File Systems Unfit as Distributed Storage Back Ends

Lessons from 10 Years of Ceph Evolution

ABUTALIB AGHAYEV, SAGE WEIL, MICHAEL KUCHNIK, MARK NELSON, GREG GANGER, AND GEORGE AMVROSIADIS

Abutalib Aghayev is a PhD student in the Computer Science Department at Carnegie Mellon University. He has broad research interests in computer systems, including storage and file systems, distributed systems, and operating systems. agayev@cs.cmu.edu

Sage Weil is the Lead Architect and co-creator of the Ceph open source distributed storage system. Ceph was created to provide a stable, next generation distributed storage system for Linux. Inktank was co-founded by Sage in 2012 to support enterprise Ceph users, and then acquired by Red Hat in 2014. Today Sage continues to lead the Ceph developer community and to help shape Red Hat's overall storage strategy. sweil@redhat.com

Michael Kuchnik is a PhD student in the Computer Science Department at Carnegie Mellon University and a member of the Parallel Data Lab. His research interests are in the design and analysis of computer systems, specifically those involving storage, high performance computing, or machine learning. Before coming to CMU, he earned his BS in computer engineering from the Georgia Institute of Technology. mkuchnik@cmu.edu

For a decade, the Ceph distributed file system followed the conventional wisdom of building its storage back end on top of local file systems. The experience with different file systems showed that this approach always leaves significant performance on the table while incurring significant accidental complexity [2]. Therefore, the Ceph team embarked on an ambitious project to build BlueStore, a new back end designed to run directly on raw storage devices. Somewhat surprisingly, BlueStore matured in less than two years. It outperformed back ends built atop file systems and got adopted by 70% of users in production.

Figure 1 shows the high-level architecture of Ceph. At the core of Ceph is the Reliable Autonomic Distributed Object Store (RADOS) service. RADOS scales to thousands of Object Storage 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; the RADOS Block Device (RBD), a virtual block device similar to Amazon EBS; 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. 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 I/O requests from `librados` clients and cooperates with peer OSDs to replicate or erasure code updates, migrate data, or recover from failures. Data is persisted to the local device via the internal *ObjectStore* interface, which is the storage back-end interface in Ceph. ObjectStore 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.

The FileStore storage back end is an ObjectStore implementation on top of a local file system. In FileStore, an object collection is mapped to a directory and object data is stored in a file. Throughout the years, FileStore was ported to run on top of Btrfs, XFS, ext4, and ZFS, with FileStore on XFS becoming the de facto back end because it scaled better and had faster metadata performance [7].

Mark Nelson joined the Ceph team in January 2012 and has 12 years of experience in distributed systems, HPC, and bioinformatics. Mark works on Ceph performance analysis and is the primary author of the Ceph Benchmarking Toolkit. He runs the weekly Ceph performance meeting and is currently focused on research and development of Ceph's next-generation object store. mnelson@redhat.com

Greg Ganger is the Jatras Professor of Electrical and Computer Engineering at Carnegie Mellon University and Director of the Parallel Data Lab (www.pdl.cmu.edu). He has broad research interests, with current projects exploring system support for large-scale ML (Big Learning), resource management in cloud computing, and software systems for heterogeneous storage clusters, HPC storage, and NVM. His PhD in CS&E is from the University of Michigan. ganger@ece.cmu.edu

George Amvrosiadis is an Assistant Research Professor of Electrical and Computer Engineering at Carnegie Mellon University and a member of the Parallel Data Lab. His current research focuses on distributed and cloud storage, new storage technologies, high performance computing, and storage for machine learning. His team's research has received an R&D100 Award and was featured on *WIRED*, *The Morning Paper*, and *Hacker News*. He co-teaches two graduate courses on Storage Systems and Advanced Cloud Computing attended by 100+ graduate students each. gamvrosi@cmu.edu

BlueStore: A Clean-Slate Approach

The BlueStore storage back end is a new implementation of ObjectStore designed from scratch to run on raw block devices, aiming to solve the challenges [2] faced by FileStore. Some of the main goals of BlueStore were:

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

BlueStore achieved all of these goals within just two years and became the default storage back end 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. First, BlueStore implements a small, special-purpose interface and not a complete POSIX I/O specification. Second, BlueStore is implemented in userspace, 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 (see "Features Enabled by BlueStore," below).

The high-level architecture of BlueStore is shown in Figure 2. A space allocator within BlueStore determines the location of new data, which is asynchronously written to raw disk using direct I/O. Internal metadata and user object metadata is stored in 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.

