:15	Clus-A	10.0.0.1	John	A	A-a	1	A-a-1	35 / 14	3200 / 9000
	17:23:15	Clus-A	10.0.0.1	John	A	A-a	2	A-a-2	20 / 0	80 / 0
Storage	**Timestamp**	**Cluster**	**Node**	**UserID**	**VM**	**VD**	**Segment**		**Throughput**	**IOPS**
	17:23:15	Clus-C	11.0.0.1	John	A	A-a	A-a-3		21 / 13	3000 / 8000
	17:23:15	Clus-C	11.0.0.2	John	A	A-a	A-a-4		14 / 1	200 / 1000

Table 2. High-level summary of the collected datasets.

Statistic	Value
Total number of user / VM / VD	10k / 60k / 140k
Median / Max number of VM per user	1 / 9879
Median / Max number of VD per user	2 / 59225
Total write / read traffic (PiB)	21.7 / 6.5
Total write / read trace (million)	247.1 / 56.9

segment that a IO passes through; Third, the trace records the latency of the IO across five major components of the EBS stack, including compute node, frontend network, BlockServer, backend network, and ChunkServer.

Metric data. Since sampled trace only provide a limited perspective of the EBS IO traffic, *DiTing* calculates second-level statistics on the full-scale IOs (i.e., all requests). The format of metric data is shown in Table 1. Specifically, the metric data includes two sub-datasets, corresponding to the compute and the storage domain. For the compute domain, the metric data records the throughput and IOPS for each QP-WT pair. Besides, the data includes VD, VM and tenant to which the QP belongs (i.e., user information), as well as the compute node and cluster to which the WT belongs (i.e., physical information). The storage domain has a similar format, except that the storage domain records the metric for each segment.

Specification data. As a supplementary, this dataset records the user's VM configurations, including the specifications of each mounted VD (e.g., capacity, IOPS and throughput limits), as well as the compute nodes hosting the VM. Moreover, the dataset also records the applications running on each VM. It should be noted that we are prohibited accessing the users' VM directly. However, we can infer the applications by comparing the user's IO to the IO of the sample VMs, i.e, the VMs created by ourselves that run standard applications.

3 Dataset Overview

3.1 Baseline Statistics

We begin by presenting the high-level summary of the collected datasets in Table 2. The dataset is collected on a 12-hour daytime (from 10:00 to 22:00). Together, we obtain 310 million traces from 10k users, 60k VMs, and 140k VDs.

Then, we analyze the spatial and temporal distribution of EBS traffic. In Table 3, we aggregate the metric data at the level of various EBS components, including compute node (CN), virtual machine (VM), storage node (SN) and segment (Seg). To quantify spatial distribution, we use the Cumulative Contribution Rate (CCR) [33]. Taking CN level as an example, "1%-CCR" refers to the proportion of traffic contributed by the top 1% of computing nodes to total traffic.

Regarding the temporal distribution, we adopt the Peak-to-Average ratio (P2A) [33]. The P2A is the ratio of the maximum to the average of the traffic and can reflect the extent of traffic bursts. We measure the P2A for each sample at a specific aggregation level. In Table 3, we use 50%tile to describe the samples' P2A distribution. The 50%ile here is to maintain consistency with [33]. We perform the same analysis on data from three data centers (DCs) to validate the generality of our observations.

3.2 High-level Observations

Observation 1. *EBS traffic exhibits more severe spatio-temporal skewness than the earlier studies.* For the spatial skewness, we observe extreme CCR values, indicating that a small fraction of users contribute majority of storage traffic. For example, 1% of VMs contribute 75.4% and 42.6% of total read and write traffic in DC-3. Even though DC-2 shows the lowest VM skewness, its 1%-CCR is still double than the previous finding (i.e., 1% of VMs generate 16.6% of total traffic [33]).

In terms of temporal skewness, we observe that both read and write traffic exhibit strong volatility. For instance, the 50%ile P2A of VM's read and write traffic can reaches 30649.0 and 1094.5, which is 13619.5.8× and 173.8× to the results in [33]. Additionally, compared to the average P2A of 2.1 reported in [35], our results also report much higher values.

Observation 2. *The spatio-temporal skewness of read traffic is significantly higher than the write.* For DC-3, 1% of the VMs contribute 75.4% of the read traffic but 42.6% of the write traffic. And, the temporal fluctuation of VM's read traffic is also prominent. The 50%ile P2A of the read traffic reaches 15813.1 which is 24.1× to the write. This finding holds across all three data centers. The read-write skewness is diametrically opposed to the finding in [33] where write traffic fluctuates more violent. The P2A of write and read is 6.29 and 2.55 as reported.

Table 3. Baseline statistics of the metric data. CN: *Compute Node*; **SN**: *Storage Node*; **Seg**: *Segment*; **CCR**: *Cumulative Contribution Rate*; **P2A**: *Peak-to-Average Ratio*. The two values separated by a slash ('/') refer to statistical value for read and write traffic, respectively.

Agg. level	DC-1			DC-2			DC-3		
	1%-CCR	20%-CCR	50%ile P2A	1%-CCR	20%-CCR	50%ile P2A	1%-CCR	20%-CCR	50%ile P2A
CN	14.3 / 8.7	90.4 / 85.3	4970.9 / 804.0	28.4 / 2.4	88.4 / 36.7	10343.9 / 417.0	32.6 / 16.5	94.4 / 71.8	5134.2 / 428.7
VM	48.9 / 39.2	99.9 / 99.6	30649.0 / 1094.5	32.9 / 3.1	86.6 / 46.2	9141.7 / 412.2	75.4 / 42.6	99.4 / 94.2	15813.1 / 657.1
SN	2.4 / 1.8	38.1 / 29.9	6.6 / 2.5	3.9 / 4.4	40.0 / 38.3	3.9 / 3.1	2.8 / 1.9	33.9 / 28.2	3.9 / 2.9
Seg	40.0 / 26.7	99.4 / 93.7	97.2 / 30.0	54.7 / 47.1	99.5 / 96.4	125.6 / 100.0	55.9 / 38.6	99.8 / 96.0	100.0 / 47.1

Table 4. Skewness statistics by types of VM application.

App.	1%-CCR	20%-CCR	Traffic share (%)
BigData	10.6 / 11.4	86.0 / 79.8	37.4 / 39.6
WebApp	40.1 / 22.6	98.6 / 99.7	1.3 / 8.0
Middleware	45.5 / 28.7	93.0 / 87.9	17.0 / 20.5
File system	47.5 / 77.9	99.8 / 98.7	1.7 / 0.4
Database	51.9 / 41.5	92.3 / 85.8	23.4 / 15.7
App in Docker	60.0 / 40.7	91.4 / 81.6	19.2 / 15.6

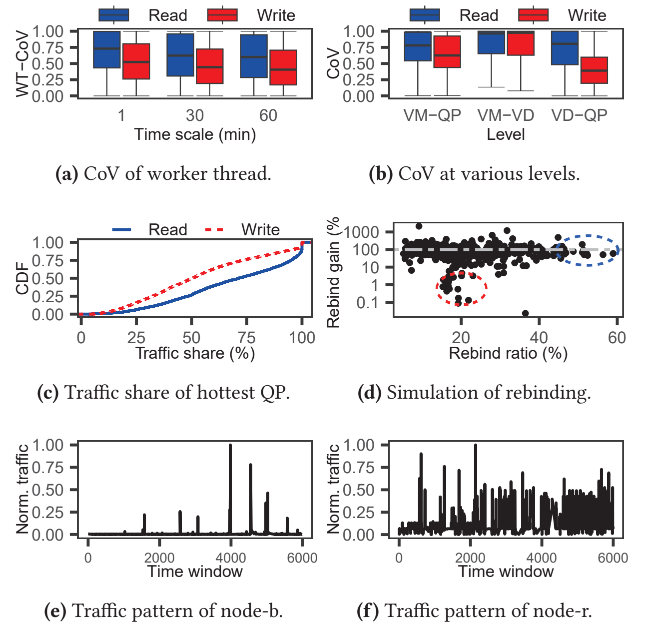
Summary. We believe that the significant divergence from the previous findings [33, 35] stems from the scale and coverage of the dataset. Lee et al. [33] conduct a similar study on the storage traffic. However, They collect data from only 262 VMs which are dedicated to Windows virtual desktop with limited applications (e.g., mailer and Microsoft Office). The VMs in our dataset run a variety of operating systems and applications. Similarly, Li et al. [35] performed analysis on two EBS production systems, including Alibaba [34] and Tencent [56]. Even if Li et al. [35] have expanded up to thousands VDs, the covered applications are still limited considering the hyper-scale of the production system. For example, the largest tenant in our dataset owns about 10k VMs and 59k VDs.

