Ceph: A Scalable Object-Based Storage System

Technical Report UCSC-SSRC-06-01
March 2006

Sage A. Weil
sage@cs.ucsc.edu

Feng Wang
cyclonew@cs.ucsc.edu

Qin Xin
qxin@cs.ucsc.edu

Scott A. Brandt
scott@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Darrell D. E. Long
darrell@cs.ucsc.edu

Carlos Maltzahn
carlosm@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
`http://www.ssrc.ucsc.edu/`

Ceph: A Scalable Object-Based Storage System

Sage A. Weil Feng Wang Qin Xin Scott A. Brandt Ethan L. Miller
Darrell D. E. Long Carlos Maltzahn

Storage Systems Research Center
University of California, Santa Cruz
{sage, cyclonew, qxin, scott, elm, darrell, carlosm}@cs.ucsc.edu

Abstract

The data storage needs of large high-performance and general-purpose computing environments are generally best served by distributed storage systems. Traditional solutions, exemplified by NFS, provide a simple distributed storage system model, but cannot meet the demands of high-performance computing environments where a single server may become a bottleneck, nor do they scale well due to the need to manually partition (or repartition) the data among the servers. Object-based storage promises to address these needs through a simple networked data storage unit, the Object Storage Device (OSD) that manages all local storage issues and exports a simple read/write data interface. Despite this simple concept, many challenges remain, including efficient object storage, centralized metadata management, data and metadata replication, and data and metadata reliability. We describe Ceph, a distributed object-based storage system that meets these challenges, providing high-performance file storage that scales directly with the number of OSDs and Metadata servers.

1 Introduction

For performance and capacity reasons, the data storage needs of large high-performance and general-purpose computing environments are best served by distributed storage systems. Traditional solutions, exemplified by NFS [25], provide a straightforward distributed storage model in which each server exports a file system hierarchy that can be mounted and mapping into the local file system name space. While widely used and highly effective, this model was originally designed for small, low-performance (by modern standards) storage systems and is relatively inflexible, difficult to grow dynamically, and incapable of providing performance that scales with the number of servers in a system (except in particularly fortuitous circumstances).

Object-based storage [21] promises to address these limitations through a simple networked data storage unit, the Object Storage Device (OSD). Each OSD consists of a CPU, network interface, local cache, and storage device (disk or small RAID configuration), and exports a high-level data object abstraction on top of the disk (or RAID) block read/write interface. By managing low-level storage details such as allocation and disk request scheduling locally, OSDs provide a building-block for scalability.

Conceptually, object-based storage provides an attractive model for distributed storage. However, there are many issues that need to be addressed in the development of a reliable, scalable, high-performance object-based storage system. These include metadata management, object management, data distribution, reliability, and replication, among others.

We present Ceph, a prototype distributed object-based storage system that meets these challenges, providing high-performance file storage that scales linearly with the number of OSDs and Metadata servers. Designed for multi-petabyte general-purpose and high-performance scientific installations with aggregate throughput of up to 1 TB/s to and from tens of thousands of clients, Ceph provides a robust and effective metadata management architecture, an efficient object file system, an effective data distribution mechanism to assign file data to objects, a robust reliability mechanism to deal with the frequent failures that can be expected in peta-scale file systems, and the ability to scale the system dynamically as new OSDs and metadata servers are added.

The Ceph storage system architecture is based upon a collection of OSDs connected by high-speed networks. A key advantage of OSDs is the ability to delegate low-level block allocation and synchronization for a given segment of data to the device on which it is stored, leaving the file system to choose only which OSD a given segment should be placed. Since this decision is simple and distributable, each OSD need only manage concurrency locally, allowing a file system built from thousands of OSDs to achieve massively parallel data transfers.

Ceph's clustered metadata management architecture, Dynamic Subtree Partitioning [38], provides high performance, scalability, and reliability. It dynamically redelegates and, as needed, replicates responsibility for subtrees of the metadata hierarchy among the metadata servers (MDSs) in the cluster to balance the workload, manage hot spots, and allow for automatic scaling when new metadata servers are added.

Running on each OSD, our object-based file system is based upon a flat object name space, employing a cluster of metadata servers to translate human-readable names into a file identifier and, ultimately, object identifiers. A special-purpose object file system, OBFS [35], provides high-performance object storage for OSDs with relatively little code, taking advantage of the flat name space, lack of directories, and lack of inter-object locality to manage objects much more efficiently than is possible with typical file systems.

Any large file system requires the ability to add storage in the form of new OSDs (or remove old ones). To allow storage scalability without sacrificing parallelism, Ceph utilizes RUSH [14, 15], a family of algorithms that allocate objects to OSDs. RUSH allows for fast and random allocation of objects (for load balancing) but, unlike simple hashing, allows new OSDs to be added to (or removed from) the system without the overhead usually required to rehash and relocate existing objects. RUSH also facilitates replication of objects across multiple OSDs for redundancy.

The raw number of commodity hard drives required to store petabytes of data means that our storage system can be expected to have frequent drive failures. Research has shown that traditional RAID cannot adequately protect the system from data loss, even in the short term. FaRM [41, 42] address this problem, rapidly reconstructing lost disks on an object-by-object basis and providing the high reliability required of such systems.

Developed separately, these components have now proven effective as part of the overall Ceph architecture. We present the design in greater detail, and discuss how Ceph, and the object-based storage model in general, provides reliable, scalable, high-performance file system services. Our results show that Ceph meets its goals in providing high performance, flexibility, and scalability.

2 System Architecture

The Ceph architecture contains four key components: a small cluster of metadata servers (MDSs) that manage the overall file system name space, a large collection of object storage devices (OSDs) that store data and metadata, a client interface, and a high-speed communications network.

2.1 Metadata Management

Metadata operations make up as much as 50% of typical file system workloads [27], making the MDS cluster critical to overall system performance. Ceph utilizes a dynamic metadata management infrastructure based on Dynamic Subtree Partitioning [38] to facilitate efficient and scalable load distribution across dozens of MDS nodes. Although metadata (like data) is ultimately stored on disk in a cluster of OSDs, the MDS cluster maintains a large distributed in-memory cache to maximize performance. Ceph uses a primary-copy replication approach to manage this distributed cache, while a two-tiered storage strategy optimizes I/O and facilitates efficient on-disk layout. A flexible load partitioning infrastructure uses efficient subtree-based distribution in most cases, while allowing hashed distribution to cope with individual hot spots.

