Re-architecting Distributed Block Storage System for Improving Random Write Performance

Myoungwon Oh*, Jiwoong Park†, Sung Kyu Park*,
Adel Choi*, Jongyoul Lee*, Jin-Hyeok Choi*, and Heon Y. Yeom‡
*Samsung Electronics, †Sungshin Women's University, ‡Seoul National University

2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00019

***Abstract*—In cloud ecosystems, distributed block storage systems are used to provide a persistent block storage service, which is the fundamental building block for operating cloud native services. However, existing distributed storage systems performed poorly for random write workloads in an all-NVMe storage configuration, becoming CPU-bottlenecked. Our roofline-based approach to performance analysis on a conventional distributed block storage system with NVMe SSDs reveals that the bottleneck does not lie in one specific software module, but across the entire software stack; (1) tightly coupled I/O processing, (2) inefficient threading architecture, and (3) local backend data store causing excessive CPU usage.**

To this end, we re-architect a modern distributed block storage system for improving random write performance. The key ingredients of our system are (1) decoupled operation processing using non-volatile memory, (2) prioritized thread control, and (3) CPU-efficient backend data store. Our system emphasizes low CPU overhead and high CPU efficiency to efficiently utilize NVMe SSDs in a distributed storage environment. We implement our system in Ceph. Compared to the native Ceph, our prototype system delivers more than 3x performance improvement for small random write I/Os in terms of both IOPS and latency by efficiently utilizing CPU cores.

***Index Terms*—Distributed storage system, NVMe SSD, Performance**

I. Introduction

The advent of fast storage technology poses unprecedented challenges in the software stack; the performance bottleneck is shifted from storage devices to software [25], [37], [39]. To overcome the software overhead, many researches focus on investing CPU cycles in optimizing storage access [31], [38], which helps achieve the full potential of a high performance storage device. High CPU usage is not a problem as long as there is enough CPU power available in the system.

However, the situation becomes different in a distributed storage system environment due to the complex architecture where multiple software layers are involved in the I/O path. For instance, a modern large-scale storage system usually consists of a number of storage nodes connected via network and they communicate with each other all the time for cluster-level I/Os, which requires both network and storage works, thereby consuming high CPU resources. Therefore, the existing optimizations [31], [38] trading CPU cycles for I/O latency are no longer effective, aggravating the CPU load. Moreover, it is common for a storage server to have multiple NVMe SSDs, which consequently increases the amount of work for I/O processing in each node by multiple times. Due to all these reasons, storage nodes tend to lack CPU resources especially when handling high IOPS for small random I/O requests; because the cost of context switches is expensive [4].

This CPU-bottleneck issue has been observed even in an industry-leading distributed block storage system, Ceph [34]. With a roofline-based performance analysis [35] on Ceph with NVMe SSDs, we identify three sources of the CPU-bottlenecked problem under random write workloads. First, tightly coupled I/O processing makes it hard to optimize for CPU efficiency without sacrificing client experience. In strongly consistent distributed system, data must be replicated and persisted before responding to a client request. However, due to the lack of decoupling between latency-critical jobs and best-effort batch jobs within the I/O path, it is hard for the I/O path to be optimized with two different optimization strategies: favoring low latency or high CPU-efficiency. Second, inefficient threading architecture that requires frequent context switches between threads. It is well known that the context switching overhead becomes non-negligible with fast devices [4], [25]. Third, local backend data store causing excessive CPU usage. Key-value stores—widely used as a backend data store for many conventional distributed storage systems—are known to use considerable CPU cycles for compaction and flushing [11], [18], [19].

To this end, we re-architect a modern distributed block storage system for improving random write performance in all-NVMe storage configuration, focusing on low CPU overhead and high CPU efficiency. In this paper, we present the following three key design choices for a new I/O architecture that enables CPU-efficient I/O processing.

1) Decoupled operation processing using non-volatile memory (NVM): We split the I/O operation into two phases: the top-half and the bottom-half. The top-half replicates and persists updates to a user-level log in NVM while the bottom-half asynchronously commits the logged updates to disk. By doing so, we can optimize the bottom-half for CPU efficiency without sacrificing client experience.
2) Prioritized thread control: We use different threading models for different optimization strategies; an event-driven threading model for latency-critical jobs and a traditional thread pool model for best-effort batch jobs. In addition, they are scheduled to the cores of separate CPU pools to eliminate the opportunity for priority-inversion.

Jiwoong Park† was with Samsung Electronics.

2575-8411/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDCS51616.2021.00019

3) Highly CPU-efficient backend object store: We design an in-place update based object store to minimize the host-side WAF and CPU overhead, which are inevitable in out-of-place update based object store that sequentializes random writes and requires background maintenance operations.

We implement the design in Ceph. The extensive evaluation results show that our design can yield up to 3x higher IOPS for small random write workloads while achieving 60% CPU load reduction, compared to the native Ceph. In addition, the results also demonstrate that the three design choices contribute to the improved performance of our system.

The main contributions of this paper are as follows:

- A roofline-based performance analysis of a conventional distributed storage system with multiple NVMe SSDs, demonstrating the source of the bottleneck is heavy CPU consumption across the entire software stack.
- Three design choices for CPU-efficient I/O processing of distributed block storage systems, but each of which can be individually applied to solve the similar problems in other systems, such as key-value stores and DBMSs.
- An extensive evaluation of our design compared to the industry-leading distributed storage system on synthetic benchmarks and real-world workloads.

II. Background

In this paper, our aim is to build a distributed block storage system delivering high random write performance by re-architecting the Ceph. This section provides the key characteristics of the block storage service, a brief overview of Ceph's architecture and Ceph I/O path, introducing terms that will be used throughout the paper.

A. Block Storage Service

In cloud ecosystems, distributed storage systems are used to provide persistent block storage services, which are responsible for providing virtual block storage devices that can be attached to virtual machines or containers. Block storage has been widely used for mission-critical and data-intensive applications in enterprise systems. A key characteristic of block storage that is necessary for the application built on top of it is strong consistency [33]; read requests always return the most recent data. Although it hides the complexity of the underlying distributed system from applications, achieving strong consistency does not come for free; data must be replicated and persisted before responding to a client request, thereby resulting in increased client-perceived latency.

Although writes can be buffered, random write performance is still important for many applications, such as DBMS, that use direct I/O or fsync to synchronize all the updates to the storage. Moreover, buffering all writes in memory is not possible when the working set is large.