To verify this, we further analyze the traffic skewness by types of applications. As mentioned in §2.3, the specification data includes the inferred application for each VM. Therefore, similar to Table 3, we calculate the skewness statistics at the VM-level by application types. Table 4 shows the 1% and 20%-CCR, as well as the traffic share of the six types of application.

We find that, for both read and write, *BigData* [3] showes the highest traffic share, but exhibit the least skewness (1%-CCR at 10.6% / 11.4%). In contrast, *Docker* demonstrates the most severe skewness, with 1% of VMs contributing 60.0% and 40.7% of the read and write traffic, respectively. Thus, we can conclude that the skewness varies significantly across different applications, a detail missed by previous studies.

These observations motivate us to further investigate the efficiency of load balancing and cache mechanisms described in §2.2. Considering the skewness of VM traffic, we analyze the three-tier traffic distribution of "VM-VD-QP" within the compute node, and provide design insights for the hypervisor on load balancing (§4) and traffic throttle (§5); Then, we shift our focus on the skewness of segment traffic. We revisit the design of the inter-BS load balancer and summarize several design insights in §6; Finally, in §7, we measure the skewness at the VD's LBA level. More importantly, we discuss the design trade-offs of deploying cache on the EBS stack and the cache algorithm.

[3] BigData includes applications such as HBase, MapReduce, Tensorflow, and other big data related applications. See details in Appendix D.



(a) CoV of worker thread. **(b)** CoV at various levels.

(c) Traffic share of hottest QP. **(d)** Simulation of rebinding.

(e) Traffic pattern of node-b. **(f)** Traffic pattern of node-r.

Figure 2. Load Balancing in the Hypervisor (§4). (a) The WT-CoV of measured in various time scale; **(b)** CoV at three aggregation levels. **(c)** Traffic contribution of the hottest QP; **(d)** The simulation results of rebinding; **(e)** / **(f)** The traffic of the hottest WT for node-b / node-r.

4 Load Balancing in the Hypervisor

4.1 Symptom

The traffic is highly skewed across the worker threads under the round-robin-based load balancer.

As mentioned in §2.2, the queue pairs (QPs) are bound to worker threads (WTs) in a round-robin manner to achieve load balancing. We use the normalized Coefficient of Variation (CoV) [19, 29] to quantify the traffic skewness of WTs denoted as WT-CoV. The CoV ranges in $(0, 1]$, with larger values indicating a greater degree of skewness.

We calculate the WT-CoV of read and write traffic for each compute node (CN) under various time scales (e.g., 1, 30 and 60 minutes). For instance, at 1-minute scale, we randomly extract 10% of the 1-minute time windows from the total observation time and calculate the WT-CoV. The results in Figure 2(a) show that the traffic of WT is highly skewed under various time scales. The median read and write WT-CoV are 0.7 and 0.5, respectively, at the 1-minute scale. Moreover, the skewness of read traffic is significantly worse than write traffic. For the compute nodes with four WTs, when the CoV reaches 0.2, on average, the traffic share of the hottest WT is 36.3%, which is 2.6× to the coldest WT.

4.2 Root Cause

Then, we deeply analyze the traffic distribution inside the compute node, and categorize the nodes into three types based on the skewness root cause:

Type I - Idle WTs. In this case, the total number of QPs is smaller than that of the WTs. Thus, at least one WT remains idle. Further, we find that 60.1% of *Type I* nodes are bare-metal nodes with only one VM hosted. This implies that these bare-metal VMs may not have high storage requirements, e.g, only mount VDs with few QPs. The remaining 39.9% of *Type I* are non-bare-metal, i.e, the node hosts multiple VMs. Even so, we still observe idle WTs. In summary, the unexpected low storage demands of users lead to the *Type I* skewness.

As described in §3.2, the skewed VM traffic indicates that the traffic distribution is largely determined by the hottest VM. In our dataset, the average traffic share of the hottest VM is 86.4% and 75.0%, for read and write respectively. Thus, we focus on the hottest VM to further break down the non-*Type I* nodes to explain why the WT traffic is still skewed when the number of QPs is larger than the WTs.

Type II - Hottest VM with single QP. For the *Type II* nodes, the hottest VM has only one VD mounted, and the VD owns only one QP. Due to the one-on-one binding between QP and WT, the corresponding WT takes all the traffic from the hottest VM, leading to an inevitable hotspot. For *Type II* nodes with four WTs, the average read and write traffic share of the hottest WT is 83.6% and 69.8%, respectively, but the ideal traffic share for each WT should only be 25%. The root cause of *Type II* skewness lies in the fact that within a compute node, a VM with only one QP is highly active, leading to the corresponding WT becoming overloaded.

Type III - Hottest VM with multiple QPs. We further analyze how the VM access multiple QPs by measuring the "VM-to-QP" skewness, i.e., the CoV of QP traffic within the hottest VM (denoted as CoV_{vm2qp}). The first two columns of Figure 2(b) depict the distribution of CoV_{vm2qp} for read and write, respectively. The results show that the CoV_{vm2qp} also presents high values, with the median for read and write being 0.78 and 0.62, respectively. These results imply that the traffic within the hottest VM tends to concentrate on a few QPs; in other words, the parallelism of multiple QPs is under-utilized.

Note that the QPs within the VM may belong to multiple VDs, hence, we delve into the traffic distribution of "VM-VD-QP". To do this, we examine the "VM-to-VD" (CoV_{vm2vd}) and "VD-to-QP" (CoV_{vd2qp}) skewness separately. The last four columns of Figure 2(b) depict the distribution of CoV_{vm2vd} and CoV_{vd2qp}, respectively. We make the following observations from the results:

- First, "VM-to-VD" traffic presents extreme skewness with the median CoV of 0.97 and 0.96 for the read and write traffic. This phenomenon is understandable, as the system disk may has lower traffic than the data disk.
- Second, for the "VD-to-QP" traffic, we also observe significant inter-QP skewness. The median CoV reaches 0.39 and 0.81 for the read and write traffc. We further discover that the improper configuration is the root cause of VD-to-QP skewness. Specifically, a multi-QP VD works as an NVMe SSD for the VM. Although the block-MQ [22] in the Linux kernel supports the multi-queue feature of NVMe, the default scheduling policy is "none". This means the IOs from one IO thread will be sent to only one queue. Moreover, multiple IO threads need to be bound to different CPU cores; if they runs on the same core, the multi-queue feature is still underutilized.

Note that among the three types, *Type III* accounts for the highest proportion at 78.9%, followed by *Type II* at 18.0%. This indicates that in most cases, WT skewness is due to the concentration of storage traffic to a few QPs.

4.3 Known Techniques

One common practice is to improve the utilization of QP parallelism. One can employ the Linux blk-MQ [22] to leverage the parallelism of NVMe multiple IO queues. Further, the user can resort to software RAID [14] to balance the workload among VDs. However, both practices require fundamental reconfiguration on VM's OS or users' application, thereby yielding high engineering effort. In addition, this solution cannot solve the *Type I* and *Type II* skewness. Since, for *Type I*, the number of QPs is less than the number of WTs; for *Type II*, the hotspot VM has only one QP. In both cases, increasing the QP parallelism will not work.

Another seemingly promising solution is to enhance virtualization framework with periodical QP-to-WT rebinding (e.g., FinNVMe [39] and LPNS [40]). By rebinding the QPs to different WTs, the system should avoid idle (i.e., *Type I*) and overheated (i.e., *Type II* and *Type III*) WTs.

However, our further analysis suggests otherwise. First, for certain compute nodes, the hottest QP contributes almost all of the traffic. Figure 2(c) depicts the CDF of the traffic share of the hottest QP. The results show that for write traffic, 20.1% of the nodes have their hottest QP contributing more than 80% of the traffic. And for the read, the proportion increases from 20.1% to 42.6%. This observation implies that to achieve the inter-WT balance, the rebinding has to be performed at a very high frequency, or even on a per-IO basis. However, as discussed by Peng et al. [39], the overhead cannot be ignored especially for NVMe virtualization.

