Building a High-Performance Metadata Service by Reusing Scalable I/O Bandwidth

Kai Ren, Swapnil Patil, Kartik Kulkarni, Adit Madan, Garth Gibson
({kair, svp}@cs.cmu.edu, {kartikku, aditm}@andrew.cmu.edu, garth@cs.cmu.edu)

CMU-PDL-13-107
May 2013

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Modern parallel and cluster file systems provide highly scalable I/O bandwidth by enabling highly parallel access to file data. Unfortunately metadata access does not benefit from parallel data transfer, so metadata performance scaling is less common. To support metadata-intensive workloads, we offer a middleware design that layers on top of existing cluster file systems, adds support for load balanced and high-performance metadata operations without sacrificing data bandwidth. The core idea is to integrate a distributed indexing mechanism with a metadata optimized on-disk Log-Structured Merge tree layout. The integration requires several optimizations including cross-server split operations with minimum data migration, and decoupling of data and metadata paths. To demonstrate the feasibility of our approach, we implemented a prototype middleware layer GIGA+TableFS and evaluated it with a Panasas parallel file system. GIGA+TableFS improves metadata performance of PanFS by as much an order of magnitude, while still performing comparably on data-intensive workloads.

Acknowledgements: This research is supported in part by The Gordon and Betty Moore Foundation, the University of Washington eScience Institute, 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 Yahoo!, APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware. We thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

Keywords: Parallel File System, Metadata, Load Balance, Log-Structured Merge Tree

1 Introduction

Lack of a highly scalable and parallel metadata service is the Achilles heel for many parallel and cluster file systems in both the HPC world [22, 30] and the Internet services world [12]. This is because most cluster file systems have used centralized, single-node metadata management, and focused merely on scaling the data path, i.e. providing high bandwidth parallel I/O to files that are gigabytes in size.

This inherent metadata scalability handicap limits numerous massively parallel applications that produce workloads requiring concurrent and high-performance metadata operations. One such example, file-per-process checkpointing, requires the metadata service to handle a huge number of file creates at the same time [5]. Another example, storage management, produces a read-intensive metadata workload that typically scans the metadata of the entire file system to perform administration tasks [14, 16]. Thirdly, even in the era of big data, most things in even the largest cluster file systems are small [8, 41]. Scalable storage systems should expect the numbers of small files stored to soon achieve and exceed billions, a known challenge for many existing cluster file systems [27].

We envision a scalable metadata service with two goals. The first goal – *evolution, not revolution* – emphasizes the need for incremental improvements to existing cluster file systems that lack a scalable metadata path. Although newer cluster file systems, including Google's Colossus file system [10], OrangeFS [21], UCSC's Ceph [39] and Copernicus [15], promise an entirely new distributed metadata service, it is undesirable to have to replace an existing cluster file system running in a large production environment just because their metadata path does not provide the desired scalability. Several large cluster file system installations, such as Panasas PanFS running at LANL [40] and Lustre running at Oak Ridge [35, 38], 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 creates, deletes, 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 middleware service GIGA+TableFS that layers on top of existing cluster file systems and load balance file system metadata, including the namespace, small directories and large directories, across many servers. Our key idea is to effectively integrate a concurrent indexing technique to distribute metadata with a tabular, log-structured on-disk representation of all file system metadata.

For distributed indexing, we re-use the concurrent, incremental, hash-based GIGA+ indexing technique [27]. The main shortcoming of the original GIGA+ prototype was the cost of splitting metadata partitions for better load-balancing. This required migrating directory entries and associated file data [27] from one storage server to another. This is inefficient for HPC systems where files can be gigabytes or more in size. Our GIGA+TableFS 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 with the actual file content stored in the underlying cluster file system.

The need for a new representation of directory entries led us to develop of a novel on-disk metadata representation called TableFS [28], based on a log-structure merge tree (LSM-tree) data structure [24]. We use the TableFS approach to pack all file system metadata (including directories, i-node attributes) and small files, into many fewer, large, flat files. This organization facilitates high-speed metadata creation, lookups and scans, even in a single computer local disk configuration [28].

Effectively integrating the TableFS metadata store with the GIGA+ 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 GIGA+TableFS and evaluated it on an existing Panasas PanFS deployment [40]

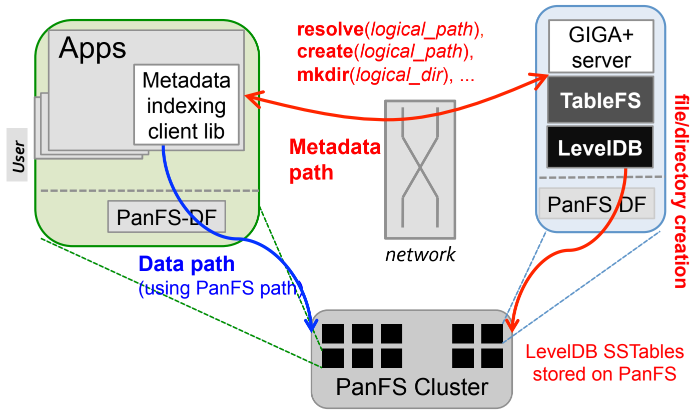


Figure 1: *Our scalable metadata service integrates two components: GIGA+ [27], a highly parallel and load-balanced indexing technique to partition metadata over multiple servers, and TableFS [28], an log-structured (LevelDB) on-disk metadata representation on each server. This integrated solution is layered on top of an existing cluster file system deployment (PanFS) to improve metadata and small file operation efficiency.*

that has 5 shelves consisting of 5 metadata servers and 50 data servers. We called the combined solution PanFS-GT. Our results show promising scalability and performance: GIGA+TableFS layering on top of PanFS (PanFS-GT) was more than 10× faster than the original PanFS for metadata intensive workloads, and performs comparably for data-intensive workloads.