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 [10] for data write, as opposed to having two cache flushes when writing data to a file on top of a journaling file system. Second, it changes RocksDB to reuse write-ahead log 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 userspace, extent-based, and journaling file system. It implements basic system calls

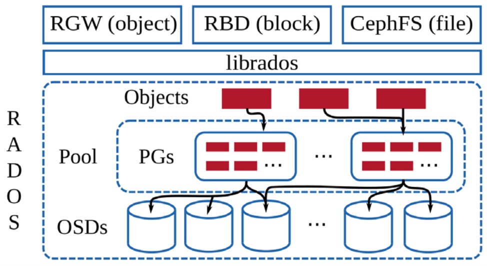

Figure 1: High-level depiction of Ceph's architecture. A single pool with 3× replication is shown. Therefore, each placement group (PG) is replicated on three OSDs.

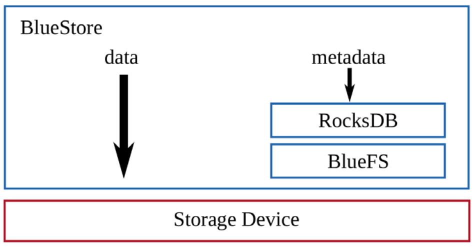

Figure 2: The high-level architecture of BlueStore. Data is written to the raw storage device using direct I/O. Metadata is written to RocksDB running on top of BlueFS. BlueFS is a userspace library file system designed for RocksDB, and it also runs on top of the raw storage device.

required by RocksDB, such as `open`, `mkdir`, and `pwrite`. 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 is stored in the *B* namespace, and collection metadata is 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 six significant bits of `e4`. Hence, the object `012.e532` is a member, whereas the object `012.e832` is not. 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 was an expensive operation performed by renaming many directories in a deeply nested hierarchy.

Data Path and Space Allocation

BlueStore is a copy-on-write back end. For incoming writes larger than a *minimum allocation size* (64 KiB for HDDs, 16 KiB for SSDs), the data is written to a newly allocated extent. Once the data is persisted, the corresponding metadata is 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 [8].

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 userspace 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 userspace, using the scan-resistant 2Q algorithm. 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.

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.

Space-Efficient Checksums

Ceph scrubs metadata every day and data every week. Even with scrubbing, however, if the data is inconsistent across replicas 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.

On the other hand, 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.

Overwrite of Erasure-Coded Data

Ceph has supported erasure-coded (EC) pools through the FileStore back end 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 was limited to RGW; for RBD and CephFS only replicated pools were used.

To avoid the "RAID write hole" problem, 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.

Transparent Compression

Transparent compression is crucial for scale-out distributed file systems because 3× replication increases storage costs. BlueStore implements transparent compression where written data is 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.

Exploring New Interfaces

Despite multiple attempts [5, 9], 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 [6]. Supporting these denser drives, however, is important for scale-out distributed file systems because it lowers storage costs.

Unconstrained by the block-based designs of local file systems, BlueStore has the freedom of exploring novel interfaces and data layouts. This has recently enabled porting RocksDB and BlueFS to run on host-managed SMR drives, and an effort is underway to store object data on such drives next [1]. In addition, the Ceph community is exploring a new back end that targets a combination of persistent memory and emerging NVMe devices with novel interfaces, such as ZNS SSDs [3].

Evaluation

This section compares the performance of a Ceph cluster using FileStore, a back end built on a local file system, and BlueStore, a back end using the storage device directly. We compare the throughput of object writes to the RADOS distributed object storage.

We ran all experiments on a 16-node Ceph cluster connected with a Cisco Nexus 3264-Q 64-port QSFP+ 40GbE switch. Each node had a 16-core Intel E5-2698Bv3 Xeon 2GHz CPU, 64GiB RAM, 400GB Intel P3600 NVMe SSD, 4TB 7200RPM Seagate ST4000NM0023 HDD, and a Mellanox MCX314A-BCCT 40GbE NIC. All nodes ran Linux kernel 4.15 on Ubuntu 18.04 and the Luminous release (v12.2.11) of Ceph. We used the default Ceph configuration parameters and focused on write performance improvements because most BlueStore optimizations affect writes.

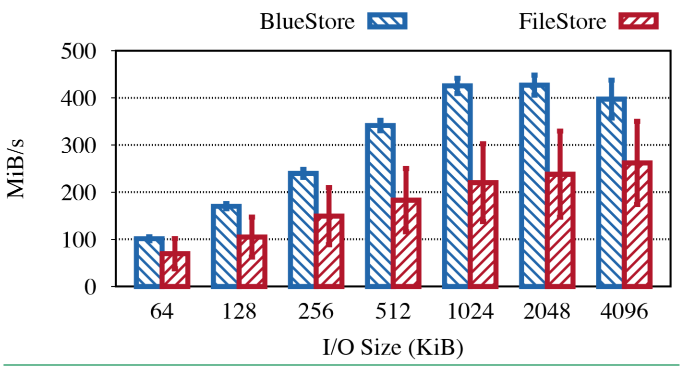

Figure 3: Throughput of steady state object writes to RADOS on a 16-node all-HDD cluster with different sizes using 128 threads. Compared to FileStore, the throughput is 50–100% greater on BlueStore and has a significantly lower variance.