Second, even if one is willing to rebind at a rather high frequency, the traffic of WT may still be skewed due to the burst. We simulate the rebinding using the trace data to evaluate its effectiveness. Specifically, for each 10ms period (0.1× to the setting in [39]), we check if the traffic of the hottest WT is more than 1.2× to the coldest one. If so, we swap the QPs bound to these two WTs. To evaluate the cost and benefit of rebinding, we define: (1) the *rebinding ratio*: the number of periods triggering rebinding divided by the total number of periods; (2) the *rebinding gain*: the WT-CoV before rebinding divided by the WT-CoV after rebinding. A gain less than 100% indicates a more balanced traffic after rebinding.

Figure 2(d) shows the results of simulation. In the figure, each point represents a compute node, where the X-axis represents the rebinding ratio, and the Y-axis represents the rebinding gain. We can see that not all nodes achieve positive gains, only 29.9% of nodes have the gain larger than 100%. Some nodes (indicated by the blue circle) have the gain close to 100%, even though the rebinding ratio reaches 60%. Other nodes (indicated by the red circle) can achieve large rebinding gain of 1% with a rebinding ratio around 20%.

To further explore this counterintuitive result, we select one node from both the blue and red circle, named as node-b and node-r. Then, we plot the traffic time series of their hottest WT on a 10ms scale in Figure 2(e) and (f). Clearly, the burst traffic of node-b is much stronger than node-r. The P2A of node-b reaches 80.6 which is 7.7× to node-r. For node-b, the duration of traffic burst can be less than 10ms. Thus, the rebinding period should be further reduced to achieve load balancing. As previously mentioned, extremely low period impies unacceptable overhead [39].

It should be noted that the above results are based on the trace data with a 1/3200 downsampling. One might be concerned that downsampling could bias the results. Hence, we indirectly validate our findings by examining the metric data. Specifically, we measure the traffic contribution of the hottest QP within a compute node. We find that for 42.6% of the nodes, the hottest QP contributed more than 80% of the read traffic. This implies that the substantial traffic contributed by a single QP requires a much higher frequency of rebinding.

4.4 Possible Solutions & Challenges

Based on the above discussion, we assume one potential solution is to perform load balancing on a per-IO basis. To achieve this, the existing single-WT hosting should be restructured to multi-WT hosting, i.e., allowing multiple WTs to share the traffic of the hottest QP.

One challenge of multi-WT hosting is the overhead of multi-thread locking. To reduce the synchronization overhead, we may rely on the hardware support like FPGA. First, FPGA supports efficient hardware queue, such as FIFO queue [7] and ring buffer [24]; Second, FPGA can efficiently distribute data between hardware queues, utilizing mechanisms like AXI stream [11] and hardware-based scheduler [42, 44]. Nevertheless, deploying NVMe virtualization on FPGA is a challenging task, as it requires careful consideration in the division between hardware and software.

Another challenge is that the polling-mode of WT coupled with the round-robin bind can ensure a well multi-tenant fairness. Even if a QP is hot, the other QPs bound to the same WT can still be served, since the WT polls requests from each QP in turn. Therefore, switching to multi-WT hosting requires additional mechanisms to ensure fairness.

Key Takeaways. *User's storage traffic often clusters in few QPs of mounted VDs, challenging the existing round-robin or rebinding-based load balancer. Instead of the current inline thread model, a dispatch model is essential. However, software-based dispatcher falls short in achieving low overhead. A hardware-based dispatcher (e.g., FPGA or ASIC) presents a promising solution.*

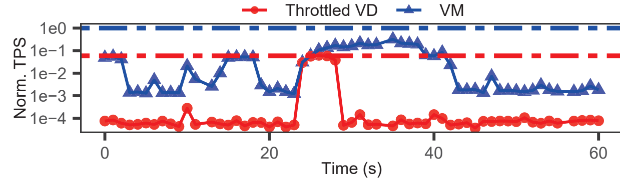
5 Traffic Throttle in the Hypervisor

We, like many other cloud vendors [1, 2, 4, 5], offer users VMs and VDs with various specifications. To ensure Service Level Objective (SLO), we impose traffic throttle on the VDs. Specifically, we place a cap both on the throughput and IOPS for each VD. Once the throughput or IOPS exceeds the threshold, additional IOs will queue in the hypervisor. Zhou et al. [58] reveal that the IOPS throttling of AWS EBS [2] can cause latency spike and significantly impair the performance of LSM store-based applications such as RocksDB. This finding prompts us to re-examine the design of throttle model in the hypervisor, considering the skewness at various levels mentioned in §4.

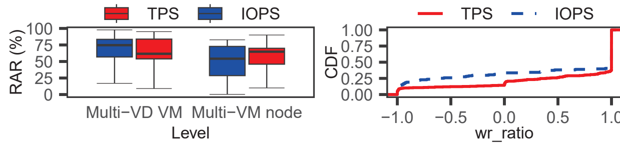
5.1 Symptom

The performance of multi-VD VM is impaired by traffic throttle of a single VD, even when other VDs have available bandwidth. Similarly, this issue arises when multiple VMs of a tenant reside on the same compute node.

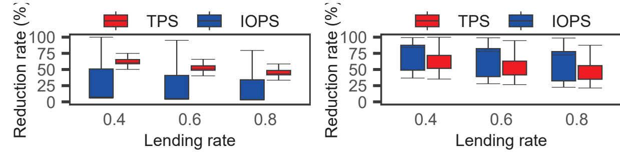
Figure 3(a) shows a real-world scenario where a VM with 32 VDs suffers from single VD throttle. The red line represents the throughput of the throttled VD, and the blue line



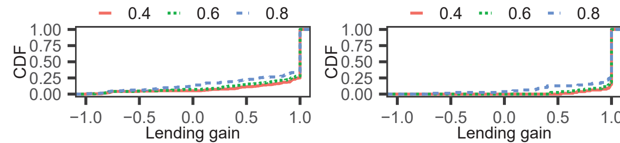
(a) Real case of single VD throttle. The Y-axis represents the normalized throughput. The values are normalized to the total throughput limit of the VM, i.e, the blue horizontal dashed line.



(b) Resource available rate. **(c)** CDF of *wr_ratio*.



(d) Reduction rate of multi-VD VM. **(e)** Reduction rate of multi-VM node.



(f) Lending gain of multi-VD VM. **(g)** Lending gain of multi-VM node.

Figure 3. Traffic Throttle in the Hypervisor (§5). (a) A 32-VD VM suffers from single VD throttle; **(b)** The resource available rate (RAR); **(c)** The CDF of write-to-read ratio under throttle; **(d)** / **(e)** The reduction rate of throttle duration; **(f)** / **(h)** The lending gain of multi-VD VM / multi-VM node.

indicates the total throughput of the VM (i.e., the summed traffic of all mounted VDs). At around 25s, a burst hits the throughput cap of the throttled VD (indicated by the red dashed horizontal line). However, the VM's total throughput is far from the overall threshold (indicated by the blue dashed horizontal line).

Simply put, when a VD is throttled, other VDs within the VM likely have available throughput or IOPS. Similarly, we have also observed such "available resource" when multiple VMs of a tenant are placed on the same compute node (i.e., multi-VM node). Here, "resource" refers to the throughput or IOPS, either of which can trigger the throttle upon reaching the cap.

Next, we formulate the available resource and measure it at scale. We denote the *Available Resource* (*AR*) for a multi-VD VM at time t as $AR(t)$. The $AR(t)$ is calculated as: $AR(t) = Cap - VM(t)$, where Cap is the summed cap of all mounted VDs and $VM(t)$ is the summed throughput (or IOPS). We further define the *Resource Available Rate* (*RAR*) as the proportion of available resource at time t:

$$RAR(t) = \frac{AR(t)}{Cap} = \frac{Cap - VM(t)}{Cap} \tag{1}$$

The *RAR* ranges in $(0, 100\%]$ with values near 0 indicating no available resource.

Figure 3(b) shows the *RAR* distribution for multi-VD VM and multi-VM node during throttling. We can see that the *RAR* exhibits much high values both for multi-VD VM and multi-VM node. For multi-VD VM, the median *RAR* of throughput and IOPS is 61.6% and 74.7%, respectively. Similar results can be observed for multi-VD node. To summarize, when a VD is throttled, the available resource is nearly always abundant. This renders a great opportunity, as long as the tenant allows, to share the available resource among VDs. Note that this observation also holds for multi-VM node.

