A case for scaling HPC metadata performance through de-specialization

Swapnil Patil, Kai Ren and Garth Gibson
Carnegie Mellon University
{svp, kair, garth}@cs.cmu.edu

I. Introduction

Lack of a highly scalable and parallel metadata service is the Achilles heel for many cluster file system deployments in both the HPC world [17], [24] and the Internet services world [10]. This is because most cluster file systems have focused on scaling the data path, i.e. providing high bandwidth parallel I/O to files that are gigabytes in size. But with proliferation of massively parallel applications that produce metadata-intensive workloads, such as large number of simultaneous file creates [6] and large-scale storage management [2], cluster file systems also need to scale metadata performance.

Numerous applications and use-cases need support for concurrent and high-performance metadata operations. One such example, checkpointing, requires the metadata service to handle large number of file creates and updates at very high speeds [6]. Another example, storage management, produces read-intensive metadata workload that typically scans the metadata of the entire file system to perform administration tasks for analyzing and querying metadata [11], [13].

We envision a scalable metadata service with two goals. The first goal – *evolution, not revolution* – emphasizes the need for a solution that adds new support to existing cluster file systems that lack a scalable metadata path. Although newer cluster file systems, including Google's Colossus file system [9], OrangeFS [16], UCSC's Ceph [27] and Copernicus [12], promise a distributed metadata service, it is undesirable to replace existing cluster file systems running in large production environments just because their metadata path does not provide the desired scalability or the desired functionality. Several large cluster file system installations, such as Panasas PanFS running at LANL [28] and PVFS running on Argonne BG/P [1], [21], can benefit from a solution that provides, for instance, distributed directory support that does not require any modifications to the running cluster file system. The second goal – *generality and de-specialization* – promises a fully, distributed and scalable metadata service that performs well for ingest, lookups, and scans. In particular, all metadata, including directory entries, i-nodes and block management, should be stored in one structure; this is different from today's file systems that use specialized on-disk structures for each type of metadata.

To realize these goals, this paper makes a case for a scalable metadata service middleware that layers on existing cluster file system deployments and distributes file system metadata, including the namespace tree, small directories and large directories, across many servers. Our key idea is to effectively synthesize a concurrent indexing technique to distribute metadata with a tabular, on-disk representation of all file system metadata.

For distributed indexing, we re-use the concurrent, incremental, hash-based GIGA+ indexing technique [20]. The main shortcoming of the GIGA+ prototype is that splitting the metadata partitions for better load-balancing involves migrating the directory entries and the associated file data [20]. This is inefficient for HPC systems where files can be gigabytes or more in size. Our middleware avoids this data migration by interpreting directory entries as symbolic links: each directory entry (the name created by the application) has a physical pathname that points to a file in the underlying cluster file system that stores the contents of the file. This representation of directory entries is enabled through the use a novel on-disk metadata representation based on a log-structure merge tree (LSM-tree) data structure [18], [22]. We use the LevelDB key-value store to implement all file system metadata, including files, directories, and their i-node attributes, in flat files sorted on a unique key [14]. This organization facilitates high-speed metadata creation, lookups and scans.

Effectively integrating the LevelDB-based metadata store with the distributed indexing technique requires several optimizations including cross-server split operations with minimum data migration, and decoupling data and metadata paths. To demonstrate the feasibility of our approach, we implemented a prototype middleware layer using the FUSE file system and evaluated it on 64-node cluster. Preliminary results show promising scalability and performance: the single-node local metadata store was 10X faster than modern local file systems and the distributed middleware metadata service scaled well with a peak performance of 190,000 file creates per second on a 64-server configuration.

