Ceph: A Scalable, High-Performance Distributed File System

Sage A. Weil Scott A. Brandt Ethan L. Miller Darrell D. E. Long
Carlos Maltzahn
University of California, Santa Cruz
{*sage, scott, elm, darrell, carlosm*}*@cs.ucsc.edu*

Abstract

We have developed Ceph, a distributed file system that provides excellent performance, reliability, and scalability. Ceph maximizes the separation between data and metadata management by replacing allocation tables with a pseudo-random data distribution function (CRUSH) designed for heterogeneous and dynamic clusters of unreliable object storage devices (OSDs). We leverage device intelligence by distributing data replication, failure detection and recovery to semi-autonomous OSDs running a specialized local object file system. A dynamic distributed metadata cluster provides extremely efficient metadata management and seamlessly adapts to a wide range of general purpose and scientific computing file system workloads. Performance measurements under a variety of workloads show that Ceph has excellent I/O performance and scalable metadata management, supporting more than 250,000 metadata operations per second.

1 Introduction

System designers have long sought to improve the performance of file systems, which have proved critical to the overall performance of an exceedingly broad class of applications. The scientific and high-performance computing communities in particular have driven advances in the performance and scalability of distributed storage systems, typically predicting more general purpose needs by a few years. Traditional solutions, exemplified by NFS [18], provide a straightforward model in which a server exports a file system hierarchy that clients can map into their local name space. Although widely used, the centralization inherent in the client/server model has proven a significant obstacle to scalable performance.

More recent distributed file systems have adopted architectures based on object-based storage, in which conventional hard disks are replaced with intelligent object storage devices (OSDs) which combine a CPU, network interface, and local cache with an underlying disk or RAID [3, 6, 7, 28, 31]. OSDs replace the traditional block-level interface with one in which clients can read or write byte ranges to much larger (and often variably sized) named objects, distributing low-level block allocation decisions to the devices themselves. Clients typically interact with a metadata server (MDS) to perform metadata operations (*open*, *rename*), while communicating directly with OSDs to perform file I/O (reads and writes), significantly improving overall scalability.

Systems adopting this model continue to suffer from scalability limitations due to little or no distribution of the metadata workload. Continued reliance on traditional file system principles like allocation lists and inode tables and a reluctance to delegate intelligence to the OSDs have further limited scalability and performance, and increased the cost of reliability.

We present Ceph, a distributed file system that provides excellent performance and reliability while promising unparalleled scalability. Our architecture is based on the assumption that systems at the petabyte scale are inherently dynamic: large systems are inevitably built incrementally, node failures are the norm rather than the exception, and the quality and character of workloads are constantly shifting over time.

Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with generating functions. This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph utilizes a highly adaptive distributed metadata cluster architecture that dramatically improves the scalability of metadata access, and with it, the scalability of the entire system. We discuss the goals and workload assumptions motivating our choices in the design of the architecture, analyze their impact on system scalability and performance, and relate our experiences in implementing a functional system prototype.

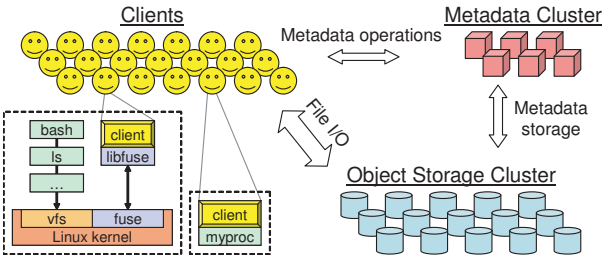


Figure 1: System architecture. Clients perform file I/O by communicating directly with OSDs. Each process can either link directly to a client instance or interact with a mounted file system.

2 System Overview

The Ceph file system has three main components: the client, each instance of which exposes a near-POSIX file system interface to a host or process; a cluster of OSDs, which collectively stores all data and metadata; and a metadata server cluster, which manages the namespace (file names and directories) while coordinating security, consistency and coherence (see Figure 1). We say the Ceph interface is near-POSIX because we find it appropriate to extend the interface and selectively relax consistency semantics in order to better align with the needs of applications and to improve system performance.

The primary goals of the architecture are scalability (to hundreds of petabytes and beyond), performance, and reliability. Scalability is considered in a variety of dimensions, including the overall storage capacity and throughput of the system, and performance in terms of individual clients, directories, or files. Our target workload may include such extreme cases as tens or hundreds of thousands of hosts concurrently reading from or writing to the same file or creating files in the same directory. Such scenarios, common in scientific applications running on supercomputing clusters, are increasingly indicative of tomorrow's general purpose workloads. More importantly, we recognize that distributed file system workloads are inherently dynamic, with significant variation in data and metadata access as active applications and data sets change over time. Ceph directly addresses the issue of scalability while simultaneously achieving high performance, reliability and availability through three fundamental design features: decoupled data and metadata, dynamic distributed metadata management, and reliable autonomic distributed object storage.

Decoupled Data and Metadata—Ceph maximizes the separation of file metadata management from the storage of file data. Metadata operations (*open*, *rename*, etc.) are collectively managed by a metadata server cluster, while clients interact directly with OSDs to perform file I/O (reads and writes). Object-based storage has long promised to improve the scalability of file systems by delegating low-level block allocation decisions to individual devices. However, in contrast to existing object-based file systems [3, 6, 7, 28] which replace long per-file block lists with shorter object lists, Ceph eliminates allocation lists entirely. Instead, file data is striped onto predictably named objects, while a special-purpose data distribution function called CRUSH [**?**] assigns objects to storage devices. This allows any party to calculate (rather than look up) the name and location of objects comprising a file's contents, eliminating the need to maintain and distribute object lists, simplifying the design of the system, and reducing the metadata cluster workload.

Dynamic Distributed Metadata Management—Because file system metadata operations make up as much as half of typical file system workloads [20], effective metadata management is critical to overall system performance. Ceph utilizes a novel metadata cluster architecture based on Dynamic Subtree Partitioning [26] that adaptively and intelligently distributes responsibility for managing the file system directory hierarchy among tens or even hundreds of MDSs. A (dynamic) hierarchical partition preserves locality in each MDS's workload, facilitating efficient updates and aggressive prefetching to improve performance for common workloads. Significantly, the workload distribution among metadata servers is based entirely on current access patterns, allowing Ceph to effectively utilize available MDS resources under any workload and achieve near-linear scaling in the number of MDSs.

Reliable Autonomic Distributed Object Storage—Large systems composed of many thousands of devices are inherently dynamic: they are built incrementally, they grow and contract as new storage is deployed and old devices are decommissioned, device failures are frequent and expected, and large volumes of data are created, moved, and deleted. All of these factors require that the distribution of data evolve to effectively utilize available resources and maintain the desired level of data replication. Ceph delegates responsibility for data migration, replication, failure detection, and failure recovery to the cluster of OSDs that store the data, while at a high level, OSDs collectively provide a single logical object store to clients and metadata servers. This approach allows Ceph to more effectively leverage the intelligence (CPU and memory) present on each OSD to achieve reliable, highly available object storage with linear scaling.

We describe the operation of the Ceph client, metadata server cluster, and distributed object store, and how they are affected by the critical features of our architecture. We also describe the status of our prototype.