B. Ceph Overview

Ceph is the industry-leading open-source distributed storage system that provides not only block storage services but also object and file storage services. Ceph implements the block service on top of the distributed object store called RADOS. For example, assuming a 10GB block image and a chunk size of 4KB, block offset 0 will map to object A, and block offset 4096 will map to object B. RADOS stores and retrieves object data by calculating the location of the object using the CRUSH [34]. *Monitor* nodes maintain a *global map* of the cluster, which is required when calculating the location of the object using the consistent hashing algorithm. On the other hand, in server storage node, there is a key module called OSD (object storage device) daemon that performs two important jobs: (1) to retrieve the client's requests and send the responses, and (2) to store and retrieve data reliably on underlying storage devices such as HDDs and SSDs.

C. Ceph OSD I/O Path

An OSD daemon consists of three modules: messenger, OSD core, and backend object store.

Messenger. It has a pool of *messenger threads* that are responsible for *message processing*; that is, message exchange with other nodes. Once it receives an I/O request from clients, it inserts the request to the corresponding PG work queue—PG is the logical group of objects.

OSD Core. It has a pool of PG threads that are responsible for *replication processing* and *transaction processing* for the incoming requests in the PG work queue. It retrieves the object's metadata, and then issues both replication requests to other nodes and a transaction request to the local backend object store. Note that we focus on primary-backup [15] based replication for data redundancy because erasure coding is not suited well for high performance [13].

Backend Object Store. It is responsible for making the transaction durable. The current Ceph's default object store is BlueStore [1]. It uses RocksDB for storing Ceph's metadata while storing user data in raw disks. Since data is updated in an out-of-place manner, it requires maintenance works, such as compaction.

III. Motivation

A. Roofline Performance Analysis

To identify the main factors limiting the performance of Ceph when using NVMe SSDs, we conduct performance analysis inspired by the roofline approach [35]. For estimating the upper bound on performance, we compare the performance of the baseline Ceph (`Original`) with those of three modified versions of Ceph: `RTC-v1`, `RTC-v2`, and `RTC-v3`. `RTC-v1` implements a run-to-completion(RTC) model, where a single thread performs the complete processing of a client request; that is, it includes message processing(MP), replication processing(RP), transaction processing(TP), object store(OS), and maintenance task(MT), which is required for the default object store (BlueStore). `RTC-v2` is implemented to show the performance of the RTC model when no object store related overheads exist; the write requests to the backend object store immediately return success. Thus, it includes only MP, RP, and TP. `RTC-v3` is implemented to show the performance

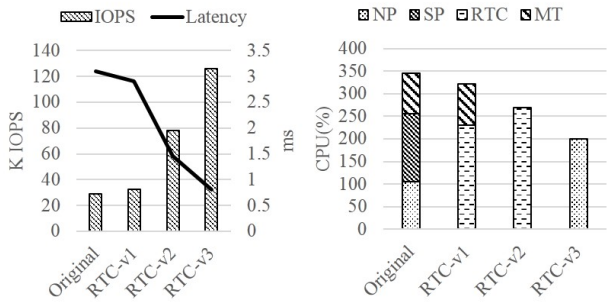


Fig. 1: Latency and CPU usage with 4KB random write workload. `NP` represents the portion of the CPU time for *Network Processing* that involves `MP` and `RP`. `SP` represents the portion of the CPU time for *Storage Processing* that involves `TP` and `OS`. `RTC` represents the portion of the CPU time for both *Network Processing* and *Storage Processing* when the RTC model is used. `MT` represents the portion of the CPU time for *Maintenance Task* required for the backend object store (e.g., compaction and sync). Note that `RTC-v1` performs `MP+RP+TP+OS+MT`, `RTC-v2` performs `MP+RP+TP`, and `RTC-v3` performs `MP+RP`.

	User	Data	Misc	Total
Original (GB)	21	42	78	120

TABLE I: Host-side write amplification during a small random write test. (User: written by the client, Data: written data with replication, Misc: written data except for Data, Total: total written data)

of RTC model when transaction processing and object store related works incur no overheads. Thus, it includes only MP and RP.

We run fio [2] with direct I/O and 4KB random writes in our Ceph cluster whose detailed configuration can be found in Section V. To eliminate other interference factors, we configure `Original` to use two *messenger threads* and two *PG threads*, while setting CPU affinity of the Ceph process to run on only four cores. For a fair comparison, we keep the total number of threads used by Ceph to four threads per node. Note that it does not include the number of threads used by the backend object store.

Figure 1 shows the performance results. In `Original`, each node shows CPU usage of 346% while achieving 29K IOPS in total for the cluster. Assuming that its performance scales linearly with the number of cores in the system, the maximum ideal performance would be about 369K IOPS (each node has 44 logical cores). This is still low when considering the number of NVMe SSDs used (8 SSDs in total); For 4K random write workloads, NVMe SSD can deliver up to 330K IOPS in the FOB (Fresh Out of Box) state or transition state. Note that the SSD performs 160K IOPS in the steady state. The CPU usage is too high for such low performance.

B. Issues with current block storage system on NVMe SSDs

Tightly Coupled I/O Processing. The problem is a long write path where the latency-critical tasks and the best-effort batch tasks are tightly coupled. Figure 3 (a) shows the write procedure with the primary-backup replication. For a single write request, it requires at least six steps: ❶The primary receives a request from a client. ❷The primary persists data in storage. ❸The primary sends a replication request to replicas. ❹The secondary persists data in storage after receiving the replication request. ❺The primary waits for the I/O completion acknowledgment from replicas. ❻The primary sends the result to the client.

The result from `RTC-v3`, in Figure 1, shows that even without the overheads of transaction processing, object store, and maintenance task, its average latency (0.8 ms) is higher than the 4KB random write latency of NVMe SSD (0.4 ms) with CPU usage of 200%. This implies that additional write procedure to commit a write aside from network processing or delaying replication processing causes high latency, and the I/O path already uses a considerable amount of CPU. Therefore, the write I/O path must be redesigned to achieve low latency and low CPU consumption, reducing the CPU usage as much as possible, which are hard to be realized at the same time; different optimization strategies are required for each of them. Thus, decoupling between latency-critical jobs and best-effort batch jobs within the I/O path is required.

Inefficient Threading Architecture. Most distributed storage systems use the thread-pool model where it manages different pools of threads for different purposes [8] [34] [37]. For example, in Ceph, there are two thread pools: a pool of *messenger threads* for network processing and a pool of *PG threads* for storage processing. However, this architecture leads to frequent context switches between threads, because a I/O request is handled over a long path where both *messenger* and *PG threads* are involved multiple times, thereby incurring the context switching overhead, which becomes non-negligible with fast devices [4], [25].

