# DiffForward: On Balancing Forwarding Traffic for Modern Cloud Block Services via Differentiated Forwarding

WENZHE ZHU[*], University of Science and Technology of China, China
YONGKUN LI[†], University of Science and Technology of China, China
ERCI XU, PDL, China
FEI LI, Alibaba Group, China
YINLONG XU, Anhui Province Key Laboratory of High Performance Computing, USTC, China
JOHN C. S. LUI, The Chinese University of Hong Kong, China

Modern cloud block service provides cloud users with virtual block disks (*VDisks*), and it usually relies on a *forwarding layer* consisting of multiple proxy servers to forward the block-level writes from applications to the underlying distributed storage. However, we discover that severe traffic imbalance exists among the proxy servers at the forwarding layer, thus creating a performance bottleneck which severely prolongs the latency of accessing *VDisk*s. Worse yet, due to the diverse access patterns of *VDisk*s, stable traffic and burst traffic coexist at the forwarding layer, and thus making existing load balancing designs inefficient for balancing the traffic at the forwarding layer of *VDisk*s, as they are unaware of and also lacks the ability to differentiate the decomposable burst and stable traffic. To this end, we propose a novel traffic forwarding scheme DiffForward for cloud block services. DiffForward differentiates the burst traffic from stable traffic in an accurate and efficient way at the client side, then it forwards the burst traffic to a decentralized distributed log store to realize real-time load balance by writing the data in a round-robin manner and balances the stable traffic by segmentation. DiffForward also judiciously coordinates the stable and burst traffic and preserves strong consistency under differentiated forwarding. Extensive experiments with reallife workloads on our prototype show that DiffForward effectively balances the traffic at the forwarding layer at a fine-grained subsecond level, thus significantly reducing the write latency of *VDisks*.

CCS Concepts: • **Information systems** → **Distributed storage**; • **Networks** → **Cloud computing**; • **Computer systems organization** → *Distributed architectures*.

Additional Key Words and Phrases: block storage service, traffic balancing

**ACM Reference Format:**
Wenzhe Zhu, Yongkun Li, Erci Xu, Fei Li, Yinlong Xu, and John C. S. Lui. 2023. DiffForward: On Balancing Forwarding Traffic for Modern Cloud Block Services via Differentiated Forwarding. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1, Article 15 (March 2023), 26 pages. https://doi.org/10.1145/3579444

[*]The work was done when Wenzhe Zhu was an intern at Alibaba.
[†]Yongkun Li is the corresponding author.

Authors' addresses: Wenzhe Zhu, wzhzhu@mail.ustc.edu.cn, University of Science and Technology of China, China; Yongkun Li, ykli@ustc.edu.cn, University of Science and Technology of China, China; Erci Xu, jostep90@gmail.com, PDL, China; Fei Li, renlei.lf@alibaba-inc.com, Alibaba Group, China; Yinlong Xu, ylxu@ustc.edu.cn, Anhui Province Key Laboratory of High Performance Computing, USTC, China; John C. S. Lui, cslui@cse.cuhk.edu.hk, The Chinese University of Hong Kong, China.


2476-1249/2023/3-ART15 $15.00
https://doi.org/10.1145/3579444

## 1 INTRODUCTION

Cloud block service, e.g., AWS EBS [4], Alibaba Block Service [1] and Azure Disk Storage [48], is a key storage infrastructure on the cloud, and it is being widely adopted to support diverse applications, e.g., virtual desktop [18, 66], big data processing [27, 73], databases [25, 28], web services [13, 33], and etc. For cloud users, cloud block service provides virtual block disks (VDisks) to enable them to seamlessly and elastically deploy their applications on the cloud. For cloud vendors, it allows them to realize more efficient resource provisioning by pooling storage resources.

Modern cloud block service often employs the layered architecture which consists of three layers, the *client layer*, the *forwarding layer* and the *storage layer* (see Figure 1(a) in §2.1) [75]. In the client layer, each *VDisk* has a stateless *thin* client, which fetches the block requests from the application/VM running in the *VDisk*, and then sends the requests to a specified proxy server in the forwarding layer. Proxy servers in the forwarding layer translate the received block requests to the lower-level reads/writes that can be processed by the storage layer, and then forwards them to the storage layer to realize actual storage on physical storage devices. Usually the storage layer implements a distributed storage to persist data, and it provides the forwarding layer as a simple abstraction of a large storage space. To forward a block request from a *VDisk* to the underlying storage layer, the proxy server maintains an AddressMap which maps the offset in the *VDisk* to the data location in the storage layer. One benefit of this architecture is that the I/Os in the storage layer can be well balanced by forwarding each *VDisk*'s data to multiple storage servers and devices at a fine granularity, e.g., at the granularity of data chunks [44, 71].

In this work, we first show the forwarding layer faces severe traffic imbalance, especially when we measure the traffic at subsecond time scale. For example, by analyzing the publicly available cloud block service workload [2], as the time interval of measuring traffic narrows down from 100s to 100ms, the CoV of the traffics among proxy servers increases from 0.38 to 1.01, indicating a significant imbalance aggravation when the timescale reaches subsecond level, and the imbalanced traffic also causes a severe latency degradation, e.g., the 99-percentile latency increases to 9.6× even when the average network bandwidth utilization is only 40%. The traffic imbalance is mainly caused due to the following two reasons. First, the traffic at the proxy servers are write-dominant, e.g., the write ratio can be over 70% of the overall traffic [40, 44, 67], this is because caching is often deployed at the application side and thus absorbs most reads [10, 26, 35, 41, 45, 52]. To avoid high concurrency control overhead, the *stateful* AddressMap corresponding to the same *VDisk* is not shared by multiple proxy servers, so a *VDisk* is bound to a specified proxy server, and thus the traffic from this *VDisk* must be forwarded by a particular proxy server. Second, traffics from different *VDisk*s are usually highly skewed [40], for example, for the cloud block trace [2], the top 2% write-intensive *VDisk*s generate over 40 times more requests than the average traffic over all *VDisk*s. Therefore, the proxy servers handling intensive *VDisk*s receive much more traffic and significantly increase the network queueing delay.

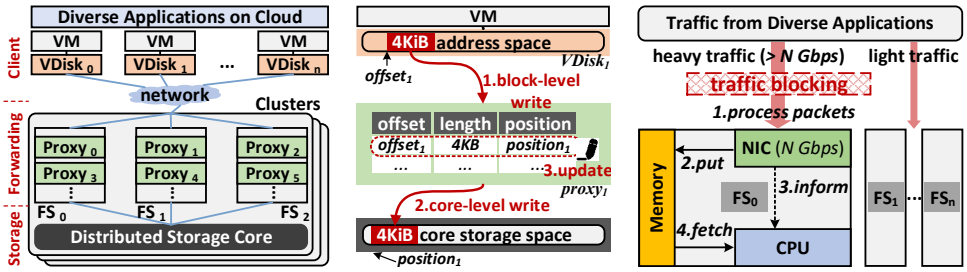
By thoroughly investigating the traffic of 1000 *VDisk*s from the publicly available cloud block service workload [2], we have a key observation on the traffic patterns. The traffic of *VDisk*s can be decomposed as stable traffic and burst traffic, and the coexistence of the two traffic patterns substantially exacerbates the traffic imbalance at a fine time granularity, e.g., subsecond time scale. We also find that the decomposable stable and burst traffic are mainly generated by two kinds of accesses to *VDisk*s. Specifically, as cloud block service inherently supports a wide variety of applications, different applications may simultaneously run on *VDisk*s. However, different applications may have different patterns of accessing *VDisk*s, thus having different spatial and temporal characteristics. In particular, interactive or real-time applications like e-commerce [33], IoT applications [11], and audio/video streaming [17, 34, 72] continuously append data to the

underlying storage like file systems (e.g., HDFS [56]) or KV stores (e.g., LevelDB [28] and Cassandra [14]), so they induce large sequential writes to *VDisk*s and thus make the traffic stable in a relatively long time interval (e.g., relative constant traffic rate), which we call *"stable traffic"*. On the other hand, batch processing tasks (e.g., MapReduce [27] and Spark [73]) are simultaneously executed to analyze collected data, and they consist of many parallel sub-tasks that persist intermediate results into temporary files at the end of each epoch, intermittently generating intensive random writes and leading to heavy traffic at a very short time period, which we call *"burst traffic"*.

Great efforts are also made to address the load balance problem in conventional storage systems, and they can be classified into three categories: (1) *segmentation* [7, 12, 23], which can be applied at the forwarding layer by dividing each *VDisk* into multiple small-size segments and using a separate proxy server to forward the traffic from each segment; (2) *migration* [21, 30, 39], which can be used to migrate the high-traffic *VDisk*/segment from heavy-loaded proxy server to light-loaded proxy server; (3) *replication* [5, 58, 61], which can also be applied by replicating each *VDisk*/segment with multiple replicas and distribute them on multiple proxy servers. However, these approaches are inefficient to address the traffic imbalance problem at the forwarding layer of cloud block service. The main reason is that the traffic at the forwarding layer can be decoupled as stable and burst traffic as they are induced by two different *VDisk* access patterns, while the above approaches are not aware of this traffic feature, so they do not leverage it for traffic balancing. Also, they lack the ability to differentiate burst traffic and stable traffic, so they all uniformly treat the aggregated traffic for traffic balancing. Experiments on our prototype also demonstrate the inefficiency of these approaches. For example, for the same workload mentioned above, at the time scale of 100ms, the coefficient of variation (CoV) of the traffics among proxy servers is 1.01, while it only decreases to 0.67 and 0.71 when using segmentation and migration, respectively. Despite replication balances traffic well, it induces expensive replica synchronization overhead to guarantee consistency, thus leading to a similar or even worse performance in latency compared with segmentation and migration (see §5.2 for details). Worse yet, we find that these approaches are also not suitable for balancing the burst traffic at the forwarding layer. Specifically, segmentation generates too many small segments and burdens the address map management at proxy servers, migration moves large amount of data and usually works only for stable traffic, and replication requires expensive synchronization among replicas to guarantee strong consistency required at the forwarding layer.

In this paper, we design a new traffic forwarding scheme DiffForward by leveraging differentiated forwarding, and also implement a prototype. DiffForward realizes efficient traffic balance at a fine-grained time granularity (e.g., at 100ms) at the forwarding layer of cloud block services. Specifically, for each *VDisk*, DiffForward first judiciously differentiates its burst traffic from stable traffic at the client side. Then the burst traffic is directed to decentralized distributed logs, which realize real-time balance by writing to proxy servers in a round-robin manner. For the stable traffic, it is balanced by leveraging the segmentation scheme which divides each *VDisk* into smaller segments with individual proxies. Meanwhile, DiffForward continuously merges data in logs back to the storage layer at background with a careful coordination with foreground writes to reduce the log size with low overhead. In addition, we also design a *lightweight* index at each *VDisk*'s client for DiffForward to help retrieve data from logs, while preserving strong consistency under differentiated traffic forwarding. Our contributions are summarized as follows.