II. Design and implementation

Figure 1 shows the architecture of our scalable metadata service that is designed to be layered on existing deployments of cluster file systems. Our approach uses a client-server architecture and has three components: unmodified applications running on clients, the GIGA+ directory indexing service on clients and servers, and the LevelDB-based persistent metadata representation managed by the server. Applications interact with our middleware using the VFS interface exposed

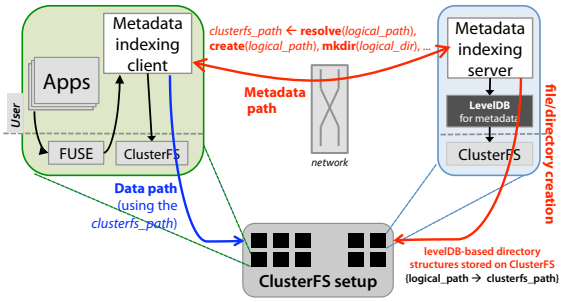

Fig. 1. Design of our scalable metadata middleware that integrates a distributed metadata indexing technique with a tabular metadata-optimized on-disk layout on each server and layers on existing cluster file systems.

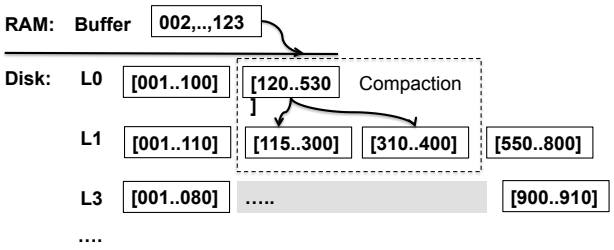

Fig. 2. LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs.

through the FUSE user-level file system [3]. All metadata requests, such as `create()`, `mkdir()` and `open()`, are handled through the GIGA+ indexing modules that address the request to the appropriate server. Each indexing server manages its local LevelDB instance to store and access all metadata information. This LevelDB instance stores flat files (in its special format) containing changes in metadata. Once the client receives the relevant metadata back from the server, our middleware allows clients to access the actual file contents directly through the cluster file system.

Using GIGA+ and LevelDB enables us to tackle two key challenges: highly concurrent metadata distribution for ingest-intensive parallel applications such as checkpointing [6] and optimized metadata representation that stores all file system metadata in structured, indexed files managed by existing cluster file system deployments [14].

Remainder of this section describes more details of our approach. Section II-A presents a primer on how GIGA+ distributes metadata. Section II-B shows how LevelDB stores all file system metadata using a single on-disk structure on each server. Section II-C describes the challenges in effectively integrating GIGA+ and LevelDB to work with existing cluster file systems.

A. Scalable partitioning using GIGA+

GIGA+ is a distributed hash-based indexing technique that incrementally divides each directory into multiple partitions that are spread over multiple servers [20]. Each filename stored in a directory entry is hashed and mapped to a partition using an index. GIGA+ selects a hash partition such that for any distribution of unique filenames, the hash values of these filenames will be uniformly distributed in the hash space. In addition to load-balanced distribution, GIGA+ also grows the directory index incrementally, i.e. all directories start small on a single server, and then expand to more servers as they grow in size.

The core idea behind GIGA+ is parallel splitting: each server splits without system-wide serialization or synchronization. Every server makes a local decision, without coordinating with other servers, about when to split a partition. Such uncoordinated growth causes GIGA+ servers to have a partial view of the entire index; there is no central server that holds the global view of the partition-to-server mapping. Each server knows about the partition it stores and the identity of another server that knows more about each "child" partition resulting from a prior split by this server. This information is known as the per-server split history of its partitions. The full GIGA+ index is a transitive closure of the split history on each server and represents the lineage of directory partitioning.

The full index (and split history) is also not maintained synchronously by any client. GIGA+ clients can enumerate the partitions of a directory by traversing its split histories starting with the first partition that was created during `mkdir`. However, such a full index that is cached by a client may be stale at any time, particularly for rapidly mutating directories. GIGA+ allows clients to keep using the stale mapping information and receiving mapping updates from servers. More discussion on the cost-benefit of using inconsistent mapping state is not relevant to this work and can be found in prior GIGA+ literature [19], [20].