To solve this performance issue with fast devices, recent studies [4] have applied the RTC model into their network solutions, where each packet is processed completely before any other packet is processed. However, although the RTC approach has benefits for latency and CPU consumption by minimizing the context switching overhead and by improving cache efficiency, the benefits cannot be achieved without careful design consideration. For instance, assume that we replace the conventional thread-pool model in Ceph with an RTC model where each RTC thread is pinned to a dedicated core. Note that to prevent reordering of the requests from a single client, all the requests from the same client are handled by the same *messenger thread*. Therefore, an RTC thread cannot process the other I/O requests until the ongoing I/O operation is complete. It delays the response time.

The performance comparison between `Original` and `RTC-v1` in Figure 1 demonstrates this. Compared to `Original`, the simple RTC model (`RTC-v1`) achieves slightly better performance at lower CPU usage by mitigating context switching overhead. Also, the simple RTC model without object store and maintenance task (`RTC-v2`) achieves high performance. However, its latency is still slower (1.45 ms) than NVMe device, and the performance improvement is only marginal because the RTC thread is blocked waiting

on the acknowledgment of write completion from replicas. Therefore, it is required an effective thread control scheme to prevent from having a long critical section.

Local Backend Object Store causing High CPU Usage. BlueStore [1], Ceph's default backend object store, directly manages storage devices without using local file systems to resolve the problems, such as journaling of journal anomaly [1], [30] and unnecessary translation overhead between *object* in distributed storage systems and *file* in local file systems. However, BlueStore can become CPU-bound under random I/O workloads when using NVMe SSD.

There are two main reasons for this. First, host-side write amplification. BlueStore uses a LSM-tree-based key-value store (RocksDB) for metadata and small data writes. However, all log-structured based storage systems [27], [28], which make all I/O patterns sequential for better random write performance, necessitates a costly compaction (or garbage collection) process to clean up obsolete data. It causes severe host-side write amplification [3], [18], as shown in Table I; the host-side write amplification caused only by backend data store is about 3. Moreover, although it is performed in background, it not only disturbs foreground I/Os but also requires high CPU time to run it [11], [19]. Figure 1 shows that the background jobs (MT) from the backend data store account for non-negligible part of the total CPU usage. Second, a single data domain. Consistency mechanisms used by modern backend data store can be a performance bottleneck for scalability [22]. For example, BlueStore requires synchronization primitives for transaction processing within a single partition. In order to overcome such problems, some studies have proposed a partitioned domain approach for local file systems [12] [16]. However, since local file system does not expose this partitioned domain information to applications, distributed storage systems are hard to achieve locality-aware processing.

IV. Design and Implementation

In this section, we present a design of a distributed block storage system that is highly optimized for CPU efficiency to deliver high random write performance in a all-NVMe configuration. Figure 2 shows the high-level overview of our design. To address the problems explained in Section III, the following design choices are made: 1) Decoupled operation processing, 2) Prioritized thread control, and 3) CPU-efficient object store.

A. Decoupled Operation Processing using NVM

Overview. Logging in NVM—e.g, Intel Optane [36] or battery-backed DRAM—for faster persistence is not a new idea. However, we exploit it to decouple the latency-critical tasks from the I/O path without losing reliability and durability. To do so, we carefully decouple the latency-critical part (MP, RP) from other parts of the I/O path, with understanding of a distributed storage system architecture. In addition, our design treats I/O requests differently without losing strong consistency promise: 1) processing the latency-critical job

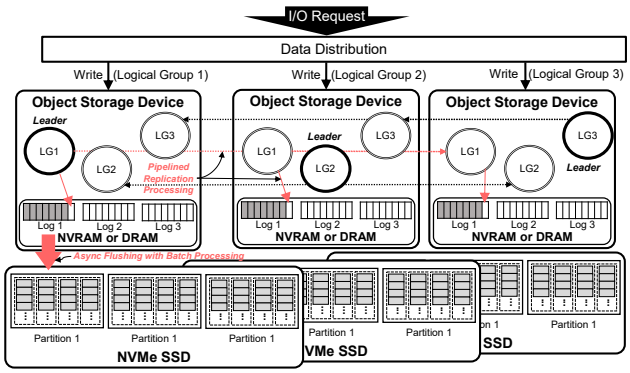


Fig. 2: High-level overview of proposed storage system environment.

using operation-log, allowing replicas to give fast responses to requests. 2) processing the best-effort job in a batch-processing manner using the underlying object store.

Figure 3 (b) shows the changed I/O behavior with *Decoupled operation processing*. Compared to the conventional replication processing, it requires fewer steps to commit a write operation, which leads to lower latency. ❶ The primary receives a request from a client. ❷ The primary sends a replication request to replicas as soon as incoming operation is logged in NVM. A read request can be served without violating strong consistency by referencing the log in NVM. ❸ The secondary sends a write completion acknowledgment back to the primary once the data is stored in memory space. ❹ The primary replies to the client after all ACKs are received from replicas. The staged operation is asynchronously flushed to the backend object store once the amount of staged data reaches a threshold.

NVM logging is only effective for performance as long as flushing can be done asynchronously; if the NVM is full, flushing needs to be synchronously done before handling I/O operations. Therefore, in the case of write-intensive workloads that keep the NVM full, NVM logging does not help for performance; it only wastes CPU resources. However, our method is feasible even with only a small amount of NVM because the underlying storage device (NVMe SSD) is fast enough, compared to the other parts of I/O processing; flushing can be finished even before the next I/O request arrives (see Section III-B). Therefore, the system can keep the amount of data logged in NVM to a very minimum.

One may concern that logging data in NVM could break the strong consistency guarantee of the system—a read could return the stale data—because some recent writes could have not been committed to disk. To address this, two data structures are used on a per-logical group basis: *operation log* and *index cache*. The former stores the incoming operations sequentially in a log-structured manner while the latter contains the cache information about recent write per object ID. We use hash map to keep track of the recent writes per object ID. *Operation log* is used as a producer/consumer buffer for communication between priority/non-priority threads, and enables asynchronous completion of operations. Figure 6 describes the details of

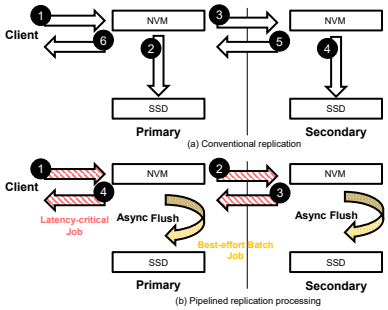


Fig. 3: Changed replication behavior with *Decoupled operation processing*.

