The Case for Custom Storage Backends in Distributed Storage Systems

ABUTALIB AGHAYEV, Carnegie Mellon University
SAGE WEIL, Red Hat, Inc.
MICHAEL KUCHNIK, Carnegie Mellon University
MARK NELSON, Red Hat, Inc.
GREGORY R. GANGER and GEORGE AMVROSIADIS, Carnegie Mellon University

For a decade, the Ceph distributed file system followed the conventional wisdom of building its storage backend on top of local file systems. This is a preferred choice for most distributed file systems today, because it allows them to benefit from the convenience and maturity of battle-tested code. Ceph's experience, however, shows that this comes at a high price. First, developing a zero-overhead transaction mechanism is challenging. Second, metadata performance at the local level can significantly affect performance at the distributed level. Third, supporting emerging storage hardware is painstakingly slow.

Ceph addressed these issues with BlueStore, a new backend designed to run directly on raw storage devices. In only two years since its inception, BlueStore outperformed previous established backends and is adopted by 70% of users in production. By running in user space and fully controlling the I/O stack, it has enabled space-efficient metadata and data checksums, fast overwrites of erasure-coded data, inline compression, decreased performance variability, and avoided a series of performance pitfalls of local file systems. Finally, it makes the adoption of backward-incompatible storage hardware possible, an important trait in a changing storage landscape that is learning to embrace hardware diversity.

CCS Concepts: • **Information systems → Distributed storage**; • **Software and its engineering → File systems management**; **Software performance**;

Additional Key Words and Phrases: Ceph, object storage, distributed file system, storage backend, file system

ACM Reference format:
Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2020. The Case for Custom Storage Backends in Distributed Storage Systems. *ACM Trans. Storage* 16, 2, Article 9 (April 2020), 31 pages.
https://doi.org/10.1145/3386362

1 INTRODUCTION

Distributed file systems operate on a cluster of machines, each assigned one or more roles such as cluster state monitor, metadata server, and storage server. Storage servers, which form the bulk

Michael Kuchnik is supported by an NDSEG Fellowship.
Q1
Authors' addresses: A. Aghayev, M. Kuchnik, G. R. Ganger, and G. Amvrosiadis, Carnegie Mellon University; S. Weil and M. Nelson, Red Hat, Inc.

1553-3077/2020/04-ART9 $15.00
https://doi.org/10.1145/3386362

of the machines in the cluster, receive I/O requests over the network and serve them from locally attached storage devices using *storage backend* software. Sitting in the I/O path, the storage backend plays a key role in the performance of the overall system.

Traditionally distributed file systems have used local file systems, such as ext4 or XFS, directly or through middleware, as the storage backend [31, 37, 40, 44, 79, 89, 98, 103, 106, 108]. This approach has delivered reasonable performance, precluding questions on the suitability of file systems as a distributed storage backend. Several reasons have contributed to the success of file systems as the storage backend. First, they allow delegating the hard problems of data persistence and block allocation to a well-tested and highly performant code. Second, they offer a familiar interface (POSIX) and abstractions (files, directories). Third, they enable the use of standard tools (`ls`, `find`) to explore disk contents.

Ceph [103] is a widely used, open-source distributed file system that followed this convention for a decade. Hard lessons that the Ceph team learned using several popular file systems led them to question the fitness of file systems as storage backends. This is not surprising in hindsight. Stonebraker, after building the INGRES database for a decade, noted that "operating systems offer all things to all people at much higher overhead" [95]. Similarly, exokernels demonstrated that customizing abstractions to applications results in significantly better performance [33, 53]. In addition to the performance penalty, adopting increasingly diverse storage hardware is becoming a challenge for local file systems, which were originally designed for a single storage medium.

The first contribution of this experience article is to *outline the main reasons behind Ceph's decision to develop BlueStore*, a new storage backend deployed directly on raw storage devices. First, it is hard to implement efficient transactions on top of existing file systems. A significant body of work aims to introduce transactions into file systems [42, 67, 69, 77, 82, 85, 92, 112], but none of these approaches have been adopted due to their high performance overhead, limited functionality, interface complexity, or implementation complexity. The experience of the Ceph team shows that the alternative options, such as leveraging the limited internal transaction mechanism of file systems, implementing Write-Ahead Logging in user space, or using a transactional key-value store, also deliver subpar performance.

Second, the local file system's metadata performance can significantly affect the performance of the distributed file system as a whole. More specifically, a key challenge that the Ceph team faced was enumerating directories with millions of entries fast, and the lack of ordering in the returned result. Both Btrfs and XFS-based backends suffered from this problem, and directory splitting operations meant to distribute the metadata load were found to clash with file system policies, crippling overall system performance.

At the same time, the rigidity of mature file systems prevents them from adopting emerging storage hardware that abandon the venerable block interface. The history of production file systems shows that on average they take a decade to mature [32, 59, 109, 110]. Once file systems mature, their maintainers tend to be conservative when it comes to making fundamental changes due to the consequences of mistakes. However, novel storage hardware aimed for data centers introduce backward-incompatible interfaces that require drastic changes. For example, to increase capacity, hard disk drive (HDD) vendors are moving to Shingled Magnetic Recording (SMR) technology [41, 66, 87] that works best with a backward-incompatible *zone* interface [49]. Similarly, to eliminate the long I/O tail latency in solid state drives (SSDs) caused by the Flash Translation Layer (FTL) [38, 56, 114], vendors are introducing Zoned Namespace (ZNS) SSDs that eliminate the FTL, again, exposing the zone interface [9, 27]. Cloud storage providers [61, 78, 116] and storage server vendors [18, 60] are already adapting their private software stacks to use the zoned devices. Distributed file systems, however, are stalled by delays in the adoption of zoned devices in local file systems.

In 2015, the Ceph project started designing and implementing BlueStore, a user space storage backend that stores data directly on raw storage devices, and metadata in a key-value store. By taking full control of the I/O path, BlueStore has been able to efficiently implement full data checksums, inline compression, and fast overwrites of erasure-coded data, while also improving performance on common customer workloads. In 2017, after just two years of development, BlueStore became the default production storage backend in Ceph. A 2018 survey among Ceph users shows that 70% use BlueStore in production with hundreds of petabytes in deployed capacity [64]. As a second contribution, this article *introduces the design of BlueStore, the challenges its design overcomes, and opportunities for future improvements*. Novelties of BlueStore include (1) storing low-level file system metadata, such as extent bitmaps, in a key-value store, thereby avoiding on-disk format changes and reducing implementation complexity; (2) optimizing clone operations and minimizing the overhead of the resulting extent reference-counting through careful interface design; (3) BlueFS—a user space file system that enables RocksDB to run faster on raw storage devices; and (4) a space allocator with a fixed 35 MiB memory usage per terabyte of disk space.

As a third contribution, to further demonstrate the advantage of clean-slate backend, this article *describes the first step taken toward adopting storage devices with the new zone interface in Ceph*. Specifically, we demonstrate how to adapt BlueFS, and thereby RocksDB, to run on high-capacity SMR drives without incurring the cost of a translation layer. This enables storing metadata in Ceph on zoned devices, and we leave developing techniques for storing data on zoned devices as a future work. Although metadata makes up a small portion of overall writes, using a small Ceph cluster we demonstrate that avoiding the translation layer overhead for metadata writes increases throughput by up to 141% and significantly reduces the write latency variance.

In addition to the above contributions, *we perform several experiments that evaluate the improvement of design changes from Ceph's previous production backend*, FileStore, to BlueStore. We experimentally measure the performance effect of issues such as the overhead of journaling file systems, double writes to the journal, inefficient directory splitting, and update-in-place mechanisms (as opposed to copy-on-write).

2 BACKGROUND

This section aims to highlight the role of distributed storage backends and the features that are essential for building an efficient distributed file system (Section 2.1). We provide a brief overview of Ceph's architecture (Section 2.2) and the evolution of Ceph's storage backend over the last decade (Section 2.3), introducing terms that will be used throughout the article.

2.1 Essentials of Distributed Storage Backends

Distributed file systems aggregate storage space from multiple physical machines into a single unified data store that offers high-bandwidth and parallel I/O, horizontal scalability, fault tolerance, and strong consistency. While distributed file systems may be designed differently and use unique terms to refer to the machines managing data placement on physical media, the storage backend is usually defined as the software module directly managing the storage device attached to physical machines. For example, Lustre's Object Storage Servers (OSSs) store data on Object Storage Targets [108] (OSTs), GlusterFS's Nodes store data on Bricks [79], and Ceph's Nodes store data on Object Storage Devices (OSDs) [103]. In these, and other systems, the storage backend is the software module that manages space on disks (OSTs, Bricks, OSDs) attached to physical machines (OSSs, Nodes).

Widely used distributed file systems such as Lustre [108], GlusterFS [79], OrangeFS [31], BeeGFS [98], XtreemFS [44], and (until recently) Ceph [103] rely on general local file systems,

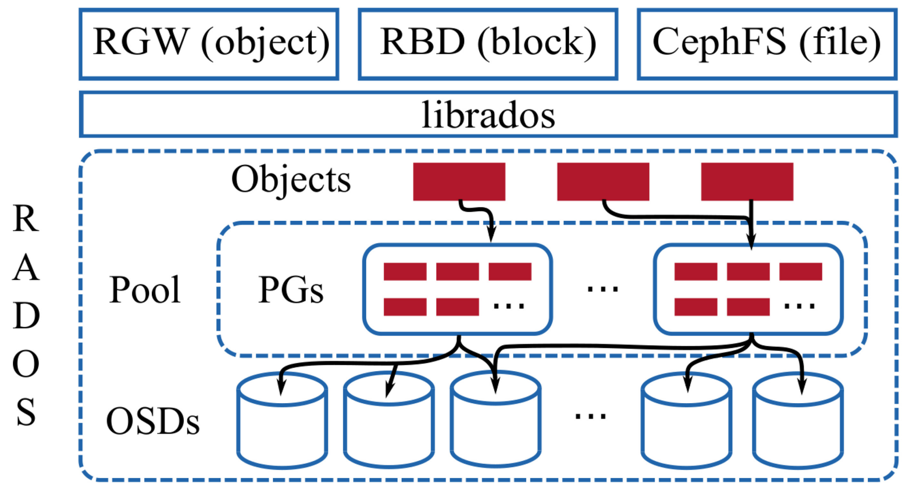

Fig. 1. High-level depiction of Ceph's architecture. A single pool with 3× replication is shown. Therefore, each placement group (PG) is replicated on three OSDs.

such as ext4 and XFS, to implement their storage backends. While different systems require different features from a storage backend, two of these features, (1) *efficient transactions* and (2) *fast metadata operations* appear to be common; another emerging requirement is (3) *support for novel, backward-incompatible storage hardware*.

Transaction support in the storage backend simplifies implementing strong consistency that many distributed file systems provide [44, 79, 103, 108]. A storage backend can seamlessly provide transactions if the backing file system already supports them [58, 82]. Yet, most file systems implement the POSIX standard, which lacks a transaction concept. Therefore, distributed file system developers typically resort to using inefficient or complex mechanisms, such as implementing a Write-Ahead Log (WAL) on top of a file system [79], or leveraging a file system's internal transaction mechanism [108].

Metadata management is another recurring pain point in distributed file systems [74]. Inability to efficiently enumerate large directory contents or handle small files at scale in local file systems can cripple performance for both centralized [106, 108] and distributed [79, 103] metadata management designs. To address this problem, distributed file system developers use metadata caching [79], deep directory hierarchies arranged by data hashes [103], custom databases [94], or patches to local file systems [12, 13, 119].

An emerging requirement for storage backends is support for novel storage hardware that operates using backward-incompatible interfaces. For example, SMR can boost HDD capacity by more than 25% and hardware vendors claim that by 2023, over half of data center HDDs will use SMR [88]. Another example is ZNS SSDs that eliminate FTL and do not suffer from uncontrollable garbage collection delays [9], allowing better tail-latency control. Both of these new classes of hardware storage present backward-incompatible interfaces that are challenging for local, block-based file systems to adopt.

2.2 Ceph Distributed Storage System Architecture

Figure 1 shows the high-level architecture of Ceph. At the core of Ceph is the Reliable Autonomic Distributed Object Store (RADOS) service [105]. RADOS scales to thousands of Object Storage

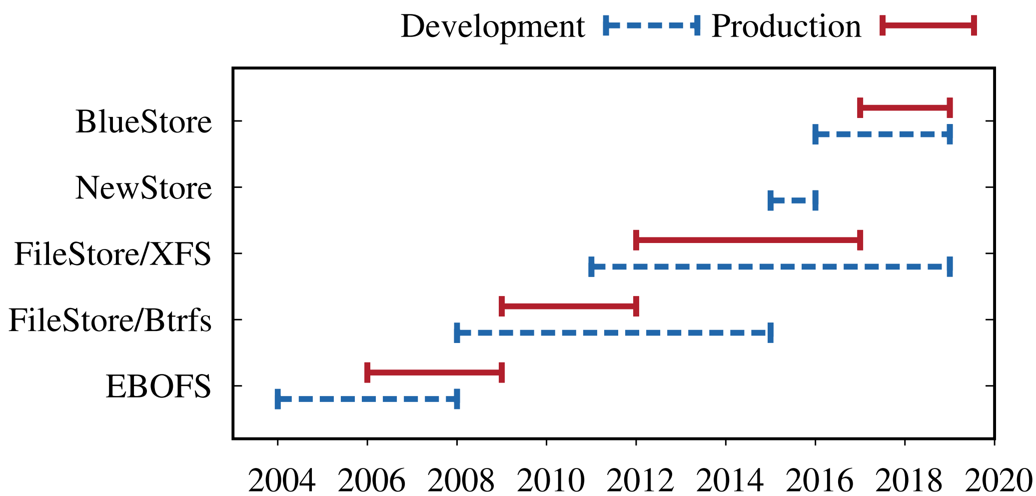

Fig. 2. Timeline of storage backend evolution in Ceph. For each backend, the period of development, and the period of being the default production backend is shown.