B. Metadata layout using LevelDB

LevelDB [14] is an open-source key-value storage library that features a log-structured merge (LSM) Tree [18] for on-disk storage. LevelDB is inspired by the per-server tablet architecture in BigTable [4]. In a simple understanding of an LSM tree, an in-memory buffer cache delays writing new and changed entries until it has a significant amount of change to record on disk. Using LevelDB as a local storage representation for metadata can transform metadata updates to large, non-overwrite, sorted and indexed logs on disks, which greatly reduces random disk seeks. The detailed design of LevelDB and how to use LevelDB to store metadata is explained in the following sections.

LevelDB and LSM trees – In LevelDB, by default, a set of changes are spilled to disk when the total size of modified entries exceeds 4 MB. When a spill is triggered, called a minor compaction, the changed entries are sorted, indexed and written to disk in a format called an SSTable[4]. These entries may then be discarded by the in memory buffer and can be reloaded by searching each SSTable on disk, possibly stopping

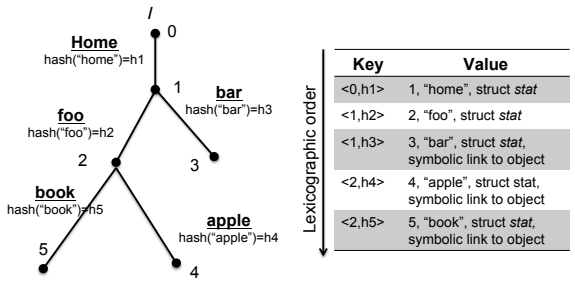

Fig. 3. An example illustrating a table schema for storing metadata into LevelDB.

when the first match occurs if the SSTables are searched most recent to oldest. The number of SSTables that need to be searched can be reduced by maintaining a Bloom filter[7] on each, but, with time, the cost of finding a record not in memory still increases. Major compaction, or simply "compaction", is the process of combining multiple SSTables into a smaller number of SSTables by merge sort.

As illustrated in Figure 2, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into sets, or levels. In 0-th level, each SSTable may contain entries with any key value, based on what was in memory at the time of its spill. The higher levels of LevelDB's SSTables are the results of compacting SSTables from their own or lower levels. In these higher levels, LevelDB maintains the following invariant: the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level. So querying for an entry in the higher levels only need read at most one SSTable in each level. LevelDB also sizes each of the higher levels differentially: all SSTables have the same maximum size and the sum of the sizes of all SSTables at level L will not exceed 10^L MB. This ensures that the number of level grows logarithmically with increasing numbers of entries. LevelDB compactions are similar to streaming B-trees used in TokuDB products [5], [8].

Table schema – The local metadata store aggregates directory entries and inode attributes into one LevelDB table with a row for each file and directory. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered by a 224-bit key consisting of the 64-bit inode number of a file's parent directory and a 160-bit SHA-1 hash value of its filename string (final component of its pathname). The value of a row contains the file's full name and inode attributes, such as inode number, ownership, access mode, file size, timestamps (*struct stat* in Linux), and a symbolic link that contains the actual path of the file object in the object store. Figure 3 shows an example of storing a sample file system's metadata into one LevelDB table.

All the entries in the same directory have rows that share the same first 64 bits in their the table's key. For $readdir$ operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's inode number as the first 64 bits of their table's key. To resolve a single pathname, the metadata server starts searching from the root inode, which has a well-known global inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of current directory with the hash of next component name in the pathname.

C. Integrating GIGA+ and LevelDB

To effectively integrate the GIGA+ distribution mechanism with the LevelDB-based metadata representation, we tackled several challenges.

Metadata representation – LevelDB stores all metadata including GIGA+ hash partitions for directories, entries in each hash partition, and other bootstrapping information such as root entry and GIGA+ configuration state. The general schema used to store all file is:

```
    <KEY>          -->      <VALUE>

{parentDirID,           {attr(dirEntry),
 gigaPartitionID, -->    symlink,
 hash(dirEntry),         gigaMetaState}
 dirEntry}
```

The main difference from the LevelDB schema described in Section II-B is the addition of two GIGA+ specific fields: `gigaPartitionID` to identify a GIGA+ hash partition and `gigaMetaState` to store the hash partition related mapping information. These GIGA+ related fields are used only if large directories are distributed over multiple metadata servers.[1]

Partition splitting – Each GIGA+ hash partition and its directory entries are stored in SSTable files in a local LevelDB instance. Recall that each GIGA+ server process splits a hash partition P on overflow and creates another hash partition P' which is managed by a different server; this split involves migrating approximately half the entries from old partition P to the new hash partition P' on another server during which the key range in write is locked. We explored several ways to perform this cross-server partition split.

