Scheduling Cloud Block Storage Proactively and Reactively with Omar

Xinqi Chen[1], Weidong Zhang[2], Zhongyu Wang[2], Erci Xu[1*], Xiaolu Zhang[2], Dong Wu[2], Junping Wu[2], Haonan Wu[1], Ruiming Lu[1], Yaheng Song[2], Chaolei Hu[2], Lijun Ding[2], Guangtao Xue[1,3], Patrick P. C. Lee[4]

[1] Shanghai Jiao Tong University, China
[2] Alibaba Group, China
[3] Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3, China
[4] The Chinese University of Hong Kong, China

Abstract

We explore the performance improvements for a production Elastic Block Storage (EBS) service at Alibaba, which serves millions of users and virtual disks daily. Despite various load balancing measures, Alibaba EBS still faces frequent performance variations or even occasional Service Level Objective (SLO) violations. Our trace analysis reveals that the existing scheduling mechanism in Alibaba EBS, which adopts a common practice of focusing on load balancing alone, is insufficient for mitigating tail latency due to negligence of correlated I/O workloads, reliance on fixed scheduling frequencies, and insufficient input metrics for scheduling decisions. We present Omar, a novel hybrid proactive and reactive scheduling framework that enables dynamic, fine-grained scheduling. Evaluation on production-scale testbeds shows that Omar reduces 64 KiB tail latency by up to 52.2% and the number of scheduling operations by 44.6% compared to the baseline scheduler, while incurring limited overhead.

CCS Concepts: • **Information systems** → **Storage management**; **Cloud based storage**.

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

ACM Reference Format:
Xinqi Chen, Weidong Zhang, Zhongyu Wang, Erci Xu, Xiaolu Zhang, Dong Wu, Junping Wu, Haonan Wu, Ruiming Lu, Yaheng Song, Chaolei Hu, Lijun Ding, Guangtao Xue,, Patrick P. C. Lee. 2026. Scheduling Cloud Block Storage Proactively and Reactively with Omar. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3767295.3769341

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

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769341

1 Introduction

Cloud computing relies heavily on robust, high-performance storage solutions, with Elastic Block Storage (EBS) serving as a cornerstone for providing virtual machines (VMs) with scalable block-level storage volumes called *virtual disks (VDs)*. In this work, we present an in-depth study of the Alibaba EBS service that serves millions of users and VDs daily. Similar to industry implementations such as AWS [2] and Azure [4], Alibaba EBS employs a compute-to-storage disaggregation architecture, where a set of *blockservers* process block I/O requests for VDs and transmit user data into storage backends (see §2.1 for details), and a few *blockmasters* manage metadata and perform traffic scheduling and load balancing. Each VD's logical block address is partitioned into fixed-size segments. This segmentation technique, a common practice in cloud storage systems [1, 3, 5, 41], can effectively achieve load balancing and alleviate I/O hotspots.

Despite various load balancing measures being deployed, Alibaba EBS, which adopts a heuristic-based scheduler (referred to as *Original*), experiences significant performance challenges, or even occasionally violates the Service Level Objective (SLO), in the face of high variability in real-world workloads. The Original scheduler monitors the average write traffic of each blockserver and migrates segments from overloaded blockservers to underloaded ones. While the Original scheduler has maintained a fairly good balance across blockservers, our trace analysis (§3) reveals that substantial latency skewness still exists, with P50 latency varying by up to 4× across blockservers and frequent tail latency (P999) spikes appearing up to 5× than normal.

We observe that *achieving load balancing alone is insufficient for mitigating tail latency in Alibaba EBS*. We identify three major root causes for such performance challenges faced in Alibaba EBS. First, the Original scheduler employs a random segment placement approach that neglects correlated traffic patterns (e.g., from VDs owned by the same user and VD lifespans). Second, it adopts a fixed scheduling frequency, which cannot adapt to workload variations. Third, it focuses solely on minimizing the write Coefficient

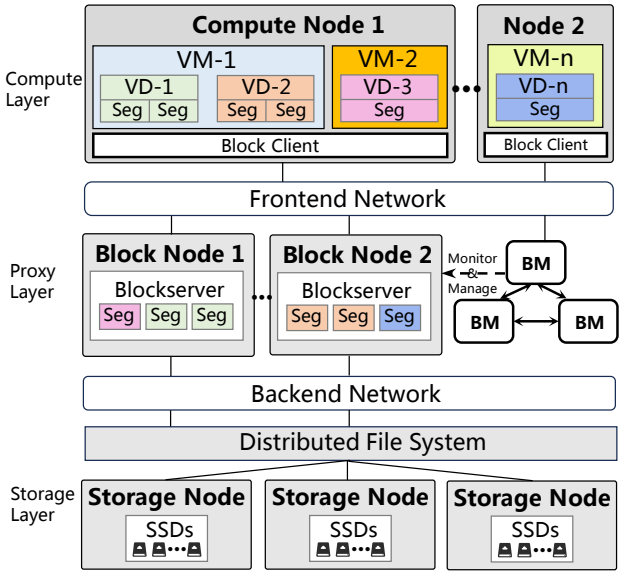


Figure 1. Overview of the Alibaba EBS architecture (§2.1).

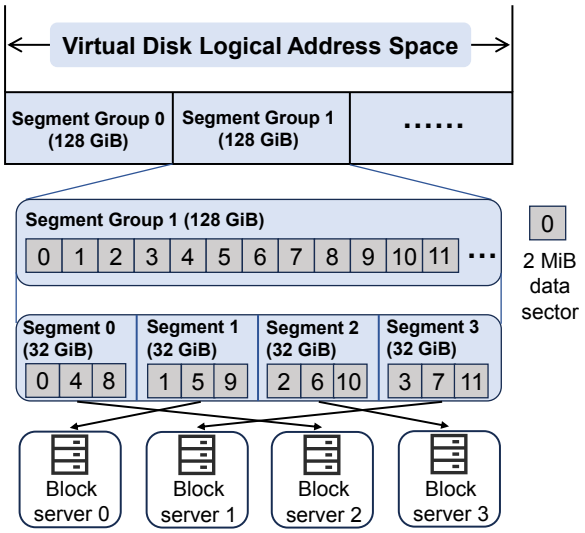


Figure 2. VD segmentation of Alibaba EBS (§2.1).

of Variation (CoV) [10], yet read traffic imbalance can further exacerbate write I/O performance degradation.

Although several studies [17, 23, 29, 37, 39, 44] have addressed similar tail latency issues, they are unsuitable for Alibaba EBS due to various limitations, including imprecise prediction of traffic bursts [29, 39], lack of real-time response [17, 23], focusing solely on single metrics like CoV [37], and neglecting the overhead of migrating segments [44].

Based on the above motivations, we propose Omar, a novel segment scheduler with the primary goal of mitigating tail latency in EBS by combining proactive and reactive strategies to achieve fine-grained scheduling decisions. Omar comprises three main components: First, we develop a "resonance" allocator to identify segments with correlated traffic using the Pearson Correlation Coefficient (PCC) and proactively (re-)distribute them among blockservers. Second, Omar filters source/destination blockservers according to current traffic statistics and employs a score-based algorithm to select segments for migration. Third, Omar dynamically adjusts the scheduling frequency based on collected system performance metrics to explore optimal values.

We evaluate Omar using two testbeds: microbenchmarks with 11 blockservers and macrobenchmarks with 50 blockservers. We compare Omar against other candidates by replaying field I/O traces from 1000 high-loaded VDs. We show that Omar reduces the 64 KiB (i.e., the most common I/O size of requests in EBS) tail latency by up to 52.2%, and decreases the number of schedules by 44.6% with minimal computing overhead. Additional evaluation results (e.g., impact of various I/O sizes, read-to-write ratios, and ablation studies on each component) further validate the effectiveness, robustness, and scalability of Omar in complex scenarios.

We summarize our contributions as follows:

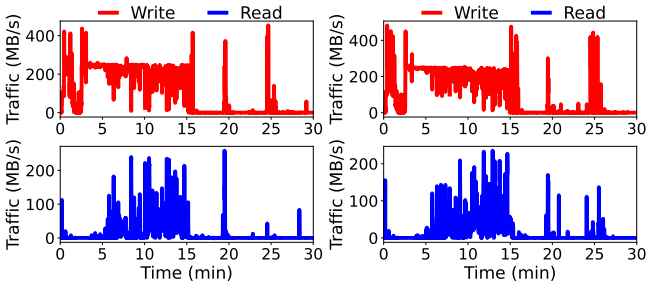
- We analyze field traces and identify the root causes of performance degradation in our production EBS service, and provide our learned lessons.
- We propose Omar, a combined proactive and reactive segment scheduler for EBS that captures various features. Evaluation shows that Omar outperforms the Original scheduler by up to 52.2% in tail latency reduction.
- We release our Omar prototype at https://github.com/Master-Chen-Xin-Qi/EuroSys26_AE, and the production I/O traces collected from Alibaba EBS at https://tianchi.aliyun.com/dataset/210122.

2 Background

2.1 Alibaba EBS Architecture

Overview of Alibaba EBS. Alibaba EBS operates as a disaggregated service that separates compute and storage functionalities across three layers, as shown in Figure 1. The *compute layer* hosts users' VMs and processes I/O requests. When a user subscribes to a VD, the VD is attached to the user's VM as a block device. I/O requests issued to the VD are forwarded by the *block client* within the VM to the remote storage backend via the inter-cluster (data center) network. The *proxy layer* translates block I/O semantics from block clients into file semantics, which are transmitted over the intra-cluster network to the *storage layer*. Then, the storage layer persists data to the underlying Alibaba distributed file system, Pangu [28], which ensures data durability and availability with erasure coding or 3-way replication. The three-layer EBS architecture is also employed by various vendors (e.g., Azure [13], Tencent [36]) and studied in academic work [25, 26].

VD segmentation. To mitigate traffic skewness and alleviate hotspots, the EBS service partitions the logical address



(a) Traffic of VD1 (b) Traffic of VD2

Figure 3. Traffic patterns of two resonant VDs (§3.3).

space of a VD into 32 GiB *segments* and aggregates every four segments into a 128 GiB *segment group*. It allocates 2 MiB data sectors to segments in a round-robin manner. For example, in Figure 2, data sectors are numbered starting from 0, with sectors 0, 4, 8, and so forth assigned to Segment 0. Each segment is managed by a specific *blockserver*, which can host multiple segments per VD. Also, the EBS service hosts multiple *blockmasters* as metadata managers to maintain mappings between segments and blockservers. The blockmasters collect real-time heartbeat messages from blockservers to monitor their health status, and initiate segment migrations in case of severe load imbalance to ensure high I/O performance.