The primary-copy caching strategy makes a single authoritative node responsible for managing cache coherence and serializing and committing 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, some recent and experimental file systems have tried hash functions to distribute directory and file metadata, 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 MDS prefetching and storage.

Ceph's metadata server cluster is based on a dynamic metadata management design that allows it to dynamically and adaptively distribute cached metadata hierarchically across a set of MDS nodes. Arbitrary and variably-sized subtrees of the directory hierarchy can be reassigned and migrated between MDS nodes to keep the workload evenly distributed across the cluster. This distribution is entirely adaptive and based on the current workload characteristics. A load balancer monitors the popularity of metadata within the directory hierarchy and periodically shifts subtrees between nodes as needed. The resulting subtree based partition it kept coarse to minimize prefix replication overhead and preserve locality. In the Ceph prototype, the choice of subtrees to migrate is based on a set of heuristics designed to minimize partition complexity—and the associated replication of prefix inodes—by moving toward a simpler distribution whenever possible. When necessary, particularly large or popular directories can then be individually hashed across the cluster, allowing a wide load distribution for hot spots only when it is needed—without incurring the associated overhead in the general case.

Clients cache information about which MDS nodes are authoritative for which directories in their local metadata caches, allowing metadata operations to be directed toward the MDS node authoritative for the deepest known prefix of a given path. The relatively coarse partition makes it easy for clients to "learn" the metadata partition for parts of the file system they're interested in, resulting in very few misdirected queries. More importantly, this basic strategy allows the MDS cluster to manipulate client consensus on the location of popular metadata to disperse potential hot spots and flash crowds (like 10,000 clients opening `/lib/libc.so`). Normally clients learn the proper locations of unpopular metadata and are able to contact the appropriate MDS directly. Clients accessing popular metadata, on the other hand, are told the metadata resides either on different or multiple MDS nodes, distributing the workload across the cluster. This basic approach allows the MDS to effectively bound the number of clients believing any particular piece of metadata resides on any particular server at all times, thus preventing potential flash crowds from overloading any particular node.

Although the MDS cluster is able to satisfy most metadata requests from its in-memory cache, all metadata updates must also be committed to disk for safety. A set of large, bounded, lazily flushed journals allows each MDS to quickly stream its updated metadata to disk in an efficient and distributed manner. The large per-MDS journal also serves to absorb repetitive metadata updates (common to most workloads) such that when dirty metadata are later flushed from the journal to long-term storage, far fewer updates are required. This two-tiered strategy provides the best from 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 (not dissimilar to C-FFS's embedded inodes [9]), allowing the MDS to exploit locality in its workload to prefetch metadata. Inode numbers are managed with journaled updates and distributed free lists, while an auxiliary *anchor table* is used to keep 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 conventional inode table. The anchor table maintains the minimum amount of information necessary to locate "anchored" inodes, providing a simple abstraction that will facilitate future Ceph features like snapshots applied to arbitrary subtrees of the directory hierarchy.

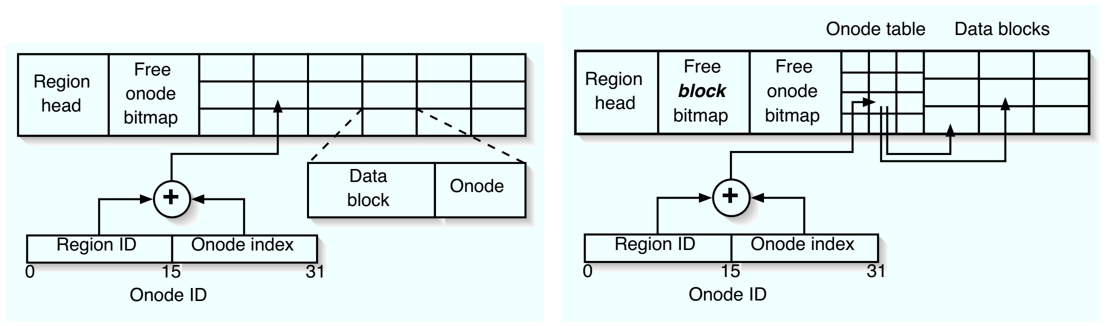
The core of the MDS design is built around a set of distributed protocols that manage distributed cache coherency and metadata locking hierarchies, allowing subtrees to be seemlessly migrated between nodes while ensuring file system consistency with per-MDS journals. In the event of an MDS failure, the journal can be rescanned to both reconstruct the contents of the failed node's in-memory cache (for quick startup) and (in doing so) recover the Ceph file system state.

2.2 Object Storage

OBFS [35] is the storage manager on each OSD—it provides object storage and manages local request scheduling, allocation, and caching. Ceph files are striped across objects to enable a high degree of parallelism, limiting objects to the system stripe unit size and (intentionally) destroying inter-object locality within a single OSD. Delayed writes in the file cache at the client side absorb most small writes and result in relatively large synchronous object reads and writes to the OSDs (a more detailed analysis of the expected object workload is provided elsewhere [36]). OBFS exploits these design choices to simply on-disk layout for both performance and reliability.

To maximize overall throughput without over-committing resources to small objects, OBFS employs multiple block sizes and uses *regions* (see Figure 1), analogous to cylinder groups in FFS [20], to keep blocks of the same size together. The block size of a region is determined at the time that a (free) region is initialized, which occurs when there are insufficient free blocks in any initialized region to satisfy a write request. When all of the blocks in an initialized region are freed, OBFS returns the region to the free region list. OBFS uses two block sizes: small (4 KB, the logical block size in Linux), and large (1 MB, the system stripe unit size). Overall, this scheme has many advantages; it minimizes file system fragmentation, simplifies allocation, avoids unnecessary wasted space and effectively uses the available disk bandwidth. The use of regions also reduces the size of other file system data structures such as free block lists or maps and thus makes the operations on those data structures more efficient.

