A Fast, Efficient, and Strongly-Consistent Object Store

Shuwen Sun
Northeastern University
Boston, Massachusetts, USA
sun.shuw@northeastern.edu

Isaac Khor
Northeastern University
Boston, Massachusetts, USA
khor.i@northeastern.edu

Ji-Yong Shin
Northeastern University
Boston, Massachusetts, USA
j.shin@northeastern.edu

Peter Desnoyers
Northeastern University
Boston, Massachusetts, USA
pjd@ccs.neu.edu

Abstract

S3-compatible object storage has become ubiquitous, used by an ever-expanding range of applications. Workload traces show that many of these applications treat object storage like a traditional file system, with many small reads and writes, yet object storage implementations have not kept up. Optimized for bulk storage, these systems cannot efficiently exploit modern SSDs, requiring large hardware installations to achieve operation rates typical of local file systems on modest machines.

ZStore is a highly-efficient object store designed for modern hardware, providing strong consistency (per-key linearizability) via a novel architecture which replicates data over independent per-device shared logs, using NVMe-over-Fabrics as its backend storage protocol. Based on a 3-node symmetric active-active cell, ZStore performs small reads and writes with minimal I/O amplification (beyond replication factor) while supporting object sizes up to the S3 maximum of 5 TB and optional erasure coding for objects larger than 128 KB. ZStore guarantees single-key linearizability using a two-phase coordination mechanism, tracking in-flight writes so that reads of stable data can be handled on a single gateway, with a heavier-weight multi-node read protocol used only when interfering writes are detected. Our evaluation shows ZStore achieving nearly an order of magnitude improvement in IOPS over widely-used systems (MinIO and Ceph) when evaluated on comparable hardware.

CCS Concepts

• **Information systems → Distributed storage**; • **Computer systems organization → Distributed architectures**.

1 Introduction

S3-like object storage has become the file system of the data center and cloud, used by a majority of surveyed cloud applications [56] to persist any data an application writer needs to store. Like files used by a single system, object store data comes in a wide variety of types and sizes; *e.g.*, IBM's 2020 measurements [34] show a median/mean/max object size of 11 KB, 2.6 MB, and 249 GB over a set of 98 week-long traces, with applications varying widely in object size distribution, operation rate, and read/write ratio.

While file systems have evolved over decades in the face of application demands to handle both high-throughput large file workloads and high-operation-rate small file ones, object storage systems have not for the most part. As an example, a recent benchmark [5] of the Ceph object store (Rados Gateway [63] or RGW) measured peak rates for writing and reading 4 KB objects of 178 K and 312 K IOPS respectively. These are roughly similar to results we see on a single-system small file create/read benchmark on Linux ext4, yielding 181 K and 210 K (uncached[1]) operations per second respectively. Yet while the ext4 benchmark achieved these results on a 6-core machine and a consumer-grade Non-Volatile Memory express [4] (NVMe) drive, the Ceph results required a 640-core cluster with 60 enterprise-class drives, at a capital cost potentially 100× or 200× higher than the single-machine filesystem.

Then, what are the requirements of a modern object store and why build one instead of focusing on a different storage abstraction? In theory an object store is equivalent to a simple key-value store, and again in theory is simple to implement over abstractions such as distributed shared logs [48]—both topics of extensive recent research[2]—yet in practice an object store is neither KV store nor shared log, with a set of unique requirements:

- **large object support:** up to 5 TB [2], with support for efficient storage (*e.g.*, erasure coding) of large objects; very small objects ($\ll$ 4 KB) must be supported, but sub-block packing is not needed due to the large mean object size.
- **extreme scalability:** unlike KV stores and shared logs, which are often deployed as per-tenant or per-application instances, object stores are typically datacenter-wide resources and must scale accordingly;
- **simple operations:** S3-compatible stores support PUT, GET (with optional byte range), DELETE, and *list object (name)* operations, with per-request authorization; more complex operations like range requests are not supported by the API.
- **simple consistency:** object stores available today offer list-after-write and per-key read-after-write consistency [10],

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '25, November 19–21, 2025, Online, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772272

[1] Cached reads were roughly 10× faster.
[2] Over a dozen KV store-related papers have been published in the last 5 years of FAST alone [29–32, 42, 44, 45, 50, 55, 58, 65–69, 71], while object storage is rarely mentioned in the literature [26].

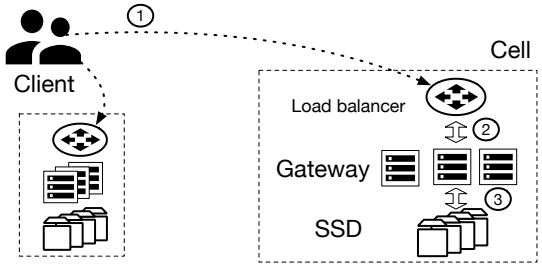


Figure 1: **ZStore architecture.** Requests are sharded across 3-gateway cells (①) and load-balanced within cells (②); gateways translate PUT and GET requests into NVMe operations to pool of remote devices (③).

rather than cross-key consistency or a total ordering of all operations.

To address these requirements, we present ZStore, an object store designed to provide local filesystem-like performance at datacenter scale. ZStore offers:

- single-key linearizability and list-after-write consistency,
- a novel architecture based on per-device (*i.e.*, independent, unreplicated) shared logs,
- high efficiency, translating small PUTs into $3 + \epsilon$ NVMe writes, and small GETs to 1 NVMe read, or 2 in its low-memory configuration,
- a highly scalable design based on sharding across 3-gateway cells with symmetric active-active redundancy,
- flexible storage layout: small objects are triple-replicated, while medium and large objects can be erasure coded or stored in non-flash pools.

The ZStore architecture is based on a 3-gateway load-balanced *cell*, seen in Figure 1. HTTP object requests are translated by gateways into read and append operations to backend storage devices, with semantics designed for NVMe-over-fabrics (NVMe-oF) [39] and the Zoned NameSpace (ZNS) Zone Append command[3] [3, 25]. An Remote Direct Memory Access (RDMA)-based protocol is used to track in-flight write operations, allowing zero-overhead reads of stable objects from any gateway while ensuring consistent access to keys which are being actively modified; in our analysis of published traces [34], fewer than one in 10^5 read operations require this additional processing. A standard coordination service, ZooKeeper [37], is used to schedule cell-wide operations such as batched metadata persistence, garbage collection, and failure recovery.

ZStore maintains a full in-memory index for all keys, which we argue is cost-effective for mean object sizes of 16 KB and up; we describe a variant requiring minimal RAM at the cost of an additional $1 + \epsilon$ NVMe reads per small GET operation.

Our prototype using Boost async I/O [17] and SPDK [7] achieves speeds of 190K small PUTs and 424K small GET operations per second on a single 16-core gateway with a backend of 6 WD DC ZN540 ZNS NVMe devices, and higher speeds with multiple gateways. Especially, with small objects, ZStore achieves an order of magnitude higher performance than MinIO and Ceph.

The contributions of this paper are:

- ZStore, an object store based on replication across a shared pool of unreplicated append-only shared logs,
- a mechanism to track in-flight writes, allowing reads of inactive data to be forwarded to backend storage without involvement of other gateways, while ensuring linearizability in contended cases, and
- a prototype implementation with a full real-world service (S3 object store) and detailed evaluation, demonstrating performance better or comparable to the Ceph benchmark described above when running on a single 16-core server with six backend devices.

In the remainder of this paper we describe S3 object storage, NVMe-oF, and ZNS in more detail (§ 2), present the architecture and design of ZStore (§ 3) and the implementation of our prototype (§ 4), evaluate its performance in isolation and in comparison with Ceph [61] and MinIO [19] (§ 5), and survey related work (§ 6).

2 Background and Motivation

Before describing ZStore, we provide (a) details of the S3 object storage interface, (b) analysis of a corpus of object storage traces, for better understanding of potential workloads, and (c) an overview of the NVMe technologies used by ZStore.

2.1 S3 Object Storage

Simple Storage Service or S3 [9] was introduced by Amazon Web Services in 2006 [1]; since then it has become a foundation of of cloud applications [56] and been adopted by other providers [11, 14, 15, 18] and open-source systems [19, 22, 63]. It is an HTTP-based storage API for unstructured named *objects* of up to 5 TB identified by URL (*bucket* + *key*), supporting single-object PUT, ranged GET, and DELETE operations, and a *list objects* command to enumerate keys, along with fine-grained access control based on per-request authentication.

