DeltaFS: A Scalable No-Ground-Truth Filesystem For Massively-Parallel Computing

Qing Zheng
Carnegie Mellon University
Pittsburgh, PA, USA
zhengq@cs.cmu.edu

Charles D. Cranor
Carnegie Mellon University
Pittsburgh, PA, USA
chuck@ece.cmu.edu

Gregory R. Ganger
Carnegie Mellon University
Pittsburgh, PA, USA
ganger@ece.cmu.edu

Garth A. Gibson
Carnegie Mellon University
Pittsburgh, PA, USA
garth@cs.cmu.edu

George Amvrosiadis
Carnegie Mellon University
Pittsburgh, PA, USA
gamvrosi@cmu.edu

Bradley W. Settlemyer
Los Alamos National Lab
Los Alamos, NM, USA
bws@lanl.gov

Gary A. Grider
Los Alamos National Lab
Los Alamos, NM, USA
ggrider@lanl.gov

ABSTRACT

High-Performance Computing (HPC) is known for its use of massive concurrency. But it can be challenging for a parallel filesystem's control plane to utilize cores when every client process must globally synchronize and serialize its metadata mutations with those of other clients. We present DeltaFS, a new paradigm for distributed filesystem metadata.

DeltaFS allows jobs to self-commit their namespace changes to logs, avoiding the cost of global synchronization. Followup jobs selectively merge logs produced by previous jobs *as needed*, a principle we term *No Ground Truth* which allows for efficient data sharing. By avoiding unnecessary synchronization of metadata operations, DeltaFS improves metadata operation throughput up to 98× leveraging parallelism on the nodes where job processes run. This speedup grows as job size increases. DeltaFS enables efficient inter-job communication, reducing overall workflow runtime by significantly improving client metadata operation latency up to 49× and resource usage up to 52×.

CCS CONCEPTS

• **Information systems** → **Distributed storage**; **Directory structures**; • **Computing methodologies** → *Massively parallel algorithms*.

ACM Reference Format:
Qing Zheng, Charles D. Cranor, Gregory R. Ganger, Garth A. Gibson, George Amvrosiadis, Bradley W. Settlemyer, and Gary A. Grider. 2021. DeltaFS: A Scalable No-Ground-Truth Filesystem For Massively-Parallel Computing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476148

1 INTRODUCTION

It is easy to slow down a C program — just add the "`_Atomic`" qualifier to all the program's variables and rerun it. Atomic variable accesses are globally synchronized to ensure that the latest data written to memory is always used. Making variables atomic is unlikely to change a program's behavior in terms of correctness, but on modern processors this will significantly reduce the effectiveness of caching and slow down the program [6, 43]. Consequently, C variables are not atomic by default [3]. Applications explicitly request it when needed.

Unfortunately, even though modern HPC applications can request memory atomicity on an as-needed basis, their persistent state — stored as files and accessed through a shared underlying parallel filesystem [65, 66, 79] — remains globally synchronized at all times. This is true even on the world's largest HPC computers: every process is guaranteed to see every other process's latest filesystem namespace mutations all the time regardless of whether these processes communicate and despite potentially huge performance penalties. Alas, today's parallel filesystems continue to feature almost the same semantics as their ancestors developed for single-core machines 50 years ago. Yet, they are now faced with having to scale as rapidly as today's parallel computing systems [75]. This is too difficult, and results in performance bottlenecks that increasingly negate the benefits of massive parallelism.

We propose DeltaFS, a new way of providing distributed filesystem metadata on modern parallel computing platforms. DeltaFS reimagines the roles filesystems play in delivering performance and consistency to applications. First, today's filesystem clients tend to synchronize too frequently with their servers for metadata reads and writes. DeltaFS provides deep relaxation of filesystem namespace synchronization and serialization through client logging and subsequent merging of filesystem namespace changes on an as-needed basis. Second, today's filesystems map all application jobs to a single filesystem namespace. DeltaFS enables jobs to self-manage their synchronization scopes to avoid false sharing and to minimize per-job filesystem namespace footprint to improve performance. Finally, modern filesystems achieve scaling primarily by dynamic namespace partitioning over multiple dedicated metadata servers [60, 77, 79, 82]. Filesystem metadata performance is a function of, and is fundamentally limited by, the amount of compute resources that are dedicated to these metadata servers. DeltaFS dynamically instantiates filesystem metadata processing functions on client nodes, enabling highly agile scaling of filesystem metadata performance beyond a fixed set of dedicated servers.

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476148

At the core of DeltaFS is a transformation of today's globally synchronized filesystem metadata, to per-job metadata log records that can be dynamically merged to form new filesystem namespace views when requested by a followup job. To achieve this, DeltaFS defines an efficient log-structured filesystem metadata format that an application job process can use to log its namespace changes as a result of its execution. DeltaFS does not require all client changes to be merged back to a server-managed global tree for a consistent view of the filesystem. Instead, a job selectively merges logs produced by previous jobs for sequential data sharing. Unrelated application jobs never have to communicate.

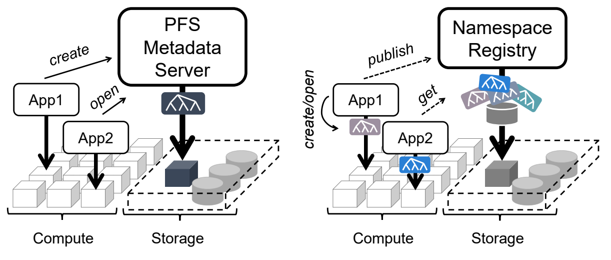
With DeltaFS we envision a drastic reduction of today's parallel filesystems to services that applications independently instantiate on compute nodes. A scalable object store provides shared underlying storage for per-job namespace management. There is no longer a global namespace. Instead, applications communicate only when they need to and communication is done primarily through sharing and publishing immutable log records stored in the shared underlying object store for minimum synchronization. Meanwhile, there no longer needs to be any dedicated metadata servers. With DeltaFS, jobs dynamically utilize their compute nodes for metadata processing, overcoming limitations and bottlenecks seen in today's parallel filesystem metadata designs. We call this new way of managing distributed filesystem metadata *No Ground Truth*, as it requires no global synchronization. Unrelated jobs are no longer required to see each other's files or be delayed by each other's namespace updates. Despite not having a global filesystem namespace, jobs using DeltaFS can still perform inter-job communication through the filesystem. In fact we show that by decoupling and parallelizing metadata accesses, DeltaFS vastly reduces a metadata-intensive workflow's inter-job communication latency compared with today's parallel filesystems.

Our experiments show that DeltaFS improves metadata performance by using the parallelism that can be found when utilizing resources at the nodes where job processes run. This parallelism is unlocked due to DeltaFS's no-ground-truth property that allows jobs to selectively merge logs produced by previous jobs as needed. We show up to 98× faster metadata operation throughput compared with the current state-of-the-art, a number that rises as job size increases. DeltaFS further enables efficient inter-job communication, vastly decreasing overall workflow runtime by significantly reducing the average latency of client metadata operations by up to 49× and the CPU time clients are blocked on such operations by up to 52×.

The rest of this paper is structured as follows. Section 2 describes the motivation and rationale behind our work. Sections 3 to 7 detail our design. We report experiment results in Section 8, related work in Section 9, and then conclude.

2 MOTIVATION

Three factors motivate our work: (1) the high cost of global synchronization for strong consistency in today's massively-parallel computing environments, (2) the inadequacy of the current state-of-the-art for scalable parallel metadata performance, and (3) the promise of a relaxed "no ground truth" parallel filesystem for non-interactive parallel computing workloads.



(a) Today's Parallel Filesystems **(b) Our Vision: DeltaFS**

Figure 1: Motivation for no ground truth. Rather than synchronously communicating with a slow global filesystem namespace, jobs communicate instead by publishing and sharing filesystem namespace snapshots on an as-needed basis through a public registry for high performance.