3 Client Operation

We introduce the overall operation of Ceph's components and their interaction with applications by describ-

ing Ceph's client operation. The Ceph client runs on each host executing application code and exposes a file system interface to applications. In the Ceph prototype, the client code runs entirely in user space and can be accessed either by linking to it directly or as a mounted file system via FUSE [**?**] (a user-space file system interface). Each client maintains its own file data cache, independent of the kernel page or buffer caches, making it accessible to applications that link to the client directly.

3.1 File I/O and Capabilities

When a process opens a file, the client sends a request to the MDS cluster. An MDS traverses the file system hierarchy to translate the file name into the *file inode*, which includes a unique inode number, the file owner, mode, size, and other per-file metadata. If the file exists and access is granted, the MDS returns the inode number, file size, and information about the striping strategy used to map file data into objects. The MDS may also issue the client a *capability* (if it does not already have one) specifying which operations are permitted. Capabilities currently include four bits controlling the client's ability to read, cache reads, write, and buffer writes. In the future, capabilities will include security keys allowing clients to prove to OSDs that they are authorized to read or write data [**?**, 17] (the prototype currently trusts all clients). Subsequent MDS involvement in file I/O is limited to managing capabilities to preserve file consistency and achieve proper semantics.

Ceph generalizes a range of striping strategies to map file data onto a sequence of objects. To avoid any need for file allocation metadata, object names simply combine the file inode number and the stripe number. Object replicas are then assigned to OSDs using CRUSH, a globally known mapping function (described in Section 5.1). For example, if one or more clients open a file for read access, an MDS grants them the capability to read and cache file content. Armed with the inode number, layout, and file size, the clients can name and locate all objects containing file data and read directly from the OSD cluster. Any objects or byte ranges that don't exist are defined to be file "holes," or zeros. Similarly, if a client opens a file for writing, it is granted the capability to write with buffering, and any data it generates at any offset in the file is simply written to the appropriate object on the appropriate OSD. The client relinquishes the capability on file close and provides the MDS with the new file size (the largest offset written), which redefines the set of objects that (may) exist and contain file data.

3.2 Client Synchronization

POSIX semantics sensibly require that reads reflect any data previously written, and that writes are atomic (*i. e.*, the result of overlapping, concurrent writes will reflect a particular order of occurrence). When a file is opened by multiple clients with either multiple writers or a mix of readers and writers, the MDS will revoke any previously issued read caching and write buffering capabilities, forcing client I/O for that file to be synchronous. That is, each application read or write operation will block until it is acknowledged by the OSD, effectively placing the burden of update serialization and synchronization with the OSD storing each object. When writes span object boundaries, clients acquire exclusive locks on the affected objects (granted by their respective OSDs), and immediately submit the write and unlock operations to achieve the desired serialization. Object locks are similarly used to mask latency for large writes by acquiring locks and flushing data asynchronously.

Not surprisingly, synchronous I/O can be a performance killer for applications, particularly those doing small reads or writes, due to the latency penalty—at least one round-trip to the OSD. Although read-write sharing is relatively rare in general-purpose workloads [20], it is more common in scientific computing applications [24], where performance is often critical. For this reason, it is often desirable to relax consistency at the expense of strict standards conformance in situations where applications do not rely on it. Although Ceph supports such relaxation via a global switch, and many other distributed file systems punt on this issue [18], this is an imprecise and unsatisfying solution: either performance suffers, or consistency is lost system-wide.

For precisely this reason, a set of high performance computing extensions to the POSIX I/O interface have been proposed by the high-performance computing (HPC) community [27], a subset of which are implemented by Ceph. Most notably, these include an `O_LAZY` flag for *open* that allows applications to explicitly relax the usual coherency requirements for a shared-write file. Performance-conscious applications which manage their own consistency (*e. g.*, by writing to different parts of the same file, a common pattern in HPC workloads [24]) are then allowed to buffer writes or cache reads when I/O would otherwise be performed synchronously. If desired, applications can then explicitly synchronize with two additional calls: *lazyio_propagate* will flush a given byte range to the object store, while *lazyio_synchronize* will ensure that the effects of previous propagations are reflected in any subsequent reads. The Ceph synchronization model thus retains its simplicity by providing correct read-write and shared-write semantics between clients via synchronous I/O, and extending the application interface to relax consistency for performance conscious distributed applications.

3.3 Namespace Operations

Client interaction with the file system namespace is managed by the metadata server cluster. Both read operations

(*e. g.*, *readdir*, *stat*) and updates (*e. g.*, *unlink*, *chmod*) are synchronously applied by the MDS to ensure serialization, consistency, correct security, and safety. For simplicity, no metadata locks or leases are issued to clients. For HPC workloads in particular, callbacks offer minimal upside at a high potential cost in complexity.

Instead, Ceph optimizes for the most common metadata access scenarios. A *readdir* followed by a *stat* of each file (*e. g.*, `ls -l`) is an extremely common access pattern and notorious performance killer in large directories. A *readdir* in Ceph requires only a single MDS request, which fetches the entire directory, including inode contents. By default, if a *readdir* is immediately followed by one or more *stat*s, the briefly cached information is returned; otherwise it is discarded. Although this relaxes coherence slightly in that an intervening inode modification may go unnoticed, we gladly make this trade for vastly improved performance. This behavior is explicitly captured by the *readdirplus* [27] extension, which returns *lstat* results with directory entries (as some OS-specific implementations of *getdir* already do).

Ceph could allow consistency to be further relaxed by caching metadata longer, much like earlier versions of NFS, which typically cache for 30 seconds. However, this approach breaks coherency in a way that is often critical to applications, such as those using *stat* to determine if a file has been updated—they either behave incorrectly, or end up waiting for old cached values to time out.

We opt instead to again provide correct behavior and extend the interface in instances where it adversely affects performance. This choice is most clearly illustrated by a *stat* operation on a file currently opened by multiple clients for writing. In order to return a correct file size and modification time, the MDS revokes any write capabilities to momentarily stop updates and collect up-to-date size and mtime values from all writers. The highest values are returned with the *stat* reply, and capabilities are reissued to allow further progress. Although stopping multiple writers may seem drastic, it is necessary to ensure proper serializability. (For a single writer, a correct value can be retrieved from the writing client without interrupting progress.) Applications for which coherent behavior is unnecesssary—victims of a POSIX interface that doesn't align with their needs—can use *statlite* [27], which takes a bit mask specifying which inode fields are not required to be coherent.

4 Dynamically Distributed Metadata

Metadata operations often make up as much as half of file system workloads [20] and lie in the critical path, making the MDS cluster critical to overall performance. Metadata management also presents a critical scaling challenge in distributed file systems: although capacity and aggregate I/O rates can scale almost arbitrarily with the addition of more storage devices, metadata operations involve a greater degree of interdependence that makes scalable consistency and coherence management more difficult.

File and directory metadata in Ceph is very small, consisting almost entirely of directory entries (file names) and inodes (80 bytes). Unlike conventional file systems, no file allocation metadata is necessary—object names are constructed using the inode number, and distributed to OSDs using CRUSH. This simplifies the metadata workload and allows our MDS to efficiently manage a very large working set of files, independent of file sizes. Our design further seeks to minimize metadata related disk I/O through the use of a two-tiered storage strategy, and to maximize locality and cache efficiency with Dynamic Subtree Partitioning [26].