Devices (OSDs), providing self-healing, self-managing, replicated object storage with strong consistency. Ceph's `librados` library provides a transactional interface for manipulating objects and object collections in RADOS. Out of the box, Ceph provides three services implemented using `librados`: the RADOS Gateway (RGW), an object storage similar to Amazon S3 [5]; the RADOS Block Device (RBD), a virtual block device similar to Amazon EBS [4]; and CephFS, a distributed file system with POSIX semantics.

Objects in RADOS are stored in logical partitions called *pools*. Pools can be configured to provide redundancy for the contained objects either through replication or erasure coding. Within a pool, the objects are sharded among aggregation units called *placement groups* (PGs). Depending on the replication factor, PGs are mapped to multiple OSDs using CRUSH, a pseudo-random data distribution algorithm [104]. Clients also use CRUSH to determine the OSD that should contain a given object, obviating the need for a centralized metadata service. PGs and CRUSH form an indirection layer between clients and OSDs that allows the migration of objects between OSDs to adapt to cluster or workload changes.

In every node of a RADOS cluster, there is a separate *Ceph OSD* daemon per local storage device. Each OSD processes client I/O requests from `librados` clients and cooperates with peer OSDs to replicate or erasure code updates, migrate data, or recover from failures. Data are persisted to the local device via the internal *ObjectStore* interface, which provides abstractions for objects, object collections, a set of primitives to inspect data, and transactions to update data. A transaction combines an arbitrary number of primitives operating on objects and object collections into an atomic operation. In principle, each OSD may make use of a different backend implementation of the ObjectStore interface, although clusters tend to be uniform in practice.

2.3 Evolution of Ceph's Storage Backend

The first implementation of the ObjectStore interface was in fact a user space file system called Extent and B-Tree-based Object File System (EBOFS). In 2008, Btrfs was emerging with attractive features such as transactions, deduplication, checksums, and transparent compression, which were lacking in EBOFS. Therefore, as shown in Figure 2, EBOFS was replaced by FileStore, an ObjectStore implementation on top of Btrfs.

In FileStore, an object collection is mapped to a directory and object data are stored in a file. Initially, object attributes were stored in POSIX extended file attributes (`xattrs`), but were later moved to LevelDB when object attributes exceeded size or count limitations of `xattrs`. FileStore on Btrfs was the production backend for several years, throughout which Btrfs remained unstable and suffered from severe data and metadata fragmentation. In the meantime, the ObjectStore interface evolved significantly, making it impractical to switch back to EBOFS. Instead, FileStore was ported to run on top of XFS, ext4, and later ZFS. Of these, FileStore on XFS became the de facto backend, because it scaled better and had faster metadata performance [39].

While FileStore on XFS was stable, it still suffered from metadata fragmentation and did not exploit the full potential of the hardware. Lack of native transactions led to a user space WAL implementation that performed full data journaling and capped the speed of read-modify-write workloads, a typical Ceph workload, to the WAL's write speed. In addition, since XFS was not a copy-on-write file system, clone operations used heavily by snapshots were significantly slower.

NewStore was the first attempt at solving the metadata problems of file-system-based backends. Instead of using directories to represent object collections, NewStore stored object metadata in RocksDB, an ordered key-value store, while object data was kept in files. RocksDB was also used to implement the WAL, making read-modify-write workloads efficient due to a combined data and metadata log. Storing object data as files and running RocksDB on top of a journaling file system, however, introduced high consistency overhead. This led to the implementation of BlueStore, which used raw disks. The following section describes the challenges BlueStore aimed to resolve. A complete description of BlueStore is given in Section 4.

3 BUILDING STORAGE BACKENDS ON LOCAL FILE SYSTEMS IS HARD

This section describes the challenges faced by the Ceph team while trying to build a distributed storage backend on top of local file systems.

3.1 Challenge 1: Efficient Transactions

Transactions simplify application development by encapsulating a sequence of operations into a single atomic unit of work. Thus, a significant body of work aims to introduce transactions into file systems [42, 67, 69, 77, 82, 85, 92, 112]. None of these works have been adopted by production file systems, however, due to their high performance overhead, limited functionality, interface complexity, or implementation complexity.

Hence, there are three tangible options for providing transactions in a storage backend running on top of a file system: (1) hooking into a file system's internal (but limited) transaction mechanism, (2) implementing a WAL in user space, and (3) using a key-value database with transactions as a WAL. Next, we describe why each of these options results in significant performance or complexity overhead.

3.1.1 Leveraging File System Internal Transactions. Many file systems implement an in-kernel transaction framework that enables performing compound internal operations atomically [19, 24, 91, 99]. Since the purpose of this framework is to ensure internal file system consistency, its functionality is generally limited, and thus, unavailable to users. For example, a rollback mechanism is not available in file system transaction frameworks, because it is unnecessary for ensuring internal consistency of a file system.

Until recently, Btrfs was making its internal transaction mechanism available to users through a pair of system calls that atomically applied operations between them to the file system [24]. The first version of FileStore that ran on Btrfs relied on these system calls, and suffered from the lack of a rollback mechanism. More specifically, if a Ceph OSD ecountered a fatal event in the middle of a transaction, such as a software crash or a `KILL` signal, Btrfs would commit a partial transaction and leave the storage backend in an inconsistent state.

Solutions attempted by the Ceph and Btrfs teams included introducing a single system call for specifying the entire transaction [101] and implementing rollback through snapshots [100], both of which proved costly. Btrfs authors recently deprecated transaction system calls [15]. This outcome is similar to Microsoft's attempt to leverage NTFS's in-kernel transaction framework for providing an atomic file transaction API, which was deprecated due to its high barrier to entry [55].

These experiences strongly suggest that it is hard to leverage the internal transaction mechanism of a file system in a storage backend implemented in user space.

3.1.2 Implementing the WAL in User Space. An alternative to utilizing the file system's in-kernel transaction framework was to implement a logical WAL in user space. While this approach worked, it suffered from three major problems.

Slow Read-Modify-Write. Typical Ceph workloads perform many read-modify-write operations on objects, where preparing the next transaction requires reading the effect of the previous transaction. A user space WAL implementation, however, performs three steps for every transaction. First, the transaction is serialized and written to the log. Second, `fsync` is called to commit the transaction to disk. Third, the operations specified in the transaction are applied to the file system. The effect of a transaction cannot be read by upcoming transactions until the third step completes, which is dependent on the second step. As a result, every read-modify-write operation incurred the full latency of the WAL commit, preventing efficient pipelining.

Non-Idempotent Operations. In FileStore, objects are represented by files and collections are mapped to directories. With this data model, replaying a logical WAL after a crash is challenging due to non-idempotent operations. While the WAL is trimmed periodically, there is always a window of time when a committed transaction that is still in the WAL has already been applied to the file system. For example, consider a transaction consisting of three operations: `clone a→b; update a; update c`. If a crash happens after the second operation, then replaying the WAL corrupts object `b`. As another example, consider a transaction: `update b; rename b→c; rename a→b; update d`. If a crash happens after the third operation, then replaying the WAL corrupts object `a`, which is now named `b`, and then fails, because object `a` does not exist anymore.

FileStore on Btrfs solved this problem by periodically taking persistent snapshots of the file system and marking the WAL position at the time of snapshot. Then on recovery the latest snapshot was restored, and the WAL was replayed from the position marked at the time of the snapshot.

When FileStore abandoned Btrfs in favor of XFS (Section 2.3), the lack of efficient snapshots caused two problems. First, on XFS the `sync` system call is the only option for synchronizing file system state to storage. However, in typical deployments with multiple drives per node, `sync` is too expensive, because it synchronizes all file systems on all drives. This problem was resolved by adding the `syncfs` system call [102] to the Linux kernel, which synchronizes only a given file system.

The second problem was that with XFS, there is no option to restore a file system to a specific state after which the WAL can be replayed without worrying about non-idempotent operations. Guards (sequence numbers) were added to avoid replaying non-idempotent operations, however, verifying correctness of guards for complex operations was hard due to the large problem space. Tooling was written to generate random permutations of complex operation sequences, and it was combined with failure injection to semi-comprehensively verify that all failure cases were correctly handled. However, the FileStore code ended up fragile and hard-to-maintain.

Double Writes. The final problem with the WAL in FileStore is that data are written twice: first to the WAL and then to the file system, halving the disk bandwidth. This is a known problem that leads most file systems to only log metadata changes, allowing data loss after a crash. It is possible to avoid the penalty of double writes for new data, by first writing it to disk and then logging only the respective metadata. However, FileStore's approach of using the state of the file system to infer the namespace of objects and their states makes this method hard to use due to corner cases, such as partially written files. While FileStore's approach turned out to be problematic, it was chosen for a technical reason: The alternative required implementing an in-memory cache for data and metadata to any updates waiting on the WAL, despite the kernel having a page and inode cache of its own.

3.1.3 Using a Key-Value Store as the WAL. With NewStore, the metadata was stored in RocksDB, an ordered key-value store, while the object data were still represented as files in a file system. Hence, metadata operations could be performed atomically; data overwrites, however, were logged into RocksDB and executed later. We first describe how this design addresses the three problems of a logical WAL, and then show that it introduces high consistency overhead that stems from running atop a journaling file system.

First, slow read-modify-write operations are avoided, because the key-value interface allows reading the new state of an object without waiting for the transaction to commit.

Second, the problem of non-idempotent operation replay is avoided, because the read side of such operations is resolved at the time when the transaction is prepared. For example, for `clone a→b`, if object `a` is small, then it is copied and inserted into the transaction; if object `a` is large, then a copy-on-write mechanism is used, which changes both `a` and `b` to point to the same data and marks the data read-only.

Finally, the problem of double writes is avoided for new objects, because the object namespace is now decoupled from the file system state. Therefore, data for a new object are first written to the file system and then a reference to it is atomically added to the database.

Despite these favorable properties, the combination of RocksDB and a journaling file system introduces high consistency overhead, similar to the *journaling of journal* problem [51, 86]. Creating an object in NewStore entails two steps: (1) writing to a file and calling `fsync` and (2) writing the object metadata to RocksDB synchronously [47], which also calls `fsync`. Ideally, the `fsync` in each step should issue one expensive `FLUSH CACHE` command [111] to disk. With a journaling file system, however, each `fsync` issues two flush commands: after writing the data and after committing the corresponding metadata changes to the file system journal. Hence, creating an object in NewStore results in four expensive flush commands to disk.

We demonstrate the overhead of journaling using a benchmark that emulates a storage backend creating many objects. The benchmark has a loop where each iteration first writes 0.5 MiB of data and then inserts a 500-byte metadata to RocksDB. We run the benchmark on two setups. The first setup emulates NewStore, issuing four flush operations for every object creation: Data are written as a file to XFS, and the metadata are inserted to stock RocksDB running on XFS. The second setup emulates object creation on raw disk, which issues two flush operations for every object creation: Data are written to the raw disk and the metadata are inserted to a modified RocksDB that runs on a raw disk with a preallocated pool of WAL files.

Figure 3 shows that the object creation throughput is 80% higher on raw disk than on XFS when running on a HDD and 70% when running on an NVMe SSD.

3.2 Challenge 2: Fast Metadata Operations

Inefficiency of metadata operations in local file systems is a source of constant struggle for distributed file systems [74, 76, 119]. One of the key metadata challenges in Ceph with the FileStore

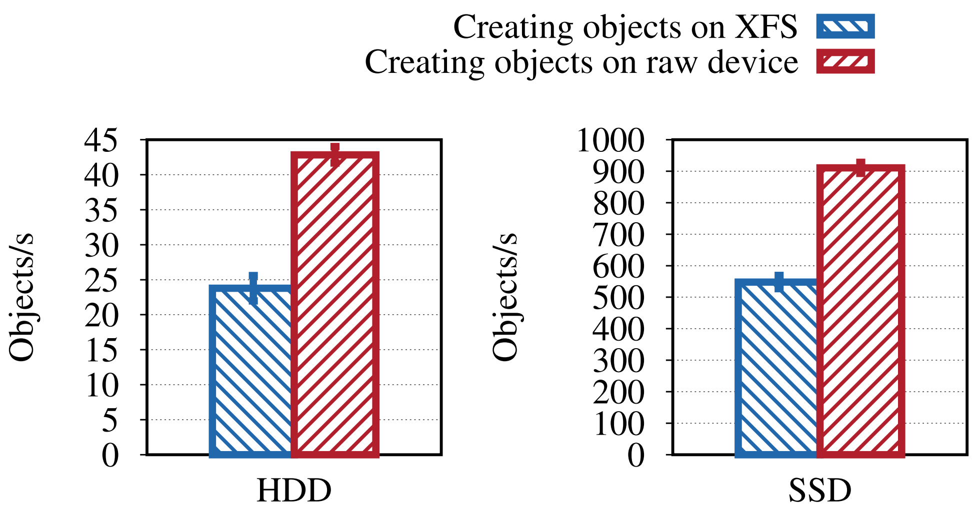

Fig. 3. The overhead of running an object store workload on a journaling file system. Object creation throughput is 80% higher on a raw HDD (4-TB Seagate ST4000NM0023) and 70% higher on a raw NVMe SSD (400-GB Intel P3600).

backend stems from the slow directory enumeration (`readdir`) operations on large directories, and the lack of ordering in the returned result [90].