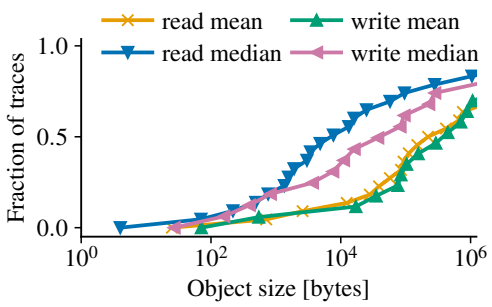
The original S3 was eventually consistent, with a risk of stale data from GET and *list objects* operations. Modern versions offer list-after write and read-after-write guarantees on a single key, beginning with AWS [10, 36]; this consistency level is now supported by most other providers (*e.g.*, Microsoft Azure [28]) and implementations (*e.g.*, Ceph [61] and MinIO [19]).

2.2 Workload Analysis

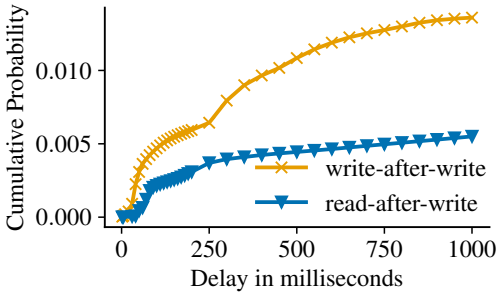
To better understand a typical object store workload, we examine the IBM Cloud Object Store corpus [34], encompassing 535M operations collected from 99 tenants over 7 days. As shown in Figure 2, the aggregate workload is read-dominated, however this varies widely by trace, with many traces being write-dominated. We highlight some additional characteristics relevant to our design.

Small operations: Median reads and writes are quite small, at 8.3 KB and 13.6 KB, with mean values hundreds of times larger, however Figure 2a shows that a third of traces have mean read and write sizes of 100 KB or less.

[3] Using either native ZNS devices or emulation over conventional devices.



(a) **Request size CDF.** CDF by trace of per-trace mean, median request sizes. Small operations are common, and dominate some traces



(b) **Truncated CDF of read-after-write, write-after-write intervals.** Interval from PUT to first GET or next PUT of the same key. Very few PUTs (< 1.4%) and GETs (< 0.6%) are within 1000ms of a previous PUT.

Figure 2: **IBM Cloud Object Store trace [34] (535M operations, 2099 TB transferred)**

Few deletes or overwrites: Roughly half of examined traces do not delete or overwrite more than a negligible fraction of objects written, while most other traces delete objects before writing the same key, rather than overwriting: of 117 M total writes in the trace, 67 M (57%) are later deleted, while only 3.8 M (3.2%) are overwritten. We note, however, that one or two traces exhibit very high overwrite ratios.

Little read-after-write and write-after-write conflict: Figure 2b shows truncated CDFs of read-after-write and write-after-write delays, ignoring all delays greater than 1 second. The fraction of reads following writes by less than 50ms is negligible; in fact only one trace shows a non-zero fraction (0.004%) of RW delays less than 10ms.

We take advantage of these characteristics in the ZStore architecture, using fast-path operations for "stable" objects which have not been recently modified, and a slow path to ensure consistency otherwise. By reducing this window to a small number of milliseconds or less, these slow-path operations have negligible effect on mean or even tail latencies for realistic workloads.

We note that this behavior is significantly different than what is observed in specialized shared-log applications. While read-after-write intervals in most IBM cloud object storage traces are in the range of hours, the common *log-following* pattern for shared log use, as described in LazyLog [48], results in typical read-after-write intervals of seconds or less.

2.3 NVMe-oF and ZNS

ZStore is designed for efficient operation using remote access to NVMe storage devices. While the original NVMe protocol was PCIe-based, the NVMe-oF [20] extensions allow the NVMe protocol to be used across both TCP (NVMe/TCP) and RDMA (NVMe/RDMA) transports, allowing access to network-remote devices with little overhead, yielding performance close to that of PCIe-connected ones [51]. While NVMe-oF is not directly supported by SSDs on the market today, they can be used in tandem with host-based NVMe-oF implementations such as SPDK (the Storage Performance Development Kit [7]), dedicated NVMe-oF enclosures [16], or even smart NIC implementations [13]. The ZStore prototype uses SPDK target mode emulation, which includes NVMe multipath features allowing shared access to a device from multiple remote hosts.

In ZStore, each device is treated as a shared append-only log, using the ZNS [21] NVMe command set. ZNS is designed to allow most flash translation layer (FTL) functionality to be moved to the host; the LBA space is divided into large *zones* which must be written sequentially, and then *reset* (*i.e.*, erased) before they may be rewritten. Since NVMe does not guarantee operation ordering across (or even within) submission queues, ZNS includes a *zone append* command [25] to avoid the need for synchronous writes.

When using zone append, the device itself is responsible for sequencing simultaneous writes. Operations are addressed to a zone, rather than a specific block address, and data is written to the current *write pointer* for that zone. The address at which data was written is returned in the response, and the write pointer incremented by the write size. When combined with NVMe-oF multipath, this allows multiple ZStore gateways to append to a single unreplicated shared log (*i.e.*, device) without need of an external sequencer.

3 Architecture and Design

ZStore is a distributed object store based on 3-gateway load-balanced active-active redundant cells which write directly to multiple independent logs on remote storage devices. PUT requests prepend metadata to generate log records which are triply replicated across independent logs using atomic append operations; all other writes to storage are batched and amortized across multiple PUTs. Small GETs translate to single NVMe read operations, ensuring efficient data retrieval.

Per-key write sequencing[4] is determined by primary-replica order, rather than using a shared sequencer as in CORFU [24] ; we describe mechanisms to ensure this order is preserved even in cases of primary replica failure. Linearizability of operations on a single key is ensured by RDMA replication of metadata across cell members via a two-phase communication protocol; at high operation rates this replication is batched for efficiency, with only modest

[4] ZStore does not guarantee a total ordering of operations on different keys.

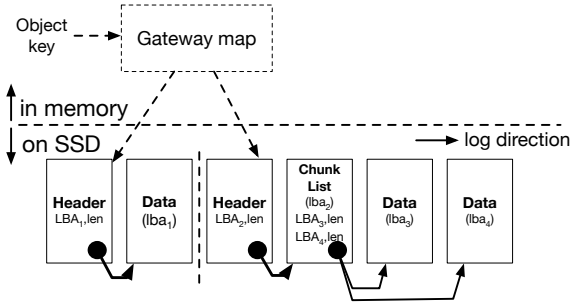


Figure 3: **Objects smaller than MDTS (128 KB) minus log header are written directly to the device as a data block following the header. Larger objects are written to the log in multiple chunks, with a final header entry to track them.**

increases in write latency. Index persistence, garbage collection, rebalancing and failure recovery are performed by electing individual cluster members using a standard coordination service [5].

3.1 High-level Architecture

The ZStore architecture is based on a 3-gateway *cell*, shown previously in Figure 1, sharing a storage pool of devices managed as independent, unreplicated append-only logs. Consistent hashing is used to choose a replica set when writing, ensuring that ordering of writes to a single key can be determined from the log. Log ordering is based on the block number at which a replica is written, which is preserved in the ZStore index in order to route read operations, thus two index entries for versions of the same object will inherently determine their ordering.

Once write ordering has been determined, the location (*i.e.*, device and block address) of each replica is stored both in memory and persistently. Since the consistent hash is not used for data retrieval, it may be readily updated to add or remove devices. Each device functions as an append-only log, with append operations returning the address at which data is written; in particular ZStore gateways connect to shared devices using NVMe-oF (TCP or RDMA), and append via ZNS zone append command.

Most ZStore operations use a cryptographic hash of the key (actually bucket+key), rather than the key itself; the short deterministic size of key hashes simplifies the process of distributing metadata when writing, and bounds memory consumption for the in-memory index. We maintain this in-memory index (referred to as the *ZStore map*) of all objects on each gateway in a cell, periodically persisting new entries to replicated storage. To satisfy bucket list requests we maintain a full *key list*, storing only recent entries in memory.

As seen in Figure 3, small objects are written in single append operations to the three targets with a log header specifying full bucket+key, length, and other metadata such as owner and permissions. The size of these append operations is limited by the NVMe *maximum data transfer size* or MDTS, typically 32 4 KiB blocks. Larger objects are broken into chunks which are written, optionally with erasure coding rather than triple replication, followed by a single inode-like triple-replicated *object descriptor*; the ordering of the write is determined by the log location of this descriptor.

Read and write requests are load-balanced uniformly across gateways in a cell; further scaling of ZStore is performed by sharding keys and buckets across multiple cells at the load balancer level. Storage devices may be managed in separate per-cell pools, or may be shared across multiple cells.

To describe the architecture in more detail, we describe (1) write and read handling, including the ZStore consistency protocol, (2) list object requests, (3) coordinated metadata persistence, (4) bucket list operations, (5) garbage collection, (6) failure recovery, and finally (7) potential design variations.