Figure 3 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 and consistency overhead.

Figure 4 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.

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. 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.

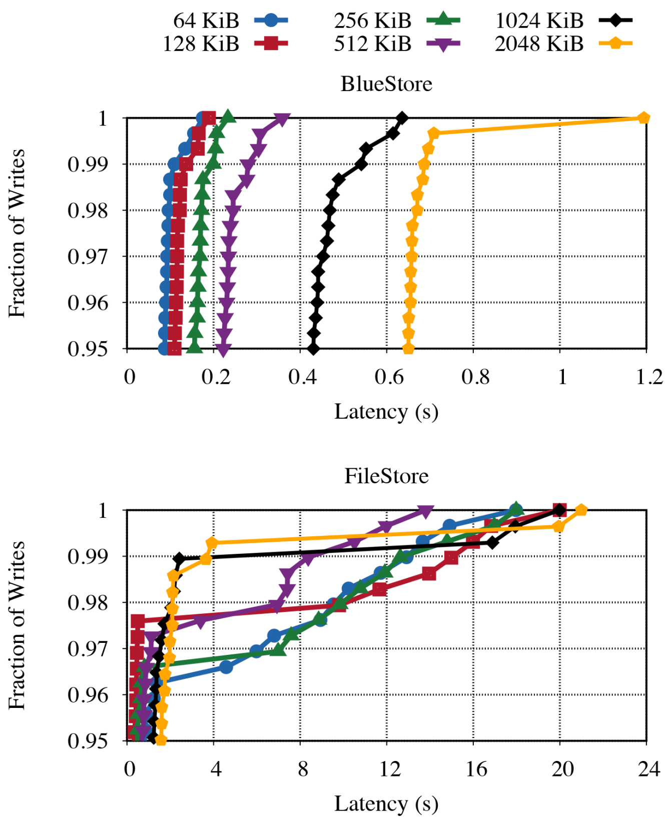

Figure 4: 95th and above percentile latencies of object writes to RADOS on a 16-node all-HDD cluster with different sizes using 128 threads. BlueStore (top graph) has an order of magnitude lower tail latency than FileStore (bottom graph).

Conclusion

Distributed file system developers conventionally adopt local file systems as their storage back end. They then try to fit the general-purpose file system abstractions to their needs, incurring significant accidental complexity [4]. At the core of this convention lies the belief that developing a storage back end from scratch is an arduous process, akin to developing a new file system that takes a decade to mature.

Our paper, relying on the Ceph team's experience, showed this belief to be inaccurate. Furthermore, we found that developing a *special-purpose*, userspace storage back end from scratch (1) reclaimed the significant performance left on the table when building a back end on a general-purpose file system; (2) made it possible to adopt novel, backward-incompatible storage hardware; and (3) enabled new features by gaining complete control of the I/O stack. We hope that this experience paper will initiate discussions among storage practitioners and researchers on fresh approaches to designing distributed file systems and their storage back ends.

References

[1] A. Aghayev, S. Weil, G. Ganger, and G. Amvrosiadis, "Reconciling LSM-Trees with Modern Hard Drives Using BlueFS," Technical Report CMU-PDL-19-102, CMU Parallel Data Laboratory, April 2019.

[2] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File Systems Unfit as Distributed Storage Back Ends: Lessons from 10 Years of Ceph Evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pp. 353–369.

[3] M. Bjørling, "From Open-Channel SSDs to Zoned Namespaces," 2019 Linux Storage and Filesystems Conference (Vault '19), USENIX Association, 2019: https://www.usenix.org/conference/vault19/presentation/bjorling.

[4] F. P. Brooks Jr., "No Silver Bullet—Essence and Accident in Software Engineering," in *Proceedings of the IFIP 10th World Computing Conference*, 1986, pp. 1069–1076.

[5] D. Chinner, "SMR Layout Optimization for XFS," March 2015: http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf.

[6] J. Edge, "Filesystem Support for SMR Devices," March 2015: https://lwn.net/Articles/637035/.

[7] C. Hellwig, "XFS: The Big Storage File System for Linux," *;login:*, vol. 34, no. 5 (October 2009): https://www.usenix.org/system/files/login/articles/140-hellwig.pdf.

[8] Facebook Inc., RocksDB Merge Operator, 2019: https://github.com/facebook/rocksdb/wiki/Merge-Operator-Implementation.

[9] A. Palmer, "SMRFFS-EXT4—SMR Friendly File System," 2015: https://github.com/Seagate/SMR_FS-EXT4.

[10] Wikipedia, Cache flushing: https://en.wikipedia.org/wiki/Disk_buffer#Cache_flushing.