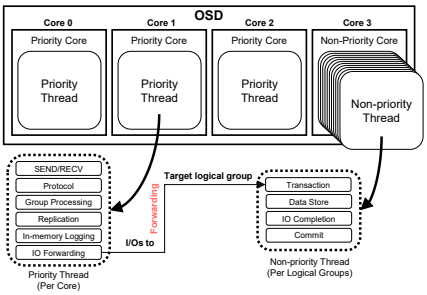


Fig. 4: *Prioritized thread control.*

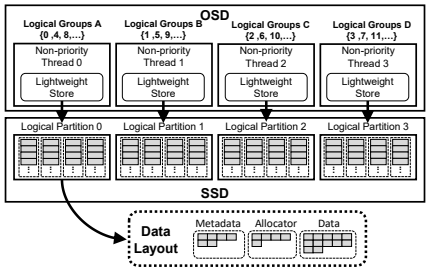


Fig. 5: *CPU-efficient object store.*

Decoupled operation processing with the *operation log* and the *index cache*.

I/O Procedure. Based on these data structures, the I/O procedure is as follows.

1) Write: Upon a write request for an object, one of the *priority threads* in an OSD daemon appends the *log entry*—⟨logical group ID, version, sequence number, offset, data, operation type⟩—to the corresponding *operation log*, based on the logical group of the object, which is calculated by the object ID (W1 in Figure 6). This logging step requires the acquisition of a logical group lock because a *non-priority thread* can access the same *operation log*. After that, without releasing the lock, an *index entry*—⟨object ID, version, sequence number, offset, length, operation type, a position of the *operation log*, *object state*⟩—is inserted to the *index cache* (W2). An *index entry* allows searching for the objects that exist in the *operation log*, maintaining object state (e.g., reading or writing), version number, sequence number, and access time. Note that we insert the *index entry* into the *index cache* even when there is an entry that has the same object ID. We do not overwrite them because each index entry is required to track the operations in the operation log. If there are read requests whose object ID is the same as the write request, the *priority thread* enforces a flush of the operations in the *operation log* by waking the *non-priority thread* (W3).

2) Read: For guaranteeing strong consistency, an OSD daemon has to return the latest data to the client's read requests. To do so, upon a read request for an object, a *priority thread* in an OSD daemon first searches the *index cache* for the recent writes that have the same object ID as the read request with acquiring a lock. If it is not in the *operation log*, the read request is appended in the *operation log* and it is then handled by the *non-priority thread* (R2 in Figure 6). If found to be in the *operation log* and if the length of the request is not larger than it of the *log entry*, the *priority thread* directly retrieves the object from the *operation log* (R1). If larger, the request is appended in the *operation log* and *priority thread* wakes *non-priority thread* so that it will read the object from the backend data object and overwrite the *log entry* with the latest object data (R3).

Unlike a write request, which can be buffered, read requests should be serviced synchronously because the client waits for

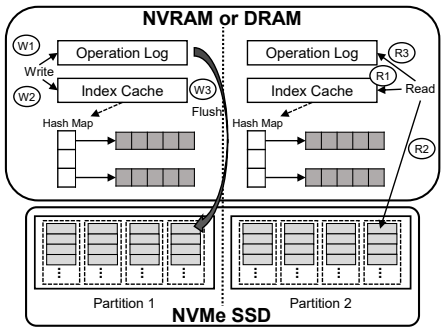


Fig. 6: *Decoupled operation processing* with index cache (Write sequence: W1 → W2 → W3, Read sequence: R1 → R2 → R3)

the data. Thus, *priority thread* needs to wake *non-priority thread* to handle the read request. However, there is no huge overhead because the number of operations in the *operation log* is kept small as possible.

3) Flush: A flush occurs if the number of operations in the *operation log* is larger than a predefined threshold; the default value is 16, which is determined empirically. The flush condition is checked whenever any operations are completed or a timeout is exceeded. In case of a flush, the log entries in the *operation log* will be flushed to the backend object store. Then, all the related data is removed both in the *operation log* and *index cache*.

4) Recovery: Since the requests are sequentially appended to the *operation log*, the writes are persisted in the order of the requests received. However, without considering the presence of the *operation log* and *index cache*, there can be a potential consistency mismatch in case of system failure. To this end, our design involves a recovery procedure as follows. Note that we assume a distributed storage system that requires the latest copy of the *global map* for locating objects.

❶ For a logical group, the requests are replicated to the replicas' *operation logs*. ❷ One of the storage nodes crashes. ❸ The other nodes in the same replication domain are notified that a failure occurs. Before starting the recovery procedure, the other nodes perform a flush to make the latest data persistent without removing entries in both *operation log* and *index cache*. ❹ A global map is updated to reflect the server's failure. ❺-a. If a secondary node crashes, its *operations log* and *index cache* are recovered from them of the primary node.

❺-b. If a primary node crashes, they are recovered from the secondary nodes. ❻ A new storage node is assigned to the logical group to replace the failed storage node. ❼ It performs synchronization with the other nodes in the same replication domain.

B. Prioritized Thread Control

Overview. *Prioritized thread control* aims to avoid I/O priority inversion between the latency critical jobs and the best effort jobs. To do that, it maintains separate CPU pools for different thread types. In addition, this scheme involves thread scheduling policy for efficient CPU utilization. Figure 4 shows the threading architecture with *prioritized thread control*.

Prioritized Thread Control. There are two thread types: *priority thread* for latency-critical tasks and *non-priority thread* for the others; the former is responsible for message exchanging and replication processing while the latter is in charge of storage processing. Although both thread types follow the RTC model for locality-aware processing, they differ in how they are scheduled to cores to run the task. For example, each *priority thread* is pinned to a dedicated core because interference from other threads can affect the client-perceived performance. In contrast, *non-priority threads* share the remaining cores and each of them is responsible for storage processing of a different logical group of objects; an object is assigned to one of the logical groups based on the object ID. A key concept behind this threading architecture is to give high priority to latency-critical tasks, so that a write operation can be completed with low latency (by *priority thread*) while reducing CPU consumption by flushing writes in a batch-processing manner (by *non-priority thread*).

1) Priority thread: A priority thread has two roles: message handling and operation logging. Each manages two queues (send/receive queues) and waits for either a send queue event or a receive queue event to handle network I/Os. If the *operation log* has more entries than a predefined threshold value, or if the read has to be serviced from the backend object store, the *priority thread* sends a wake-up signal to the corresponding *non-priority thread*. Once the *non-priority thread* completes storage processing, it enqueues the result to the *priority thread*'s send queue. Then, the *priority thread* sends an ACK to the client after dispatching the result from the queue. One may think that letting the *non-priority thread* reply would be a better choice. However, to preserve the order between operations from a client, a single processing thread per connection is a practical way; receiving and sending have to be processed by the same thread.