3.2 Write and read handling

Writes are replicated across three independent logs, with one distinguished as the *primary replica*. The devices themselves are used to sequence writes; the replica set is determined by a consistent hash on the key, guaranteeing that all writes to a single object go to the same replica set, and the order of writes on the primary replica (as in CORFU [24]) defines a total order on writes to that key. As is the case with other object stores, we do not guarantee ordering of or consistency for PUT and GET operations across multiple keys.

Individual writes are identified by (a) writer gateway and per-gateway write sequence number, (b) epoch number, and (c) a location triple, specifying the length (in 4 KB blocks) and the device ID and logical block address (LBA) of each replica of the object or object descriptor. DELETE writes a tombstone to the log, and is handled almost the same as a write.

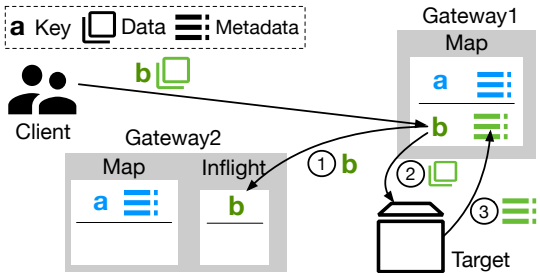
Given two index entries for writes to the same object in the current epoch, their order can be distinguished by their location tuples: a higher LBA on the primary replica (first in tuple) indicates a more recent write. The global epoch number for new writes is incremented each time map entries are persisted, and duplicate entries removed, thus (with one exception noted below) indexes with a more recent epoch number always take precedence over those from previous epochs.

The writer uses an RDMA-based two-phase protocol to broadcast write metadata to the other two cell gateways, ensuring single-key linearizability; the full sequence of steps in response to a client PUT is as follows, abbreviating phase 1 and 2 as $\phi 1$ and $\phi 2$, as seen in Figure 4:

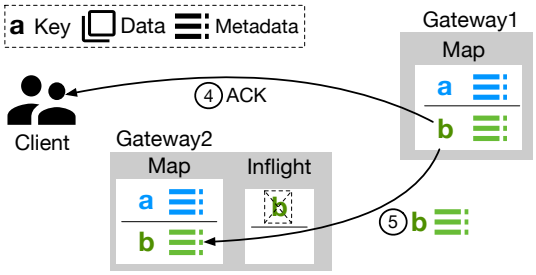
(1) $\phi 1$ notification: (gwID, seq#, key hash) (① in Figure 4a)
(2) dispatch append operations to 3 devices (② in Figure 4a)
(3) receive append completions (including LBAs) (③ in Figure 4a)
(4) acknowledge write to client (④ in Figure 4b)
(5) $\phi 2$ notification: (gwID, seq#, key hash, location triple) (⑤ in Figure 4b)

Steps (1) and (2) may be performed in any order, or in parallel, but the client acknowledgment must wait until (i) the other two gateways have received the $\phi 1$ notification, and (ii) device operations have completed (step (3)). Note that step (1) requires an *acknowledged* write, *i.e.*, the notification is guaranteed to be in memory on both remote gateways before client acknowledgment in step (4).

[5] Our prototype uses ZooKeeper [37]



(a) **PUT phase 1.** Writing gateway (Gateway 1) receives the PUT request from client, creates an entry in the gateway map, broadcasts object key to other gateways (①) and issues appends to devices (②). Once all appends complete, it updates the entry with location information (③).



(b) **PUT phase 2.** The writing gateway acknowledges the writes to the clients (④). Writing gateway then broadcasts the ongoing writes to other gateways, where the corresponding gateway will delete the current object key in the in-flight map, and create a map entry in its gateway map (⑤).

Figure 4: **ZStore two-phase coordination protocol**

Each gateway maintains an *in-flight map* holding $\phi 1$ notifications received from the other two gateways. On receiving a GET request, a gateway first checks the in-flight map; if no entry is found, we can use the current entry in the gateway map to dispatch a read operation and complete the request. If one or more entries for the key are found in the in-flight map, a request is sent to the gateway(s) the entry was received from; on receiving this request we (a) wait until $\phi 1$ notifications have been pushed to all other gateways, and (b) respond with the most recent map entry for that object. Finally, when a $\phi 2$ notification is received, the corresponding $\phi 1$ notification is removed from the in-flight map, and if newer (based on primary-replica position) than the current gateway map entry, the gateway map is updated.

Read-after-write consistency is ensured by notifying other gateways of a write ($\phi 1$ notification) before acknowledging to the client, ensuring that any read received at any gateway after write acknowledgment will see that write. In the period before the write is acknowledged to the client (step (4)) it is possible that gateway A has received the $\phi 1$ notification while gateway B has not; if a read to A is followed by a read to B, it may be possible to see reads "go backwards in time". Steps (a) and (b) above prevent this, ensuring *monotonic reads*: before returning version V of an object, we ensure that $\phi 1(V)$ has been delivered to all gateways.

RDMA-based queues are used for notifications; reads check the head of the two $\phi 1$ queues to determine if any notifications must be ingested before checking the in-flight map. Notifications are batched, with a short timeout, limiting the aggregate rate of RDMA operations to at most 100-200K per second; at this rate batching delay is lower than device write latency, and thus does not contribute to latency of PUT operations.

The "in-flight window" during which slow-path read operations are required is bounded by the latency of device writes and (batched) $\phi 1$ and $\phi 2$ notifications, *i.e.*, $\ll$ 1ms for reasonable hardware configurations. This latency-bounded window is much shorter than that of LazyLog [48], where the window duration is bounded by periodic activity requiring coordination of nodes. We note that there are fewer than 6000 sub-10ms read-after-write delays among the 418M reads in the IBM traces, although the fraction might be higher if the same applications were run against a very-low-latency object store.

3.3 List Objects handling

S3 [6] does not have a range request or "next" mechanism, but provides the *ListObjects* and *ListObjectsV2* APIs, which enumerate up to 1000 keys in lexicographic order, optionally starting after a specified value; this returns the key, size, checksum, owner and timestamp for each key. Object stores are expected to provide *list-after-write* consistency [36]—a ListObjects request received after a write completes will include information from that (or a newer) write. Application expectations for performance of this operation are rather low, with performance on AWS typically in the 10s of thousands of keys listed per second, allowing the use of simple mechanisms.

Each ZStore gateway in the cell maintains a sorted list of key names written in the current epoch, along with metadata described above and the ZStore per-gateway write sequence number. Gateways can request a sub-range or the full set of in-memory key names from each of the other two gateways. Keys written in prior epochs are persisted in storage, in an LSM tree-like series of sorted files. Fulfilling a *list objects* request requires requesting in-memory keys from other gateways, then merging them with the corresponding range from multiple level files in storage.

3.4 Metadata persistence

Like most log-structured systems, ZStore is able to recover fully from the data records in storage, but periodically aggregates its in-memory index and *checkpoints* it to persistent storage in order to bound this recovery process. This process is coordinated via Zookeeper [37], with each checkpoint and the period preceding it comprising a numbered *epoch*. We use a *fuzzy* checkpoint process with the following properties:

- Each write belongs to an epoch, and almost all index entries for epoch E are persisted in checkpoint E
- Stale indexes are tolerated, *e.g.*, a write at the beginning of epoch $E + 1$ may be superseded by a write at the end of epoch E, persisted in checkpoint E.
- No writes will be missed in log recovery.
- Failure of the checkpoint process can be recovered from without loss of data.

ZStore maintains write information (*i.e.*, key to location mappings) independently on each gateway, and periodically persists a searchable index checkpoint to flash. In this process we want to ensure that:

- after completion of a checkpoint, all writes to that point are either (a) recorded in the most recent or a previous checkpoint, or (b) resident in memory and guaranteed to be recorded in the next checkpoint;
- log recovery, under a failure scenario, will start at a point early enough to include all updates not included in the newest checkpoint, and preferably not long before that;
- checkpointing is safe against gateway or target failure

In normal operation a gateway maintains two in-memory maps from key to location: one map reflects all writes in epochs up through the previous one ($E-1$), and a smaller map holds writes in the current epoch E, being updated from local writes and $\phi 2$ notifications. Reads check the current epoch map first, then the full map. We note that the order in which writes are added to the map may not reflect the actual (*i.e.*, primary-replica) write order; we therefore compare any new write information with the existing index, and discard stale updates. In particular, a new entry is older than an existing one, and will be discarded, if either (1) its epoch number is less than the existing one by 2 or more, or (2) its primary replica location is earlier in the log.

