ByteTier: A Cloud-Scale Pool-Based Tiering System for Heterogeneous Cloud Storage

Xiuqi Huang[‡|], Qingsong Zhu[†|*], Yaohua Tang[†], Honghao Wang[†], Xin Wei[†], Jinxin Liu[†], Guanyu Liu[†], Zongjia Chen[†], Tianyue Lan[†], Yunlong Wang[†], Hang Zhang[†], Rusong Jiao[†], Yuanwang Yue[†], Jie Ding[†], Xiaodong Luo[†], Zijin Han[†], Huan Sun[†], Maoyin Liu[†], Wei Chen[‡*], Chengyu Dong[†], Rui Shi[†]

[†]*ByteDance Inc.* [‡]*State Key Lab of CAD&CG, Zhejiang University*

Abstract

Heterogeneous cloud storage relies on tiering to balance performance and cost. However, intra- and inter-node tiering face three major challenges: limited flexibility in redundancy and scaling; excessive disk and network traffic amplification; inefficiency in handling small I/Os. We propose *ByteTier*, a pool-based tiering system. Based on decoupled performance and capacity pools, *ByteTier* designs a cross-pool two-phase write mechanism: writes are cached and stored in the performance pool in the foreground, and re-encoded from the cache to the capacity pool in the background, providing flexibility and reducing disk and network overhead. Within this mechanism, *ByteTier* performs a workload-aware I/O aggregation method, adaptively merging small writes and batching them into device-friendly patterns to enhance throughput and reduce latency. *ByteTier* has been deployed in production at ByteDance for three years, serving diverse workloads. Evaluation shows that it significantly reduces both average and P99 write latency while improving write and read throughput.

1 Introduction

Cloud service providers (CSPs) deliver services such as media hosting, interactive applications, and data analytics [7]. To support these services, CSPs widely employ distributed file systems (DFSs) [21, 27, 28, 31] as the common storage foundation for large-scale clusters. These DFSs must support diverse workloads with distinct storage requirements, including Object Storage, which prioritizes cost-efficient scalability for massive unstructured data; Online Analytical Processing (OLAP), which requires high-throughput sequential scans and aggregations; and Block Storage, which demands low-latency, high-IOPS random access to hot data.

As these workloads are increasingly consolidated onto shared DFS infrastructure, production deployments must balance throughput, latency, and capacity within cost and availability constraints [24, 29]. A fundamental approach is tiering (or hybrid) storage, which leverages heterogeneous hardware: SSDs provide low-latency, high-IOPS access, while HDDs offer lower cost per TB and high sequential bandwidth [22].

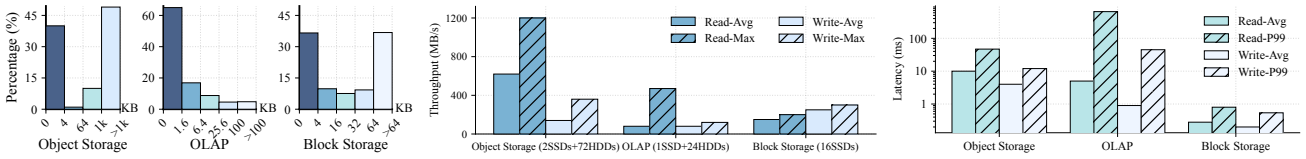
Tiering Designs in Practice. Existing DFS deployments mainly follow two tiering architectures. Intra-node tiering colocates SSDs and HDDs within each node and migrates data locally [3, 30, 32]. Inter-node tiering separates SSD and HDD resources across nodes and performs cross-node data migration and re-encoding in the background [16, 19, 35]. ByteStore, ByteDance's append-only DFS (similar to Azure WAS [8], Colossus [11], and Pangu [19]), serves diverse production workloads including Douyin, Toutiao, and Volcengine cloud services. As of 2026, ByteStore manages tens of exabytes of data across heterogeneous storage clusters.

Challenges & Limitations. ByteStore has evolved from intra-node tiering to inter-node tiering and further introduced resource pooling, progressively improving deployment flexibility. The lack of a unified way to manage data placement and transformation has led to several fundamental limitations.

① *Tight coupling between SSD and HDD restricts redundancy and scaling flexibility.* In intra-node tiering, SSDs and HDDs remain coupled in data placement and redundancy configurations, which typically requires a uniform erasure coding (EC) strategy [2]. It prevents SSDs from adopting narrow EC stripes that optimize foreground performance, while HDDs cannot leverage wide EC stripes that reduce storage cost. Moreover, to simplify cluster management and balance traffic, the SSD-to-HDD ratio is usually fixed [17, 23, 26, 33]. As workload demands evolve, this rigid configuration results in the over-provisioning of certain storage media during scale-up and incurs repair overhead during scale-down.

② *Data migration incurs significant disk and network traffic amplification.* Inter-node tiering enables SSDs and HDDs to adopt different redundancy schemes, thereby offering flexibility in balancing performance and cost [12]. However, an additional read pass as well as an extra network traversal are introduced during re-encoding and dumping. This results in significant disk and network traffic amplification: data must

[|]Co-first authors. [*]Co-corresponding authors. [|*]Emails: huangxiuqi@zju.edu.cn, zhuqingsong.0909@bytedance.com, chenvis@zju.edu.cn



(a) I/O size distribution (b) Throughput of a DataNode (c) Latency characteristics

Figure 1: Three representative workloads in ByteDance production clusters

be fully read from SSDs, transferred across the network, and re-encoded before being written to HDDs [16, 17, 19]. Such amplification not only burdens foreground I/O performance but also reduces cluster throughput, making it unsuitable for latency-sensitive or high-throughput services.

③ *Mismatch of workload and media characteristics leads to inefficient small I/O handling.* Production workloads often generate small, fragmented writes [18]. On SSDs, such requests are often unaligned with the device's Logical Block Address (LBA) boundaries, triggering read-modify-write operations that amplify write traffic and inflate tail latency [36]. Efficient mitigation requires cross-file and cross-user aggregation to reshape these small writes into device-friendly patterns [9,15,28,31], which can also reduce random I/O pressure on HDDs to sustain throughput. However, existing intra-node and inter-node designs struggle to achieve efficient I/O merging without introducing complex logical-to-physical mapping overheads and additional consistency mechanisms.

Our Solution. To address these challenges, we design and implement *ByteTier*, a pool-based tiering framework built atop ByteStore that provides a unified abstraction for managing data placement and transformation across heterogeneous storage media. *ByteTier* elevates resource pooling into a tiering abstraction, enabling SSDs and HDDs to scale independently and adopt tailored redundancy configurations. To eliminate migration-driven amplification, *ByteTier* introduces a two-phase write mechanism: user writes are first persisted in the performance pool with lightweight redundancy, then asynchronously re-encoded and applied to the capacity pool from the in-memory cache, avoiding redundant disk reads and extra network transfers. Within this mechanism, to further address inefficient I/O patterns, *ByteTier* incorporates a workload-aware I/O aggregation method including an adaptive foreground write merger that reshapes small and fragmented writes into SSD-aligned entries and a background migration-free applier that organizes data with large EC units suitable for DFS files, thereby improving write and read throughput while reducing tail latency.

Contributions. *ByteTier* has been deployed in thousands of clusters with tens of thousands of nodes for ByteDance's multiple representative storage workloads over three years. The contributions are summarized as follows:

- A pool-based tiering abstraction that provides a unified interface for managing data placement and transformation from the performance pool to the capacity pool, enabling flexible scaling and tailored redundancy strategies.
- A two-phase write mechanism that eliminates the read-migrate cycle in conventional tiering designs, significantly reducing disk and network traffic amplification.
- A workload-aware I/O aggregation method that transforms small and fragmented writes into device-efficient access patterns, improving device efficiency and reducing latency.
- Comprehensive evaluations demonstrate the effectiveness of *ByteTier* across diverse storage workloads, improving throughput and reducing latency for both write and read.

2 Background and Motivations

This section provides the production context of *ByteTier*. We first review common tiering architectures, then characterize ByteStore workloads and further extend the motivations.

2.1 Production Storage Tiering Architectures

DFSs are the universal storage substrate in CSPs, abstracting heterogeneous devices into a unified namespace. A typical DFS consists of *MetaNodes* that maintain metadata (namespace, chunk mapping, access control) and *DataNodes* that store file chunks and serve I/O requests [21]. Production DFSs [8, 11, 19], including ByteDance's ByteStore [4], follow append-only semantics: data is written sequentially and not updated in place. To exploit heterogeneous media such as SSDs and HDDs, these systems commonly adopt either intra-node tiering or inter-node tiering [16, 30, 32, 35]. These deployment choices expose important trade-offs in write aggregation, redundancy transition, and migration overhead.