2) Non-priority thread: A *non-priority thread* plays two roles: issuing flushes and completing I/Os. After waking up, either by a signal or by a timeout, a *non-priority thread* processes the operations in the *operation log* in a batch-processing manner. Note that although it does not batch multiple small requests into a single large request, it greatly reduces the scheduling overhead between *non-priority threads*; they compete for CPU time. A *non-priority thread* issues an I/O to the backend object store, enqueues the result after waiting for the I/O completion, repeats this until no more pending request presents in the *operation log*, and goes to sleep.

3) Thread Scheduling: Our design involves effective thread scheduling among latency-critical tasks and best-effort batch tasks. Each *priority threads* is pinned to a dedicated CPU core when the OSD daemon starts. When awakening the *non-priority thread*, the *priority thread* checks if there is a *non-priority core* that can afford to run the task. If exists, the *priority thread* schedules the *non-priority thread* to that core. Otherwise, it leaves the scheduling work to OS scheduler; *non-priority thread* can be scheduled on one of *priority cores*.

C. CPU-efficient Object Store

Overview. *CPU-efficient object store* is designed to minimize the number of I/Os required to perform an operation issued by a client and to maximize parallelism during storage processing. Since it runs on a raw device, it has several advantages over using local file systems, as explained in Section III-B. Figure 5 shows the high-level architecture of *CPU-efficient object store*.

Towards an object store that induces no host-side write amplification, it uses an in-place update disk layout for each partition such as journaling file systems [5], [32]; it consists of metadata, block map, and data areas. Since it allows overwrites, no cleaning process, such as compaction, is required. While it may seem counter-intuitive to use the flash-unfriendly layout for the best performance with an NVMe-only system, it does so by reducing not only host-side write amplification but also CPU consumption on the host side.

To further reduce the number of I/Os required for metadata update, we use the pre-allocation technique, which is commonly used to avoid unnecessary metadata update [17]. It exploits the fact that overwriting the pre-allocated objects does not need any changes to their metadata such as block bitmap, inode table and etc. Thus, no metadata update is required. This is only feasible if the object size is fixed. Fortunately, in typical block storage services, a block device image is striped over fixed size objects. For example, in Ceph RBD, which is Ceph's block device service, the default object size is 4MB. Thus, we can pre-allocate all the objects at image creation time.

For high performance, our object store uses NVM as a metadata cache. It also divides the entire disk space into partitions such that one *non-priority thread* can be assigned to one partition. Therefore, I/O operations can be handled in parallel without lock contention.

Basic operations. The procedures for basic operations are as the following.

1) Look up: To look up the object, *CPU-efficient object store* uses the radix tree where the object ID is the key; the high bits of object ID represents the logical group ID. A few bits (leftmost bits) of the object id are used to determine the sharded partition where the object is located, and the rest of bits are used for object look up within the sharded partition.

2) I/O Distribution: Objects are distributed to the sharded partitions depending on their logical group id—each sharded partition has a responsibility for dealing with the logical group ID. A sharded partition is assigned to one of non-priority threads via simple modulo hashing (e.g., total sharded partition number % the number of non-priority thread). General transactions—writes to the object including the metadata— are handled within a logical group.

3) Write: Upon a write request, the object is overwritten in the allocated data block in an in-place update manner, and then the *onode* (object metadata entry) is updated. At that time, the updated metadata—contains the key, the location where the value is recorded, and free list information—is written at the position of block_map in onode. Note that the pre-allocation technique helps avoid metadata update when the object size is known. If the write is complete, the object store updates the metadata in memory. Also, the corresponding entry in *operation log* will be deleted.

4) Read: Upon a read request, it first requires the location of metadata (*onode*) for the object. It can be obtained either by searching from the radix tree or by looking up the cached metadata. Then, the object can be read from the data block in the block_map in the *onode*.

5) Delete: We use the *delayed deallocation* technique to delete the existing objects. Upon a delete request, it marks the *onode* as deleted.

6) Crash Consistency: Operation log in NVM works like WAL [23]. Therefore, our design can maintain consistency for sudden crash. For instance, if crash occurs in the middle of flush, the store can recover the latest state by replaying the operation log (REDO), regardless of the pre-allocation technique used. Note that isolation and ordering between operations are provided in the higher layer (OSD Core).

7) Metadata Management: Metadata contains information that needs to be updated along with object updates, such as object version, and mtime. Since updating metadata triggers additional disk writes apart from data write, our design involves metadata cache layer in NVM to avoid additional writes. This is feasible because typical object size of distributed storage system is large (e.g., 4MB), so that the cache covers a large number of objects. If there is not enough space in NVM, an update on metadata area is required.

Typically, the versioning and the status information of the logical group are used for checking the current state of the group and keeping consistency within a group. In our design, those information is always kept in the *operation log* to reduce the number of I/Os. They can be recovered in case of a server crash because they are replicated across replicas. Also, to implement version control and rollback without log-structured layout, we can add postfix notation to the object name (e.g., OID = {OID:version}). By doing so, *CPU-efficient object store* can identify the version of the object and rollback to a previous version.

Disk Layout. In our current design, the disk space is statically partitioned such that a *non-priority thread* is in charge of only a single partition; it can not access the other partitions. When deploying storage cluster, each sharded partition needs to be allocated according to the configuration that specifies the partition size. Each partition has *freeblock tree info*, *onode tree info*, and *date blocks* areas, similar to XFS [32].

1) Superblock: It contains partition-level information. For instance, it has start pointers to each partition area, the number of total *onodes* and the size of each partition; it is similar to existing local file systems such as XFS [32].

2) Freeblock Tree Info Area: The root node information of free data blocks is stored in this area. Also, the count of the free data blocks is maintained on per sharded partition basis. Like XFS [32], *CPU-efficient object store* constructs a b+tree to track all of the free data blocks.

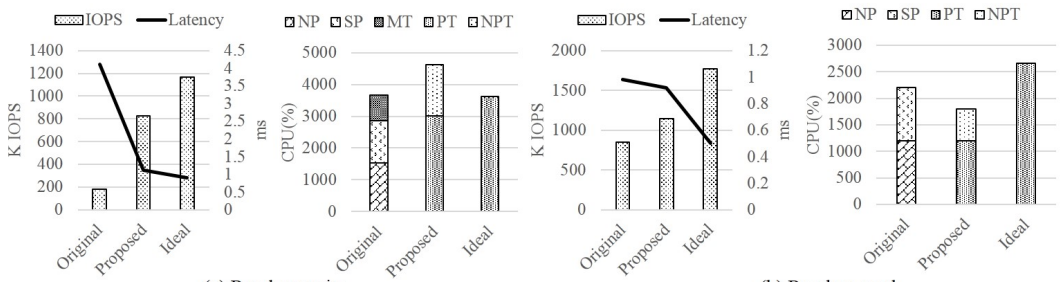
3) Onode Tree Info Area: The main purpose of this area is to store information, where the root onode is located. Using the root onode, other onode allocated in the data blocks area can be tracked. Each onode contains the radix tree nodes (e.g., sibling, child) for looking the onode up in the radix tree, which means *CPU-efficient object store* can search objects using tree node information in the onode. The onode has a fixed size (512 byte). If the data size is small, the onode can embed the data. Also, it contains block_map based on extent tree and extended attribute map which is b+tree to store key-value data. The block_map and extended attribute map are variable-length, and the onode only has a pointer to indicate the real data location of the block.