The checkpoint process is as follows:

(1) Using the coordination service, one node elects itself as the *checkpoint leader*, and announces the transition to epoch $E+1$.
(2) The other gateways increment the current epoch to $E+1$ and ack; all new writes will be tagged $E+1$.
(3) The checkpoint leader pulls all entries for epoch E (and earlier, if any) from its current-epoch map, writes them to storage, and then merges them into the full map.
(4) The other gateways are notified that the checkpoint is complete; they remove entries with epoch $\leq E$ from the current-epoch map and merge them into the full map.

This creates a series of files (actually replicated block extents) holding checkpoint information, like an LSM tree but without ancillary lookup structures; we use Zookeeper to track the location of these files. Additional information stored in the checkpoint includes the key list, described above, and the *epoch start vector*: for each device, the lowest LBA holding data from this epoch, which may be calculated from the index entries persisted in the checkpoint.

We call the checkpoints "fuzzy" because the checkpoint leader may miss a few straggler $\phi 2$ notifications from one of the other two gateways when persisting its map. We assume the delay of these stragglers is much less than a checkpoint period; for recovery we create our full map from the most recent checkpoint, then begin log recovery from the epoch start vector, guaranteeing that no writes are missed.

If the checkpoint leader fails before completion, another gateway will time out and elect itself to replace it. This increments the current epoch to $E+2$, and we continue as described above; the failed gateway is not allowed to rejoin until the checkpoint is complete. To periodically compact the checkpoints, a gateway elects itself *compaction leader*, merges several older checkpoint files in LSM fashion into a new file, and updates the checkpoint file list in Zookeeper. Checkpointing and compaction are exclusive; an attempt to perform one will wait until the other has completed.

3.5 Garbage collection.

In the ZNS version of ZStore, data is appended to *zones* of roughly 1 GB on each device (assuming a large-zone ZNS); as storage fills, zones holding out-dated data (*i.e.*, *garbage*) must be cleaned and erased before being re-used. Again, we use Zookeeper to elect a GC leader, which performs one or more garbage collection cycles before stepping down. A zone is selected for cleaning using the Greedy criteria (*i.e.*, with least live data), and remaining live data is copied to other devices. This will update one entry in the object's location triple; a $\phi 2$ notification for this new location set is sent to all gateways, and applied to the current-epoch map.

Handling of new writes while garbage collecting is a challenge in many systems, but is readily handled by ZStore's map update rules. Moving data for a specific object version changes its location tuple, but not its epoch, so as long as the zone being cleaned contains only data from epochs $E-2$ and earlier, index updates due to garbage collection will be properly superseded by new writes in the current epoch. When all data has been copied out of a zone and all resulting $\phi 2$ notifications have been posted, it is safe to reset (*i.e.*, erase) the zone and add it to a free zone list in Zookeeper.

Garbage collection can proceed during checkpointing or compaction, as the $\phi 2$ notifications it generates will be checked against existing map entries. GC can ignore the key-to-replica set hash and choose arbitrary targets for data, as the object version's write order is already specified by epoch number.

3.6 Failure Recovery

ZStore is able to recover from target failure, gateway failure, and combined gateway+target failure.

Target failure: If a target fails, the gateways have full knowledge of what data was stored on the device. When a gateway determines (*e.g.*, through command timeout) that a target is dead, it uses Zookeeper to advertise this and to elect itself *failure recovery leader* for that target. It first creates a new consistent hash map without the failed device, and broadcasts that to the other gateways as part of a checkpoint and epoch change.

It then proceeds to recover all data stored on that device by (1) reading from another replica (or equivalent recovery of larger erasure-coded objects), (2) writing a copy somewhere else, and (3) broadcasting a $\phi 2$ notification with an updated location triple. As with garbage collection, locations may be chosen without regard to the consistent hash, and the rules for updating the current-epoch map when receiving $\phi 2$ notifications ensure that new writes to an object will take precedence over a recovered version.

Gateway failure: When handling a write, a gateway (1) sends $\phi 1$ notifications to other gateways, (2) writes to the backend, (3) acknowledges to the client, and (4) sends $\phi 2$ notifications to other gateways. If the gateway fails before step (3), the write did not complete and can be abandoned. If it fails after step 4, the other gateways have a record for that write which will be persisted in the

next checkpoint, but writes which were in an intermediate stage may have been acknowledged, and must be recovered from the log.

To do this, one gateway uses Zookeeper to declare the failed gateway as dead and elect itself as recovery leader, and traverses the log on each device from the starting point identified in the most recent checkpoint. The recovery leader sends $\phi 2$ notifications for each log entry it finds which was written by the failed gateway, which if current will be added to each gateway map and persisted in the next checkpoint. Once recovery is complete, a checkpoint is performed before allowing the dead gateway to rejoin the cell.

Target+Gateway failure: The final case is simultaneous (*i.e.*, overlapping) target and gateway failure. In this case the primary replica may not be available, preventing its use in determining ordering for writes in progress. This is only an issue when writes closely follow each other; we detect this case via the in-flight map of $\phi 1$ notifications, and perform additional steps to ensure that write ordering is persisted across multiple replicas.

When processing a PUT request, we consider it a *non-interfering* write if its key is not found in the in-flight map. This is flagged in the object header, and writing proceeds as described in § 3.2. During recovery we search for secondary replicas of "lost" writes, and "make up" location triples by using real non-primary-key locations and an appropriately ordered but fake location on the failed primary.

If the object key is present in the in-flight map at write time, ZStore must ensure that primary-key order is replicated across devices before acknowledging the write. We flag this in the object header, and after data is written and location information is available, the location tuple is *piggybacked* on another write before responding to the client. During recovery we ignore log entries flagged as interfering, and instead recover the piggybacked location tuples.

3.7 Discussion and Variations

Memory: The ZStore architecture described above relies on a full in-memory map from key hash to location, requiring 64 B per key, and is replicated across 3 gateways in a cell. For a mean object size of 16 KB this would require 1 byte of DRAM for every 256 bytes of flash; assuming a 40:1 DRAM:flash cost ratio, the index would add an additional 16% to the cost of deployed storage. This is significant but not impractical; for object sizes seen in the IBM traces (mean write size 1.8 MB) the index DRAM would cost less than 0.2% as much as SSDs.

Alternately, memory consumption can be drastically reduced by managing checkpoints as an LSM tree, with map entries retrieved from flash storage as necessary and cached in memory. In the worst case this would require an additional $1 + \epsilon$ NVMe-oF reads per GET request; analysis of the merged IBM traces (33K peak IOPS, 342M objects) shows a hit rate of 82% for a 10M-entry cache consuming roughly 1GB of memory. Assuming a 30× greater peak rate for a 3-gateway cluster, and a corresponding increase in cache size, the cost of DRAM for this cache would be negligible.

Weaker consistency: ZStore can be modified to support *read-after-write consistency*, rather than linearizability, by removing $\phi 1$ notifications and the in-flight map, and ensuring $\phi 2$ notifications are received by other gateways before acknowledging a write. This would cut notification overhead in half, eliminate the need for slow-path handling of any GET requests, and simplify failure recovery as all acknowledged writes are replicated across the memory of three gateways.

Consistent hash: ZStore uses a table-based hash, similar to that of Ceph [62] or Flat Datacenter Storage [53]: a hash of the key is used to index a table of replica sets constructed to ensure redundancy and uniform distribution across devices. Since data is not migrated when the table changes, this need not be a true consistent hash [41], and secondary replicas for a key may be changed between table iterations; however changes to the primary replica for a key require coordination (*e.g.*, via ZooKeeper) to preserve write ordering.

S3 authentication: S3 requests include an authentication header, which includes an *access key*, equivalent to an username, and a signature of header fields and a *secret key* using HMAC-SHA1. The number of bytes hashed is not large, consisting of the concatenation of the URL and several HTTP header fields, and in typical applications the access/secret keys in active use may be readily held in memory. The resulting overhead is minor, only modestly more than that required to compute a SHA1 hash of the URL for indexing.

ZStore storage protocol: The ZStore storage protocol provides two properties needed by the architecture: (a) append operations with device-local sequencing, returning location data (*i.e.*, LBA) which may also be used to establish ordering, and (b) log accessibility (including ordering and location) from other gateways. It is currently implemented as a thin shim (an ordered list of zones) on top of ZNS and NVMe-oF, but alternate implementations are possible.

Ongoing work retains NVMe-oF, due to the wide availability of highly-tuned and efficient implementations, but as we transition to SPDK's ZNS emulation over conventional[6] SSDs to address availabiity issues, we are exploring non-standard modifications to the emulation logic, such as very large, sparse zones to improve garbage collection performance.

3.8 Scaling and Load Balancing

A ZStore cell cannot be scaled to large numbers of gateways, as although the consistency communication within a cell may be efficient, its asymptotic cost is $O(N^2)$. We scale instead by sharding across ZStore cells, assigning a fraction of the bucket+key space to each cell at the load balancer.