Global Synchronization. Filesystems are the main way applications interact with persistent data. While a local filesystem manages the files of a single node, distributed parallel filesystems such as Lustre [66], GPFS [65], PVFS [18], and the Panasas PanFS filesystem [79] manage the files of today's largest supercomputers [75]. By striping data across a large pool of object storage devices, parallel filesystems enable fast concurrent access to file data [20, 26]. However, in terms of metadata management, a strategy not too different from early network filesystems is used: all client metadata mutations are synchronously processed. They are first sent to a server, checked and serialized by it, and then appended to the server's write-ahead log for eventual merging into the filesystem's on-disk metadata representation managed by the server [31, 52, 64].

While not necessarily the best way to handle file metadata on a large supercomputer, an important reason modern parallel filesystems continue to use this strategy is that it enables distributed application processes to communicate as if they were on a local machine thanks to constant global synchronization [61]. Unfortunately, early network filesystems were not developed with today's massively-parallel computing environments in mind. While the early CM-5 computer at LANL — the fastest machine of its time — had only 1024 CPU cores, the fastest computer today has as many as 7 million cores [30, 51]. As the best way to utilize a modern supercomputer is to keep all of its compute cores busy, global synchronization in modern parallel filesystems is rendering itself a growing source of performance bottlenecks in today's leading computing systems, increasingly nullifying the very parallelism that these systems enable in the first place. This needs to change.

Inadequacy of the Current State-of-the-Art. To attain high metadata performance, modern scalable parallel metadata services use dynamic namespace partitioning over multiple metadata servers [60, 77, 79, 82]. In these filesystems, each metadata server manages a partition of the filesystem's namespace. The overall metadata performance is a function of the number and compute power of these servers. However, dynamic namespace partitioning does not remove global synchronization; it partitions it. Meanwhile, even these scalable filesystems can require a significant number of dedicated metadata servers to achieve high performance. Worse, as applications do not use the filesystem the same way, the amount of compute resources to be devoted to the filesystem can be difficult to determine beforehand. This leads to performance bottlenecks when the demand of the application is too high compared with the estimated amount and is a waste of resources otherwise.

In addition to dynamic namespace partitioning, another way to improve performance is to employ an efficient log-structured metadata format that hides the processing delay associated with today's parallel filesystem metadata servers [47, 60]. Such a system is able to quickly absorb a large amount of client changes without immediately optimizing them for fast reads — a separate set of server threads do so asynchronously in the background so that the client need not be blocked. However, in cases where the background process cannot keep up with the foreground insertion, the time required for these background operations will have to be amortized immediately. When the server compute resources are insufficient for the given workload, a client still experiences delays.

Previous work has also showed ultra fast file creation speed through deep client logging [12, 60, 85]. Yet client logging alone does not address read performance and does not support inter-job communication.

From One-Size-Fits-All to No Ground Truth. As we keep increasing the capacity and parallelism of our computers, an emerging reality we need to confront is that we are fast approaching a point in time when there will be no one-size-fits-all parallel metadata systems [7, 71]. While the high cost of global synchronization will continue to be necessary in cases where applications use the filesystem to communicate, it is also important to realize that today's parallel applications are for the most part non-interactive batch jobs that do not necessarily benefit from many of the semantic obligations that early network filesystems carried in their computing environments [31, 52, 64].

A modern HPC application is first submitted to a job scheduler queue [75]. When scheduled, it reads from the parallel filesystem, writes to the parallel filesystem, and then ends. Looking at the job, the input it reads is likely ready and static at the time the job is submitted. The output it generates is probably not examined, except possibly by the job owner trying to figure out how the job is progressing, until after the job is done [2, 12, 45]. This is effectively sequential data sharing. We argue that a parallel filesystem could serve this through simple publication and sharing of filesystem namespace snapshots without requiring any global synchronization, as we show in Figure 1. To achieve this, we envision running a public namespace registry to which all jobs can publish their namespaces as snapshots. When a job starts, it selects a subset of these snapshots as input and ends by publishing a new snapshot comprising all of the job's output. This new snapshot can then be used by an interested followup job to serve as its input, achieving efficient inter-job data propagation.

Namespace snapshots can be compact and easy to generate given a log-structured filesystem metadata format: each snapshot is simply pointers to a set of previously executed filesystem metadata changes [58, 60, 86]. In addition, with each job referencing a snapshot to start, there need not even be a global filesystem namespace. When a job needs to read data from multiple input snapshots, it simply merges all of these snapshots to form a single, materialized view of the filesystem for fast metadata read performance. Better yet, the job can start its own filesystem metadata server processes on its compute nodes to perform the merge and then to serve the reads. This allows the job to better utilize the massive parallelism in today's computing platforms to achieve scalable read performance which is not restricted by the resources set aside by cluster administrators for centralized metadata management. Additionally, unrelated jobs never have to communicate: they simply work on different snapshots. We have designed DeltaFS around this relaxed, scalable, log-structured, no global namespace, no ground truth, parallel filesystem metadata principle.

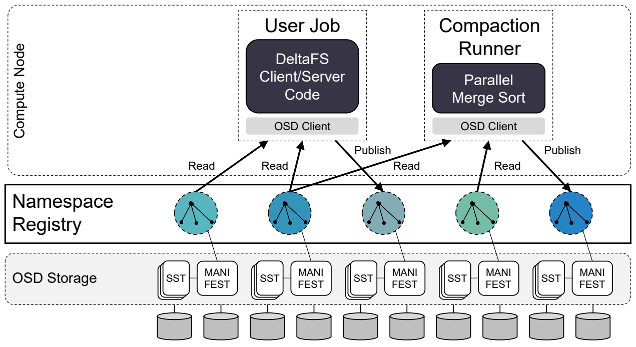


Figure 2: Architecture of DeltaFS. A DeltaFS cluster consists of per-job DeltaFS client/server instances and dynamically instantiated compaction runners on compute nodes reading, merging, and publishing filesystem namespace snapshots to a public registry that maps snapshot names to snapshot data stored in a shared underlying object store.

3 SYSTEM OVERVIEW

DeltaFS is a collection of library routines and daemon processes that provide scalable parallel filesystem metadata access on top of a shared underlying object store. As Figure 2 illustrates, the main components of DeltaFS include *User Jobs*, *Compaction Runners*, and *Namespace Registries*. The DeltaFS library code running inside each user job serves as DeltaFS metadata servers and clients. Compaction runners are user-scheduled DeltaFS log compaction code that reorganizes a given subset of on-storage DeltaFS metadata for fast future access. Namespace registries are long-running daemons that enable users to efficiently track and discover available DeltaFS namespace snapshots. All DeltaFS data and metadata are persisted in a shared underlying object store for long-term storage [13, 49, 78].

User Jobs. Jobs are parallel programs or scripts that are submitted to run on compute nodes [45]. Unlike today's parallel filesystems in which a single global filesystem namespace is provided to all jobs and dedicated metadata servers are deployed to manage it, in DeltaFS jobs act as managers of their own filesystem metadata. Each job starts by self-defining its filesystem namespace. This is done by looking up and potentially merging namespace snapshots published by previous jobs. The job then instantiates DeltaFS client and server instances to serve the namespace. At the end of the job, it may release its namespace as a public snapshot searchable and mergeable by other jobs.

A job instantiates DeltaFS client and server instances by running them inside the job's processes on compute nodes. A job may have many processes (e.g., a parallel simulation). Each of these processes can act as a client, and additionally as a server — the DeltaFS library code linked into them is capable of being both. Not all processes are required to be a server.

Client-server communication is done through RPCs [14, 69]. The addresses of the servers are sent to clients using a bootstrapping mechanism [28, 63] at the beginning of each job. When these addresses need to be known by code outside the job (e.g., the owning user's job monitoring tools), they may be published at an external coordination service [16, 34, 41] for public queries.

When a set of related jobs form a workflow and are scheduled to run consecutively (e.g., a simulation directly followed by post processing and data analytics), it is possible to use a single DeltaFS instance (the ensemble of all DeltaFS clients and servers instantiated by a job) to serve the entire workflow to improve performance — there is no need to repeatedly publish and search filesystem namespace snapshots within a workflow and repeatedly restart from an empty filesystem metadata cache. To achieve this, the workflow manager (or job script) spawns DeltaFS servers as standalone processes (not embedded in a job process) on compute nodes. These standalone servers can then outlive each individual step in the workflow and be reused by these steps for efficient metadata access. The workflow manager shuts down the servers when the last step of the workflow completes.