4.1 Metadata Storage

Although the MDS cluster aims to satisfy most requests from its in-memory cache, metadata updates must be committed to disk for safety. A set of large, bounded, lazily flushed journals allows each MDS to quickly stream its updated metadata to the OSD cluster in an efficient and distributed manner. The per-MDS journals, each many hundreds of megabytes, also absorb repetitive metadata updates (common to most workloads) such that when old journal entries are eventually flushed to long-term storage, many are already rendered obsolete. Although MDS recovery is not yet implemented by our prototype, the journals are designed such that in the event of an MDS failure, another node can quickly rescan the journal to recover the critical contents of the failed node's in-memory cache (for quick startup) and in doing so recover the file system state.

This strategy provides the best of both worlds: streaming updates to disk in an efficient (sequential) fashion, and a vastly reduced re-write workload, allowing the long-term on-disk storage layout to be optimized for future read access. In particular, inodes are embedded directly within directories, allowing the MDS to prefetch entire directories with a single OSD read request and exploit the high degree of directory locality present in most workloads [20]. Each directory's content is written to the OSD cluster using the same striping and distribution strategy as metadata journals and file data. Inode numbers are allocated in ranges to metadata servers and considered immutable in our prototype, although in the future they could be trivially reclaimed on file deletion. An auxiliary *anchor table* [25] keeps the rare inode with multiple hard links globally addressable by inode number—all without encumbering the overwhelmingly common case of singly-linked files with an enormous, sparsely populated and cumbersome inode table.

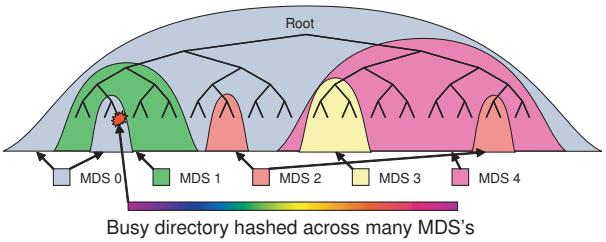


Figure 2: Ceph dynamically maps subtrees of the directory hierarchy to metadata servers based on the current workload. Individual directories are hashed across multiple nodes only when they become hot spots.

4.2 Dynamic Subtree Partitioning

Our primary-copy caching strategy makes a single authoritative MDS responsible for managing cache coherence and serializing updates for any given piece of metadata. While most existing distributed file systems employ some form of static subtree-based partitioning to delegate this authority (usually forcing an administrator to carve the dataset into smaller static "volumes"), some recent and experimental file systems have used hash functions to distribute directory and file metadata [3], effectively sacrificing locality for load distribution. Both approaches have critical limitations: static subtree partitioning fails to cope with dynamic workloads and data sets, while hashing destroys metadata locality and critical opportunities for efficient metadata prefetching and storage.

Ceph's MDS cluster is based on a dynamic subtree partitioning strategy [26] that adaptively distributes cached metadata hierarchically across a set of nodes, as illustrated in Figure 2. Each MDS measures the popularity of metadata within the directory hierarchy using counters with an exponential time decay. Any operation increments the counter on the affected inode and all of its ancestors up to the root directory, providing each MDS with a weighted tree describing the recent load distribution. MDS load values are periodically compared, and appropriately-sized subtrees of the directory hierarchy are migrated to keep the workload evenly distributed. The combination of shared long-term storage and carefully constructed namespace locks allows such migrations to proceed by transferring the appropriate contents of the in-memory cache to the new authority, with minimal impact on coherence locks or client capabilities. Imported metadata is written to the new MDS's journal for safety, while additional journal entries on both ends ensure that the transfer of authority is invulnerable to intervening failures (similar to a two-phase commit). The resulting subtree-based partition is kept coarse to minimize prefix replication overhead and to preserve locality.

When metadata is replicated across multiple MDS nodes, inode contents are separated into three groups, each with different consistency semantics: security (owner, mode), file (size, mtime), and immutable (inode number, ctime, layout). While immutable fields never change, security and file locks are governed by independent finite state machines, each with a different set of states and transitions designed to accommodate different access and update patterns while minimizing lock contention. For example, owner and mode are required for the security check during path traversal but rarely change, requiring very few states, while the file lock reflects a wider range of client access modes as it controls an MDS's ability to issue client capabilities.

4.3 Traffic Control

Partitioning the directory hierarchy across multiple nodes can balance a broad range of workloads, but cannot always cope with hot spots or flash crowds, where many clients access the same directory or file. Ceph uses its knowledge of metadata popularity to provide a wide distribution for hot spots only when needed and without incurring the associated overhead and loss of directory locality in the general case. The contents of heavily read directories (*e. g.*, many opens) are selectively replicated across multiple nodes to distribute load. Directories that are particularly large or experiencing a heavy write workload (*e. g.*, many file creations) have their contents hashed by file name across the cluster, achieving a balanced distribution at the expense of directory locality. This adaptive approach allows Ceph to encompass a broad spectrum of partition granularities, capturing the benefits of both coarse and fine partitions in the specific circumstances and portions of the file system where those strategies are most effective.

Every MDS response provides the client with updated information about the authority and any replication of the relevant inode and its ancestors, allowing clients to learn the metadata partition for the parts of the file system with which they interact. Future metadata operations are directed at the authority (for updates) or a random replica (for reads) based on the deepest known prefix of a given path. Normally clients learn the locations of unpopular (unreplicated) metadata and are able to contact the appropriate MDS directly. Clients accessing popular metadata, however, are told the metadata reside either on different or multiple MDS nodes, effectively bounding the number of clients believing any particular piece of metadata resides on any particular MDS, dispersing potential hot spots and flash crowds before they occur.

5 Distributed Object Storage

From a high level, Ceph clients and metadata servers view the object storage cluster (possibly tens or hundreds of thousands of OSDs) as a single logical object store and namespace. Ceph's Reliable Autonomic Distributed Object Store (RADOS) achieves linear scaling in both

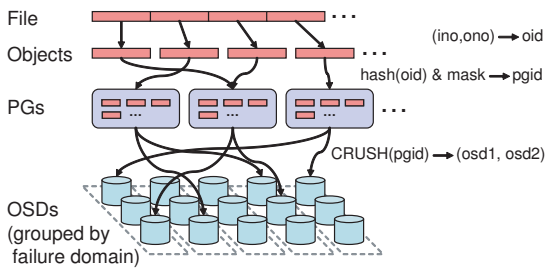


Figure 3: Files are striped across many objects, grouped into *placement groups* (PGs), and distributed to OSDs via CRUSH, a specialized replica placement function.

capacity and aggregate performance by delegating management of object replication, cluster expansion, failure detection and recovery to OSDs in a distributed fashion.