Simple sharding based on *e.g.*, bucket name or key range would lead to load imbalances across cells. To remedy this, physical SSDs could be shared across most or all cells, by maintaining multiple logs (one per cell) on each device. This in turn would incur multiple (per-cell) failure recovery operations upon failure of a single device, impacting scalability. Instead ZStore shards based on a strong hash—the SHA1 used by the index—to distribute data equally across cells, and associates each storage target with one and only one cell.

To allow dynamic expansion of a deployed system we use multiple ZStore gateway instances per physical gateway, migrating instances to new hardware as it is deployed. Migration of an instance is straightforward and does not involve transfer of state—the instance de-registers from the load balancer and drops out of the

[6] If available, Flexible Data Placement [8, 33] devices could be used for this purpose, as well.

cluster for a full epoch, then advertises itself (*e.g.*, address, RDMA targets) to its cluster neighbors from its new host, and finally subscribes to its key range on the load balancer.

4 Implementation

The ZStore prototype is implemented in ≈7000 lines of C++ code, using the Boost [12] async I/O framework and Beast [17] HTTP library, and SPDK [7] for NVMe-oF initiator support, with TCP sockets for gateway-to-gateway communication; we are examining the Seastar async framework [57] and eRPC [40] for higher performance, but will need to upgrade our networking hardware to see performance improvements for all but the smallest object sizes. The storage backend is comprised of native ZNS devices (Western Digital DC ZN540), using SPDK target mode to expose them as NVMe-oF targets. The prototype uses ZooKeeper [37] to coordinate map persistence, garbage collection, and failure recovery.

ZStore constructs a continuous shared log on zoned storage devices: writes are sent as *zone append* commands, which either succeed, returning the LBA at which data was written, or return a status indicating that the end of the zone has been reached. In the latter case we send a *finish-zone* command to the device, which seals the zone against further writes and releases any resources held for writing that zone. An ordered free list of zones on a device is maintained in ZooKeeper and known by all gateways, so each can "wrap around" to the next zone without coordination. Finally, the order in which zones are written must be known in order to determine the relative log position of two objects or replicas, although it is only necessary to keep enough information to order LBAs used within the last 2 epochs.

The indexes (last 2 epochs, previous writes, and in-flight), $\phi 1$ and $\phi 2$ notifications are based on a standard 64-byte structure with the following fields:

- SHA1 key hash
- ID of gateway performing this write
- per-gateway write sequence number
- epoch
- length in blocks
- (target ID, LBA) ×3 — location of replicas

The on-disk format includes a header with the information above, as well as exact data length, full key name, object metadata such as owner, permissions, and timestamps, as well as additional failure recovery information such as the full replica set and the conflicted-write flag and optional piggybacked location tuple described in Section 3.6. For large objects the header indicates the tree height, and the body holds a block extent list identifying either data blocks or next-level descriptors.

Our prototype supports triple-replicated small and large objects, range requests, and ListObjectsV2, as well as full metadata persistence, garbage collection, and failure recovery. It does not yet support erasure coding, multipart upload, or S3 authentication, which are in progress but not completed. Due to limitations in our experimental hardware we have not yet tested sharding by key across multiple ZStore cells. Source code is available at https://github.com/shuwens/zstore.

Table 1: **Hardware specifications.**

Hardware	Spec	Speed
CPU	AMD Ryzen 9 9950X with 16 cores AMD Ryzen 5 7600 with 6 cores	
RAM	64 GB	
ZNS SSD	Western Digital ZN540 4 TB	3.2R/2.0W GB/s 450R/180W k IOPS
NVMe SSD	Sk Hynix P41 500 GB	7.0R/6.5W GB/s 960R/1,000W k IOPS
Network	ConnectX Pro 3	40 Gbps

5 Evaluation

We next evaluate the ZStore prototype to answer the following questions:

How fast is it?: What performance does it achieve on read and write micro-benchmarks, with varying object sizes, in single-gateway and multi-gateway configurations?

What is the overhead of linearizability?: We create artificial workloads to trigger slow-path processing for contended keys, and compare performance with and without the slow-path processing needed for linearizability.

What is the overhead of background operations?: We measure benchmark slowdown during checkpoint persistence, failure recovery, and garbage collection.

How does it perform compared to widely-used object stores?: We compare small-object and large-object performance against Ceph RADOS Gateway and MinIO.

5.1 Experimental setup

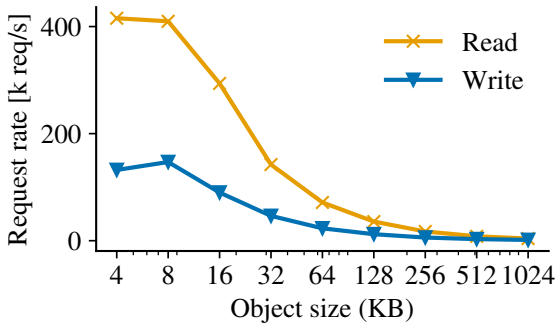
The hardware configuration for our experiments is shown in Table 1. Three 16-core AMD Ryzen9 servers are used as ZStore gateways, with 40 Gbit/s Ethernet and RoCE v2 RDMA support. Three 6-core Ryzen 5 workstations serve as targets, each equipped with two 4 TB Western Digital ZN540[7] SSDs, with SPDK target-mode emulation used to expose them as multipath-capable NVMe-oF targets.

Another 6-core Ryzen 5 workstation is used for client emulation. Requests are generated using S3Bench [52], a widely-used benchmarking tool for object storage, and `wrk` [35], a configurable general-purpose HTTP benchmarking tool.

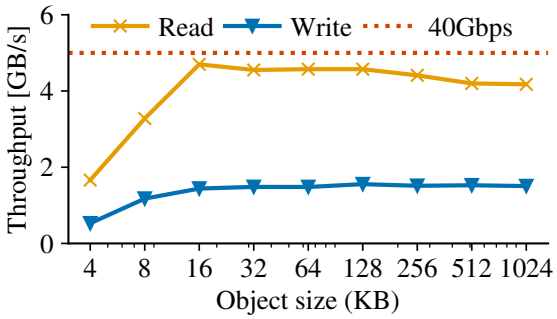
5.2 Microbenchmarks

We run a series of tests using wrk as our workload generator, generating random GET requests across a pre-populated pool of objects, and/or PUT requests to randomly-chosen names. The number of outstanding requests generated by wrk is limited by the number of open *connections* it creates, which serves a role similar to that of *queue depth* in other storage benchmarks. Wrk was configured with 12 threads (2 per CPU core) and 120 connections, although smaller connection counts were sufficient to achieve full throughput with larger object sizes.

[7]Limited availability of these drives has hampered our experiments; we are currently expanding our testbed by switching to SPDK ZNS emulation on conventional SSDs, allowing us to substantially enlarge our experimental setup.



(a) **Single-gateway read and write IOPS.** 4KB read/write performance is bottlenecked by the HTTP framework. Write shows lower throughput due to triple replication.



(b) **Single-gateway read and write GB/s .** ZStore quickly saturates the network and the maximum throughput is bounded by the 40 Gb/s network. Write shows lower throughput due to triple replication.

Figure 5: **Single-gateway microbenchmarks (4 KB to 1 MB).** Results of object larger than 1 MB is omitted.

Figure 5a shows the single-gateway performance for object sizes ranging from 4 KB to 1 MB[8]. For 4 KB reads, the peak request rate of 420 K requests/sec is quite close to the raw performance of the HTTP framework, as measured with a custom application serving a 4 KB buffer of zeros in response to every GET request. We are investigating further tuning of this framework, as well as an alternate HTTP framework [57] which may be able to provide higher performance.

At request sizes of 16 KB and larger, performance is limited by the 40 Gb/s network, as seen in Figure 5b. We omit results for our experiments on a complete cell of 3 gateways, as network bandwidth limits to the single client machine resulted in performance very close to that of the single-gateway case, although in this case saturating the network with 8 KB requests rather than 16 KB.

Write request rate is CPU-limited: although the three replicated writes are launched in parallel, resulting in write latencies (4 KB: 110 μs) comparable to that of read operations (4 KB: 115μs), the CPU time spent on NVMe operations is tripled, yielding request rates of 190k RPS for 4 KB writes, or roughly 1/3 those seen for read operations. In the single-gateway configuration, throughput is limited to 1/3 that of read operations, as well, as each replica must be written through the same 40 Gb/s network interface.

5.3 Linearizability overhead

To measure the cost of strong consistency, we compare baseline ZStore, with full *single-key linearizability*, with the *read-after-write-consistent* variant described in § 3.7, which does not detect read-after-write conflicts and has no slow path for reads. We measure "flat-out" performance on a real-world trace—IBM Trace 042, with 400K PUTs and 200K GETs over a range of object sizes—while adding variable fractions of closely-spaced write/read operation pairs, separated by 5 operations (in most cases 100μs to 1ms), in order to trigger detection of in-flight writes and resulting slow-path read processing.