4) Data Blocks Area: This area is used for storing object data. The area is divided into fixed-size blocks.

V. Evaluation

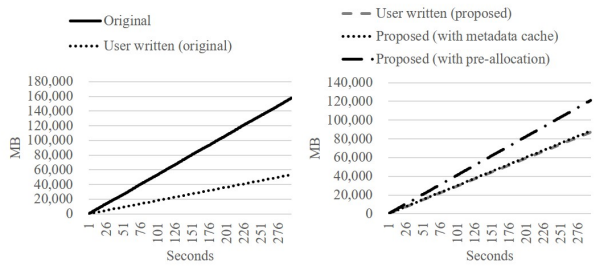
A. Environmental setup

We implement the proposed design in Ceph 15.0.2. We modify around 8,000 lines of code to prototype our proposal. With reference to Crimson [9], which is Ceph's ongoing project to replace the current Ceph OSD, we reuse a part of its design as the I/O path. We also implement a new object store based on BlueStore and XFS. For evaluation, we use nine machines in total, each of which is equipped with a 2-way Intel Xeon Gold 6152 2.1GHz with 22 cores (44 threads) per NUMA node and 128GB of DRAM memory. Four of them are used for storage nodes while five machines are used for clients generating workloads. They are connected with a 100 GbE network. The Ceph storage cluster is configured to use a replication factor of two considering the environment for block storage appliances [24]. Each storage node has two NVMe SSDs (Samsung PM1725a) and runs four OSD daemons per SSD; each NVMe SSD is partitioned into four logical partitions. Before each experiment, we perform secure-erase commands on every SSDs to return them to their FOB performance states. For our proposed system, eight *priority threads* and ten *non-priority threads* are used for each OSD daemon. We also use a ramdisk to emulate NVM (8GB per server node), which is used for *operation log*. We evaluate the block storage service and object service of Ceph using FIO and YCSB benchmarks.



(a) Random write (b) Random read

Fig. 7: IOPS and CPU usage of small random workload (NP represents *network processing*. SP represents *storage processing*. PT: priority thread, NPT: non-priority thread, MT: compaction and sync threads).



(a) Cumulative written bytes (Original). (b) Cumulative written bytes (Proposed).

Fig. 8: Host-side write amplification during small random write workload. All results are measured by IOSTAT.

We compare the performance of three systems: `Original`, `Proposed`, and `Ideal`. `Original` represents the results of default Ceph with BlueStore while `Proposed` shows the results of the proposed solution implemented on Ceph. Note that for a fair comparison, `Proposed` makes use of only a small memory space per partition (flush happens if there are 16 entries on the *operation log* per partition). `Ideal` represents the ideal performance of Ceph when storage processing overhead no longer exists, which is similar to the `RTC-v3` test in Section III. For performance metrics, we report the utilization of logical CPU cores as well as the IOPS and latency. For example, if two logical cores are fully used, CPU usage is 200 %.

B. Small Random Performance

To measure the small random performance, we run FIO [2] with 4KB random I/O for each 30GB RBD (RADOS Block Device) image using the RBD I/O engine while the number of jobs and the I/O queue depth are set to two and sixteen, respectively. Five FIO processes run for five minutes in each client (total 25 connections to the Ceph cluster).

In the left of Figure 7-(a), there is a huge performance improvement in `Proposed` compared to `Original`. `Proposed` achieves 820K IOPS with latency of 1.11 ms. On the other hand, `Original` achieves only 181K IOPS. This is too low when considering the number of NVMe SSDs used, although its performance may be improved by optimizing configuration.

As shown in the right of Figure 7-(a), `Original` shows a CPU usage of 3700 % per node and the storage processing

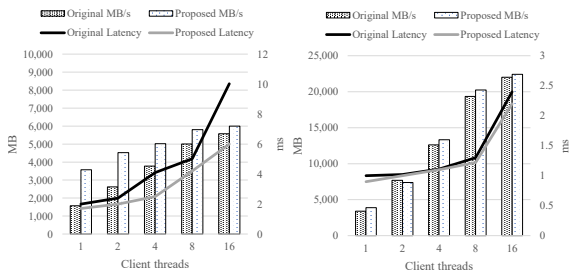
	Original	COS	PTC	DOP
K IOPS	181	471	641	820
Latency (ms)	4.3	3.1	2.2	1.11

TABLE II: Performance improvement in small random write by each proposed technique (COS: CPU-efficient object store, PTC: Prioritized thread control, DOP: Decoupled operation processing)

(`SP`) consumes a considerable amount of CPU cycles. We have two observations from this result. First, although a storage node has the enough number of storage threads (80 per node), they cannot be utilized efficiently; total CPU usage is only 1500 %. One of the reasons of this is the existence of *compaction threads*, which is used by RocksDB in BlueStore [1]. As shown in the right of Figure 7-(a), `MT` shows a CPU usage of 800 %; it is a non-negligible part of total CPU usage. Even though compaction threads do not use most of the CPUs, they have a negative impact on the performance because they compete for CPU resources and leads to a synchronization to other threads. Second, `Original`'s IOPS per core is low due to lack of efficient thread control and host-side write amplification. Note that `Proposed` uses more CPU cycles than `Original`. However, `Proposed` achieves higher IOPS per core than `Original`. There is still a gap between `Proposed` and `Ideal`. This is because `Proposed` could delay handling the incoming request due to acquiring logical group lock shared between priority thread and non-priority thread to store the log entry.

Figure 7-(b) shows a small random read result. `Proposed` achieves higher IOPS and lower latency than `Original`. This is because the read requests can be completed without a dependency with other I/Os if there are only read requests. Therefore, locality-aware processing has a large impact on performance. Note that a read request does not require additional I/Os to the replicas because a primary node handles all read requests.

Table II shows performance impact on the Ceph cluster when each technique is applied. COS has a considerable effect on performance improvement in IOPS because it reduces the number of writes from the write request. Note that Ceph issues many key-value writes (e.g., object_info_t, snapset, pglog) to the object store whenever a write request is being handled. PTC represents the result of COS with PTC. PTC reduces the latency by 1 ms because it provides the locality. However, there is still a latency gap between the `Ideal` case and the



(a) Seq. write (b) Seq. read

Fig. 9: Throughput of 128 KB sequential read and write on a four nodes cluster with different threads using FIO.