Objects in RADOS are mapped to a PG based on a hash of their name, and enumerated by hash order. Enumeration is necessary for operations like scrubbing [83], recovery, or for serving `librados` calls that list objects. For objects with long names—as is often the case with RGW—FileStore works around the file name length limitation in local file systems using extended attributes, which may require a `stat` call to determine the object name. FileStore follows a commonly adopted solution to the slow enumeration problem: A directory hierarchy with large fan-out is created, objects are distributed among directories, and then selected directories' contents are sorted after being read.

To sort them quickly and to limit the overhead of potential `stat` calls, directories are kept small (a few hundred entries) by splitting them when the number of entries in them grows. This is a costly process at scale, for two primary reasons. First, processing millions of inodes at once reduces the effectiveness of dentry cache, resulting in many small I/Os to disk. And second, XFS places subdirectories in different *allocation groups* [48] to ensure there is space for future directory entries to be located close together [65]; therefore, as the number of objects grows, directory contents spread out, and split operations take longer due to seeks. As a result, when all Ceph OSDs start splitting in unison the performance suffers. This is a well-known problem that has been affecting many Ceph users over the years [16, 28, 93].

To demonstrate this effect, we configure a 16-node Ceph cluster (Section 7) with roughly half the recommended number of PGs to increase load per PG and accelerate splitting, and insert millions of 4 KiB objects with queue depth of 128 at the RADOS layer (Section 2.2). Figure 4 shows the effect of the splitting on FileStore for an all-SSD cluster. While the first split is not noticeable in the graph, the second split causes a precipitous drop that kills the throughput for 7 minutes on an all-SSD and 120 minutes on an all-HDD cluster (not shown), during which a large and deep directory hierarchy with millions of entries is scanned and even a deeper hierarchy is created. The recovery takes an order of magnitude longer on an all-HDD cluster due to high cost of seeks.

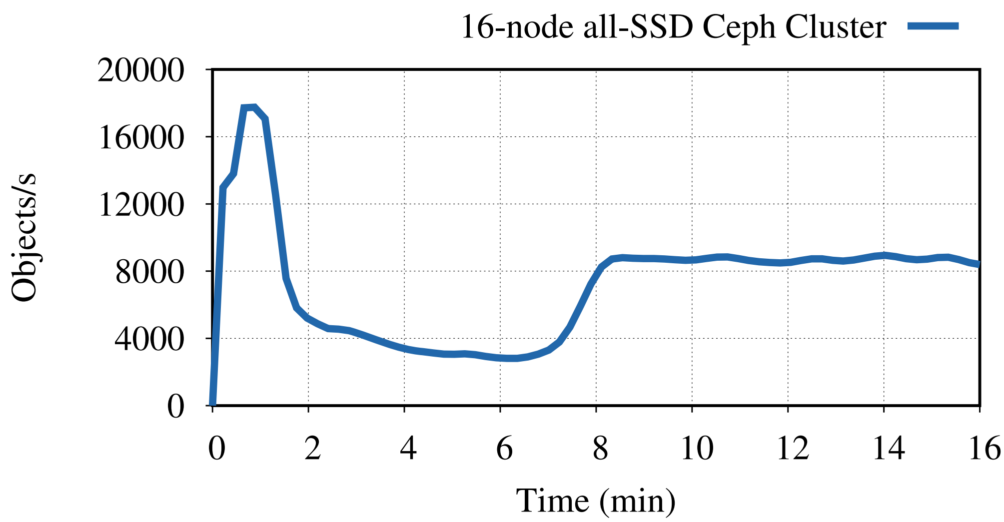

Fig. 4. The effect of directory splitting on throughput with FileStore backend. The workload inserts 4-KiB objects using 128 parallel threads at the RADOS layer to a 16-node Ceph cluster (setup explained in Section 7). Directory splitting brings down the throughput for 7 minutes on an all-SSD cluster. Once the splitting is complete, the throughput recovers but does not return to peak, due to combination of deeper nesting of files, increased size of the underlying file system, and an imperfect implementation of the directory hashing code in FileStore.

3.3 Challenge 3: Support for New Storage Hardware

The changing storage hardware landscape presents a new challenge for distributed file systems that depend on local file systems. To increase capacity, hard disk drive vendors are shifting to SMR that works best when using a backward-incompatible interface. While the vendors have produced *drive-managed* SMR (DM-SMR) drives that are backward compatible, these drives have unpredictable performance [1]. For leveraging the extra capacity and achieving predictable performance at the same time, *host-managed* SMR (HM-SMR) drives with a backward-incompatible zone interface should be used [49]. The zone interface, however, manages the disk as a sequence of 256 MiB regions that must be written sequentially, encouraging a log-structured, copy-on-write design [81]. This design is in direct opposition to in-place overwrite design followed by most mature file systems.

Data center SSDs are going through a similar change. OpenChannel SSDs eliminate the FTL, leaving the management of raw flash to the host. Lacking an official standard, several vendors have introduced different methods of interfacing OpenChannel SSDs, resulting in fragmented implementations [11, 22, 36]. To prevent this, major vendors have joined forces to introduce a new NVMe standard called Zoned Namespaces (ZNS) that defines an interface for managing SSDs without an FTL [10]. Eliminating the FTL results in many advantages, such as reducing the write amplification, improving latency outliers and throughput, reducing overprovisioning by an order of magnitude, and cutting the cost by reducing DRAM—the highest costing component in SSD after the NAND flash.

Both of these technologies—host-managed SMR drives and ZNS SSDs—are becoming increasingly important for distributed file systems, yet, both have a backward incompatible zone interface that requires radical changes to local file systems [9, 27]. It is not surprising that attempts to modify production file systems, such as XFS and ext4, to work with the zone interface have so far been unsuccessful [20, 73], primarily because these are overwrite file systems, whereas the zone interface requires a copy-on-write approach to data management.

3.4 Other Challenges

Many public and private clouds rely on distributed storage systems like Ceph for providing storage services [72]. Without the complete control of the I/O stack, it is hard for distributed file systems to enforce storage latency SLOs. One cause of high-variance request latencies in file-system-based storage backends is the OS page cache. To improve user experience, most OSs implement the page cache using write-back policy, in which a write operation completes once the data are buffered in memory and the corresponding pages are marked as *dirty*. On a system with little I/O activity, the dirty pages are written back to disk at regular intervals, synchronizing the on-disk and in-memory copies of data. On a busy system, however, the write-back behavior is governed by a complex set of policies that can trigger writes at arbitrary times [8, 25, 113].

Hence, while the write-back policy results in a responsive system for users with lightly loaded systems, it complicates achieving predictable latency on busy storage backends. Even with a periodic use of `fsync`, FileStore has been unable to bound the amount of deferred inode metadata write-back, leading to inconsistent performance.

Another challenge for file-system-based backends is implementing operations that work better with copy-on-write support, such as snapshots. If the backing file system is copy-on-write, then these operations can be implemented efficiently. However, even if the copy-on-write is supported, a file system may have other drawbacks, like fragmentation in FileStore on Btrfs (Section 2.3). If the backing file system is not copy-on-write, then these operations require performing expensive full copies of objects, which makes snapshots and overwriting of erasure-coded data prohibitively expensive in FileStore (Section 5.2).

4 BLUESTORE: A CLEAN-SLATE APPROACH

BlueStore is a storage backend designed from scratch to solve the challenges (Section 3) faced by backends using local file systems. Some of the main goals of BlueStore were as follows:

(1) Fast metadata operations (Section 4.1)
(2) No consistency overhead for object writes (Section 4.1)
(3) Copy-on-write clone operation (Section 4.2)
(4) No journaling double-writes (Section 4.2)
(5) Optimized I/O patterns for HDD and SSD (Section 4.2)

BlueStore achieved all of these goals within just two years and became the default storage backend in Ceph. Two factors played a key role in why BlueStore matured so quickly compared to general-purpose POSIX file systems that take a decade to mature [32, 59, 109, 110]. First, BlueStore implements a small, special-purpose interface, and not a complete POSIX I/O specification. Second, BlueStore is implemented in user space, which allows it to leverage well-tested and high-performance third-party libraries. Finally, BlueStore's control of the I/O stack enables additional features whose discussion we defer to Section 5.

The high-level architecture of BlueStore is shown in Figure 5. BlueStore runs directly on raw disks. A space allocator within BlueStore determines the location of new data, which is asynchronously written to disk using direct I/O. Internal metadata and user object metadata are stored in RocksDB, which runs on BlueFS, a minimal user space file system tailored to RocksDB. The BlueStore space allocator and BlueFS share the disk and periodically communicate to balance free space. The remainder of this section describes metadata and data management in BlueStore.

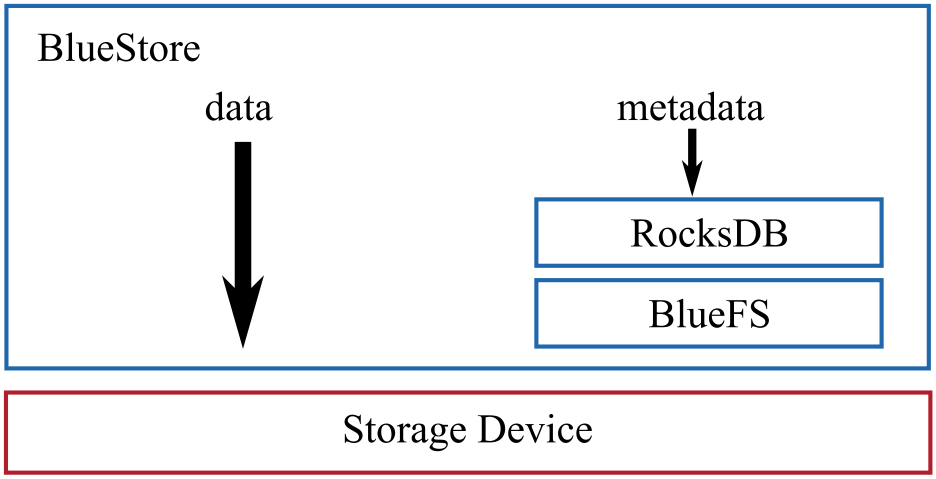

Fig. 5. The high-level architecture of BlueStore. Data are written to the raw storage device using direct I/O. Metadata are written to RocksDB running on top of BlueFS. Although BlueFS logically is a separate component, it is a user space library file system designed for RocksDB that compiles and links with RocksDB and reads and writes the raw storage device.

4.1 BlueFS and RocksDB

BlueStore achieves its first goal, *fast metadata operations*, by storing metadata in RocksDB. BlueStore achieves its second goal of *no consistency overhead* with two changes. First, it writes data directly to raw disk, resulting in one cache flush for data write. Second, it changes RocksDB to reuse WAL files as a circular buffer, resulting in one cache flush for metadata write—a feature that was upstreamed to the mainline RocksDB.

RocksDB itself runs on BlueFS, a minimal file system designed specifically for RocksDB that runs on a raw storage device. RocksDB abstracts out its requirements from the underlying file system in the *Env* interface. BlueFS is an implementation of this interface in the form of a user space, extent-based, and journaling file system. It implements basic system calls required by RocksDB, such as `open`, `mkdir`, and `pwrite`. A possible on-disk layout of BlueFS is shown in Figure 6. BlueFS maintains an inode for each file that includes the list of extents allocated to the file. The superblock is stored at a fixed offset and contains an inode for the journal. The journal has the only copy of all file system metadata, which is loaded into memory at mount time. On every metadata operation, such as directory creation, file creation, and extent allocation, the journal and in-memory metadata are updated. The journal is not stored at a fixed location; its extents are interleaved with other file extents. The journal is compacted and written to a new location when it reaches a preconfigured size, and the new location is recorded in the superblock. These design decisions work, because large files and periodic compactions limit the volume of metadata at any point in time.

Metadata Organization. BlueStore keeps multiple namespaces in RocksDB, each storing a different type of metadata. For example, object information is stored in the *O* namespace (that is, RocksDB keys start with *O* and their values represent object metadata), block allocation metadata are stored in the *B* namespace, and collection metadata are stored in the *C* namespace. Each collection maps to a PG and represents a shard of a pool's namespace. The collection name includes the pool identifier and a prefix shared by the collection's object names. For example, a key-value pair `C12.e4-6` identifies a collection in pool 12 with objects that have hash values starting with the 6 significant bits of `e4`. Hence, the object `O12.e532` is a member, whereas the object `O12.e832` is not.

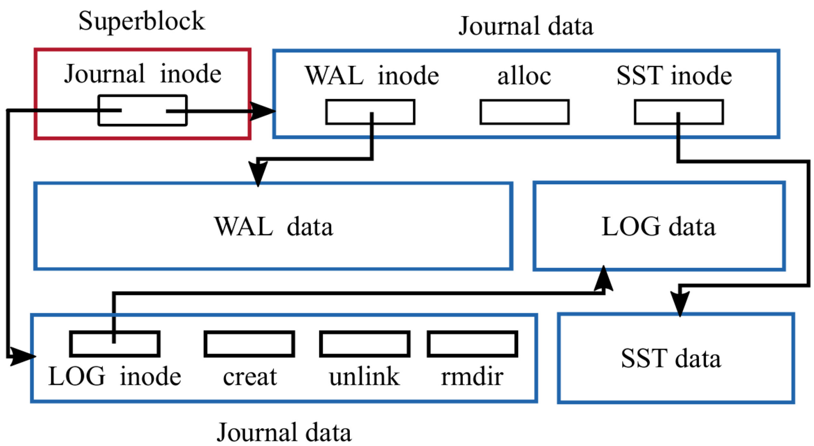

Fig. 6. A possible on-disk data layout of BlueFS. The metadata in BlueFS lives only in the journal. The journal does not have a fixed location—its extents are interleaved with file data. The WAL, LOG, and SST files are write-ahead log file, debug log file, and a sorted-string table files, respectively, generated by RocksDB.