5.1 Data Distribution with CRUSH

Ceph must distribute petabytes of data among an evolving cluster of thousands of storage devices such that device storage and bandwidth resources are effectively utilized. In order to avoid imbalance (*e. g.*, recently deployed devices mostly idle or empty) or load asymmetries (*e. g.*, new, hot data on new devices only), we adopt a strategy that distributes new data randomly, migrates a random subsample of existing data to new devices, and uniformly redistributes data from removed devices. This stochastic approach is robust in that it performs equally well under any potential workload.

Ceph first maps objects into *placement groups* (PGs) using a simple hash function, with an adjustable bit mask to control the number of PGs. We choose a value that gives each OSD on the order of 100 PGs to balance variance in OSD utilizations with the amount of replication-related metadata maintained by each OSD. Placement groups are then assigned to OSDs using CRUSH (Controlled Replication Under Scalable Hashing) [**?**], a pseudo-random data distribution function that efficiently maps each PG to an ordered list of OSDs upon which to store object replicas. This differs from conventional approaches (including other object-based file systems) in that data placement does not rely on any block or object list metadata. To locate any object, CRUSH requires only the placement group and an *OSD cluster map*: a compact, hierarchical description of the devices comprising the storage cluster. This approach has two key advantages: first, it is completely distributed such that any party (client, OSD, or MDS) can independently calculate the location of any object; and second, the map is infrequently updated, virtually eliminating any exchange of distribution-related metadata. In doing so, CRUSH simultaneously solves both the data distribution problem ("where should I store data") and the data location problem ("where did I store data"). By design, small changes to the storage cluster have little impact on existing PG mappings, minimizing data migration due to device failures or cluster expansion.

The cluster map hierarchy is structured to align with the clusters physical or logical composition and potential sources of failure. For instance, one might form a four-level hierarchy for an installation consisting of shelves full of OSDs, rack cabinets full of shelves, and rows of cabinets. Each OSD also has a weight value to control the relative amount of data it is assigned. CRUSH maps PGs onto OSDs based on *placement rules*, which define the level of replication and any constraints on placement. For example, one might replicate each PG on three OSDs, all situated in the same row (to limit inter-row replication traffic) but separated into different cabinets (to minimize exposure to a power circuit or edge switch failure). The cluster map also includes a list of down or inactive devices and an epoch number, which is incremented each time the map changes. All OSD requests are tagged with the client's map epoch, such that all parties can agree on the current distribution of data. Incremental map updates are shared between cooperating OSDs, and piggyback on OSD replies if the client's map is out of date.

5.2 Replication

In contrast to systems like Lustre [3], which assume one can construct sufficiently reliable OSDs using mechanisms like RAID or fail-over on a SAN, we assume that in a petabyte or exabyte system failure will be the norm rather than the exception, and at any point in time several OSDs are likely to be inoperable. To maintain system availability and ensure data safety in a scalable fashion, RADOS manages its own replication of data using a variant of primary-copy replication [**?**], while taking steps to minimize the impact on performance.

Data is replicated in terms of placement groups, each of which is mapped to an ordered list of n OSDs (for n-way replication). Clients send all writes to the first non-failed OSD in an object's PG (the *primary*), which assigns a new version number for the object and PG and forwards the write to any additional *replica* OSDs. After each replica has applied the update and responded to the primary, the primary applies the update locally and the write is acknowledged to the client. Reads are directed at the primary. This approach spares the client of any of the complexity surrounding synchronization or serialization between replicas, which can be onerous in the presence of other writers or failure recovery. It also shifts the bandwidth consumed by replication from the client to the OSD cluster's internal network, where we expect greater resources to be available. Intervening replica OSD failures are ignored, as any subsequent recovery (see Section 5.5) will reliably restore replica consistency.

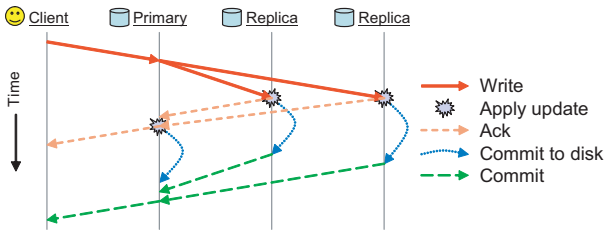


Figure 4: RADOS responds with an *ack* after the write has been applied to the buffer caches on all OSDs replicating the object. Only after it has been safely committed to disk is a final *commit* notification sent to the client.

5.3 Data Safety

In distributed storage systems, there are essentially two reasons why data is written to shared storage. First, clients are interested in making their updates visible to other clients. This should be quick: writes should be visible as soon as possible, particularly when multiple writers or mixed readers and writers force clients to operate synchronously. Second, clients are interested in knowing definitively that the data they've written is safely replicated, on disk, and will survive power or other failures. RADOS disassociates synchronization from safety when acknowledging updates, allowing Ceph to realize both low-latency updates for efficient application synchronization and well-defined data safety semantics.

Figure 4 illustrates the messages sent during an object write. The primary forwards the update to replicas, and replies with an *ack* after it is applied to all OSDs' in-memory buffer caches, allowing synchronous POSIX calls on the client to return. A final *commit* is sent (perhaps many seconds later) when data is safely committed to disk. We send the *ack* to the client only after the update is fully replicated to seamlessly tolerate the failure of any single OSD, even though this increases client latency. By default, clients also buffer writes until they commit to avoid data loss in the event of a simultaneous power loss to all OSDs in the placement group. When recovering in such cases, RADOS allows the replay of previously acknowledged (and thus ordered) updates for a fixed interval before new updates are accepted.

5.4 Failure Detection

Timely failure detection is critical to maintaining data safety, but can become difficult as a cluster scales to many thousands of devices. For certain failures, such as disk errors or corrupted data, OSDs can self-report. Failures that make an OSD unreachable on the network, however, require active monitoring, which RADOS distributes by having each OSD monitor those peers with which it shares PGs. In most cases, existing replication traffic serves as a passive confirmation of liveness, with no additional communication overhead. If an OSD has not heard from a peer recently, an explicit ping is sent.

RADOS considers two dimensions of OSD liveness: whether the OSD is reachable, and whether it is assigned data by CRUSH. An unresponsive OSD is initially marked *down*, and any primary responsibilities (update serialization, replication) temporarily pass to the next OSD in each of its placement groups. If the OSD does not quickly recover, it is marked *out* of the data distribution, and another OSD joins each PG to re-replicate its contents. Clients which have pending operations with a failed OSD simply resubmit to the new primary.

Because a wide variety of network anomalies may cause intermittent lapses in OSD connectivity, a small cluster of monitors collects failure reports and filters out transient or systemic problems (like a network partition) centrally. Monitors (which are only partially implemented) use elections, active peer monitoring, short-term leases, and two-phase commits to collectively provide consistent and available access to the cluster map. When the map is updated to reflect any failures or recoveries, affected OSDs are provided incremental map updates, which then spread throughout the cluster by piggybacking on existing inter-OSD communication. Distributed detection allows fast detection without unduly burdening monitors, while resolving the occurrence of inconsistency with centralized arbitration. Most importantly, RADOS avoids initiating widespread data re-replication due to systemic problems by marking OSDs *down* but not *out* (*e. g.*, after a power loss to half of all OSDs).