5.2 Root Cause

Like other EBS vendors [1, 2, 4, 5], we aggregate read and write traffic to monitor the resource cap. Therefore, we further analyze the contribution of read and write traffic to the throttle. Specifically, we calculate the *normalized write-to-read ratio* (*wr_ratio*) when a VD is throttled, which is defined as:

$$wr_ratio = (W - R)/(W + R) \tag{2}$$

where, W and R represent the throughput (or IOPS) of write and read, and *wr_ratio* ranges in [-1, 1].

Figure 3(c) presents the CDF of *wr_ratio* separated by throughput and IOPS. We make the following observations. First, for both throughput and IOPS, the CDF shows significant increases near 1. In other words, write traffic is the main contributor of throttle. Second, only a small portion of samples fall within the range of $[-1/3, 1/3]$ [4] (11.7% for throughput and 6.9% for IOPS). This indicates that the throttle is primarily caused by traffic from either read or write, rather than both. Additionally, we observe that throttle by throughput occurs much more frequently (14.3× higher) than that by IOPS.

5.3 Potential Solutions & Challenges

Based on the observations in §5.2, an intuitive solution is to place different caps for read and write traffic. However, this requires the tenant to have accurate profiling of their workloads. Even if most EBS vendors support elastic virtual disks to accommodate fluctuating performance demands, the scaling capability is still insufficient. For example, AWS EBS may requires hours to achieve scaling [3].

[4] A *wr_ratio* equal to $1/3$ ($-1/3$) means that the write (read) traffic is 2× to the read (write).

Given the abundant available resource, another method is "resource pooling". We can pool the purchased throughput or IOPS across tenant's VDs. Nevertheless, this can backfire with VDs freely preempting resources from the pool and blocking the IO of other VDs.

Hence, a more practical solution is "limited lending", which allows the throttled VD to harvest the available resources by increasing its cap and decreasing the cap of other unthrottled VDs. We first theoretically present the benefits of limited lending. Suppose for a multi-VD VM, a VD is throttled at time t. The throttled VD is allowed to lend an extra cap of $p \times AR(t)$ from other unthrottled VDs, where $p \in (0, 1)$ is the lending rate. The lent cap, i.e., $p \times AR(t)$, allows us to measure the *Reduction Rate* (*RR*) of the throttle duration as follows:

$$RR = \frac{VD(t)}{VD(t) + p \times AR(t)} \tag{3}$$

Clearly, the reduction rate falls within $(0, 100\%]$, with lower rate indicating shorter throttle duration after lending. This definition can also extent to the multi-VM node.

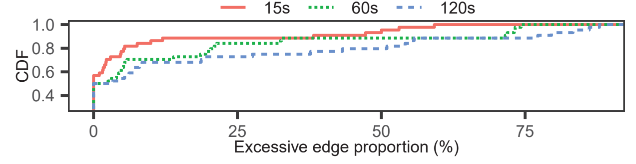
Figure 3(d) and (e) depict the distribution of the reduction rate at various p for multi-VD VM and multi-VM node. For multi-VD VM, with $p = 0.8$, we can observe a median reduction rate of 43.7% in throughput and 3.9% in IOPS, i.e, the throttle due to IOPS can be greatly alleviated. And for multi-VM node, we can still observe a clear reduction.

Although theoretically, the limited lending is effective, reaping such benefits in practice remains challenging. In the previous analysis, we implicitly assumed that the extra cap has already been allocated before the VD is throttled. However, the extra cap is allocated only after the throttle during runtime. This can lead to a scenario where the VD lending the cap out is throttled again.

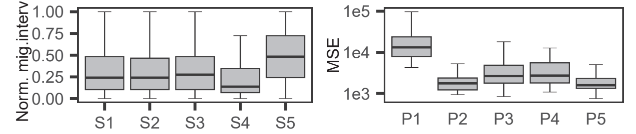
To evaluate a more realistic limited lending, we design a primary lending mechanism for the proof-of-concept. Specifically, the limited lending operates in a periodic manner. In each period, assume that a VD is first throttled at time t. The throttled VDs can lend an extra cap of $p \times AR(t)$ from unthrottled ones. Details of the lending mechanism are in Appendix B.

We simulate the above mechanism on the metric data and measure the *Lending Gain* for each multi-VD VM (or multi-VM node). The lending gain is defined as $(t_{w/o} - t_w)/(t_{w/o} + t_w)$, where $t_{w/o}$ and t_w are the total throttle duration without and with lending. The lending gain ranges in $(-1, 1)$, and a value larger than 0 indicates that the throttle duration can be reduced with lending.

Figure 3(f) and (g) show the distribution of lending gain for multi-VD VM and multi-VM node under various lending rates. The results show that the lending yields positive gains in most cases. For multi-VD VM, 85.9% of the samples experience a positive gain with $p = 0.8$. This observation also holds for multi-VM node.



(a) Proportion of frequent migration.



(b) Normalized migration interval. **(c)** Mean squared error.

Figure 4. Frequent Segment Migration (§6.1). (a) Proportion of frequent migration under various time scale; **(b)** Normalized migration interval under five importer selection; **(c)** Mean squared error of five traffic prediction methods.

However, even at a conservative lending rate ($p = 0.4$), positive gains are not guaranteed. For multi-VD VM, 5.2% of the samples still exhibit a negative gain. The primary lending design may, in return, make throttle worse. The underlying reason is that the VD that lends cap out may have a burst, and hit the cap again.

Thus, a practical lending requires traffic prediction to adjust the lending rate, ensuring the VD lending cap does not get throttled again. However, predicting EBS traffic is rather challenging as discussed in [37]. Additionally, lending may poses security issues by disrupting VD isolation. For example, an attacker may launch side-channel attack by controlling a compromised VD covertly consuming resources.

> ***Key Takeaways.*** *The hard-threshold throttle does not fit users' skewed traffic. Hence, pooling the user's purchased bandwidth can effectively alleviate the long-tail caused by throttle. However, it requires the hypervisor to predict hotspots to set the dynamic threshold.*

6 Load Balancing in the Storage Cluster

The inter-BS load balancer is crucial for maintaining the performance and reliability of the storage cluster. Our balancer operates on principles similar to the HDFS Balancer [10] and Ceph's metadata server balancer [50], balancing traffic by migrating segments between BSs (details in Appendix A). When facing the skewed traffic, we identify two symptoms of the current balancer, namely frequent segment migration (§6.1) and skewed read (§6.2).

6.1 Frequent Segment Migration

6.1.1 Symptom. *In certain storage clusters, we have observed frequent but unnecessary segment migration.*

Even for the load balancing, the migration cannot be executed wildly, as the migration temporarily halts the service. We measure the migration frequency of the current load balancer as follows. First, we divide the 12-hour observation time into small time windows (e.g., 15s). Then, for each time window, we check whether a BS has both incoming and outgoing migrations. If so, we mark the these migrations as frequent. After iterating all the time windows and BSs, we actually divide the migrations into two subsets: frequent and non-frequent.

We then calculate the proportion of frequent migrations to the total migrations for each storage cluster. Figure 4(a) shows the CDF of the proportion of frequent migrations across all storage clusters. The red line indicates that 56.8% of the clusters have no frequent migrations in 15s time scale (i.e., the proportion is equal to 0). But in one cluster, the proportion reaches a maximum as 59.2%. This suggests that, in some clusters, segments are migrated out of a BS soon after being migrated in, or vice versa.

6.1.2 Root Cause. Analyzing the traffic of the storage cluster with the most frequent migrations, we identify the importer selection as the root cause. Currently, the balancer selects the BS with the lowest traffic in the current period as the importer candidate. However, the rapid traffic fluctuation can turn the importer into a new hotspot. Ideally, the importer should be the BS with the lowest future traffic growth.

To verify this, we build the balancer with various importer selection methods and simulate the balancer on the metric data. Specifically, we use the following importer selection methods: (1) *Random*: randomly select a BS as the importer; (2) *Minimum traffic*: select the BS with the lowest traffic as the importer (current method); (3) *Minimum variance*: select the BS with the minimum traffic variance as the importer; (4) *Lunule* [49]: Lunule uses the linear fitting to predict the traffic for the next period, and selects the BS with the lowest predicted traffic as the importer; (5) *Ideal*: directly select the BS with the lowest traffic in the next period as the importer, since we know all the future traffic in the simulation.