2 Design and implementation

Figure 1 presents the overall architecture of our scalable metadata service. Our metadata service is a middleware inserted into existing deployments of cluster file systems to improve metadata efficiency while maintaining high I/O bandwidth for data transfers. The system uses a client-server architecture, and consists of three core components:

- **Client:** Applications interact with our middleware through the FUSE user-level file system [1], through a library directly linked into the application or through a module in a common library such as MPI-IO [7]. The stateless client-side code redirects applications' file operations to the appropriate destination according to the types of operations. All metadata requests (e.g. `create()` and `mkdir()`), and data requests on small files (e.g. `read()` and `write()`), are handled by the metadata indexing modules that address these requests to the appropriate server. For all data operations on large size files, the client code redirects the request directly to the underlying cluster file system to take full advantage of data I/O bandwidth. A newly created, but growing file may be transparently reopened by the client module.
- **Metadata Indexing Server:** Each indexing server manages its local metadata storage backend to store and access all metadata information and small file data. It uses the GIGA+ algorithm to partition large directories across indexing servers. It also monitors the growth of small files, and migrates newly large files into the underlying cluster file system when its size exceeds a threshold.
- **Metadata Storage Backend:** The metadata storage backend is a modified version of TableFS which packs metadata and small file data into large, log-structured, flat files, and stores these files in the underlying cluster file system. Since TableFS converts random updates into sequential writes, it greatly

improves disk performance. In order to dynamically redistribute large directories, the metadata storage backend also modifies TableFS to support exporting and importing flat files in batch.

Remainder of this section describes more details of our system. Section 2.1 presents a primer on how GIGA+ distributes metadata. Section 2.2 shows how TableFS stores all file system metadata and small files using a single on-disk structure on each server. Section 2.3 focus on the challenges in effectively integrating GIGA+ and TableFS to work with existing cluster file systems.

2.1 Distributed Metadata Indexing

GIGA+ is a distributed hash-based indexing technique that incrementally divides each directory into multiple partitions that are spread over multiple servers [27]. Each filename stored in a directory entry is hashed and mapped to a partition using a per-directory 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. This makes load balancing much easier. 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 number of entries.

A core scalability idea in GIGA+ is parallel splitting: each server splits without system-wide serialization or synchronization. Such uncoordinated growth causes GIGA+ servers to have only a partial view of the entire per-directory index; there is no central server that holds the global view of the partition-to-server mapping. Each server knows about the partitions it stores and knows the identity of other server that know more about each “child” partition resulting from a prior split by this server. This information is known as the per-server split history of a directory’s partitions. The full per-directory GIGA+ index is a transitive closure of the split history on each server and represents the lineage of the directory’s partitioning.

The full index (and split history) of a directory is also not maintained synchronously by any client. GIGA+ clients can search through the partitions of a directory by traversing its split histories starting with the first partition that was created during `mkdir` and clients can cache what they learn. However, such an opportunistically cached index by a client may be incomplete or stale at any time, particularly for rapidly mutating directories. GIGA+ allows clients to keep using stale mapping information because addressed servers verify and brief to clients as needed. More discussion of the cost-benefit of using inconsistent mapping state is not relevant to this work and can be found in prior GIGA+ literature [25, 27].

2.2 Metadata Storage Backend

Our metadata storage backend implements a modified version of TableFS to pack together and manage all the metadata and small files, hiding them from the underlying cluster file system.

TableFS [28] is a stacked file system which uses another file system as an object store, and organizes all metadata and small files into a single on-disk table using a Log-Structured Merge (LSM) tree [24, 17]. The reason for using LSM tree is that it buffers new and changed entries in memory and translate small random disk writes into large sequential writes. Because LSM tree can dramatically reduce random disk seeks, it is a natural fit for metadata intensive workloads. We decribe the structure of an LSM tree and how LSM trees are used in TableFS in greater detail in the following sections.

LSM trees and LevelDB – TableFS uses an open-source implementation of an LSM tree called LevelDB [17]. LevelDB provides a simple key-value store interface, supporting point queries and range queries. 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 [3]. These entries may then be discarded from the in memory buffer. Discarded entries can be reloaded by searching each SSTable on disk, possibly stopping when the

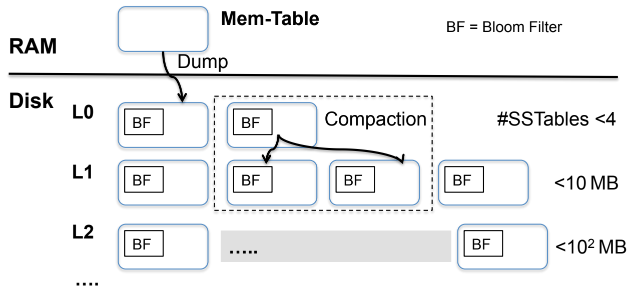


Figure 2: *LevelDB is an on-disk LSM-tree implementation that represents data in multiple files (called SSTables) containing sorted key-value pairs. SSTables are grouped into different levels with lower-numbered levels containing more recently inserted key-value pairs. Finding a specific pair on disk may search up to all SSTables in level-0 and at most one in each higher-numbered level. Compaction is the process of combining SSTables by merge sort and moving combined SSTables into higher-numbered levels.*

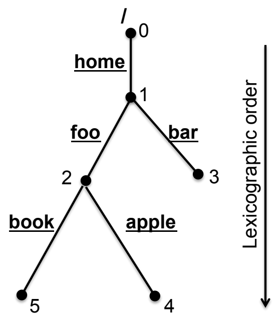
first match occurs if the SSTables are searched most recent to oldest. The number of SSTables that need to be searched is reduced by maintaining the minimum and maximum key value and a Bloom filter[6] on each, but, with time, the cost of finding a record not in memory still increases. Major compaction, or simply "compaction", is the process of combining multiple overlapping range SSTables into a number of disjoint range SSTables by merge sort.

As illustrated in Figure 2, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into 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-numbered levels of LevelDB's SSTables are the results of compacting SSTables from their own or lower-numbered levels. In levels excepth the 0-th level, 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 needs to 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.