A simple approach to splitting would be to perform a LevelDB range scan on partition P and deleting about half the results (corresponding to the keys that are migrated to the new partition) from P. All entries that need to be moved to the new partition P' are batched together and sent in an RPC message to the server that will manage partition P'. The recipient server inserts each key in the batch in its own LevelDB instance. While the simplicity of this approach makes it attractive, we would like a faster technique to reduce the time that the range is write locked.

[1]Since we already store the `hash` of the directory entry, we can use the hash-values to identify hash partitions if we chose to use the same hash function for both GIGA+ and LevelDB keys. This optimization can eliminate the need for `gigaPartitionID` in the schema.

The immutability of LevelDB SSTables makes such a fast bulk insert possible – an SSTable can be added to Level 0 without its data being pushed through the write-ahead log and minor compaction process. To take advantage of this opportunity, we extended LevelDB to support a three-phase split operation. First, the split initiator performs a range scan on its LevelDB instance to find all entries in the hash-range that needs to be moved to another server. The results of this scan are written in a LevelDB-specific SSTable format to file in the underlying cluster file system. In the second step, the split initiator notifies the split receiver about the new LevelDB-format file in a much smaller RPC message. The split receiver then bulk inserts the file into the LevelDB tree structure instead of iteratively inserting one key at a time. The final step is a clean-up and commit phase: after the receiver completes the bulk insert operation, it notifies the initiator, who then deletes the migrated hash-range from its LevelDB instance and unlocks the range.

Decoupled data and metadata path – All metadata operations go through the GIGA+ server; however, following the same path for data operations would incur an unnecessary performance penalty of shipping data over the network on extra time. This penalty can be significant in HPC use-cases where files can easily be gigabytes to terabytes in size.

To avoid this penalty our middleware is designed to perform all data-path operations directly through the cluster file system module in client machine. Figure 1 illustrates this data path (in BLUE color). Once the client completes a lookup on a desired file name, it gets back a symbolic link to the physical path in the cluster file system. All subsequent accesses using this symbolic link force the client operating system to resolve this link into the cluster file system. While the file is open, some of its attributes (e.g., file size and last access time) may change relative to LevelDB's per-open copy of the attributes. GIGA+ will capture these changes on file close on the metadata path. Other attribute changes relatvie to permissions can be updated on-flight through GIGA+ servers.

III. Preliminary Evaluation

We built a FUSE-based middleware filesystem prototype and tested its metadata performance on a 64-node cluster of dual core machines with 16GB memory interconnected with one GigE NIC. Each node had a GIGA+ indexing server process that managed its own LevelDB instance that was stored on a local disk running Linux Ext3 file system. This hardware does not have HPC-class networking or cluster file system, so our preliminary experiments exclusively used metadata-intensive microbenchmarks. To emulate shared storage for split operations, we used a NFS-mounted volume accessible from all machines; this volume was only used for our cross-server LevelDB split optimization. We evaluated the performance of a single-node LevelDB-based metadata store and the scalability of our distributed middleware on 64 nodes.

We first evaluated the performance of a single-node LevelDB-based metadata store by running a test that creates

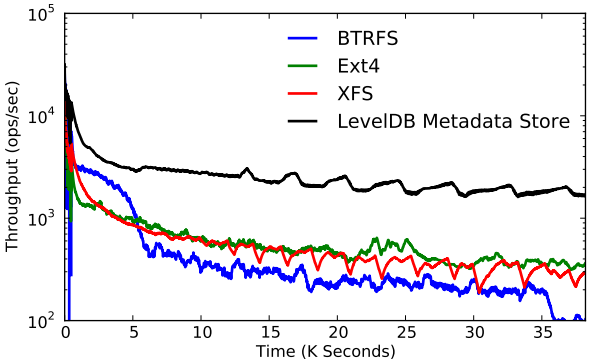

Fig. 4. Single-node LevelDB-based metadata store is 10X faster than modern Linux filesystems for a workload that creates 100 million zero-length files. X-axis only shows the time until LevelDB finished all insertions because the other file systems were much slower. Y-axis has a logarithmic scale.