Table 1: Comparison of redundancy-transition designs

System	Write Aggregation	Redundancy Transition	Migration Path
Lustre HSM	Page-cache-assisted	Single-copy archive	Inter-node
GPFS [27]	Per-file aggregation	Policy-driven re-layout	Inter-/intra-node
AWS S3 [6]	Service-managed	Replica→EC	Inter-node
Azure WAS [8]	Service-managed	Replica→EC	Inter-node
Colossus [11]	Per-file aggregation	Placement/layout transition	Inter-node
Pangu [19]	No explicit aggregation	Narrow→wide EC	Inter-/intra-node
ByteTier	Cross-file aggregation	Narrow→wide EC	Write-path apply

Tab. 1 qualitatively compares representative write-path redundancy-transition designs in production storage systems.

Existing systems typically perform redundancy transition after data persistence through lifecycle management, migration, or background reorganization, whereas *ByteTier* integrates redundancy transition into the write-open phase.

2.2 Diverse Workloads in Production

ByteStore serves multiple storage services as the shared DFS substrate, primarily including Object Storage, OLAP, and Block Storage, each with distinct I/O characteristics and performance goals. Object Storage prioritizes high throughput with bounded tail latency. OLAP emphasizes low query latency and high read throughput. Block Storage mainly handles small, latency-critical I/Os, needing stable IOPS under load.

Fig. 1 summarizes production observations. The I/O size distributions (Fig. 1a) vary significantly: Object Storage exhibits a bimodal pattern with both small and large requests, OLAP is dominated by sub-KB accesses, and Block Storage primarily issues sub-64KB I/Os. Their throughput and latency characteristics (Fig. 1b&1c) further highlight distinct bottlenecks across workloads. Such diversity hinders intra- or inter-node tiering from effectively serving all workloads simultaneously. In ByteDance's previous practice, services adopt separate tiering configurations, complicating storage stack development and maintenance.

2.3 Extended Motivations

Driven by the workload diversity above, production deployments of intra- and inter-node tiering reveal three fundamental limitations. Fig. 2 illustrates the I/O paths and data transformations for comparison.

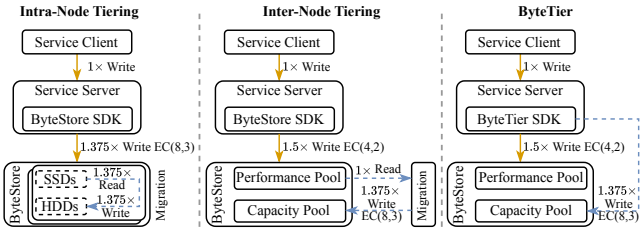


Figure 2: Intra-, inter-node tiering, and *ByteTier* comparison

Redundancy and scaling flexibility. As shown in Fig. 2 left, intra-node tiering tightly couples SSDs and HDDs within a node, binding redundancy configuration and resource scaling. As EC stripes are distributed across nodes, data migrated from SSDs to HDDs cannot be easily reorganized (e.g., re-encoding or compression). As a result, systems often adopt wide EC stripes (e.g., EC(12,3)) with small EC units (e.g., 2KB) to balance space efficiency and low latency, which in turn hurts small-write latency and large-read performance. Moreover, the fixed SSD-to-HDD ratio limits elasticity: scaling out for write bursts needs hybrid nodes, even when HDD capacity is underutilized. Furthermore, scaling down incurs significant overhead associated with repairing parity blocks.

Reducing network traffic amplification overhead. Inter-node tiering allows changing redundancy schemes during data migration, but incurs significant disk and network traffic amplification. As shown in Fig. 2 center, latency-sensitive workloads can adopt a low-latency redundancy scheme (e.g., EC(4,2)) when persisting data on the performance pool, and are later migrated, re-read, and re-encoded before being written to the capacity pool (e.g., EC(8,3)). For Block Storage, traffic includes foreground write (1×), EC encoding (1.5×), background read (1×), and EC re-encoding (1.375×). With 50% garbage-collection overhead, total network traffic amplification reaches 6.06×, and remains 4.04× even with a 60% compression ratio. Such traffic amplification increases disk pressure and limits system throughput.

Efficient small I/O handling. Small, fragmented writes are inefficient for both SSDs and HDDs. On SSDs, misaligned writes (e.g., sub-4KB writes to a 4KB flash page) trigger costly read-modify-write cycles, leading to write amplification and reducing SSD lifespan. Inline metadata (e.g., 32-byte checksum headers) further breaks 4KB alignment, resulting in overlapping writes and reduced SSD throughput. On HDDs, small I/Os can disrupt sequential streaming and increase disk seek times. Furthermore, small I/Os cannot fill the entire EC stripe, requiring additional zero padding, increasing storage and metadata overhead.

These observations motivate *ByteTier*'s new tiering design. As shown in Fig. 2 right, *ByteTier* exposes performance and capacity pools through a unified abstraction. With its two-phase write mechanism, *ByteTier* reduces redundant disk reads and cross-pool network traffic.

3 ByteTier Overview

Building on the abstraction motivated above, *ByteTier* is guided by three principles: ① *Pool-based abstraction for flexibility.* Rather than managing heterogeneous media directly at service servers, *ByteTier* builds on ByteStore's pool organization and provides a unified abstraction for data placement and transformation across performance and capacity pools. This abstraction enables more flexible scaling and redundancy strategies. ② *Migration-free staging to reduce traffic amplification.* Traditional tiering suffers from high disk and network overheads due to repeated reads for re-encoding during migration. *ByteTier* reduces this amplification by staging data in in-memory caches in the foreground and applying cached data directly to capacity pools in the background, thus avoiding redundant I/O and network transfers. ③ *I/O reshaping for efficient small writes.* Small and unaligned writes amplify write traffic on SSDs and fragment sequential throughput on HDDs. *ByteTier* reshapes such writes through foreground merging and background batching, aligning I/O patterns with device characteristics and significantly reducing tail latency.

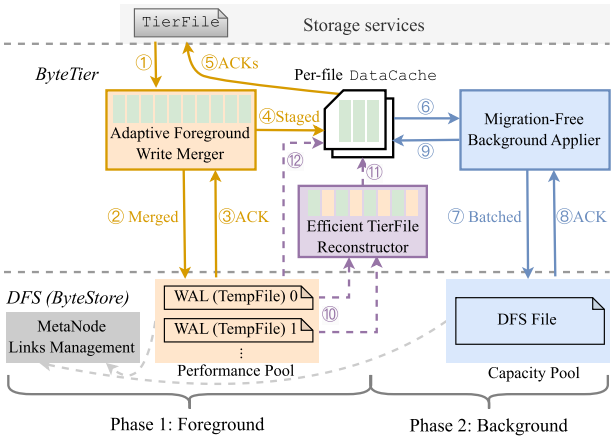


Figure 3: *ByteTier* Architecture

At a high level, as shown in Fig. 3, *ByteTier* is deployed in the service layer above the underlying DFS (ByteStore in production), acting as a DFS client while providing a pool-based abstraction to upper-layer services. ByteStore already manages metadata, provides namespace consistency, and organizes different storage media across DataNodes into pools with intra-pool load balancing by MetaNodes. Building on this substrate, *ByteTier* exposes a new `TierFile` interface to upper-layer services. Each `TierFile` is an append-only abstraction that transparently spans performance and capacity pools, and may consist of multiple Write-Ahead-Log (WAL) files in the performance pool and one DFS file in the capacity pool. The `TierFile` uses WAL files to buffer user write requests and eventually writes the data to the DFS file. This design hides device heterogeneity and redundancy differences, enabling direct leverage of pool-based tiering without modifying I/O logic. When opening a `TierFile`, users can specify configuration options (e.g., `storage_type`) to configure the DFS file. WAL file configurations can also be customized by users, with *ByteTier* providing default settings.

ByteTier's core is its two-phase write mechanism. In the foreground phase, user writes from each I/O thread are adaptively merged and quickly persisted into WAL files in the performance pool, while simultaneously staged in per-file `DataCache` in memory, ensuring low-latency acknowledgments and immediate data availability. In the background phase, staged data is periodically flushed to DFS files in the capacity pool with large, re-encoded batches, enabling efficient layouts of wider EC stripes and larger EC units without foreground interference. By staging instead of migrating, *ByteTier* eliminates redundant disk reads and network transfers, reducing traffic amplification while improving efficiency.