Such organization of metadata allows a collection of millions of objects to be split into multiple collections merely by changing the number of significant bits. This *collection splitting* operation is necessary to rebalance data across OSDs when, for example, a new OSD is added to the cluster to increase the aggregate capacity or an existing OSD is removed from the cluster due to a malfunction. With FileStore, collection splitting, which is different than directory splitting (Section 3.2), was an expensive operation that was done by renaming directories.

4.2 Data Path and Space Allocation

BlueStore is a copy-on-write backend. For incoming writes larger than a *minimum allocation size* (64 KiB for HDDs, 16 KiB for SSDs) the data are written to a newly allocated extent. Once the data are persisted, the corresponding metadata are inserted to RocksDB. This allows BlueStore to provide an *efficient clone* operation. A clone operation simply increments the reference count of dependent extents, and writes are directed to new extents. It also allows BlueStore to *avoid journal double-writes* for object writes and partial overwrites that are larger than the minimum allocation size.

For writes smaller than the minimum allocation size, both data and metadata are first inserted to RocksDB as promises of future I/O, and then asynchronously written to disk after the transaction commits. This deferred write mechanism has two purposes. First, it batches small writes to increase efficiency, because new data writes require two I/O operations whereas an insert to RocksDB requires one. Second, it *optimizes I/O based on the device type;* 64-KiB (or smaller) overwrites of a large object on an HDD are performed asynchronously in place to avoid seeks during reads, whereas in-place overwrites only happen for I/O sizes less than 16 KiB on SSDs.

Space Allocation. BlueStore allocates space using two modules: the FreeList manager and the Allocator. The FreeList manager is responsible for a *persistent* representation of the parts of the disk currently in use. Like all metadata in BlueStore, this free list is also stored in RocksDB. The first implementation of the FreeList manager represented in-use regions as key-value pairs with offset and length. The disadvantage of this approach was that the transactions had to be serialized: the old key had to be deleted first before inserting a new key to avoid an inconsistent free list. The second implementation is bitmap-based. Allocation and deallocation operations use

RocksDB's merge operator to flip bits corresponding to the affected blocks, eliminating the ordering constraint. The merge operator in RocksDB performs a deferred atomic read-modify-write operation that does not change the semantics and avoids the cost of point queries [46].

The Allocator is responsible for allocating space for the new data. It keeps a copy of the free list in memory and informs the FreeList manager as allocations are made. The first implementation of Allocator was extent-based, dividing the free extents into power-of-two-sized bins. This design was susceptible to fragmentation as disk usage increased. The second implementation uses a hierarchy of indexes layered on top of a single-bit-per-block representation to track whole regions of blocks. Large and small extents can be efficiently found by querying the higher and lower indexes, respectively. This implementation has a fixed memory usage of 35 MiB per terabyte of capacity.

Cache. Since BlueStore is implemented in user space and accesses the disk using direct I/O, it cannot leverage the OS page cache. As a result, BlueStore implements its own write-through cache in user space, using the scan resistant 2Q algorithm [52]. The cache implementation is sharded for parallelism. It uses an identical sharding scheme to Ceph OSDs, which shard requests to collections across multiple cores. This avoids false sharing, so that the same CPU context processing a given client request touches the corresponding 2Q data structures.

5 FEATURES ENABLED BY BLUESTORE

In this section, we describe new features implemented in BlueStore. These features were previously lacking, because implementing them efficiently requires full control of the I/O stack.

5.1 Space-Efficient Checksums

Ceph scrubs metadata every day and data every week. Even with scrubbing, however, if the data are inconsistent across replicas, then it is hard to be sure which copy is corrupt. Therefore, checksums are indispensable for distributed storage systems that regularly deal with petabytes of data, where bit flips are almost certain to occur.

Most local file systems do not support checksums. When they do, like Btrfs, the checksum is computed over 4-KiB blocks to make block overwrites possible. For 10 TiB of data, storing 32-bit checksums of 4-KiB blocks results in 10 GiB of checksum metadata, which makes it difficult to cache checksums in memory for fast verification.

However, most of the data stored in distributed file systems is read-only and can be checksummed at a larger granularity. BlueStore computes a checksum for every write and verifies the checksum on every read. While multiple checksum algorithms are supported, `crc32c` is used by default, because it is well-optimized on both x86 and ARM architectures, and it is sufficient for detecting random bit errors. With full control of the I/O stack, BlueStore can choose the checksum block size based on the I/O hints. For example, if the hints indicate that writes are from the S3-compatible RGW service, then the objects are read-only and the checksum can be computed over 128 KiB blocks, and if the hints indicate that objects are to be compressed, then a checksum can be computed after the compression, significantly reducing the total size of checksum metadata.

5.2 Overwrite of Erasure-Coded Data

Ceph has supported erasure-coded (EC) pools (Section 2.2) through the FileStore backend since 2014. However, until BlueStore, EC pools only supported object appends and deletions—overwrites were slow enough to make the system unusable. As a result, the use of EC pools were limited to RGW; for RBD and CephFS only replicated pools were used.

To avoid the "RAID write hole" problem [97], where crashing during a multi-step data update can leave the system in an inconsistent state, Ceph performs overwrites in EC pools using two-phase commit. First, all OSDs that store a chunk of the EC object make a copy of the chunk so that they can roll back in case of failure. After all of the OSDs receive the new content and overwrite their chunks, the old copies are discarded. With FileStore on XFS, the first phase is expensive, because each OSD performs a physical copy of its chunk. BlueStore, however, makes overwrites practical, because its copy-on-write mechanism avoids full physical copies.

5.3 Transparent Compression

Transparent compression is crucial for scale-out distributed file systems, because 3× replication increases storage costs [35, 43]. BlueStore implements transparent compression where written data are automatically compressed before being stored.

Getting the full benefit of compression requires compressing over large 128 KiB chunks, and compression works well when objects are written in their entirety. For partial overwrites of a compressed object, BlueStore places the new data in a separate location and updates metadata to point to it. When the compressed object gets too fragmented due to multiple overwrites, BlueStore compacts the object by reading and rewriting. In practice, however, BlueStore uses hints and simple heuristics to compress only those objects that are unlikely to experience many overwrites.

6 TOWARD SUPPORTING HM-SMR DRIVES IN BLUESTORE

Despite multiple attempts [20, 73], local file systems are unable to leverage the capacity benefits of SMR drives due to their backward-incompatible interface, and it is unlikely that they will ever do so efficiently [30, 32]. Supporting these denser drives, however, is important for scale-out distributed file systems, because it lowers storage costs [62]. Unconstrained by the block-based designs of local file systems, BlueStore has the freedom of exploring novel interfaces and data layouts. In this section we describe the first step we took toward adopting HM-SMR drives (Section 3.3) with the new zone interface.

Figure 5 shows that the data in BlueStore is written to raw disk, while the metadata are written to RocksDB, a widely used key-value store based on the Log-Structured Merge-Tree (LSM-Tree) data structure [71]. Today, the LSM-Tree is the predominant method for implementing persistent key-value stores, and it is at the core of many databases and scale-out storage systems [17, 34, 57], including Ceph. Since enabling LSM-Trees to run zoned drives is a more general problem with a potentially large impact, we chose first to adapt RocksDB, an LSM-Tree instance, to run on zoned drives.

6.1 RocksDB Primer

Every key-value inserted to RocksDB is first individually written to a Write-Ahead Log (WAL) file using the `pwrite` system call, and then buffered in an in-memory data structure called *memtable*. By default, RocksDB performs *asynchronous inserts*: `pwrite` returns as soon as the data are buffered in the OS page cache and the actual transfer of data from the page cache to storage is done later by the kernel writeback threads. Hence, a machine crash may result in loss of data for an insert that was acknowledged. For applications that require the durability and consistency of transactional writes RocksDB also supports *synchronous inserts* that do not return until data are persisted on storage.

When the memtable reaches a preconfigured size, a new one is created. In batches, memtables are merge-sorted and written to storage as a Sorted String Table (SST) file. SSTs in RocksDB are organized into multiple levels, as shown in Figure 7. The aggregate size of each level L_i is a multiple of L_{i-1}, starting with a fixed size at L_1. At first data are written at level L_0. Once the number of L_0 SSTs reach a threshold, the compaction process selects all of L_0 SSTs, reads them into memory, merge-sorts them, and writes them out as new L_1 SSTs. For higher levels, compactions are triggered

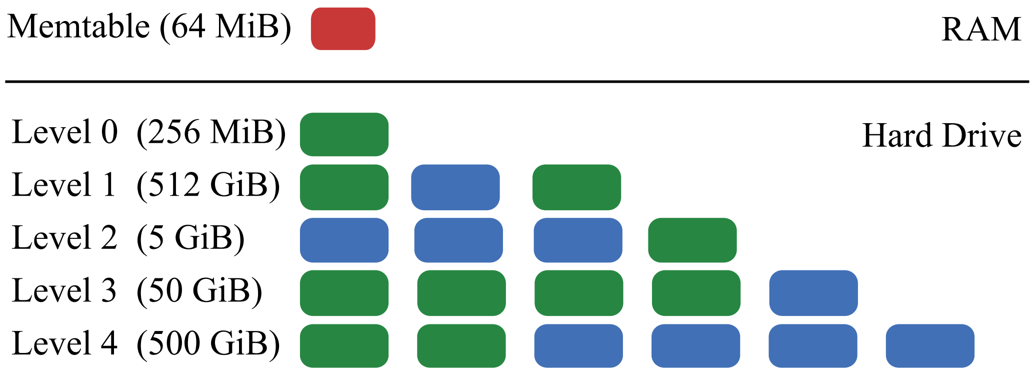

Fig. 7. Data organization in RocksDB. Green squares represent Sorted String Tables (SSTs). Blue squares represent SSTs selected for two different concurrent compactions.

when the aggregate size of the level exceeds a threshold, in which case one SST from the lower level and multiple SSTs from a higher level are compacted, as shown in Figure 7. If memtable flushes or compactions cannot keep up with the rate of inserts, then RocksDB stalls inserts to avoid filling storage and to prevent lookups from slowing down.

6.2 SMR Primer

SMR increases drive capacities by partially overlapping adjacent magnetic tracks, leaving narrower tracks for the drive read heads to still be able to access the data, similar to roof shingles. While this technique does not affect purely sequential write workloads, random (over)writes are challenging as they would corrupt the data of adjacent tracks. To mitigate this, SMR drives are partitioned in zones. Within each zone (e.g., 256 MiB), tracks are shingled and acceptable operations are limited to sequential writes or zone erases.

DM-SMR drives use a Shingled Translation Layer (STL) to present a block interface to the host instead of zones. Random writes are buffered in a *persistent cache* of reserved tracks, and are later written to their final locations by overwriting existing zones [1] during *cleaning*. Large sequential writes are directly written to their final locations [2].

HM-SMR drives expose zones through a novel interface [49]. The first few hundred zones are conventional tracks that can be written randomly, while the rest are shingled, i.e., strictly sequential. For each sequential zone the drive keeps a *write pointer* that starts at the beginning of the zone and is updated after each append. A write to a location other than the write pointer will fail, but the write pointer can be reset to the beginning of a zone.

6.3 Challenges and Solutions of Running RocksDB on HM-SMR Drives

Running RocksDB or similar LSM-Trees on HM-SMR drive leads to multiple challenges. Below we describe these challenges and our solutions to them.

Garbage collection: Simply placing SST files generated by RocksDB or other LSM-Trees into the zones of an HM-SMR drive leads to the garbage collection problem of LFS [81], because the default SST sizes are much smaller than the zones of the drive. After multiple compactions zones will contain fragmented free space from SSTs that have been merged to a new SST. Reclaiming the space occupied by dead SSTs requires migrating live SSTs to another zone. Recent work proposes a new data format and compaction algorithm to avoid HM-SMR garbage collection for an LSM-Tree with 4 MiB SSTs [115].

Our solution to this problem is to align the SST and zone sizes. This way, cleaning can be eliminated, because at the end of compaction, the space from "dead" SSTs can be reclaimed by merely resetting the zone's write pointer. There are other compelling reasons for increasing SST size, such as enabling disks to do streaming reads with fewer seeks, reducing expensive sync operations, and reducing the number of open file handles. Applying this simple idea to production-grade key-value stores and real HM-SMR drives, however, involves other challenge described next.

Reordered Writes: Like most LSM-Tree implementations, RocksDB uses buffered I/O when writing compacted SSTs. This improves performance significantly, because compacted SSTs can be kept in the OS page cache. As a result, lookups are served from memory and files are read from memory during compaction, reserving the disk bandwidth for memtable flushes and thereby increasing transaction throughput.

Using buffered I/O, however, does not guarantee write ordering that is essential for HM-SMR drives. Page writeback can happen from different contexts at the same time, and the pages picked up by each context will not be necessarily zone-aligned. Furthermore, there are no write-alignment constraints with buffered writes, so an application may write parts of a page across different operations. In this case, however, the same last page cannot be overwritten to add the remaining data when the sequential write stream resumes.

This requires the use of direct I/O with HM-SMR drives, which gives up the aforementioned OS page cache advantages. To mitigate performance issues, we implement a user-space file cache within BlueFS so that reads are not always served from disk during compaction.