100 million zero-length files in a single directory. Figure 4 compares the instantaneous throughput of LevelDB-based metadata store with three Linux file systems: Ext4 [15], XFS [26], and BTRFS [23]. All systems perform well at the beginning of the test, but the file create throughput drops gradually for all systems. BTRFS suffers the most serious throughput drop, slowing down to 100 operations per second. The LevelDB-based store, however, maintains a more steady performance with an average speed of 2,200 operations per second respectively, *and is 10X faster than all other tested file systems.*

Next, we evaluated the scalability of our distributed metadata middleware prototype. Figure 5 shows the instantaneous throughput during the *concurrent create* workload in a strong scaling experiment, i.e. creating 1 million files per server, for a total of 64 million files in the 64-server configuration. The main result in this figure is that as the number of servers doubles the throughput of the system also scales up. With 64 servers, GIGA+ can achieve a peak throughput of about 190,000 file creates per second. The prototype delivers peak performance after the directory workload has been spread among all servers. Reaching steady-state, the throughput quickly grows due to the splitting policies adopted by GIGA+.

After reaching the steady state, throughput slowly drops as LevelDB builds a larger metadata store. In fact, in large setups with 8 or more servers, the peak throughput drops by as much as 25% (in case of the 64-server setup). This is because when there are more entries already existing in LevelDB, it requires more compaction work to maintain LevelDB invariants and to perform a negative lookup before each create has to search more SSTables on disk. In theory, the work of inserting a new entry to a LSM-tree is $O(\log_B(n))$ where n is the total number of inserted entries, and B is a constant factor proportional to the average number of entries transferred in each disk request [5]. Thus we can use the formula $\frac{a \cdot S + b}{\log T}$ to approximate the

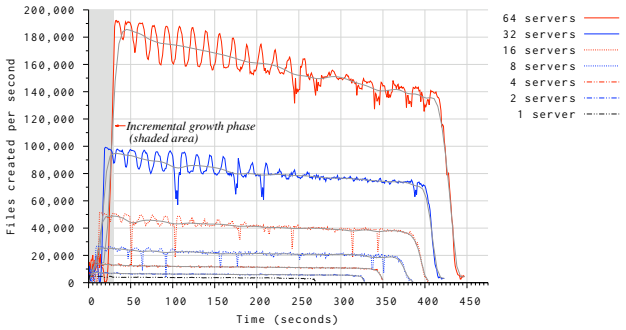

Fig. 5. Our middleware metadata service prototype shows promising scalability up to 64 servers. Note that at the end of the experiment, the throughput drops to zero because clients stop creating files as they finish 1 million files per client. And the solid lines in each configuration are Bezier curves to smooth the variability.

throughput timeline in Figure 5, where S is the number of servers, T is the running time, and a as well as b are constant factors relative to the disk speed and splitting overhead. This estimation projects that when inserting 64 billion files with 64 servers, the system may deliver an average of 1,000 operations per second per server, i.e. 64,000 operations per second in aggregate.

IV. Future Challenges

Before our work can be useful in real HPC deployments, we need to address several issues.

First, we will layer our middleware on top of a real cluster file system. This will allow us to inherit the data path scalability when accessing the file data as well as LevelDB's SSTables. We also plan to explore how we can effectively leverage the fault tolerance mechanisms and system configuration tools already present in the cluster file systems.

Second, we will minimize the FUSE overheads associated with accessing files. Even after the application gets a symbolic link pointing to the physical location of the file, our current prototype will rely of FUSE and VFS to dereference the symbolic link. We want to avoid this FUSE interposition by changing the FUSE kernel module to support distributed file system file handles but still receive prompt notifications of attribute changes on file close.