[8]Using ZStore large object support for objects of 128 KB and larger.

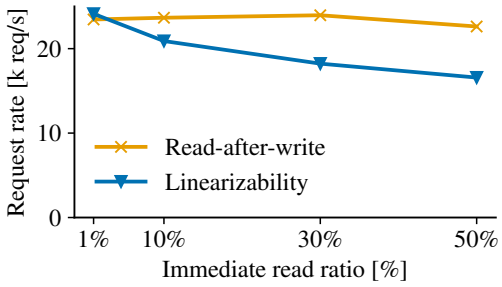


Figure 6: **Linearizability overhead, a comparison between read-after-write and linearizability modes of ZStore.** Real-world trace is modified to include varying fraction of < 1ms read-after-write. Extremely high rates of read-after-write impacts the linearizability performance modestly.

Results may be seen in Figure 6. Throughput is unaffected at realistic fractions of short read-after-write intervals, *i.e.*, several percent or less. Extremely high rates of such requests result in a relatively modest reduction (roughly 20%) in performance.

5.4 Overhead of background operations

ZStore achieves its efficiency in part by amortizing the cost of bulk operations over many requests. To quantify this amortized overhead, we measure performance degradation under high load from checkpointing, recovery of a failed target, and garbage collection.

5.4.1 Checkpointing: Map size vs. Overhead Checkpointing writes a series of replicated extents containing all index entries generated during the last checkpoint period; the volume of data written is thus determined by the request rate and checkpoint period. For a worst-case figure we assume a request rate of 200K requests/sec, corresponding to an average object size of 10s of KB; checkpointing

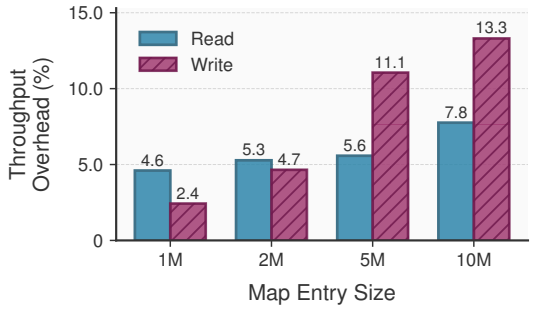


Figure 7: **Throughput decrease while checkpointing 1 M-10 M index entries.** Small checkpoints have <5% impact; worst-case 10 M entries decrease writes by 13%.

every minute this would write 10M map entries. At a more realistic average object size (*e.g.*, 1.8 MB across the IBM Cloud Object Storage traces [34]), and a checkpoint interval of 10 seconds, each checkpoint would be 10× to 100× smaller.

To measure overhead we simulate persisting a map of 1M to 10M entries while running read and write benchmarks; results are seen in Figure 7. Overhead is modest—persisting a 1M-entry map results in less than a 5% decrease in performance in either case. As map size increases the overhead becomes worse, reaching 13% for write operations at 10M entries—a map size corresponding to a worst-case PUT-only workload (200K req/s × 3 gateways) on current hardware and a checkpoint interval of 16.7 seconds.

5.4.2 Failure recovery target and gateway recovery

Failure recovery is another source of overhead. We note that the "blast radius" of failure recovery is relatively small in ZStore, being limited to a single cell sharing a pool of devices; although failure may be common in a very large deployment, a specific cell or gateway will be affected by it only infrequently. To assess the impact of target and failure failure recovery we measure the worst-case impact on benchmark execution during recovery.

Target failure: To recover from target failure, ZStore identifies all objects which lost a replica due to the failure; for each of these it reads one of the remaining replicas and writes it to another device. To test this we load our system with varying numbers of 4 KB objects, then fail one of the 6 target devices and measure the impact on read and write benchmarks during the recovery process.

In Figure 8a we see that read and write benchmarks are impacted significantly, up to 20% and 35% respectively. We note that these are worst-case measurements, as (a) they are measured on the leader gateway, while disruption at non-leader gateways should be much less, and (b) the interference is (mostly) proportional to the number of NVMe reads and writes, not their total size, and will be significantly less at more realistic mean object sizes.

Nevertheless we note that target failure recovery can result in significant performance degradation; we are implementing a recovery pacing mechanism in ZStore to bound this impact.

Gateway failure: Recovery from gateway failure requires "walking the log" from the beginning of the previous checkpoint period, reading all object headers written from the start of that period until the present. As with checkpointing, the work required depends on the number of write operations during the period.

In Figure 8b we see the performance impact of walking a log of 1M through 10M objects; as in the checkpointing case we expect that object size and checkpoint frequency will result in scanning 1M or fewer objects during this process. Read and write benchmarks (again measured on the leader node) are impacted modestly, slowing from 5 to 15%. We note that this overhead is unlikely to be noticed, as it represents 2% to 5% of aggregate cell performance, vs. the 33% of performance lost when a gateway fails.

5.4.3 Garbage collection The ZStore prototype uses large-zone ZNS drives, where each zone of roughly 1 GB must be *reset* (*i.e.*, erased) before being re-used. This necessitates a garbage collection process, which identifies candidate zones with lower-than-average fractions of live data, and then copies that remaining live data somewhere else. In a long-running system the rate of garbage collection is determined by the rate at which data is written, as the cleaning process needs to produce clean zones quickly enough to accommodate arriving data.

Rather than running the system until it reaches this steady state, we simulate it by configuring storage with a "pre-aged" map containing garbage, so that zones selected for cleaning contain 50% live data in 4 KB objects. We adjust the cleaning algorithm to copy data out of a zone in 2, 5, and 10 seconds, and again measure the impact on read and write benchmarks on the GC leader; results may be seen in Figure 9. The performance impact of a 10 second GC interval is modest—10% write and 5% read—rising to 25% and 17% for a 2 second interval.

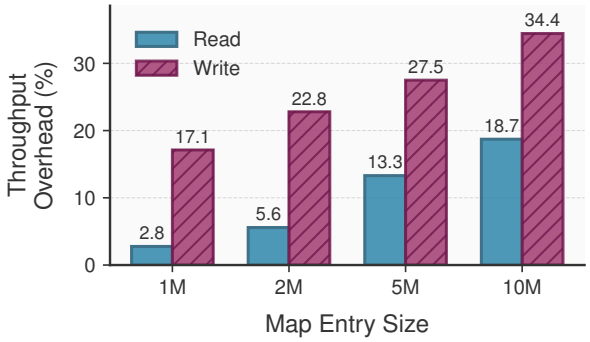
Finally, we note that the impact of garbage collection may be significantly lower than this in environments similar to that of the IBM Cloud Object Storage traces, due to lack of garbage to collect. In many of the traces the objects created were (almost) never deleted, leading to an ever-expanding storage footprint. In this case there is no space for garbage collection to reclaim, and the only way to allow continued writing as the system fills up is to deploy additional storage capacity.

5.5 ZStore vs Ceph and MinIO

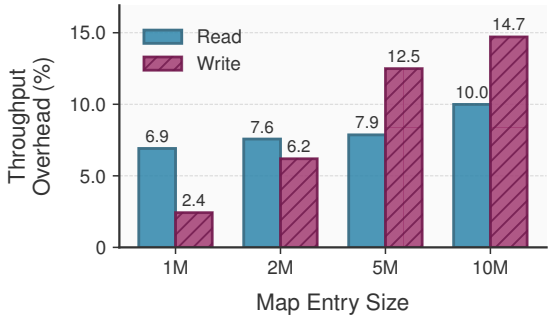
The final part of our evaluation compares ZStore to Ceph RGW [61] and MinIO [19], two widely-used open-source object stores. We run these systems on the same cluster as ZStore, replacing the ZNS SSDs used in the experiments above with the same number of 500 GB SK Hynix P41 NVMe drives.

Figure 10a illustrates benchmark results for 4 KB objects on the three systems, using the widely-used S3Bench [52] benchmark. Results for Ceph and MinIO are roughly comparable, while ZStore read performance is 2× higher, and write performance 3×. Yet this does not tell the whole story, as S3Bench is optimized for high-throughput large object accesses, achieving only 37 K 4 KB reads per second with ZStore.

In Figure 10b we see more detailed results using the much higher-performance wrk tester, plotting throughput in GB rather than operations per second. ZStore saturates the 40 Gbps network with 16 KB requests, while neither Ceph nor MinIO reach equivalent throughput until the object size increases to 1 MB. This experiments



(a) **Performance impact during target failure recovery.** Overhead scales with objects recovered (1M-10M), reaching 35% for writes in worst case.