Object metadata, stored in *onodes*, is used to track the status of each object. Onodes are preallocated in fixed positions at the head of small block regions, similar to the way inodes are placed in cylinder groups in FFS [20]. In large block regions, shown in Figure 2, onodes are colocated with the data block on the disk, similar to embedded inodes [9]. This allows for very efficient metadata updates



(a) Large Block Region (b) Small Block Region

Figure 2: *Region structure and data layout.*

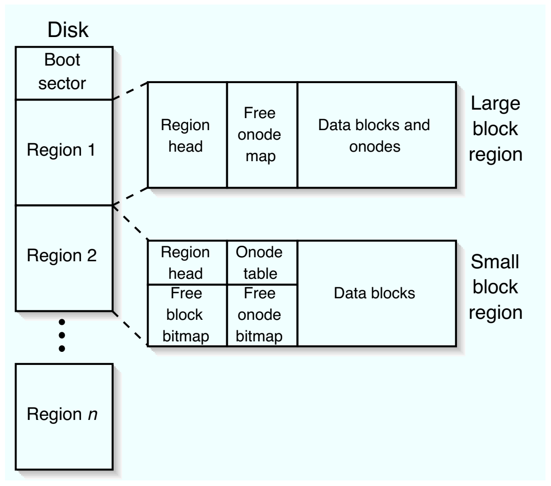


Figure 1: *OBFS structure*

as the metadata can be written with the corresponding data block.

As shown in Figure 2, each onode has a unique 32-bit identifier consisting of two parts: a 16 bit region identifier and a 16 bit in-region object identifier. If a region occupies 256 MB on disk, this scheme will support OSDs of up to 16 TB, and larger OSD volumes are possible with larger regions. To locate a desired object, OBFS first finds the region using the region identifier and then uses the in-region object identifier to index the onode. This is particularly effective for large objects because the object index points directly to the onode and the object data, which are stored contiguously. In the current implementation, onodes for both large and small objects are 512 bytes, allowing OBFS to avoid using indirect blocks entirely: the maximum size of a small object will always be less than the stripe unit size, which is 1 MB in our design, ensuring that block indices will fit inside the onode.

An *Object Lookup Table* (OLT) manages the mapping between object identifiers and onode identifiers. The size of the OLT is proportional to the number of objects in the OSD and remains quite small: with 20,000 objects residing in an OSD, the OLT requires only 233 KB. For efficiency, the OLT is loaded into main memory and updated asynchronously. A Region Head List (RHL) stores information about each region in the file system, including pointers to the free block bitmap and the free onode bitmap. On an 80 GB disk, the RHL occupies 8 MB of disk space. Like the OLT, the RHL is loaded into memory and updated asynchronously. After obtaining an onode identifier, OBFS searches the RHL using the upper 16 bits of the onode identifier to obtain the corresponding region type. If the onode belongs to a large block region, the object data address can be directly calculated. Otherwise, OBFS searches the in-memory onode cache to find that onode, loading it from disk if the search fails.

OBFS asynchronously updates important data structures such as the OLT and the RHL to achieve better performance. In order to guarantee system reliability, OBFS updates some important information in the onodes synchronously. If the system crashes, OBFS can quickly scan all of the regions and onodes on the disk to rebuild the OLT and the RHL. For each object, the object identifier and the region identifier are used to assemble a new entry in the OLT. The block addresses for each object are then used to rebuild each region free block bitmap. Because the onodes are synchronously updated, we can safely rebuild the entire OLT and RHL and restore the system. OBFS updates onode metadata either without an extra disk seek or with one short disk seek (depending on object type). In so doing, it keeps the file system reliable and maintains system integrity with very little overhead.

2.3 OSD Cluster Management

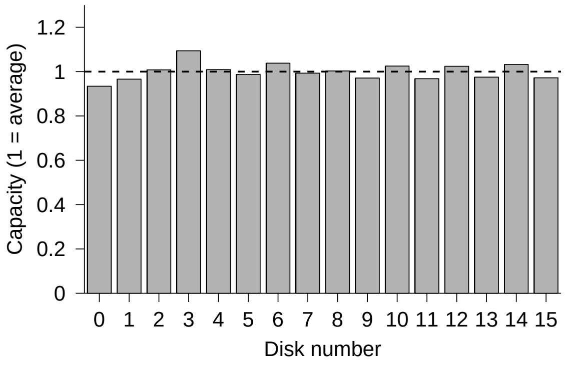
Ceph's OSD cluster is used for storing both data and metadata (although the same set of disks need not be responsible for both). Intelligent OSDs allow data replication, failure detection and recovery activities take place semi-autonomously under the supervision of the MDS cluster.

This intelligence in the storage layer allows the OSD cluster to collectively provide a reliable, scalable, and high-performance object storage service to client and MDS nodes.

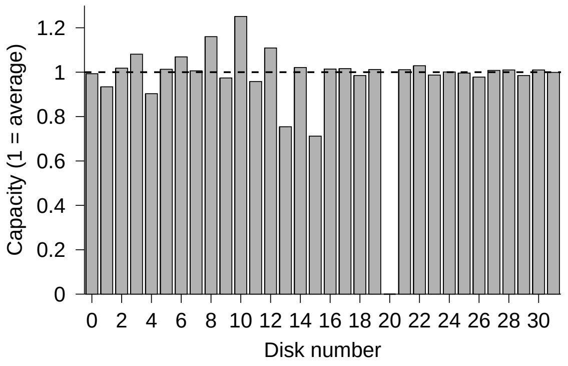
2.3.1 Data Distribution Using RUSH

In Ceph, the contents of each file are striped over a sequence of objects stored on OSDs. To ensure that tens of thousands of clients can access pieces of a single file spread across thousands of object-based disks, Ceph must use a distribution mechanism free of central bottlenecks. This mechanism must accommodate replication, allow for the storage system to be easily expanded, and preserve load balance in the face of added capacity or drive failures. We chose the RUSH_R variant of the RUSH [15] algorithm to distribute data because it meets all of these goals.

The RUSH algorithm first maps a $\langle \textit{fileId}, \textit{offset} \rangle$ pair into one of a large number (10^7) of *redundancy groups*. A redundancy group is a collection of data blocks (objects) and their associated replicas or parity blocks. There are several ways to use the redundancy groups, including mirroring, parity, and more advanced erasure coding schemes, with the trade-offs between the different redundancy mechanisms lying in the complexity of parity management and recovery operations, the bandwidth consumed by recovery [37], and the storage efficiency. RUSH performs a second mapping that identifies all of the OSDs on which a redundancy group is stored, with the guarantee that no OSD will appear twice in the list. Since Ceph currently uses replication, it stores the *n* replicas of the objects in the redundancy group on the first *n* operational OSDs in the list. The only other input into this mapping is a list of $\langle \textit{numDisks}, \textit{weight} \rangle$ tuples describing each cluster of OSDs that has been added to the system, where *numDisks* is the size of the cluster added and *weight* is a weight for the new disks relative to other disks in the system. The *weight* parameter can be used to bias the distribution of objects to OSDs, as would be necessary if, for example, newer disks had a higher capacity than older disks. The mapping process is very fast, requiring less than $1\,\mu$s per cluster to compute; for a system that had 100 clusters added, this mapping would take less than $100\,\mu$s. Ceph additionally must keep a list of failed OSDs so it can skip over them when selecting OSDs from the list generated by the RUSH mapping. These lists change slowly—on the order of a few times per day at most—and are small, allowing them to be cached cached at each client. Most importantly, clients are able to quickly locate file data based only on a *fileId* (inode number) and file offset, without consulting a central metadata server.