Within a job, application code interacts with DeltaFS by making DeltaFS library calls. DeltaFS handles all metadata operations such as creating new files, setting permissions, and removing existing directories. Data operations are redirected to the underlying object store for scalable processing [26, 60, 77]. A newly created but growing file may be transparently striped across multiple data objects for parallel data operations within a single file [18, 20]. When a file is opened for writing, some of its attributes such as file size and last access time may change relative to DeltaFS's per-open copy of the attributes. DeltaFS captures these changes on file close using its metadata path.

To attain high metadata performance, DeltaFS aggressively partitions a namespace to achieve fast reads, uses client logging to quickly absorb bursts of writes, and packs metadata into large log objects (SSTables) stored in the shared underlying object store for efficient storage accesses (further explained in later sections).

Namespace Registries. Registries are keepers of all published DeltaFS namespace snapshots. Each registry can be thought of as a simple Key-Value (KV) table mapping snapshot names (K) to pointers (V) to the snapshots' manifest objects stored in the shared underlying object store.

As Figure 3 shows, each DeltaFS namespace snapshot is made up of packed metadata mutation logs that are stored as SSTables [27] on storage. The manifest is a special metadata object that is inserted into a snapshot to serve as its root index. It contains the names of all member logs (SSTables) of the snapshot and the key range of each of these logs. The DeltaFS metadata read path code uses this

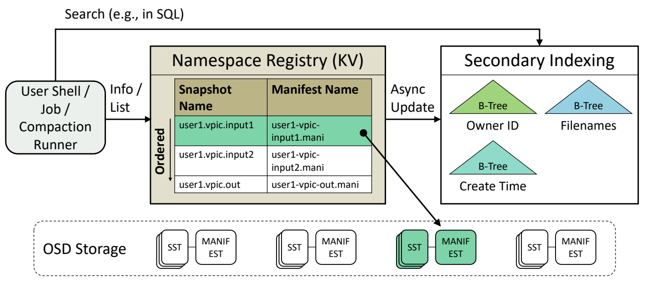


Figure 3: Locating snapshots in DeltaFS. A job, a compaction runner, or an interactive user uses namespace registries to locate snapshots according to their names. One or more secondary indexes may be built to allow for rich SQL queries.

information to locate snapshot data and to speed up queries against it. Section 5 explains this in more detail.

In DeltaFS, unrelated jobs never have to communicate. Related jobs, on the other hand, may communicate using DeltaFS to achieve efficient sequential data sharing. This is done first by a preceding job publishing its namespace as a snapshot and then by a followup job looking up the snapshot at a later point in time. To publish a namespace as a snapshot, a job flushes its in-memory state to storage, writes the manifest, and then sends the object name of the manifest and the name of the snapshot to the registry for publication. To read back a snapshot, a followup job sends the name of the snapshot to the registry in exchange for the name of the snapshot's manifest object. The job then reads the manifest object and uses it for queries into the snapshot.

In DeltaFS, namespace snapshots are named by jobs in the same manner as files are named by applications in today's filesystems — jobs present names and DeltaFS performs uniqueness checks. Similarly, just as today's applications must know the name of a file in order to operate on it, a DeltaFS job must know the name of a snapshot in order to look it up in a registry. To obtain filenames, today's applications can list files in a given parent filesystem directory. DeltaFS retains the same capability by allowing jobs to list snapshots according to a job-specified prefix string (snapshot names are indexed as ordered strings). To enable queries beyond simple snapshot listing, DeltaFS registries can be paired with a secondary indexing tier where snapshots are indexed by attributes other than their names (e.g., owner ID, create time, filenames within a snapshot) as Figure 3 shows. Modern database techniques could implement this and allow for rich SQL-like queries [10, 17, 40, 50]. We also imagine running an Internet-style search engine where users can search snapshots as if they were searching the web (e.g., "the latest App X's input deck"). We leave this as future work.

Registry daemons run on dedicated server nodes in a computing cluster. A cluster may be paired with one or more registries. In the latter case, each registry manages a partition of the snapshots' key space [19, 42]. Note that the dedicated servers hosting the registries do not sit on the critical path for regular filesystem metadata operations. As a result, their performance is less critical to the overall metadata performance of DeltaFS even for metadata intensive workloads. In practice, for write operations such as snapshot publication we expect registries to be no busier than today's job scheduler queues [8, 54]. To provide low-latency interactive read access for operations such as snapshot queries and users invoking DeltaFS commands on login nodes (e.g., `DeltaFS-snap-list` and `DeltaFS-snap-info`), DeltaFS registries can be scaled up using well-known techniques such as bigger memory, replication, and an increase in registry count.

Compaction Runners. Compaction runners are parallel log compaction jobs dynamically launched by users (i.e., not by DeltaFS) on compute nodes to merge and re-partition the metadata mutation logs (SSTables) generated by one or more previous jobs to form a compact, read-optimized view of a filesystem namespace for efficient queries by a followup job. The ability to explicitly schedule compaction over a large number of client compute cores on an as-needed basis is an important way DeltaFS differs from today's parallel filesystems. In those systems, global filesystem metadata is maintained by a system process on a dedicated server node for all jobs, leaving the server often unable to keep up with the clients under metadata-intensive workloads [7, 12]. To run parallel compaction, a user submits a special DeltaFS program (`DeltaFS-compaction-runner`) to the job scheduler queue and waits for it to be launched on compute nodes, as we discuss later in Section 7.

4 NO GROUND TRUTH

DeltaFS does not provide a global filesystem namespace to its users. Instead, it records the metadata mutations each job generates as immutable logs in a shared underlying object store. Subsequent jobs independently use these logs as "facts" for composing their own filesystem namespaces. DeltaFS does not impose a global ordering on logs. Nor does it require all logs to be merged. Enabling jobs to self-define their namespace consistency avoids unnecessary synchronization in a large computing cluster. A smaller filesystem metadata footprint per job further helps improve overall metadata performance.

A Log-Structured Filesystem. In DeltaFS, filesystem metadata information is persisted as logs. Metadata write operations such as `mkdir` and `chmod` apply changes by writing new log entries to storage. Metadata read operations (e.g., `lstat`) get file information by searching and reading related log entries from storage. The DeltaFS library code linked into each job process knows the log format. Logs written by one job can be understood by all jobs, making cross-job communication possible.

Rather than assuming a global filesystem namespace, a DeltaFS job starts by defining a base namespace that is private to the job. In the simplest case, a job starts with an empty base and ends with a log recording all the filesystem metadata mutations that the job has performed on the base. In cases in which a job needs to access the data output of a previous job, it uses the log produced by the previous job when instantiating its base. This allows the job to include all files and directories created by the previous job in its own private filesystem namespace view. As the job later executes, it records all of its metadata mutations in the form of new log entries and can decide to publish them at the end of the job. Published log entries can later be used by subsequent jobs for their namespace instantiation, allowing for efficient inter-job data propagation.

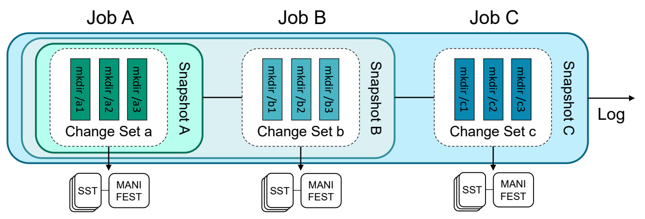


Figure 4: Job execution in DeltaFS. Each job generates a change set, extending a previous namespace snapshot and producing a new snapshot. For example, job B takes snapshot A as input, produces change set b, and generates snapshot B as output.

When the log entries generated by a job are published, these published log entries are called a *change set*. It represents the sum of all the filesystem metadata mutations that the job has performed (across all its processes) on its base. The state of the job's namespace at log publication — the state that a followup job will inherit — is called a *snapshot*. Thus each DeltaFS job can be viewed as a big log append operation: it appends a change set (a collection of filesystem metadata mutations) onto a snapshot and produces a new snapshot, as Figure 4 shows. At the same time, each DeltaFS snapshot can effectively be viewed as a rooted Directed Acyclic Graph (DAG) of change sets. The root of the DAG is the very change set that the job appends in producing the new snapshot. We call this special change set the *root* change set of the snapshot, with its manifest representing not only the root change set itself but also the entire snapshot (the entire change set DAG) that it encompasses.