In the simulation, we record the timestamp when the segments are migrated out of a BS, and then calculate the normalized time interval between two adjacent migrations. The distribution of normalized migration interval is plotted in Figure 4(b). We make the following observations. First, the distributions of *Random* (S1) and *Minimum traffic* (S2) are almost identical, with median migration interval as 0.24. This suggests that the current importer selection (S2) is ineffective. Second, *Lunule*'s (S4) linear fit model can worsen the situation, reducing the median migration interval to 0.14. Lunule is designed for the load balancing of CephFS metadata server, where the workload fluctuation might be less severe than the EBS. Thus linear fit may provide inaccurate prediction; Third, *Ideal* (S5) can effectively extends the migration interval, with a median as 0.48, which is 2.0× to the current balancer (S2).

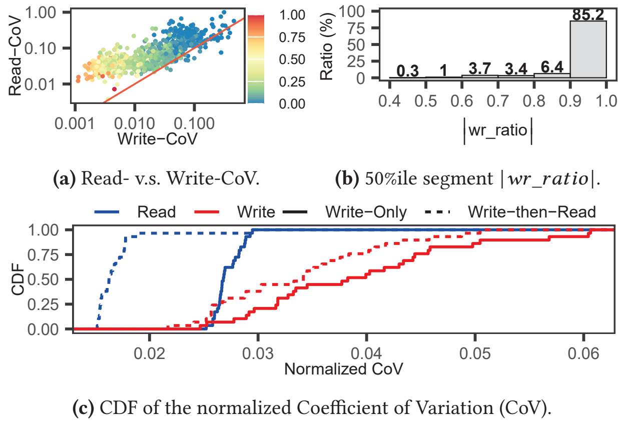
6.1.3 Potential Solutions & Challenges. The discussion above indicates that reducing the unnecessary frequent migration requires accurate prediction of the future traffic. However, the prediction is rather challenging in the public cloud with varying applications. Mao et al. [37] claim the deficiency of both standard [23, 45] and machine learning-based [46] method in predicting EBS traffic (Finding 6 of [37]). We reverify this on our dataset.

Recall that the balancer operates in a periodic manner. For each period, we use four methods to predict the traffic for the next period on a per-BS basis, including linear fit [13], ARIMA [8], XGBoost [18] and Transformer [47]. XGBoost and Transformer, as machine learning models, require sufficient data for training. Given the training overhead, it is not practical to build the model for every period. Instead, we build the model every 200 periods (referred as an epoch). We use the model from the previous epoch to predict the period traffic of the current epoch. Linear fit and ARIMA, as statistical models, can be updated every period. Detailed implementations of the four prediction algorithms can be found in Appendix C.

We evaluate the accuracy of the four prediction models using Mean Squared Error (MSE) [15], and the results are shown in the first four columns of Figure 4(c). We can see that ARIMA (P2) achieves the lowest MSE and linear fit (P1) has the highest error. Although ARIMA's MSE is relatively small, its predictions still deviate from the ground truth. This validates that traditional prediction methods are not effective.

The MSE of XGBoost (P3) and Transformer (P4) are nearly the same and much higher than the ARIMA (P2). As claimed by Huang et al. [28], Transformer can effectively predict the workload in multi-tenant cloud platforms. This is contradicted to our result. One possible reason is that our Transformer is updated on a per-epoch basis, the long update interval may miss the recent traffic fluctuation. Therefore, we perform updating on a per-period basis, and present the result as P5 in Figure 4(c). Clearly, at a higher update frequency, Transformer can achieve a lower MSE. Additionally, the parameter fine-tuning is also an important factor, but this is beyond the scope of this paper.

To sum up, deploying the prediction-based balancer in the production system is rather challenging. First, traditional statistical models perform poorly. Second, for machine learning model, a careful trade-off must be made between prediction accuracy and retraining overhead. A viable approach is to perform fine-tuning on a baseline model, i.e., use the newly



(a) Read- v.s. Write-CoV. **(b)** 50%ile segment $|wr_ratio|$.

(c) CDF of the normalized Coefficient of Variation (CoV).

Figure 5. Balanced Write but Skewed Read (§6.2). (a) 2D-plot of read- and write-CoV of storage clusters. **(b)** Histogram of average segment $|rw_ratio|$ of storage clusters. **(c)** Normalized CoV for read and write in cases of Write-then-Read and Write-Only migration.

arrived traffic to update the model for capturing the short-term fluctuation. Third, there are additional constraints on segment migration. For example, for reliability, the number of segments carried by a BS is limited, and segments from the same VD should be distributed across different BSs. These constraints further complicate the design of the balancer.

6.2 Balanced Write but Skewed Read

As noted in §2.2, our balancer performs migration solely based on write traffic. Given the EBS's write dominance, migration that considers both read and write may introduce interference. In other words, migrating segments based on read skewness can disrupt the write balance and trigger more migrations. The rise of read-intensive applications (such as AI training and big data analytics [20, 32, 48, 59]) motivates us to reconsider the balancer of leaving the read migration in the wild.

6.2.1 Symptom.

Read traffic exhibits more severe inter-BS skewness compared to write traffic.

We use the normalized CoV to measure the traffic skewness of BSs within the storage cluster. To compare that between read and write, we use the 2D-plot shown in Figure 5(a). Each point represents a storage cluster, with the X-axis representing the CoV of write traffic, and the Y-axis representing the read traffic. The color of the points represents the normalized write traffic of the cluster, and the red diagonal line is the reference line $y = x$.

We make the following observations. First, the skewness of read traffic is much worse than the write. 96.8% of the points are located above the reference line, i.e., read-CoV ≥ write-CoV. Second, as the write traffic increases, both write-CoV and read-CoV decrease. This means that the balancer solely based on write traffic can also balance the read traffic. Third, as the write traffic further increases, the points move further away from the reference line. This shows that, when the write traffic is large, relying solely on the write traffic migration is not sufficient to further balance the read traffic.

6.2.2 Potential Solutions.

One concern of leaving the read migration out is the interference between read and write migration. Hence, we further analyze the segment's read and write behavior by measuring the absolute write-to-read ratio, i.e., $|wr_ratio|$. In Equation 2, if $|wr_ratio|$ is close to 1, the segment is either read-dominant or write-dominant.

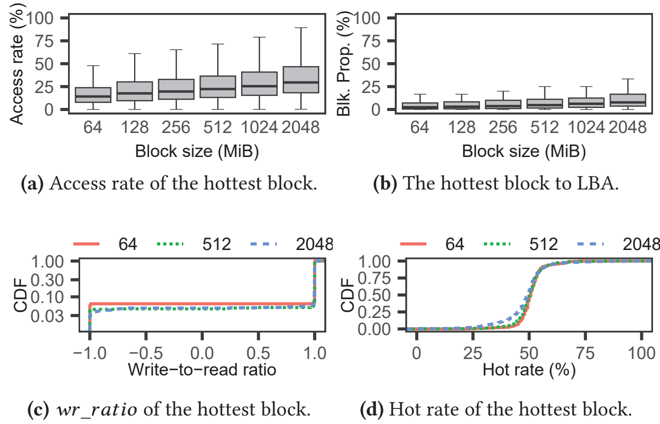
For each storage cluster, we calculate the 50%ile of segment's $|wr_ratio|$ and depict the histogram in Figure 5(b). Note that, for each cluster, we only select the segments with cumulative traffic contribution more than 80% for statistic. This is because the traffic is contributed by only a small portion of segments (see Table 3).

From the figure, we observe that 85.2% of the storage clusters have an 50%ile $|wr_ratio|$ larger than 0.9. This indicates that the read traffic is 19× more than to the write, or the other way around (i.e., write to read). To sum up, for most of the storage clusters, the segments tend to be either write-dominant or read-dominant. This observation can help simplify the design of the balancer since migrations for reads and writes can be conducted concurrently without concern for interference.

To validate this finding, we perform the following simulation. Similar to §6.1.2, we select the cluster with the most frequent migrations and choose the *Ideal* method for importer selection. We calculate the normalized CoV for read and write traffic in each migration period using two algorithms: (1) *Write-then-Read*: migrate write traffic first, then the read traffic; (2) *Write-Only*: migrate only write traffic.

Figure 5(c) shows the CoV for read and write traffic under the two algorithms. First, as expected, the skewness of read traffic is significantly alleviated by the *Write-then-Read* migration. Second, surprisingly, the migration of read traffic does not intensify the skewness of write traffic but actually alleviates it. We speculate that the read migration incidentally randomizes the segments with smaller write traffic, making the write traffic more balanced.