Table schema – TableFS (prior to GIGA+TableFS) aggregates directory entries, i-node attributes and small files into one LSM tree with an entry for each file and directory. To translate the hierarchical structure of the file system namespace into key-value pairs, the 224-bit key is chosen to consist of the 64-bit i-node number of a entry's parent directory and a 160-bit SHA-1 hash value of its filename string (final component of its pathname). The value of an entry contains the file's full name and i-node attributes, such as i-node number, ownership, access mode, file size, timestamps (*struct stat* in Linux). For small files with size less than T (defaulting to 4KB), the value field also contains the file's data. For large files, the file data field in a file row of the table is replaced by a symbolic link that points to the actual file object in the underlying file system.

Figure 3 shows an example of representing a sample file system's metadata into one table. Embedding the i-node number of parent directory into the key helps with resolving the pathname. Traversing the user's directory tree involves constructing a search key by concatenating the i-node number of current directory with the hash of next component name in the pathname. Another benefit of this schema is that all the entries in the same directory have rows that share the same first 64 bits in their row key. For *readdir* operations, once the i-node number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's i-node number as the first 64 bits of their table's key.

Previous evaluation [28] has shown using the TableFS schema with a LevelDB backend can greatly



Key	Value
<0,hash(home)>	1, "home", struct *stat*
<1,hash(foo)>	2, "foo", struct *stat*
<1,hash(bar)>	3, "bar", struct *stat*
<2,hash(book)>	5, "book", struct *stat*, inline small file(<*T*)
<2,hash(apple)>	4, struct *stat*, large file pointer (>*T*)

Figure 3: *An example illustrating a table schema for TableFS to store metadata and file data as key-value pairs.*

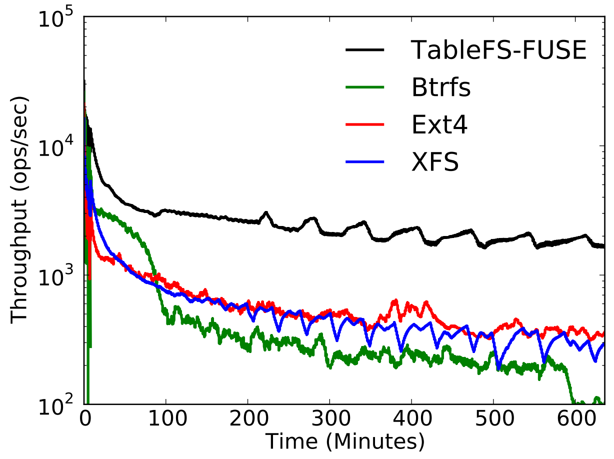


Figure 4: *This graph (from prior work [28]) shows that the performance of single-node TableFS with FUSE is 10X faster than modern Linux filesystems. The workload is to create 100 million zero-length files in one directory. X-axis only shows the time until TableFS finished all insertions because the other file systems were much slower. Y-axis has a logarithmic scale.*

improve metadata performance of a local file system. Figure 4 is taken from a prior TableFS paper, which compared the instantaneous single disk throughput of a FUSE-based TableFS with three underlying Linux file systems: Ext4 [19], XFS [36], and Btrfs [29]. The workload created 100 million zero-length files in a single directory on one disk. All systems perform well at the very beginning of the test, but the file create throughput drops significantly for all systems as the total data gets larger. Btrfs suffers the most serious throughput drop, slowing down to 100 operations per second. TableFS, 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.*

2.3 Integrating GIGA+ and TableFS

To effectively build GIGA+TableFS middleware that integrates the GIGA+ distribution mechanism and the TableFS metadata representation, we have to tackle two main challenges: modify TableFS to support for migrating directory partitions as required by GIGA+; and decouple metadata and data paths to achieve high performance for both paths. This section discusses the modifications we made to overcome these two challenges.

Metadata representation – TableFS stores all additional metadata including GIGA+ per-directory hash partitions, entries in each hash partition, and GIGA+ bootstrapping information such as root partition entry and GIGA+ configuration state. The general schema used to store all metadata is:

key	`parentDirID,gigaPartitionID,hash(dirEntry)`
value	`attributes, symlink|data|gigaMetaState]`

The main difference from the TableFS schema described in Section 2.2 is the addition of two GIGA+ specific fields: `gigaPartitionID` to identify a GIGA+ hash partition within a directory and `gigaMetaState` to store the hash partition related mapping information for directories. These GIGA+ related fields are used only if a large directory is large enough to be distributed over multiple metadata servers. [1]

Partition splitting – Each server's TableFS instance stores metadata, including GIGA+ directory partitions and their directory entries in LevelDB which stores them as a set of files (whose format is called SSTable) in a server specific directory in the underlying cluster file system. Each GIGA+ server process splits a large partition P on into itself and 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 partition P' on another server. During splitting, the partition in migration is locked against client for simplification. We explored several ways to perform this cross-server partition split.

A straightforward solution would be to perform a range scan on partition P, and remove about half the entries (that will be migrated to the new partition P') from P. All removed entries would then be batched together and sent in a large RPC message to the server that will manage partition P'. The split receiver would insert each key in the batch into its own TableFS instance. While simplicity of this solution makes it attractive, it is slow in practice and vulnerable to failures during splitting.

We have devised a faster and safer technique to reduce the time that the splitting range is locked. The immutability of SSTables in LevelDB makes a fast bulk insert possible – an SSTable whose range does not overlap any part of a current LSM tree 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 TableFS to support a three-phase GIGA+ split operation:

- Phase 1: The split initiator locks and then performs a range scan on its TableFS instance to find all entries in the hash-range that needs to be moved to another server. Instead of packing these into an RPC message, the results of this scan are written in SSTable format to a file in the underlying cluster file system.

- Phase 2: The split initiator notifies the split receiver about the path to the SSTable-format split file in a much smaller RPC message. Since this file is stored in shared storage, the split receiver directly inserts this file as a symbolic link into the LevelDB tree structure without actually copying this file. The insertion of this file into the split receiver is the commit part of the split transaction.

[1]A optimization is to eliminate `gigaPartitionID` in the key by using the same hash function for both GIGA+ and TableFS keys, since the hash of entry name can determine the partition ID.