PTC. DOP shortens the latency gap by applying *Decoupled operation processing* with NVM.

C. Host-side Write Amplification

Figure 8 compares `Original` with `Proposed` in terms of the written bytes during the small random write experiment.

Figure 8-(a) shows how many bytes are written to storage and the number of bytes user issues. In `Original`, the total bytes written to storage (160 GB) is about three times larger than the user's writes including replication (56 GB). It is too much although Ceph's metadata adds some storage overhead (1020 - 2048 bytes per object). As many studies mentioned [18], [19], this is due to the RocksDB's write amplification.

On the other hand, Figure 8-(b) shows the result of `Proposed` regarding written bytes. With the pre-allocation technique, the host-side write amplification increases to about 1.4 because writing an object requires I/Os to metadata area in *CPU-efficient object store*. However, it is less than half of `Original`'s, thanks to the in-place update disk layout. In addition, with metadata cache, the number of written bytes is almost similar to the number of bytes user writes (slightly larger because flushing metadata happens). The use of the in-place update disk layout and the pre-allocation technique with metadata cache facilitates near-zero host-side write amplification.

D. Large Sequential I/O Performance

Figure 9 shows a large sequential performance result using FIO. We create one RBD image per connection and each client makes a mapping between its block device and a RBD image. Note that network or storage bandwidth is saturated with fewer IOPS than small random workload due to large block size (128 KB), so CPU consumption is not a bottleneck. Because the number of clients is limited, we increase the number of threads on five clients to evaluate sequential performance. In 16 threads result, the throughput is 5.5 GB because we use two NVMe devices on a server and four OSDs use an NVMe device. Therefore, even if an OSD issue 128 KB sequential writes, the workload becomes 128 KB random writes. On the other hand, the read throughput increases up to 22 GB/s which is nearly an `Ideal` result because each of the NVMe devices can achieve 3 GB/s.

In large sequential workload, what storage system needs is how the storage system processes the I/Os in parallel without considering the locality. `Proposed` can distribute its I/O to all CPUs in parallel. So, the performance is nearly similar to `Original`.

E. YCSB

To evaluate `Proposed` design under the real workload, we run YCSB [6] on the client nodes using block device. Each client uses increased record count (10000K) and thread number (10 threads) as a YCSB configuration parameter, and YCSB issues I/Os using Ceph's object service. Also, to compare `Proposed` to `Original` equally, in-memory space is limited (16 entries per partition) when `Proposed` is tested. Figure 10 shows the result of YCSB.

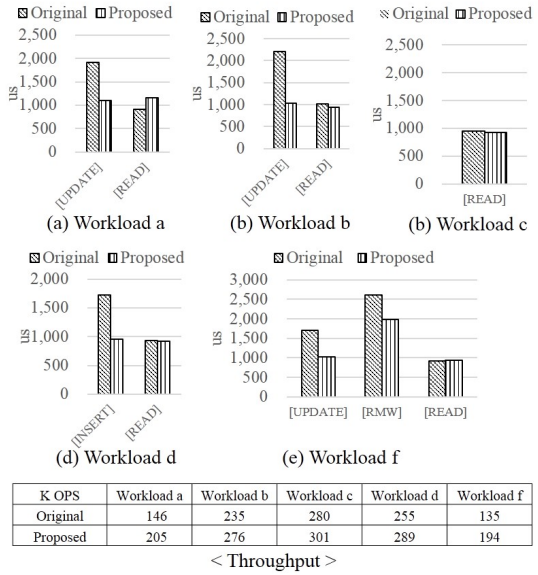
Workload a is a mixed workload (the ratio between a read and write is 50:50). Each client issues small and unaligned I/O to the cluster. When looking at the update result, `Proposed`'s write latency is significantly lower than `Original`. But, it is not as good as the latency `Proposed` achieves in Section V-B. This is because of unaligned data which has a smaller length than the block size. So, `Proposed` needs to overwrite data after reading the data as long as the block size when processing unaligned data. As a result, a read-modify-write occurs in the object store. The read latency is similar to `Original` because a flush occurs in `Proposed` and `Original` uses the cache in BlueStore. Also, in the case of `Original`, most of the reads hit the cache in the object store. On the other hand, `Proposed` does not use the data cache in the object store.

Workload b is also a mixed workload, but the ratio between a read and write is 95:5 (write once, read many). Due to most of the operation is a read and the size of the operation is small, `Proposed` can deal with the request with the locality. Therefore, the read latency is slightly lower than `Original`. Compared to workload a test, `Proposed` has low latency because frequent reads and writes lead to a flush from in-memory operation log to the storage. Despite the number of the write request is smaller than workload a, the update latency slightly higher than workload a case. This is because read/write control. A write should wait for the completion of many read requests. On the contrary, workload a issues read and write requests at a rate of 50 to 50.

Workload c is read-only workload. `Proposed`'s latency is slightly better than `Original`. Because `Proposed` can exploit the locality of thread as explained above.

Workload d is similar to workload b, but it focuses on the recent data. `Original`'s insert latency is better than workload b because re-write to the object happens. `Original` also uses fallocate() systemcall to reduce additional metadata write. However, due to a compaction and sync threads, the insert latency is much higher than `Proposed`. In the case of the read latency, `Proposed` is lower than `Original` unlike workload a. This is because workload d is read-dominant, a flush rarely happens.

Workload f is based on read-modify-write. As shown in Figure 10, `Original` takes considerable time to complete the operations, compared to `Proposed`. The reason why `Original` is much slower than Propose is the update latency.



K OPS	Workload a	Workload b	Workload c	Workload d	Workload f
Original	146	235	280	255	135
Proposed	205	276	301	289	194

< Throughput >

Fig. 10: YCSB latency ((a) - (e)) and Throughput with `Original` Ceph and `Proposed` design.

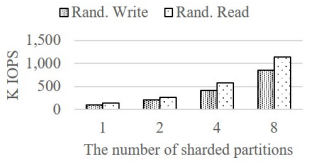


Fig. 11: Partition scalability.

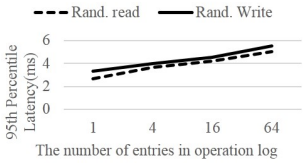


Fig. 12: Small Rand. read and write latency in write dominant workload.

The update latency is about 1.7 ms, while `Proposed` is only 1.02 ms.