- We analyze the publicly available real-world workload traces and demonstrate the consistent imbalance of traffics at the forwarding layer of cloud block services. We also investigate the inefficiency of three commonly used load balancing approaches in distributed storage systems,




(a) Three-layer Architecture (b) Write Path (c) Blocking by Heavy Traffic

Fig. 1. Illustration on the architecture of a cloud block service and the detailed write path.

and analyze the special access patterns of *VDisk*s to reveal the reasoning of burst traffic and motivate our new traffic forwarding design.

- We design DiffForward, which realizes differentiated traffic forwarding via several techniques: (i) accurate *burst traffic detection* at the client side of *VDisk*s, (ii) a decentralized *distributed log store* to achieve real-time balance of burst traffic, (iii) *asynchronous log merging* that is well coordinated with foreground traffic, and (iv) a *lightweight client-side index* to help read data from logs and preserve strong consistency.
- We implement a prototype of a cloud block service by following the three-layer architecture and integrating the differentiated forwarding scheme. We implement both the client layer and forwarding layer from scratch, and leverage Ceph Rados [70] as the underlying distributed storage. Experiments show that DiffForward outperforms all the three categories of existing load balancing approaches. In particular, it decreases the CoV by up to 65%. It also reduces the average write latency by up to 44% and improves the 99-percentile write latency by up to 78% under various network configurations and workloads.

The source code of DiffForward is at https://github.com/wzhzhu/DiffForward.

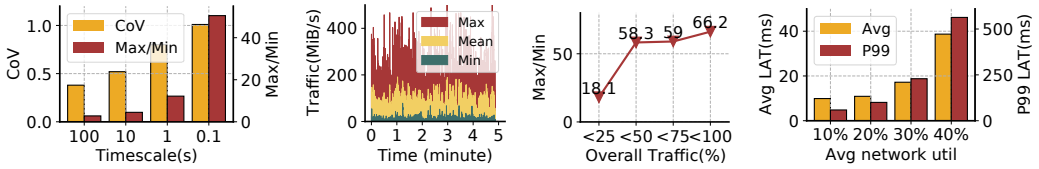
## 2 BACKGROUND AND MOTIVATION

In this section, we first introduce the three-layer architecture of modern cloud block services (§2.1), and analyze the traffic imbalance issue at the forwarding layer (§2.2). Then we introduce existing load balancing approaches and analyze their limitations for balancing the forwarding traffic (§2.4). Finally, we motivate our design by analyzing the traffic patterns at the forwarding layer (§2.3).

### 2.1 Cloud Block Service Architecture

Cloud block service provides computing instances (VMs rent by cloud users) with *VDisk*s to fulfill the storage need of diverse applications on the cloud. Figure 1(a) depicts the general architecture of a cloud block service, which is composed of three layers, i.e., *client layer*, *forwarding* layer, and *storage layer* [75]. In the client layer, the *VDisk*s' clients simply take block requests from VMs and deliver them to proxy servers. In the forwarding layer, as illustrated in Figure 1(b), each proxy server maintains a local AddressMap, which records the mapping from the offsets at a *VDisk* to the data location in the storage layer. When a proxy server receives the block-level requests from client, it first translates them into low-level reads/writes according to the local AddressMap, then forwards them to the storage layer for data persistence. The storage layer consists of a distributed storage core, which is commonly a distributed file system or an object store, and it provides a unified storage space for storing the data from all *VDisk*s.

Note that the benefits of integrating a forwarding layer are two fold. First, as the processing of the block-level requests from VMs and the stateful metadata are pushed down to the forwarding




(a) Imbalance Degree (b) Traffic Dynamics (c) CDF (d) Latency

Fig. 2. Traffic imbalance among proxy servers and its impact on latency.

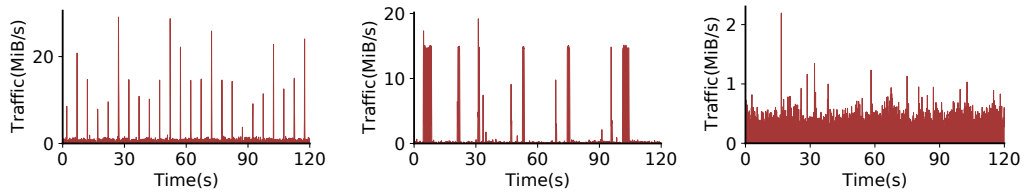
layer, the clients can be simplified so VMs can quickly reconnect to *VDisk*s after VM migration, thus supporting seamless computing resource scaling on the cloud [32, 51, 55]. Second, since the AddressMaps record the actual data positions in the storage layer, the data from *VDisk*s can be evenly spread across storage servers, thus the balance of I/O loads in the underlying storage can be well realized [44, 71]. However, we point out that if the traffic to a proxy server becomes too heavy, then the traffic may be blocked due to the limited network bandwidth and processing power of the NIC at the proxy server. This is because the NIC at the proxy server has to process every request by first writing data to memory and then informing the CPU to fetch data to finish the following steps of traffic forwarding (see Figure 1(c)).

While substantially optimizing its architecture to offer more great features, cloud block service retains a block-level compatible interface and strong consistency as traditional physical block devices, so as to allow server-based applications to seamlessly migrate to the cloud. Albeit some research work has explored the possibility of improving performance by relaxing some of block service's semantic guarantee [47], in this paper we opt to preserve these requirements, such as strong consistency, which are essential especially for large-scale public clouds.

## 2.2 Imbalance of Forwarding Traffic

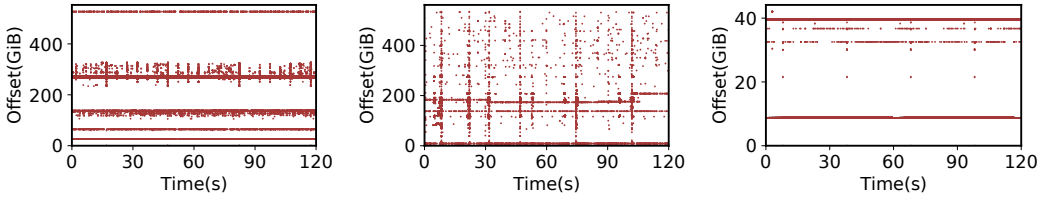
The three-layer architecture brings a new traffic imbalance issue in the critical forwarding layer. Since proxies at the forwarding layer are stateful, requests of a *VDisk* must be delivered to the fixed proxy that stores its AddressMap, instead of an arbitrary proxy as in conventional stateless services like Nginx [59]. Considering that *VDisk*s support diverse applications with very different traffic intensities and patterns, traffics at different proxy servers are usually unevenly distributed. Besides, we note that the traffic imbalance at the forwarding layer is mainly caused by small writes, For example, write traffic may take up 75% of the total traffic, and 75% of writes are no larger than 16 KiB [40]. This is because reads are mostly absorbed by the widely-used cache at the VM side [10, 26, 45, 52], and large I/Os are also split by *BIO splitting* in the block layer of VM kernels.

To further demonstrate the traffic imbalance at the forwarding layer, we analyze large-scale production trace (see §2.3 for details). Figure 2(a) first shows the coefficient of variation (CoV) and Max/Min of the traffics at different proxy servers. We see that as the length of time interval changes from 100s to 100ms, CoV increases from 0.38 to 1.01, implying very severe traffic imbalance among servers, while the Max/Min increases from 2.8 to 50.5, meaning that the most-loaded server has over 50 times more traffic than the least-loaded one. Figure 2(b) further shows the dynamics of three traffic patterns in 5 minutes, i.e., the traffic of the most-loaded server (Max), the average traffic over all servers (Mean), and the traffic of the least-loaded server (Min). We see that traffic spikes are very common if we observe at a small time granularity (e.g., 100 ms), especially for the traffic at the most-loaded proxy server. Besides, as the traffic spikes from different *VDisk*s may appear simultaneously, the imbalance is more serious in high-traffic intervals than low-traffic intervals, e.g., as shown in Figure 2(c), the traffic in the case when the ratio between the maximum traffic and the minimum traffic is larger than 58 accounts for more than 50%. Due to the traffic imbalance, traffic




(a) VDisk1 Traffic (b) VDisk2 Traffic (c) VDisk3 Traffic

Fig. 3. Traffic patterns over a long time period: stable traffic and burst traffic coexist.




(a) Accessed Addresses on VDisk1 (b) Accessed Addresses on VDisk2 (c) Accessed Addresses on VDisk3

Fig. 4. The distribution of accessed addresses for three typical kinds of VDisks.

at heavy-loaded servers may be blocked at the network, thus increasing the latency. To show the impact of traffic imbalance on latency, we further investigate the average and P99 tail latency under different network utilizations. Figure 2(d) shows that both the average and P99 latency severely increase when network utilization increases because of processing the heavier traffic.

Since block service is widely adopted to support latency-critical applications, e.g., interactive web applications, its prolonged latency directly impairs the fluidity and predictability of applications' response time [58]. Moreover, as requests from these applications are often broken into sub-requests whose responses are aggregated to produce the ultimate results, even temporary latency spikes may significantly degrade end-to-end latency [3]. All these necessitate effective traffic balancing for block service to improve its average and P99 tail latency.

### 2.3 Traffic Analysis and Key Findings

To explore the potentials of improving traffic balance at the forwarding layer, we investigate a real-world trace to reveal the key features of *VDisk* traffic patterns, and then present our key findings on the reasons resulting in the *VDisk* traffic patterns.

**Coexistence of stable traffic and burst traffic.** We find that the traffic of each *VDisk* over a long time period can be decoupled as *stable traffic* and *burst traffic*. For example, Figure 3(a) - 3(c) show the traffic patterns in 2 minutes of three *VDisk*s from a publicly available real-world trace (Alibaba Block Trace [2]). Note that the traffic is observed at a fine time granularity, e.g., 100ms, that is, each vertical line in the figure represents the total traffic in a 100ms interval. We can see that severe traffic spikes commonly exist, while the traffics can be classified into two categories, one is the stable traffic which has a similar size in different time intervals, and the other is the burst traffic which corresponds to the large spikes shown in the figures.

**Observation.** The coexistence of stable and burst traffic comes from the fact that applications running on *VDisk*s mainly exhibit two different access patterns. On the one hand, interactive or real-time applications like e-commerce [33], IoT applications [11], and audio/video streaming [17, 34, 72] continuously append data to the underlying storage like file systems (e.g., HDFS [56]) or KV stores (e.g., LevelDB [28], Cassandra [14]). On the other hand, batch processing tasks (e.g., MapReduce [27] and Spark [73]) are simultaneously executed to analyze collected data, which consist of parallel sub-tasks that persist intermediate results into temporary files at the end of each

epoch, intermittently generating intensive random writes. The above two different accesses lead to two different patterns of traffic at each *VDisk*. In particular, if sequential writes are dominated in a time interval, then the traffic size is usually stable, and thus generating *stable traffic*. In contrast, if a large amount of random writes are issued in a time interval, then the traffic in this interval may become very heavy, thus generating *burst traffic*.