To realize these design principles within the two-phase write mechanism, *ByteTier* incorporates three key components to realize the workload-aware I/O aggregation method:

- *Adaptive Foreground Write Merger*: reshapes small and fragmented writes into device-friendly, aligned I/Os. Foreground I/O threads follow a run-to-completion model, with each thread handling multiple `TierFiles` in a polling loop. Write requests from multiple `TierFiles` are merged into WAL entries (①), which are promptly appended to WAL files (②). `ACK` (③) is returned once WAL persistence is complete, ensuring low latency for upper layers.
- *Migration-Free Background Applier*: eliminates redundant SSD reads and cross-pool network transfers by staging and directly applying data. Once persisted in WAL files, data is staged in a per-file `DataCache` (④) and becomes immediately readable (⑤). When the cache exceeds a size or time threshold, background threads batch-apply the data to the capacity pool (⑥). During this process, *ByteTier* configures wider EC stripes and larger EC units (⑦), improving storage efficiency. After applying (⑧), cached data is purged to free memory (⑨), and WAL files are deleted to minimize performance pool space usage.
- *Efficient TierFile Reconstructor*: ensures fast recovery and availability under failures or file movement. Instead of full WAL file scans, which incur prohibitive I/O overhead, *ByteTier* reconstructs state using metadata from WAL checkpoints and extended attributes (Xattr) (⑩) that record unapplied data ranges, while a WAL meta cache is built in memory to further avoid repeated WAL reads. After reconstructing `DataCache` (⑪), *ByteTier* reads user data from WAL files (⑫) in the background and serves new `TierFile` I/Os concurrently.

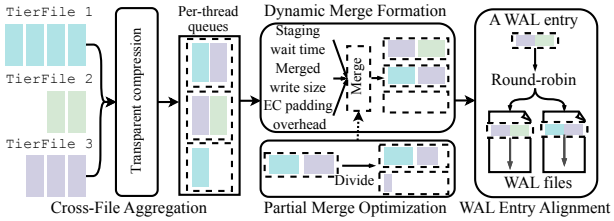


Figure 4: Adaptive Foreground Write Merger

4 Implementation Details

This section details the implementation of *ByteTier*, focusing on three core components: *Adaptive Foreground Write Merger* in §4.1, *Migration-Free Background Applier* in §4.2, and *Efficient TierFile Reconstructor* in §4.3.

4.1 Adaptive Foreground Write Merger

As shown in Fig. 4, *Adaptive Foreground Write Merger* aggregates writes from multiple `TierFiles`, merges them dynamically, and constructs WAL entries that are persisted to multiple WAL files in parallel. The merger consists of: cross-file write aggregation, which collects writes from multiple `TierFiles` to increase merging opportunities; dynamic merge formation, which determines when staged writes should be consolidated

into WAL entries; partial merge optimization, which divides writes as needed to better match the EC stripe size; and WAL entry alignment, which aligns WAL entries with the foreground EC stripe size to prevent overlapping writes. These work together to balance latency, throughput, and write traffic amplification, optimizing the performance pool's efficiency.

4.1.1 Cross-File Write Aggregation

To mitigate latency overheads caused by CPU context switches and cache misses in multi-threaded execution, *ByteTier* adopts a run-to-completion model. Hardware resources, including CPU cores and HugePage, are bound to individual I/O threads. All write requests of a given `TierFile` are executed exclusively on a single thread. This design preserves cache locality and minimizes scheduling overhead, while also leveraging high-performance networking techniques such as user-space TCP and RDMA.

Within each I/O thread, write requests from multiple `TierFiles` are pushed into a shared per-thread staging queue. Pooling writes across files significantly increases opportunities for merging compared to per-file processing, enabling cross-file aggregation to collect more data within a given staging interval. It particularly benefits workloads dominated by small writes, where a single file may not generate sufficient traffic to form larger WAL entries.

Cross-file aggregation further integrates transparent compression to reduce space and bandwidth overheads. When this feature is enabled, *ByteTier* employs hardware accelerators such as Intel QuickAssist Technology [10] to compress incoming write requests before they are enqueued. By performing compression early, it reduces staging queue occupancy and improves effective write throughput.

4.1.2 Dynamic Merge Formation

After writes are staged in the per-thread queue, *ByteTier* dynamically determines when they should be consolidated into a WAL entry. The formation balances three constraints: staging wait time, merged write size, and EC padding overhead, which are dynamically adjusted according to current write throughput. For throughput-intensive workloads, *ByteTier* prefers longer staged times and larger merged sizes to maximize merging efficiency. For latency-sensitive workloads, a shorter staged time and a smaller merged size are chosen to ensure timely write completion. Besides, small or misaligned writes usually need extra zero padding to fill an EC stripe, incurring write traffic amplification. *ByteTier* considers the expected EC padding overhead of current writes and selectively merges with lower traffic amplification.

Algo. 1 outlines the decision process. Based on the current throughput (`cur_thp`), it derives the expected staging time (`need_stage`) from a set of predefined throughput gears configured by upper-layer services (lines 1-3). If the actual staging time exceeds this constraint, all staged writes and the current write are immediately merged and written (lines 4-5). Otherwise, it evaluates the staged size (`prev_sz`, `cur_sz`) and padding size (`prev_pad_sz`, `cur_pad_sz`) of both the previous and current write. If the previous write produces less EC padding overhead and already meets the minimum mergeable size (`MIN_SZ`), all previous staged writes will be merged to a new WAL entry, and the current write is staged (lines 7-8). Conversely, if the current write meets `MIN_SZ` and its EC padding overhead remains within the allowed bound (`MAX_WA`), the current write is merged and committed immediately (lines 9-10). If neither condition is satisfied, the current write is held in the staging queue for future merging (line 11).

Algorithm 1: Dynamic Merge Formation

```
Input: cur_thp, prev_sz, prev_pad_sz, cur_sz, cur_pad_sz,
       cur_stage
1  for (i,g) in enumerate(thp_gears) do
2      if cur_thp belong to g then
3          need_stage ← gears_stage[i];
4  if cur_stage ≥ need_stage then
5      return MERGE_AND_WRITE
6  else
7      if prev_pad_sz < cur_pad_sz and (prev_sz +
         prev_pad_sz) ≥ MIN_SZ then
8          return QUEUED_AND_WRITE
9      else if (cur_sz + cur_pad_sz) ≥ MIN_SZ and MAX_WA
         ≥ (cur_sz+cur_pad_sz)/cur_sz then
10         return MERGE_AND_WRITE
11     return QUEUED
```

4.1.3 Partial Merge Optimization

To further minimize padding overhead, *ByteTier* introduces a partial merge optimization to divide writes to better fit the EC stripe size. If a set of writes requires excessive padding, one of the writes will be selectively divided into two sub-writes: one to complete the current WAL entry without padding, and one deferred for the next merge. This reduces write traffic amplification and improves effective throughput.

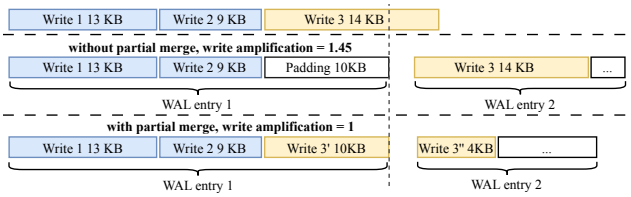


Figure 5: Example of partial merge optimization

As shown in Fig. 5, there are three consecutive writes staged for merging. Without partial merge optimization, the

WAL entry (32KB) formed by write 1 and write 2 would require an extra 10KB padding to align with the EC stripe. With partial merge optimization, write 3 is divided into two sub-writes: write $3'$ (10KB) and write $3''$ (4KB). Write 1, write 2, and write $3'$ are merged to form a no-padding WAL entry 1 with write amplification reduction, while write $3''$ is staged for further merging to form WAL entry 2.

4.1.4 WAL Entry Alignment

Once a merge decision is made, the selected staged writes are consolidated into a WAL entry. In practice, upper-layer writes are not always aligned with the SSD LBA size, and distributed file systems often append small inline metadata to each block. These factors may cause subsequent writes to overlap previously written LBAs under high concurrency.

To avoid such overlap, *ByteTier* aligns WAL entries with the foreground EC stripe size while accounting for inline metadata to prevent overlapping writes and latency spikes. For example, when a 32B DataNode inline footer is appended to each 4KB block, the effective data region per block is set to $(4\text{KB} - 32\text{B})$, ensuring that WAL entries remain LBA-aligned after the footer is appended.

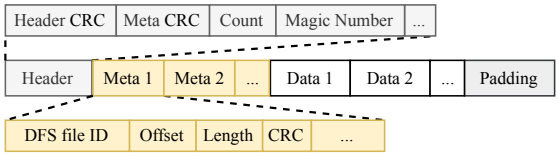


Figure 6: WAL entry structure

As shown in Fig. 6, a WAL entry begins with a header containing key fields: *Cyclic Redundancy Check (CRC)* for checking information (*Header CRC*, *Meta CRC*), *Count* of write requests within this entry, and *Magic Number* identifying the WAL entry. Following the header, metadata for each user write is stored sequentially, including *DFS file ID* that uniquely identifies the DFS file, *Offset* and *Length* of the user data within the DFS file, and *CRC* of the user data. Subsequently, the actual user writes are appended in the order corresponding to their metadata. If necessary, a small amount of padding is added at the end to fill the EC stripe.