5.5 Recovery and Cluster Updates

The OSD cluster map will change due to OSD failures, recoveries, and explicit cluster changes such as the deployment of new storage. Ceph handles all such changes in the same way. To facilitate fast recovery, OSDs maintain a version number for each object and a log of recent changes (names and versions of updated or deleted objects) for each PG (similar to the replication logs in Harp [12]) .

When an active OSD receives an updated cluster map, it iterates over all locally stored placement groups and calculates the CRUSH mapping to determine which ones it is responsible for, either as a primary or replica. If a PG's membership has changed, or if the OSD has just booted, the OSD must *peer* with the PG's other OSDs. For replicated PGs, the OSD provides the primary with its current PG version number. If the OSD is the primary for the PG, it collects current (and former) replicas' PG versions. If the primary lacks the most recent PG state, it retrieves the log of recent PG changes (or a complete content summary, if needed) from current or prior OSDs in the PG in order to determine the correct (most recent) PG contents. The primary then sends each replica an incremental log update (or complete content summary, if needed), such that all parties know what the PG contents

should be, even if their locally stored object set may not match. Only after the primary determines the correct PG state and shares it with any replicas is I/O to objects in the PG permitted. OSDs are then independently responsible for retrieving missing or outdated objects from their peers. If an OSD receives a request for a stale or missing object, it delays processing and moves that object to the front of the recovery queue.

For example, suppose `osd1` crashes and is marked *down*, and `osd2` takes over as primary for `pgA`. If `osd1` recovers, it will request the latest map on boot, and a monitor will mark it as *up*. When `osd2` receives the resulting map update, it will realize it is no longer primary for `pgA` and send the `pgA` version number to `osd1`. `osd1` will retrieve recent `pgA` log entries from `osd2`, tell `osd2` its contents are current, and then begin processing requests while any updated objects are recovered in the background.

Because failure recovery is driven entirely by individual OSDs, each PG affected by a failed OSD will recover in parallel to (very likely) different replacement OSDs. This approach, based on the Fast Recovery Mechanism (FaRM) [33], decreases recovery times and improves overall data safety.

5.6 Object Storage with EBOFS

Although a variety of distributed file systems use local file systems like ext3 to manage low-level storage [3, 11], we found their interface and performance to be poorly suited for object workloads [24]. The existing kernel interface limits our ability to understand when object updates are safely committed on disk. Synchronous writes or journaling provide the desired safety, but only with a heavy latency and performance penalty. More importantly, the POSIX interface fails to support atomic data and metadata (*e. g.*, attribute) update transactions, which are important for maintaining RADOS consistency.

Instead, each Ceph OSD manages its local object storage with EBOFS, an Extent and B-tree based Object File System. Implementing EBOFS entirely in user space and interacting directly with a raw block device allows us to define our own low-level object storage interface and update semantics, which separate update serialization (for synchronization) from on-disk commits (for safety). EBOFS supports atomic transactions (*e. g.*, writes and attribute updates on multiple objects), and update functions return when the in-memory caches are updated, while providing asynchronous notification of commits.

A user space approach, aside from providing greater flexibility and easier implementation, also avoids cumbersome interaction with the Linux VFS and page cache, both of which were designed for a different interface and workload. While most kernel file systems lazily flush updates to disk after some time interval, EBOFS aggressively schedules disk writes, and opts instead to cancel pending I/O operations when subsequent updates render them superfluous. This provides our low-level disk scheduler with longer I/O queues and a corresponding increase in scheduling efficiency. A user-space scheduler also makes it easier to eventually prioritize workloads (*e. g.*, client I/O versus recovery) or provide quality of service guarantees [32].

Central to the EBOFS design is a robust, flexible, and fully integrated B-tree service that is used to locate objects on disk, manage block allocation, and index collections (placement groups). Block allocation is conducted in terms of extents—start and length pairs—instead of block lists, keeping metadata compact. Free block extents on disk are binned by size and sorted by location, allowing EBOFS to quickly locate free space near the write position or related data on disk, while also limiting long-term fragmentation. With the exception of per-object block allocation information, all metadata is kept in memory for performance and simplicity (it is quite small, even for large volumes). Finally, EBOFS aggressively performs copy-on-write: with the exception of superblock updates, data is always written to unallocated regions of disk.

6 Performance and Scalability Evaluation

We evaluate our prototype under a range of microbenchmarks to demonstrate its performance, reliability, and scalability. In all tests, clients, OSDs, and MDSs are user processes running on a dual-processor Linux cluster with SCSI disks and communicating using TCP. In general, each OSD or MDS runs on its own host, while tens or hundreds of client instances may share the same host while generating workload.

6.1 Data Performance

EBOFS provides superior performance and safety semantics, while the balanced distribution of data generated by CRUSH and the delegation of replication and failure recovery allow aggregate I/O performance to scale with the size of the OSD cluster.

6.1.1 OSD Throughput

We begin by measuring the I/O performance of a 14-node cluster of OSDs. Figure 5 shows per-OSD throughput (y) with varying write sizes (x) and replication. Workload is generated by 400 clients on 20 additional nodes. Performance is ultimately limited by the raw disk bandwidth (around 58 MB/sec), shown by the horizontal line. Replication doubles or triples disk I/O, reducing client data rates accordingly when the number of OSDs is fixed.

Figure 6 compares the performance of EBOFS to that of general-purpose file systems (ext3, ReiserFS, XFS) in handling a Ceph workload. Clients synchronously

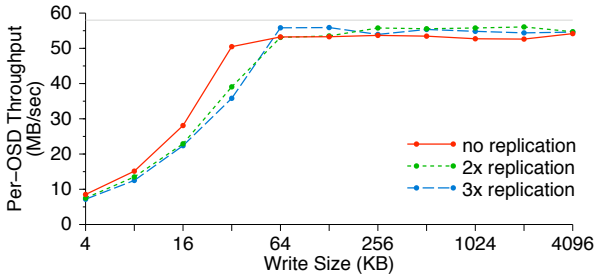


Figure 5: Per-OSD write performance. The horizontal line indicates the upper limit imposed by the physical disk. Replication has minimal impact on OSD throughput, although if the number of OSDs is fixed, n-way replication reduces total *effective* throughput by a factor of n because replicated data must be written to n OSDs.

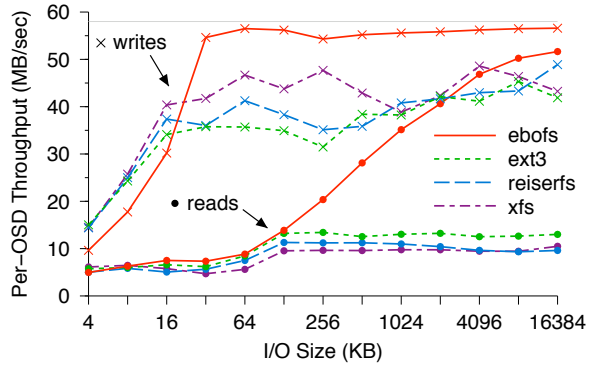


Figure 6: Performance of EBOFS compared to general-purpose file systems. Although small writes suffer from coarse locking in our prototype, EBOFS nearly saturates the disk for writes larger than 32 KB. Since EBOFS lays out data in large extents when it is written in large increments, it has significantly better read performance.

write out large files, striped over 16 MB objects, and read them back again. Although small read and write performance in EBOFS suffers from coarse threading and locking, EBOFS very nearly saturates the available disk bandwidth for writes sizes larger than 32 KB, and significantly outperforms the others for read workloads because data is laid out in extents on disk that match the write sizes—even when they are very large. Performance was measured using a fresh file system. Experience with an earlier EBOFS design suggests it will experience significantly lower fragmentation than ext3, but we have not yet evaluated the current implementation on an aged file system. In any case, we expect the performance of EBOFS after aging to be no worse than the others.