- Phase 3: The final step is a clean-up phase: after the split receiver completes the bulk insert operation, it notifies the initiator, who then deletes the migrated key-range from its TableFS instance and unlocks the range. [2]

Decoupled data and metadata path – All metadata and small file operations go to the GIGA+ server; however, following the same path for data operations on large files would incur a large and unnecessary performance penalty from shipping data through a single busy machine and over the network an extra time. This penalty can be significant in HPC use-cases because large files are easily be gigabytes to terabytes and approaching petabyte in size.

To avoid this penalty our middleware is designed to perform all data-path operations on large files directly through the cluster file system module in client machine. Figure 1 illustrates this data path (BLUE colored lines). Once the application opens a file with size greater than T, the GIGA+TableFS client library code will get back a symbolic link to the physical path in the cluster file system, which it will open locally. All subsequent accesses to this large file will go through the client operating system to the native cluster file system in parallel transfers. Thus applications should achieve the same data bandwidth as clients access to the underlying cluster file system.

FUSE-based clients can use the same trick but with additional overhead of double context switching and memory copying [5]. Recent work [13] shows that the FUSE kernel module can be modified to make performance degradation less than 3%. We have not implemented this modification in our current prototype yet, because FUSE is not preferred for HPC parallel codes, and is mostly used for external user access and storage management (e.g. *ls*, *mkdir*, $find$).

While a file is open, some of its attributes (e.g., file size and last access time) may change in the cluster file system relative to TableFS's pre-open and stale copy of the file's attributes. GIGA+TableFS monitors what large files are currently open. For attribute queries to open files, GIGA+TableFS will directly query the underlying cluster file system to get the most up-to-date attribute values. Later on or after the file is closed, GIGA+TableFS will synchronize both copies of these attributes. [3] By doing so, GIGA+TableFS servers can achieve the same level of metadata consistency as provided by the underlying cluster file system.

Layering on the Panasas file system – In PanFS [40], the *Volume* is the basic unit administrators use to manage the storage pool of PanFS. The volume is a directory hierarchy with a quota limit, and appears as a directory below the single mount point for the whole storage system. Each PanFS volume is only managed by a single metadata manager, although the data of all files created in all volumes are spread over the entire storage pool.

A typical PanFS storage cluster has multiple shelves, each shelf consisting of one or two metadata managers and 9 or 10 storage nodes. However, since a volume is assigned to a particular metadata manager, all the metadata accesses to any file/directory in a volume can only be served by that single metadata manager.

Since GIGA+TableFS distributes a single namespace over all GIGA+TableFS servers at fine granularity (the partition of a directory or an entire small directory), all the metadata managers of PanFS can be exploited and load balanced if each is responsible for volumes backing the same number of GIGA+TableFS servers.

The simple way to do this is to run one GIGA+TableFS server per PanFS server and has the LevelDB instance in a GIGA+TableFS server store its SSTables in a volume specific to that GIGA+TableFS server. Large files can be randomly assigned to hidden directories in all PanFS volumes.

[2] The three phases of splitting can be refined even further: Assume that the splitting is initiated at the time T_{split}. The split initiator can generate SSTables containing entries older than T_{split} without locking the hash range. When the generation of SSTables with entries older than T_{split} is finished, the split initiator can lock the hash range and then write SSTables with newly added or updated entries later than T_{split}. By doing so, the duration of locking splitting hash range can be further reduced. However, due to the code complexity of this optimization, we left this optimization for future work.

[3] Some attribute changes, such as permissions, can be updated by GIGA+TableFS servers.

Fault tolerance – The middleware design and techniques used for scaling the performance of GIGA+TableFS does not impose any major new challenges for handling failures. The primary functions required for fault tolerance include replication of the GIGA+TableFS server's write-ahead logging for file system state changes into the storage of another machine, detection of server failure, failover to a backup server, and replaying of the replicated write-ahead log. These are all standard techniques found in parallel file systems like PanFS. The replication of the write-ahead log may not even be necessary if GIGA+TableFS is layered on a full-featured file system such as PanFS, and its write-ahead log is RAIDed or replicated by the underlying file system.

For directory partition splitting and other operations requiring distributed transactions (e.g. `hardlink` and `rename`), these are implemented as three-phase operations with failure protection from write-ahead logging in source and destination server and eventual garbaging collection of incorrupted, hidden, interrupted states. LevelDB also implements write-ahead logging and atomic batching operation which is used as primitives for fault tolerance by TableFS.

The library version of the GIGA+TableFS client does not bring any new failure properties either. Directory-specific client state recovery is naturally supported by the GIGA+ protocal, which tolerates incomplete or stale state by re-fetching the current server list and rebuilding its directory index cache each time it contact specific servers. For most operations except `open` and `close`, the GIGA+TableFS server does not need to keep state about its clients. Revoking the `open` reference count for a large file can be achieved by maintaining renewable lease on the server side, or by polling the state if the file in the underlying file system using proprietary APIs, if at all. The migration of small files from TableFS to the underlying file system is handled by the GIGA+TableFS server, and is protected by its write-ahead logging and server failure protocol.

3 Experimental Evaluation

This section evaluates the scale and performance of our prototype using a mix of metadata-intensive and data-intensive benchmarks. Our evaluation layers GIGA+TableFS on the Panasas PanFS file system, and focuses on three questions:

- How does GIGA+TableFS enhance metadata throughput, particularly number of file creates per second for concurrent file creation workloads, on the PanFS storage system?
- What is the data throughput of GIGA+TableFS layered on PanFS for N-N checkpointing workload found in HPC systems?
- How portable is GIGA+TableFS to be used on other cluster storage systems and configurations?

3.1 Setup and Methodology

Our prototype is implemented in about 10,000 lines of C code using a modular design comprising of TableFS, LevelDB and PanFS layers. In particular, the PanFS layer is unmodified and can be replaced with any other cluster file system. The current version implements most common POSIX file system operations except `hardlink`, `rename`, and `xattr` related operations. The first two operations, `hardlink` and `rename`, are particularly complicated because they need distributed transaction support for correct and fault tolerant cross-server operations; this problem is beyond the scope of this paper.