Finally, constructed WAL entries are distributed across multiple WAL files in a round-robin manner, preventing write contention on a single file and improving throughput. Once an entry is persisted, several `ACK`s corresponding to `TierFiles` are immediately returned to the upper-layer services, ensuring low-latency responsiveness while the data is simultaneously staged in the in-memory `DataCache` for subsequent background apply. Since concurrent writes may be persisted out of order across different WAL files, *ByteTier* additionally enforces that the `ACK` order for each `TierFile` strictly follows the enqueue order of its writes.

4.2 Background Migration-Free Applier

Background Migration-Free Applier asynchronously flushes staged data from the per-file `DataCache` into DFS files in the capacity pool, which is entirely transparent to users. Besides, directly applying the staged data from `DataCache` enables wider EC stripes and larger EC units without interfering with foreground I/O. During apply, to ensure both efficiency and flexibility, *ByteTier* coordinates: per-file batch control decides when to trigger apply; two-step apply process minimizes HDD contention; and WAL lifetime management promptly releases performance pool space.

4.2.1 Per-File Batch Control

Applying data to DFS files minimally impacts foreground I/Os, allowing *Background Migration-Free Applier* to wait longer and batch more writes, thereby improving I/O efficiency. However, excessively prolonging this waiting time can introduce two issues. First, the `DataCache` may consume a lot of memory, potentially causing resource contention or even out-of-memory (OOM) errors. Second, WAL files cannot be deleted until their data is fully applied, occupying critical space in the performance pool. This trade-off highlights the need to tune the data application time according to workload characteristics to balance memory and storage efficiency with the benefits of batched writes.

The per-file batch control employs two constraints for each `TierFile` to trigger background apply: (i) the size constraint: unapplied data in the `DataCache` exceeds a predefined threshold; and (ii) the time constraint: the elapsed time since the last application exceeds a threshold. Once either constraint is met, the per-file batch control (⑥ in Fig. 7) triggers the two-step apply for flushing cached data to the corresponding DFS file, releasing memory and performance pool space. Furthermore, if *ByteTier* detects that the `DataCache` or performance pool is nearing exhaustion, it relaxes the above two constraints to reduce space usage in the cache or pool.

4.2.2 Two-Step Apply Process

HDDs suffer from IOPS bottlenecks due to long seek times, and performing apply in `O_SYNC` mode that forces the data and file metadata, even journals in the Ext4 file system, to be persisted synchronously, further degrades foreground read performance due to disk contention. To mitigate this, the two-step apply process minimizes the active sync times and leverages the inherent aggregation capability of the file system to reduce disk contention and improve throughput under concurrent operations.

In the first step (⑦-1 in Fig. 7), data is written to the local file system's page cache via buffered I/O in the DataNode. The data is released from `DataCache` while the `DataCache` index is preserved. This index tracks metadata for each `TierFile` write, including the WAL file ID, offset, and length. In the

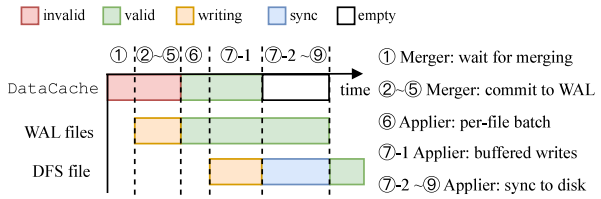


Figure 7: Two-step apply in *ByteTier*'s data flow: labels ①-⑨ correspond to those depicted in Fig. 3.

second step (⑦-2 in Fig. 7), the page cache is flushed to the HDD using ByteStore's `Sync` interface. This step is delayed after file close (`delay sync`), relying on the flush workers of the operating system to minimize sync frequency and further reduce disk contention.

During this process, reads can be served without waiting for durable sync: from `DataCache` in memory (⑥–⑦-1) and from WAL files in the performance pool (⑦-2–⑨). If an irrecoverable failure occurs, the DFS file is truncated to the last successful sync offset. The affected data is then re-read from the WAL file based on the `DataCache` index, refilled into `DataCache`, and re-applied.

After these two steps are completed, the `DataCache` index is deleted, and the corresponding WAL data is marked as fully applied. Writes are acknowledged only after WAL persistence, and WAL data is released only after durable DFS sync, ensuring recoverability throughout the apply lifecycle. This two-step apply process ensures efficient I/O, reliable recovery, and balanced memory usage, while minimizing HDD contention during concurrent applies.

4.2.3 WAL Lifetime Management

Since WAL files are utilized for temporary data persistence and considering the limited capacity of the performance pool, a WAL file should be promptly deleted once all its data has been applied. *ByteTier* manages WAL lifetimes through the common DFS link/unlink feature of ByteStore, which maintains associations between WAL files and DFS files.

Each write to a `TierFile` is first persisted in a WAL file and linked to its corresponding DFS file. These links form a many-to-many relationship: a single WAL file may contain data for multiple DFS files, and a single DFS file may span multiple WAL files. For example, when a user writes to `TierFile 1`, the request is persisted in WAL file A and staged for applying to DFS file 1. If no link exists, *ByteTier* establishes a link between WAL file A and DFS file 1. Once all data belonging to DFS file 1 in WAL file A has been applied, the link is removed. When a WAL file has no remaining links and is closed, the MetaNode immediately deletes it to reclaim performance pool space. This lifetime management ensures that WAL files remain temporary, avoiding long-lived consumption of performance pool capacity.

4.3 Efficient TierFile Reconstructor

When machine failures (e.g., host crashes or process exits) or file movements (e.g., cluster upgrades or hot data migration) occur, the in-memory `DataCache` of `TierFile` is lost. In such cases, *ByteTier* must scan the linked WAL files to reconstruct the metadata of unapplied data before the `TierFile` can resume service. Without additional support, this reconstruction requires scanning GB-scale WAL files, causing severe I/O overhead and delaying file availability.

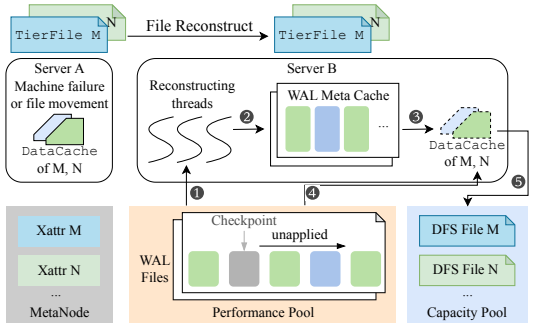


Figure 8: Efficient TierFile Reconstructor

As shown in Fig. 8, *Efficient TierFile Reconstructor* incorporates WAL file checkpoints, WAL block footers with a WAL meta cache in failure recovery, and Xattr in file movement to minimize scan overhead and accelerate reconstruction. *ByteTier* launches multiple reconstructing threads to scan all linked WAL files in parallel (❶). Metadata of user writes are added to the WAL meta cache to avoid repeated reads (❷). When the `DataCache` index is successfully rebuilt (❸), reconstruction completes, and the `TierFile` can serve new I/Os. *ByteTier* reads data from the WAL files (❹) in the background and applies the cached data to the DFS file concurrently (❺).

4.3.1 Metadata-Guided Failure Recovery

To avoid high costs of full-file scans and reduce reconstruction time for failure recovery, *ByteTier* enriches WAL files with auxiliary metadata that guides reconstruction, including WAL file checkpoints and WAL block footers. Moreover, *Efficient TierFile Reconstructor* keeps an in-memory WAL meta cache to reuse parsed WAL blocks shared by multiple `TierFiles`.

WAL File Checkpoints. Avoiding scanning of applied data in the WAL files can significantly reduce reconstruction time. Therefore, *ByteTier* periodically records checkpoints in WAL files that contain the range of applied data for all linked DFS files. During recovery, the checkpoint allows the reconstructor to skip WAL file regions that have already been applied to the capacity pool. Since multiple checkpoints may coexist in a WAL file, with older ones being out-of-date, the footer of each WAL block (see below) stores the checkpoint offset of the nearest checkpoint. During recovery, *Efficient TierFile*

Reconstructor reads the last block footer to obtain the latest valid WAL file checkpoint directly.

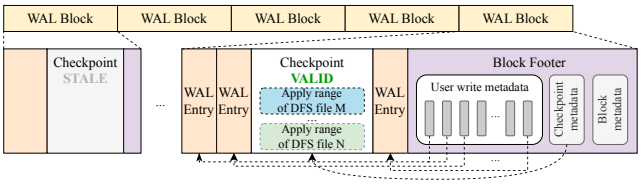


Figure 9: WAL block structure

WAL Block Footers. Each WAL file is partitioned into multiple fixed-size WAL blocks, thereby localizing recovery to MB-scale regions rather than the entire file. As shown in Fig. 9, each WAL block contains a sequence of WAL entries and ends with a block footer that consolidates their metadata. The metadata for all user writes within the block's WAL entries is recorded as five-tuples: <*offset in WAL file, length in WAL file, DFS file ID, offset in DFS file, length in DFS file*>. If a checkpoint exists within the block, the footer updates the position of the latest checkpoint; if not, it inherits the checkpoint metadata from the previous block. Even if the block footer is incomplete, falling back to block scans remains cheaper than performing WAL file scans.