Our investigation on the publicly available real-world trace (Alibaba Block Trace [2]) also validates the above understanding. For example, Figure 4(a) shows the visited offsets of a particular *VDisk* (denoted as $VDisk_1$) at 100ms time scale during a 2-minute period. We see that sequential writes continuously visit small address ranges (represented by the horizontal lines in the figure), random writes also occasionally appear, and they visit across a large address range (represented by the vertical lines). As a result, these two access patterns generate the stable and burst traffic, respectively, as shown in Figure 3(a). We also study the access patterns of other *VDisk*s, and observe a similar conclusion. In the interest of space, we show only two typical *VDisk*s in Figure 4(b) and Figure 4(c), and their corresponding traffic patterns are shown in Figure 3(b) and Figure 3(c), respectively. Furthermore, for all *VDisk*s, if we use $S$ and $P$ to represent the traffic size and the proportion of random writes during each time interval, then we find that the average Spearman Coefficient of $S$ and $P$ lies at ~0.61, indicating strong correlation between burst traffic and intensive random writes, which further validates our finding.

### 2.4 Existing Balancing Approaches and Their Limitations

Existing load balancing approaches can be classified into three classes [7, 12, 21, 23, 30, 39, 47, 58]. In the following, we first summarize their key ideas, then study their effect on balancing the traffic at the forwarding layer, and finally analyze the key reasons of their inefficiency.

**Segmentation.** The approach of segmentation divides each *VDisk*'s address space into $n$ smaller *segments*, then uses $n$ proxies, which are randomly placed at different proxy servers, by assigning each *segment* with a separate proxy. By doing this, the heavy traffic of a *VDisk* is distributed to multiple proxies at different servers, thus benefiting the load balance. In practical implementation, *striping* is widely used in splitting the address space, thus each segment may consist of multiple small address spaces uniformly scattered in the whole address space. Specifically, each *VDisk*'s address space is first logically divided into very small strips, e.g., 64KiB, then all strips are assigned to segments in a round-robin manner [47]. For example, suppose that a 1TiB *VDisk* is split into four 256GiB segments, say $segment_0$ ~ $segment_3$, then the four small address ranges 0 ~ 64KiB, 64 ~ 128KiB, 128 ~ 192KiB, and 192KiB ~ 256KiB are assigned to $segment_0$ ~ $segment_3$, respectively. Thus, sequential access (e.g., 0 ~ 256KiB) will be balanced across segments placed at different servers.

**Migration.** The key idea of migration is to predict the traffic of *VDisk*s/segments and then move high-traffic *VDisk*s/segments from heavy-loaded servers to light-loaded servers, so as to balance the traffic. The performance of migration strongly depends on the prediction accuracy. Usually, two prediction strategies are widely adopted. The first strategy is to simply use the average traffic during the last time interval as a prediction in the next interval. The second strategy uses the method of Exponentially Weighted Moving Average (EWMA) by considering multiple historical intervals [20], and it can be formulated as $P_{t+1} = EWMA_t = \alpha \cdot r_t + (1-\alpha) \cdot EWMA_{t-1}$, where $P_{t+1}$ is the predicted value for interval $t+1$, $\alpha$ is a weight factor and $r_t$ is the real value at interval $t$.

**Replication.** Replication leverages the benefit of "power of $d$ choices" to balance load. It first replicates the proxy of a *VDisk*/*segment* into $d$ replicas with exactly the same AddressMap, and locates them at different servers. Then each *VDisk*/segment records the position of its $d$ replicas at the client. Upon receiving a request from VMs, the client selects the "least-loaded" one out of the $d$ proxy replicas to forward the request. In terms of selecting the proxy replica, we focus on the state-of-the-art policy developed in C3 [58], which can be described as follows. For each server

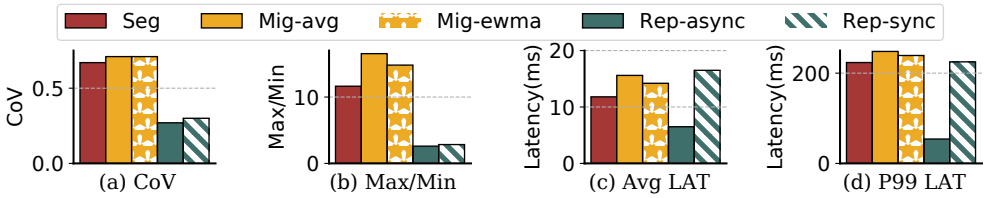



Fig. 5. Inefficiency of existing approaches.

$s$, a client maintains a counter $\overline{q}_s$ that represents the number of all requests waiting at $s$, and an instantaneous count of its own outstanding requests $o_s$ (the requests without receiving responses yet). Then the client estimates the queue size ($\hat{q}_s$) of each server as $\hat{q}_s = 1 + os_s \cdot w + \overline{q}_s$, where $w$ is a weight factor which is usually set as the number of concurrent clients. Finally, the client will select the replica that has the minimum queue size $\hat{q}_s$.

**Effect on traffic balance.** To investigate the effect of existing approaches on balancing the traffic at forwarding layer, we implement them in our prototype of a cloud block service and evaluate their performance. We set the segment size as 64GiB to limit the overhead at the client side, and set the time scale as 100ms as the traffic spikes becomes severe at this time scale without load balancing. For replication, we adopt chain replication [65], which updates the AddressMap from the replica at head to tail before acknowledging a write request (see §5.4 for details of other settings). Figure 5(a) and 5(b) show the imbalance metrics of CoV and Max/Min, and Figure 5(c) and 5(d) show the average and P99 latency. We see that the traffic is still very unbalanced for both segmentation and migration, e.g., the CoV is still up to 0.67 and 0.71, and the maximum traffic is still more than 10× of the minimum traffic. Replication performs much better in terms of traffic balance, but it only has a similar P99 latency and even worse average latency, because of the large overhead of synchronizing the AddressMap required by strong consistency at the forwarding layer.

**Inefficiency analysis.** Note that existing approaches are inefficient to address the traffic imbalance problem at the forwarding layer, because they are unaware of the decomposable stable and burst traffic, but simply treat all the accumulated traffic in the same way. Furthermore, as these approaches also lack the ability to accurately differentiate the burst traffic from stable traffic, it is also not a good choice to directly apply them to address the traffic imbalance problem at the forwarding layer. Specifically, for segmentation, balancing the burst traffic needs to divide the *VDisk* into very small segments, as the capacity of each *VDisk* can be hundreds of gigabytes, using too small segment size inevitably generates lots of segments and introduces high overhead of metadata management at the client side. However, complicated design at clients should be avoided as it prohibits scalability and also eliminates the benefits of the three-layer architecture of cloud block services. On the other hand, using large segment size can reduce the metadata management overhead, but it limits the benefit of balancing burst traffic. For migration, it is usually used by assuming to have a stable traffic, while facing challenges to deal with instantaneous traffic spikes. One reason is that it is hard to accurately predict highly changing traffic, especially at a small time granularity. For example, for the workload we studied in §2.3, even if we set the time interval of traffic prediction as 5s, the relative error between the estimated traffic and the real value is still larger than 100% even using the sophisticated method EWMA. This implies that the prediction is even not better than a random guess. If we reduce the length of prediction interval to 100ms, which is comparable to the time granularity of observing traffic spikes at the forwarding layer, then the relative error increases to 200%, making it impractical to perform migration even the migration overhead could be completely ignored. Finally, for replication, it must introduce a large synchronization overhead, as the forwarding layer manages the metadata, e.g., the AddressMap, thus requiring strong consistency.

### 2.5 Main Idea and Key Challenges

**Main idea.** As stable and burst traffic patterns coexist and induce the traffic dynamics, uniformly processing the coexisted traffic is challenging to realize a good balance. Our main idea is to decouple these two kinds of traffics at a small time granularity, e.g., 100ms time scale, then we can leverage different designs to balance the stable and burst traffics more efficiently. For example, stable traffic can be well balanced with segmentation, while still limiting the overhead at the client side. As for the burst traffic, as long as its total size is not large, it is also much easier to develop a specialized new forwarding design to realize good balance with low overhead, and we leverage distributed logs to evenly distribute the burst traffic across proxy servers.

**Challenges.** However, to realize the idea of differentiated forwarding, we face multiple key design challenges, which are summarized as follows.

- **Differentiating burst traffic with high accuracy and low overhead.** To leverage differentiated forwarding to balance burst traffic and stable traffic separately, the first key challenge is to accurately identify the burst traffic in a timely manner. Furthermore, as the traffic must be differentiated before sending to proxy servers, the identification can only be executed at clients, thus requiring a very lightweight design, so as to keep the clients of *VDisk*s being agile and flexible. Therefore, we must fully exploit the access patterns of *VDisk*s to design a simple yet effective traffic differentiation algorithm to achieve high accuracy and low overhead simultaneously.
- **Efficient design of distributed logs.** Burst traffic can be well balanced by leveraging distributed logs. But it not only needs to achieve traffic balance at subsecond time scale under highly concurrent *burst traffic* from numerous *VDisk*s, but also should avoid potential write conflicts to ensure high performance of concurrent writes from different *VDisk*s. Besides, it is also required to provide a clean and compact data layout in the log so as to efficiently serve subsequent read operations. Naively distributing the *burst traffic* among proxy servers (e.g., random) is hard to satisfy these requirements simultaneously. Thus, both the placement of log entries and the log structure need to be carefully designed to fulfill the above goals of burst traffic balancing, concurrent writes and efficient reads.
- **Fast log merging while minimizing network contention.** The distributed logs introduce extra overhead, including the storage consumption and extra indexing overhead to retrieve data from logs. So the logs can only be used as a temporary store to process the *burst traffic*, and need to be finally merged into the underlying *storage layer* so as to reduce the log size. However, log merging consumes extra network bandwidth, naively merging the logs could incur severe contention with foreground requests, prolonging *VDisk*s' latency. While limiting the merging speed at a low rate could mitigate the contention, it easily leads to extremely large logs. To address this issue, we must carefully coordinate the log merging procedure with foreground requests, so as to timely merge logs and minimize the network contention.
- **Rapidly retrieving fresh data from distributed logs.** Despite logs are write-friendly for their append-only feature, searching data from them is fairly time consuming. Therefore, we must maintain an efficient index at the clients to help rapidly retrieve data from distributed logs. However, one subtle thing is that with logs being continuously merged to the underlying storage layer, the locations of some data might be altered without being perceived by the index, leading to accumulation of outdated records, which not only face the risk of returning expired data but also multiply the index's size. To solve this problem, an efficient index at the clients must be developed, and it must timely remove outdated records as distributed logs are merged, so as to keep the index being lightweight and ensure consistency.

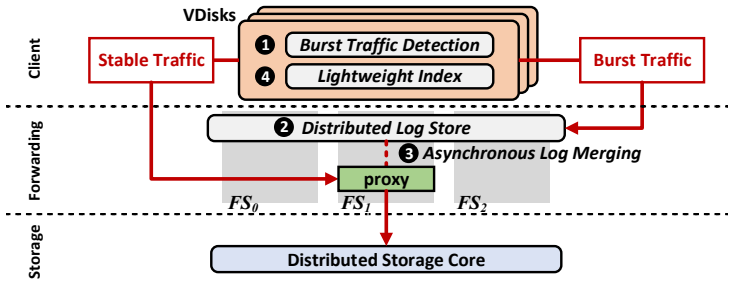



Fig. 6. DiffForward overview

## 3 DESIGN OF DIFFFORWARD

We present DiffForward, a novel forwarding scheme for cloud block service, which differentiates *stable traffic* and *burst traffic* to balance the workloads in *forwarding layer*. We first introduce its overall architecture (§3.1) and then elaborate the specific techniques (§3.2-§3.5).