Third, we will explore several recently published optimizations to minimize the background compaction operations triggered by data stores built on the LSM trees and similar data-structures [5], [8], [25]. Compactions are a necessary evil: in order to speed up future reads and scans, the steal resources from foreground operations that happen simultaneously with these background operations. We want to explore heuristics that can minimize the impact of foreground operations for metadata-specific workloads.

V. Summary

Modern cluster file systems provide highly scalable I/O bandwidth along the data path by enabling highly parallel access to file data. Unfortunately metadata scaling is lagging behind data scaling. We propose a file system design that inherits the scalable data bandwidth of existing cluster file systems and adds support for distributed and high-performance metadata operations. Our key idea is to integrate a distributed indexing mechanism with general-purpose optimized on-disk metadata store. Early prototype evaluation shows that our approach outperforms popular Linux local file systems and scales well with large numbers of file creations.

Acknowledgment

This research is supported in part by The Gordon and Betty Moore Foundation, NSF under award, SCI-0430781 and CCF-1019104, Qatar National Research Foundation 09-1116-1-172, DOE/Los Alamos National Laboratory, under contract number DE-AC52- 06NA25396/161465-1, by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by gifts from Actifio, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, NEC, NetApp, Oracle, Panasas, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital. We thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

References

[1] BG/P File Systems. https://www.alcf.anl.gov/resource-guides/bgp-file-systems.
[2] File System Metadata Management in ISSDM. https://issdm.soe.ucsc.edu/node/242.
[3] FUSE. http://fuse.sourceforge.net/.
[4] Fay Chang and. BigTable: a distributed storage system for structured data. In *OSDI*, 2006.
[5] Michael Bender and et al. Cache-oblivious streaming B-trees. In *SPAA*, 2007.
[6] John Bent and et al. PLFS: a checkpoint filesystem for parallel applications. In *SC*, 2009.
[7] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.
[8] John Esmet and et al. The TokuFS streaming file system. *HotStorage*, 2012.
[9] Andrew Fikes. Storage Architecture and Challenges (Jun 2010). Talk at the Google Faculty Summit 2010.
[10] HDFS. Hadoop file system. http://hadoop.apache.org/.
[11] Stephanie Jones and et al. Easing the burdens of HPC file management. 2011.
[12] Andrew Leung and et al. Copernicus: A scalable, high-performance semantic file system. Technical Report UCSC-SSRC-09-06, University of California, Santa Cruz, 2009.
[13] Andrew Leung and et al. Magellan: A searchable metadata architecture for large-scale file systems. Technical Report UCSC-SSRC-09-07, University of California, Santa Cruz, 2009.
[14] LevelDB. A fast and lightweight key/value database library. http://code.google.com/p/leveldb/.
[15] Avantika Mathur and et al. The new EXT4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.
[16] Micheal Moore and et al. OrangeFS: Advancing PVFS. *FAST Poster Session*, 2011.
[17] Henry Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf, 2008.
[18] Patrick O'Neil and et al. The log-structured merge-tree. *Acta Informatica*, 1996.

[19] Swapnil Patil and et al. Giga+: scalable directories for shared file systems. In *PDSW*, 2007.
[20] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST*, 2011.
[21] PVFS2. Parallel Virtual File System, Version 2. http://www.pvfs2.org.
[22] Kai Ren and Garth Gibson. TableFS: Enhancing Metadata Efficiency in the Local File System. *CMU Parallel Data Laboratory Technical Report CMU-PDL-12-110*, 2012.
[23] Ohad Rodeh and et al. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.
[24] Rob Ross and et al. High End Computing Revitalization Task Force (HECRTF), Inter Agency Working Group (HECIWG) File Systems and I/O Research Guidance Workshop 2006. http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf, 2006.
[25] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *SIGMOD*, 2012.
[26] Adam Sweeney and et al. Scalability in the XFS file system. In *USENIX ATC*, 1996.
[27] Sage A. Weil and et al. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*, 2006.
[28] Brent Welch and et al. Scalable Performance of the Panasas Parallel File System. In *FAST*, 2008.