2.2 Segment Scheduling

The association between a segment and a blockserver is dynamic and may change under three conditions:

- **Blockserver outage.** If a blockserver becomes unavailable (e.g., due to a crash), the affected segments are migrated to another blockserver to ensure service continuity.
- **Traffic skewness.** Based on the I/O request distribution, segments are redistributed to avoid overloading a single blockserver during workload bursts.
- **Scope of impact.** Based on the number of segments and associated VDs per blockserver, segments are migrated to mitigate the impact on users under blockserver failures.

The migration process between a pair of source and destination blockservers is orchestrated by a blockmaster's *segment scheduler* and involves three steps. First, the blockmaster initiates a load task on the destination blockserver, which acquires the segment lock. Second, the destination blockserver loads and reconstructs the index map (i.e., mappings between segments and files) from the storage layer and seals files being written by the source blockserver. Third, if I/O requests are issued to the source blockserver, it returns an error with the destination blockserver's address for retries.

Study scope. In this paper, we make the following design choices. We focus on intra-cluster migrations, each of which typically completes in tens of milliseconds. Inter-cluster migrations may occur when a cluster's traffic exceeds a predefined threshold, but they are rare in practice. Also, a prior study on Alibaba EBS [47] analyzes I/O latency breakdowns and reveals that blockserver stack processing contributes the largest fraction of the latency of a read/write request, especially for tail latency (75.4% for writes and 64.6% for reads [47]). To mitigate tail latency, we focus on performing scheduling in the proxy layer to address the access overhead from blockservers to the storage layer.

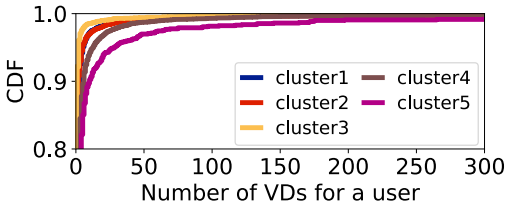


Figure 4. Number of VDs for users in five clusters (§3.3).

2.3 Requirements for a Segment Scheduler

In this work, our primary goal is to *design a segment scheduler that mitigates the tail latency of requests in EBS*. Given the diverse applications served by Alibaba EBS and its deployment across a large number of clusters, the segment scheduler should adhere to the following design principles:

- **Real-time performance.** Since EBS is latency-sensitive, the scheduler should generate migration decisions at the microsecond level.
- **Explainability.** The scheduling policy should be transparent and adjustable that can be readily understood and fine-tuned by administrators. The scheduler should be extensible with new scheduling objectives to adapt to evolving EBS requirements.
- **Robustness and scalability.** The scheduler should perform reliably across diverse traffic patterns and scale effectively with varying cluster sizes.
- **Ease of deployment.** The scheduler should be adaptable to various hardware and software environments in different EBS clusters.

3 Motivation

In this section, we analyze the limitations of the segment scheduler (referred to as *Original*) currently used in Alibaba EBS, using production traces to highlight key issues and their root causes. We also evaluate alternative scheduling approaches and their shortcomings.

3.1 Trace Overview

Our traces consist of two parts. The first part includes per-I/O traces from VDs across five production clusters in Alibaba EBS, each with 68 blockservers on average, over a 30-minute period for trace replay. The write-to-read traffic ratio is approximately 4.2. The traces capture representative access patterns (e.g., read/write-dominant, steady/burst), and common cloud application workloads (e.g., database and big data analysis). We present detailed trace fields and I/O sizes in §6.

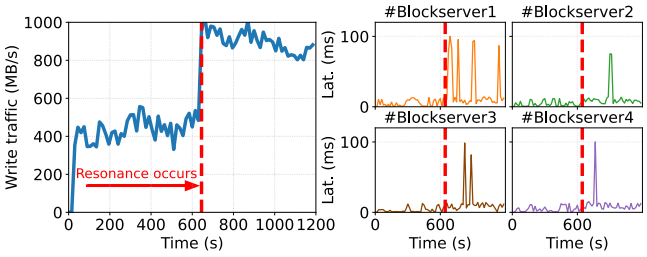


Figure 5. Write traffic and 4 KiB P99 latency results when resonance occurs in an EBS cluster (§3.3).

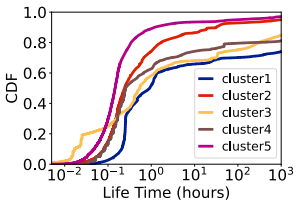


Figure 6. Lifespans of VDs from five production clusters within 65 days (§3.3). *74% VDs live less than one hour.*

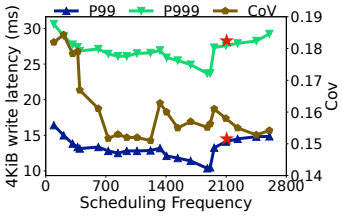


Figure 7. 4 KiB write tail latency and average CoV results (§3.4). *Red stars are results of the Original scheduler.*

The second part is the VD-level statistics from a 65-day monitoring, including VD-to-user mappings, lifespans of VDs, and read/write traffic among VDs.

3.2 Issues with the Original Segment Scheduler

As in common practices (e.g., K8s [6], Nginx [7], and TiDB [9]), the Original segment scheduler prioritizes balancing traffic skewness, while outages (with a monthly rate of 0.33%) and adjustments for scope of impact are infrequent. In addition, it focuses on balancing write traffic since Alibaba EBS is write-dominant (§3.1) and caches in the compute layer can effectively absorb read traffic.

The Original scheduler operates as follows. It calculates the average write throughput T_w of each blockserver at a fixed 6-second interval. If $\max(T_w)$ exceeds a threshold S (70% of NIC bandwidth by default), it determines that heavy traffic exists and further evaluates skewness. It determines that skewness exists if either the ratio $\frac{\max(T_w)}{\text{mean}(T_w)}$ exceeds a predefined threshold $R \in (1, 1.5)$ (i.e., hotspot existence) or $\frac{\min(T_w)}{\text{mean}(T_w)}$ falls below $2 - R$ (i.e., resource underutilization). When skewness is detected, the Original scheduler adopts a greedy approach to migrate the highest-traffic segment from the $\max(T_w)$ blockserver to the $\min(T_w)$ blockserver, continuing until no skewness exists. Note that the Original scheduler always migrates all (instead of partial) segments of the same VD on the $max(T_w)$ blockserver to minimize the distribution of VDs across blockservers and hence maintain a consistent scope of impact.

The Original scheduler aims to reduce the Coefficient of Variation (CoV) of write traffic. However, this greedy approach, despite achieving balanced blockservers (within 0.1× deviation), often fails to meet Quality of Service (QoS) requirements, leading to Service Level Objective (SLO) violations in Alibaba EBS, especially tail latency see §3.3-§3.5 for reasons). Field statistics reveal significant latency discrepancies, with the 4 KiB P50 latency varying by up to 4× across blockservers and the P999 I/O latency surging up to 5× above normal.

Takeaway. This indicates that *simply balancing traffic is not able to guarantee low tail latency.* In the following, we analyze the root causes (RCs) of these issues.

3.3 RC 1: Initial Random Placement

Previous VM scheduling studies [11, 21] rely on the central limit theorem, which assumes independent and identically distributed traffic across VMs. Thus, Alibaba EBS initially adopted a random placement strategy for segments across blockservers. However, VDs owned by the same user often exhibit correlated traffic due to shared business logics (e.g., cross-VD data ingestion). Figure 3 depicts two VDs of the same user with highly correlated write and read traffic. We term this phenomenon *resonance*, defined as the duration when the correlation coefficient of traffic exceeds 0.7. Figure 4 shows that large users (owning tens of VDs) constitute less than 20% of users but account for 80% of VDs, thereby amplifying the impact of resonant traffic. Collectively, resonant VDs contribute 43.5%-69.4% of total cluster traffic.

The Original scheduler does not address resonance, which disrupts load balance and causes latency spikes. Figure 5 depicts a resonance event (red dashed line) where write traffic doubles, leading to 10× increase in the 4 KiB write P99 latency on affected blockservers (e.g., blockserver1). While unaffected blockservers (e.g., blockserver2) initially show a stable latency distribution, subsequent segment migrations due to traffic imbalance cause collateral latency degradation.

Also, the random placement strategy fails to address the heterogeneity of VD lifespans (i.e., the durations from creation to destruction). Figure 6 shows that 74% of VDs have lifespans under one hour (e.g., serverless applications or spot instances), based on 65 days of data from five clusters. However, if we take a lifespan snapshot of a cluster every 10 minutes, long-lived VDs (over one day) dominate and comprise over 90% of active VDs. Migrating short-lived VDs incurs unnecessary scheduling overhead, so VD lifespans should be addressed to mitigate scheduling overhead.

Takeaway. Instead of random allocation, the widespread resonance and short-lived VDs call for a more proactive and selective scheduling strategy.

3.4 RC 2: Fixed Scheduling Frequency

If we stick to the triggering conditions in §3.2, the EBS service would face excessive migrations, which degrade both the performance of blockmasters and the availability across VDs.

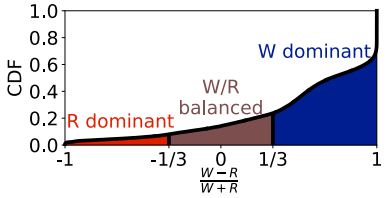


Figure 8. CDF of $\frac{W-R}{W+R}$ ratios of segments (§3.5).

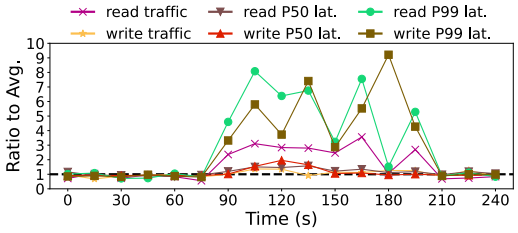


Figure 9. Subpar P99 latency performance due to read traffic imbalance (§3.5).