	Cluster 1 (Storage cluster)	Cluster 2 (Test applications)
#Nodes	5	64
OS Kernel	CentOS 6.3 2.6.32 x86_64	Ubuntu 12.10 3.6.6 x86_64
CPU	AMD Opteron 6272 64 Cores	AMD Opteron 242 Dual Core
Memory	128GB DDR	16GB DDR
Network	40GE NIC	1GE NIC
Storage System	PanFS 5 Shelves (5 MDS + 50 ODS)	Western Digital Local hard disk 2TB per node
		100 seeks/sec random seeks 137.6 MB/sec seq. reads 135.4 MB/sec seq. writes

Table 1: *Settings of two clusters used for evaluation.*

All experiments were performed on two clusters described in Table 1. The first cluster is a 5-shelf PanFS storage cluster, and the second cluster is a 64-node setup to run GIGA+TableFS code and applications. Table 1 describes the hardware and software configuration of these clusters. The first cluster is used to evaluate the scale and performance of the underlying cluster file system that uses our middleware. The second cluster is used to demonstrate that our middleware solution can be layered on other file system deployments without any configuration changes; this setup emulates a case that each GIGA+ server runs in a different NFS node and manages its own TableFS instance locally to scale the metadata performance of NFS. In all tests, the client uses library version code; the threshold for splitting a partition is always 8,000 entries; and TableFS managed by GIGA+ server syncs its data every 5 seconds.

In the following sections, we will first show the evaluation on the end-to-end performance of the integrated system on top of PanFS, and then present the results of a scaling experiment on another platform.

3.2 End-to-end System Evaluation

The goal of our end-to-end evaluation is to understand how the cluster file system scales with multiple metadata servers. As shown in Table 1, PanFS consists of five shelves; in other words, PanFS has a total of five metadata servers (called director blades) and fifty object servers (called OSD blades). Ideally, a GIGA+TableFS server should be co-located with the metadata server; such a setup will allow the cluster file system to scale metadata performance as more GIGA+TableFS servers and metadata servers are added. However, this would require a few configuration changes to allow GIGA+TableFS server process to run on the PanFS metadata servers.

Because our goal is to avoid any configuration changes, we used five additional test nodes, each with 64 cores and a 40 GigE NIC, that together can saturate the 5-shelf PanFS storage cluster. Each test node has three subsystems: one GIGA+TableFS server, 32 GIGA+TableFS clients and a PanFS DirectFlow (client)

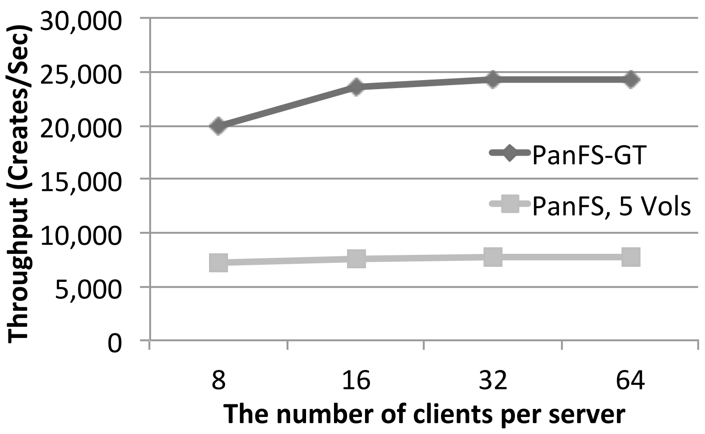


Figure 5: *Average throughput during creating five million zero-length files in one empty directory with different number of clients per test node. Running 32 and more clients per test node is able to saturate PanFS-GT and original PanFS*

module. Each GIGA+TableFS client is a library that is linked with a workload generating application. These clients send file operations to the GIGA+TableFS servers. The GIGA+TableFS server stores and accesses file system metadata through the PanFS DirectFlow module. In addition, the GIGA+TableFS clients can also communicate with the cluster file system through the PanFS DirectFlow module (this path is used to access the file data as described in earlier sections). In the rest of this section, this layering of GIGA+TableFS on PanFS is called PanFS-GT. In summary, the PanFS-GT configuration consists of a total of 160 workload generating application threads (linked with 160 GIGA+TableFS clients) that communicate with 5 GIGA+TableFS servers that use the 5 metadata servers in the underlying PanFS storage cluster.

To generate application workloads, we use two HPC benchmarks that are used widely by vendors and users of parallel file systems. To test the metadata path, we used the *mdtest* synthetic benchmark [2]. To test the data path, we used LANL's File System Test Suite checkpoint benchmark[23].

Metadata Intensive Workloads – We use the synthetic *mdtest* benchmark to generate a three-phase workload: The first phase creates 5 million zero-files in a single shared directory [39, 27]. The second phase performs $stat()$ on random files in this large directory. The third phase deletes all files in this directory in a random order.

We compare the performance of native PanFS and PanFS-GT for this benchmark. However, to ensure a fair comparison we cannot compare them directly. This is because of two reasons: first, a single directory can only utilize one PanFS metadata server, and second, PanFS directories can contain at most 1 million files. For a fair comparison, we compare PanFS-GT with native PanFS that creates 1 million files in 5 different directories owned by 5 different metadata servers.

Figure 5 shows the aggregated throughput during the first phase. We vary the number of clients running in each test node to determine the number of clients needed to saturate both PanFS-GT and native PanFS systems. Both systems achieve the highest aggregated throughput using 32 or more number of clients per node. In all the experiments shown later, by default, we report results with 32 clients per test node. For all cases, PanFS-GT is approximately 3.5 times faster than the native PanFS using 5 volumes. The aggregated peak throughput for the 5-server and 160-client system is about 25,000 file creates per second.