(b) **Performance impact during gateway failure recovery.** Log scanning overhead remains modest (5-15%) even for 10M entries.

Figure 8: **Failure recovery overhead of ZStore.**

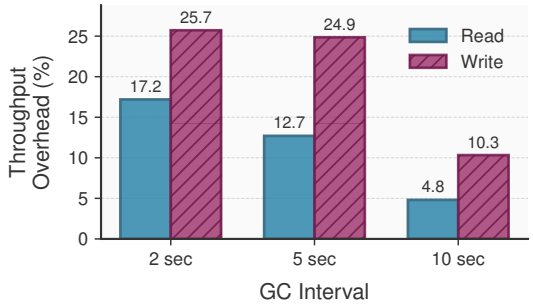


Figure 9: **Garbage collection overhead with different aggressiveness (2-10 sec interval).** Fast GC (2 sec) reduces throughput by up to 25%; slow GC (10 sec) has <10% impact.

was performed with both read and write operations; results were similar for both, and only one is shown in the graph for readability.

Discussion: We believe that much of the performance advantage of ZStore over Ceph RGW is architectural, rather than an artifact of implementation.

To create a small S3 object, RGW must create at least two *RADOS objects*[9] on the backend — a small inode-like object (the *head object*) and a *data object*. Chain replication within RADOS results in serialization of the replica writes for each of the two objects; RGW then serializes the creation of the two as part of its atomicity mechanism. Additional writes are needed at each RADOS object replica in order to persist block allocation metadata for the RADOS objects; we have not determined whether these may be performed in parallel with the data writes. While RGW must perform at minimum 12 writes, ZStore performs 3, for a 4× decrease in device IOPS load, and serialization of these writes (vs. parallel writes in ZStore) yields a 6 – 12× increase in latency. The messaging for these operations results in a much larger number of code paths across multiple nodes; even if Ceph were to use an equivalent or better framework than ZStore, it would likely incur higher software overhead.

We note that both Ceph RGW and MinIO have been carefully tuned for large-object performance, with throughput comparable to that of ZStore. We have not performed a full analysis of MinIO small-object performance, due to the complexities of its relationship with the Linux file system, but expect similar I/O expansion and serialization across layers.

5.6 Additional functionality

In addition to basic object functionality — PUT, GET,and DELETE — ZStore implements the *ListObjectsV2* [6] API, which returns an ordered list of up to 1000 keys in a bucket, with starting point and continuation mechanisms to allow its use in larger buckets. Application expectations for performance of this API are low: published reports indicate a listing rate of 13.5 K names/sec (13.5 requests/sec) for AWS S3 using the official AWS API, and 93.8 K names/sec for a highly-tuned client library.

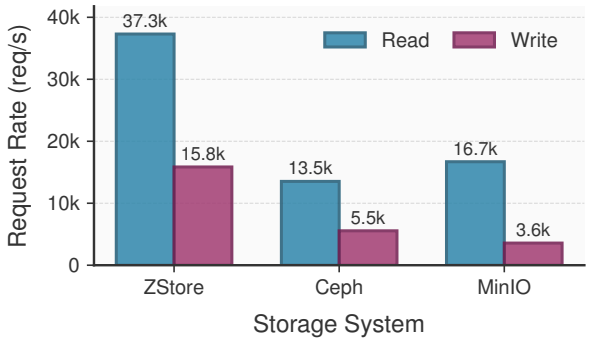
The ZStore prototype can perform 2400 1000-key requests/sec, listing 2.4 M names/sec. We have not yet implemented ListObjectsv2 for the sharded multi-cell architecture; we expect a significant drop in performance (perhaps as much as 2-3×) due to the need to merge request streams from multiple cells, but expect performance to remain well above application expectations.

6 Related Work

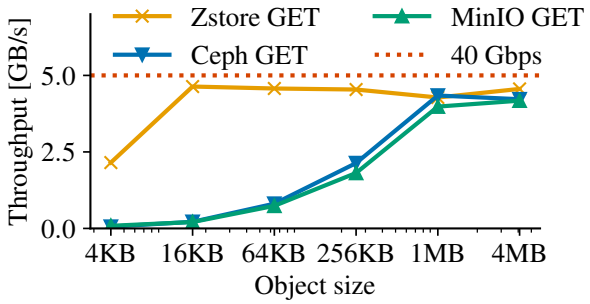
In this section we contrast ZStore with prior work which overlaps in features (shared logs and shared register protocols) or in underlying technologies (ZNS, RDMA, NVMe-oF).

Shared Logs: The CORFU replicated shared log [24] uses an "external" sequencer to allocate increasing storage addresses in a shared volume, and write-once storage semantics on each replica in order to resolve conflicts. In contrast ZStore relies on storage devices to directly sequence individual append operations, using the ZNS Zone Append operation in its current implementation. CORFU relies on this shared sequencer for performance, falling back on a "thundering herd" approach when it fails, while ZStore's expected-case logic

[9]These are replicated, mutable, named file-like objects used by all Ceph services including CephFS and RGW [61].



(a) **S3Bench with 4 KB objects.** ZStore achieves 37K/15K read/write ops vs Ceph's 13 K/5 K and MinIO's 17 K/4 K on identical hardware



(b) **Read workload with varying object sizes.** ZStore saturates 40 Gbps at 16 KB objects while Ceph and MinIO require 1 MB objects, demonstrating ZStore's small-object efficiency.

Figure 10: **Performance of ZStore, Ceph, and MinIO.**

requires no per-request coordination outside of individual devices. Finally, while CORFU provides total ordering of all requests, this guarantee is not required of object stores.

The authors of FuzzyLog [46] realized that achieving global ordering is expensive, and proposed a system that allows for partial ordering of writes using colors. LazyLog [48] further improves performance by observing that writes are not immediately read back, thus total ordering can be delayed until read time, and achieves linearizability by deferring reads until all writes are persistent. ZStore shares the similar key insight, but tracks ongoing writes via its novel two-phase protocol, deferring reads only until the write has finished, while in LazyLog reads are deferred until all writes are persisted. The use of independent logs in ZStore is similar to these approaches but ZStore's goal is not implementing the log abstraction but designing a near-zero-overhead key-value store with strong per-key guarantees.

Shared register protocols: ZStore's two phase notification system for linearizability bears some resemblance to shared atomic register protocols ([23, 27, 49]). Their quorum-based write has a benefit over our two-phase write protocol in terms of fault tolerance[10], but the shared registers require quorum-based reads with an additional quorum rewrite if conflicts are encountered. These quorum/multicast-based reads result in higher overhead even under one-phase read scenarios than ZStore, which always reads from a single node. Given the need for an HTTP gateway for S3 compatibility, the communication latency for a shared register becomes identical or larger (one extra RTT) than in ZStore.

Chain replication: This approach is used by a number of storage and shared log systems, such as CORFU [24], CRAQ [60], Harmonia [70], and Ceph [61]. Given the latency of SSD writes (many 10s of μs), serialization of device writes by chain replication incurs significantly higher latency than the parallel append operations performed by ZStore.

ZNS-related prior work: A large body of work focuses on adopting ZNS SSDs to improve performance of LSM tree-based stores (*e.g.*, RocksDB), typically on ZenFS [38, 43, 54]. WAZone [47] and ZNSKV [64] both propose to use ZNS SSDs to further improve the LSM-tree compaction workload, where SSD lifetime can be extended, which improves throughput, space utilization, and write amplification. However, most of the research along this line focuses only on LSM-tree based storage systems, RocksDB to be more specific. These efforts adopt the new feature in ZNS SSD, and patch existing mechanisms to improve different aspect of SSD devices, such as performance, latency, or lifetime

The current implementation of ZStore uses Zone Append as a sequencer for an unreplicated shared log, constructing a distributed object store and managing conflicts between logs.

NVMe-over-Fabrics: Recent work such as Scalio [59] has explored the use of NVMe-oF to build a key-value store with RDMA. Scalio uses RDMA for their client interface, and to flatten data structures into memory. ZStore's phase 1 and phase 2 are messaging protocols, using RDMA writes for optimization, and we implement a standard HTTP/TCP S3 client interface.

7 Conclusion

This paper presents ZStore, a novel distributed object storage architecture that leverages a pool of unreplicated append-only shared logs and works without any file system or layer between the storage and the devices for reduced software overhead. We present the principles behind the design of ZStore that optimizes writes using the zone append command and ZNS SSDs, and introduce the mechanism which addresses the challenge of forming a total order from multiple, independent, partially ordered logs. We demonstrate how ZStore can address conflicts and recover from failures at different levels. We show that ZStore has the potential to enable a fast, efficient distributed object store.

8 Acknowledgments

We thank the anonymous reviewers of SoCC'25, SOSP'25, ATC'25, and ApSys'24 for useful feedback. This research is supported in part by the Mass Open Cloud (massopen.cloud) and its industrial partners.