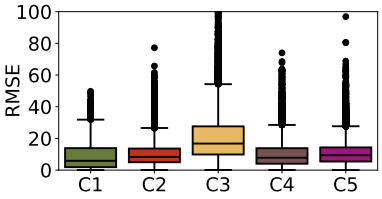


Figure 10. Results of RMSE prediction on segment traffic in five clusters (§3.6).

Thus, we cap the scheduling frequency (*SF*) by periodically (every one second by default) issuing tokens for migration. Each token allows one segment migration.

Intuitively, one would suggest increasing the scheduling frequency to aggressively redistribute traffic and hence reduce the influence of resonance. We explore this idea by replaying the analyzed 30-minute production trace on a 14-node cluster (see Testbeds in §6 for more details) multiple times with different scheduling frequencies.

Selecting an optimal scheduling frequency is critical yet challenging. Figure 7 depicts the 4 KiB write latency for various scheduling frequencies under the replay. Both P99 and P999 tail latencies follow a curve, with optimal performance at *SF*=1914. The Original scheduler at *SF*=2096 has 26.5% worse P99 latency. In fact, Figure 7 shows that a conservative or overly aggressive scheduling frequency can both hurt tail latency: few schedules cannot mitigate traffic skewness, while excessive migration introduces momentary service unavailability. Even if we manage to locate the optimal scheduling frequency (e.g., *SF*=1914 in our case), the varying nature of workloads can make our efforts of exploration futile.

Takeaway. Varying workloads and the non-linear relationship between scheduling frequency and tail latency necessitate a strategy that automatically and dynamically adapts to current traffic conditions.

3.5 RC 3: Suboptimal Candidate Selection

The Original scheduler also selects suboptimal candidates (i.e., segments and blockservers) for migration. Recall that the Original scheduler focuses exclusively on minimizing the CoV of write traffic, but this objective does not consistently align with high QoS. Figure 7 shows that the minimum CoV (at *SF*=1219) results in 24.2% worse P99 latency compared to the optimal point (at *SF*=1914). The reason is that the Original scheduler overlooks critical metrics (e.g., traffic deviation, latency, and IOPS) and neglects read traffic balance.

Figure 8 analyzes the traffic ratio $\frac{W-R}{W+R}$, where W and R denote the write and read traffic, respectively; we use this ratio to simplify the handling of W=0 or R=0. A positive ratio (maximum at 1) implies more writes than reads, while a negative ratio (minimum at -1) implies more reads than writes. We say that a segment is write-dominant if $\frac{W-R}{W+R} > 1/3$, or read-dominant if $\frac{W-R}{W+R} < -1/3$. The figure shows that 76% of segments are write-dominant, while 8% of segments are read-dominant. The Original scheduler initially focuses on write traffic due to the write-intensive nature of EBS and the use of read caches in the VM file system and blockservers (reducing the average read latency by 70% for high-performance VDs).

However, imbalanced read traffic degrades performance. Figure 9 shows that read traffic on some blockservers can be 2-3× higher than average, based on 240 seconds of data. While the P50 read latency remains relatively stable (up to 1.9× above average), the P99 read latency can increase up to 8×. More importantly, severe read traffic imbalance exacerbates write latency. While write traffic is relatively balanced (up to 1.3× above average), the P99 write latency can reach 9× higher. The reason is that since reads have much less traffic than writes, the Original scheduler opts to process read and write requests in the same thread to optimize resource utilization, and the FIFO thread scheduling causes CPU contention. In addition, read and write requests share the same Rx queue of the NIC, leading to contention for NIC resources. Thus, effective scheduling should consider both read and write traffic.

Apart from read traffic, there are other factors impacting the candidate selection in the Original scheduler. First, the Original scheduler migrates all segments of a VD on the source blockserver to the destination one to reduce the scope of impact. But this coarse-grained migration can relocate segments with low traffic, and prolong the inaccessibility time of the VD. Second, migrating the highest-traffic VD without considering overloading the destination blockserver can cause "ping-pong" effects (i.e., segments moved back and forth), as noted in prior studies [31, 34, 39]. Third, the Original scheduler only triggers migrations based solely on blockserver traffic, but ignores resource consumption differences across traffic types (e.g., kernel/user TCP and RDMA).

Takeaway. Multiple factors affecting the QoS of EBS suggest that candidate selection should be more inclusive in terms of diverse traffic types, finer-grained targets, and resource consumptions on source/destination blockservers.

3.6 Limitations of Alternative Schedulers

Prediction-based schedulers. Load balancers [15, 29, 39, 45] often predict future traffic to enable proactive scheduling. However, segment-level traffic in the multi-tenant public

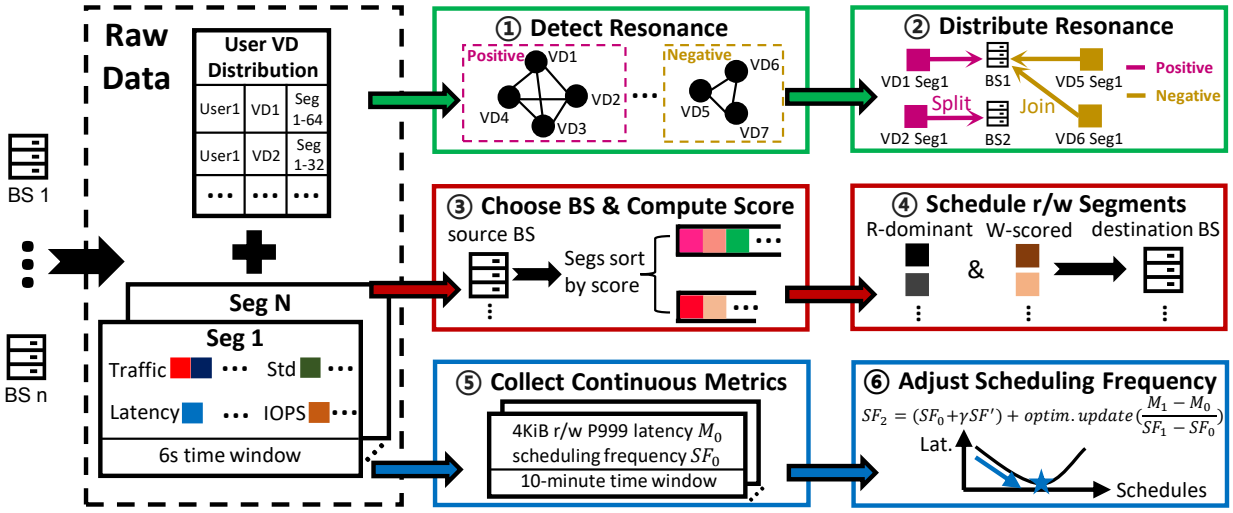


Figure 11. The workflow of Omar (§4.1). *Omar is proactive (①~②) and reactive (③~⑥) mixed.*

cloud EBS is bursty and varying. We consider a commonly used time-series prediction algorithm, ARIMA [12, 16], and use a 10-minute training window to predict 1-minute traffic. We train independent models per segment with parameter searching for optimal results (note that this is infeasible in production due to high training overhead and inference latency). Figure 10 shows that the average Root Mean Square Error (RMSE) of all clusters exceeds 5 and even reaches 100, implying low prediction accuracy. As the number of users increases, the traffic pattern becomes more diversified. This emphasizes the need for reactive scheduling.

Optimization-based schedulers. Complex models (e.g., ML models [30, 40] and Integer Linear Programming solvers [17, 23]) provide accurate scheduling decisions, but require tens of seconds or even minutes to compute and hence violate the real-time requirement (§2.3). This delay can render scheduling decisions obsolete as burst traffic disappears.

CoV-centric schedulers. Prior studies [31, 37] focus on optimizing load balancing using CoV on metrics such as traffic and IOPS via simulations. However, they share the limitations of RC 3 (§3.5) and do not address tail latency, a critical performance metric in EBS. A recent work [50] optimizes I/O latency, but it introduces hardware with extra SSDs installed on blockservers, metadata overhead, and consistency impacts (i.e., only one copy of data on blockservers before flushing), thereby increasing system complexity.

Migration-agnostic schedulers. YouTube's Prequal [44] prioritizes Requests In Flight (RIF) and latency over load balancing, but it is unsuitable for EBS. First, Prequal assumes ephemeral tasks with multiple replica choices and static placement, while EBS requires dynamic migration of long-lived segments bound to single blockservers. Second, Prequal neglects scheduling overhead, which is in contrast critical for EBS. Third, reliance on instantaneous RIF is infeasible for EBS due to telemetry fluctuations that risk migration thrashing. Finally, Prequal requires per-request balancing and placement, whereas EBS only migrates specific segments during load skewness.

4 Omar Design

4.1 Design Overview

Our analysis in §3 highlights that the EBS tail latency is influenced by a complex interaction of multiple factors, including VD resonance, scheduling candidate selection, and scheduling frequency. We present Omar, a segment scheduling framework that combines *proactive* and *reactive* strategies to mitigate the tail latency in EBS. Figure 11 depicts the overall workflow of Omar, which comprises three core components.

Resonance-based allocator (proactive, §4.2). Omar first obtains VD-to-user mappings and collects real-time statistics (e.g., traffic and latency) from all blockservers within the cluster. Then, Omar identifies all resonant VDs under the same user (①) and distributes their segments across blockservers (②). Also, Omar co-locates VDs with negative correlations together to further smooth out potential traffic bursts.

Score-based candidate selector (reactive, §4.3). Instead of relying on a single metric (i.e., traffic), Omar employs a score-based approach that selects source and destination blockservers for segment migration by periodically computing a score (i.e., an estimation of QoS by summarizing an extensive set of metrics) for each segment (③). It then performs segment-level migration by scores (④).

Scheduling frequency adjuster (reactive, §4.4). Recall that the correlation between the scheduling frequency and tail latency is non-linear (§3.4). Omar collects latency metrics (e.g., 4 KiB P999 read/write latency) and the number of schedules in fixed time windows (⑤), and uses a gradient descent method to determine the optimal scheduling frequency (⑥).

4.2 Resonance-based Allocator

The resonance allocator proactively mitigates traffic bursts caused by correlated VDs following a three-phase procedure: evaluate, split, and join.

Evaluate phase. Omar uses Pearson Correlation Coefficient (PCC) [14], a widely used statistical tool, to identify resonant VDs. It collects 10-minute read/write traffic data for each VD from the blockmaster and computes pairwise PCCs for VDs owned by the same user. We use a PCC threshold T_{cc}=0.7 as in prior work [20, 33] to determine whether a pair of VDs indeed exhibits positively correlated traffic (i.e., resonant).