### 3.1 Overview

As shown in Figure 6, DiffForward mainly consists of four components, *burst traffic detection*, *distributed log store*, *asynchronous log merging* and *lightweight client-side index*.

- **Burst traffic detection.** DiffForward judiciously differentiates *burst traffic* from *stable traffic* at each *VDisk* client by identifying the traffic intensity and randomness of accessing *VDisk* address. With a carefully designed algorithm, it could detect *burst traffic* with high accuracy while incurring small CPU and memory consumption.
- **Distributed log store.** DiffForward migrates burst traffic to a *distributed log store* residing in all forwarding servers to balance workload in real time. Its log structure is judiciously designed to ensure writing efficiency under highly concurrent *VDisk*s, so as to support efficient point query (fetching target data from the log) and range query (traversing the log).
- **Asynchronous log merging.** DiffForward asynchronously gathers logs from *distributed log store* via range query, and merges them to the underlying *storage layer*. During this, DiffForward finely coordinates background and foreground traffic to avoid network contention and to fully utilize idle network bandwidth to minimize the log size and management overhead.
- **Lightweight client-side index.** DiffForward maintains an index at each *VDisk*'s client to promptly retrieve data from logs via point query. The index also acts as a guide to reschedule *VDisk* writes to preserve strong consistency under DiffForward's *differentiated forwarding*. Besides, the index automatically trims outdated nodes to keep it lightweight.

### 3.2 Burst Traffic Detection.

There are two reasons to realize the *burst traffic* detection. One is to promptly discover the arrival of overloaded traffic on forwarding servers so as to trigger the traffic migration at the client side of each *VDisk*/*segment*; The other is to differentiate the traffic into *stable traffic* and *burst traffic*, and then only migrate the challenging *burst traffic*. Moreover, the algorithm should be accurate at small time granularity (e.g., at 100 milliseconds), and induce small CPU and memory overheads as well.

Recall that *burst traffic* and *stable traffic* respectively correlate with the two representative access patterns we introduced in §2.3, which exhibits different spatial and temporal characteristics, i.e. distinct traffic load and randomness. DiffForward exploits this to detect the *burst traffic* out of each *VDisk*'s traffic. Specifically, for each *VDisk*, we maintain a *Virtual Request Queue* (*VRQ*), and use its length, *QLEN*, to estimate the incoming write traffic intensity in real time. Suppose that the write request sequence of a $VDisk_j$ is $r_0$, $r_1$, $r_2$,..., where $r_i$ arrives at $VDisk_j$ at time $t_i$ and is to write

**Algorithm 1:** Burst Traffic Detection

**Input** : Current write request $r_i$; Current time: $t_i$;
**Output**: Whether write request $r_i$ induces *stable* traffic or *burst* traffic

1 $QLEN \leftarrow QLEN - R \cdot (t_i - t_{i-1})$; // shorten VRQ
2 $QLEN \leftarrow \min(QLEN, 0)$; // Limit QLEN $\geq$ 0
3 $t_{i-1} \leftarrow t_i$; // update time
4 **if** $r_i$ is a *random* request **then** $W_i \leftarrow W_{rand}$ **else** $W_i \leftarrow W_{seq}$; // assign weight to $r_i$
5 $QLEN \leftarrow QLEN + W_i \cdot l_i$; // lengthen VRQ
6 **if** $QLEN > TH$ **then**
7 $\quad QLEN \leftarrow TH$; // cap QLEN to TH
8 $\quad$ return *burst*; // $r_i$ is identified as burst
9 **else**
10 $\quad$ return *stable*; // $r_i$ is identified as stable

data of length $l_i$ to offset $off_i$. When $r_i$ arrives at a $VDisk$, we insert it to $VRQ$ by updating $QLEN$ in three steps. (1) Shorten the queue according to the $VDisk$'s average traffic rate (Algorithm 1 line 1), formulated as

$$QLEN = QLEN - R \cdot (t_i - t_{i-1}), \tag{1}$$

where $R$ is the average traffic rate, thus $R \cdot (t_i - t_{i-1})$ is the expected length of the traffic processed within time interval $[t_{i-1}, t_i]$. (2) Weighing requests according to their spatial pattern (line 4). We compare $r_i$'s offset with previous $n$ requests, and define its *randomness* $ran_i$ as

$$ran_i = min_{j=0}^{n-1} |off_i - off_{i-j}|. \tag{2}$$

If $ran_i$ is greater than a pre-defined threshold $T_{ran}$, we consider it as a *random write*, which is the main source of *burst traffic* according to our observation, and then assign a higher weight $W_{rand}$ to $r_i$; Otherwise, we assign a lower weight $W_{seq}$ to $r_i$ (Algorithm 1 line 4). In our experiments, $n$ and $T_{ran}$ are set to 32 and 128KiB respectively, similar to previous works [40, 60], and $W_{rand} = 2 \cdot W_{seq}$. (3) Lengthen the queue due to $r_i$'s arrival (line 5), formulated as

$$QLEN \leftarrow QLEN + W_i \cdot l_i. \tag{3}$$

As a result, when *burst traffic* starts to arrive at a *VDisk*, due to its high randomness and traffic load, $QLEN$ of the $VRQ$ will be quickly accumulated up for requests' higher weights (step 2) and more intense queue lengthening (step 3). Once the $QLEN$ is greater than a predefined threshold $TH$, the detection algorithm would confirm the arrival of *burst traffic* (line 6-8), and inform to launch the *burst traffic* balancing procedure (§3.3). Note that $QLEN$ is reset to $TH$ each time when a burst traffic is detected (line 7), so as to switch its output back to *stable* once the current traffic spike has ended. By contrast, *stable traffic* could pass through the $VRQ$ without being detected (line 10). According to our experiments (see §5.5), the average recall and precision of the *burst traffic detection* algorithm are about 80% and 90%, respectively.

### 3.3 Distributed Log Store

DiffForward adopts a *distributed log store* deployed atop forwarding servers to balance *burst traffic* at sub-second timescales. Suppose that there are $n$ forwarding servers, denoted as $FS_0$, $FS_1$, ..., $FS_{n-1}$. For each $VDisk_i$, we allocate a log $log_i$ which is distributed among all forwarding servers, where $log_{i,j}$ is the sub-log of $log_i$ on $FS_j$. When a *burst traffic* is detected, consisting of $m$ write requests $r_0$, $r_1$, ..., $r_{m-1}$, each of the requests will be assigned an incremental serial number, then directed to forwarding servers in a round-robin manner. Consequently, $FS_j$ receives $r_j$, $r_{j+n}$, $r_{j+2n}$, ..., then appends them to local $log_{i,j}$ as log entries $entry_j$, $entry_{j+n}$, $entry_{j+2n}$, ..., and finally acknowledges the writes. For example, Figure 7 shows how the log entries of $VDisk_0$ (appended to $log_0$) and

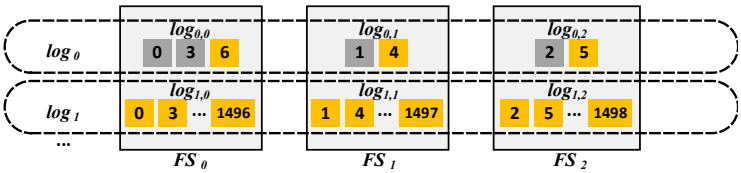



Fig. 7. Distributed placement of log entries.

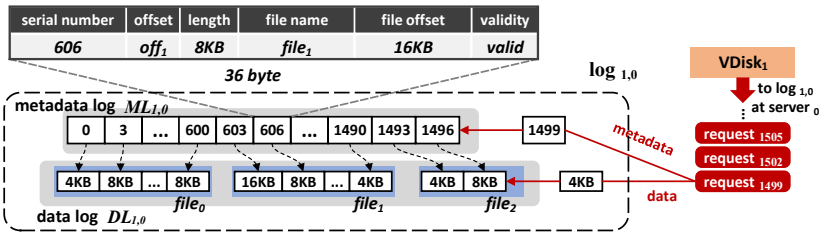



Fig. 8. **Distributed log structure.** This figure shows how $log_{1,0}$ (sub-log of $log_1$ at server $FS_0$) is organized.

$VDisk_1$ (appended to $log_1$) are assigned when the forwarding server number $n$ is set to 3, where yellow rectangles represent valid entries and grey rectangles represent outdated entries caused by *asynchronous log merging* (§3.4). The numbers on each log entry mark their assigned serial numbers. By doing this, each *VDisk*'s *burst traffic* can be finely balanced across forwarding servers, so as the overall *burst traffic*. In our experiments, the CoV and Max/Min of the overall *burst traffic* can be as low as 0.13 and 1.63, respectively.

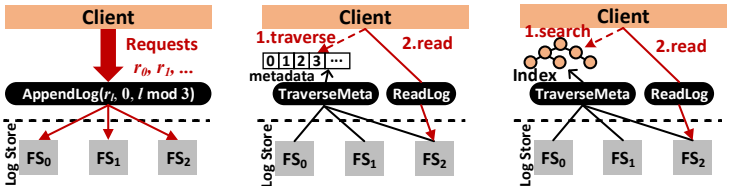
**Distributed log structure.** *distributed log store* manages *data* and *metadata* of received write requests, where *data* is the user-written raw data, while *metadata* is the attached descriptive information. As Figure 8 shows, for each forwarding server, say $FS_j$, upon receiving a write request from $VDisk_i$, $FS_j$ first appends the *data* into a data log $DL_{i,j}$, then packs the *metadata* into a fixed-sized byte string and appends the string to metadata log $ML_{i,j}$ (e.g., the *metadata* fields in Figure 8). $DL_{i,j}$ is composed of multiple local files with a maximum capacity (1MiB), where data are continuously being appended. Once a file in $DL_{i,j}$ is full, it will be sealed and replaced by a new empty data log file for appending the coming *data*. The sealed data log files will be timely removed once the logged entries in it have been merged into *storage layer*, and their storage spaces are reclaimed (see details in §3.4).

Consequently, for $VDisk_i$, its $DL_{i,j}$ and $ML_{i,j}$ at server $FS_j$ together form its sub-log $log_{i,j}$, and sub-logs at all servers constitute the entire $log_i$. Since metadata logs and data logs of different *VDisk* are disjoint (e.g., $log_0$ and $log_1$ in Figure 7), intensive *burst traffic* from multiple *VDisks* can independently and concurrently written to *distributed log store* without location conflicts [8, 9, 69], so as to enhance its writing efficiency.