6.1.2 Write Latency

Figure 7 shows the synchronous write latency (y) for a single writer with varying write sizes (x) and replication. Because the primary OSD simultaneously retransmits updates to all replicas, small writes incur a minimal latency increase for more than two replicas. For larger writes, the cost of retransmission dominates; 1 MB writes (not shown) take 13 ms for one replica, and 2.5 times longer (33 ms) for three. Ceph clients partially mask this latency for synchronous writes over 128 KB by acquiring exclusive locks and then asynchronously flushing the data to disk. Alternatively, write-sharing applications can opt to use `O_LAZY`. With consistency thus relaxed, clients can buffer small writes and submit only large, asynchronous writes to OSDs; the only latency seen by applications will be due to clients which fill their caches waiting for data to flush to disk.

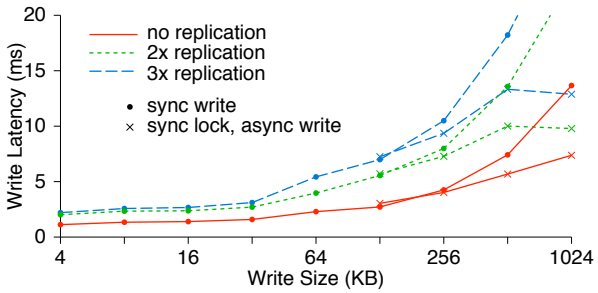


Figure 7: Write latency for varying write sizes and replication. More than two replicas incurs minimal additional cost for small writes because replicated updates occur concurrently. For large synchronous writes, transmission times dominate. Clients partially mask that latency for writes over 128 KB by acquiring exclusive locks and asynchronously flushing the data.

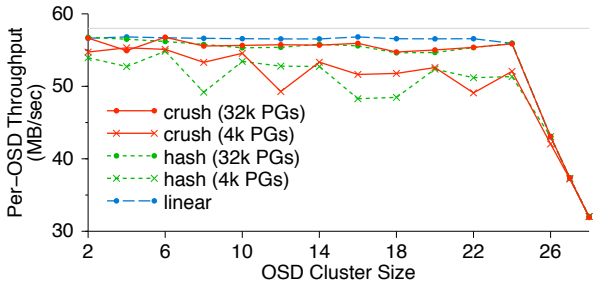


Figure 8: OSD write performance scales linearly with the size of the OSD cluster until the switch is saturated at 24 OSDs. CRUSH and hash performance improves when more PGs lower variance in OSD utilization.

6.1.3 Data Distribution and Scalability

Ceph's data performance scales nearly linearly in the number of OSDs. CRUSH distributes data pseudo-randomly such that OSD utilizations can be accurately modeled by a binomial or normal distribution—what one expects from a perfectly random process [**?**]. Variance

in utilizations decreases as the number of groups increases: for 100 placement groups per OSD the standard deviation is 10%; for 1000 groups it is 3%. Figure 8 shows per-OSD write throughput as the cluster scales using CRUSH, a simple hash function, and a linear striping strategy to distribute data in 4096 or 32768 PGs among available OSDs. Linear striping balances load perfectly for maximum throughput to provide a benchmark for comparison, but like a simple hash function, it fails to cope with device failures or other OSD cluster changes. Because data placement with CRUSH or a hash is stochastic, throughputs are lower with fewer PGs: greater variance in OSD utilizations causes request queue lengths to drift apart under our entangled client workload. Because devices can become overfilled or overutilized with small probability, dragging down performance, CRUSH can correct such situations by offloading any fraction of the allocation for OSDs specially marked in the cluster map. Unlike the hash and linear strategies, CRUSH also minimizes data migration under cluster expansion while maintaining a balanced distribution. CRUSH calculations are $O(\log n)$ (for a cluster of n OSDs) and take only tens of microseconds, allowing clusters to grow to hundreds of thousands of OSDs.

6.2 Metadata Performance

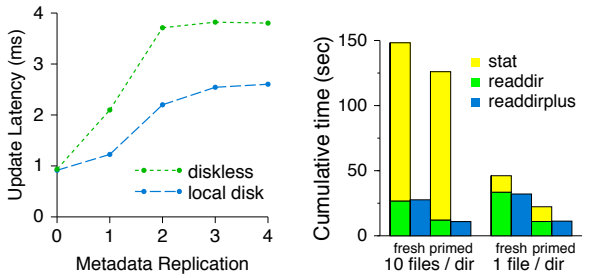
Ceph's MDS cluster offers enhanced POSIX semantics with excellent scalability. We measure performance via a partial workload lacking any data I/O; OSDs in these experiments are used solely for metadata storage.

6.2.1 Metadata Update Latency

We first consider the latency associated with metadata updates (*e. g.*, *mknod* or *mkdir*). A single client creates a series of files and directories which the MDS must synchronously journal to a cluster of OSDs for safety. We consider both a diskless MDS, where all metadata is stored in a shared OSD cluster, and one which also has a local disk serving as the primary OSD for its journal. Figure 9(a) shows the latency (y) associated with metadata updates in both cases with varying metadata replication (x) (where zero corresponds to no journaling at all). Journal entries are first written to the primary OSD and then replicated to any additional OSDs. With a local disk, the initial hop from the MDS to the (local) primary OSD takes minimal time, allowing update latencies for $2\times$ replication similar to $1\times$ in the diskless model. In both cases, more than two replicas incurs little additional latency because replicas update in parallel.

6.2.2 Metadata Read Latency

The behavior of metadata reads (*e. g.*, *readdir*, *stat*, *open*) is more complex. Figure 9(b) shows cumulative time (y) consumed by a client walking 10,000 nested directories with a *readdir* in each directory and a *stat* on each file. A



(a) Metadata update latency for an MDS with and without a local disk. Zero corresponds to no journaling.

(b) Cumulative time consumed during a file system walk.

Figure 9: Using a local disk lowers the write latency by avoiding the initial network round-trip. Reads benefit from caching, while *readdirplus* or relaxed consistency eliminate MDS interaction for *stat*s following *readdir*.