Multi-Inheritance & Name Resolution. When a DeltaFS job instantiates its base, it may use multiple input snapshots. To achieve consistency within a job, a job specifies a priority ordering for all its input snapshots such that records from a higher priority snapshot take precedence. Figure 5 shows an example where jobs A, B, C, and D each take 0, 1, or more preceding jobs' snapshots as input, append a new change set onto that input, and generate a new snapshot as output. Job A takes null as input, generates change set a, and produces snapshot A. Job B takes job A's output snapshot as input, appends change set b onto it, and produces snapshot B. Job C takes job A's output snapshot as input, appends change set c onto it, and produces snapshot C. Job D takes both job B's and job C's output snapshots as input, appends change set d onto them, and produces snapshot D. While jobs B and C have both created a "`/p/y`" in their respective snapshots, D sees the `/p/y` created in B rather than that in C due to B having a higher priority than C in D.

Custom client filesystem namespace views are also available from systems such as UnionFS [81] and OverlayFS [1]. DeltaFS differs from them in that it allows complex client namespace views to be efficiently materialized for fast reads through a parallel compaction mechanism (§7) that can be dynamically invoked on compute nodes on an as-needed basis. At the same time, DeltaFS's log-structured metadata format (§5) enables efficient recording of client metadata mutations without being limited by copy-on-write and other overlay filesystem techniques. Finally, as a parallel filesystem, DeltaFS is able to spread workloads to distributed job processes to achieve

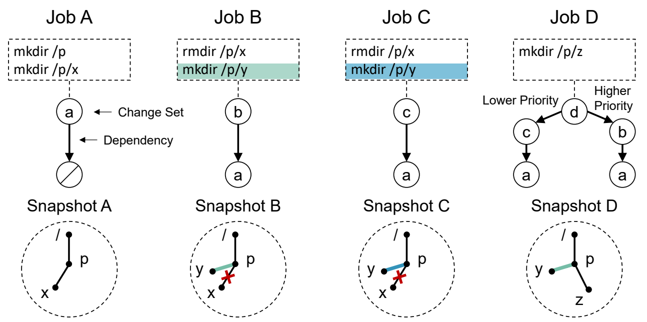


Figure 5: Priority-based name resolution in DeltaFS. Job D sees the /p/y created in job B rather than that in job C due to snapshot B having a higher priority than snapshot C.

scaling (§6) while a local overlay filesystem's performance is fundamentally limited by the capacity of the local machine.

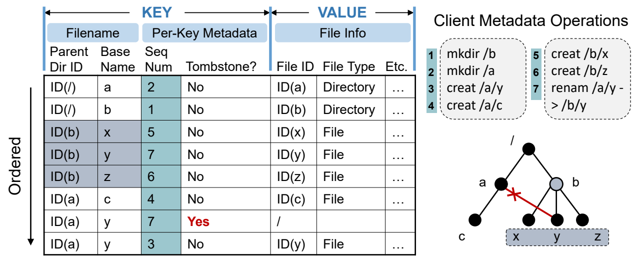
Complexity. While a collection of DeltaFS jobs may produce output snapshots that are complex DAGs, a user only needs to specify and order direct dependencies (i.e., direct input snapshots) when initializing a new job. Specification can be done through DeltaFS command-line arguments. Indirect dependencies are automatically resolved and ordered by DeltaFS at job start time, similar to Linux's dynamic loading of shared libraries. DeltaFS resolves dependencies by reading and tracing the manifest object associated with each published change set. We discuss change set and manifest format in more detail in Section 5.

The need to specify input and name output snapshots does not fundamentally change the way people run jobs — even with a global namespace, a user needs to know and specify the input and output paths of a job when launching it [45]. Also, global namespaces are rarely truly global: sites often divide storage into multiple filesystems (e.g., scratches, NFS homes) leading to many independent namespaces. While recent work has leveraged database techniques for a grand unified file index that spans multiple filesystems [44], DeltaFS, as we discussed in Section 3, can be paired with a similar file index in which all published job snapshots are indexed allowing users to more easily locate their files. We also envision building a high-level software stack on top of DeltaFS (such as `DeltaFS-Orchestra`) that automates per-job namespace instantiation as well as cross-job data propagation and compaction, making DeltaFS more accessible to its users.

5 PER-JOB LOG MANAGEMENT

DeltaFS jobs execute metadata operations by recording them as logs on storage. To attain high performance, DeltaFS uses a log format in which each filesystem metadata mutation is recorded as a KV pair in a table constructed with a Log-Structured Merge (LSM) Tree [53]. Tree data is persisted as named SSTables [27] indexed by a manifest object in a per-job change set. Background log compaction improves log storage for fast reads and handles garbage collection.

Log Format. DeltaFS logs a KV pair for each filesystem metadata mutation executed. The key stores the name of the file involved



KEY				VALUE		
Filename		Per-Key Metadata		File Info		
Parent Dir ID	Base Name	Seq Num	Tombstone?	File ID	File Type	Etc.
ID(/)	a	2	No	ID(a)	Directory	…
ID(/)	b	1	No	ID(b)	Directory	…
ID(b)	x	5	No	ID(x)	File	…
ID(b)	y	7	No	ID(y)	File	…
ID(b)	z	6	No	ID(z)	File	…
ID(a)	c	4	No	ID(c)	File	…
ID(a)	y	7	Yes	/		
ID(a)	y	3	No	ID(y)	File	…

Ordered
Client Metadata Operations
1 mkdir /b
2 mkdir /a
3 creat /a/y
4 creat /a/c
5 creat /b/x
6 creat /b/z
7 renam /a/y -> /b/y
/
a
b
c
x
y
z

Figure 6: DeltaFS's table-based metadata log format. Each executed filesystem metadata mutation has an associated row in the table. Keys are ordered, enabling fast reads and scans.

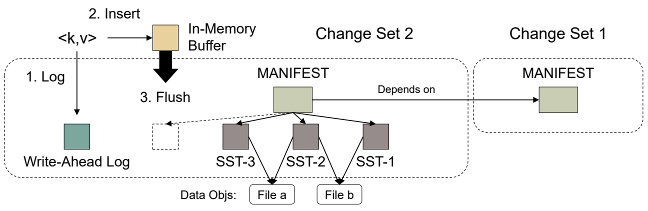


Figure 7: On-storage representation of an DeltaFS LSM-Tree in a per-job change set consisting of a manifest object, a write-ahead log, and a set of SSTables with references to separately stored data objects for large files.

in a mutation. The value stores the metadata information (i.e., the inode) of the file after the change. A special tombstone bit is recorded in each key to indicate whether a logged mutation is a delete. Additionally, each key is associated with a sequence number. Keys with higher sequence numbers supersede lower-numbered keys allowing newly logged changes to override older ones. All keys are inserted into a per-job table constructed with a high performance DeltaFS-modified LevelDB realization of an LSM-Tree [53].

As Figure 6 illustrates, we use parent directory IDs and the base names of files to represent filenames. Using parent directory IDs as key prefixes (instead of their full pathnames) allows DeltaFS to avoid having to update file keys when a user renames a directory along the files' parent paths [15, 82]. The metadata information we store for each file includes file ID, file type, file permissions for hierarchical access control, and file data for small files [60]. DeltaFS keys are ordered, allowing for efficient filesystem metadata lookups and directory scans [47, 58].