(a) Data distribution across a system with 16 nodes. The system was built incrementally, starting with 8 disks, adding 4 more, and then adding 4 more for a total of 16 disks.



(b) Data distribution across a cluster of 32 nodes with one node failed. This system was built from the system in the graph above by adding 16 disks all at once, and declaring disk 20 as failed.

Figure 3: *Data distribution balance using RUSH.*

By generating a list with more replicas than absolutely required, Ceph can handle OSD failures by making an additional replica of each affected group on the next OSD in the list. Since each redundancy group on an OSD has a distinct list, it is likely that an OSD failure will result in the new replicas for the groups being placed on different OSDs, increasing reconstruction speed and decreasing the likelihood that a second failure will occur before the system has restored the necessary level of replication. The RUSH algorithm probabilistically guarantees that data will be distributed evenly (modulo weighting) across all of the OSDs, regardless of the number of clusters added to the system or the number of individual OSDs that have failed, as shown in Figure 3.

2.3.2 Replication

Object replication is managed in terms of the redundancy groups (a collection of objects, as described in Section 2.3.1) and the associated list of OSDs. In the Ceph prototype, these OSDs are used to store whole object replicas for *n*-way replication; parity and erasure coding schemes are also possible, but not implemented. Within each redundancy group, the first OSD is responsible for storing the primary copy of an object and managing replication, while the following one or more OSDs store replicas. Each OSD will participate in many thousands of redundancy groups made up of a different (and seemly random) sets of OSDs, and will act in a different role (primary or replica) for each.

All object read and write operations are directed to the redundancy group's primary OSD, which is ultimately responsible for ensuring an object's reliable storage (both locally and on the replica OSDs). Read operations can be satisfied locally either from the OSDs buffer cache or by reading from the local disk. (Although it is tempting to balance read traffic across an object's replicas, this only complicates replication without benefit; RUSH already distributes load evenly across OSDs.)

In the case of object writes and updates, the primary OSD forwards the request on to the replica(s) before committing the changes locally, and does not acknowledge the operation until the update is safely stored both on the local disk and in the buffer cache on the replica OSD(s). This approach allows the latency of replication to be masked by that of the local disk write, while freeing the OSD user from dealing with any replication-related activities (and related consistency considerations). It is also likely that the internal OSD network bandwidth will be greater than the external bandwidth in typical installations, making this a desirable (and often ideal) network data path. The primary OSD is then responsible for maintaining certain local state to ensure that updated objects are subsequently committed to disk on replicas.

This choice of when to acknowledge a write as "safe" is based on consideration of the most common failure scenarios. If the primary OSD fails, the write isn't acknowledged, and the user can retry. If the replica fails, the primary will re-copy affected objects to a new replica during the recovery process based on its receipt of an "in buffer" but no "on disk" acknowledgment. And in a correlated failure (like a power outage), the data is either never acknowledged or safely on disk on the primary OSD.

2.3.3 Failure Detection and Recovery

OSD failure detection in Ceph is fully distributed, while being centrally managed by the MDS cluster. Although failure modes that do not include network disconnection (like local file system or media defects) involve self-reporting by OSDs to the MDS cluster, communication failure detection requires active monitoring and by necessity a distributed approach in a cluster of 10,000 or more OSDs. Each OSD in Ceph is responsible for monitoring a small set of its peers, and notifying an MDS of any communication problems detected. Overlapping monitoring groups are assigned by a pseudo-random hash function, such that each OSD monitors *n* other OSDs and is conversely monitored by *n* peers. We use the RUSH function for peer set assignment, although not all of its properties are required. Liveness information piggybacks on existing inter-OSD (replication) chatter or explicit "ping" messages when necessary. The MDS cluster collects failure reports and verifies failures to filter out transient or systemic problems (like a network partition) centrally. This combination of distributed detection and a centralized coordinator in Ceph takes the best from both worlds: it allows fast detection without unduly burdening the MDS cluster, and resolves the occurrence of inconsistency with the arbitrament of the MDS cluster.

Non-responsive OSDs are initially marked as *down* to indicate a potentially transient failure. During this period the next replica in each redundancy group works as the *acting primary* by completing write requests locally. If the primary OSD recovers, a log is used to resynchronize redundancy group content. If the OSD does not recover after some interval, it is marked as *failed* and recovery is initiated, at which point the acting primary becomes the new primary and the objects in each redundancy group are replicated to the next replica in line. OSD state changes are distributed to active clients and OSDs via an epidemic-style broadcast (piggybacking on existing messages when possible). Clients attempting to access a newly failed disk simply time out and retry OSD operations until a new OSD status (*down* or *failed*) is learned, at which point requests are redirected toward the new or acting primaries.

Ceph's RUSH-based data distribution algorithm utilizes FaRM [41, 42], a distributed approach to fast failure recovery. Because replicas for the objects stored on any individual OSD are declustered across a large set of redundancy groups and thus OSDs, failure recovery can proceed in a parallel and distributed manner as the set OSDs with replicas copy objects to a new (similarly distributed) set of OSDs. This approach eliminates the disk "rebuild" bottleneck typical of traditional RAID systems (in which recovery in limited by a single disk's write bandwidth), thus greatly speeding the recovery process and minimizing the window of opportunity for a subsequent failure to cause data loss. Prior research has shown that FaRM's fast recovery can reduce a large storage system's probability of data loss by multiple orders of magnitude.

2.3.4 Scalability

The Ceph prototype stripes file data across 1 MB objects, scattered across different OSDs. In contrast to object-based storage systems like Lustre [3, 30] that stripe data over a small set of very large objects, Ceph instead relies on a large set of medium-sized and well distributed objects. This approach simultaneously allows massive I/O parallelism to both individual large files, whose contents may be spread across the entire OSD cluster, and large sets of smaller files. We believe the 1 MB object size provides a good tradeoff between seek latency and read or write time in individual OSD workloads—which exhibit little or no locality in a large system—while scaling in terms of both aggregate and single file performance. The Ceph architecture accommodates arbitrary striping strategies as well, including arbitrary stripe widths, stripe set sizes, and objects sizes. This flexibility allows one to trade single file parallel throughput for the preservation of some locality in OSD workloads and small gains in OSD efficiency under streaming applications.