Synchronous Writes to the Log: The `libzbc` [107] library is the de-facto way of interacting with HM-SMR drives, and used in LevelDB-derived key-value stores designed for HM-SMR [63, 115]. As part of `libzbc`, the `zbc_pwrite` call is provided for positional writes to the device, which has similar semantics to the `pwrite` system call. Even though `pwrite` is a synchronous call, since it is usually used with buffered I/O, it is effectively made asynchronous, because it returns once the data are copied into OS memory (Section 6.1). The `zbc_pwrite` call, however, waits for the drive to acknowledge the write, since the HM-SMR drive can only be used with direct I/O. This works well when data are buffered in memory and written in large chunks, which is the case for memtable flushes and SSTs writes during compaction. Writes to the Write-Ahead Log (WAL), however, happen after every key-value insertion. As a result, with direct I/O every insertion must be acknowledged by the disk, limiting the throughput of the key-value store to that of small synchronous writes to drive.

To remedy this bottleneck we use the `libaio`, an in-kernel asynchronous I/O framework. This approach works as long as the asynchronous I/O operations are issued in order and the right I/O scheduler is used.

Misaligned Writes to the Log: Random and misaligned writes violate the zone interface, and therefore cannot be used with HM-SMR drives (Section 6.2). In RocksDB, there are three sources of such writes. First, RocksDB produces a handful of small files that receive negligible amounts of non-sequential I/O. Placing those in a conventional zone solves the problem. Second, the last block of SST files suffers from overwrites, which we found were due to a bug in RocksDB, which we reported and has since been fixed by the RocksDB team [29]. Third, and more surprisingly, the last block of the WAL tends to be overwritten if the previous append operation was not aligned, making it unsuitable for placing on a sequential zone. Deployments of RocksDB typically shard key space and run multiple instances where WALs are close to the data, thereby avoiding long seeks. Placing the WAL in a conventional zone, however, would result in expensive seeks—especially for synchronous inserts—because conventional zones are typically concentrated in one part of the drive. Furthermore, this would waste conventional track space on an append-only file.

We introduce a special format for the WAL that allows it to be written sequentially so that it can be placed on a sequential zone. We modify BlueFS to wrap every write to the WAL in a record that keeps the length of the write inline and always pads out the record to a 4 KiB boundary. With this change, the read code for the WAL is no longer a direct mapping from an extent start to offset, and record lengths have to be read to determine the actual content. This, however, is not a problem, because the WAL is only read sequentially and only during crash recovery, and the space overhead is negligible (less than 1% in our benchmarks), because such unaligned writes are rare.

7 EVALUATION

This section compares the performance of a Ceph cluster using FileStore, a backend built on a local file system, and BlueStore, a backend using the storage device directly. First, we compare the throughput of object writes to the RADOS distributed object storage (Section 7.1). Second, we compare the end-to-end throughput of random writes, sequential writes, and sequential reads to RBD, the Ceph virtual block device built on RADOS (Section 7.2). Third, we compare the throughput of random writes to an RBD device allocated on an erasure-coded pool (Section 7.3). Finally, we demonstrate some early results from our ongoing work of adapting BlueStore to work with HM-SMR hard drives.

We run all experiments, except HM-SMR experiments, on a 16-node Ceph cluster connected with a Cisco Nexus 3264-Q 64-port QSFP+ 40GbE switch; for HM-SMR experiments we use a 3-node Ceph cluster due to limited HM-SMR samples. Each node has a 16-core Intel E5-2698Bv3 Xeon 2-GHz CPU, 64-GiB RAM, 400GB Intel P3600 NVMe SSD, 4TB 7200RPM Seagate ST4000NM0023 HDD, and a Mellanox MCX314A-BCCT 40-GbE NIC. All nodes run Linux kernel 4.15 on Ubuntu 18.04, and the Luminous release (v12.2.11) of Ceph. We use the default Ceph configuration parameters and for each experiment we set up Ceph to use only HDDs or SSDs.

7.1 Bare RADOS Benchmarks

We start by comparing the performance of object writes to RADOS when using the FileStore and BlueStore backends. We focus on write performance improvements, because most BlueStore optimizations affect writes.

Figure 8 shows the throughput for different object sizes written with a queue depth of 128. At the steady state, the throughput on BlueStore is 50–100% greater than FileStore. The throughput improvement on BlueStore stems from avoiding double writes (Section 3.1.2) and consistency overhead (Section 3.1.3).

Figure 9 shows the 95th and above percentile latencies of object writes to RADOS. BlueStore has an order of magnitude lower tail latency than FileStore. In addition, with BlueStore the tail latency increases with the object size, as expected, whereas with FileStore even small-sized object writes may have high tail latency, stemming from the lack of control over writes (Section 3.4).

The read performance on BlueStore (not shown) is similar or better than on FileStore for I/O sizes larger than 128 KiB; for smaller I/O sizes FileStore is better because of the kernel read-ahead [6]. BlueStore does not implement read-ahead on purpose. It is expected that the applications implemented on top of RADOS will perform their own read-ahead.

BlueStore eliminates the directory splitting effect of FileStore by storing metadata in an ordered key-value store. To demonstrate this, we repeat the experiment that showed the splitting problem in FileStore (Section 3.2) on an identically configured Ceph cluster using a BlueStore backend. Figure 10 shows that the throughput on BlueStore does not suffer the precipitous drop, and in the steady state it is 2× higher than FileStore throughput on SSD (and 3× higher than FileStore throughput on HDD—not shown). Still, the throughput on BlueStore drops significantly before reaching a steady state due to RocksDB compaction whose cost grows with the object corpus.

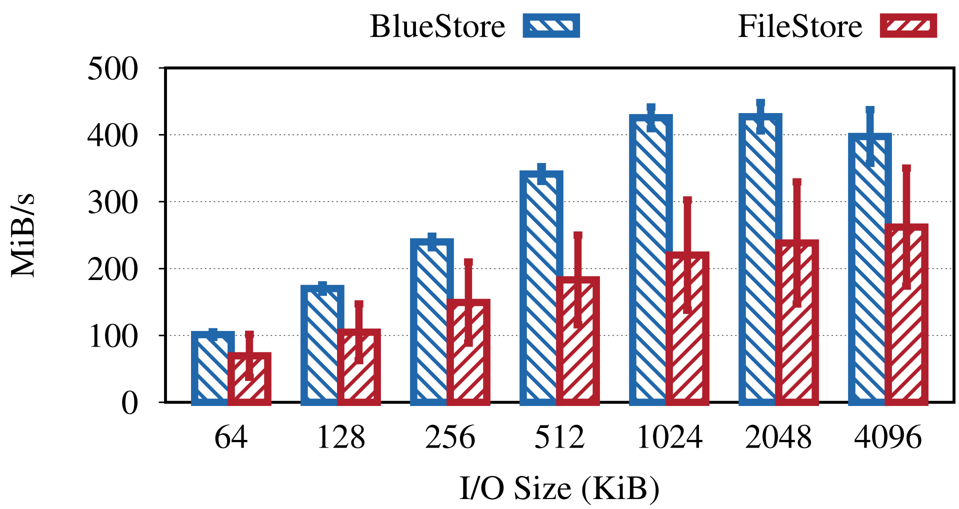

Fig. 8. Throughput of steady-state object writes to RADOS on a 16-node all-HDD cluster with different sizes using 128 threads. Compared to FileStore, the throughput is 50–100% greater on BlueStore and has a significantly lower variance.

7.2 RADOS Block Device Benchmarks

Next, we compare the performance of RADOS Block Device (RBD), a virtual block device service implemented on top of RADOS, when using the BlueStore and FileStore backends. RBD is implemented as a kernel module that exports a block device to the user, which can be formatted and mounted like a regular block device. Data written to the device are striped into 4-MiB RADOS objects and written in parallel to multiple OSDs over the network.

For RBD benchmarks we create a 1-TB virtual block device, format it with XFS, and mount it on the client. We use `fio` [7] to perform sequential and random I/O with queue depth of 256 and I/O sizes ranging from 4 KiB to 4 MiB. For each test, we write about 30 GiB of data. Before starting every experiment, we drop the OS page cache for FileStore, and we restart OSDs for BlueStore to eliminate caching effects in read experiments. We first run all the experiments on a Ceph cluster installed with FileStore backend. We then tear down the cluster, reinstall it with BlueStore backend, and repeat all the experiments.

Figure 11 shows the results for sequential writes, random writes, and sequential reads. For I/O sizes larger than 512 KiB, sequential and random write throughput is on average 1.7× and 2× higher with BlueStore, respectively, again mainly due to avoiding double-writes. BlueStore also displays a significantly lower throughput variance, because it can deterministically push data to disk. In FileStore, however, arbitrarily triggered writeback (Section 3.4) conflicts with the foreground writes to the WAL and introduces long request latencies.

For medium I/O sizes (128–512 KiB) the throughput difference decreases for sequential writes, because XFS masks out part of the cost of double writes in FileStore. With medium I/O sizes the writes to WAL do not fully utilize the disk. This leaves enough bandwidth for another write stream to go through and not have a large impact on the foreground writes to WAL. After writing the data synchronously to the WAL, FileStore then asynchronously writes it to the file system. XFS buffers these asynchronous writes and turns them into one large sequential write before issuing to disk. XFS cannot do the same for random writes, which is why the high throughput difference continues even for medium-sized random writes.

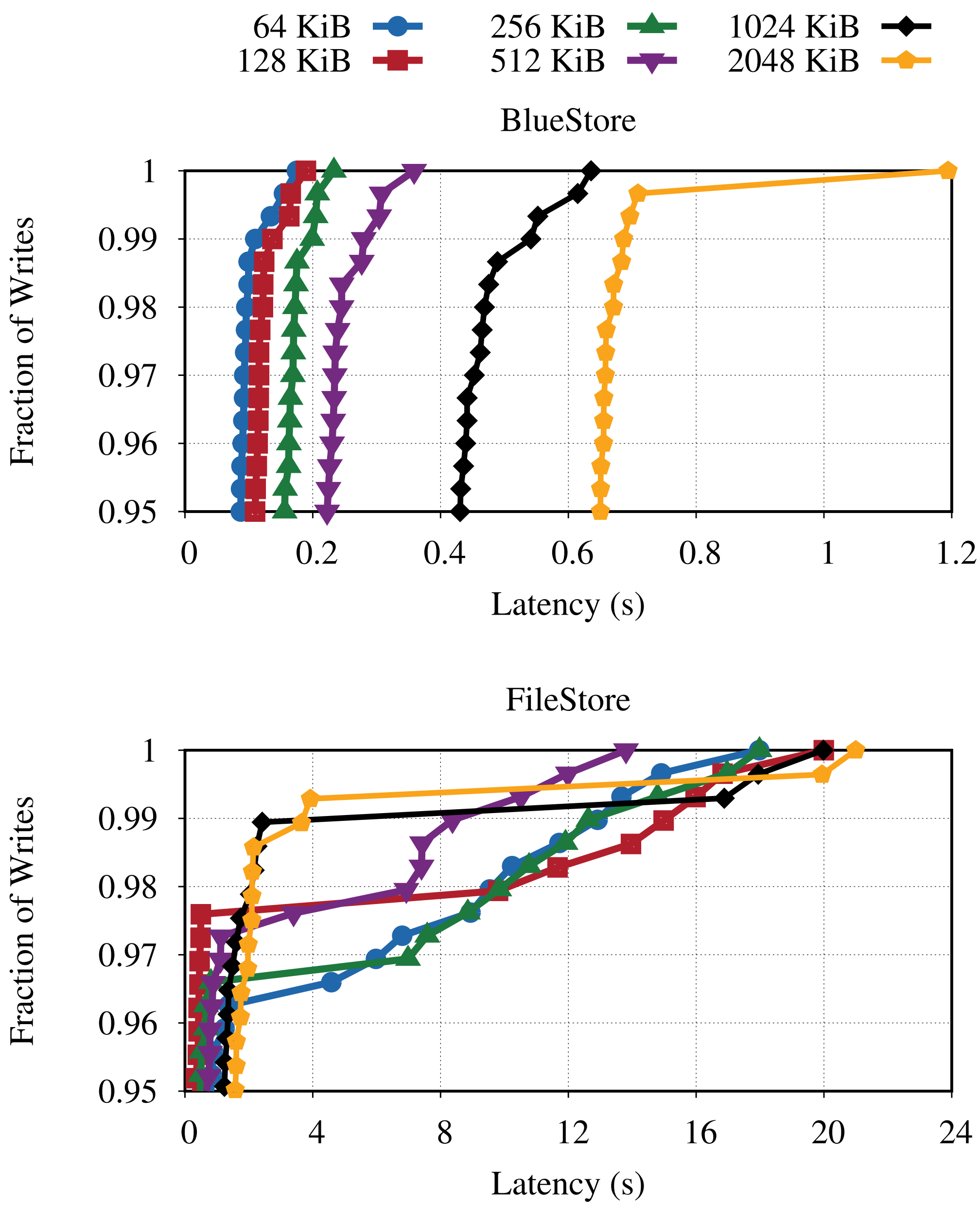

Fig. 9. The 95th and above percentile latencies of object writes to RADOS on a 16-node all-HDD cluster with different sizes using 128 threads. BlueStore (top graph) has an order of magnitude lower tail latency than FileStore (bottom graph).

Finally, for I/O sizes smaller than 64 KiB (not shown) the throughput of BlueStore is 20% higher than that of FileStore. For these I/O sizes BlueStore performs deferred writes by inserting data to RocksDB first, and then asynchronously overwriting the object data to avoid fragmentation (Section 4.2).