On-Storage Log Management. DeltaFS uses LSM-Trees [53] to manage the logged filesystem metadata mutations within each job change set. As Figure 7 shows, when initializing an LSM-Tree for a change set, a job first creates a manifest object in the underlying object store to record high-level information on the change set. This includes the name of the change set and the names of all its dependencies defined as the root change sets of all the job's input snapshots. Next, an in-memory buffer space is allocated in the job's process to buffer incoming metadata changes as the job runs. These changes are formatted as KV pairs. Whenever the in-memory write buffer is full, all KV pairs in the buffer will be sorted and written to storage as an SSTable [27]. The name of the SSTable, as well as

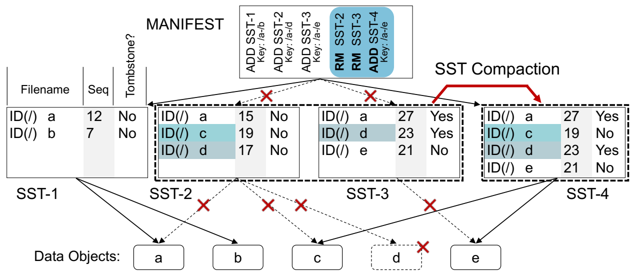


Figure 8: Example of an LSM-Tree compaction within a DeltaFS job change set. SSTable (SST) 2 and 3 are merged to form SSTable 4 reducing the number of SSTable lookups needed for key /c and garbage collecting data object d.

the key range of the table, is then recorded in the manifest for use by subsequent queries. To prevent data loss, a write-ahead log is created in the underlying object store for failure recovery of the job process's in-memory write buffer. A KV pair is first recorded at the write-ahead log before it is inserted into the in-memory buffer. The manifest, write-ahead log, and the SSTables listed in the manifest constitute the entire state of a change set.

Change sets can be deleted when they are no longer needed. A user deletes a change set by invoking a special DeltaFS program (`DeltaFS-changeset-delete`) using the name of the change set as argument. To prevent deleting a change set while others may still depend on it, DeltaFS has each change set hold a reference to itself and to each of its dependencies. When a user deletes a change set, its reference to itself is removed. All member objects of the set — the manifest, the write-ahead log, and all of its SSTables — will be deleted when no other change set has a reference to it. When a change set is deleted, all its references to others will be removed enabling these change sets to be deleted too. A utility program (`DeltaFS-changeset-clean`) is provided that a user can periodically run to delete change sets that are no longer referenced. When a change set is deleted, DeltaFS deregisters its corresponding snapshot from the registry. For large files, their data objects are referenced counted as well. A data object is deleted when all the SSTables and write-ahead logs referencing it are deleted.

Log Compaction. As a job runs, DeltaFS writes SSTables to persist changes whenever its in-memory write buffer is full. Entries in these SSTables may then be candidates for LRU replacement in the job's memory and get reloaded later by querying SSTables on storage until the first match occurs. SSTables are queried backwards from the most recent to the oldest. Over time, the cost of finding a record that is not in the cache increases as the number of SSTables increases. To improve read performance, DeltaFS runs compaction to merge sort overlapping SSTables and decrease the number of SSTables that might share a key. As SSTables are merged, new SSTables are generated and old ones are deleted. Data objects no longer referenced by any SSTable can then be deleted.

Figure 8 shows an example where SSTable 2 and 3 are compacted to form SSTable 4. Before the compaction, looking up key `/c` would require searching two SSTables. SSTable 3 is searched first as its key range [`/a-/e`] overlaps key `/c`. Since SSTable 3 does not have the key, SSTable 2 is searched next (the key will be found here). With compaction merging SSTable 3 and 2 into new SSTable 4, key `/c` can now be found with a single SSTable lookup, improving read performance. When tables are merged, records sharing a same filename prefix are merged such that only the one with the highest sequence number is copied into the new table. The rest are discarded. After tables are merged, information on the newly constructed table is logged in the manifest and references to the old tables are dropped. These tables are then deleted from the underlying storage along with their references to the data objects for large files, enabling them to be garbage collected too.

6 DYNAMIC SERVICE INSTANTIATION

A DeltaFS filesystem has no dedicated metadata servers. Instead, a job dynamically instantiates DeltaFS client and server instances in the job's processes to provide parallel filesystem metadata access private to that job. DeltaFS aggressively partitions a namespace across a job's servers to achieve scalable read performance, and it uses client logging to quickly absorb bursts of writes.

Per-Job Metadata Processing. Within a job, metadata operations are performed by clients sending RPCs to servers. A server is capable of executing both read and write operations. Write operations are executed by logging the resulting metadata mutation on storage using an LSM-Tree making up the job's change set. Read operations are executed first with queries into the job's own change set. If the required metadata is not found in the job's change set, then dependent change sets are queried using the user-defined priority ordering (§4). A server performs log compaction asynchronously in the background as the job executes (§5).

When a job instantiates multiple DeltaFS servers, each server manages a partition of the job's private filesystem namespace view. DeltaFS uses a namespace partitioning scheme derived from GIGA+ [55, 60] in which each newly created directory is randomly assigned to a server and gets gradually partitioned to more servers as it grows. Per-server metadata mutations are logged into their own separate LSM-Tree in the job's change set. Each LSM-Tree represents a partition and is indexed by a dedicated manifest object. The manifest object of the 0-th partition additionally serves as the manifest of the entire change set and is referenced by registries in their mapping tables (§3).

Client Logging. Synchronization between DeltaFS clients and servers within a parallel job ensures that files created by one process are immediately visible to all processes in that job. For workloads (e.g., N-N checkpointing) where files are opened for per-process writing [12, 13], DeltaFS allows a client to defer job-wide synchronization and to directly log metadata mutations in a per-client LSM-Tree for ultra high metadata write performance. A client can perform background log compaction against its private LSM-Tree to improve its read performance. However, when files created by a client are known to be write-only and are not opened for read until after the job completes, a DeltaFS client can elect to further defer its log compaction and later use a subsequent parallel log compaction program to merge and re-partition all of the job clients' LSM-Trees in a single large batch as described below in Section 7.

Namespace Curation. When a job logs its mutations in per-process logs for ultra high write performance, read requests unrelated to these mutations can still be served through the job's per-process DeltaFS servers according to the partitioning of the job's namespace. When the job's compute cores are insufficient for the workload at hand, it can allocate a separate larger set of compute nodes to run DeltaFS servers, utilizing the compute cores and memory on those nodes to scale reads and to achieve low-latency access to the job's metadata on storage. These separately allocated DeltaFS servers may be reused by multiple jobs within a workflow, a project, or even a campaign [45]. Campaign managers can request a persistent allocation of a set of compute nodes for running DeltaFS servers for an extended period of time. We call these read-only, job-specific, potentially long-running DeltaFS servers namespace curators. Critically, the amount of compute resources available to these curator processes is not decided by cluster administrators. Instead it is decided by the owners of the jobs, projects, or running campaigns for scalable filesystem metadata read performance.

Fault Tolerance, Aging, and Sequential Data Sharing. To resist compute node failures, DeltaFS performs write-ahead logging when performing filesystem metadata mutations. All DeltaFS write-ahead logs, file data, and filesystem metadata are stored in a shared underlying object store to enable failure recovery from a different set of compute nodes when a job fails. Write-ahead logs are flushed every 5 seconds (this period is configurable) and whenever `fsync` is called.

All filesystems age; DeltaFS is able to age more smoothly by facilitating explicit cross-job compaction (§7) and by not coupling all files into a single namespace for improved parallelism and cache performance. Having millions of snapshots in a registry is fine. Modern KV stores on recent hardware can routinely perform millions of operations per second (op/s). More importantly, with DeltaFS the total number of snapshots will be orders of magnitude smaller than the total number of files (even though a file may accumulate many metadata mutation log entries, snapshots are always per job). This drastically reduces the total number of keys that a central metadata server (or snapshot registries in DeltaFS) will have to handle compared with today's parallel filesystems.

In DeltaFS, if job X checks out a snapshot and rewrites a file while job Y later checks out the same snapshot, job Y will see the old data of the file. This is because that DeltaFS allows multiple versions of a file to exist in different snapshots and an object store that supports copy-on-write will enable this efficiently. If job Y actually wants to see job X's changes, job Y waits until job X publishes its changes and then checks out job X's snapshot — this is known as sequential data sharing which DeltaFS also supports.

7 CROSS-JOB PARALLEL LOG COMPACTION

All log-structured filesystems require compaction to achieve good read performance [58, 59, 62]. While today's parallel filesystem designs limit compaction activities to only dedicated metadata servers, DeltaFS allows a user to dynamically launch compaction on compute nodes. This enables utilizing a potentially large amount of compute cores to accelerate compaction operations. At the same time, with users only scheduling compaction on job change sets that are known to be read by a followup job, per-compaction data footprint can be minimized, further reducing compaction delays.