WAL Meta Cache. During recovery, *Efficient TierFile Reconstructor* parses block footers and populates an in-memory WAL meta cache. Since multiple `TierFiles` often share the same WAL blocks, cached metadata avoids redundant reads of the same footer when reconstructing subsequent `TierFiles`. To ensure efficient memory utilization, the cache is managed with a Least Recently Used (LRU) policy, which retains frequently accessed footers while evicting stale ones. Besides, multiple block footers can be loaded in parallel to further reduce the reconstruction time.

By combining checkpoints, block-level footers, and metadata caching, *Efficient TierFile Reconstructor* reduces recovery from a GB-scale scan to a metadata-guided process, ensuring that `TierFiles` become serviceable quickly after failures.

4.3.2 Accelerated File Movement with Xattr

Besides failure recovery, reconstruction is also required during file movement, when ownership of a `TierFile` is transferred across servers. *Efficient TierFile Reconstructor* utilizes the Xattr to accelerate file movement.

Before movement, the source server will stop new writes to `TierFile`. *ByteTier* knows the exact range of the unapplied data in the WAL files and records the range in the Xattr of DFS. At the beginning of the movement, the `TierFile` tries its best to apply data to the DFS file in the source server; meanwhile, the destination server starts to take ownership of the `TierFile`. When the destination server succeeds in taking ownership of the `TierFile`, *Efficient TierFile Reconstructor* can directly scan the data according to the range in the Xattr, minimizing unnecessary WAL file reads.

5 Evaluation

We evaluate *ByteTier* using real workloads and controlled experiments on production-configured storage clusters. We first measure end-to-end performance by generating workloads from Object Storage, OLAP, and Block Storage services. We then evaluate system-level properties, including elasticity under dynamic workloads and service availability during failure recovery and file movement. Finally, we perform a design breakdown to comprehensively quantify the impact of *ByteTier*'s key components.

5.1 Performance under Production Workloads

We evaluate *ByteTier* under representative production workloads from three major ByteStore-backed services: Object Storage, OLAP, and Block Storage. Experiments run on production-configured storage clusters: Object Storage uses 16 DataNodes (each 2 SSDs, 72 HDDs), OLAP uses 14 DataNodes (each 1 SSD, 24 HDDs), and Block Storage uses 14 DataNodes (each 16 SSDs). We conduct end-to-end evaluations by generating production workloads through the corresponding service clients and measure client-visible performance. Comparisons within each workload use matched cluster settings and the prior ByteStore production configuration as the baseline.

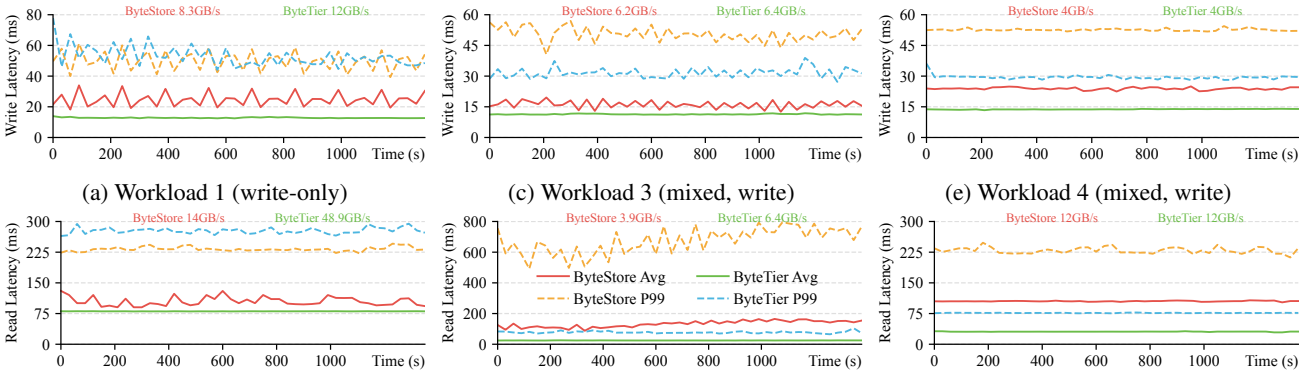
5.1.1 Object Storage Performance

We evaluate *ByteTier* using four representative production Object Storage workloads, including one write-only, one read-only, and two mixed read-write. The baseline is ByteStore deployed with its original intra-node tiering architecture. Tab. 2 summarizes the full-run throughput and latency, while Fig. 10 shows the time-series trends of average and P99 latency.

Workload 1 (Fig. 10a) is a write-only workload with variable request sizes and strict latency requirements. Compared with ByteStore, *ByteTier* improves write throughput by 44.6% and reduces average write latency by 48.2%. Workload 2 (Fig. 10b) is a read-only model inference workload with high concurrency. *ByteTier* improves read throughput by 3.5× while reducing average read latency by 23.3%. Both P99 latencies increase slightly due to higher throughput.

Workloads 3&4 (Fig. 10c&10f) involve mixed reads and writes, where the former is album service traffic with random I/O sizes (500KB–2.5MB writes and 300–700KB reads), and the latter is a model training workload with larger I/Os (average ∼1.5MB, under business flow control). *ByteTier* reduces both average (P99) read and write latency by 28–88%.

Across all workloads, *ByteTier* consistently delivers substantial performance improvements. ByteStore is constrained



(a) Workload 1 (write-only) (c) Workload 3 (mixed, write) (e) Workload 4 (mixed, write)

(b) Workload 2 (read-only) (d) Workload 3 (mixed, read) (f) Workload 4 (mixed, read)

Figure 10: Average and P99 latency over time for ByteStore and *ByteTier* under four production Object Storage workloads

Table 2: Write/read throughput and latency of ByteStore and *ByteTier* under four Object Storage workloads

Metric	Architecture	Workload 1	Workload 2	Workload 3	Workload 4
Throughput (GB/s)	ByteStore	8.3/-	-/14	6.2/3.9	4/12
	ByteTier	12/-	-/48.9	6.4/6.4	4/12
Avg. Latency (ms)	ByteStore	24.7/-	-/105.0	16.1/132.7	23.9/105.2
	ByteTier	12.8/-	-/80.5	11.4/24.7	13.8/30.7
P99 Latency (ms)	ByteStore	49.8/-	-/ 232.5	50.5/673.7	52.6/229.2
	ByteTier	52.7/-	-/277.3	31.6/77.7	29.4/76.5

Note: A dash (-) indicates the metric is not applicable.

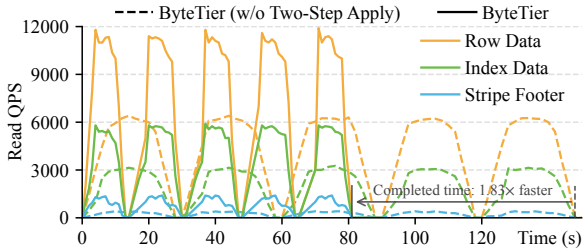


Figure 11: Read QPS and query completion time for OLAP

by intra-node tiering, which requires wide EC stripes and small EC units, leading to significant read QPS amplification for large reads. In contrast, *ByteTier* decouples foreground and background EC schemes through its two-phase write mechanism. This allows larger EC units for DFS files (§4.2) while merging small writes in the foreground (§4.1), greatly reducing I/O amplification. During the experiments, we have observed up to 3.8× lower HDD read IOPS and 3× lower SSD traffic on DataNodes.

5.1.2 OLAP Performance

OLAP workloads are dominated by large analytical scans over massive datasets. A typical query may consist of one or several SQL statements and may scan up to TB-level data (e.g., querying logs from the past seven days). The data is stored in ORC format, and each ORC stripe contains row data, index data, and a stripe footer. The OLAP engine uses index data to prune irrelevant row groups, parses stripe footers for metadata, and scans only the required column data. As a result, it generates massive sequential reads on capacity pool storage, making read efficiency critical to query performance.

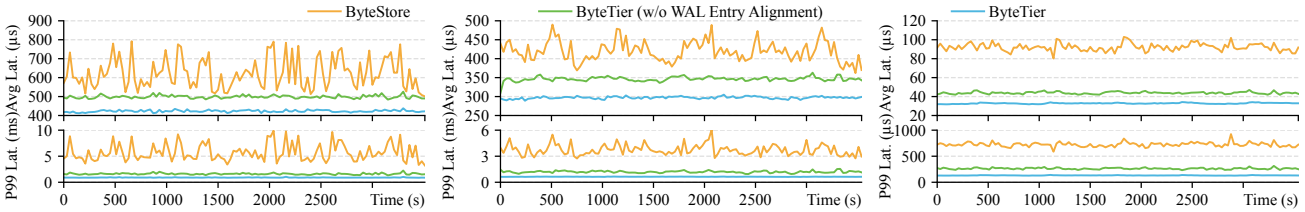
Since the two-step apply (§4.2.2) determines DFS file layout and effective read granularity, we evaluate *ByteTier* with and without this mechanism. Fig. 11 reports read QPS by ORC component (stripe footer, index data, and row data) of the representative production query sets in five runs.