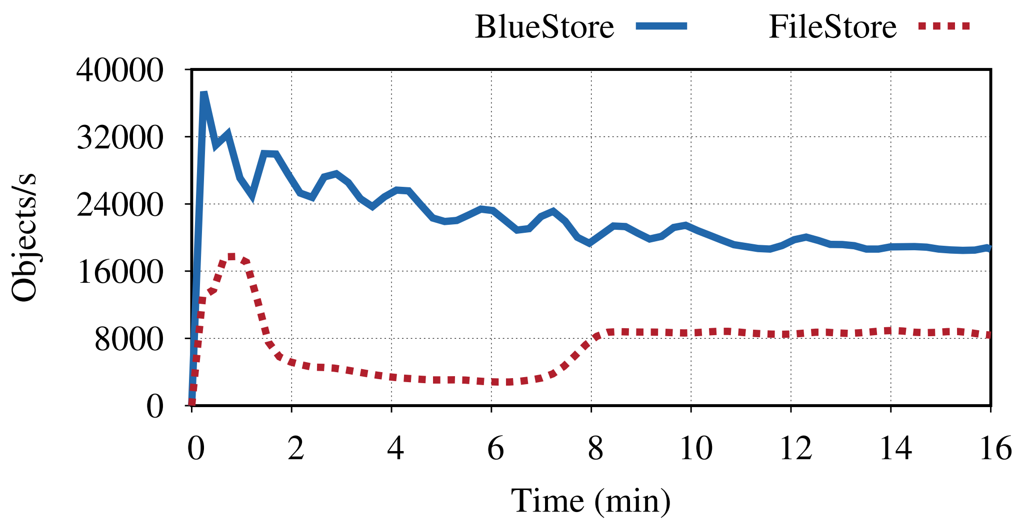

Fig. 10. Throughput of 4-KiB RADOS object writes with queue depth of 128 on a 16-node all-SSD cluster. At steady state, BlueStore is 2× faster than FileStore on SSD. BlueStore does not suffer from directory splitting; however, its throughput is gradually brought down by the RocksDB compaction overhead.

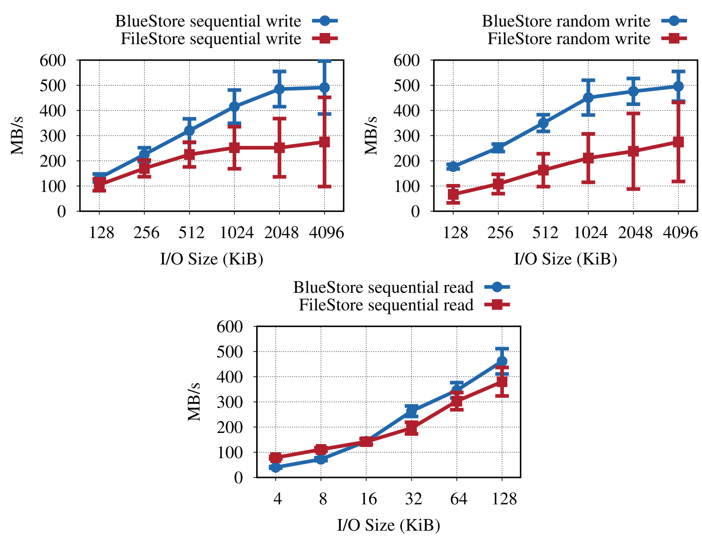

Fig. 11. Sequential write, random write, and sequential read throughput with different I/O sizes and queue depth of 256 on a 1 TB Ceph virtual block device (RBD) allocated on a 16-node all-HDD cluster. Results for an all-SSD cluster were similar but not shown for brevity.

The throughput of read operations in BlueStore is similar or slightly better than that of FileStore for I/O sizes larger than 32 KiB. For smaller I/O sizes, as the lower graph in Figure 11 shows, FileStore throughput is better because of the kernel readahead. While RBD does implement a readahead, it is not as well tuned as the kernel readahead.

7.3 Overwriting Erasure-coded Data

One of the features enabled by BlueStore is the efficient overwrite of EC data. We have measured the throughput of random overwrites for both BlueStore and FileStore. Our benchmark creates

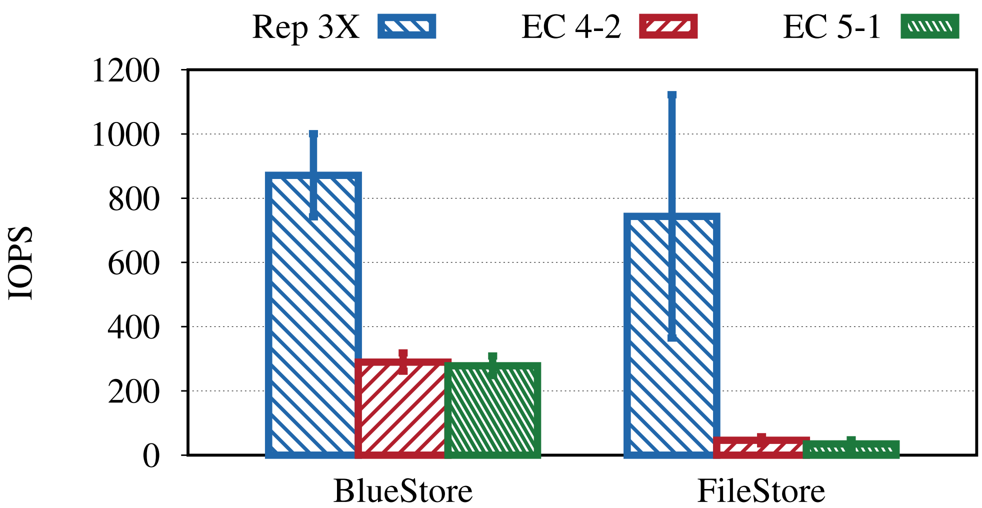

Fig. 12. IOPS observed from a client performing random 4-KiB writes with queue depth of 256 to a Ceph virtual block device (RBD). The device is allocated on a 16-node all-HDD cluster.

1 TB RBD using one client. The client mounts the block device and performs 5 GiB of random 4-KiB writes with queue depth of 256. Since the RBD is striped in 4-MiB RADOS objects, every write results in an object overwrite. We repeat the experiment on a virtual block device allocated on a replicated pool, on an EC pool with parameters $k = 4$ and $m = 2$ (*EC4-2*), and $k = 5$ and $m = 1$ (*EC5-1*).

Figure 12 compares the throughput of replicated and EC pools when using BlueStore and FileStore backends. BlueStore EC pools achieve 6× more IOPS on EC4-2, and 8× more IOPS on EC5-1 than FileStore. This is due to BlueStore avoiding full physical copies during the first phase of the two-phase commit required for overwriting EC objects (Section 5.2). As a result, it is practical to use EC pools with applications that require data overwrite, such as RBD and CephFS, with the BlueStore backend.

7.4 Storing Metadata on HM-SMR Hard Drives

In this section, we first demonstrate the performance of standalone RocksDB running on an HM-SMR hard drive. We then demonstrate the performance of Ceph configured to store metadata on a RocksDB instance running on an HM-SMR drive.

Standalone RocksDB on HM-SMR Drive Evaluation: We establish two baselines for the standalone RocksDB experiments. The first is RocksDB running on an XFS-formatted regular hard drive, which uses Conventional Magnetic Recording (CMR). This is the baseline we want to achieve with RocksDB on an HM-SMR drive, a similar mechanical device with a more restricted interface but higher capacity. The second baseline is RocksDB running on an XFS-formatted DM-SMR drive. This is a baseline we want to beat given that it is the only viable option for running RocksDB on a high-capacity SMR drive, but has suboptimal performance due to garbage collection. We use 3-TB Hitachi HUA72303 as a CMR drive, 10-TB Seagate ST8000AS0022 as a DM-SMR drive, and 14-TB HGST HSH721414AL as an HM-SMR drive.

For all of the experiments described in this section, we use the `fillrandom` benchmark of `db_bench` tool that comes with RocksDB. We perform an asynchronous insertion of 150 million key and value pairs of size 20 bytes and 400 bytes, respectively. During the benchmark run, RocksDB writes 200 GiB of data through memtable flushing, compaction, and WAL inserts, and reads 100 GiB

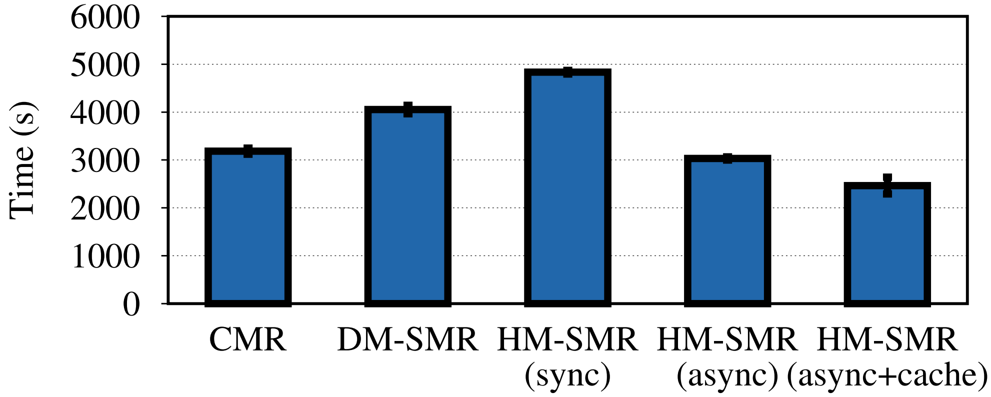

Fig. 13. Benchmark runtimes of the CMR and DM-SMR baselines and RocksDB on HM-SMR iterations. The benchmark performs 150 million *asynchronous* inserts of key-value pairs of size 20 bytes and 400 bytes, respectively.

of data due to compaction. The size of the database is 59 GiB uncompressed and 31 GiB compressed. To emulate a realistic environment where the amount of data in the OS page cache is a small fraction of the data stored on a high-capacity drive, we limit the operating system memory to 6 GiB, which leaves slightly more than 2 GiB of RAM for the page cache after the memory used by the OS and the benchmark application, resulting in 1:15 ratio of cached to on disk data.

We perform an extensive tuning of our baselines, focusing on two RocksDB parameters that have the highest impact on performance: `compaction_readahead_size` and `write_buffer_size`. We omit the detailed analysis of our performance tuning [3] and suffice by saying that tuning improved the performance of CMR and DM-SMR baselines by 34% and 63%, respectively.

We implement our solutions to the previously described challenges (Section 6) in multiple iterations. Figure 13 shows the time it takes to complete the benchmark for our DM-SMR drive and CMR drive baselines, as well as the different iterations of our implementation on the HM-SMR drive.

For our first iteration, we modify the BlueFS extent allocator to dedicate complete zones to large sequentially written files, such as SSTs, WALs, and the BlueFS journal, and to store small files with non-sequential I/O in conventional zones. We also modify BlueFS to use `libzbc` with direct I/O for all I/O operations. The middle bar in Figure 13, identified by "HM-SMR (sync)", shows that this iteration is 18% and 39% slower than, the DM-SMR and CMR baselines, respectively. Our detailed analysis of RocksDB performance [3] reveals that in the absence of the OS page cache, synchronous `zbc_pwrite` calls to the WAL file become the bottleneck.

In our second iteration, we switch to using the asynchronous `libaio` framework for all data reads and writes, and continue using `libzbc` for zone reset commands. The completion time of the benchmark for this iteration is indicated by the "HM-SMR (async)" bar in Figure 13. As we can see, using asynchronous I/O we surpass the runtime of the DM-SMR baseline and match the runtime of the CMR baseline.

In our third and final iteration, we incorporate a user-space file-cache that caches SSTs produced as a result of compaction. This prevents all of the compaction reads from hitting the disk, leaving more disk bandwidth for memtable flushes, which determines the insertion throughput. The rightmost bar in Figure 13 shows the runtime of our final iteration. It is 20% faster than the iteration without the cache and, 22% faster than the CMR baseline, and 38% faster than the DM-SMR baseline. While the advantage over the DM-SMR baseline is completely due to avoiding garbage

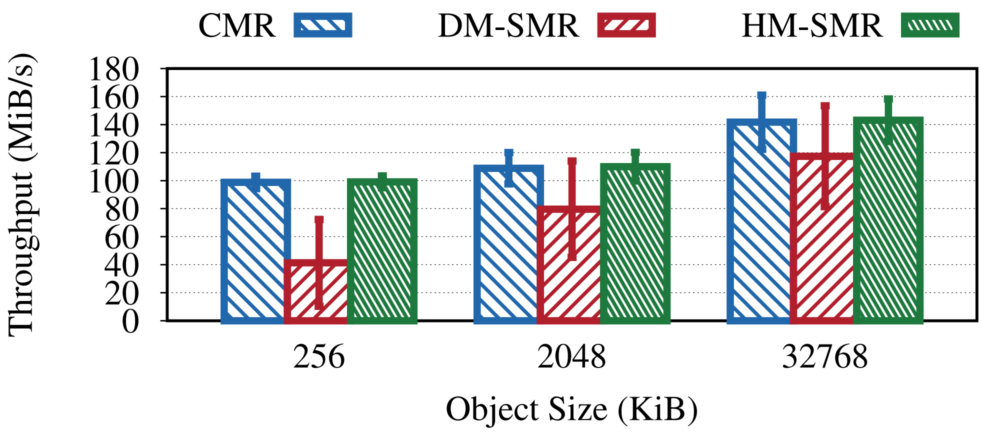

Fig. 14. The write throughput of a small Ceph cluster with metadata being stored on CMR, DM-SMR, and HM-SMR drives. The throughput in the DM-SMR case is 141%, 38%, and 22% lower for 256-KiB, 2-MiB, and 32-MiB object writes and has a larger variance than the CMR and the HM-SMR cases.

collection overhead, the advantage over CMR baseline stems mostly from the high sequential write throughput of the high-capacity HM-SMR drive.