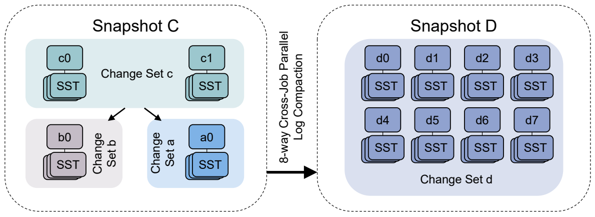


Figure 9: Example of a DeltaFS cross-job parallel log compaction. Snapshot C with a DAG of change sets a, b, and c is parallel compacted into snapshot D with only a single change set d consisting of 8 partitions.

Unlike per-job log compaction which, as we discussed in Section 6, is implicitly (i.e., without user intervention) done by servers embedded in the job's processes as a job runs, cross-job log compaction is explicitly scheduled by users. This is done by users launching a special parallel log compaction program that merges and re-partitions a set of related job change sets in a single batch. Typically, a user launches cross-job compaction either when a complex job change set hierarchy needs to be flattened for efficient queries (§4), a large set of per-client logged SSTables within a job change set needs to be parallel sorted for fast lookups (§6), or a given change set or a change set hierarchy contains too few partitions for sufficient load balance across the job processes of a followup job.

DeltaFS uses a scalable parallel merge sort pipeline for cross-job log compaction. Each pipeline process acts as a mapper for a subset of input SSTables and simultaneously as a reducer responsible for a partition of the target change set. Figure 9 shows an example where snapshot C — made up of a DAG of change set a, b, and c with c being the root — is parallel compacted to form snapshot D. Before parallel compaction, reading a key from snapshot C required searching potentially a partition of change set c, a partition of change set b, and a partition of change set a. After compaction, each key lookup requires searching only a single partition of change set d, significantly reducing query overhead. Note also that change sets a, b, and c were originally partitioned by the jobs that generated them. Jobs A, B, and C had only 1, 1, and 2 server partitions, resulting in their change sets to be partitioned accordingly. Compaction expands change set d to have 8 partitions. Followup jobs with 8 job processes can assign a partition to each of their per-process servers, fully load balancing their read operations.

8 EXPERIMENTS

We implemented a prototype of DeltaFS in C++. A modular design was used such that DeltaFS can be layered on top of different object storage backends such as Ceph RADOS [78], PVFS [18], HDFS [67], and other generic POSIX parallel filesystems [65, 66, 79].

Our experiments evaluate the performance of DeltaFS both in terms of a single application job (§8.1) and multiple jobs sharing a single computing cluster (§8.2). We test cases in which jobs are related and use the filesystem for sequential data sharing and cases in which jobs are unrelated and do not read each other's files. We compare DeltaFS with current state-of-the-art approaches: IndexFS

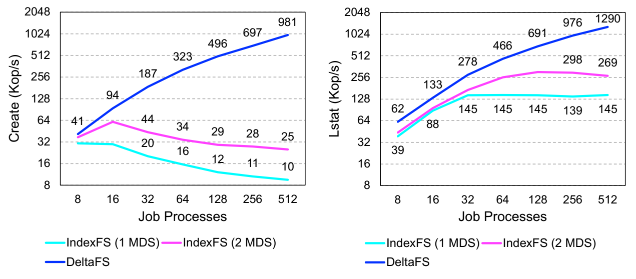
[60] for scalable parallel metadata performance and PLFS [12] for ultra fast client-based metadata logging. We also compare against a special IndexFS mode that allows clients to log metadata mutations for later bulk insertion. We use `mdtest` [4] to generate our test workloads. All our experiments store file metadata in a shared underlying object store implemented with Ceph RADOS on top of 8 dedicated Ceph OSD nodes along with 1 Ceph Manager and 1 Ceph Monitoring node. Each Ceph OSD features one 1GbE connection for foreground communication between Ceph clients and OSD servers and another 1GbE connection for background communication among Ceph OSDs, Managers, and Monitors.

8.1 Single-Job Performance

Current parallel filesystems use dedicated metadata servers. Their performance is limited by the compute resources that a cluster admin assigns to the filesystem. With DeltaFS we show that parallel filesystems scale better when not constrained to dedicated metadata servers. DeltaFS enables jobs to self-instantiate their metadata services on compute nodes, decoupling them from decisions made by cluster administrators and enabling scaling beyond a fixed set of server machines. To demonstrate this, our first experiment compares DeltaFS with IndexFS [60], a state-of-the-art approach for scalable parallel filesystem metadata performance using dynamic namespace partitioning.

Dynamic Namespace Partitioning. IndexFS is a scalable parallel filesystem whose metadata is partitioned for load balancing across multiple dedicated metadata servers [60]. Both IndexFS and DeltaFS implement the same namespace partitioning strategy [55], except that IndexFS partitions the namespace across dedicated metadata server nodes while DeltaFS normally partitions the namespace across the compute nodes of a running job. To compare DeltaFS with IndexFS, we created an IndexFS-like configuration of DeltaFS that uses dedicated servers for metadata operations instead of job compute nodes. We ran our IndexFS configuration using one or two server nodes. In the latter case, the filesystem's namespace is divided into two partitions and there is one dedicated DeltaFS server on each node to manage a partition.

We used Parallel Data Lab (PDL)'s Susitna cluster at Carnegie Mellon University (CMU) for these tests. Each Susitna compute node has four 16-core AMD Opteron 6272 2.1GHz CPUs, 128 GB memory, one 40GbE NIC, and one 1GbE NIC. A total of ten nodes are allocated: eight as client nodes (512 CPU cores) and two as dedicated metadata servers (for IndexFS). We use the 40GbE network for filesystem operations and the 1GbE network for accessing the shared underlying RADOS storage. Each test consists of running a parallel `mdtest` job that inserts empty files into a pool of parent directories and then queries the files it just created using the `stat` command. All runs start with an empty filesystem namespace, and files are created and `stat`'ed in random order in the namespace. Each job process creates and `stats` 200K files. Our smallest run used eight job processes and created 1.6M files. Our largest run consisted of 512 job processes and created 102.4M files. For IndexFS runs, all filesystem metadata operations are processed by one or two dedicated metadata servers. For DeltaFS runs, filesystem metadata operations are handled by the DeltaFS servers embedded in the job's processes on the client nodes. Each job process launches one server instance and manages a partition of the job's namespace.



(a) Metadata Writes **(b) Metadata Reads**

Figure 10: Results of parallel mdtest runs against IndexFS and DeltaFS. DeltaFS is up to 98.1x faster in filesystem metadata writes and 8.9x faster in reads due to decoupling and parallelizing filesystem metadata workload across all available job compute cores.

Figure 10a shows the file insertion performance. IndexFS performance is limited by the number of dedicated metadata servers used. The performance reduces as job size grows due to increased log compaction overhead at the server(s) as more files are inserted into the filesystem. Adding additional dedicated metadata servers to IndexFS would alleviate this bottleneck, but a large number of servers might have to be dedicated permanently for this task. DeltaFS decouples per-job metadata performance from dedicated resources. **By distributing work across all available compute nodes within a job, DeltaFS shows scalable performance that increases as job size grows.** At 512 job processes, DeltaFS is up to 98.1× faster than IndexFS, thanks both to having more CPU cores for handling client RPC requests and to less log compaction overhead due to aggressive namespace partitioning.

Figure 10b shows file `stat` command performance. Similar to file creates, DeltaFS performance increases as job size grows without being limited by dedicated resources. At 512 job processes, DeltaFS is up to 8.9× faster compared with IndexFS runs.

Client Logging. Recent work has shown ultra fast filesystem metadata insertion rates through client side logging for metadata-intensive workloads such as N-N checkpointing in which newly created files are not immediately opened for read [12, 60]. While client logging allows for fast writes, it does not address the performance of the read operations that may occur after data is written. Our second experiment shows that by combining client logging with post-write client log compaction (§7), DeltaFS is able to achieve fast reads in addition to fast writes. Compared with the current state-of-the-art, DeltaFS more completely addresses the metadata bottlenecks produced by extreme workloads.