With two-step apply enabled, effective scan throughput increases from about 10GB/s to 18GB/s, reducing query completion time from over 30 seconds to around 15 seconds, achieving a 1.83× speedup in total. This improvement comes from larger sequential reads enabled by supporting configuring a larger EC unit size (e.g., 1MB) in DFS files. During the experiment, we observe that the average HDD read I/O size increases by up to 4×, enhancing disk bandwidth utilization and accelerating query execution.

5.1.3 Block Storage Performance

We generate production Block Storage workloads through the service stack and measure write latency at three points along the I/O path: service clients, *ByteTier*, and DataNodes. We compare *ByteTier* with ByteStore using inter-node tiering (three-way replication→EC(8,3)) and with a *ByteTier* variant without WAL entry alignment (§4.1). The workload spans approximately one hour, with metrics sampled every 30s.

Fig. 12 shows the average and P99 write latency over time. *ByteTier* consistently maintains lower latency than the baseline and exhibits significantly reduced tail-latency fluctuations. At service clients (Fig. 12a), *ByteTier* reduces average latency from 622.6µs to 423.0µs and lowers P99 from 5710µs



(a) Block Storage Service Clients (b) *ByteTier* (DFS Clients) (c) DataNodes

Figure 12: Average and P99 write latency over time for Block Storage at three points along the I/O path

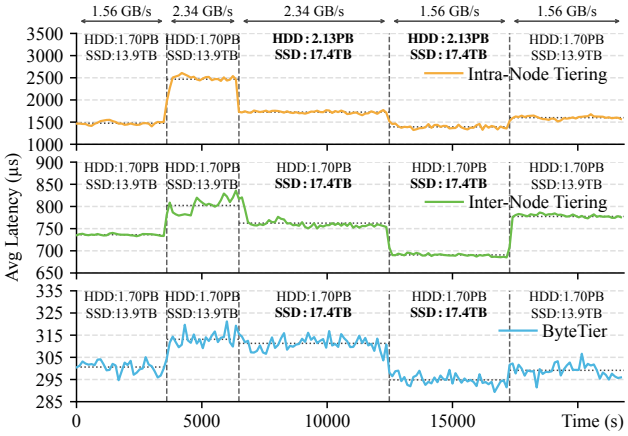
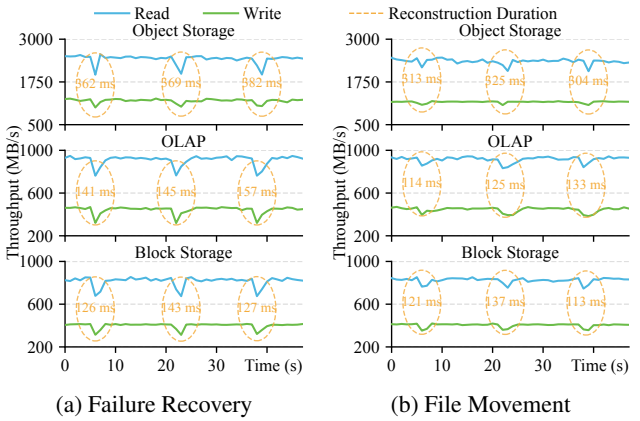


Figure 13: Average write latency and allocated storage of tiering architectures during scaling with fluctuating workloads



(a) Failure Recovery (b) File Movement

Figure 14: Read and write throughput of *ByteTier* during `TierFile` reconstruction across three workloads

to 905µs (84% reduction). Similar improvements appear at the DFS client (Fig. 12b), with average latency decreasing from 419.5µs to 297.1µs. At DataNodes (Fig. 12c), average (P99) latency drops 64% (82%), indicating substantially lower disk traffic amplification. Comparing the two *ByteTier* configurations further highlights WAL entry alignment, which reduces average service client latency from 498.6µs to 423.0µs.

These improvements stem primarily from the reduced I/O amplification enabled by *ByteTier*'s two-phase write mechanism. By eliminating migration reads and merging small writes into LBA-aligned writes, *ByteTier* significantly reduces storage-side traffic amplification and tail latency.

5.2 System Elasticity and Availability

We evaluate *ByteTier*'s ability to support dynamic cluster operations in production environments. Specifically, we examine system elasticity under workload fluctuations and cluster scaling, and service availability during *TierFile* reconstruction.

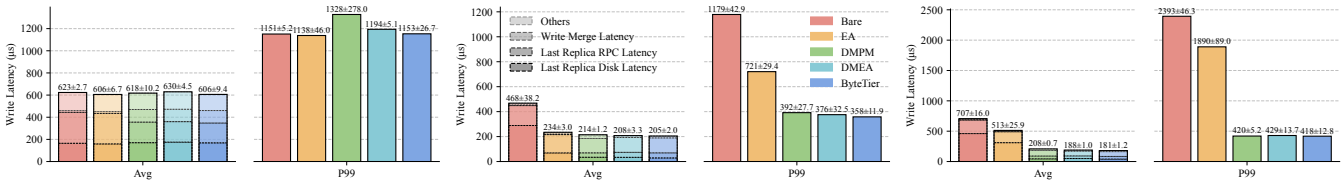
5.2.1 Elastic Scaling

To evaluate the benefits of *ByteTier*'s decoupled pool architecture, we simulate workload fluctuations and cluster scaling events. Fig. 13 shows the average write latency across five periods with different workload intensities and cluster sizes.

Under baseline workload, *ByteTier* achieves the lowest latency (300.62µs), significantly outperforming intra- and inter-node tiering (1474.49µs and 736.30µs, respectively). When the write workload increases by 50%, *ByteTier* remains stable (313.13µs), while both baselines experience noticeable degradation. After scaling out the cluster by adding 25% more SSDs (intra-node tiering requires synchronously expanding HDDs), latency decreases across all architectures. *ByteTier* maintains stable latency (311.30µs), while intra (inter)-node tiering increases to 1725.20µs (762.43µs). When workload pressure returns to normal and the cluster scales back in, *ByteTier* quickly returns to near-baseline latency (299.14µs). In contrast, both baselines exhibit higher post-scale-in latency: inter-node tiering is impacted by EC reconstruction reads, while intra-node tiering incurs parity-repair overhead on HDDs.

5.2.2 Service Availability

We evaluate the impact of `TierFile` reconstruction on service availability under three production workloads independently. For each workload, we conduct failure recovery and file movement experiments, injecting three reconstruction events with the same GB-scale amount of unapplied data in `DataCache`.



(a) Object Storage (b) OLAP (c) Block Storage

Figure 15: Average (P99) write latency under different merger configurations across three representative workloads

During failure recovery (Fig. 14a), the reconstructor scans WAL files to rebuild metadata and read data, causing a temporary throughput drop of about 16.5%. During file movement (Fig. 14b), metadata is restored from Xattr without scanning WAL files, reducing the throughput drop to about 8.9%.

Across all workloads, reconstruction completes within 400ms, measured from the start of the first *TierFile* recovery to the completion of the last in detailed logs. During reconstruction, we also observe that P99 read latency remains stable, while P99 write latency increases by about 20% due to temporary WAL reads. Since upper-layer services typically require seconds to handle failures or file movements, this sub-second reconstruction introduces limited impact.

5.3 Design Breakdown Evaluation

We further analyze performance gains of *Adaptive Foreground Write Merger*, *Migration-Free Background Applier*, and *Efficient TierFile Reconstructor* with component-level experiments using FIO [1]. For the merger and applier, write sizes follow simplified production-derived distributions: Object Storage (2KB/40%, 32KB/1%, 512KB/10%, 1MB/49%), OLAP (1KB/65%, 4KB/17%, 16KB/8%, 128KB/10%), and Block Storage (4KB/40%, 12KB/20%, 32KB/30%, 64KB/10%). For the reconstructor, FIO generates 100B–64KB random writes to emulate complex WAL traffic.

In these experiments, *ByteTier* uses EC(4,2) in the foreground and EC(8,3) in the background. Each experiment runs five times for 15–25 minutes per run. Metrics are collected every 30 seconds at the FIO clients, including mean latency, P99 latency, and throughput aggregated across all threads. We report averages across runs and include error bars for the merger and applier experiments.

5.3.1 Adaptive Foreground Write Merger

Merger experiments run on 12-DataNode clusters with production-matched SSD configurations: 2 SSDs for Object Storage, 1 for OLAP, and 16 for Block Storage. FIO generates ∼1.2GB/s write throughput per client using the corresponding size distribution. We evaluate five configurations that progressively enable merger features, as shown in Tab. 3.