Split phase. After obtaining pairwise PCCs, Omar constructs a graph, termed G_p, based on the correlated VD pairs, where each node represents a VD and each edge represents a correlation between two VDs. Since users can run different applications simultaneously (e.g., data collection and ML model training), multiple resonant groups (i.e., connected components) co-exist in each graph. Omar then identifies all resonant groups, namely $G_{pc} = \{G^1_{pc}, G^2_{pc}, ...\}$. As shown in ② of Figure 11, for each resonant group, segments with similar traffic from each VD are distributed to different blockservers to prevent hotspots.

Join phase. Interestingly, the VD traffic of the same user can also be *negatively correlated* (i.e., PCC < $-T_{cc}$), possibly due to continuous read and write behaviors of users across multiple VDs. This renders an opportunity to further offset the resonance phenomenon. Similar to positive graph G_p and groups G_{pc}, Omar also constructs a graph G_n and identifies groups G_{nc} for negatively correlated VDs. In contrast, these segments under negative groups are joined on the same blockserver to smooth out burst traffic. Once resonant segments are placed, they are bound to their blockservers (i.e., cannot be migrated) until the resonance groups are recalculated. Omar only migrates non-resonant segments (elaborated in §4.3), which greatly reduces the decision space and improves scheduling efficiency.

4.3 Score-based Candidate Selector

Recall that the Original scheduler relies solely on blockserver-level traffic and migrates all segments of a VD from one blockserver together. This incurs substantial unnecessary background traffic and ping-pong effects, leading to high tail latency. The score-based candidate selector in Omar is designed to alleviate the above issues by prioritizing segments for migration based on a comprehensive QoS score. Omar collects multiple metrics in 6-second time windows, including segment traffic (t), standard deviation of traffic (std), latency (l), and IOPS (I), with subscripts w and r denoting write and read operations, respectively. The selection process involves three steps.

Step 1: Selecting blockserver candidates. Omar categorizes cluster load based on the highest blockserver traffic into three levels: (i) *low* (0-300 MB/s), (ii) *medium* (300-800 MB/s), and (iii) *high* (800-1200 MB/s) (note that a single blockserver's traffic is capped at 1200 MB/s in Alibaba EBS). The low level indicates sufficient resources, thus migration is not needed. The medium level indicates that blockservers generally maintain adequate throughput, but limited CPU resources may incur high latency due to varying I/O size distributions (causing up to 1.9× tail latency variation). Thus, Omar prioritizes the migration of segments based on both traffic and average I/O size, calculated as $ios = \frac{t}{I}$. The high level indicates that the throughput is the primary bottleneck, so Omar prioritizes the traffic metric exclusively. In summary, Omar determines the current level based on the highest traffic of all blockservers. It uses the corresponding dominant metric to rank and select the top and bottom blockservers as the source and destination for segment migration, respectively.

Step 2: Selecting segment candidates. For candidate blockservers, Omar selects segments for migration. Since VDs have varying lifespans (Figure 12), Omar sets up a multi-level priority queue to group VDs by lifespan (e.g., 0-30 minutes and over 30 minutes), prioritizing the scheduling of long-lived VDs. The selection process is based on the expected remaining lifespan: given two VDs with lifespans a and b, we have $E[X - b|X > b] > E[X - a|X > a]$. Lower-priority (i.e., shorter-lived) segments are considered only if higher-priority (i.e., longer-lived) segments have all been migrated and the blockserver remains unbalanced. The rationale is that short-lived VDs are often bursty, and scheduling them for migration may cause further imbalance and destabilize destination blockservers.

Omar prioritizes read-dominant segments over write-dominant segments due to lower read traffic in Alibaba EBS, and subsequent scheduling of writes may disrupt the balance of read traffic. For read traffic, Omar selects segments with high read-to-write traffic ratios $\frac{t_r}{t_w}$ to limit subsequent write traffic rescheduling. Omar computes a read traffic score as follows:

$$Score_r = \alpha_r * t_r - (1 - \alpha_r) * t_w, \tag{1}$$

where α_r is the weight of read traffic t_r. For write traffic, Omar computes a QoS score to evaluate the service quality of each segment. It uses α_w to represent the weight on average write traffic. The formulation of the QoS score can be summarized as follows:

$$Score_w = \frac{(\alpha_w * t_w - (1 - \alpha_w) * s_w) * I_w}{l_w}, \tag{2}$$

which incorporates four components: (i) write traffic t_w, which quantifies the resource usage per segment; (ii) standard deviation of write traffic s_w, which avoids migrating bursty segments to prevent post-migration imbalance; (iii) IOPS I_w, which reflects I/O throughput; and (iv) latency l_w, which uses the reciprocal of latency per I/O $\frac{I_w}{l_w}$ to prioritize segments with higher reciprocal (i.e., better service quality).

Segments with higher $Score_r$ or $Score_w$ are prioritized for migration, as they are less likely to incur latency penalties.

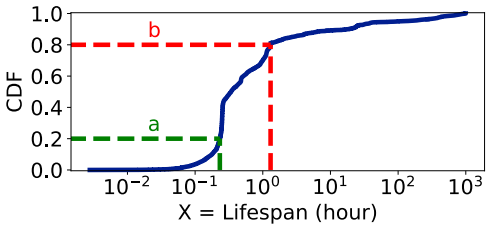


Figure 12. CDF of VD lifespans (§4.3).

Step 3: Concluding scheduling round. After confirming source/destination blockservers and segment candidates, Omar determines when to stop migrations (i.e., how many segments to migrate) in each scheduling round. It employs thresholds $1+T_b$ and $1-T_b$ to assess the balance degree of the dominant metric DM (from Step 1) via ratios $\frac{\max(DM)}{\text{mean}(DM)}$ and $\frac{\min(DM)}{\text{mean}(DM)}$, respectively. Migration stops when these ratios are below predefined thresholds. Based on field practice, Omar applies different balancing standards at varying blockserver traffic levels. Specifically, when the highest blockserver traffic is below 300 MB/s, there would be no migration. When the highest blockserver traffic lies between 300 MB/s and 800 MB/s, Omar adopts a loose balance threshold T_b=0.3 to avoid frequent scheduling, since both hardware and software resources of each blockserver are fairly abundant. When the highest blockserver traffic exceeds 800 MB/s, Omar uses T_b=0.1 to prevent insufficient throughput resources, while maintaining blockserver performance.

4.4 Scheduling Frequency Adjuster

The scheduling frequency (SF) significantly impacts the I/O tail latency, but its effect varies with workload dynamics. Omar dynamically adjusts SF based on recent performance statistics via the scheduler frequency adjustor. Specifically, Omar configures the monitoring window as 10 minutes since the cluster traffic is stable at this granularity. Within each window, Omar collects the number of schedules and the tail latency metrics (e.g., 4 KiB P999 write/read latency). To determine the optimal SF, Omar employs a *gradient descent* method by comparing the performance across two consecutive windows and estimating the gradient as $g = \frac{M_1 - M_0}{SF_1 - SF_0}$, where M_0 and M_1 are tail latency metrics and SF_0 and SF_1 are the scheduling frequencies of the two windows. To avoid local optima during gradient descent calculation, Omar incorporates the Adam optimizer [22], which is widely used in machine learning training, to adjust the first and second moments of the gradient. The scheduling frequency for the next window SF_2 is updated as $SF_2 = SF_0 + \text{optimizer.update}(g)$. Omar iterates the optimization process until convergence to the global optimum.

To control the scheduling frequency, Omar uses a token bucket approach, as in the Original scheduler, to limit the maximum number of schedules per second. While the number of scheduled segments is limited by the current remaining tokens, Omar can borrow additional tokens from future windows to smooth out bursty traffic. This borrowed portion of scheduling attempts, SF', is accounted for in the next time window, with an associated penalty γ. The adjusted scheduling frequency is expressed as: $SF_2 = (SF_0 + \gamma SF') + \text{optimizer.update}(g)$. This ensures additional tokens are allocated in subsequent windows. The design leverages our observation that cluster throughput is fairly stable at the minimum level, with latency performance primarily driven by the segment scheduler.

5 Implementation

We implemented Omar with around ~2500 lines of Python and C++ code. Omar's scheduler workflow is completely decoupled from the current blockmaster process and implemented as a plugin process. Specifically, Omar employs a shared memory mechanism to collect statistics from the blockmaster. Segment migrations are triggered using RPC calls after each scheduling decision. This design enables seamless integration into existing EBS systems and facilitates porting Omar to other systems with similar issues.

Data collection. Omar obtains and updates the VD-to-user mappings from VM subscription managers in the compute layer. Segment metrics, such as traffic and latency, are collected every second from all blockservers within a cluster. Due to memory constraints, we only keep track of the historical metrics in a 10-minute time window.

Parameter configuration. The weights of α_r and α_w in the score-based candidate selector (§4.3) are set to 0.7. As for the Adam optimizer, we set the learning rate lr to 0.001, the exponential decay rate for the first moment estimates β_1 to 0.9, and the exponential decay rate for the second moment estimates β_2 to 0.999, consistent with Pytorch defaults [8].

6 Evaluation

We evaluate Omar to answer the following questions:

- Does Omar outperform existing schedulers across various write ratios and numbers of VDs? (§6.1)
- Does Omar achieve better performance in large-scale deployment? (§6.2)
- How do Omar's individual components and parameters affect its performance? (§6.3)

Traces. We collect 30-minute I/O traces from 1000 high-loaded VDs running different applications in Alibaba EBS. Each entry includes five fields: VD, I/O type, Offset, Length, and Timestamp. The traces cover various workload patterns, including traffic bursts and write-/read-dominant VDs. We provide evaluation results for the most dominant I/O size: 64 KiB (56.0%). Other sizes, such as 4 KiB (7.3%) and 16 KiB (2.6%), show similar results. We also control the workload

pattern by selecting VDs to form different write ratios ranging from 0 to 1 in increments of 0.1.

Testbeds. For microbenchmarks, we use 8 compute nodes and 14 storage nodes (11 blockservers and 3 blockmasters). For macrobenchmarks, we use 40 compute nodes and 53 storage nodes (50 blockservers and 3 blockmasters), a typical production cluster setup. Each frontend compute node has 128 GB DRAM and an Intel(R) Xeon(R) Platinum 8331C CPU (2.50 GHz), and each blockserver has 128 GB DRAM, an Intel(R) Xeon(R) Silver 4114 CPU (2.20 GHz), and a 25 Gbps NIC. All experiments, except for macrobenchmarks (§6.2), use the microbenchmark setup.