We compare DeltaFS against IndexFS and PLFS. PLFS was developed for concurrently writing a single file (e.g., N-1 checkpoints) [12]. It defers global synchronization of writes by logging the writes of each process instead of processing them immediately. In addition to file data mutations, PLFS-like techniques have also been used to record filesystem namespace mutations [11]. The IndexFS scalable parallel filesystem includes an extension that allows a set of client

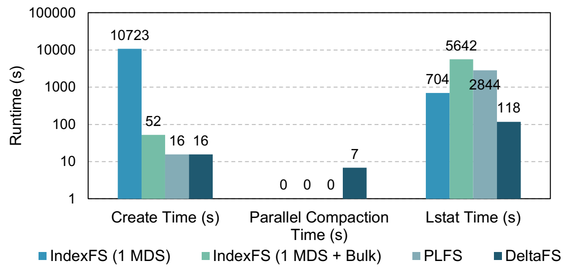


Figure 11: Comparison of client logging and subsequent read performance among IndexFS, PLFS, and DeltaFS. DeltaFS techniques achieve both fast writes and fast reads.

processes to write-lock a newly created directory and instead of synchronously integrating every filesystem metadata mutation beneath this directory, they each simply log operations to be applied later in a bulk insertion [60]. We implemented both PLFS and the IndexFS's bulk insertion extension in our DeltaFS code, taking advantage of DeltaFS's log-structured metadata representation and the presence of a shared underlying object store.

We run the same test as we did in our first experiment but this time with client-side logging enabled. We focus on the configuration in which 512 job processes are launched creating 102.4M files. For IndexFS bulk insertion runs, all clients log their file creates in per-client SSTables and inform the server of their SSTables at the end of the write phase. Subsequent read operations are processed by the server, as in the original IndexFS runs. For PLFS runs, per-client SSTables logged at the write phase are directly opened by clients at the read phase. All clients open all other clients' SSTables for random reads. Finally, for DeltaFS runs, client logged SSTables at the write phase are merged and re-partitioned through a 512-way parallel log compaction process invoked by DeltaFS in the job's processes before moving to the read phase, which enables fast subsequent reads.

Figure 11 shows the time it takes for each run to finish the write, the parallel log compaction time (DeltaFS only), and the read phase time. For reference, we included in the figure the original IndexFS (1 MDS) results from our previous experiment. By not synchronously integrating every file create operation to a dedicated server, client logging significantly improves a job's write performance for all of IndexFS, PLFS, and DeltaFS. Bulk-inserting IndexFS takes a little longer to finish due to the additional overhead of having to wait at the end of the write phase for the metadata server to update its LSM metadata with the newly inserted SSTables.

On the read side, even with bulk insertion, IndexFS performance is limited due to having only a single dedicated metadata server for reads. Worse, server-based background log (SSTable) compaction — which asynchronously optimizes SSTables to a read-optimized representation — has been deferred to the read phase after clients bulk insert their tables, resulting in a much slower read phase overall. PLFS spreads reads across all of the job's processes. However, each read is likely to have to query an excessive number of SSTables due to their overlapping key ranges in the absence of log compaction operations. **DeltaFS leverages parallelism available in the job's processes to complete log compaction faster and speed up subsequent reads.** Specifically, DeltaFS queries files 5.9×, 47.8×, 24.1× faster than IndexFS, IndexFS bulk insertion, and PLFS (Figure 11). It took DeltaFS longer to finish the reads (118s) than it did in Figure 10b (80s) due to having to start from a cold metadata cache following client logging and parallel log compaction. Nevertheless, through decoupling and parallelizing all of the write, read, and log compaction operations, DeltaFS managed a much shorter overall run time (write + read + compaction) compared with IndexFS, IndexFS bulk insertion, and PLFS.

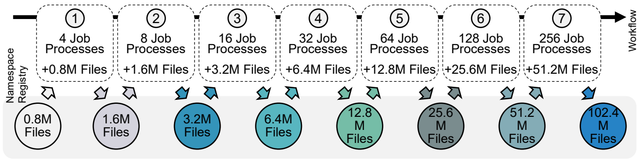


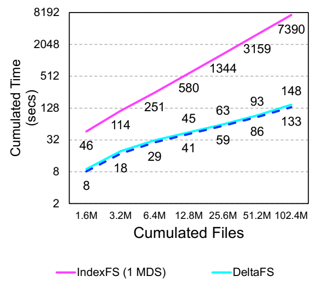
Figure 12: Illustration of our test workflow. Increasing file counts require increasing parallelism to minimize delays.

8.2 Multi-Job Performance

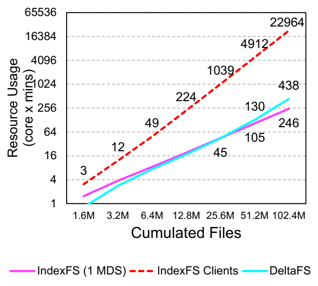
Enabling jobs to self-fund their metadata read, write, and compaction operations allows DeltaFS to greatly improve per-job metadata performance. In this section, we show that the same DeltaFS techniques can be applied to multi-job scenarios as effectively as they are for single jobs. This is true even in cases where related jobs use the filesystem for sequential data sharing, and it works in the absence of a global filesystem namespace.

No Ground Truth. Freedom from global serialization comes at the cost of the additional need for jobs to explicitly merge and compact related namespace snapshots before they can access them efficiently. It is difficult to identify typical multi-job workflow patterns [5], so to measure cost we devised a synthetic 7-stage workflow shown in Figure 12. Each stage (or job) in the workflow takes a previous filesystem namespace snapshot as input, doubles the number of files in the namespace by inserting new files, and then ends by publishing it as a new snapshot. New files are created with unique names that do not conflict with previously created files. The workflow starts with an initial snapshot containing 0.8M files. It ends with seven new snapshots, with the last one consisting of 102.4M files. We compare running this workflow using IndexFS (1 MDS) with running it using DeltaFS. For the IndexFS runs, all new files are directly inserted into the global namespace managed by the dedicated IndexFS metadata server. For DeltaFS, newly created files are first logged through the DeltaFS instance running at each client process. These logged file creates are later merged and compacted into a unified job-wide namespace when the output snapshot of the job is generated (more details below).

We used PDL's Narwhal cluster at CMU for these tests. Each Narwhal compute node has 4 Dual-Core AMD Opteron 2210 1.8GHz CPUs, 16 GB memory, and two 1GbE NICs. We use one NIC for filesystem operations and the other for accessing the shared underlying RADOS. Up to 128 nodes were allocated to run workflow jobs. We used `mdtest` to create files. Each workflow stage runs an increasing amount of `mdtest` job processes. The first workflow



(a) Runtime



(b) Resource Usage

Figure 13: Results of running a seven-stage workflow against IndexFS and DeltaFS. DeltaFS uses 52.9x fewer total CPU core-minutes than IndexFS while improving overall workflow runtime by 49.9x.

stage consists of 4 `mdtest` job processes inserting 0.8M files. The last stage consists of 256 processes inserting 51.2M files. For IndexFS runs, all job processes synchronize with a dedicated IndexFS metadata server to create files. For DeltaFS runs, file creates are first logged as per-client SSTables. A parallel log compaction program follows each `mdtest` job. It merges both the per-client SSTables generated by the job and the SSTables belonging to the original input snapshot to form a combined, read-optimized namespace view which is then published as a new snapshot. This new snapshot is logically equivalent to the IndexFS's global filesystem namespace at that moment, as both IndexFS and the DeltaFS snapshot contain all files that have been created so far and both are read-optimized (IndexFS server runs log compaction in the background). The extra processing time and compute resources that DeltaFS uses at the end of each workflow stage to run parallel log compaction to generate this namespace snapshot captures the cost of no ground truth in DeltaFS.

Figure 13a shows the cumulative time it takes for each filesystem to finish the workflow stages. There are two main factors that account for the difference in performance between IndexFS and DeltaFS. First, IndexFS is not workflow aware and must maintain a globally synchronized namespace on its metadata server at all times. The IndexFS background log compaction thread running on the dedicated metadata server may start running at any point in the workflow and interfere with performance. DeltaFS, on the other hand, is instantiated by the user and is aware of the phases of the workflow. DeltaFS can safely log client metadata operations when the application is in the file create phase and defer expensive compaction operations until the end of the phase when the output snapshot is made. Second, all IndexFS metadata processing takes place on a single dedicated metadata server that may bottleneck performance. With DeltaFS the metadata processing is distributed across all available client compute nodes enabling DeltaFS to take advantage of parallelism on these nodes to minimize compaction time when making a snapshot. These two factors result in DeltaFS being 49.9× faster than IndexFS on our workflow.