The RUSH data distribution allows Ceph storage to scale by providing a balanced, random data distribution that facilitates OSD cluster expansion with minimal data relocation. By defining the mapping process recursively, RUSH results in the approximate minimal amount of data movement that is required in order to restore balance to disk capacities after new (empty) OSDs are added. This allows the overall storage capacity to be expanded (or reduced) in an optimal fashion: data movement is minimized while data distribution is preserved.

More specifically, OSD cluster expansion results in the relocation of some set of redundancy groups to new OSDs. When each OSD learns of a cluster expansion, it iterates over the set of redundancy groups for which it is newly responsible (a non-trivial but reasonable computation) to determine which objects it need to migrate from the old primaries. During the migration process, read requests for any object not yet migrated are proxied to the old primary. Updates are committed locally, along with additional state to track which parts of the not-yet-migrated object are new.

2.4 Client Interface

The Ceph client combines a local metadata cache, a buffer cache, and a POSIX call interface. Our prototype implementation is in userspace, allowing an application to either link to it directly (as with our synthetic workloads and trace replays), or for a Ceph file system to be mounted directly on a Linux system via a thin FUSE (user space file system [33]) glue module.

The client communicates with the MDS cluster to open and close files and manipulate the file name space, while maintaining a local metadata cache both for efficiency purposes and to "learn" the current partition of metadata across the MDS cluster. Metadata consistency is currently similar to that of NFS: inode information in the cache remains valid for a fixed period (we use 10 seconds) before subsequent `stat` operations require contacting the MDS cluster. The client reads from and writes to files by communicating with OSDs directly: once opened, the inode number and byte offset specify (via RUSH) an object identifier and OSD to interact with. A buffer cache serves to filter out redundant file I/O while maximizing the size of file requests submitted to OSDs.

Buffer cache coherence among clients is controlled by the MDS cluster by sending capability updates to clients with open files. These updates can cause the client to either flush all dirty buffers or invalidate all buffers of a particular file. In addition to MDS-initiated buffer cache flushes the client cleans buffers which have been dirty for longer than a fixed time period (we use 30 seconds).

2.5 Network

The Ceph prototype currently uses a single gigabit Ethernet switch for its network; the relatively small scale of the system makes this feasible. This approach limits bandwidth to a single node to about 80–100 MB/s. Because current disks cannot transfer data faster than this rate, gigabit bandwidth is sufficient. Future OSDs will likely require faster networks, such as 10-gigabit Ethernet or other networking technologies.

A more important concern for larger OSD systems is that a single monolithic switch will no longer be feasible since a large OSD system may have thousands of nodes and require hundreds of gigabytes or terabytes per second of total bandwidth. This network must also be reliable in the face of a small number of link and switch failures. The two basic approaches to such a network would be the construction of a multi-layer switching network or a network in which each OSD has its own network switch. The latter approach may be more scalable, and resembles the decades-old problem of building massively parallel computers using relatively slow networks. The details of how to build a large-scale interconnection network for an OSD system are beyond the scope of this paper, but other research has considered the issue [16, 43].

3 Performance

We evaluate the performance of the Ceph prototype by examining both the performance of individual subsystems under pathological workloads, and by measuring performance of the entire system under more typical load.

3.1 Metadata Server

We evaluated the scalability of the MDS cluster with a series of performance tests that scale the size of the OSD

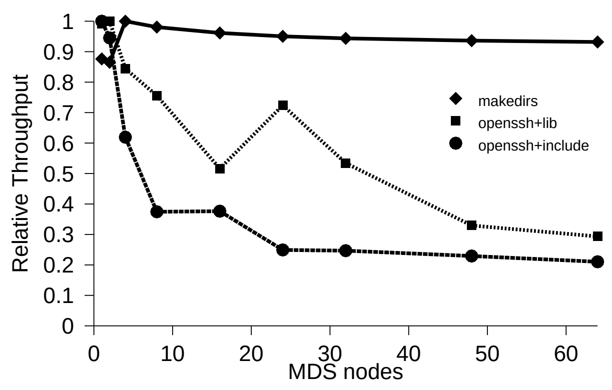


Figure 4: *Relative MDS performance as cluster size scales under three workloads. Heavily shared directories limit load distribution.*

cluster and workload in unison with the MDS cluster size. In all cases we use twice as many OSDs as MDS nodes for metadata storage. An additional set of nodes (one per MDS) is used to generate a synthetic workload, each node running 50 independent instances of the client and associated workload generator.

The `makedirs` workload creates a huge tree of nested directories and files *n* (4) levels deep, each directory containing *m* (25) children. The `openssh+lib` workload replays a trace of a compile of openssh with the `/usr/lib` directory shared with other clients, while `openssh+include` shares `/usr/include` instead. Both traces include extracting the source tree, doing the compile, and then removing everything before repeating. In all three tests MDS performance is CPU-bound; in practice MDS performance will additionally be limited by memory cache size and the resulting I/O rates from cache misses, making MDS performance further dependent on that of the OSD cluster.

Figure 4 shows the MDS cluster scaling from 1 up to 64 nodes under the three workloads. The `makedirs` workload is trivially separable under the dynamic subtree-based metadata partitioning scheme, resulting in almost perfectly linear scaling[1]. The compilations show a reduction in individual MDS throughput for large cluster sizes due to heavy traffic in shared directories. In both tests `/usr/lib` or `/usr/include` is replicated to distribute read access, but file opens are still directed toward the primary copy residing on a single node. The `openssh+lib` workload fares much better because access to the shared `/usr/lib` compromises only approximately 30% of the trace, versus about 65% for `openssh+include`. The strangely poor performance of the 16-node MDS cluster under `openssh+lib` (the erratic dip in Figure 4) is caused by "thrashing" in the load balancer, due to a subtle interaction between the balancing heuristics and the workload that causes metadata to be moved unnecessarily between MDS nodes. Although tuning balancer parameters should resolve the issue, it's inclusion here highlights the importance of better intelligence in the load balancing algorithm—intelligence that is completely lacking (by design) in conventional static subtree partitioning strategies.