Metrics. We evaluate Omar using the following metrics:

- *E2E latency* presents the fullstack end-to-end I/O request latency from the proxy to the storage layer, including the network and queuing delays.
- *CoV* quantifies load balancing across blockservers. We record the CoV of read/write traffic per scheduling interval and obtain the average over a 30-minute trace replay.
- *Number of schedules* is the total count of issued segment migrations. For schedulers with similar performance, the one with fewer schedules is preferred, as each migration incurs potential failures and segment unavailability.
- *VD per blockserver (VN)* records the number of unavailable VDs upon blockserver failures. During trace replay, we record the VN at 6-second intervals, and calculate the average and P95 VN over a 30-minute replay. A lower value indicates fewer affected VDs and higher system availability under blockserver outages.

Baselines. We compare Omar against the following:

- *GreedySpill* (GS) [32] is a file-system load balancer. While it is not designed for EBS, we adapt it for EBS by splitting the highest-traffic blockserver's load to neighbors.
- *Lunule* [39] balances metadata in Ceph clusters using linear regression for traffic prediction. We adapt it for EBS by calculating and redirecting segments for source/destination blockservers based on predictions.
- *WPM* [31] considers both traffic and VD distribution to control the scope of impact. We adapt it for EBS by considering the top 0.1% segments with the highest write traffic and computing scores based on the throughput and number of segments for each blockserver to trigger scheduling.
- *Original schedulers* include three variants, namely *Original* (currently used in deployment), *T-Std* (which extends Original to account for write-traffic standard deviation), and *T-Std-Rw* (which extends T-Std by evaluating both write and read traffic).
- *Throttle* is a non-scheduling strategy that caps write traffic at the block client, simulated by redistributing excess I/Os to other timestamps on the same blockserver with trace reorganization.
- *UpperBound* (UB) represents the ideal case with traces reorganized for even read/write traffic distribution across blockservers (i.e., each blockserver receives the same amount of reads/writes at all times). UB approximates *DiffForward* [50], which only balances write traffic with extra SSDs but not the same read/write traffic over time. Note that DiffForward is impractical since it requires a fundamental overhaul on hardware, software, and reliability modeling. Nevertheless, including UB enables us to examine the difference between Omar and the ideal case.

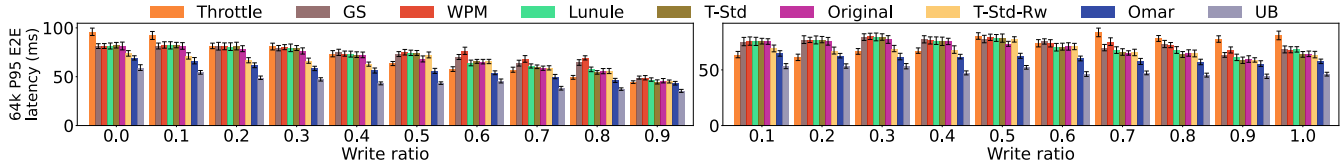
6.1 Microbenchmarks

Overall performance. Figure 13 presents the 64 KiB P95, P999, and P9999 E2E latency with write ratios ranging from 0 (read-only) to 1 (write-only). Omar shows 22.5%/23.3% lower read/write latency performance than UB, and is the best of the rest. Specifically, Throttle shows up to 2.1× higher P999 latency than Omar due to severe skewness (Figure 14b) since Throttle only caps traffic per blockserver without balancing. WPM incurs up to 40.3%/37.1% higher read/write P999 latency than Omar, as it focuses on write CoV and the scope of impact rather than latency. Similarly, GS only splits the extra traffic of the most-loaded blockserver to the peers, thereby creating new hotspots. Lunule underperforms by 37.8%/30.7% in read/write P999 latency, highlighting the difficulty of accurately predicting traffic patterns.

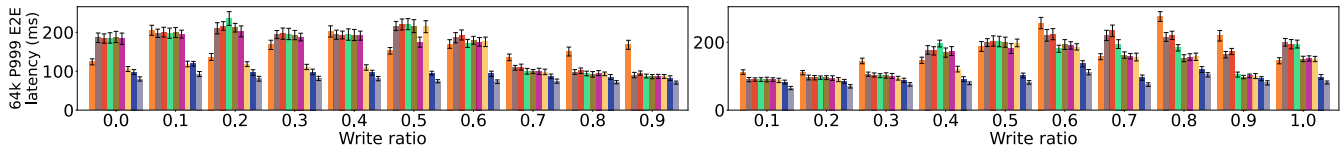
While T-Std improves upon these methods by incorporating multiple metrics, it still trails Omar by 28.8%/22.8% on average. T-Std-Rw shows better performance on P9999 latency than other candidates since it takes both traffic into consideration, but still shows 17.8%/15.9% worse read/write latency than Omar. In general, Omar outperforms Original in P9999 read/write latency by up to 41.8%/48.8%. Even when the traffic is write-dominant (i.e., a write ratio above 0.5), Omar still outperforms Original (38.5%/30.7% better in read/write latency). In addition, we observe that Omar raises the machine CPU utilization by around 1.5% on average due to an improved scheduling strategy.

VN. Figure 14a shows that Omar has 10.3%/10.2% higher average/P95 VN than Original, indicating a slight increase in the scope of impact. The reason is that Omar schedules at the segment granularity, which may spread more VDs across more blockservers to reduce schedules. Both Throttle and UB maintain initial VN (no migrations), while other candidates have comparable VN to Omar.

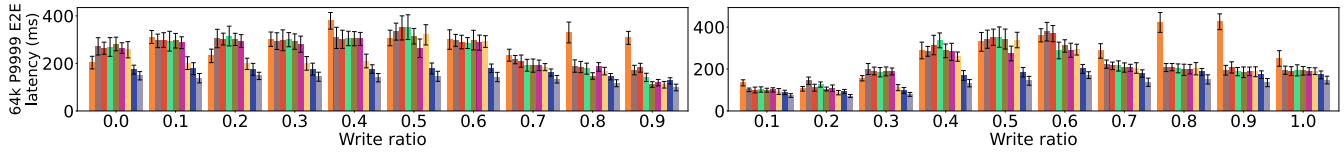
CoV. Figure 14b presents the CoV across write ratios. Throttle shows the worst CoV for both reads and writes due to no migration, while UB achieves near-zero CoV with already balanced traces. GS, WPM, Lunule, T-Std, and Original are suboptimal since they do not address read traffic. T-Std-Rw, at a write ratio of 0.8, achieves a 25.5% lower write CoV than Omar. However, a lower CoV does not ensure improved performance. In fact, its P999 latency is 23.9% higher. Overall, Omar achieves up to 37.7%/30.4% improvement on read/write CoV than Original.



(a) 64 KiB P95 read (left) and write (right) E2E latency. *Omar achieves up to 23.4%/19.8% improvement on read/write compared to Original.*

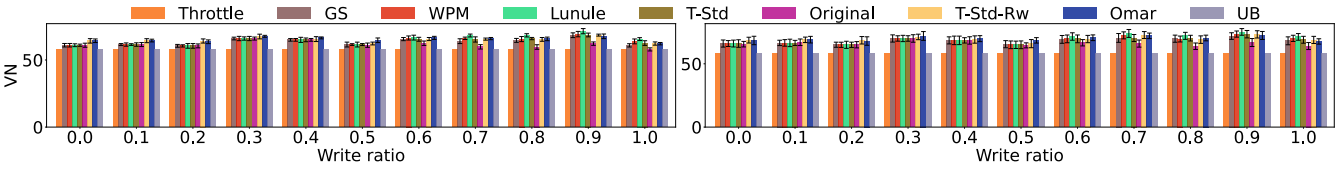


(b) 64 KiB P999 read (left) and write (right) E2E latency. *Omar achieves up to 52.2%/47.4% improvement on read/write compared to Original.*

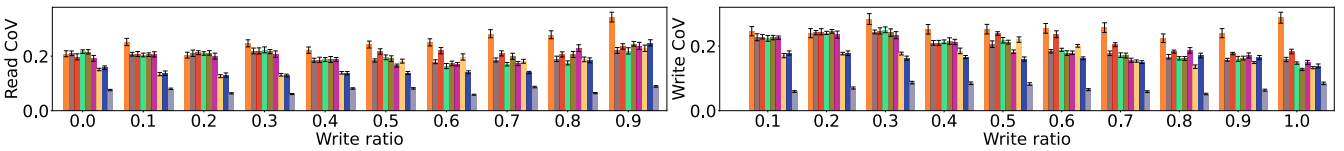


(c) 64 KiB P9999 read (left) and write (right) E2E latency. *Omar achieves up to 41.8%/48.0% improvement on read/write compared to Original.*

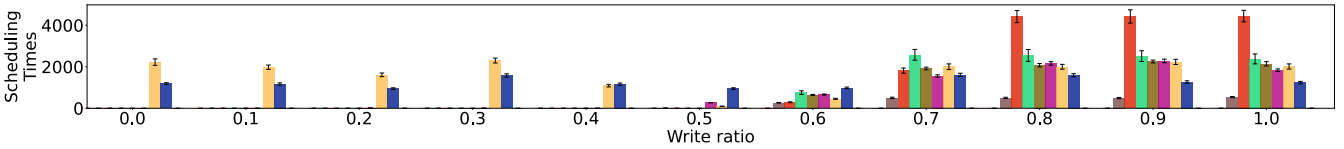
Figure 13. 64 KiB P95, P999, and P9999 E2E latency for different write ratios (§6.1).