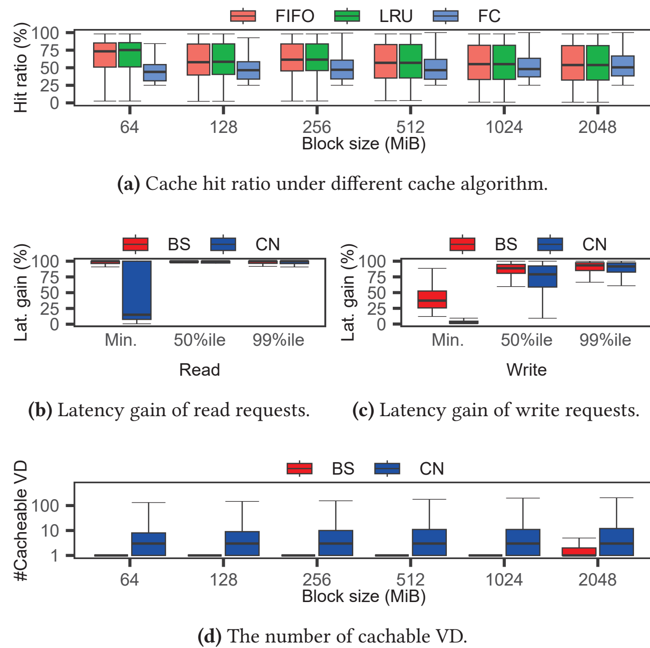
Key Takeaways. *The heuristic of selecting the traffic importer by the historical minimum can be easily broken by the volatile traffic. The balancer needs to be prophetic and avoid the potential hotspots. However, predicting hotspots can be rather challenging. Even if deep learning offers better accuracy, it requires further exploration to manage the training overhead.*



(a) Access rate of the hottest block. **(b)** The hottest block to LBA.

(c) wr_ratio of the hottest block. **(d)** Hot rate of the hottest block.

Figure 6. Cache across the EBS Stack (§7). (a) Access rate of the hottest block under various block size; **(b)** Proportion of the VD's hottest block size to the LBA; **(c)** Write-to-read ratio of the hottest block; **(d)** Hot rate of the hottest block.



(a) Cache hit ratio under different cache algorithm.

(b) Latency gain of read requests. **(c)** Latency gain of write requests.

(d) The number of cachable VD.

Figure 7. Cache across the EBS Stack (§7). (a) Cache hit ratio under three cache algorithms; **(b)** / **(c)** Latency gain of CN and BS-cache for read / write requests; **(d)** Cache space utilization of CN and BS-cache.

7 Cache Across the EBS Stack

This section delves into the spatio-temporal skewness at the VD's LBA level. Previous studies [34, 35], under a smaller scale, show that the VD's IO tends to concentrate on a small proportion of the LBA. In this section, we first re-examine this finding at a larger scale. More importantly, we discuss several design insights for employing the cache across the EBS stack, including choices of algorithms, potential locations and trade-offs.

7.1 Symptom

The VD's IO still shows significant spatio-temporal hotspots on the LBA level at large scale.

For each VD, we divide its LBA into fixed-size blocks, and calculate the access rate of each block. We refer to the block with the highest access rate as the VD's hottest block. Figure 6(a) presents the access rate of the hottest block under various block size, and Figure 6(b) shows the proportion of the block size to the VD's LBA size.

Figure 6(b) shows that the 64 MiB block constitutes only 3.0% of the LBA in the median term, yet the access rate can reach 18.2% as shown in Figure 6(a). This observation holds true with larger block sizes. Therefore, the VDs show obvious hot spot at LBA level, which is coherent with the findings in prior studies [34, 35]. Even with page cache in the VM, significant hotspots are still observed.

7.2 Root Cause

We now demonstrate the access pattern of the hottest block by analyzing its write-to-read ratio and temporal continuity. In Figure 6(c), we calculate the write-to-read ratio (defined in Equation 2) of the hottest block for each VD and plot the CDF under various block sizes. We make the following observations. First, most of the hottest blocks are write-dominant. For the 64 MiB block, 93.9% of the hottest blocks have a wr_ratio greater than 1/3. Namely, write traffic is twice as much as that of the read. Second, only 5.5% of the hottest blocks are read-dominant (i.e., $wr_ratio \leq -1/3$). To conclude, the data prefetching mentioned in §2.2 has limited effects because writes, which can be dominant, are not buffered in the pre-fetching cache.

Then, we measure the temporal continuity of the VD's hottest block. Assume that the hottest block of a VD has a $p\%$ access rate during the observation time (e.g. 12 hours). We recalculate the access rate of this hottest block over a shorter window (e.g. 5 minutes), and count the percentage of the time windows whose access rate exceeds $p\%$. We name this percentage as the *hot rate* of the hottest block. Figure 6(d) shows the CDFs of the hot rate under various block sizes. The results show that the hot rate approximately follows a Gaussian distribution with an average of 50.0%. This result implies that the hottest block of the VD has well temporal continuity.

7.3 Potential Solutions & Challenges

The temporal continuity and write-dominance of the hottest block prompt us to reconsider the design of cache algorithm (§7.3.1) and its location (§7.3.2).

7.3.1 Cache Algorithm. A recent work, FrozenHot [41], introduces the concept of "Frozen Cache", which caches the hot accessed data without eviction. This significantly reduces the overhead of cache management, especially in high-concurrency scenario. We run a cache simulation on the trace data to verify the feasibility of the frozen cache (FC) in EBS. For comparison, we also introduce two classic cache algorithms, including First-In-First-Out (FIFO), Least-Recently-Used (LRU). Specifically, we set the cache's page size to 4 KiB, and the total size of the cache to match the size of the hottest block. For the frozen cache, we set the cache at the LBA where the hottest block is located, and do not perform any page eviction.

For each VD, we calculate the cache hit ratio after simulation. Figure 7(a) shows the distribution of the hit ratio under various block sizes. We draw the following observations. First, LRU and FIFO exhibit nearly identical distributions across various block sizes. Given the write-dominance revealed in §7.2, we infer that the hottest block may perform sequential write, resulting in similar hit ratios for FIFO and LRU. Second, when the block size is small (such as 64 MiB), the hit ratio of FC is significantly lower than the FIFO and LRU. However, when the block size increases to 2048 MiB, the hit ratio of FC is already comparable to the FIFO and LRU, and its lower bound is significantly higher. This is consistent with the previous inference that larger cache space (block size) can accommodate more sequential writes, thereby achieving a higher hit ratio. To conclude, frozen cache can indeed achieve effective caching in EBS, but at the expense of a large cache space.

Readers may be concerned that the 1/3200 downsampling rate could affect the results. It is important to emphasize that the hottest block in §7.3.1 is identified on an hourly scale. In other words, the identified hottest block is long-term existence (not instantaneous IO behavior). Therefore, we believe that a higher sampling rate could make the hotspot even more apparent, which does not contradict with existing findings.

7.3.2 Cache Deployment Location. Although caching in compute-storage disaggregation is not new [21, 52, 56, 57], deploying cache in EBS presents challenges. First, the cache must adhere to the reliability requirement of EBS, i.e., the writes must be persisted (with redundancy) before confirming success. In other words, the cache should be a persistent cache (e.g., flash or PMEM) rather than an in-memory cache. Second, there are two available locations for cache deployment: Compute Node (CN) and BlockServer (BS). These two locations can have quite different pros and cons.

Next, we perform a detailed comparison of the two cache locations in terms of *latency gain* and *cache space utilization*. We assume the use of frozen cache [41] in the following experiments. First, FC shows a hit ratio comparable to LRU with larger cache space (2 GiB in Figure 7(a)). This aligns well with the need for persistent memory in EBS, as it can provide larger space than volatile memory. Second, FC can save more CPU resources by eliminating the maintenance of metadata and the page eviction.

Latency gain. If the cache is deployed on the CN (i.e., CN-cache), IO requests can save the latency of accessing the storage cluster, while BS-cache can save the latency of accessing the ChunkServer. We define the *latency gain* as the ratio of IO end-to-end latency with and without cache. For example, to obtain the 50%ile latency gain of BS-cache, we first calculate the 50%ile latency with and without cache as $p50_{BS}$ and $p50_{w/o}$. Then, we derive the latency gain as $p50_{BS}/p50_{w/o}$. Clearly, the latency gain ranges in $(0, 100\%]$, with a smaller value indicating a greater improvement by cache.