Ceph with RocksDB Running on HM-SMR Drive Evaluation: To evaluate the performance of BlueStore with RocksDB running on an HM-SMR drive, we setup a three-node Ceph cluster and configure BlueStore to store data and metadata on separate drives. We run three experiments where we always store data on a CMR drive and alternate storing metadata on a CMR drive, on a DM-SMR drive, and on an HM-SMR drive. For CMR and DM-SMR cases we use stock BlueStore code that runs RocksDB on a raw block device, and for the HM-SMR case we run RocksDB on the modified BlueFS with aforementioned optimizations. In each experiment we write small (256-KiB), medium (2-MiB), and large (32-MiB) objects to the object store from a single client using 64 threads.

Figure 14 shows that when the metadata are stored on DM-SMR drive, the throughput is 141%, 38%, and 22% lower for small, medium, and large objects, respectively, from when it is stored on HM-SMR drive, and it has a large variance. When metadata are stored on CMR and HM-SMR drives, however, the throughput is similar and has lower variance.

The speedup of RocksDB on HM-SMR drive that we observed before does not directly translate to Figure 14, because most of the I/O is directed at the CMR drive, which stores object data. While the metadata traffic is not large enough to demonstrate the advantage of the HM-SMR drive, we have enabled Ceph to successfully store metadata on HM-SMR drive with zero overhead, making it one step away from fully leveraging the high bandwidth and capacity advantage of SMR. We are currently developing techniques for storing object data in HM-SMR drives as well.

8 CHALLENGES OF BUILDING EFFICIENT STORAGE BACKENDS ON RAW STORAGE

This section describes some of the challenges that the Ceph team faced when building a storage backend on raw storage devices from scratch.

8.1 Cache Sizing and Writeback

The OS fully utilizes the machine memory by dynamically growing or shrinking the size of the page cache based on the application's memory usage. It writes back the dirty pages to disk in the background trying not to adversely affect foreground I/O, so that memory can be quickly reused when applications ask for it.

A storage backend based on a local file system automatically inherits the benefits of the OS page cache. A storage backend that bypasses the local file system, however, has to implement a similar mechanism from scratch (Section 4.2). In BlueStore, for example, the cache size is a fixed configuration parameter that requires manual tuning. Building an efficient user space cache with the dynamic resizing functionality of the OS page cache is an open problem shared by other projects, like PostgreSQL [26] and RocksDB [45]. With the arrival of fast NVMe SSDs, such a cache needs to be efficient enough that it does not incur overhead for write-intensive workloads—a deficiency that current page cache suffers from [21].

8.2 Key-value Store Efficiency

The experience of the Ceph team demonstrates that moving all of the metadata to an ordered key-value store, like RocksDB, significantly improves the efficiency of metadata operations. However, the Ceph team has also found that embedding RocksDB in BlueStore is problematic in multiple ways: (1) RocksDB's compaction and high write amplification have been the primary performance limiters when using NVMe SSDs in OSDs; (2) since RockDB is treated as a black box, data are serialized and copied in and out of it, consuming CPU time; and (3) RocksDB has its own threading model, which limits the ability to do custom sharding. These and other problems with RocksDB and similar key-value stores keeps the Ceph team researching better solutions.

8.3 CPU and Memory Efficiency

Modern compilers align and pad basic datatypes in memory so that CPU can fetch data efficiently, thereby increasing performance. For applications with complex `structs`, the default layout can waste a significant amount of memory [23, 68]. Many applications are rightly not concerned with this problem, because they allocate short-lived data structures. A storage backend that bypasses the OS page cache, however, runs continously and controls almost all of a machine's memory. Therefore, the Ceph team spent a lot of time packing structures stored in RocksDB to reduce the total metadata size and also compaction overhead. The main tricks used were delta and variable-integer encoding.

Another observation with BlueStore is that on high-end NVMe SSDs the workloads are becoming increasingly CPU-bound. For its next-generation backend, the Ceph community is exploring techniques that reduce CPU consumption, such as minimizing data serialization-deserialization, and using the SeaStar framework [84] with shared-nothing model that avoids context switches due to locking.

9 RELATED WORK

The primary motivator for BlueStore is the lack of transactions and unscalable metadata operations in local file systems. In this section we compare BlueStore to previous research that aims to address these problems.

Transaction Support. Previous works have generally followed three approaches when introducing transactional interface to file system users.

The first approach is to leverage the in-kernel transaction mechanism present in the file systems. Examples of this are Btrfs' export of transaction system calls to userspace [24], Transactional NTFS [54], Valor [92], and TxFS [42]. The drawbacks of this approach are the complexity and incompleteness of the interface, and a significant implementation complexity. For example, Btrfs and NTFS both recently deprecated their transaction interface [15, 55] citing difficulty guaranteeing correct or safe usage, which corroborates FileStore's experience (Section 3.1.1). Valor [92], while not tied to a specific file system, also has a nuanced interface that requires correct use of a combo of seven system calls, and a complex in-kernel implementation. TxFS is a recent work that introduces a simple interface built on ext4's journaling layer; however, its implementation requires non-trivial amount of change to the Linux kernel. BlueStore, informed by FileStore's experience, avoids using file systems' in-kernel transaction infrastructure.

The second approach builds a user space file system atop a database, utilizing existing transactional semantics. For example, Amino [112] relies on Berkeley DB [70] as the backing store, and Inversion [69] stores files in a POSTGRES database [96]. While these file systems provide seamless transactional operations, they generally suffer from high performance overhead, because they accrue the overhead of the layers below. BlueStore similarly leverages a transactional database, but incurs zero overhead, because it eliminates the local file system and runs the database on a raw disk.

The third approach provides transactions as a first-class abstraction in the OS and implements all services, including the file system, using transactions. QuickSilver [82] is an example of such system that uses built-in transactions for implementing a storage backend for a distributed file system. Similarly, TxOS [77] adds transactions to the Linux kernel and converts ext3 into a transactional file system. This approach, however, is too heavyweight for achieving file system transactions, and such a kernel is tricky to maintain [42].

Metadata Optimizations. A large body of work has produced a plethora of approaches to metadata optimizations in local file systems. BetrFS [50] introduces B^{ϵ}-Tree as an indexing structure for efficient large scans. DualFS [75], hFS [117], and ext4-lazy [2] abandon traditional FFS [65] cylinder group design and aggregate all metadata in one place to achieve significantly faster metadata operations. TableFS [80] and DeltaFS [118] store metadata in LevelDB running atop a file system and achieve orders of magnitude faster metadata operations than local file systems.

While BlueStore also stores metadata in RocksDB—a LevelDB derivative—to achieve similar speedup, it differs from the above in two important ways: (1) in BlueStore, RocksDB runs atop a raw disk incurring zero overhead, and (2) BlueStore keeps all metadata, including the internal metadata, in RocksDB as key-value pairs. Storing internal metadata as variable-sized key-value pairs, as opposed to fixed-sized records on disk, scales more easily. For example, the Lustre distributed file system that uses an ext4-derivate called LDISKFS for the storage backend, has changed on-disk format twice in a short period to accommodate for increasing disk sizes [12, 13].

10 CONCLUSION

Distributed file system developers conventionally adopt local file systems as their storage backend. They then try to fit the general-purpose file system abstractions to their needs, incurring significant accidental complexity [14]. At the core of this convention lies the belief that developing a storage backend from scratch is an arduous process, akin to developing a new file system that takes a decade to mature.

Our article, relying on the Ceph team's experience, shows this belief to be inaccurate. Furthermore, we find that developing a *special-purpose*, user space storage backend from scratch (1) reclaims the significant performance left on the table when building a backend on a general-purpose file system, (2) makes it possible to adopt novel, backward incompatible storage hardware, and (3) enables new features by gaining complete control of the I/O stack. We hope that this experience article will initiate discussions among storage practitioners and researchers on fresh approaches to designing distributed file systems and their storage backends.

ACKNOWLEDGMENTS

We thank Robert Morris (our shepherd), Matias Bjørling, and the anonymous reviewers for their feedback. We acknowledge the BlueStore authors, which include Igor Fedotov, Xie Xingguo, Ma

Jianpeng, Allen Samuels, Varada Kari, and Haomai Wang. We also thank the members and companies of the PDL Consortium: Alibaba Group, Amazon, Datrium, Facebook, Google, Hewlett Packard Enterprise, Hitachi Ltd., Intel Corporation, IBM, Micron, Microsoft Research, NetApp, Inc., Oracle Corporation, Salesforce, Samsung Semiconductor Inc., Seagate Technology, and Two Sigma for their interest, insights, feedback, and support.

REFERENCES

[1] Abutalib Aghayev and Peter Desnoyers. 2015. Skylight—A window on shingled disk operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 135–149.
[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for shingled disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 105–120.
[3] Abutalib Aghayev, Sage Weil, Greg Ganger, and George Amvrosiadis. 2019. *Reconciling LSM-Trees with Modern Hard Drives Using BlueFS.* Technical Report CMU-PDL-19-102. CMU Parallel Data Laboratory.
[4] Amazon.com, Inc. 2019. Amazon Elastic Block Store. Retrieved from https://aws.amazon.com/ebs/.
[5] Amazon.com, Inc. 2019. Amazon S3. Retrieved from https://aws.amazon.com/s3/.
[6] Jens Axboe. 2009. Queue sysfs Files. Retrieved from https://www.kernel.org/doc/Documentation/block/queue-sysfs.txt.
[7] Jens Axboe. 2016. Flexible I/O Tester. Retrieved from git://git.kernel.dk/fio.git.
[8] Jens Axboe. 2016. Throttled Background Buffered Writeback. Retrieved from https://lwn.net/Articles/698815/.
[9] Matias Bjørling. 2019. From open-channel SSDs to zoned namespaces. In *Proceedings of the Linux Storage and Filesystems Conference (Vault 19)*. USENIX Association.
[10] Matias Bjørling. 2019. New NVMe Specification Defines Zoned Namespaces (ZNS) as Go-To Industry Technology. Retrieved from https://nvmexpress.org/new-nvmetm-specification-defines-zoned-namespaces-zns-as-go-to-industry-technology/.
[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 359–374.
[12] Artem Blagodarenko. 2016. Scaling LDISKFS for the Future. Retrieved from https://www.youtube.com/watch?v=ubbZGpxV6zk.
[13] Artem Blagodarenko. 2017. Scaling LDISKFS for the Future. Again. Retrieved from https://www.youtube.com/watch?v=HLfEd0_Dq0U.
Q2 [14] Frederick P. Brooks Jr. 1986. No Silver Bullet—Essence and Accident in Software Engineering.
[15] Btrfs. 2019. Btrfs Changelog. Retrieved from https://btrfs.wiki.kernel.org/index.php/Changelog.
[16] David C. 2018. [ceph-users] Luminous | PG Split Causing Slow Requests. Retrieved from http://lists.ceph.com/pipermail/ceph-users-ceph.com/2018-February/024984.html.
[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (Jun. 2008). DOI: https://doi.org/10.1145/1365815.1365816
[18] Luoqing Chao and Thunder Zhang. 2015. Implement Object Storage with SMR Based Key-Value Store. Retrieved from https://www.snia.org/sites/default/files/SDC15_presentations/smr/QingchaoLuo_Implement_Object_Storage_SMR_Key-Value_Store.pdf.
[19] Dave Chinner. 2010. XFS Delayed Logging Design. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/xfs-delayed-logging-design.txt.
[20] Dave Chinner. 2015. SMR Layout Optimization for XFS. Retrieved from http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf.
[21] Dave Chinner. 2019. Re: Pagecache Locking (Was: bcachefs Status Update) Merged). Retrieved from https://lkml.org/lkml/2019/6/13/1794.
[22] Alibaba Clouder. 2018. Alibaba Deploys Alibaba Open Channel SSD for Next Generation Data Centers. Retrieved from https://www.alibabacloud.com/blog/alibaba-deploys-alibaba-open-channel-ssd-for-next-generation-data-centers_593802.
[23] William Cohen. 2016. How to Avoid Wasting Megabytes of Memory a Few Bytes at a Time. Retrieved from https://developers.redhat.com/blog/2016/06/01/how-to-avoid-wasting-megabytes-of-memory-a-few-bytes-at-a-time/.
[24] Jonathan Corbet. 2009. Supporting Transactions in Btrfs. Retrieved from https://lwn.net/Articles/361457/.
[25] Jonathan Corbet. 2011. No-I/O Dirty Throttling. Retrieved from https://lwn.net/Articles/456904/.
[26] Jonathan Corbet. 2018. PostgreSQL's fsync() Surprise. Retrieved from https://lwn.net/Articles/752063/.
[27] Western Digital. 2019. Zoned Storage. Retrieved from http://zonedstorage.io.