In both traces scaling is limited by the separability of the workload, specifically in terms of the replication and distribution of popular metadata (the shared directories) and the load distribution of the independent (per-client) localized workload components. Although the shared "hot spot" directories were replicated across all nodes, file open and close operations are currently directed toward the primary inode copy residing on a single node; although the metadata architecture is designed to allow individual directories to be hashed (thus distributing the contents of hot directories), Ceph does not yet fully implement this feature. The irregularity in the `openssh+lib` trace also indicates the importance of a load balancer that is robust to a wide variety of workloads. The current balancer used by Ceph does reasonably well with most workloads, but is still relatively primitive and can thrash in certain pathological cases (*e.g.* a walk of the directory tree).

Figure 5 shows the latency experienced by clients as a 4-node load balanced MDS cluster approaches saturation under the `makedirs` (exclusively write) workload. The MDS cluster is synchronously journaling metadata update operations and committing a stream of recently created directories to long-term storage. Typical latency experienced by clients in our prototype is on the order of 1–2 ms for metadata read operations and 3-4 ms for write operations, where the difference is due to an additional network round trip (to the OSD) and synchronous disk write. This latency penalty for updates can of course be avoided by disabling Ceph's synchronous metadata journaling (at the expense of NFS-like safety semantics) or using NVRAM on the MDS to mask the write latency.

3.2 File and Object Storage

For compatibility and comparison purposes we implemented a simple object storage module that would function on top of any existing kernel file system. The flat object name space was implemented by hashing object names over a set of directories to avoid directory size

[1]The small divergence from perfectly linear in the `makedirs` workload is in fact most likely due to lock contention in the messaging subsystem.

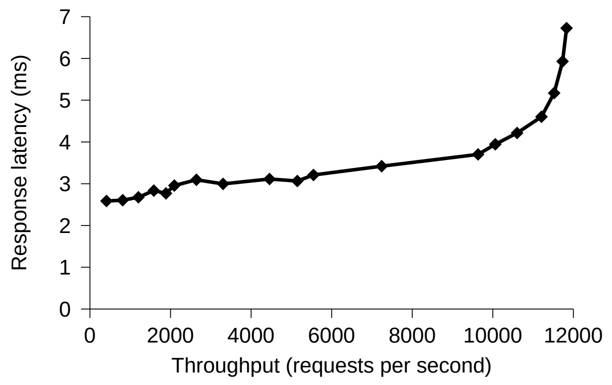


Figure 5: *Metadata latency versus throughput curve as the cluster approaches saturation.*

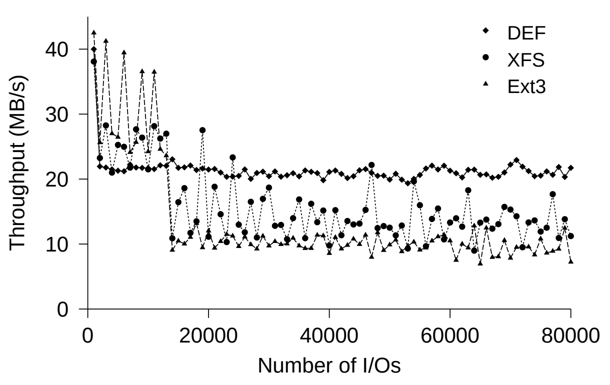


Figure 6: *OBFS write performance on a single node*

and lookup performance limitations present in popular file systems like ext3. We then compared performance of the OSDs in Ceph using both ext2/3 and the special-purpose OBFS.

3.2.1 ext3

For Ceph metadata I/O bound workloads (such as create-only workloads reliant on efficient MDS journaling), we found (not unsurprisingly) that synchronous OSD write performance was critical to metadata throughput. In particular, the use of `fsync()` or `fdatasync()` on ext2 (no journaling) restricted streaming performance when making small synchronous writes to a small number of objects. We avoided this limitation by striping metadata updates over a large set of (16) objects with a very small (256-byte) stripe size. This improved performance even when striping over a small set of (4) OSDs, suggesting strange synchronous write behavior in ext2 and underscoring the importance of an OSD file system properly tuned for Ceph workloads.

3.2.2 OBFS

We evaluated OBFS performance on an individual OSD, as shown in Figure 6, relative to Linux XFS and ext3. We used a synthetic workload composed of 80% of 512 KB large objects, and 20% small objects, with small object sizes uniformly distributed between 1 KB and 512 KB. Disk aging effects were simulated by introducing a large number of write/delete operations, which always maintained the disk usage around 60%. As we can see from the figure, OBFS provides sustained throughput around 20 MB/sec, which improves the performance by a factor of 40% over XFS, and double that of ext3.

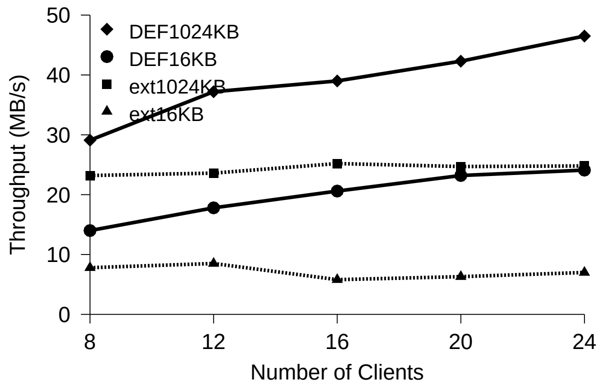


Figure 7: *Large file write throughput per OSD as the number of clients scales*

3.3 Large File Performance

Streaming I/Os, most common in multi-media and scientific environments, can easily pass through the client buffer cache and stress the back end storage. We evaluated the streaming I/O performance by sequentially reading and writing several very large files across a 6-node OSD cluster. A single MDS was used, while a variable number of client nodes were used to generate workload. Each client process simply opened a very large file and sequentially wrote to or read from it. The I/O request sizes varied from 16 KB to 1 MB for each run. We compare the performance of OBFS and ext2 as the underlying file system used for each OSD. We used ext2 over ext3 (despite its poor consistency properties) because it performed better.