Figure 6 shows the aggregated throughput of different operations during the three-phase *mdtest* workload. In addition to the PanFS-GT and 5-shelf PanFS, this figure also reports the aggregated throughput of

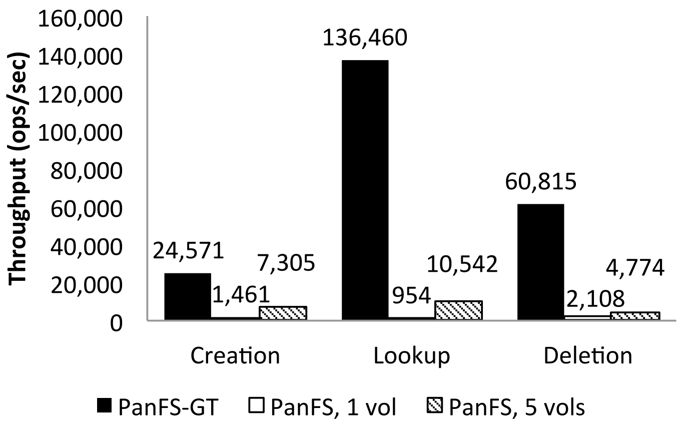


Figure 6: *The average aggregated throughput of different operations in mdtest when generating 5 million zero-length files in a single shared directory. Since PanFS has a hard limit to allow only create 1 million entries in one directory, the bar showing PanFS with 1 volume only gives the average throughput for the case of creating 1 million entries.*

creating 1 million files in single-shelf PanFS. For *lookup* and *deletion* workloads, PanFS-GT outperforms native PanFS by a factor to 10 to 15 higher operations per second. Fast lookup performance results from memory indexing and Bloom filters in LevelDB. For deletion of a key, LevelDB essentially just inserts a new entry with the deletion mark next to the key to be deleted, and delays the actual deletion in later compaction processes.

Small File Workloads – We also use *mdtest* benchmark to generate small file workloads to evaluate the effectiveness of embedding file content with the metadata inside TableFS. In this test, *mdtest* benchmark creates 5 million files in a single directory, and writes 4KB and 16 KB of data in these files. Our choice of file sizes is motivated by prior file system studies that have observed that 4KB is the median file size for many desktop workloads [20] and 16KB is the median file size for large PanFS storage clusters [41]. The threshold for embedding files, T, for our prototype is set to 64KB, so all these files are stored in TableFS instance.

Figure 7 shows the aggregated throughput of both PanFS-GT and native PanFS during the small file workload test. We observe that PanFS-GT is about $2.5\times$ faster than native PanFS for 4KB files. However, PanFS-GT is about 35% slower than PanFS for 16KB file size. We found that embedding 16KB file makes the key-value pair significantly larger. This causes higher write amplification during LevelDB compactions because LevelDB compactions will try to merge-sort both file data and metadata. Additionally, LevelDB writes the inserted key-value pairs twice: first to the transaction log and then (during a compaction) to the SSTable. With larger key-value pairs, LevelDB's per-operation efficiency is reduced due to the increased cost of extra copies of large values. This suggests that the threshold T for embedding file size should not be greater than 16KB. Embedding larger files may sacrifice read performance for faster insertion rate; this may be done using a column-oriented layout approach that stores metadata and small file separately. These tradeoffs are not studied in this paper and left for future work.

Data Intensive Workloads – To study the data path throughput, we use the LANL filesystem checkpoint benchmark to generate different types for HPC checkpoint I/O patterns [23]. Our experiments use this benchmarking tool to generate a concurrent N- N checkpoint write and read workload.

All checkpoint I/O is performed by a set of processes that synchronize with each other using MPI

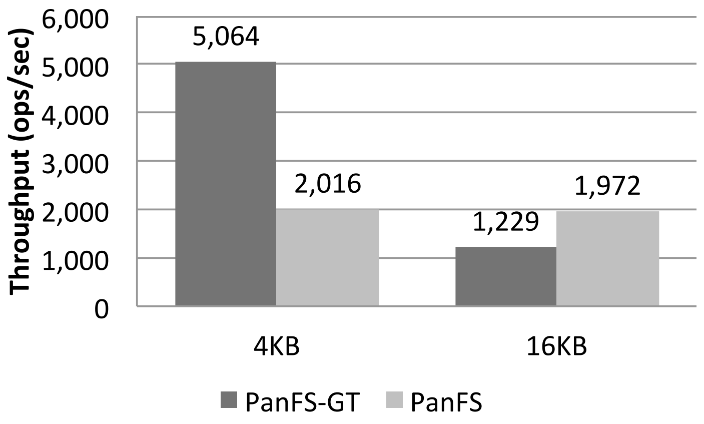


Figure 7: *Average aggregated throughput during creating 5 million small files with different size in one shared directory*

barriers. At the beginning, each process creates and opens a new checkpoint file for writing. Each process then waits at a barrier until all processes are ready to write. Once all processes are ready, each process starts to write the checkpoint data to its own file (and all processes do this writing concurrently). Once a process has written the specified number of bytes, it waits for all other processes to finish writing, and then syncs its data to the file system before closing the file. Before starting the read phase, we terminate all processes accessing these checkpoint files so that we can unmount the filesystem in order to ensure that all freshly written data has been flushed out from all the nodes' memory to prevent any performance side-effects from caching. After the filesystem has been mounted again, the benchmark reads the checkpoint in the same way it was written, however we shift, so each process will read the file generated by another process.

In this test, we also vary the number of clients per test node from 8 to 64 clients. The clients in each node will generate a total of 640GB checkpoint data to the underlying file system. The size of data buffer for each file system call is set to be 16KB. For PanFS-GT, the checkpoint files generated in the test will be first stored in TableFS, and then migrated to the underlying PanFS.

Figure 8 and 9 show the average throughput during the write phase and read phase in the N-N checkpoint workload respectively. For the checkpoint writing case, in Figure 8, we observe that PanFS-GT write throughput is comparable with the native PanFS. In fact, for smaller configurations with 8 clients per server, PanFS-GT outperforms native PanFS by about 20%. For the checkpoint reading workload, in Figure 9, PanFS-GT reads about 10% slower than native PanFS.