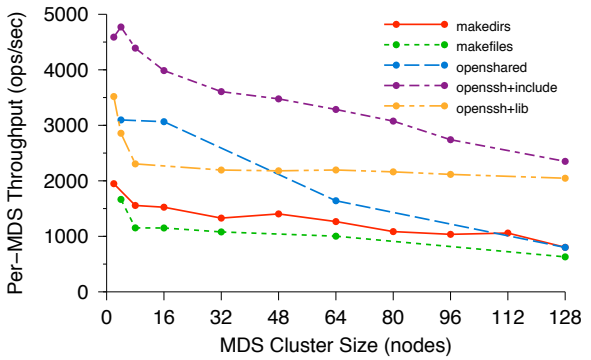


Figure 10: Per-MDS throughput under a variety of workloads and cluster sizes. As the cluster grows to 128 nodes, efficiency drops no more than 50% below perfect linear (horizontal) scaling for most workloads, allowing vastly improved performance over existing systems.

primed MDS cache reduces *readdir* times. Subsequent *stat*s are not affected, because inode contents are embedded in directories, allowing the full directory contents to be fetched into the MDS cache with a single OSD access. Ordinarily, cumulative *stat* times would dominate for larger directories. Subsequent MDS interaction can be eliminated by using *readdirplus*, which explicitly bundles *stat* and *readdir* results in a single operation, or by relaxing POSIX to allow *stat*s immediately following a *readdir* to be served from client caches (the default).

6.2.3 Metadata Scaling

We evaluate metadata scalability using a 430 node partition of the `alc` Linux cluster at Lawrence Livermore National Laboratory (LLNL). Figure 10 shows per-MDS throughput (y) as a function of MDS cluster size (x), such that a horizontal line represents perfect linear scaling. In the *makedirs* workload, each client creates a tree

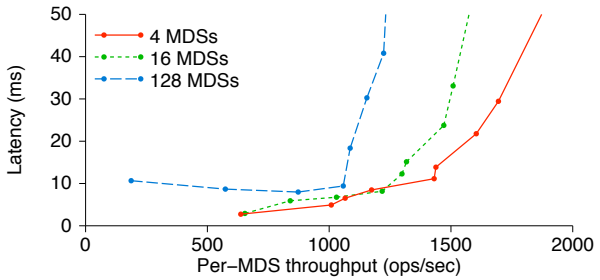


Figure 11: Average latency versus per-MDS throughput for different cluster sizes (*makedirs* workload).

of nested directories four levels deep, with ten files and subdirectories in each directory. Average MDS throughput drops from 2000 ops per MDS per second with a small cluster, to about 1000 ops per MDS per second (50% efficiency) with 128 MDSs (over 100,000 ops/sec total). In the *makefiles* workload, each client creates thousands of files in the same directory. When the high write levels are detected, Ceph hashes the shared directory and relaxes the directory's *mtime* coherence to distribute the workload across all MDS nodes. The *openshared* workload demonstrates read sharing by having each client repeatedly open and close ten shared files. In the *openssh* workloads, each client replays a captured file system trace of a compilation in a private directory. One variant uses a shared `/lib` for moderate sharing, while the other shares `/usr/include`, which is very heavily read. The *openshared* and *openssh+include* workloads have the heaviest read sharing and show the worst scaling behavior, we believe due to poor replica selection by clients. *openssh+lib* scales better than the trivially separable *makedirs* because it contains relatively few metadata modifications and little sharing. Although we believe that contention in the network or threading in our messaging layer further lowered performance for larger MDS clusters, our limited time with dedicated access to the large cluster prevented a more thorough investigation.

Figure 11 plots latency (y) versus per-MDS throughput (x) for a 4-, 16-, and 64-node MDS cluster under the *makedirs* workload. Larger clusters have imperfect load distributions, resulting in lower average per-MDS throughput (but, of course, much higher total throughput) and slightly higher latencies.

Despite imperfect linear scaling, a 128-node MDS cluster running our prototype can service more than a quarter million metadata operations per second (128 nodes at 2000 ops/sec). Because metadata transactions are independent of data I/O and metadata size is independent of file size, this corresponds to installations with potentially many hundreds of petabytes of storage or more, depending on average file size. For example, scientific applications creating checkpoints on LLNL's Bluegene/L might involve 64 thousand nodes with two processors each writing to separate files in the same directory (as in the *makefiles* workload). While the current storage system peaks at 6,000 metadata ops/sec and would take minutes to complete each checkpoint, a 128-node Ceph MDS cluster could finish in two seconds. If each file were only 10 MB (quite small by HPC standards) and OSDs sustain 50 MB/sec, such a cluster could write 1.25 TB/sec, saturating at least 25,000 OSDs (50,000 with replication). 250 GB OSDs would put such a system at more than six petabytes. More importantly, Ceph's dynamic metadata distribution allows an MDS cluster (of any size) to reallocate resources based on the current workload, even when all clients access metadata previously assigned to a single MDS, making it significantly more versatile and adaptable than any static partitioning strategy.

7 Experiences

We were pleasantly surprised by the extent to which replacing file allocation metadata with a distribution function became a simplifying force in our design. Although this placed greater demands on the function itself, once we realized exactly what those requirements were, CRUSH was able to deliver the necessary scalability, flexibility, and reliability. This vastly simplified our metadata workload while providing both clients and OSDs with complete and independent knowledge of the data distribution. The latter enabled us to delegate responsibility for data replication, migration, failure detection, and recovery to OSDs, distributing these mechanisms in a way that effectively leveraged their bundled CPU and memory. RADOS has also opened the door to a range of future enhancements that elegantly map onto our OSD model, such as bit error detection (as in the Google File System [6]) and dynamic replication of data based on workload (similar to AutoRAID [30]).

Although it was tempting to use existing kernel file systems for local object storage (as many other systems have done [3, 6, 8]), we recognized early on that a file system tailored for object workloads could offer better performance [24]. What we did not anticipate was the disparity between the existing file system interface and our requirements, which became evident while developing the RADOS replication and reliability mechanisms. EBOFS was surprisingly quick to develop in user-space, offered very satisfying performance, and exposed an interface perfectly suited to our requirements.

One of the largest lessons in Ceph was the importance of the MDS load balancer to overall scalability, and the complexity of choosing what metadata to migrate where and when. Although in principle our design and goals seem quite simple, the reality of distributing an evolving workload over a hundred MDSs highlighted additional subtleties. Most notably, MDS performance has

a wide range of performance bounds, including CPU, memory (and cache efficiency), and network or I/O limitations, any of which may limit performance at any point in time. Furthermore, it is difficult to quantitatively capture the balance between total throughput and fairness; under certain circumstances unbalanced metadata distributions can increase overall throughput [26].

Implementation of the client interface posed a greater challenge than anticipated. Although the use of FUSE vastly simplified implementation by avoiding the kernel, it introduced its own set of idiosyncrasies. `DIRECT_IO` bypassed kernel page cache but didn't support *mmap*, forcing us to modify FUSE to invalidate clean pages as a workaround. FUSE's insistence on performing its own security checks results in copious *getattr*s (*stat*s) for even simple application calls. Finally, page-based I/O between kernel and user space limits overall I/O rates. Although linking directly to the client avoids FUSE issues, overloading system calls in user space introduces a new set of issues (most of which we have yet to fully examine), making an in-kernel client module inevitable.