Figure 7 shows the average per-OSD write throughput as the number of clients scales. With an I/O request size of 16 KB, 8 client nodes can easily saturate the 6-node

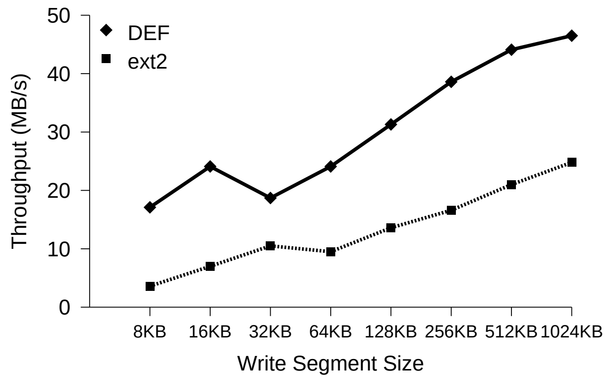


Figure 8: *Large file write throughput per OSD as request size varies*

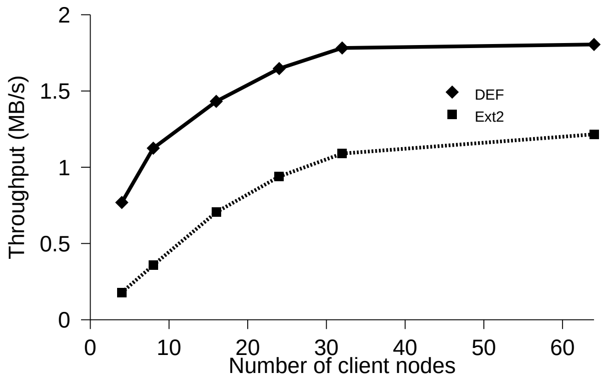


Figure 9: *Throughput per OSD with small 4 KB requests as client load varies*

OSD cluster using the ext2 file system. The average OSD throughput is around 8 MB/sec, which yields 48 MB/sec overall throughput. The OSD cluster with OBFS shows much better performance, saturating only after more than 20 client nodes were used. The average OSD throughput is more than 20 MB/sec, for a combined throughput of up to 120 MB/sec for this small cluster. Using large I/O request sizes significantly improves OSD efficiency: the OSD cluster with ext2 achieves an average of 20 MB/sec throughput per OSD. OBFS demonstrates extremely good performance under the same workload, servicing 24 client nodes before saturating with an average per-OSD throughput of 44 MB/sec, almost 80% of the raw disk performance.

Figure 8 shows the average OSD write performance as the I/O request size changes. Small I/O sizes incur many more object metadata operations on each OSD and destroy the sequentiality of the original streams. By using larger request sizes, the average OSD throughput can be improved by a factor of two, underscoring the importance of the client buffer cache.

3.4 Small File Performance

We evaluated the performance of the OSD cluster with random reads and writes: small files, and all file requests less than 4k (as would be seen without a client buffer cache). The workload consisted of 20% reads and 80% writes. Figure 9 shows the per-OSD throughput on a cluster 16 OSDs as the number of client load generators varies. Performance under this extreme workload is much lower (just over 1 MB/sec), again emphasizing the importance of the client buffer cache for coalescing small requests whenever possible.

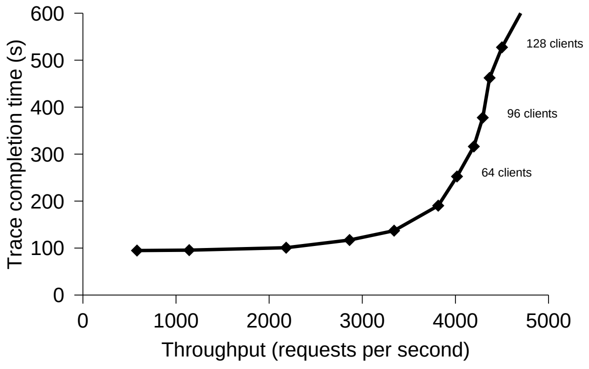


Figure 10: *Compile trace completion time versus throughput under varying client levels*

3.5 Overall Performance

To evaluate the overall performance of the Ceph file system we look at a 4 MDS, 16 OSD storage cluster with a workload consisting of multiple clients simulating a compile of openssh. Each client replays a file access trace generated from an actual compilation (including both metadata and data operations) in a private directory. Figure 10 shows average client completion times on the *y* axis as a function of system throughput (measured in the number of metadata operations on the *x* axis). System throughput in this case is limited by OSD I/O (not surprising given the large MDS to OSD ratio) as a hundred clients hammer only 16 disks.

4 Related Work

High-performance, scalable file systems have long been a goal of the high-performance computing (HPC) community. HPC systems place a heavy load on the file system [24, 31, 36], placing a high demand on the file system to prevent it from becoming a bottleneck. As a result, there have been many scalable file systems that attempt to meet this need; however, these file systems do not support the same level of scalability that Ceph does. Some large-scale file systems, such as OceanStore [18] and Farsite [1] are designed to provide petabytes of highly reliable storage, and may be able to provide simultaneous access to thousands of separate files to thousands of clients. However, these file systems are not optimized to provide high-performance access to a small set of files by tens of thousands of cooperating clients. Bottlenecks in subsystems such as name lookup prevent these systems from meeting the needs of a HPC system. Similarly, grid-based file systems such as LegionFS [40] are designed to coordinate wide-area access and are not optimized for high performance in the local file system.

Parallel file and storage systems such as Vesta [7], Galley [23], RAMA [22], PVFS and PVFS2 [6, 19], the Global File System [32] and Swift [5] have extensive support for striping data across multiple disks to achieve very high data transfer rates, but do not have strong support for scalable metadata access. 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, Vesta, like many other parallel file systems, 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 as well as workloads that require many metadata operations. They also typically suffer from block allocation issues: blocks are either allocated centrally or, in the Global File System, via a lock-based mechanism. As a result, these file systems do not scale well to write requests from thousands of clients to thousands of disks. Similarly, the Google File System [10] is optimized for very large files and a workload consisting largely of reads and file appends, and is not well-suited for a more general HPC workload because it does not support high-concurrency general purpose access to the file system.

Recently, many file systems and platforms, including Federated Array of Bricks (FAB) [28], IceCube [17], Lustre [3, 30], GPFS [29], the Panasas file system [39], pNFS [13], Sorrento [34], and zFS [26] have been designed around network-attached storage [11, 12] or the closely related object-based storage paradigm [2]. All of these file systems can stripe data across network-attached devices to achieve very high performance, but they do not have the combination of scalable metadata performance, expandable storage, fault tolerance, and POSIX compatibility that Ceph provides. pNFS [13] and the Panasas object-based file system [39] stripe data across network-attached disks to deliver very high data transfer rates, but they both suffer from a bottleneck in metadata lookups. Lustre [3, 30] has similar functionality: it supports nearly arbitrary striping of data across object storage targets, but it hashes path names to metadata servers. This approach distributes the metadata load, but destroys locality and makes POSIX compatibility difficult, despite approaches such as LH3 [4]. GPFS [29] also suffers from metadata scaling difficulties; while block allocation is largely lock-free, as it is in most object-based storage systems, metadata is not evenly distributed, causing congestion in metadata lookups. Moreover, none of these systems permits a client to locate a particular block of a file without consulting a centralized table. Sorrento [34] alleviates this problem somewhat and evenly distributes data and metadata among all of the servers, but only performs well in environments with low levels of write sharing in which processors work on disjoint data sets. FAB [28] focuses on continuously providing highly reliable storage; while its performance is acceptable, FAB provides very high reliability at the cost of somewhat reduced performance. Ceph takes the opposite approach: provide very high performance and reasonable reliability.