We decompose average write latency into: write merge latency (WAL entry construction time, including write aggregation, metadata preparation, and memory copies); last replica RPC latency; and last replica disk latency. As shown in Fig. 15, the full *ByteTier* configuration consistently achieves the lowest average and P99 latency across all workloads. For OLAP and Block Storage (Fig. 15b&15c), compared with the bare configuration, *ByteTier* significantly reduces average latency by 56.2% and 74.4%, and P99 latency by 69.6% and 82.5%, respectively. Taking Block Storage as an example, the improvement mainly comes from reductions in network and disk latency. The last replica RPC latency drops from 223µs to 43µs, while the last replica disk latency decreases from 461µs to 41µs. Although write merging introduces additional overhead (about 80µs), aggregating multiple small writes into larger WAL entries significantly reduces the number of replication RPCs and disk writes. In contrast, Object Storage (Fig. 15a) shows limited improvement because it contains a large fraction of large writes (e.g., 1MB) that already amortize per-request RPC and persistence costs.

Table 3: Evaluated configurations of the merger

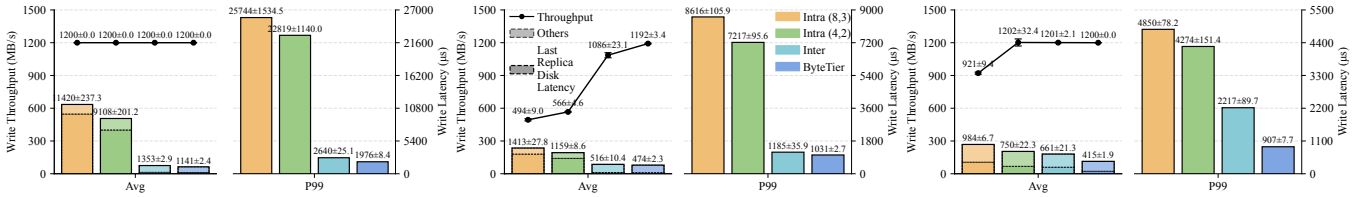
Merger Feature	Bare	EA	DMPM	DMEA	*ByteTier*
Dynamic Merge Formation	✗	✗	✓	✓	✓
Partial Merge Optimization	✗	✗	✓	✗	✓
WAL Entry Alignment	✗	✓	✗	✓	✓

5.3.2 Background Migration-Free Applier

Applier experiments use the same 12-DataNode clusters and SSD setups. As background migration in inter- and intra-node tiering requires extra SSD reads that may contend with foreground writes, some baselines cannot sustain full pressure. We therefore report both write latency (including last replica disk latency) and achieved write throughput.

We compare *ByteTier* with three tiering architectures: intra-node tiering (EC(8,3)), intra-node tiering (EC(4,2)), and inter-node tiering (EC(4,2)→EC(8,3)). Intra-node tiering baselines migrate data locally through background reads and writes, while the inter-node tiering performs migration across storage nodes via a write-read-write pipeline. *ByteTier* instead applies data directly to the capacity pool without migration reads.

Fig. 16 shows the average and P99 write latency under different tiering architectures. Across all workloads, *ByteTier*



(a) Object Storage (b) OLAP (c) Block Storage

Figure 16: Average (P99) write latency and throughput of background apply under different tiering architectures

consistently achieves the lowest latency while sustaining high throughput, demonstrating the benefit of migration-free apply. For Object Storage and OLAP (Fig. 16a&16b), intra-node tiering suffers from severe disk contention caused by background migration reads, while inter-node tiering alleviates local disk contention but still incurs additional reads. By directly applying data to the capacity pool, *ByteTier* reduces average latency by 8–10× and 2–3× compared with intra-node tiering for Object Storage and OLAP, respectively, and further improves latency by 8–16% over inter-node tiering. For Block Storage (Fig. 16c), disk contention is less significant since each DataNode is equipped with 16 SSDs. The latency difference mainly reflects the disk and network traffic amplification introduced by migration. *ByteTier* reduces average (P99) latency by 45–58% (79–81%) over intra-node tiering and 37% (59%) over inter-node tiering.

5.3.3 Efficient TierFile Reconstructor

We simulate a machine failure that loses a server's in-memory `DataCache`. The server manages 128 `TierFiles`, all of which must be reconstructed from linked WAL files in a new server before becoming serviceable again. We evaluate three configurations by progressively enabling reconstruction optimizations: base (WAL meta cache only), base + checkpoint, and base + checkpoint + Xattr. To evaluate scalability under different recovery data sizes, we reconstruct 1GB, 4GB, and 16GB of unapplied data. We report the average and P99 reconstruction times (`DataCache` index rebuild times) across the 128 affected `TierFiles`.

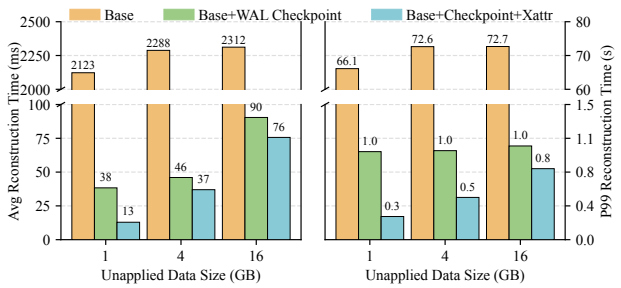


Figure 17: Average and P99 `TierFile` reconstruction time under progressively enabled reconstruction optimizations

As shown in Fig. 17, enabling WAL checkpoints reduces both the average and P99 reconstruction time by up to 98% compared with the base. Checkpoints record the starting position of unapplied data in each WAL file, allowing *ByteTier* to skip previously applied regions and scan only the last incomplete WAL block. Further enabling Xattr reduces the average (P99) reconstruction time by up to 66% (74%). By storing the precise WAL scan range in the Xattr of DFS files, *ByteTier* can directly locate the relevant WAL regions and avoid unnecessary scans. Reconstruction time grows sub-linearly with unapplied data volume because multiple `TierFiles` often share WAL files, allowing cached metadata to be reused.

6 Lessons Learned

We summarize key lessons from designing, deploying, and operating *ByteTier* in production.

Lessons on engineering complexity. Building *ByteTier* reveals the substantial engineering effort required for a production hybrid storage system. The implementation includes 60K+ lines of logic code and 90K+ lines of tests, integrating components across storage, networking, and system layers. It has evolved through 9 major releases, each requiring months of validation across diverse OS versions and hardware configurations in large-scale clusters under 24/7 workloads.

Lessons on hardware resource configuration. In production, we observe a strong coupling between EC configuration, disk I/O amplification, and memory usage. Larger EC units reduce HDD read QPS for workloads such as Object Storage and OLAP, but significantly increase `DataCache` consumption. In contrast, Block Storage favors smaller EC units with lower memory overhead. *ByteTier* uses the per-file batch control to trigger apply, effectively managing the total size of `DataCache` within a service node. In practice, the additional memory overhead remains below 10% of system memory (<0.1% total cost of ownership).

Lessons on downgrade mode. Despite optimizations, resource exhaustion may still occur in extreme scenarios (e.g., disk failures, memory pressure, or stalled background apply), leading to saturation of the performance pool or `DataCache`. To ensure service availability, *ByteTier* introduces a downgrade mode in which foreground writes bypass WAL and are directly persisted to DFS files. The system automatically exits

this mode once resources recover. In production, downgrade mode is extremely rare, with occurrence rates below 10^{-4} per cluster-day over the past three years. Observed incidents increased write latency by approximately 5% while service availability remained unaffected.

Lessons on the deployment topology. *ByteTier* is currently deployed as a client-side library, which simplifies integration with storage services but introduces challenges in system evolution and resource management. Updating the library requires restarting service processes, while tightly coupling storage logic with application servers complicates performance attribution and independent scaling across services. We have since introduced a shared proxy layer to decouple storage functionality from service processes and developed CLAPS [14] to pool proxy resources across storage clusters.

Lessons on software-hardware co-design. In practice, we try to exploit hardware capabilities. (i) Network. Under high load, background network traffic contributes up to 60% of tail latency, while hardware QoS (e.g., DSCP) reduces average and P99 network latency by 30% and 50%, respectively. (ii) SSDs. Garbage collection accounts for about 20% of tail latency; host-collaborative techniques such as FDP and ZNS separate foreground and background write lifetimes, reducing average write/read latency by over 22%/30% and P99 write/read latency by over 30%/54%. (iii) HDDs. Two-step apply reduces interference from background apply with foreground reads, but I/Os still receive fair scheduling once issued to HDDs. By leveraging on-disk I/O prioritization supported by modern HDDs, such as Command Duration Limits and NCQ Priority, we raise the priority of read requests and reduce average and tail read latency by 75% and 67%, respectively.