F. Partition Scalability

We run the performance test with varying the number of sharded partitions in an OSD. Whenever the number of sharded partitions increases, the clients add six connections with the same workload. As shown in Figure 11, the performance is improved every time the number of sharded partitions increases.

G. Worst Case Latency

Figure 12 demonstrates 95th percentile latency when we issue a mixed random workload (the ratio between write and read is 80:20) at a constant rate (300K per second). `Proposed` has a drawback as to asynchronous flush because it forces the priority thread to flush all entries when a read request comes in. In Figure 12, 95th percentile latency grows considerably depending on the number of entries in an operation log because `Proposed` stores many entries at once to minimize context switch overhead.

VI. Discussion

User-level I/O Frameworks. To exploit the network or storage devices, a dedicated core based on polling can be used [7] [38]. However, the polling might not be the right solution for CPU-efficient I/O processing in a distributed storage system, especially for small random workloads where the performance bottleneck is CPU. If the distributed storage system uses those frameworks, there are two different I/O frameworks which have their own polling mechanism. In addition, polling wastes CPU resources if many I/Os do not continuously come into the storage system. Therefore, we believe that a new technique which can mitigate CPU consumption is needed rather than maximizing the storage device's performance via polling.

Device-side Write Amplification. Due to erase-before-write characteristics of NAND flash memory, in-place update strategy is known to lead to frequent device-side GC, which consequently increases WAF and shortens lifespan of the device. However, current distributed storage systems tend not to fully exploit modern NVMe SSDs. So, NVMe SSDs can afford to perform frequent device-side GC. Reduced burden on host CPU can result in improved I/O performance. Also, a recent study [21] reveals that most SSDs used in production do not reach maximum program erase cycle limit until the SSDs are replaced (less than two percent of the rated life is consumed on average). Thus, we no longer have to consider the life span of the device.

Metadata Overhead. In-place update based object store may require two extra writes for an object write to update internal metadata (metadata area and data allocation area) in the *no pre-allocation* scenario. As a result, CPU consumption and write amplification increase more than the *pre-allocation* scenario. However, we believe that it introduces still lower application-level WAF than the out-of-place update approach that requires host-side GC or compaction process.

VII. Related Work

Use of Non-volatile Memory. Several researches use NVM for decoupled I/O processing. Strata [14] exploits a user-level log in NVM to asynchronously manage longer-term storage in a 3-layer storage hierarchy. SplitFS [10] is a file system only for NVM and it proposes a new architecture that provides high performance with low software overhead by splitting of responsibilities between user- and kernel-level file systems. These works are similar to our approach in that they have a split architecture for efficient I/O processing and take advantage of NVM. However, our approach differs that we exploit NVM in a distributed storage system in order to have an opportunity of optimizing the I/O path for low CPU overhead without sacrificing client experience; the target of the above works is local storage systems.

Threading Architecture. SILK [3] introduces a user-level I/O scheduler for LSM-based key-value store. It prioritizes the operations that affect the performance the most. On the other hand, our approach is to have separate CPU pools for different priorities. However, we can adopt their approach for more efficient scheduling in a high CPU congestion scenario. Crimson [9] is an ongoing project in Ceph. It includes a new threading architecture based on Seastar framework [29] to minimize CPU overheads during I/O processing. We can adopt

their optimization techniques for the decoupled operations with NVM.

CPU-efficient Key-value Store. KVell [18] is a CPU-efficient key-value store design that focuses on the efficient use of the CPU. Our design principles are in line with that of KVell, such as no sequentializing random I/Os and use of sharded partition. However, our *CPU-efficient object store* is specifically designed as a backend store of Ceph, consuming raw devices for efficient transaction support.

Low overhead replication. Several studies [20] [37] minimize replication overhead by using asynchronous model. However, our proposal mainly targets block service without sacrificing strong consistency between replicas.

Asynchronous flush. Slm-db [11] and PebblesDB [26] present asynchronous flushing mechanisms allowing concurrent read/write, which is similar to our approach. However, they are designed for local LSM-based KV stores while ours is designed for distributed storage system to reduce CPU consumption.

VIII. Conclusion

In this paper, we have analyzed the performance problems of distributed block storage systems in an all-NVMe environment. Based on the observations, we present a design of high-performance low-latency distributed block storage system. It is realized by reducing CPU consumption and by eliminating unnecessary overheads, such as host-side write amplification and context switches. Extensive evaluation results show that our prototype significantly improves small random performance while using fewer CPU resources, compared to Ceph.

References

[1] A. Aghayev et al. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[2] J. Axboe. Fio-flexible io tester. https://github.com/axboe/fio, 2020. [Online].

[3] O. Balmau et al. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[4] A. Belay et al. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[5] M. Cao, S. Bhattacharya, and T. Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[6] B. F. Cooper et al. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[7] DPDK. Dpdk. https://www.dpdk.org, 2020. [Online].

[8] Gluster. Gluster: Storage for your cloud. https://www.gluster.org, 2020. [Online].

[9] S. Just. Crimson: A new ceph osd for the age of persistent memory and fast nvme storage. https://www.usenix.org/conference/vault20/presentation/just, 2020. [Online].

[10] R. Kadekodi et al. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*.

[11] O. Kaiyrakhmet et al. Slm-db: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[12] J. Kang et al. Spanfs: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, 2015.

[13] S. Koh et al. Exploring fault-tolerant erasure codes for scalable all-flash array clusters. *IEEE Trans. Parallel Distributed Syst.*, 31(6):1460, 2020.

[14] Y. Kwon et al. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 460–477, 2017.

[15] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[16] C. Lee et al. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*.

[17] W. Lee et al. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 235–247, 2015.

[18] B. Lepers et al. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[19] L. Lu et al. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[20] T. Ma et al. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*.

[21] S. Maneas et al. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 137–149, 2020.

[22] C. Min et al. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[23] C. Mohan et al. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[24] NetApp. Netapp. https://www.netapp.com/media/10511-tr4606.pdf, 2019. [Online].

[25] M. Oh et al. Lalca: Locality-aware lock contention avoidance for nvme-based scale-out storage system. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1143–1152, 2018.

[26] P. Raju et al. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514, 2017.

[27] RocksDB. A persistent key-value store for fast storage environments. https://rocksdb.org/, 2020. [Online].

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[29] Seastar. Seastar. http://seastar.io/, 2020. [Online].

[30] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 287–293, 2014.

[31] W. Shin et al. Os i/o path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, 2014.

[32] A. Sweeney et al. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[33] D. B. Terry et al. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324, 2013.

[34] S. A. Weil et al. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[35] S. Williams et al. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[36] K. Wu et al. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[37] J. Yang et al. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

[38] Z. Yang et al. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[39] J. Zhang et al. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.