5 Future Work

Ceph builds upon many distinct and active research topics and opens up a variety of areas for future research. Now that the basic Ceph infrastructure is in place, we can begin to focus on performance optimization and additional functionality.

A number of key MDS optimizations and enhancements are planned: the hashing of individual directories, MDS failure recovery, and a distributed *anchor* subsystem used for facilitating multiple hard links and snapshots. Although most of the more challenging design problems have been solved, a number of the remaining systems and protocols will involve substantial effort, particularly the distributed journaling and failure recovery processes, and related correctness proofs.

The MDS load balancer currently employed is relatively simplistic, taking only a single performance and resource metric into consideration. Simulations [38] suggest that a number of resources may limit MDS performance, and that all of them should potentially be considered. The load balancer should further be robust to a wide variety of metadata workloads, a prospect that suggests the importance of additional intelligence. The application of machine learning algorithms to the variety of replication and load distribution parameters is an exciting area for potential further research.

A number of Ceph protocol revisions are planned as well, most notably surrounding the possibility of end-to-end metadata consistency at the application (POSIX) level through the use of short-term leases on inode and name space metadata. Lazy management of file access capabilities and leases on path to inode mappings will further allow successive `open` and `close` operations to proceed local to clients without MDS interaction, while streamlining buffer cache consistency issues—a potential boon for client performance.

Additional research in alternative object file systems for the OSDs, storage system quality of service, a *collection* interface that allows named groups of objects to managed by each OSD, and improved replication and recovery management are also in progress. Finally, recent research has described abstract models for common large-scale distributed processing applications like MapReduce [8] that involve distributed data processing, structured communication, and storage. These types of operations are common to data mining, search engine, and other applications. The OSD intelligent disk model and Ceph's data distribution and distributed replication and recovery processes make it an ideal platform for implementing a generalized distributed data processing architecture (for which MapReduce might be but one application).

6 Conclusions

Object-based storage promises scalability and high performance by distributing low-level allocation and scheduling operations to the storage devices, enabling a tremendous amount of data parallelism in the storage subsystem. However, the basic model and the scale of the storage systems it enables present many significant challenges including managing the metadata, storing the data efficiently on each OSD, distributing the data effectively, scaling the storage system dynamically, and managing the frequent disk failures that are expected in a multi-petabyte storage system.

The Ceph object-based storage system addresses these challenges, providing robust, high-performance, flexible, scalable storage. Ceph's metadata management addresses one of the most vexing problems in highly scalable storage—how to efficiently provide a single uniform directory hierarchy obeying POSIX directory semantics with performance that scales linearly with the number of metadata servers. Dynamic Subtree Partitioning is a uniquely scalable approach, offering both efficiency and the ability to adapt to varying workloads.

Ceph's data distribution algorithm, RUSH, addresses another key challenge—how to assign data to nodes such that it can be rapidly located, workloads are evenly balanced, and minimal data movement is required as object stores are added to or removed from the system. The RUSH family of algorithms achieve these goals with a fast iterative process that distributes data according to a hash function, but does not require the massive data movement that standard rehashing would require upon the addition or removal of storage devices.

Objects can be considered files, and stored using a general-purpose file system such as ext2/3 or XFS. However, our results demonstrate that by targeting the specific needs of objects (as opposed to files), the OBFS object file system provides excellent low-level object storage performance tailored to object workloads present in the Ceph file system. OBFS provides efficient storage and high performance through a region-based architecture supporting multiple block sizes. Small blocks allocated with extents provide efficient performance for small objects, while large blocks equal to the system stripe unit size provide extremely high performance for large objects.

Our results demonstrate Ceph's performance and scalability and hint at the tremendous potential inherent in the object-based storage model. Now that the basic Ceph infrastructure is complete, our future work will focus on characterizing the bottlenecks that arise in distributed object-based storage systems and optimizing Ceph's performance, as well as developing new avenues of research that are enabled by Ceph's highly distributed architecture. Although not quite ready for public distribution, a goal of the Ceph project is to release the code to the public domain where it can serve as a reference implementation that others can use for their own research, to experiment with, build upon, and compare against[2].

Acknowledgments

We would like to thank the faculty and students of the Storage Systems Research Center for their help and guidance. This research was sponsored in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. Additional support was provided by SSRC industrial sponsors.

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.

[2]Our specific goal is to release the code before FAST

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] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290–298, Apr. 2003.

[5] 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.

[6] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.

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

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[9] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, Jan. 1997.

[10] 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.

[11] 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.

[12] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[13] 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.

[14] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the 17th International Parallel & Distributed Processing Symposium (IPDPS 2003)*, Nice, France, Apr. 2003.

[15] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[16] A. Hospodor and E. L. Miller. Interconnection architectures for petabyte-scale high-performance storage systems. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 273–281, College Park, MD, Apr. 2004.

[17] IBM Corporation. IceCube – a system architecture for storage and Internet servers. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB_Hardware/.

[18] 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.

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

[20] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[21] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8), Aug. 2003.

[22] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, 1997.

[23] 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.

[24] 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.

[25] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, , D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, Maastricht, Netherlands, May 2000.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[31] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 49–59. IEEE, 1996.

[32] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 319–342, College Park, MD, 1996.

[33] M. Szeredi. File System in User Space README. http://www.stillhq.com/extracted/fuse/README, 2003.

[34] 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)*, Pittsburgh, PA, Nov. 2004.

[35] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300, College Park, MD, Apr. 2004. IEEE.

[36] 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.

[37] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, Cambridge, Massachusetts, Mar. 2002.

[38] 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.

[39] 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.

[40] 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.

[41] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.

[42] 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.

[43] Q. Xin, E. L. Miller, T. J. E. Schwarz, and D. D. E. Long. Impact of failure on interconnection networks in large storage systems. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Monterey, CA, Apr. 2005.