3.3 Understanding Scaling Behavior

This section shows how GIGA+TableFS scales over time in large multi-server setup. To understand this behavior we layered GIGA+TableFS on a different cluster (cluster #2 from Table 1) comprising of 64-nodes running GIGA+TableFS clients and servers. The main difference is the backend storage: this analysis emulates a federated NFS setup where each PanFS-GT server process in node manages its own TableFS instance and store SSTables into a local disk. To emulate shared storage for split operations, we used a NFS-mounted shared directory accessible from all machines; this shared directory was only used for moving SSTables of splitting directory partitions across servers.

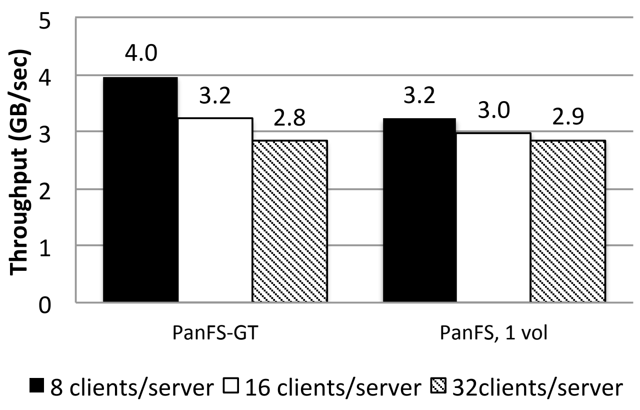


Figure 8: *The aggregated write throughput in N-N check-pointing workload. Each volume receives 640 GB data.*

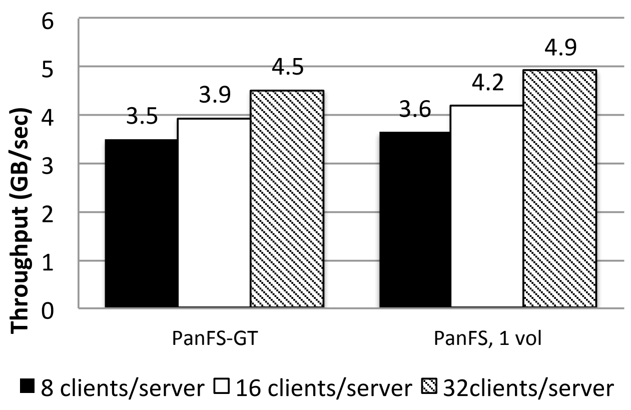


Figure 9: *The aggregated read throughput in N-N check-pointing workload.*

This analysis uses a *weak scaling* experiment that creates 1 million files per server, for a total of 64 million files in the 64-server configuration. We vary the number of servers from 1 to 64 to understand how performance scales with more servers.

Figure 10 shows the instantaneous throughput during the concurrent create workload. 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 TableFS 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 TableFS, it requires more compaction work to maintain invariants inside LevelDB 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 [4]. Thus we can use the formula $\frac{a \cdot S+b}{\log T}$ to

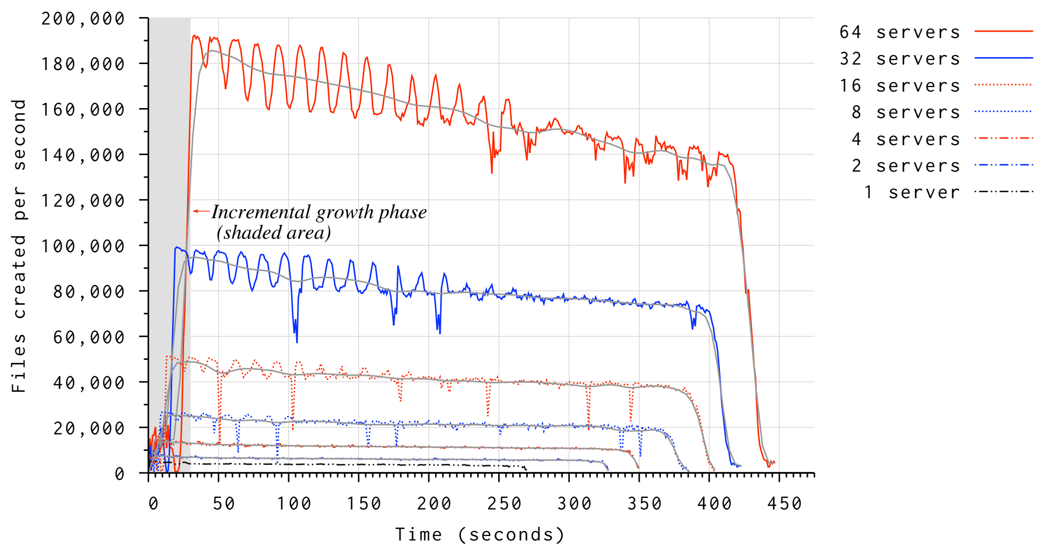


Figure 10: *Our middleware metadata service prototype shows promising scalability up to 64 servers. Note that at the end of the experiment, the throughput drops to zero because clients stop creating files as they finish 1 million files per client. (Solid lines are Bezier curves to smooth the variability.)*

approximate the throughput timeline in Figure 10, 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. This still exceeds today supercomputer's most rigorous scalability demands for 40,000 file creates per second in a single directory [22].

4 Related Work

This paper proposes a layered approach that allows a cluster file system to distribute both namespace and directories. A prior proposal of this work was presented in a workshop [26]. In this section, we discuss prior work related to metadata systems in modern cluster file systems and optimized layouts for high-performance metadata.

In many cluster file systems, the metadata path lacks the concurrency and scalability found in the data path. Many file systems, including Lustre [18], Google file system [11] and HDFS [12], continue to rely on a single metadata server to store all file system metadata. This simplifies the complexity of administration and implementation, but limits the scalability to how fast one server can service all metadata operations. Some cluster file systems distribute metadata on multiple metadata servers. The main difference is what metadata is distributed and how it is distributed.