We compare the read and write latency gain for the two cache locations in terms of 0%ile, 50%ile and 99%ile. The results of read and write are separately shown in Figure 7(b) and (c). First, for the read, neither CN-cache nor BS-cache provides significant latency gains, except that the CN-cache offers gain for 0%ile latency. This is understandable as most of the hottest blocks are write-dominant. Second, for the write, CN-cache has a better gain than the BS-cache in 0%ile and 50%ile. However, neither CN nor BS-cache is able to improving the 99%ile latency. This is as expected since long-tail IOs may not fall into the hottest block.

Cache space utilization. For a large-scale deployment, we may suffer from cache under-utilization if over-provisioned. We define the *cache space utilization* to measure the cache cost. Specifically, assume that the cache is deployed on a per-node basis. As shown in Figure 6(a), not all the hottest blocks have a high access rate. Hence, an intuition is to set up the frozen cache only on the VDs whose hottest block access rate exceeds a threshold. In the following experiment, the threshold is set to 25%; and VD with a hottest block access rate exceeding 25% is called *cacheable VD*.

As for field deployment, the cache space are normally the same across nodes. Thus, the cache space utilization can be quantified by the number of cacheable VDs. For example, for CN-cache, we count the number of cachable VD for each compute node, since we know the node where each VD is located (refer to Table 1). The BS-cache is similar. A wider range in the number of cacheable VDs indicates more severe under-utilization.

Figure 7(d) shows the distribution of cachable VD number under various block sizes. The results show that the standard deviation for CN-cache is significantly higher than the BS-cache. With 2048 MiB block size, the standard deviation of CN-cache is 21.0× to the BS-cache. To summarize, the lower standard deviation of BS-cache indicates much less over-provisioning.

Both advanced CN-cache [53] and BS-cache [56] have been proposed in academia. However, several challenges remain in applying them to the production EBS system. For the CN-cache, although it offers better latency gain, the violation of the design principle of compute-storage disaggregation induces further issues. For example, the consistency of cached data with the persistent data, and the overhead it creates for the VM live migration. And for BS-cache, the latency gain is significantly weaker than the CN-cache.

For a cost-benefit trade-off, CN-cache and BS-cache can be deployed in a hybrid manner. For example, CN-cache can be deployed on a per-rack basis for low cost and use NVMe-oF [27] for fast caching. Meanwhile, BS-cache can serve as a backup for CN-cache to handle situations where CN-cache space is insufficient.

Key Takeaways. *The VD's IO shows apparent LBA hotspots, leaving much headroom for caching. The emerging cache algorithm, e.g., Frozen Cache, can deliver high hit ratio with large cache space. This naturally meets the EBS demand for persistent caching with high-capacity flash or PMEM. For the cost-benefit trade-off, hybrid cache deployment should be considered.*

8 Potential Limitation

Specific to our cloud. Throughout the paper, we focus on studying the traffic patterns and potential solutions in the context of our EBS. It is possible that these findings may not be directly applicable to similar services in other cloud providers. However, we believe that the learned lessons can reach a broader audience. First, our EBS share architectural similarities with other cloud storage systems from high-level design (e.g., compute-to-storage disaggregation) to low-level implementation (e.g., IO virtualization framework). Second, the traffic patterns we observed, such as the "VM-VD-QP" traffic distribution and the LBA hotspots, are architecture-independent. Given the large-scale of datasets, we believe the access patterns, instead of being specific to our cloud, shall reflect a common trend. Moreover, we commit to release the traces to motivate further research on the open-ended challenges we have identified or other related topics.

Trace-driven analysis. Due to the complexity of the stack and the large scale, it is unlikely for us to validate potential solutions on real clusters. Hence, our study relies on trace-driven simulation to validate our findings and hypothesis. We mainly focus on the context of algorithm design such as the load balancing and the cache algorithm. In this context, simulation on the production data is sufficient to provide a well preliminary validation. Besides, simulation-based analysis has also been adopted in previous studies, such as the cache simulation in [34, 35].

9 Related Work

Several prior works have made significant efforts to characterize the traffic of storage systems. Early research focused on understanding the IO characteristics of single-machine storage system, providing design insights for power saving [38], file system [26] and block cache [30]. Subsequent studies shifted their focus toward profiling distributed storage systems in high-performance computing (HPC) scenario [31, 36]. Research efforts on cloud block storage system are also abundant [33–35, 60]. Lee et al. [33] characterized the behavior of storage traffic collected from 300 VMs dedicated to virtual desktop. Li et al. [34, 35] analyzed block-level IO traces collected from over 6,000 VDs, providing insights for load balancing and cache design. Zou et al. [60] focused on the statistical modeling of IO inter-arrival times and self-similarity.

The studies most closely related to ours are [34] and [35]. Our study is distinguished in three key aspects. First, our study is comprehensive. We measure multiple components of the EBS stack, spanning across the compute and storage clusters. The previous studies focused solely on either the compute [33] or the storage [34, 35, 37]. Second, our study is fine-grained. We analyze traffic down to the IO queue pairs of virtual disks and the worker threads of the hypervisor. Third, our study is large scale which includes approximately 60k VMs and 140k VDs. We believe this scale allows for a more accurate profiling for large-scale EBS system.

10 Conclusion

In this paper, we conduct a large-scale analysis on the traffic of Elastic Block Storage (EBS) system. We characterize the spatio-temporal skewness across multiple components of the EBS IO stack. We reveal several symptoms in the hypervisor and cluster management caused by traffic skewness. For each symptom, we conduct an in-depth analysis of its root cause and derive several design insights on load balancing, throttle mechanism, and cache deployment. Additionally, we further validate the potential solutions through simulation experiments.

Acknowledgments

We thank the EuroSys reviewers and our shepherd, Youyou Lu, for their valuable comments, the DiTing and KuaFu team for their timely support. This paper is supported in part by the NSFC (61936015, 62072306), Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3, and Alibaba Innovative Research (AIR) project.

References

[1] 2023. AliCloud - Elastic block storage. https://www.alibabacloud.com/zh/product/disk.
[2] 2023. Amazon Elastic Block Store. https://aws.amazon.com/cn/ebs.
[3] 2023. Amazon Elastic Block Store Pricing. https://aws.amazon.com/cn/ebs/pricing/.

[4] 2023. Azure Disk Storage. https://azure.microsoft.com/zh-cn/products/storage/disks.

[5] 2023. Google Cloud - Persistent Disk. https://cloud.google.com/persistent-disk?hl=zh-CN.

[6] 2024. Amazon S3. https://aws.amazon.com/cn/s3/.

[7] 2024. AMD FIFO Generator. https://www.xilinx.com/products/intellectual-property/fifo_generator.html.

[8] 2024. Autoregressive integrated moving average. https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average.

[9] 2024. Build Ultra High-Performance Storage Applications with the Storage Performance Development Kit. https://spdk.io/.

[10] 2024. Hadoop. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[11] 2024. How AXI4-Stream Works. https://docs.amd.com/r/en-US/ug1399-vitis-hls/How-AXI4-Stream-Works.

[12] 2024. HUAWEI EVS. https://www.huaweicloud.com/product/evs.html.

[13] 2024. Linear regression. https://en.wikipedia.org/wiki/Linear_regression.

[14] 2024. Linux Raid. https://raid.wiki.kernel.org/index.php/Linux_Raid.

[15] 2024. Mean squared error. https://en.wikipedia.org/wiki/Mean_squared_error.

[16] 2024. NVM Express Explained. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf.

[17] 2024. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[18] 2024. XGBoost Documentation. https://xgboost.readthedocs.io/en/stable/.

[19] Hervé Abdi. 2010. Coefficient of variation. *Encyclopedia of research design* 1, 5 (2010).

[20] Alex Aizman, Gavin Maltby, and Thomas Breuel. 2019. High performance I/O for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 5965–5967.

[21] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. 2016. CloudCache: On-demand Flash Cache Management for Cloud Computing. In *14th USENIX Conference on File and Storage Technologies (FAST)*.

[22] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference (SYSTOR)*. 1–10.

[23] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. 2014. Workload prediction using ARIMA model and its impact on cloud applications' QoS. *IEEE transactions on cloud computing* 3, 4 (2014), 449–458.

[24] Pong P Chu. 2011. *FPGA prototyping by Verilog examples: Xilinx Spartan-3 version.* John Wiley & Sons.