(a) Average (left) and P95 (right) VN. *Omar exhibits only up to 10.3% and 10.2% more average/P95 VN compared to Original.*



(b) Read (left) and Write (left) CoV. *Omar achieves up to 37.7%/30.4% less read/write CoV compared to Original.*



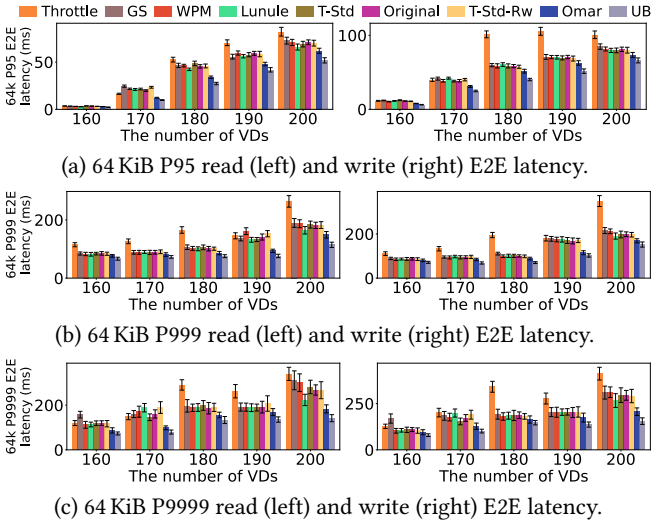
(c) Number of schedules. *Omar achieves up to 44.6% less schedules compared to Original when the write ratio is above 0.5.*

Figure 14. Average/P95 VN, read/write CoV, and number of schedules for different write ratios (§6.1).

Number of schedules. Figure 14c shows that Omar triggers 26.2%-48.5% fewer schedules than Original for write ratios above 0.5. The reason is that Omar distributes correlated segments proactively and schedules segments with stable statistics (e.g., lower standard deviation of traffic). As the write ratio increases, the number of schedules of WPM increases since it tends to schedule write-dominant segments. For write ratios below 0.5, most candidates rarely schedule any segment as the predefined write traffic threshold is difficult to reach (§3.2). Although T-Std-Rw schedules read traffic, it results in 46.8% more schedules than Omar. Throttle and UB are excluded as they do not migrate any segments.

Number of VDs. We use 160-200 high-loaded VDs with 6130-7334 segments to verify Omar's scalability. Each VD has 1.17 TiB of data on average. Each blockserver receives an average traffic of 55.6%-81.3% of the traffic threshold. As a result, Omar achieves up to 37.8%/31.6% improvements on 64 KiB read/write tail E2E latency compared to Original (Figure 15). T-Std-Rw shows drastically varying performance (e.g., high performance for 160 VDs, but low performance for 190 VDs) due to insufficient prior information like resonance, IOPS, and latency.

Production trace replay. We also evaluate Omar on I/O traces collected from three days, labelled as light (533.6 MB/s write traffic per blockserver), moderate (611.6 MB/s, the



(a) 64 KiB P95 read (left) and write (right) E2E latency.

(b) 64 KiB P999 read (left) and write (right) E2E latency.

(c) 64 KiB P9999 read (left) and write (right) E2E latency.

Figure 15. 64 KiB P95, P999, and P9999 E2E latency for a different number of VDs (§6.1).

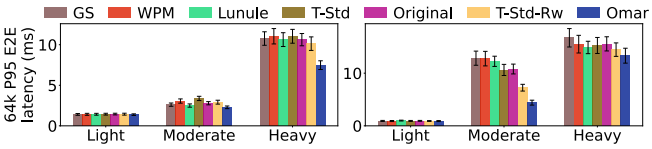
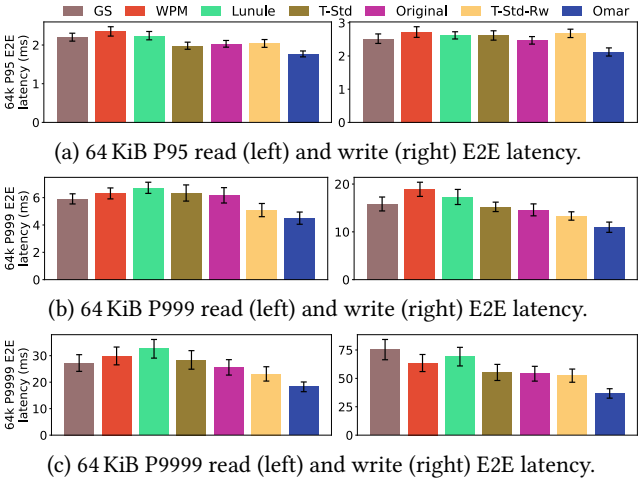


Figure 16. 64 KiB P95 read (left) and write (right) E2E latency in three days (§6.1). *Omar achieves up to 30%/60% performance improvement on read/write compared to Original.*

most common production scenario), and heavy (702.4 MB/s). Throttle and UB are excluded from our evaluation since they rely on trace reordering. Figure 16 shows that Omar achieves up to 30%/60% performance improvement in read/write 64 KiB P95 E2E latency compared to Original, validating Omar's robustness in production settings. Also, Omar outperforms other candidates, most notably under moderate traffic. The reason is that, in low traffic, hardware resources are sufficient to handle I/O requests and scheduling plays a less important role. In contrast, the heavy traffic can drain the resources on most blockservers, thereby reducing the benefits of migration.

6.2 Macrobenchmarks

We evaluate Omar on a cluster of 50 blockservers, including 600 high-loaded VDs with 28,732 segments, representing a standard production-scale configuration. As shown in Figure 17, Omar demonstrates clear advantages over all baselines. GS underperforms Omar by 51.2% on 64 KiB P9999 E2E latency. Similarly, WPM trails Omar by up to 42.1% due to excessive scheduling at each interval, deviating significantly from the optimal point of the performance curve described in §3.4. Lunule performs even worse than in its microbenchmark results, as the mixed I/O patterns in the macro-scale setup make traffic prediction more difficult. Even though



(a) 64 KiB P95 read (left) and write (right) E2E latency.

(b) 64 KiB P999 read (left) and write (right) E2E latency.

(c) 64 KiB P9999 read (left) and write (right) E2E latency.

Figure 17. 64 KiB P95, P999, and P9999 E2E latency of 600 VDs in a 50 blockservers cluster (§6.2).

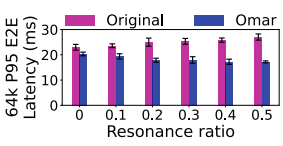


Figure 18. 64 KiB P95 write E2E latency with various resonance ratios (§6.3).

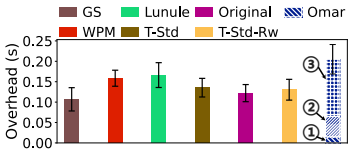


Figure 19. Time overhead of different schedulers (§6.3). *Omar is decomposed into three parts.*

T-Std and T-Std-Rw outperform other candidates, they are still 30.0% worse than Omar in P9999 E2E latency. Compared to Original, Omar reduces the 64 KiB read/write P95 latency by 10.7%/14.2%, P999 latency by 27.1%/25.8%, and P9999 latency by 28.7%/32.1%. These results further demonstrate the effectiveness of Omar's three-level scheduling design, which leverages runtime statistics to improve scheduling decisions.

6.3 Omar Deep Dive

We take a closer look at the effectiveness and parameter sensitivity in Omar's design.

Resonance ratio. The resonance ratio is defined as the proportion of VDs from the same user, a key criterion that affects Omar's performance. Figure 18 presents the 64 KiB P95 E2E write latency of Original and Omar with a resonance ratio ranging from 0 to 0.5 (results are similar for read). As a result, Omar reduces E2E latency by 11.5%-36.4% compared to Original. Even when the resonance ratio is zero (i.e., all VDs are from independent users), Omar still outperforms Original with the help of the score-based candidate selector and the frequency adjuster.

Overhead. Figure 19 shows the time needed to make scheduling decisions for each scheduler when replaying the same I/O traces. The time varies significantly when handling bursty traffic, as the number of segments requiring scheduling differs. For Omar, we decompose the total overhead into three

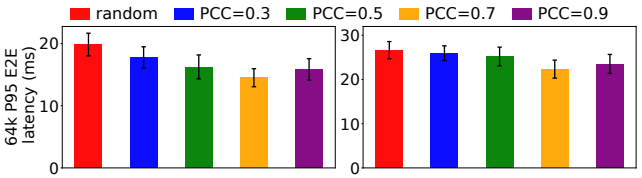


Figure 20. 64 KiB P95 read (left) and write (right) E2E latency under random placement and different PCC thresholds (§6.3).

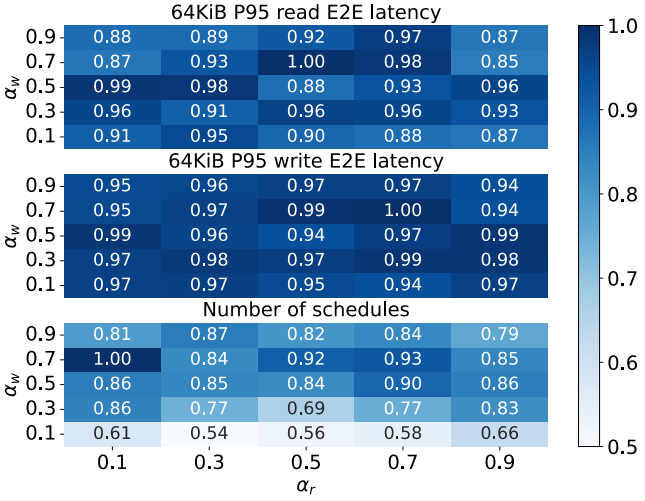


Figure 21. Parameter sensitivity study (§6.3). *The higher the value, the better the performance.*

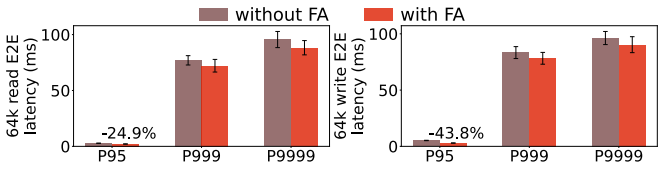
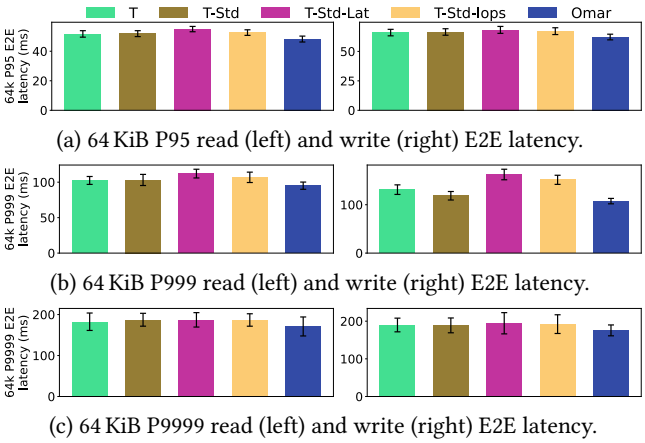


Figure 22. 64 KiB P95, P999, and P9999 E2E latency with and without the frequency adjuster (§6.3).