7 Related Work

We discuss related work on intra- and inter-node tiering, with a focus on heterogeneous-media organization, redundancy transition policies, and small-I/O aggregation.

Heterogeneous-media tiering. Tiered storage systems combine fast and capacity media to balance performance and cost. Prior work studies application- or engine-level placement across heterogeneous devices [24, 29], while distributed systems often colocate SSDs and HDDs within each node. BCW redirects writes from SSDs to HDDs to exploit HDD controller buffers [23]; AutoTiering and FlashTier manage placement or write-back within hybrid nodes [26, 33]; and CacheLib and Baleen optimize cache admission and prefetching [3, 32]. These designs improve locality, caching, or local write-back, whereas *ByteTier* exposes independent performance and capacity pools through a unified abstraction for write-path transformation.

Redundancy transition. Redundancy schemes determine the performance and cost trade-offs of distributed storage. Erasure coding provides a foundation for reducing storage overhead, with prior work studying coding schemes, deployment, and repair efficiency [2, 13, 25]. Production systems also change redundancy over time or across storage pools. Alibaba EBS and Pangu optimize block-storage performance and capacity efficiency through storage-pool evolution and narrow-to-wide EC transition [19, 35]. Morph manages file-lifetime redundancy by using replication for hot data and converting data to EC as it cools [16]. URSA combines SSD and HDD replicas for cloud-scale virtual disks [17]. Unlike these migration- or lifecycle-driven designs, *ByteTier* integrates redundancy transition into the write path, directly applying staged data to the capacity pool and avoiding migration reads and extra cross-pool traffic.

I/O aggregation. Existing techniques aggregate I/Os at the client, server, or backend. HDFS and CephFS use per-file client buffers, which offer limited merging opportunities under concurrent small-file workloads and may consume substantial memory [28, 31]. Server-side schedulers aggregate or reorder requests after network transmission [5, 20], while Bigtable-style systems serialize updates through WALs [9], limiting write parallelism. Alluxio packs small files offline to improve backend bandwidth utilization but incurs additional rewrite overhead [15]. Recent work also optimizes request-to-I/O transformation for high-bandwidth SSDs [34]. *ByteTier* integrates cross-file aggregation into tiering, reshaping foreground writes and directly batching staged data into capacity-pool files without offline rewriting or migration reads.

8 Conclusion

We present *ByteTier*, a cloud-scale pool-based tiering system for heterogeneous cloud storage. Built on decoupled performance and capacity pools, *ByteTier* enhances redundancy and scaling flexibility. Compared with intra- and inter-node tiering, *ByteTier*'s two-phase write mechanism reduces excessive disk and network traffic amplification. Diverse user writes are reshaped into device-friendly patterns for higher throughput and lower latency by *Adaptive Foreground Write Merger* and *Background Migration-Free Applier*. To ensure high availability, *Efficient TierFile Reconstructor* incorporates WAL file checkpoints, WAL block footers with a WAL meta cache, and Xattr to accelerate reconstruction. *ByteTier* has supported diverse storage services at ByteDance for over three years, improving performance and resource efficiency.

Acknowledgments

We would like to express our gratitude to all those who contributed to the design and development of *ByteTier*, including Pan Liu, Qiang Niu, Biyun Zhu, etc. This work was supported in part by the National Natural Science Foundation of China (62502430), Zhejiang Provincial Natural Science Foundation of China (LQ26F020004), and ByteDance Research Project (CT20250218122839).

References

[1] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio.

[2] S. B. Balaji, M. Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P. Vijay Kumar. Erasure coding for distributed storage: An overview. *Science China Information Sciences*, 61(10):100301, 2018.

[3] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 753–768, 2020.

[4] Runhua Bian, Liqiang Zhang, Jinxin Liu, Jiacheng Zhang, Jianong Zhong, Jiahao Gu, Hao Guo, Zhihong Guo, Yunhao Li, Fenghao Zhang, Jiangkun Zhao, Yangming Chen, Guojun Li, Ruwen Fan, Haijia Shen, Chengyu Dong, Yao Wang, Rui Shi, Jiwu Shu, and Youyou Lu. Discard-based garbage collection for distributed log-structured storage systems in ByteDance. In *USENIX Conference on File and Storage Technologies*, pages 493–508, 2026.

[5] Filip Blagojević, Cyril Guyot, Qingbo Wang, Timothy Tsai, Robert Mateescu, and Zvonimir Bandić. Priority IO scheduling in the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[6] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon S3. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 836–850, 2021.

[7] Praveen Borra. An overview of cloud computing and leading cloud service providers. *International Journal of Computer Engineering and Technology*, 15:122–133, 2024.

[8] 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. Windows Azure storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[10] Intel Corporation. Intel quickassist technology. https://github.com/intel/qatlib.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 20–43, 2003.

[12] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *USENIX Conference on File and Storage Technologies*, pages 233–248, 2021.

[13] Cheng Huang and Huseyin Simitci. Erasure coding in Windows Azure Storage. In *USENIX Annul Technical Conference*, pages 15–26, 2012.

[14] Xiuqi Huang, Shiwei Cao, Honghao Wang, Ruijie He, Huameng Zhou, Cheng Wang, Linyu Chen, Qingsong Zhu, Shunxin Min, Yunlong Wang, Xunzhi Xu, Yangming Chen, Chen Ling, Bingwen Yuan, Yushan Sun, Maoyin Liu, Chengyu Dong, Rui Shi, and Wei Chen. CLAPS: A load-aware proxy resource pooling system for reducing resource redundancy in large-scale cloud storage. In *International Conference on Management of Data*, pages 346–358, 2026.

[15] Saurabh Kadekodi, Bin Fan, Adit Madan, Garth A. Gibson, and Gregory R. Ganger. A case for packing and indexing in cloud file systems. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2018.

[16] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R. Ganger, and K. V. Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *ACM Symposium on Operating Systems Principles*, pages 330–346, New York, NY, USA, 2024.

[17] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. URSA: Hybrid block storage for cloud-scale virtual disks. In *European Conference on Computer Systems*, pages 1–17, 2019.

[18] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. *ACM Transactions on Storage*, 19(2):16:1–16:32, 2023.

[19] 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. More than capacity: Performance-oriented evolution of Pangu in Alibaba. In *USENIX Conference on File and Storage Technologies*, pages 331–346, 2023.

[20] Jianwei Liao, Dong Yin, and Xiaoning Peng. Block i/o scheduling on storage servers of distributed file systems. *Journal of Grid Computing*, pages 299–316, 2018.

[21] Peter Macko and Jason Hennessey. Survey of distributed file system design choices. *ACM Transactions on Storage*, 18(1):4:1–4:34, 2022.

[22] Lu Pang, Anis Alazzawe, Madhurima Ray, Krishna Kant, and Jeremy Swift. Adaptive intelligent tiering for modern storage systems. *Performance Evaluation*, 160:102332, 2023.

[23] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. Combining buffered I/O and direct I/O in distributed file systems. In *USENIX Conference on File and Storage Technologies*, pages 17–33, 2024.

[24] Yujie Ren, David Domingo, Jian Zhang, Paul John, Rekha Pitchumani, Sanidhya Kashyap, and Sudarsun Kannan. PolyStore: Exploiting combined capabilities of heterogeneous storage. In *USENIX Conference on File and Storage Technologies*, pages 539–555, 2025.

[25] Maheswaran Sathiamoorthy and Megasthenis Asteris. Xoring elephants: Novel erasure codes for big data. *VLDB Endowment*, 6(5):325–336, 2013.

[26] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *European Conference on Computer Systems*, pages 267–280, 2012.

[27] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[28] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[29] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 588–602, 2023.

[30] Chunxu Tang, Bin Fan, Jing Zhao, Chen Liang, Yi Wang, Beinan Wang, Ziyue Qiu, Lu Qiu, Bowen Ding, Shouzhuo Sun, Saiguang Che, Jiaming Mai, Shouwei Chen, Yu Zhu, Jianjian Xie, Yutian (James) Sun, Yao Li, Yangjun Zhang, Ke Wang, and Mingmin Chen. Data caching for Enterprise-Grade Petabyte-Scale OLAP. In *USENIX Annual Technical Conference*, pages 901–915, 2024.

[31] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Operating Systems Design and Implementation*, pages 307–320, 2006.

[32] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: ML admission & prefetching for flash caches. In *USENIX Conference on File and Storage Technologies*, pages 347–371, 2024.

[33] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: Automatic data placement manager in multi-tier all-flash datacenter. *CoRR*, abs/1805.01988, 2018.

[34] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Qiang Cao, Hong Jiang, Shaohua Wang, and Jie Yao. Rethinking the Request-to-IO transformation process of file systems for full utilization of High-Bandwidth SSDs. In *USENIX Conference on File and Storage Technologies*, pages 69–86, 2025.

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

[36] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes. In *USENIX Conference on File and Storage Technologies*, pages 187–202, 2021.