[25] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 519–533.

[26] María Engracia Gomez and Vicente Santonja. 2000. A new approach in the modeling and generation of synthetic disk workload. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

[27] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. 2018. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Transactions on Storage (TOS)* (2018).

[28] Shaoyuan Huang, Zheng Wang, Heng Zhang, Xiaofei Wang, Cheng Zhang, and Wenyu Wang. 2023. One for all: Unified workload prediction for dynamic multi-tenant edge cloud platforms. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

[29] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review* 37, 2 (2007), 51–62.

[30] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization (IISWC)*. 119–128.

[31] Youngjae Kim and Raghul Gunasekaran. 2015. Understanding I/O workload characteristics of a Peta-scale storage system. *The Journal of Supercomputing* 71 (2015), 761–780.

[32] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 283–296.

[33] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*. 1–11.

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

[35] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. 2023. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. *ACM Transactions on Storage (ToS)* 19, 2 (2023), 1–32.

[36] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[37] Haoyu Mao, Yongkun Li, Wenzhe Zhu, Fei Li, and Yinlong Xu. 2023. On Optimizing Traffic Imbalance in Large-scale Block-based Cloud Storage: Trace Analysis and Algorithm Design. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 728–736.

[38] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 1–23.

[39] Bo Peng, Ming Yang, Jianguo Yao, and Haibing Guan. 2020. A throughput-oriented nvme storage virtualization with workload-aware management. *IEEE Transactions on Computers (ToC)* 70, 12 (2020), 2112–2124.

[40] Yao Jianguo Peng Bo, Guo Cheng and Guan Haibing. 2023. LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSD in Clouds. In *2023 USENIX Annual Technical Conference (ATC)*. 785–800.

[41] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*.

[42] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[43] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[44] Yi Tang and Neil W Bergmann. 2014. A hardware scheduler based on task queues for FPGA-based embedded real-time systems. *IEEE Transactions on Computers (ToC)* (2014).

[45] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[46] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles. 2020. A review on the long short-term memory model. *Artificial Intelligence*

Review 53 (2020), 5929–5955.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems (NIPS)* (2017).

[48] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. 2020. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *Proceedings of the 49th International Conference on Parallel Processing (ICPP)*. 1–11.

[49] Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. 2021. Lunule: an agile and judicious metadata load balancer for CephFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[50] Sage A Weil. 2007. Ceph: reliable, scalable, and high-performance distributed storage. (2007).

[51] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. 2004. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC)*. IEEE, 4–4.

[52] Qirui Yang, Runyu Jin, Ni Fan, Devasena Inupakutika, Bridget Davis, and Ming Zhao. 2023. AdaCache: A Disaggregated Cache System with Adaptive Block Size for Cloud Block Storage. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 348–359.

[53] Qirui Yang, Runyu Jin, Ni Fan, Devasena Inupakutika, Bridget Davis, and Ming Zhao. 2023. AdaCache: A Disaggregated Cache System with Adaptive Block Size for Cloud Block Storage. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 348–359.

[54] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. 2018. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE, 67–76.

[55] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, et al. 2024. 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)*.

[56] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 785–798.

[57] Ke Zhou, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu. 2020. Efficient SSD cache for cloud block storage via leveraging block reuse distances. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2020).

[58] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. 2023. Calcspar: A Contract-Aware LSM Store for Cloud Storage with Low Latency Spikes. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*.

[59] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. 2019. Efficient user-level storage disaggregation for deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[60] Qiang Zou, Yifeng Zhu, Jianxi Chen, Yuhui Deng, and Xiao Qin. 2023. Characterization of I/O Behaviors in Cloud Storage Workloads. *IEEE Transactions on Computers (ToC)* (2023).

A Inter-BS Load Balancer

In our EBS system, the inter-BS load balancer shares the same design as the HDFS Balancer [10] and the CephFS MDS balancer [50]. As shown in Algorithm 1, our balancer operates in a periodic manner, detecting high-traffic BlockServer (BS) and migrating segments to low-traffic BS.

Algorithm 1 Inter-BS balancer

Require: BS traffic: w_j^i denotes the total write traffic of $BS_j, j \in \{1, \ldots, n\}$ in period i;
Segment traffc: $ws(k)_j^i$ denotes the write traffic of segment−k of BS_j. By definition, $w_j^i = \sum_k^{m_j} ws(k)_j^i$, where m_j is the number of segments in bs_j;
Segment-to-BS mapping: *Seg2BS*.

1: **for** each period i **do**
2: avg$_i \leftarrow (\sum_{j=1}^{n} w_j^i)/n$; // the cluster's average traffic
3: **for** each bs$_j \in$ BS Set **do**
4: **if** $w_j^i \geq 1.2 \times$ avg$_i$ **then**
5: sorted_segs ← sort($\{ws(k)_j^i\}$, descending);
6: mig_segs ← the top-x segments whose summed traffic exceeding $0.2 \times avg_i$;
7: importer ← $\arg\min_k \{w_k^i\}$; // **select the BS with the minimum traffic as the importer**
8: $w_{importer}^i \leftarrow w_{importer}^i$ + sum(mig_segs);
9: Modify the *Seg2BS*;
10: **end if**
11: **end for**
12: **end for**

B Limited Lending

We design a primary limited lending for proof-of-concept. The Algorithm 2 is described in the context of the multi-VD VM, and a similar design is applied to the multi-VM node.

C Implementation of Traffic Prediction

In §6.1.3, we emply four traffic prediction algorithms. The implementation of each algorithm is as follows. Assume the balancer operates in a 30-second period.

• **Linear Fit.** We use the `LinearRegression` from Python's `sklearn` library. We utilize the data from the past four migration periods to build a linear regression model for each BS, and predict the traffic for the next period.

• **ARIMA.** Similar to linear fitting, we build ARIMA models on a per-BS basis. The ARIMA models are implemented using Python's `statsmodels` library, and the `pmdarima` library is used for automatic parameter searching.

• **XGBoost.** For XGboost, we train a model for each BS on a per-epoch basis, i.e., retraining every 200 periods. We feed the model with 120s historical traffic, and predict the traffic of the next 30s. Since XGBoost is a single-output model, future traffic is predicted using a rolling forecast method, where the predicted output serves as the new input. XGBoost is implemented by the `GradientBoostingRegressor` from the Python `sklearn` library.

• **Transformer.** Unlike the above three methods, Transformer can establish a prediction model for all BSs. This is due to the multi-input multi-output capability of deep learning model. Transformer is retrained on a per-epoch basis, and predict the future traffic for all BSs at once. We use the Transformer model provided by `PyTorch`, with both the encoder and decoder having 2 layers.

Algorithm 2 Limited Lending

Require: VD traffic: $VD_i(t)$ denotes the throughput (or IOPS) of VD_i at timestamp t;
VD cap: Cap_i denotes the throughput (or IOPS) cap of VD_i;
Lending ratio: $p \in (0, 1)$

1: **for** each VD i **do**
2: **for** each timestamp t **do**
3: **if** exists VD$_i(t) \geq Cap_i$ **then**
4: // Calculate available resource defined in §5.1
5: $AR(t) \leftarrow \sum Cap_i - \sum VD_i(t)$
6: $Cap_i \leftarrow Cap_i + \text{AR}(t)$
7: **for** each $j \neq i$ **do**
8: // Decrease the Cap of other VDs
9: $Cap_j \leftarrow Cap_j - (VD_j(t) - Cap_j) \times p$
10: **end for**
11: **end if**
12: break;
13: **end for**
14: Init $\{Cap_i\}$; // Initialize the CAP of the VD
15: **end for**

Table 5. Application types of collected VMs.

Type	Application
BigData	Hbase, Flink, Hadoop, Tensorflow, Alibaba E-MapReduce, Alibaba Elastic HPC
WebApp	Nginx, Jenkins, Git, Crawler, Game, httpd
Middleware	Elasticsearch, Kafka, etcd, Zookeeper, Dubbo, Nacos, Nomad, SLB
File system	FTP, CPFS
Database	Redis, MySQL, Postgress, MsSQL, MongoDB, Oracle, ClickHouse, Prometheus, InfluxDB
App in Docker	K8S, Alibaba ECI, Alibaba ESS

D Application types of the collected VMs

We categorize the applications of the collected VMs into six main types based on their characteristics. The specific applications included in each category are shown in the Table 5.