[10]Note that S3's HTTP transport allows ZStore to request a client retry in these cases.

References

[1] 2006. Amazon Simple Storage Service 2006-03-01. https://docs.aws.amazon.com/aws-sdk-php/v3/api/api-s3-2006-03-01.html

[2] 2010. Amazon S3 – Object Size Limit Now 5 TB | AWS News Blog. https://aws.amazon.com/blogs/aws/amazon-s3-object-size-limit/ Section: Blog.

[3] 2012. De-indirection for Flash-based SSDs with Nameless Writes. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. https://www.usenix.org/conference/fast12/de-indirection-flash-based-ssds-nameless-writes

[4] 2020. Specifications - NVM Express. https://nvmexpress.org/specifications/

[5] 2023. Ceph.io — Ceph Reef Freeze Part 2: RGW Performance. https://ceph.io/en/news/blog/2023/reef-freeze-rgw-performance/

[6] 2024. Amazon S3 - Cloud Object Storage - AWS. https://aws.amazon.com/s3/

[7] 2024. Intel Storage Performance Development Kit. https://spdk.io/

[8] 2024. *NVMe FDP - A promising new SSD data placement approach*. https://semiconductor.samsung.com/news-events/tech-blog/nvme-fdp-a-promising-new-ssd-data-placement-approach

[9] 2024. S3 API Reference - Amazon Simple Storage Service. https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html

[10] 2024. What is Amazon S3? - Amazon Simple Storage Service. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#ConsistencyModel

[11] 2025. Azure Blob Storage | Microsoft Azure. https://azure.microsoft.com/en-us/products/storage/blobs

[12] 2025. Boost.Asio - 1.87.0. https://www.boost.org/doc/libs/1_87_0/doc/html/boost_asio.html

[13] 2025. ConnectX-5 EN Card. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf

[14] 2025. DigitalOcean Spaces | S3-Compatible Object Storage. https://www.digitalocean.com/products/spaces

[15] 2025. Google Cloud Storage. https://cloud.google.com/storage

[16] 2025. HPE J2000 Dual IOM 2x100GbE NVMe-oF SFF JBOF Storage | SHI. https://www.shi.com/product/41942206/HPE-J2000-Dual-IOM-2x100GbE-NVMe-oF-SFF-JBOF-Storage

[17] 2025. HTTP and WebSocket built on Boost.Asio in C++11. https://github.com/boostorg/beast original-date: 2013-06-16T22:26:09Z.

[18] 2025. IBM Cloud Object Storage. https://www.ibm.com/products/cloud-object-storage

[19] 2025. MinIO | S3 & Kubernetes Native Object Storage for AI. https://min.io/

[20] 2025. NVM ExpressTM over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf

[21] 2025. NVMe Zoned Namespaces (ZNS) Devices | Zoned Storage. https://zonedstorage.io/docs/introduction/zns

[22] 2025. Swift - OpenStack. https://wiki.openstack.org/wiki/Swift

[23] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. In *J. ACM* (1995-01-03), Vol. 42. 124–142. https://doi.org/10.1145/200836.200869

[24] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[25] Matias Bjørling. 2020. Zone Append: A New Way of Writing to Zoned Storage. In *(VAULT 20)*. https://www.usenix.org/conference/vault20/presentation/bjorling

[26] 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. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[27] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 591–617. https://www.usenix.org/conference/nsdi20/presentation/burke

[28] 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 Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

[29] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage. In *19th USENIX conference on file and storage technologies (FAST 21)* (2021-02). USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[30] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 187–201. https://www.usenix.org/conference/fast25/presentation/chen-weijian-impress

[31] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[32] Zhuohui Duan, Hao Feng, Haikun Liu, Xiaofei Liao, Hai Jin, and Bangyu Li. 2025. AegonKV: A High Bandwidth, Low Tail Latency, and Low Storage Cost KV-Separated LSM Store with SmartSSD-based GC Offloading. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 321–335.

[33] N. V. M. Express. 2025. *Overcoming the Write Amplification Problem with NVM Express® Flexible Data Placement - NVM Express*. https://nvmexpress.org/nvmeflexible-data-placement-fdp-blog/

[34] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. 2020. It's Time to Revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[35] Will Glozer. 2025. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk original-date: 2012-03-20T11:12:28Z.

[36] Dr Werner Vogels https://www.allthingsdistributed.com. 2021. Diving Deep on S3 Consistency. https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html

[37] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (USA, 2010-06-23) *(USENIX ATC'10)*. USENIX Association, 11.

[38] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. 2022. Accelerating RocksDB for small-zone ZNS SSDs by parallel I/O mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track (Middleware Industrial Track '22)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3564695.3564774

[39] Yichen Jia, Eric Anger, and Feng Chen. 2019. When NVMe over Fabrics Meets Arm: Performance and Implications. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. 134–140. https://doi.org/10.1109/MSST.2019.000-9 ISSN: 2160-1968.

[40] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[41] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[42] Igjae Kim, J. Hyun Kim, Minu Chung, Hyungon Moon, and Sam H. Noh. 2022. A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 363–380. https://www.usenix.org/conference/fast22/presentation/kim-igjae

[43] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 93–99. https://doi.org/10.1145/3538643.3539743

[44] Asaf Levi, Philip Shilane, Sarai Sheinvald, and Gala Yadgar. 2024. Physical vs. Logical Indexing with IDEA: Inverted Deduplication-Aware Index. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 243–258. https://www.usenix.org/conference/fast24/presentation/levi

[45] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[46] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 357–372. https://www.usenix.org/conference/osdi18/presentation/lockerman

[47] Linbo Long, Shuiyong He, Jingcheng Shen, Renping Liu, Zhenhua Tan, Congming Gao, Duo Liu, Kan Zhong, and Yi Jiang. 2024. WA-Zone: Wear-Aware Zone Management Optimization for LSM-Tree on ZNS SSDs. In *ACM Transactions on Architecture and Code Optimization* (2024-01-18), Vol. 21. 16:1–16:23. https://doi.org/10.1145/3637488

[48] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312.

[49] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)* (USA, 1997-06-25) *(FTCS '97)*. IEEE Computer Society, 272.

[50] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 1–16. https://www.usenix.org/conference/fast21/presentation/ma

[51] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 106–122. https://doi.org/10.1145/3452296.3472940

[52] Mark Nelson. 2024. Hotsauce S3 Benchmark. https://github.com/markhpc/hsbench

[53] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage. 1–15.

[54] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. 2023. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access* 11 (2023), 26344–26357. https://doi.org/10.1109/ACCESS.2023.3257354 Conference Name: IEEE Access.

[55] Seonggyun Oh, Jeeyun Kim, Soyoung Han, Jaeho Kim, Sungjin Lee, and Sam H. Noh. 2024. MIDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 259–275.

[56] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. 2025. Cloudscape: A Study of Storage Services in Modern Cloud Architectures. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 103–121. https://www.usenix.org/conference/fast25/presentation/satija

[57] ScyllaDB. 2025. Seastar. https://seastar.io

[58] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[59] Xun Sun, Mingxing Zhang, Yingdi Shan, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2025. Scalio: Scaling up DPU-based JBOF Key-value Store with NVMe-oF Target Offload. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)* (2025). 449–464. https://www.usenix.org/conference/osdi25/presentation/sun

[60] Jeff Terrace and Michael J. Freedman. 2009. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (USA, 2009-06-14) *(USENIX'09)*. USENIX Association, 11.

[61] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. https://www.usenix.org/conference/osdi-06/ceph-scalable-high-performance-distributed-file-system

[62] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 31–31. https://doi.org/10.1109/SC.2006.19

[63] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07 (PDSW '07)*. Association for Computing Machinery, New York, NY, USA, 35–44. https://doi.org/10.1145/1374596.1374606

[64] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. 2022. ZNSKV: Reducing Data Migration in LSMT-Based KV Stores on ZNS SSDs. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. 411–414. https://doi.org/10.1109/ICCD56317.2022.00067 ISSN: 2576-6996.

[65] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. 2024. IONIA: High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 225–241. https://www.usenix.org/conference/fast24/presentation/xu

[66] Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2022. Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 381–396. https://www.usenix.org/conference/fast22/presentation/yang

[67] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 65–80. https://www.usenix.org/conference/fast23/presentation/yu

[68] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, and Si Wu. 2022. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 397–412.

[69] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.

[70] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. In *Proc. VLDB Endow.* (2019-11-01), Vol. 13. 376–389. https://doi.org/10.14778/3368289.3368301

[71] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. 2024. In-Memory Key-Value Store Live Migration with NetMigrate. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 209–224. https://www.usenix.org/conference/fast24/presentation/zhu