**Distributed log write & read.** The *distributed log store* provides the following APIs.

- `AppendLog`(*write request r*, *log-id i*, *server-id j*) appends $r$ to $log_i$'s sub-log $log_{i,j}$ on server $s_j$.
- `ReadLog`(*log-id i*, *server-id j*, *range R*) reads log data of $VDisk_i$ from range $R$ in data log $DL_{i,j}$.
- `TraverseMeta`(*log-id i*, *serial-number n*) gathers the *metadata* of log entries whose serial number $\geq n$ from $log_i$. It first notifies all servers to scan their local metadata log $ML_{i,j}$ ($0 \leq j \leq n-1$) and return the target metadata, then sorts the gathered metadata locally by their serial numbers and completes the operation.

This group of APIs work as the building blocks for more complex operations as follows. First, based on `AppendLog`, DiffForward could easily realize the round-robin traffic balancing strategy.




(a) Traffic balancing (b) Range query (c) Point query

Fig. 9. Distributed log write & read.

Suppose there are $n$ servers and $m$ write requests $r_0$, $r_1$, ..., $r_{m-1}$ of $VDisk_i$, DiffForward invokes `AppendLog`($r_l$, $i$, $l$ mod $n$) for each $r_l$ ($0 \leq l \leq m-1$) to balance the traffic (Figure 9(a)). Second, by combining `ReadLog` and `TraverseMeta`, DiffForward could simultaneously realize range query (traverse log entries by order) to facilitate *Distributed Log* merging (§3.4), and prompt point query for *VDisk* clients (retrieve data by *VDisk* offset) with the help of *lightweight client-side index* (§3.5). Specifically, to traverse log from a specific position, suppose $log_i$ from serial number $N$, DiffForward first collects sorted metadata by `TraverseMeta`($i$, $N$), and then traverse the $log_i$ accordingly by invoking `ReadLog` (Figure 9(b)). Meanwhile, to retrieve data by given *VDisk* offset from the log, each *VDisk* client firstly gathers metadata via `TraverseMeta`, then caches and organizes the metadata into an tree-structured index (§3.5), so as to search target data from logs promptly (Figure 9(c)). We like to note that during the above reading processes, benefiting from the separate management of *data* and *metadata*, the critical `TraverseMeta` operation can be quickly done without prohibitively expensive log *data* scanning.

### 3.4 Asynchronous Log Merging

Large-scale logs induce high storage overhead, and also complicate the indexes at *VDisk* clients. Therefore, DiffForward regularly merges distributed logs into *storage layer* in the background.

**Log merging procedure.** Each forwarding server, say $FS_j$, continuously merges logs of *VDisk*s, say $VDisk_1$, $VDisk_2$,..., whose forwarding proxy is located at $FS_j$, with the following four steps.

(1) $FS_j$ collects and caches the newest log *metadata* of $VDisk_1$, $VDisk_2$,..., by invoking `TraverseMeta`.
(2) According to local *metadata*, $FS_j$ selects current longest log, say $log_i$, then pulls log entries from its head, i.e., which with the smallest serial number of all valid log entries.
(3) $FS_j$ transforms log entries in $log_i$ back to write requests, and submits them to $proxy_i$ in their original written order.
(4) $FS_j$ invalidates merged log entries in *distributed log store*, and returns to step 2.

For those log entries invalidated during the process, its *state* field in corresponding metadata log is changed to *outdated*, and waits to be removed from *distributed log store* (e.g., in Figure 7, $VDisk_0$'s has appended 11 entries to $log_0$, among them, $entry_0$ to $entry_3$ have been invalidated). To reclaim storage space from those invalidated log entries, $FS_j$ launches a garbage collection (GC) thread to periodically scan local metadata logs. If the *state* fields of all log entries in a sealed data log file are *outdated*, the GC thread will remove it.

**Prioritized traffic coordination.** When $FS_j$ merges logs, pulling log entries from *distributed log store* (step 2) and merging log entries to *storage layer* (step 3) will interfere the foreground requests in a *VDisk*, leading to longer latency to *VDisk*s. To guarantee low latency of the foreground requests, DiffForward assigns a higher priority to the foreground requests than the merging operations.

As Figure 10 shows, to enable $FS_j$ to perceive foreground traffic timely, before sending a request, each *VDisk* client will attach its local queue-depth *LQD* to the message head. Then, by summing up all received *LQD*s, $FS_j$ could maintain a global queue-depth *GQD*, which represents the number of

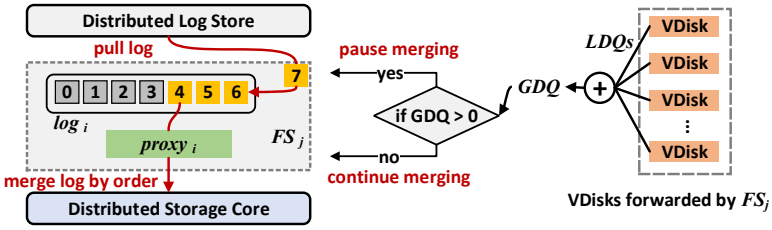



Fig. 10. **Asynchronous log merging.** $FS_j$ is merging $log_i$, and it is now processing entry 4.

requests being forwarded or waiting to be forwarded by $FS_j$. Upon receiving a foreground request, $FS_j$ uses the attached *LQD* to update *GQD*, to keep pace with the real-time foreground traffic status. $FS_j$ in turn controls the log merging procedure based on its *GQD*:

- Case 1: $GQD > 0$. $FS_j$ suspends log merging and yields network bandwidth to foreground traffic, so as to ensure the SLA of latency-critical *VDisk*s.
- Case 2: $GQD = 0$. $FS_j$ continues log merging, so as to fully utilize the idle network bandwidth to reduce log size.

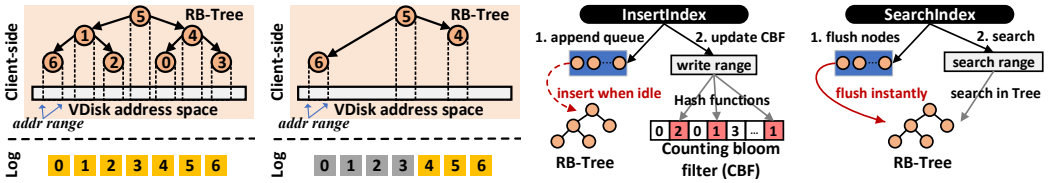
Owing to the timely log merging, DiffForward allows cloud block service to enjoy the benefits of *distributed log store* on balancing traffic, while minimizing its overhead. In our experiment, the average log size can be kept at ~3 MiB per log in average even under heavy foreground traffic. And the latency degradation to *VDisk*s caused by log merging is fairly small (13% and 47% increase in average and P99 latency, respectively).

### 3.5 Lightweight Client-side Index

Since part of *VDisk*'s data is appended to $log_i$, DiffForward keeps a red-black tree $RBT_i$ at each *VDisk*'s client to retrieve data from $log_i$ based on given *VDisk* offsets. In $RBT_i$, each node corresponds to an entry in $log_i$ by recording its *metadata*, and all nodes are sorted by the their address range. Before sending a write to *distributed log store*, *VDisk*'s client will insert a node into $RBT_i$ to record the write, and disables overlapping address range in $RBT_i$ to keep all nodes' total order. To serve reads to $VDisk_i$, suppose to read address range $R = [off, off + len)$, the client first searches $RBT_i$ to check if there is overlapping node with range $R$. If there is, the client gets the target data from *distributed log store* according to the log position recorded by these nodes (data log name and offset); Otherwise, the client submits the read to its proxy to get data from *storage layer* directly.

**Index trimming to keep index lightweight.** To trim outdated nodes timely, we add a pointer `next` to each node in $RBT_i$, which points to the node with the next serial number. Then all nodes in $RBT_i$ can be orderly linked as a chain. We also maintain a pointer `index_head` to point to the node with the smallest serial number. When performing index trimming, we monitor the current log head of $log_i$, i.e., the smallest serial number of all *valid* entries in $log_i$. Suppose it's `log_head`, then we repeatedly compare `index_head`'s serial number with `log_head`, as long as it's less than `log_head`, we delete the node pointed by `index_head` and move `index_head` one step forward along `next`. For example, in Figure 11(a), `index_head` initially points to $node_0$ and `log_head` is 0. Assume $entry_0$ to $entry_3$ are merged and invalidated, changing `log_head` to 4, then the index will trim $node_0$ to $node_3$ in turn, making `index_head` equal to `log_head` (Figure 11(b)).

**Writes rescheduling to preserve consistency.** With DiffForward's *differentiated forwarding*, naively direct *stable* writes to their proxies may cause a consistency issue. Suppose from $VDisk_i$, a *burst* write $r_1$ updating address range $R = [off_1, off_1+len_1]$ has been appended to $log_i$ as $entry_1$, but not merged into *storage layer* yet, and meanwhile a *stable* write $r_2$ also visits adress range $R$. At this time, if $r_2$ is naively directed to the $proxy_1$ which persists it into *storage layer*, there will




(a) Index before trimming (b) Index after trimming (c) Insert index (d) Search index

Fig. 11. Index structure and key operations.

be two versions of data in $R$, including an older version $entry_1$ and the fresh version in *storage layer*. Since $RBT_i$ maps $R$ to $entry_1$, subsequent reads visiting $R$ will get the older version, violating consistency. DiffForward reschedules write in advance to prevent such inconsistency, i.e., before sending a *stable* write updating address range $R$, $VDisk_i$'s client will firstly search $RBT_i$ to check whether there are overlapping ranges with $R$ in $log_i$. If there are, the client would reschedule the write to *distributed log store*, i.e., to append the write into $log_i$ to make it the newest version. By doing this, subsequent reads are guaranteed to get fresh data.

**Accelerating index with counting bloom filter.** Maintaining the index requires the client to insert nodes before writing to *distributed log store*, so as to use it for: i) searching for data positions in *distributed log store* for subsequent reads based on their address ranges, and ii) detecting overlapping address ranges for rescheduling subsequent *stable* writes. Due to the dominance of writes, at most time, the client doesn't need the concrete data position of searching range for operation i, but only whether overlapping ranges exist for operation ii. So we can further accelerate the index searching with a counting bloom filter (CBF). As Figure 11(c) shows, when inserting nodes into an index, the client just appends the nodes in a local queue and updates the CBF to record the write range with only O(1) time complexity. The nodes in the queue are then inserted into RB-Tree asynchronously, when the client has no traffic to send or receive. Then the client could just look up in the CBF to tell whether overlapping ranges exist. Once the client needs the concrete data positions to serve reads, it instantly flushes all waiting nodes at the queue, then search the up-to-date RB-Tree (Figure 11(d)). The accelerated index could prevent intensive inserts from delaying traffic from clients in case of *burst traffic*. Due to its lightweight and combining CBF, accessing the index only increases *VDisk* average write latency less than 10%, and increases average read latency about 1%.

**Corner case handling.** Index trimming may slightly lag behind log merging. As a result, a very small number of outdated nodes may exist in $RBT_i$. To avoid getting outdated data, when reading from logs, *distributed log store* first checks if the *validity* field in the target entry's *metadata* has been set to *outdated*. If it has, an outdated exception is returned to the client, forcing it to trim the outdated node from $RBT_i$ and retry the read. Similarly, if the client intends to read an uncompleted entry, an uncompleted exception forces it to await and resubmit the request.