8 Related Work

High-performance scalable file systems have long been a goal of the HPC community, which tends to place a heavy load on the file system [16, 24]. Although many file systems attempt to meet this need, they do not provide the same level of scalability that Ceph does. Large-scale systems like OceanStore [10] and Farsite [1] are designed to provide petabytes of highly reliable storage, and can provide simultaneous access to thousands of separate files to thousands of clients, but cannot provide high-performance access to a small set of files by tens of thousands of cooperating clients due to bottlenecks in subsystems such as name lookup. Conversely, parallel file and storage systems such as Vesta [5], Galley [15], PVFS [11], and Swift [4] have extensive support for striping data across multiple disks to achieve very high transfer rates, but lack strong support for scalable metadata access or robust data distribution for high reliability. For example, Vesta permits applications to lay their data out on disk, and allows independent access to file data on each disk without reference to shared metadata. However, like many other parallel file systems, Vesta does not provide scalable support for metadata lookup. As a result, these file systems typically provide poor performance on workloads that access many small files or require many metadata operations. They also typically suffer from block allocation issues: blocks are either allocated centrally or via a lock-based mechanism, preventing them from scaling well for write requests from thousands of clients to thousands of disks. GPFS [22] and StorageTank [14] partially decouple metadata and data management, but are limited by their use of block-based disks and their metadata distribution architecture.

Grid-based file systems such as LegionFS [29] are designed to coordinate wide-area access and are not optimized for high performance in the local file system. Similarly, the Google File System [6] is optimized for very large files and a workload consisting largely of reads and file appends. Like Sorrento [23], it targets a narrow class of applications with non-POSIX semantics.

Recently, many file systems and platforms, including Federated Array of Bricks (FAB) [21] and pNFS [8] have been designed around network attached storage [7]. Lustre [3], the Panasas file system [28], zFS [19], Sorrento, and Kybos [31] are based on the object-based storage paradigm [2] and most closely resemble Ceph. However, none of these systems has the combination of scalable and adaptable metadata management, reliability and fault tolerance that Ceph provides. Lustre and Panasas in particular fail to delegate responsibility to OSDs, and have limited support for efficient distributed metadata management, limiting their scalability and performance. Further, with the exception of Sorrento's use of consistent hashing [9], all of these systems use explicit allocation maps to specify where objects are stored, and have limited support for rebalancing when new storage is deployed. This can lead to load asymmetries and poor resource utilization, while Sorrento's hashed distribution lacks CRUSH's support for efficient data migration, device weighting, and failure domains.

9 Future Work

Some core Ceph elements have not yet been implemented, including MDS failure recovery and several POSIX calls. Two security architecture and protocol variants are under consideration, but neither have yet been implemented [**?**, 17]. We also plan on investigating the practicality of client callbacks on namespace to inode translation metadata. For static regions of the file system, this could allow opens (for read) to occur without MDS interaction. Several other MDS enhancements are planned, including the ability to create snapshots of arbitrary subtrees of the directory hierarchy [25].

Although Ceph dynamically replicates metadata when flash crowds access single directories or files, the same is not yet true of file data. We plan to allow OSDs to dynamically adjust the level of replication for individual objects based on workload, and to distribute read traffic across multiple OSDs in the placement group. This will allow scalable access to small amounts of data, and may facilitate fine-grained OSD load balancing using a mechanism similar to D-SPTF [13].

Finally, we are working on developing a quality of service architecture to allow both aggregate class-based traffic prioritization and OSD-managed reserva-

tion based bandwidth and latency guarantees. In addition to supporting applications with QoS requirements, this will help balance RADOS replication and recovery operations with regular workload. A number of other EBOFS enhancements are planned, including improved allocation logic, data scouring, and checksums or other bit-error detection mechanisms to improve data safety.

10 Conclusions

Ceph addresses three critical challenges of storage systems—scalability, performance, and reliability—by occupying a unique point in the design space. By shedding design assumptions like allocation lists found in nearly all existing systems, we maximally separate data from metadata management, allowing them to scale independently. This separation relies on CRUSH, a data distribution function that generates a pseudo-random distribution, allowing clients to calculate object locations instead of looking them up. CRUSH enforces data replica separation across failure domains for improved data safety while efficiently coping with the inherently dynamic nature of large storage clusters, where devices failures, expansion and cluster restructuring are the norm.

RADOS leverages intelligent OSDs to manage data replication, failure detection and recovery, low-level disk allocation, scheduling, and data migration without encumbering any central server(s). Although objects can be considered files and stored in a general-purpose file system, EBOFS provides more appropriate semantics and superior performance by addressing the specific workloads and interface requirements present in Ceph.

Finally, Ceph's metadata management architecture addresses one of the most vexing problems in highly scalable storage—how to efficiently provide a single uniform directory hierarchy obeying POSIX semantics with performance that scales with the number of metadata servers. Ceph's dynamic subtree partitioning is a uniquely scalable approach, offering both efficiency and the ability to adapt to varying workloads.

Ceph is licensed under the LGPL and is available at `http://ceph.sourceforge.net/`.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. Research was funded in part by the Lawrence Livermore, Los Alamos, and Sandia National Laboratories. We would like to thank Bill Loewe, Tyce McLarty, Terry Heidelberg, and everyone else at LLNL who talked to us about their storage trials and tribulations, and who helped facilitate our two days of dedicated access time on `alc`. We would also like to thank IBM for donating the 32-node cluster that aided in much of the OSD performance testing, and the National Science Foundation, which paid for the switch upgrade. Chandu Thekkath (our shepherd), the anonymous reviewers, and Theodore Wong all provided valuable feedback, and we would also like to thank the students, faculty, and sponsors of the Storage Systems Research Center for their input and support.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 165–176, Apr. 2003.

[3] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[4] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[5] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003. ACM.

[7] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[8] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. Technical Report CITI-05-1, CITI, University of Michigan, Feb. 2005.

[9] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[11] R. Latham, N. Miller, R. Ross, and P. Carns. A next-generation parallel file system for Linux clusters. *LinuxWorld*, pages 56–59, Jan. 2004.

[12] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 226–238. ACM, 1991.

[13] C. R. Lumb, G. R. Ganger, and R. Golding. D-SPTF: Decentralized request distribution in brick-based storage systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–47, Boston, MA, 2004.

[14] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank—a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[15] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

[16] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, Oct. 1996.

[17] C. A. Olson and E. L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, Fairfax, Virgina, USA, Nov. 2005.

[18] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[19] O. Rodeh and A. Teperman. zFS—a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Apr. 2003.

[20] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[21] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

[22] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.

[23] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Nov. 2004.

[24] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, College Park, MD, Apr. 2004.

[25] S. A. Weil. Scalable archival data and metadata management in object-based file systems. Technical Report SSRC-04-01, University of California, Santa Cruz, May 2004.

[26] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004. ACM.

[27] B. Welch. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.

[28] B. Welch and G. Gibson. Managing scalability in object storage systems for HPC Linux clusters. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 433–445, Apr. 2004.

[29] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*, Denver, CO, 2001.

[30] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.

[31] T. M. Wong, R. A. Golding, J. S. Glider, E. Borowsky, R. A. Becker-Szendy, C. Fleiner, D. R. Kenchammana-Hosekote, and O. A. Zaki. Kybos: self-management for distributed brick-base storage. Research Report RJ 10356, IBM Almaden Research Center, Aug. 2005.

[32] J. C. Wu and S. A. Brandt. The design and implementation of AQuA: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, May 2006. To appear.

[33] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172–181, Honolulu, HI, June 2004.