(a) 64 KiB P95 read (left) and write (right) E2E latency.

(b) 64 KiB P999 read (left) and write (right) E2E latency.

(c) 64 KiB P9999 read (left) and write (right) E2E latency.

Figure 23. 64 KiB P95, P999, and P9999 E2E latency of various segment scores (§6.3). *Omar achieves up to 15.3%/33.9% improvement on read/write compared to others.*

parts corresponding to the three components in §4.1. The resonance-based allocator (①) and frequency adjuster (②) operate at hourly and minute-level intervals, respectively, and hence incur negligible overhead. In contrast, the score-based candidate selector (③) runs at second-level intervals, and contributes a more noticeable overhead of 0.14 s. Nevertheless, the average overhead for all schedulers remains within around 0.25 s, which adheres to the real-time performance requirement in scheduling design (§2.3).

PCC threshold. We evaluate Omar under different PCC thresholds along with a random placement baseline (i.e., randomly distributing 20% of segments while constraining them to their blockservers). As shown in Figure 20, random shows the worst 64 KiB P95 latency performance since it does not consider the resonance feature. PCC=0.3 and PCC=0.5 demonstrate 10.7%-18.5% worse performance compared to Omar, primarily due to their distribution of non-resonance segments. PCC=0.9 also exhibits 8.4% worse than Omar, as it excessively restricts the opportunity to exploit resonance.

Parameter sensitivity. Omar primarily relies on two parameters: α_r and α_w. By iterating through different values, we further study parameter sensitivity and plot the performance heatmap in Figure 21. Note that we normalize the results based on the best-performing cell. As a result, E2E tails remain rather stable conditioned on different parameter choices. For the number of schedules, the impact of varying α_w is more pronounced. The reason is that a small α_w may drive Omar to schedule segments with more stable but milder traffic, thereby increasing the number of schedules. Conversely, a larger α_w only considers the traffic without the stability information, causing the "ping-pong" effect.

Frequency adjuster. We evaluate Omar with and without the scheduling frequency adjuster (FA). As shown in Figure 22, we replay the same trace for both scenarios. Omar with FA achieves up to 25.0%/43.9% improvement on 64 KiB read/write P95 E2E latency, and outperforms in P999 and P9999 latencies compared to Omar without FA. This demonstrates the effectiveness of dynamically adjusting the scheduling frequency to optimize latency performance.

Different segment scores. We conduct an ablation study to evaluate the effectiveness of each component in Omar's score-based candidate selector with them enabled/disabled. We evaluate Omar and its variants with different scoring metrics: T (only traffic), T-Std (traffic and std), T-Std-Lat (traffic, std, and latency), and T-Std-Iops (traffic, std, and IOPS). As shown in Figure 23, T-Std-Lat shows the worst performance in all percentile latencies, since the latency metric may change drastically over time, where Omar includes IOPS to smooth over latency. In general, Omar demonstrates clear performance improvement over other schedulers, achieving

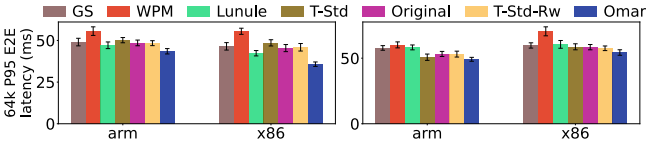


Figure 24. 64 KiB P95 read (left) and write (right) E2E latency for different hardware settings (§6.3).

Table 1. Key configurable parameters of Omar (§7).

Parameter	Meaning
T_{cc}	Threshold of the PCC correlation to determine resonance (§4.2).
Tpt_1	Throughput threshold for low level scheduling (§4.3).
Tpt_2	Throughput threshold for high level scheduling (§4.3).
L	Lifespan threshold for the priority queue (§4.3).

up to 15.3%/33.9% improvement on 64 KiB read/write E2E tail latency.

Generalization. We adopt two main machine architectures in the deployed clusters of blockservers: x86 (90%) and ARM (10%). We evaluate Omar on both architectures by sampling a 12-blockserver cluster from each, and compare the results under two different hardware settings by replaying the same I/O trace. As shown in Figure 24, Omar achieves the best performance on both architectures (up to 21.3%/7.7% improvement on 64 KiB read/write P95 E2E latency compared to Original), justifying Omar's robustness across hardware.

7 Discussion

Hard-coded parameters. Table 1 presents Omar's key configurable parameters. We elaborate on our parameter tuning strategy as follows. T_{cc} is configured to 0.7 to optimize tail latency performance (Figure 20). Tpt_1 is set to 300 MB/s since it is the average throughput of all deployed EBS clusters, and Tpt_2 is set to 800 MB/s since it is 80% of the traffic control limit of a blockserver. L is set to 1 hour because 70% of short-lived VDs are deleted within this duration (Figure 12). Omar can adapt to different deployment scenarios (e.g., hardware upgrades) through a grid-based parameter search approach.

Deployment experience. We have deployed Omar in a production Alibaba EBS cluster for a month, and plan to gradually scale out to all clusters. We observe improvements on tail latency performance and system stability (i.e., fewer schedules). We elaborate on our deployment experience for Omar as follows. First, even if the average traffic at the minute level seems balanced, the burst traffic imbalances at the millisecond level can result in varying CPU utilization and QoS queuing delays. We plan to collect fine-grained information of burst traffic and distribute them accordingly. Second, slow I/Os (e.g., end-to-end latency exceeding 1 s) can occur due to abnormal distributed file system behaviours and slow segment loading. Thus, we have implemented inspection logic before segment scheduling, including status verification of the distributed file system, to mitigate slow I/O occurrences.

8 Related Work

Cloud block storage. Several studies have advanced the design and performance of cloud block storage systems. Lithium [19] and LSVD [18] propose a log-structured VD design to improve block I/O performance for virtualization workloads running in clouds. SepBIT [38] implements an optimized data placement algorithm to mitigate write amplification in EBS. URSA [48] proposes a hybrid block store for efficient cloud services. Calcspar [49] proposes a contract-aware mechanism to mitigate latency spikes under constrained IOPS budgets in Amazon's EBS. Zhang et al. [47] present the evolution of Alibaba's EBS over the years. Shu et al. [35] propose a burstable I/O scheduler to address inter- and intra-thread load balancing within EBS compute nodes. Wu et al. [43] investigate traffic skewness across EBS stacks. In contrast to these studies that address complete EBS system designs, Omar focuses on the traffic scheduling problem within EBS.

Load balancing. Prior studies address load balancing in EBS. WPM [31] schedules high-traffic segments for EBS load balancing. Li et al. [27] analyze load balancing challenges in EBS systems. DiffForward [50] differentiates burst write traffic at the forwarding layer and adopts a decentralized distributed log store for load balancing. VDMig [45] predicts short-term burst traffic in VDs for scheduling. Load imbalance is also prevalent in emerging applications, such as large language model (LLM) serving systems [24, 42, 46].

However, these approaches have limitations. WPM [31] is evaluated by simulations only without considering real-world I/O performance, and fails to consider read traffic in its design. DiffForward [27] is limited in three aspects: (i) mitigating burst writes only (but not burst reads), (ii) relying on hardware-dependent implementation with significant I/O stack modifications, and (iii) lacking data durability guarantees during blockserver failures. VDMig [45] works at the VD granularity and cannot be readily extended for fine-grained segment-level traffic prediction. Omar optimizes both read and write I/Os with minimal overhead.

9 Conclusion

This paper introduces Omar, a proactive and reactive mixed scheduler for Alibaba EBS. The design of Omar focuses primarily on three components: the resonance-based allocator, the score-based candidate selector, and the scheduling frequency adjuster. Extensive evaluations show that Omar achieves high robustness and scalability, effectively decreases the tail latency, and reduces the number of schedules.

Acknowledgments

We thank all the anonymous reviewers and our shepherd, Jacob Gorm Hansen, for their valuable comments, and the EBS team for their timely support. This work is supported in part by the Natural Science Foundation of China (62072306, 623B2072), Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3, Alibaba Research Fellowship (ARF) and Alibaba Innovative Research (AIR) project.

References

[1] 2025. Alibaba Cloud - Elastic Block Storage. https://www.alibabacloud.com/en/product/disk.
[2] 2025. Amazon Elastic Block Store. https://aws.amazon.com/ebs.
[3] 2025. Amazon S3. https://aws.amazon.com/s3.
[4] 2025. Azure Disk Storage. https://azure.microsoft.com/en-us/products/storage/disks.
[5] 2025. Hadoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[6] 2025. Kubernetes Load Balancer. https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer.
[7] 2025. Nginx. https://nginx.org/en.
[8] 2025. Pytorch. https://pytorch.org/.
[9] 2025. TiDB Load Balancer. https://docs.pingcap.com/tidb/dev/tiproxy-load-balance.
[10] Hervé Abdi. 2010. Coefficient of Variation. *Encyclopedia of research design* 1, 5 (2010), 169–171.
[11] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: Peak prediction-driven resource overcommitment in datacenters. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*.
[12] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control*. John Wiley & Sons.
[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*.
[14] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009).
[15] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.
[16] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: Proactive, liveput-optimized DNN training on preemptible instances. In *Proceedings of the 21th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[17] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the 13th European Conference on Computer Systems conference (EuroSys)*.
[18] Mohammad Hossein Hajkazemi, Vojtech Aschenbrenner, Mania Abdi, Emine Ugur Kaynar, Amin Mossayebzadeh, Orran Krieger, and Peter Desnoyers. 2022. Beating the I/O bottleneck: A case for log-structured virtual disks. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*.
[19] Jacob Gorm Hansen and Eric Jul. 2010. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*.
[20] Jiyao Hu, Zhenyu Zhou, and Xiaowei Yang. 2022. Characterizing physical-layer transmission errors in cable broadband networks. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[21] Pawel Janus and Krzysztof Rzadca. 2017. SLO-aware colocation of data center tasks based on instantaneous processor requirements. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*.
[22] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[23] Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang. 2024. Optimizing resource allocation in hyperscale datacenters: Scalability, usability, and experiences. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.
[25] 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 14th European Conference on Computer Systems (EuroSys)*.
[26] Huiba Li, Yiming Zhang, Haonan Wang, and Ping Zhong. 2020. URSAL: Ultra-efficient, reliable, scalable, and available block storage at low cost. In *Proceedings of the 2020 IEEE Conference on Computer Communications (INFOCOM)*.
[27] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2023. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. *ACM Transactions on Storage (TOS)* 19, 2 (2023).
[28] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. 2023. More than capacity: Performance-oriented evolution of Pangu in Alibaba. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*.
[29] Yuan Liu, Wenxin Li, Wenyu Qu, and Heng Qi. 2022. BULB: Lightweight and automated load balancing for fast datacenter networks. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*.
[30] Ziyang Liu, Renyu Yang, Jin Ouyang, Weihan Jiang, Tianyu Ye, Menghao Zhang, Sui Huang, Jiaming Huang, Chengru Song, Di Zhang, Tianyu Wo, and Chunming Hu. 2024. Kale: Elastic GPU scheduling for online DL model training. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC)*.
[31] Haoyu Mao, Yongkun Li, Wenzhe Zhu, Fei Li, and Yinlong Xu. 2022. On optimizing traffic imbalance in large-scale block-based cloud storage: Trace analysis and algorithm design. In *Proceedings of the IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*.
[32] Swapnil Patil and Garth Gibson. 2011. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*.
[33] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*.