## 4 DISCUSSION

Finally, we discuss limitations in DiffForward design, and present possible optimizations and key challenges, while their implementation and evaluation are left to our future work.

**Detection sensitivity adjustment.** DiffForward leverages a queue (*VRQ*) to detect *burst traffic* of each *VDisk*, during which the threshold of queue length (*TH*) is kept as a static value in our current design. Though *TH* could be manually altered to make a trade-off between traffic balance and induced overhead (e.g., lower *TH* indicates higher sensitivity to *burst traffic*, therefore better traffic balance but more system overhead, and vice versa), sub-optimal configurations may impede DiffForward's performance, especially considering the variance and dynamics of real-world workloads. A straightforward solution is to allow DiffForward to autonomously and dynamically tune its

detection sensitivity according to administrator-defined requirements, e.g., minimizing end-to-end latency or bounding system overhead; The main challenge lies in the precise prediction of system behavior after adjustment, similar to automatically tuning parameters for databases. Therefore, auto-tuning techniques, e.g, learning-based methods [42, 64, 74], could be employed to tackle it.

**Fine-grained traffic categorization.** DiffForward now categorizes overall traffic from applications into *stable traffic* and *burst traffic*, which dominate our production workloads. While we believe such classification is, for the most part, sufficient to achieve satisfactory traffic balance, finer-grained categorization may help further enhance DiffForward 's adaptability. For example, rather than the bimodal classification, an optimized design would project the overall traffic into a continuous spectrum, then dispatch traffic categories to their selected forwarding paths. Yet more complex categorization and forwarding rules entail greater system complexity and even rigidity. To tackle this, we could borrow ideas from *software-defined storage (SDS)* [29, 57, 62] to decouple specific policies from the data plane. In that way, not only customized traffic categorization and forwarding rules are allowed to meet the diverse needs of clusters caused by variance in workload, hardware, SLA/cost constraints, etc., but also dynamic rule addition to accommodate new traffic patterns.

**Opportunities to optimize traffic management**. DiffForward focuses on balancing traffic, especially the trickier burst traffic. However, the idea of separating traffic by patterns also opens up new opportunities for improving traffic management in other aspects. For example, since categories of traffic may imply their respective cache requirements or SLA needs, they ought to be provided with customized pre-fetching/caching policies and different priorities similar to co-deployed *best-effort (BE)* and *latency-critical (LC)* applications [16, 43]. To this end, relevant block traces should be further explored, along with application-level information when necessary, to reveal the correlation between traffic categories and upper-level semantics.

## 5 EVALUATION

### 5.1 Setup

**Testbed.** Our experiments run on a cluster consisting of 6 bare-metal servers, connected via a 56Gb/s network. Each server has 2 Intel(R) Xeon(R) E5-2650 V4 CPUs, 64 GB memory and 1.6TB NVMe SSD (Intel P4610), and runs Ubuntu 16.04 LTS. We use one machine to deploy *VDisk* clients which replay workload traces and record latency. Meanwhile, we use the remaining machines as proxy servers to forward traffic from *VDisk* clients, and also use them as the storage servers as well. We adopt *Ceph Rados*, a widely used distributed object store, as the *Distributed Storage Core*.

**Workloads.** We mainly focus on the publicly available could block trace [2], which is the largest trace being public. It is collected from a production cluster and records requests of 1000 VDisks running mainstream applications including operating systems, big data processing software, web servers, etc. According to our observation on the trace, the workload exhibits obvious diurnal pattern, i.e., the overall traffic first stays rather low during 0:00 am - 8:00 am, then starts to increase slowly from 8:00 am to around 11:00 am and keeps high traffic until around 8:00 pm, after that it falls back to a low traffic rate. Therefore, we intercept two representative periods, and denote the workloads as *W1* and *W2*, respectively (see Table 1 for details).

| Workload Name | Period | Traffic Pattern | Read/Write Traffic (GiB) |
|---|---|---|---|
| *W1* | 9am - 11am | Rising | 189/1617 |
| *W2* | 3pm - 5pm | Steady | 275/1767 |

Table 1. Workload description.

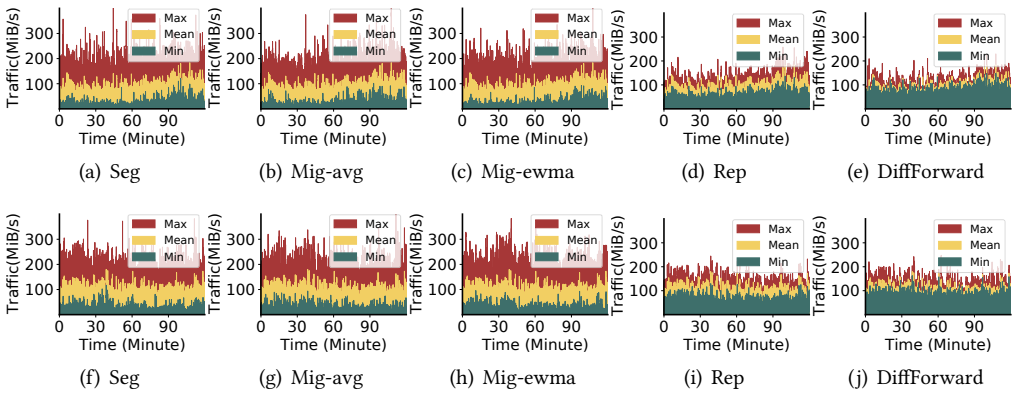



Fig. 12. Traffic distribution after deploying different approaches under workloads *W1* ((a)-(e)) and *W2* ((f)-(j)).

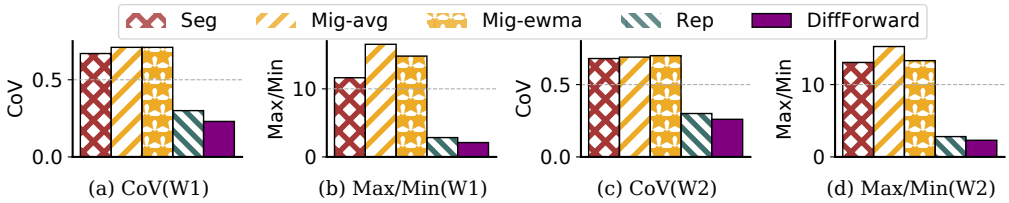



Fig. 13. Traffic balance metrics (CoV and Max/Min) under workloads *W1* and *W2*.

**Comparison.** We implement all the three approaches introduced in §2.4, including *segmentation*, *migration*, and *replication*. Specifically, for *migration*, we implement two versions with different representative prediction strategies, including predicting *VDisk* traffic as the average traffic in the last interval (*Mig-avg*), and predicting with EWMA by considering multiple previous intervals (*Mig-ewma*). For *replication*, we implement the state-of-the-art replica selection strategy of C3 [58], which selects the most appropriate replica by estimating the waiting and in-flight requests, and we adopt chain replication to synchronize replicas (*Replication*).

**Default settings.** We set the default segment size as 64GiB so as to limit the metadata management overhead at clients, and we also study its impact by adjusting the segment size from 128GiB to 16GiB in §5.4. The default time interval of prediction and migration is set as 5s for *migration*, a recommended value from production cluster to limit the migration overhead. As for network provision setting, according to our observation on over 70 in-house cloud block service clusters in production, we find that the average network utilization of all clusters is below 40%, and the 50/75/95 percentile utilizations are ~10%/~20%/~30%, respectively. Therefore, by default we limit to use 30% network bandwidth to simulate the scenario of intensive traffic, and we also adjust the network provision to study its impact in §5.4.

## 5.2 Improvement on Traffic Balance

**Traffic distribution.** We first evaluate the effectiveness of different approaches in balancing the traffic among proxy servers by showing the traffic distribution over time. We show the traffic at the most-loaded server (Max) and the least-loaded server (Min), as well as the average traffic over all servers (Mean). Figure 12(a) - 12(e) show the results under workload *W1*, and Figure 12(f) - 12(j) show the results under *W2*. Note that the smaller the gap between different curves is, the higher degree of load balance is achieved. From the results, we can see that DiffForward and *Replication* realize much better traffic balance among servers than *Segmentation*, *Mig-avg*, and *Mig-ewma*.

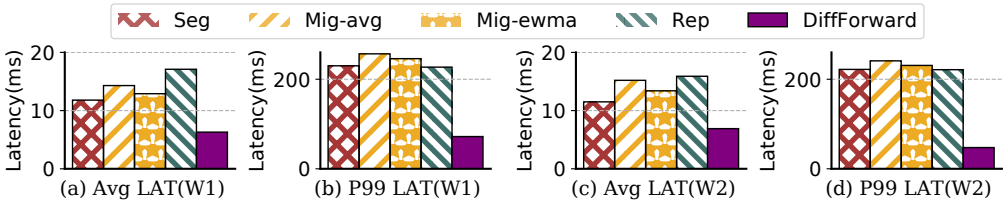



Fig. 14. Write latency.

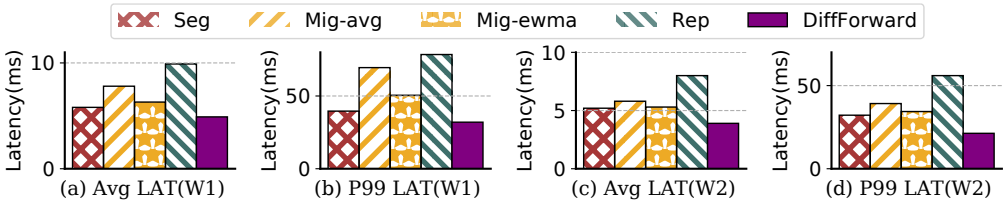



Fig. 15. Read latency.

Besides, for DiffForward and *Replication*, as traffics are more balanced, the traffic sizes at the most loaded server are largely reduced, and the traffic spikes are also smaller.

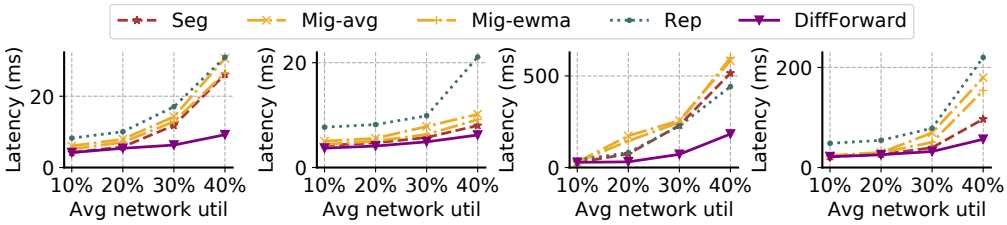
**Traffic balance metrics.** To quantitatively evaluate the improvement on the traffic balance degree, we further show the results of two balance metrics: CoV and Max/Min. CoV denotes the coefficient of variation and Max/Min is defined as the ratio of the traffic at the most-loaded server to that at the least-loaded server. Figure 13 shows the results under the two workloads. Compared with *segmentation*, DiffForward decreases CoV and Max/min by 66% and over 82%, respectively. Due to the inaccurate traffic prediction as mentioned in §2.4, *migration* even aggravates the imbalance compared with *segmentation*. So DiffForward achieves larger balance improvement compared with *mig-avg* and *mig-ewma* (~70% and 85% for CoV and Max/Min). Finally, even compared with *Replication*, which realizes the best balance among existing approaches, DiffForward further reduces the CoV and Max/Min by 24% and 25%, respectively.