PanFS and PVFS cluster distribute different parts of the namespace on different metadata servers. PanFS uses a coarse-grained distribution by assigning a subtree of the namespace (called volume) to each metadata server (called directory blade) [40]. PVFS is more fine-grained: it spreads different directories, even the ones in the same sub-tree, on different metadata servers [31]. However, both PanFS and PVFS lack support for distributed large directories.

Few cluster file systems have added support for distributed directories but each system uses a different technique to spread these directories. A beta release of OrangeFS, a commercially supported PVFS distribution, uses a simplified version of GIGA+ to distribute large directories on several metadata servers [21]. Ceph uses an adaptive partitioning technique called CRUSH for distributing its metadata and directories on multiple metadata servers [39]. IBM GPFS uses extensible hashing to distribute directories on different disks on a shared disk subsystem [32]. GPFS has the most complete and stable distributed directory implementation, but its design relies heavily on cache consistency and distributed locking mechanisms. When compared to GIGA+, this may result is much lower file create performance (when files are created in a single directory) but GPFS delivers very high read-only directory read performance by caching directory blocks on all readers [27].

Another novel aspect of this paper is the use of optimized single-node metadata representation to faster metadata performance. Several recent efforts have focused on using advanced data-structures, such as modified LSM-trees [33], stratified B-trees [37], fractal trees [9], and VT-trees [34]. These approaches speed up the number of IOs per second for hard drives or solid state devices by improving compaction process or by using new functionality such as data versioning. Although TableFS may benefit from all of these improvements, GIGA+ integration in some cases may result in a complex implementation and deployment. Understanding the interaction between these single-node TableFS optimizations and cross-server operations will be explored in future work.

5 Conclusion

Many HPC cluster file systems lack a general-purpose scalable metadata service that distributes both namespace and directories. This paper presents an approach that allows *existing* file systems to deliver scalable and parallel metadata performance. The key idea is to re-use a cluster file system's scalable data path to provide concurrent access on the metadata path. Our experimental prototype has demonstrated a *10X* improvement in the metadata performance of a production cluster file system, Panasas's PanFS.

This paper makes three contributions. The first contribution is an efficient combination of scale-out indexing technique (GIGA+) with a scale-up metadata representation (TableFS) to enhance the scalability and performance of metadata operations. The second contribution is the ease of deployment: our system can be layered on a cluster file system without any changes to the file system. This layering also facilitates easy federation of different file systems through a middleware library, an approach that is popular with other HPC systems such as MPI-IO and PLFS. The third contribution is the portable design that can use any existing file system deployments without any configuration changes (but with different fault tolerance assumptions) to the file system or the systems software on compute nodes.

References

[1] FUSE. http://fuse.sourceforge.net/.

[2] mdtest: HPC benchmark for metadata performance. http://sourceforge.net/projects/mdtest/.

[3] Fay Chang and. BigTable: a distributed storage system for structured data. In *OSDI*, 2006.

[4] Michael Bender and et al. Cache-oblivious streaming B-trees. In *SPAA*, 2007.

[5] John Bent and et al. PLFS: a checkpoint filesystem for parallel applications. In *SC*, 2009.

[6] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

[7] Peter Corbett and et al. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[8] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical report, Carnegie Mellon University, 2008.

[9] John Esmet and et al. The TokuFS streaming file system. *HotStorage*, 2012.

[10] Andrew Fikes. Storage Architecture and Challenges (Jun 2010). Talk at the Google Faculty Summit 2010.

[11] Sanjay Ghemawat, Howard Gobioff, and Shuan-Tek Lueng. Google file system. In *ACM SOSP*, 2003.

[12] HDFS. Hadoop file system. http://hadoop.apache.org/.

[13] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing Local File Accesses for FUSE-Based Distributed Storage. *High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012.

[14] Stephanie Jones and et al. Easing the burdens of HPC file management. 2011.

[15] 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.

[16] 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.

[17] LevelDB. A fast and lightweight key/value database library. http://code.google.com/p/leveldb/.

[18] Lustre. Lustre file system. http://www.lustre.org/.

[19] Avantika Mathur and et al. The new EXT4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[20] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *FAST*, 2011.

[21] Micheal Moore and et al. OrangeFS: Advancing PVFS. *FAST Poster Session*, 2011.

[22] Henry Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf, 2008.

[23] James Nunez and John Bent. LANL MPI-IO Test. http://institutes.lanl.gov/data/software/, 2008.

[24] Patrick O'Neil and et al. The log-structured merge-tree. *Acta Informatica*, 1996.

[25] Swapnil Patil and et al. Giga+: scalable directories for shared file systems. In *PDSW*, 2007.

[26] Swapnil Patil and et al. A case for scaling hpc metadata performance through de-specialization. 2012.

[27] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST*, 2011.

[28] Kai Ren and Garth Gibson. TableFS: Enhancing metadata efficiency in the local file system. *CMU Parallel Data Laboratory Technical Report CMU-PDL-13-102*, 2013.

[29] Ohad Rodeh and et al. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.

[30] 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.

[31] Robert B. Ross and et al. PVFS: a parallel file system. In *SC*, 2006.

[32] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST*, 2008.

[33] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *SIGMOD*, 2012.

[34] Pradeep Shetty and et al. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on File and storage technologies (FAST)*, 2013.

[35] Galen Shipman and et al. Lessons learned in deploying the world's largest scale lustre file system. In *The 52nd Cray User Group Conference*, 2010.

[36] Adam Sweeney and et al. Scalability in the XFS file system. In *USENIX ATC*, 1996.

[37] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioning dictionaries. *HotStorage*, 2011.

[38] Feiyi Wang and et al. Understanding Lustre Filesystem Internals. http://users.nccs.gov/~fwang2/papers/lustre_report.pdf, 2009.

[39] Sage A. Weil and et al. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*, 2006.

[40] Brent Welch and et al. Scalable performance of the panasas parallel file system. In *FAST*, 2008.

[41] Brent Welch and Geoffrey Noer. Optimizing a hybrid ssd/hdd hpc storage system based on file size distributions. *29th IEEE Conference on Massive Data Storage*, 2013.