[34] Jiuchen Shi, Kaihua Fu, Jiawen Wang, Quan Chen, Deze Zeng, and Minyi Guo. 2024. Adaptive QoS-aware microservice deployment with excessive loads via intra-and inter-datacenter scheduling. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 35, 9 (2024).

[35] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable cloud block storage with data processing units. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[36] Difan Tan, Jiawei Li, Hua Wang, Xiaoxiao Li, Wenbo Liu, Zijin Qin, Ke Zhou, Ming Xie, and Mengling Tao. 2025. Tela: A temporal load-aware cloud virtual disk placement scheme. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[37] Hua Wang, Yang Yang, Ping Huang, Yu Zhang, Ke Zhou, Mengling Tao, and Bin Cheng. 2020. S-CDA: A smart cloud disk allocation approach in cloud block storage system. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*.

[38] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*.

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

[40] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. 2024. Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[41] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI)*.

[42] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *Proceedings of the 18th USENIX symposium on operating systems design and implementation (OSDI)*.

[43] Haonan Wu, Erci Xu, Ligang Wang, Yuandong Hong, Changsheng Niu, Bo Shi, Lingjun Zhu, Jinnian He, Dong Wu, Weidong Zhang, Qiuping Wang, Changhong Wang, Xinqi Chen, Guangtao Xue, Yichao Chen, and Dian Ding. 2025. Hey Hey, My My, Skewness is here to stay: Challenges and opportunities in cloud block store traffic. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*.

[44] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. 2024. Load is not what you should balance: Introducing Prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[45] Guangjie Xing, Shuheng Gao, Hua Wang, Ke Zhou, Yaodong Han, and Mengling Tao. 2024. VDMig: An adaptive virtual disk migration scheme for cloud block storage system. In *Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD)*.

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-based generative models. In *Proceedings of the 16th USENIX symposium on operating systems design and implementation (OSDI)*.

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

[48] Yiming Zhang, Huiba Li, Shengyun Liu, and Peng Huang. 2023. Hybrid block storage for efficient cloud volume service. *ACM Transactions on Storage* 19, 4 (2023).

[49] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. 2023. Calcspar: A contract-aware LSM store for cloud storage with low latency spikes. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC)*.

[50] Wenzhe Zhu, Yongkun Li, Erci Xu, Fei Li, Yinlong Xu, and John CS Lui. 2023. DiffForward: On balancing forwarding traffic for modern cloud block services via differentiated forwarding. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*.

A Artifact Appendix

A.1 Abstract

Omar is a novel proactive and reactive mixed scheduler for cloud block storage. It employs a resonance-based allocator, a score-based candidate selector, and a scheduling frequency adjuster for tail latency performance improvement.

A.2 Hosting

The artifact source code is available on GitHub at https://github.com/Master-Chen-Xin-Qi/EuroSys26_AE. The dataset can be accessed through the Alibaba Tianchi platform at https://tianchi.aliyun.com/dataset/210122. The packaged code and data files are available on Zenodo at https://zenodo.org/records/17197572.

A.3 Content

This artifact includes the prototype implementation of Omar, and the source code to reproduce the main results shown in the paper. For each figure, we provide data files and Python scripts to reproduce its result. In addition, the artifact includes block I/O traces collected from 1,000 VDs in Alibaba EBS. Please refer to the README.md file in the repository for more details.

A.4 Hardware Dependencies

As Omar is a scheduling framework designed for production EBS systems, it requires approximately 10 compute nodes, 10 blockservers, 1 blockmaster, and 10 storage nodes to conduct minimal tests.

A.5 Software Dependencies

Omar cooperates with the block client in compute layer, the entire EBS stack, and the underlying distributed file system. We decouple Omar from the blockmasters as an independent process to enhance portability and facilitate deployment across different environments. Please refer to the README.md file in the repository for more details on Omar's adaptation.

B Technical Supplement

B.1 I/O Size Distribution

As shown in Figure 25, we present the top five I/O sizes (i.e., 4 KiB, 8 KiB, 16 KiB, 64 KiB, and 128 KiB) probability distribution of the collected I/O traces. The 64 KiB I/O size is the most common, accounting for 56.0% of the total I/O records. The 4 KiB and 16 KiB I/O sizes account for 7.3% and 2.6% of the total I/O records, while 8 KiB and 128 KiB I/O sizes account for 2.1% and 10.9%, respectively.

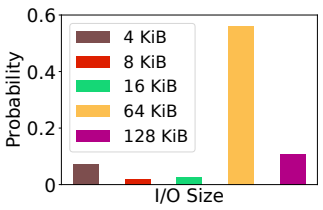


Figure 25. Distribution of the top five I/O sizes in the collected I/O traces (§B.1).

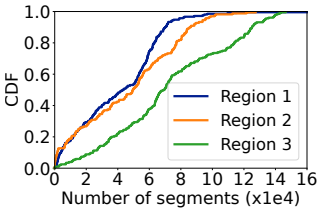


Figure 26. Number of segments in each cluster from three regions (§B.2).

Algorithm 1: Omar scheduling procedure.

Input: All blockservers BSs, read and write traffic of blockservers $BS_t = \{BS_r, BS_w\}$, VD traffic VD_u of the same user u, all segments $Segs$.
Output: Scheduling decisions of segments Dec.

// **Step1: Find resonant VDs at hourly intervals**
$corr_w, corr_r \leftarrow PearsonCorr(VD_u)$
for $corr$ *in* $[corr_w, corr_r]$ **do**
  $G_p, G_n \leftarrow BuildGraph(corr)$
  $G_{pc}, G_{nc} \leftarrow FindConnectedComponent(G_p, G_n)$
  for *group* g_p *in* G_{pc} **do**
    for *segment* s_p *in* g_p **do**
      $Dec.append(Split(s_p, BSs))$
  for *group* g_n *in* G_{nc} **do**
    for *segment* s_n *in* g_n **do**
      $Dec.append(Merge(s_n, BSs))$
// **Step2: Choose BS and segments at second intervals**
for IO_t *in* $[read, write]$ **do**
  while $!Balance(BS_t, IO_t)$ **do**
    $BS_{min}, BS_{max} \leftarrow FindMinMax(BS_t, IO_t)$
    $seg_s \leftarrow ComputeSegScore(BS_{max}, IO_t)$
    $Dec.append(Migrate(max(seg_s), BS_{min}, BS_{max}))$
// **Step3: Adjust scheduling frequency at minute intervals**
$optimizer = AdamOptimizer(lr = 0.001, \beta1 = 0.9, \beta2 = 0.999)$
$M_0, M_1 \leftarrow CollectMetrics(Segs)$
$SF_0, SF_1 \leftarrow CollectSchedFreq(Segs)$
$g = \frac{M_1 - M_0}{SF_1 - SF_0}$
$SF_2 = SF_0 + optimizer.update(g)$
return Dec

B.2 Segment Distribution

Figure 26 presents the number of segments in each cluster from three main EBS regions (each with hundreds of clusters). The number of segments varies significantly across different clusters, ranging from a few hundred to several thousand. This highly skewed distribution of segments necessitates designing a scheduler with enhanced scalability and robustness.

B.3 Proof of Lifespan Estimation

We provide a proof of the lifespan estimation discussed in §4.3.

Theorem B.1. *For any $a < b$ following the lifespan CDF in Figure 12, we have $E[X - b \mid X > b] > E[X - a \mid X > a]$.*

Proof. $E[X-b|X>b] > E[X-a|X>a] = \frac{\int_b^\infty (x-b)f(x)dx}{P(X>b)} - \frac{\int_a^\infty (x-a)f(x)dx}{P(X>a)} = \frac{P(X>a)\int_b^\infty (x-b)f(x)dx - P(X>b)\int_a^\infty (x-a)f(x)dx}{P(X>b)P(X>a)}$

Since: $\frac{P(X>a)}{P(X>b)} = \frac{1-P(X<=a)}{1-P(X<=b)} > \frac{\int_a^\infty (x-a)f(x)dx}{\int_b^\infty (x-b)f(x)dx}$

Therefore: $\frac{P(X>a)\int_b^\infty (x-b)f(x)dx - P(X>b)\int_a^\infty (x-a)f(x)dx}{P(X>b)P(X>a)} > 0$

Thus, we conclude: $E[X-b|X>b] > E[X-a|X>a]$ □

B.4 Put it All Together

Algorithm 1 outlines the pseudocode for Omar's complete scheduling process. Omar first proactively detects the resonant VD groups belonging to the same user. It distributes the resonant segments across different blockservers and consolidates the negatively correlated segments within the same blockserver (lines 1-11). Subsequently, Omar evaluates the balance state among blockservers under the current traffic conditions, prioritizing read before write. Then it schedules segments according to their score (lines 12-17). Finally, it fine-tunes the scheduling frequency by using the Adam optimizer based on the collected performance metrics (lines 18-23), and returns the migration decisions. Note that the number of schedules in each round is constrained by the current scheduling frequency.