[28] Anton Dmitriev. 2017. [ceph-users] All OSD Fails after Few Requests to RGW. Retrieved from http://lists.ceph.com/pipermail/ceph-users-ceph.com/2017-May/017950.html.
[29] Siying Dong. 2018. Direct I/O Close() Shouldn't Rewrite the Last Page. Retrieved from https://github.com/facebook/rocksdb/pull/4771.
[30] Jake Edge. 2015. Filesystem Support for SMR Devices. Retrieved from https://lwn.net/Articles/637035/.
[31] Jake Edge. 2015. The OrangeFS Distributed Filesystem. Retrieved from https://lwn.net/Articles/643165/.
[32] Jake Edge. 2015. XFS: There and Back ... and There Again? Retrieved from https://lwn.net/Articles/638546/.
[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, New York, NY, 251–266. DOI: https://doi.org/10.1145/224056.224076
[34] Facebook, Inc. 2018. A RocksDB Storage Engine with MySQL. Retrieved from http://myrocks.io/.
[35] Andrew Fikes. 2010. Storage Architecture and Challenges. Retrieved from https://cloud.google.com/files/storage_architecture_and_challenges.pdf.
[36] Mary Jo Foley. 2018. Microsoft readies new cloud SSD storage spec for the Open Compute Project. Retrieved from https://www.zdnet.com/article/microsoft-readies-new-cloud-ssd-storage-spec-for-the-open-compute-project/.
[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 29–43. DOI: https://doi.org/10.1145/945445.945450
[38] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Santa Clara, CA, 263–276.
[39] Christoph Hellwig. 2009. XFS: The big storage file system for Linux. *USENIX ;login* 34, 5 (2009).
[40] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. 1987. Scale and performance in a distributed file system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*. ACM, New York, NY, 1–2. DOI: https://doi.org/10.1145/41457.37500
[41] Joel Hruska. 2019. Western Digital to Demo Dual Actuator HDD, Will Use SMR to Hit 18TB Capacity. Retrieved from https://www.extremetech.com/computing/287319-western-digital-to-demo-dual-actuator-hdd-will-use-smr-to-hit-18tb-capacity.
[42] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2018. TxFS: Leveraging file-system crash consistency to provide ACID transactions. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 879–891.
[43] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, 15–26.
[44] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. 2008. The XtreemFS architecture—A case for object-based file systems in grids. *Concurr. Comput.: Pract. Exper.* 20, 17 (Dec. 2008), 2049–2060. DOI: https://doi.org/10.1002/cpe.v20:17
[45] Facebook Inc. 2019. RocksDB Direct IO. Retrieved from https://github.com/facebook/rocksdb/wiki/Direct-IO.
[46] Facebook Inc. 2019. RocksDB Merge Operator. Retrieved from https://github.com/facebook/rocksdb/wiki/Merge-Operator.
[47] Facebook Inc. 2019. RocksDB Synchronous Writes. Retrieved from https://github.com/facebook/rocksdb/wiki/Basic-Operations#synchronous-writes.
[48] Silicon Graphics Inc. 2006. XFS Allocation Groups. Retrieved from http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/Allocation_Groups.html.
[49] INCITS T10 Technical Committee. 2014. *Information Technology—Zoned Block Commands (ZBC)*. Draft Standard T10/BSR INCITS 536. American National Standards Institute, Inc. Retrieved from http://www.t10.org/drafts.htm.
[50] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: Write-optimization in a kernel file system. *Trans. Stor.* 11, 4, Article 18 (Nov. 2015), 29 pages. DOI: https://doi.org/10.1145/2798729
[51] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX, 309–320.
[52] Theodore Johnson and Dennis Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann, San Francisco, CA, 439–450. http://dl.acm.org/citation.cfm?id=645920.672996
[53] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application performance and flexibility on

exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 52–65. DOI: https://doi.org/10.1145/268998.266644

[54] John Kennedy and Michael Satran. 2018. About Transactional NTFS. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/fileio/about-transactional-ntfs.

[55] John Kennedy and Michael Satran. 2018. Alternatives to using Transactional NTFS. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/fileio/deprecation-of-txf.

[56] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO complying SSDs through OPS isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 183–189.

[57] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (Apr. 2010), 35–40. DOI: https://doi.org/10.1145/1773912.1773922

Q3 [58] Butler Lampson and Howard E. Sturgis. 1979. Crash recovery in a distributed data storage system. (1979).

[59] Adam Leventhal. 2016. APFS in Detail: Overview. Retrieved from http://dtrace.org/blogs/ahl/2016/06/19/apfs-part1/.

[60] Peter Macko, Xiongzi Ge, John Haskins Jr., James Kelley, David Slik, Keith A. Smith, and Maxim G. Smith. 2017. SMORE: A cold data object store for SMR drives (extended version). *CoRR* abs/1705.09701 (2017). http://arxiv.org/abs/1705.09701 Q4

[61] Magic Pocket & Hardware Engineering Teams. 2018. Extending Magic Pocket Innovation with the First Petabyte Scale SMR Drive Deployment. Retrieved from https://blogs.dropbox.com/tech/2018/06/extending-magic-pocket-innovation-with-the-first-petabyte-scale-smr-drive-deployment/.

[62] Magic Pocket & Hardware Engineering Teams. 2019. SMR: What We Learned in Our First Year. Retrieved from https://blogs.dropbox.com/tech/2019/07/smr-what-we-learned-in-our-first-year/.

[63] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien LeMoal, Carlos Maltzahn, and Zvonimr Bandic. 2016. ZEA, a data management approach for SMR. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*. USENIX Association.

[64] Lars Marowsky-Brée. 2018. Ceph User Survey 2018 Results. Retrieved from https://ceph.com/ceph-blog/ceph-user-survey-2018-results/.

[65] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst.* 2, 3 (1984), 181–197.

[66] Chris Mellor. 2019. Toshiba Embraces Shingling for Next-gen MAMR HDDs. Retrieved from https://blocksandfiles.com/2019/03/11/toshiba-mamr-statements-have-shingling-absence/.

[67] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, 221–234.

[68] Sumedh N. 2013. Coding for Performance: Data alignment and structures. Retrieved from https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures.

Q5 [69] Michael A. Olson. 1993. The design and implementation of the inversion file system. In *USENIX Winter*. USENIX Association, Berkeley, CA.

[70] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'99)*. USENIX Association, Berkeley, CA, 43–43.

[71] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (Jun. 1996), 351–385. DOI: https://doi.org/10.1007/s002360050048

[72] OpenStack Foundation. 2017. 2017 Annual Report. Retrieved from https://www.openstack.org/assets/reports/OpenStack-AnnualReport2017.pdf.

[73] Adrian Palmer. 2015. SMRFFS-EXT4—SMR Friendly File System. Retrieved from https://github.com/Seagate/SMR_FS-EXT4.

[74] Swapnil Patil and Garth Gibson. 2011. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, 13–13. http://dl.acm.org/citation.cfm?id=1960475.1960488

[75] Juan Piernas, Toni Cortes, and José M. García. 2002. DualFS: A new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing (ICS'02)*. Association for Computing Machinery, New York, NY, 137–146. DOI: https://doi.org/10.1145/514191.514213

[76] Poornima G and Rajesh Joseph. 2016. Metadata Performance Bottlenecks in Gluster. Retrieved from https://www.slideshare.net/GlusterCommunity/performance-bottlenecks-for-metadata-workload-in-gluster-with-poornima-gurusiddaiah-rajesh-joseph.

[77] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 161–176. DOI: https://doi.org/10.1145/1629575.1629591

[78] Lee Prewitt. 2019. SMR and ZNS—Two Sides of the Same Coin. Retrieved from https://www.youtube.com/watch?v=jBxzO6YyMxU.

[79] Red Hat Inc. 2019. GlusterFS Architecture. Retrieved from https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/.

[80] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX, 145–156.

[81] Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. ACM, New York, NY, 1–15. DOI: https://doi.org/10.1145/121132.121137

[82] Frank Schmuck and Jim Wylie. 1991. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. ACM, New York, NY, 239–253. DOI: https://doi.org/10.1145/121132.121171

[83] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. 2004. Disk scrubbing in large archival storage systems. In *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*. IEEE Computer Society, 409–418. http://dl.acm.org/citation.cfm?id=1032659.1034226

[84] Seastar. 2019. Shared-nothing Design. Retrieved from http://seastar.io/shared-nothing/.

[85] Margo I. Seltzer. 1993. Transaction support in a log-structured file system. In *Proceedings of the 9th International Conference on Data Engineering*. IEEE Computer Society, 503–510.

[86] Kai Shen, Stan Park, and Men Zhu. 2014. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX, 287–293.

[87] Anton Shilov. 2017. Seagate Ships 35th Millionth SMR HDD, Confirms HAMR-Based Drives in Late 2018. Retrieved from https://www.anandtech.com/show/11315/seagate-ships-35th-millionth-smr-hdd-confirms-hamrbased-hard-drives-in-late-2018.

[88] A. Shilov. 2019. Western Digital: Over Half of Data Center HDDs Will Use SMR by 2023. Retrieved from https://www.anandtech.com/show/14099/western-digital-over-half-of-dc-hdds-will-use-smr-by-2023.

[89] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST'10)*. IEEE Computer Society, 1–10. DOI: https://doi.org/10.1109/MSST.2010.5496972

[90] Chris Siebenmann. 2011. About the Order That readdir() Returns Entries In. Retrieved from https://utcc.utoronto.ca/ cks/space/blog/unix/ReaddirOrder.

[91] Chris Siebenmann. 2013. ZFS Transaction Groups and the ZFS Intent Log. Retrieved from https://utcc.utoronto.ca/cks/space/blog/solaris/ZFSTXGsAndZILs.

[92] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. 2009. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST'09)*. USENIX Association, 29–42.

[93] Stas Starikevich. 2016. [ceph-users] RadosGW performance degradation on the 18 millions objects stored. Retrieved from http://lists.ceph.com/pipermail/ceph-users-ceph.com/2016-September/012983.html.

[94] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. 2010. BabuDB: Fast and efficient file system metadata storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'10)*. IEEE Computer Society, 51–58. DOI: https://doi.org/10.1109/SNAPI.2010.14

[95] Michael Stonebraker. 1981. Operating system support for database management. *Commun. ACM* 24, 7 (Jul. 1981), 412–418. DOI: https://doi.org/10.1145/358699.358703

[96] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD'86)*. ACM, New York, NY, 340–355. DOI: https://doi.org/10.1145/16894.16888

[97] ZAR team. 2019. "Write hole" phenomenon. Retrieved from http://www.raid-recovery-guide.com/raid5-write-hole.aspx.

[98] ThinkParQ. 2018. An introduction to BeeGFS. Retrieved from https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf.

[99] Stephen C. Tweedie. 1998. Journaling the Linux ext2fs Filesystem. In *Proceedings of the 4th Annual Linux Expo*.

[100] Sage Weil. 2009. Re: [RFC] Big Fat Transaction ioctl. Retrieved from https://lwn.net/Articles/361472/.

[101] Sage Weil. 2009. [RFC] Big Fat Transaction ioctl. Retrieved from https://lwn.net/Articles/361439/.

[102] Sage Weil. 2011. [PATCH v3] Introduce sys_syncfs to Sync a Single File System. Retrieved from https://lwn.net/Articles/433384/.

[103] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 307–320.

[104] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*. Association for Computing Machinery, New York, NY, 122–es. DOI: https://doi.org/10.1145/1188455.1188582

[105] 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)*. ACM, New York, NY, 35–44. DOI: https://doi.org/10.1145/1374596.1374606

[106] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, Article 2, 17 pages.

[107] Western Digital Inc. 2018. ZBC device manipulation library. Retrieved from https://github.com/hgst/libzbc.

[108] Lustre Wiki. 2017. Introduction to Lustre Architecture. Retrieved from http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf.

[109] Wikipedia. 2018. Btrfs History. Retrieved from https://en.wikipedia.org/wiki/Btrfs#History.

[110] Wikipedia. 2018. XFS History. Retrieved from https://en.wikipedia.org/wiki/XFS#History.

[111] Wikipedia. 2019. Cache flushing. Retrieved from https://en.wikipedia.org/wiki/Disk_buffer#Cache_flushing.

[112] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. 2007. Extending ACID semantics to the file system. *ACM Trans. Stor.* 3, 2 (Jun. 2007), 4–es. DOI: https://doi.org/10.1145/1242520.1242521

[113] Fengguang Wu. 2012. I/O-less Dirty Throttling. Retrieved from https://events.linuxfoundation.org/images/stories/pdf/lcjp2012_wu.pdf.

[114] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 15–28.

[115] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. GearDB: A GC-free key-value store on HM-SMR drives with gear compaction. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 159–171.

[116] Lawrence Ying and Theodore Ts'o. 2017. Dynamic Hybrid-SMR: An OCP proposal to improve data center disk drives. Retrieved from https://www.blog.google/products/google-cloud/dynamic-hybrid-smr-ocp-proposal-improve-data-center-disk-drives/.

[117] Zhihui Zhang and Kanad Ghose. 2007. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM, New York, NY, 175–187. DOI: https://doi.org/10.1145/1272996.1273016

[118] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling embedded in-situ indexing with deltaFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. IEEE Press, Article 3, 15 pages. http://dl.acm.org/citation.cfm?id=3291656.3291660

[119] Alexey Zhuravlev. 2016. ZFS: Metadata Performance. Retrieved from https://www.eofs.eu/_media/events/lad16/02_zfs_md_performance_improvements_zhuravlev.pdf.

Received January 2020; accepted March 2020

Author Queries

Q1: AU: Please provide complete mailing and email addresses for all authors.
Q2: AU: Ref. 14: Please update and complete.
Q3: AU: Ref. 58: Please update and complete.
Q4: AU: If possible, please update and complete all arXiv/CoRR refs in this list.
Q5: AU: Ref 69: Is this a Proceedings? Please update and complete.