### 5.3 Improvement on VDisk Latency

Now we evaluate the performance in improving *VDisks*' latency. To minimize the impact of segment size, for each traffic balancing approach, we use the segment size leading to the best result for comparison (64GiB for *Segmentation*,*Mig-avg*,*Mig-ewma*, and DiffForward; no splitting for *Replication*), and we will further discuss the impact of segment size on latency in §5.4. As the setting of the network bandwidth also influences the latency, we provision the network bandwidth to keep the average utilization at 30% in this experiment. Note that this utilization is very common in production clusters, and we also study the impact of network provision in §5.4.

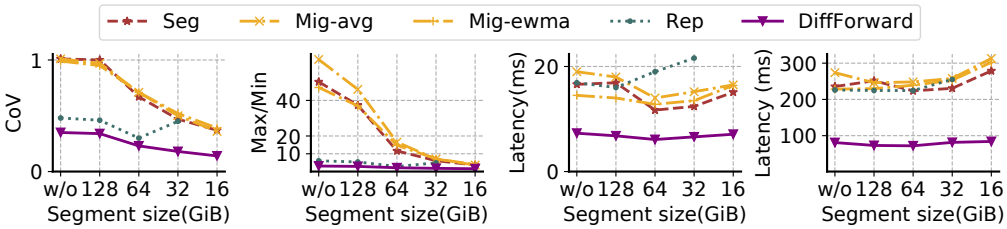
Figure 14 shows the write latency under the two workloads *W1* and *W2*. First, for both average latency and P99 latency, DiffForward significantly outperforms all existing approaches. For example, for workload *W1*, compared with *Segmentation*, *Mig-avg*, and *Mig-ewma*, DiffForward decreases the average write latency from 11.8ms, 14.3ms and 12.9ms to 6.3ms, and the reduction ratio is 47%, 56% and 52%, respectively. In addition, even though *Replication* realizes good balance, the average latency (17.1ms) is even larger than other existing approaches due to the synchronization overhead. As for the P99 tail latency, compared with *Segmentation*, *Mig-avg*, *Mig-ewma*, and *Replication* DiffForward recduces the tail latency from 230ms, 256.8ms, 246.0.1ms, and 227ms to 72ms, reduced by ~70%. We have similar conclusion for workload *W2*.

For read latency, according to Figure 15, DiffForward also achieves better performance compared to all existing approaches. The reduction ratios are over 15% and 19% for the average and tail




(a) Avg write LAT (b) Avg read LAT (c) P99 write LAT (d) P99 read LAT

Fig. 16. Latency under different network provisions.




(a) CoV (b) Max/Min (c) Avg Latency (d) P99 Latency

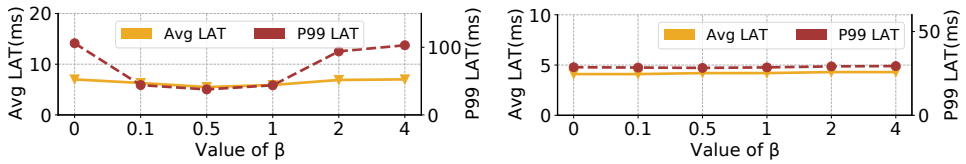
Fig. 17. Impact of different segment sizes.

latency under *W1*, and the ratios are over 25% and 20% for the average and tail latency under *W2*. In conclusion, DiffForward could significantly improve the write performance, which is more critical for VDisks as the workload is write dominant, and it also slightly improves the read performance.

### 5.4 Impact of System Configurations

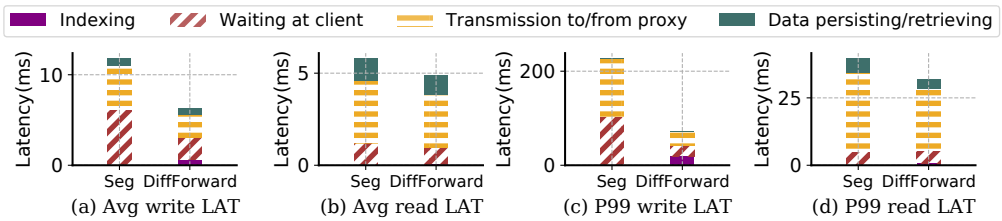
**Impact of network provision.** Recall that requests are blocked at the network of proxy servers if the traffic is too heavy (§2.1). We now adjust the network provision to make the network bandwidth utilization keep at 10%, 20%, 30%, and 40%, respectively. As the network resource mainly affects the latency performance, we show the average and P99 latency for both writes and reads in Figure 16. First, we see that DiffForward always outperforms existing approaches for both the average and P99 latency under all network settings. The improvement is much larger for writes compared with reads, and this conforms with previous results. Second, as the network utilization increases, the latency also increases significantly, especially when the utilization increases from 30% to 40%, the write latency increases exponentially. This also validates the reason why network utilization is below 40% in all in-house production clusters we studied. Finally, we can see that the improvement of DiffForward usually becomes larger under higher network utilization, this is because traffic balancing becomes more meaningful when network resources are more scarce. In particular, under 40% utilization, DiffForward can reduce the average and P99 write latency by 66% and 60%, respectively, compared with all existing approaches.

**Impact of segment size.** We now evaluate the impact of segment size. We divide *VDisk*s into segments of different sizes (including no segmentation and segment sizes of 128GiB, 64GiB, 32GiB, and 16GiB). We show both traffic balance metrics (in Figure 17(a) and (b)) and *VDisk* write latency (in Figure 17(c) and (d)). We see that under the same segment size, DiffForward always improves the balance with smaller CoV and Max/Min compared with other approaches. Besides, using small segments indeed improves the degree of traffic balance for all approaches. However, it also introduces metadata management overhead for existing approaches and thus limits the improvement on latency. For example, after segmenting *VDisk*s, the same amount of traffic is concurrently sent through more network connections of different *segments* proxies, and aggressive segmentation




(a) Latency under 30% network utilization (b) Latency under 10% network utilization

Fig. 18. **Impact of different sensitivity of detecting burst traffic:** smaller $\beta$ represents higher sensitivity.




(a) Avg write LAT (b) Avg read LAT (c) P99 write LAT (d) P99 read LAT

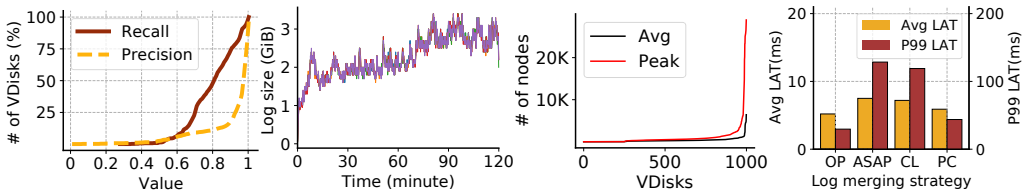
Fig. 19. Read and write latency breakdown.

causes contention between connections, thus degrading latency. Yet for DiffForward, multiple *segments* of a *VDisk* could share a same distributed log without inducing additional connections, and since the most intensive *burst traffic* is separated apart, the contention between remaining contentions transferring *stable traffic* could also be alleviated greatly. As a result, DiffForward achieves a consistent improvement on latency performance. For *Replication*, as it realizes replication at a per-request granularity, segmenting the *VDisks* only brings extra overhead, so its latency is the smallest in the case without segmentation. The results of *Replication* under 16GiB segment size are not presented because replicating proxies further multiplies the number of connections and therefore runs out of sockets.

**Impact of traffic detection sensitivity.** There is a trade-off between traffic balance and system overhead when setting the sensitivity of detecting *burst traffic*, which depends on the threshold *TH* of *VRQs*. Specifically, while higher sensitivity achieves better traffic balance by directing more traffic to distributed logs, it induces more system overhead. By contrast, lower sensitivity reduces overhead but causes worse traffic balance. By default, we set the *TH* as the average accumulated traffic of each *VDisk* in 50 milliseconds. Now we manually tune the *TH*, by multiplying it with a variable factor $\beta$, to investigate its impact on DiffForward. Figure 18(a) shows the latency results under 30% network utilization, with $\beta$ set from 0.1 to 4. As we can see, compared with the best-performing case ($\beta$ around 0.5), both insensitivity and over-sensitivity contribute to sub-optimal end-to-end latency due to consequent traffic imbalance or excessive system overhead. Yet the impact shrinks when the cluster is provided with rich redundant network resource, e.g., under 10% network utilization (see Figure 18(b)). In §4 we discuss the autonomous and dynamic tuning of *TH* to help further optimize DiffForward, while its implementation and evaluation are left to our future work.

### 5.5 Performance Breakdown and Overhead

**Latency breakdown.** To better understand the benefits of each design technique in DiffForward, we further show the latency breakdown. The end-to-end latency can be divided into the following parts: i) waiting time at clients as requests may wait at local queues before sending to proxy servers through network, ii) data transmission time when transmitting data from clients to proxies for writes or from proxies to clients for reads, and iii) time of persisting/retrieving data to/from the *storage layer* for writes/reads. As *segmentation* also has the same three parts for latency breakdown,




(a) Burst traffic detection (b) Log size (c) Size of client-side index (d) Impact of log merging

Fig. 20. **Overhead analysis:** (a) effectiveness of the burst traffic detection; (b) storage overhead of distributed logs; (c) overhead of the index at clients; (d) overhead of log merging.

we take it as the baseline for comparison. Note that for DiffForward, clients spend extra indexing time to update or search its index before sending requests. As shown in Figure 19, for writes, while the data persisting time is similar, both the waiting time and transmission time are reduced significantly for DiffForward (by around 50% for average write latency, and by 76% and 77% for P99 write latency, respectively). Besides, the extra time spent accessing index at the client-side is limited compared to other parts of latency. This demonstrates that the design of DiffForward is still lightweight at clients. Compared with writes, since read requests are seldom blocked at the client, and so the improvement is smaller, and it mainly comes from faster transmission of responses.

**Accuracy of burst traffic detection.** We now evaluate the effectiveness of the *burst traffic detection* method in DiffForward by showing both recall and precision. We define the ground truth of burst traffic as follows. We first split each *VDisk*'s service time into 100ms intervals, then we select those intervals in which the traffic is 10 times higher than the average over the past one minute, and consider the traffic in these selected intervals as *burst traffic*. Figure 20(a) shows the recall and precision of our algorithm. As we can see, the median (50 percentile) recall and precision of all *VDisk*s lies roughly at 70% and 95%, and the average recall and precision are 0.8 and 0.92, respectively. Therefore, our *burst traffic detection* method not only detects the *burst traffic* accurately, but also avoids high false positive by recognizing *stable traffic* as *burst traffic*.

**Overhead of distributed logs.** We monitor the size of the storage space occupied by *distributed logs* at each proxy server. We focus on workload *W1* and provision the network to keep 30% network utilization. Figure 20(b) shows the results. Due to the use of round-robin writes for the distributed logs, the log sizes at different servers are roughly the same during service time, and the log size at each proxy server first gradually increases from ~2GiB when the overall traffic becomes heavier as *W1* has an increasing traffic pattern (see Table 1). The log size finally remains stable at ~3GiB. This demonstrates that despite *burst traffic* with speed of hundreds of megabytes per second is continuously appended to distributed logs, our timely log merging and garbage collection could ensure the small storage space occupation of logs (GiB level).

**Overhead of client-side index.** By polling indexes of all 1000 *VDisk*s, we monitor the size of the client-side index, i.e., the number of nodes in the red-black tree, which directly determines its memory footprint and inserting/searching efficiency. Figure 20(c) shows both the average number of index nodes and the peak number of index nodes of the 1000 *VDisk*s under *W1*, where these *VDisk*s are sorted in a descending order. For 90% of *VDisk*s, their average and peak index node number are below 645 and 1658, and the largest index out of these 1000 *VDisk*s has 6426 and 28953 nodes. Since each node only takes up ~70 bytes, the largest index among all *VDisk*s only takes up 0.42 MiB on average, and 1.93 MiB at most.

**Overhead of asynchronous log merging.** We now quantify the impact of performing log merging on *VDisk* latency. Note that DiffForward leverages prioritized coordination (PC) to coordinate log merging operations by setting the foreground traffic a higher priority than log merging. We

compare it with other three cases: i) no log merging, i.e., directly removing sealed files in data logs without merging them to the *storage layer*. Note that this baseline can be regarded as the optimal case (OP); ii) ASAP (as soon as possible) that maximizes the log merging speed by concurrently pulling and merging all log entries of the selected log, and iii) CL (concurrency limited) that limits the size of the concurrently merged log at 1 MiB. As shown in Figure 20(d), all schemes (ASAP, CL and PC) introduce unavoidable latency increase. Compared to the optimal case, ASAP and CL degrade the average latency by 44% and 38%, and degrade the P99 latency both by 300%. DiffForward outperforms other schemes, and it reduces the degradation ratio to 13% and 47%, respectively.

## 6 RELATED WORK

**Block-level trace analysis.** A number of prior works have analyzed block-level traces in various architectures from Windows servers [36, 49], containerized applications [31], to smartphone applications [76]. Recently, Li et al. [40] conducted an extensive study on a large-scale trace of a cloud block storage that supports diverse applications in production, uncovering various *VDisk*s' characteristics including their load intensity, read/write aggregation, RAW (read-after-write time), WAW (write-after-write time), etc. In this work, we mainly focus on balancing the traffic, especially the burst traffic, in the forwarding layer, and we also thoroughly investigate existing workload traces to study the traffic patterns so as to guide our design.

**Block storage optimization.** Many techniques have been proposed to optimize the performance of cloud block storage. For example, Strata [21] partitions functions into an *address virtualization layer* to meet the high performance of PCIe flash devices. Salus [68] seeks to simultaneously maximize scalability, robustness, and strong availability guarantees under server failures. Blizzard [47] relaxes strong consistency to crash consistency and decouples the durability and ordering requirements expressed by flush requests, so as to improve the storage performance under intensive small random I/Os. Ursa [38] presents an SSD-HDD hybrid structure and uses journals to bridge their performance gap in order to achieve near-all-flash performance at a lower cost. SWR [44] and BCW [67] use SSDs as a faster buffer to HDDs to keep high performance, while also lengthening SSDs' lifetime by bypassing SSDs as much possible. OSCA [75] is an online model based scheme for cache allocation for shared cache servers of *VDisk*s which could find a near-optimal configuration scheme at very low complexity. Differ from these works, we intend to balance the traffic at the forwarding layer so as to optimize *VDisk*s' latency.

**Load balance in distributed storage.** Due to its critical importance for ensuring utilization and meeting stringent SLA, load balance in distributed storage systems have been widely investigated. For example, many balancing techniques have been proven to perform well in specific scenarios. Fan et al. [26] show that adding a small and fast popularity-based front-end cache can ensure load balancing for read-abundant storage systems. Some works [35, 41, 45] further exploit the capabilities of programmable switch to cache hot items in the switch data plane, and route requests to the appropriate nodes at line speed. By consuming extra network bandwidth, Narayanan et al. [49, 50] offload written data of high-loaded servers or devices to less-loaded ones to balance I/O load under write-intensive workload. In the meantime, lots of storage systems apply and combine more general methods to ensure their load balance, including *segmentation* [6, 7, 12, 15, 19, 23, 53], *migration* [22, 24, 30, 37, 39, 46, 54, 63], and *replication* [5, 58, 59, 61]. Rather than balancing the loads from diverse applications uniformly, we adopt a differentiated approach to address the problem of balancing both burst and stable traffic at the forwarding layer for cloud block services.

## 7 CONCLUSION

In this paper, we develop a novel differentiated traffic forwarding scheme DiffForward for cloud block services. DiffForward accurately differentiates *burst traffic* and *stable traffic* in a lightweight

manner at clients, and leverages different forwarding paths so as to balance the burst traffic efficiently at the subsecond time scale. DiffForward utilizes distributed logs to balance burst traffic and manages the data in an asynchronous way to reduce overhead while still preserving strong consistency guarantee. Experiments on our prototype demonstrate that DiffForward outperforms all existing approaches and thus significantly optimizes the latency of accesses to VDisks under different network resource provisions.

## ACKNOWLEDGMENTS

The work is supported by NSFC 62172382 and the Alibaba Group through Alibaba Innovative Research Program. We deeply appreciate all teammates from Alibaba group for the significant supports for the online experiments. The work of John C.S. Lui was supported in part by the RGC's GRF 14215722.

## REFERENCES

[1] Alibaba. 2011. Alibaba Cloud Block Storage. https://www.alibabacloud.com/product/disk?spm=a3c0i.7911826.6791778070.dnavproductstorage7.44193870i1BSO8.
[2] Alibaba. 2020. Alibaba Block Trace. https://github.com/alibaba/block-traces.
[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74.
[4] Amazon. 2022. Amazon Elastic Block Storage (EBS). https://aws.amazon.com/cn/ebs/.
[5] Amazon. 2022. Amazon ELB. http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/TerminologyandKeyConcepts.html.
[6] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the shards: managing datastore locality at scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 445–460.
[7] Apache. 2022. Apache HBASE. http://hbase.apache.org/.
[8] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan
[9] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340.
[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
[11] Hongming Cai, Boyi Xu, Lihong Jiang, and Athanasios V Vasilakos. 2016. IoT-based big data storage systems in cloud computing: perspectives and challenges. *IEEE Internet of Things Journal* 4, 1 (2016), 75–87.
[12] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.
[13] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
[14] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* 13 (2014).
[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2011. Bigtable: A distributed storage system for structured data. (2011).
[16] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 107–120.
[17] Maureen Chesire, Alec Wolman, Geoffrey M Voelker, and Henry M Levy. 2001. Measurement and analysis of a streaming media workload. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*.

[18] Citrix. 2022. Xendesktop. https://www.citrix.com/products/xendesktop/overview.html.
[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2012. Spanner: Google's globally distributed database. (2012).
[20] Stephen V. Crowder and Marc D. Hamilton. 1992. An EWMA for Monitoring a Process Standard Deviation. *Journal of Quality Technology* 24, 1 (1992), 12–21.
[21] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. 2014. Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 17–31.
[22] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.
[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
[24] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. 2011. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*.
[25] Facebook. 2022. A Persistent Key-Value Store for Fast Storage Environment. http://www.rocksdb.org.
[26] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages.
[27] Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. *OSDI, 2004* (2004).
[28] Sanjay Ghemawat and Jeff Dean. 2022. A Persistent Key-Value Store for Fast Storage Environment. https://github.com/google/leveldb.
[29] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. 2017. Crystal:{Software-Defined} Storage for {Multi-Tenant} Object Stores. In *15th USENIX Conference on File and storage Technologies (FAST 17)*. 243–256.
[30] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. 2018. Elmem: Towards an elastic memcached system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 278–289.
[31] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 181–195.
[32] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in cloud computing: What it is, and what it is not. In *10th international conference on autonomic computing (ICAC 13)*. 23–27.
[33] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.
[34] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2014. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 187–198.
[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136.
[36] 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*. 119–128.
[37] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 390–405.
[38] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. URSA: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 17 pages.
[39] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 387–406.

[40] 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.

[41] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 31–44.

[42] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 981–992.

[43] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*. 347–360.

[44] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 2019. Analysis of and Optimization for Write-Dominated Hybrid Storage Nodes in Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 403–415.

[45] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 143–157.

[46] Chenyang Lu. 2002. Aqueduct: Online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST 02)*.

[47] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. 2014. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 257–273.

[48] Microsoft. 2022. Azure Disk Storage. https://azure.microsoft.com/en-us/services/storage/disks/#overview.

[49] 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.

[50] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony IT Rowstron. 2008. Everest: Scaling Down Peak Loads Through I/O Off-Loading.. In *OSDI*, Vol. 8. 15–28.

[51] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. {AGILE}: Elastic Distributed Resource Scaling for {Infrastructure-as-a-Service}. In *10th International Conference on Autonomic Computing (ICAC 13)*. 69–82.

[52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[53] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 29–41.

[54] Swapnil Patil and Garth Gibson. 2011. Scale and Concurrency of {GIGA+}: File System Directories with Millions of Files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*.

[55] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 1–14.

[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[57] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. 2016. {sRoute}: Treating the Storage Stack Like a Network. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 197–212.

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

[59] Igor Sysoev. 2022. Nginx. http://nginx.org/en/docs/http/load_balancing.html.

[60] Mojtaba Tarihi, Hossein Asadi, and Hamid Sarbazi-Azad. 2015. DiskAccel: Accelerating Disk-Based Experiments by Representative Sampling. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Portland, Oregon, USA) *(SIGMETRICS '15)*. Association for Computing Machinery, New York, NY, USA, 297–308.

[61] Basho Technologies. 2022. Riak. Load Balancing and Proxy Configuration. http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[62] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 182–196.

[63] Nguyen Tran, Marcos K Aguilera, and Mahesh Balakrishnan. 2011. Online migration for geo-distributed storage systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.

[64] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[65] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *OSDI*, Vol. 4.

[66] VMware. 2022. View. http://www.vmware.com/products/horizon-view.

[67] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 2020. BCW: Buffer-Controlled Writes to HDDs for SSD-HDD Hybrid Storage Server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 253–266.

[68] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus Scalable Block Store. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 357–370.

[69] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 35–49. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

[70] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. 2007. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. 35–44.

[71] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *Proc. of USENIX ATC*.

[72] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 325–338.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[74] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[75] 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)*. USENIX Association, 785–798.

[76] Deng Zhou, Wen Pan, Wei Wang, and Tao Xie. 2015. I/O Characteristics of Smartphone Applications and Their Implications for eMMC Design. In *2015 IEEE International Symposium on Workload Characterization*. 12–21.

Received October 2022; revised December 2022; accepted January 2023