Figure 13a also shows how DeltaFS behaves when jobs in the workflow are configured to be unrelated. In this case each stage in the workflow starts with an empty namespace rather than seeing files created by the previous stage. The dashed line in Figure 13a shows DeltaFS performance in this case. Since DeltaFS does not have to merge logs from previous snapshots, it finishes faster. The difference between the two DeltaFS runs demonstrates the cost of sequential data sharing in DeltaFS. Note that the difference is small due to the efficiency of parallel compaction (which increases as job size increases).

Figure 13b shows the cumulative resource usage (in the form of CPU cores × mins) each workflow takes to process all of the filesystem metadata operations. For IndexFS, this is the usage of its dedicated metadata server. For DeltaFS, we aggregate its resource usage across all of its job processes. DeltaFS used about 2× more compute resources than IndexFS. This is due to DeltaFS employing aggressive deep logging at the clients. DeltaFS first logs the entire set of metadata changes of a stage of the workflow and then subsequently uses a compute and I/O intensive parallel log compaction to merge all the changes together into an output snapshot. **While IndexFS spent fewer total CPU core-minutes at the server, it effectively wastes a massive amount of client-side compute node resources by having application processes blocked on filesystem metadata operations** (plotted in Figure 13b as a dashed line). The job as a whole cannot make progress if processes are bottlenecked waiting on IndexFS metadata server operations to complete. DeltaFS uses 52.9× fewer CPU core-minutes than IndexFS in total, of which 98.9% were core-minutes spent on the clients alone. This result is due to IndexFS limiting metadata processing to dedicated server nodes.

Additional experiments showed that this 52.9× reduction in resource usage (core-minutes) of DeltaFS over IndexFS would be decreased to around 28× if IndexFS had two dedicated metadata servers instead of one. That is, if two metadata servers were used, IndexFS could finish all operations more quickly, blocking clients for a less amount of time and leading to reduced total client and server CPU time. If even more metadata servers were added, resource usage for IndexFS would further reduce, until IndexFS is over-provisioned at the server-side and becomes inefficient again. By contrast, DeltaFS allows filesystem resources to be adjusted on a per-job basis and does not require metadata servers to be permanently dedicated. We imagine future work that utilizes machine learning algorithms to guide per-job resource allocation for both filesystem metadata processing and parallel log compaction.

9 RELATED WORK

Large-scale parallel filesystems have long served as an important storage infrastructure in modern computing environments [75]. While the conventional wisdom is to put both namespace servers and file storage into a single layered system [18, 65, 66], DeltaFS envisions them to be loosely related but separate systems [63]. This allows metadata to be accessed from a provisional service spawned by each application on-demand. Data may be placed on a set of unrelated traditional "forever-running" service silos that can be upgraded or extended independently.

Namespace stage-in and stage-out services are available through workload aware filesystems such as the Confuga cluster filesystem [22] and modern burst-buffer software such as Cray's DataWarp [29]. DeltaFS differs from them by not requiring all changes to be merged back to a single, global namespace at job completion (eager synchronization) and instead enables jobs to communicate only on an as-needed basis.

Novel burst-buffer filesystems such as BurstFS [76], GekkoFS [74], and Gfarm/BB [72] provide applications with an ephemeral namespace that has the same life cycle as the job. DeltaFS is able to provide the same semantics as these filesystems. In addition, DeltaFS allows namespaces to outlive their jobs as immutable snapshots in a public registry for inter-job communication. Related snapshots can be merged as needed through scalable parallel log compaction on compute nodes.

Prior work such as Coda [39], BatchFS [86], and Pacon [48] has discussed multiple forms of filesystem metadata writeback caching. DeltaFS differs from them by not requiring even an asynchronously updated global filesystem namespace for maximum parallelism.

Prior work such as FusionFS [84] has demonstrated scalable performance through using distributed compute node resources for an ultra fast parallel filesystem. DeltaFS differs from it by not requiring a portion of compute node resources to be forever dedicated and by enabling a smaller namespace footprint per job.

Searchable filesystems have been extensively studied by work including TagIt [68], BRINDEXER [56], SmartStore [32], Glance [33], Spyglass [46], and HP's StoreAll Storage [38]. DeltaFS may be enhanced with these techniques to enable rich metadata queries.

To quickly absorb a large amount of small files, PLFS [11, 12] used an append-to-end format for high metadata insertion performance at the expense of subsequent reads. To achieve more balanced read and write performance, recent filesystems leveraged more advanced data formats for on-storage metadata management. Examples of such efforts include TABLEFS [58], IndexFS [60], and XtreemFS [35, 70] which used LSM-Trees [53], BetrFS which used Fractal Trees [24, 37], and LocoFS [47] which used a combination of hash tables and B-Trees for file and directory management separately.

Many filesystems partitioned their namespaces for scalable performance. Ceph [77] used a subtree-based partitioning scheme for improved access locality. Farsite [23] leveraged its unique tree-structured file identifiers for namespace partitioning preventing data movement when directories are renamed. IndexFS [60] and skyFS [83] used a directory-based partitioning scheme that aggressively splits a directory as it grows. ShardFS [82], LocoFS [47], and MarFS [36] used a hash-based scheme and separated files from directory partitioning in favor of improved access performance on files.

It is not new to have a cluster of computers collectively share a filesystem on a distributed data store without requiring a dedicated metadata manager [9, 41, 73]. In these shared environments, each filesystem client runs an embedded metadata manager. This manager serves both the client and other clients sharing the same filesystem in a local area network. All metadata managers understand the filesystem's on-storage data format and can dynamically assume responsibility for any files or directories in the filesystem when accessed. To achieve synchronization, distributed locking is used to control access to the shared filesystem and to client data and metadata caches.

Today, such a filesystem metadata design approach is mostly seen in symmetric Storage Area Network (SAN) filesystems [25, 57, 80] as well as the GPFS filesystem [65] widely used in HPC environments. Scalability is often an issue due to the large amount of synchronization needed to access the filesystem [21]. To mitigate this problem, real world deployments typically dedicate a small set of nodes to run filesystem clients with embedded metadata managers. These clients then act as filesystem servers, exporting the filesystem to a larger cluster of filesystem users on job-running compute nodes without metadata managers. Notwithstanding many benefits, such deployment approaches largely defeat the goal of having no dedicated metadata managers and fail to utilize compute node resources to accelerate filesystem metadata operations.

CONCLUSION

As HPC platforms evolve, it is important to periodically stop and reassess their filesystem design to determine if it has become a bottleneck and needs to be rearchitected. A key redesign effort was made by the NASD project [26], which decoupled filesystem data communication from metadata management and leveraged object storage devices for scalable data access. Similar bold ideas that reassess component communication are needed to advance parallel filesystem performance if we are to keep with up the rapidly increasing scale of today's massively-parallel computing environments.

DeltaFS is based on the premise that at exascale and beyond, synchronization of anything global should be avoided. Conventional parallel filesystems, with fully synchronous and consistent namespaces, mandate synchronization with every file create and other metadata operations. This has to stop. Moreover, the idea of dedicating a single filesystem metadata service to meet the needs of all applications in a shared computing environment is archaic and inflexible. This too must stop. DeltaFS shifts away from dedicated metadata servers towards the notion of viewing the filesystem as a service that is dynamically instantiated in the processes of each running application. Cross-job synchronization happens as-needed depending on the description of the workload at hand. DeltaFS leverages client resources to achieve scalable performance. An efficient log-structured format is used that lends itself to deep metadata writeback buffering and merging.

Our evaluation of DeltaFS suggests that its aggressive approach to handling filesystem metadata may be the way forward in order to unlock scalable parallel metadata performance that is unattainable with today's monolithic, one-size-fits-all storage solutions.

ACKNOWLEDGMENT

We thank the anonymous reviewers of our paper for their valuable comments on improving descriptions of DeltaFS cross-job data sharing, aging, resource usage, fault tolerance, complexity, and data conflicts. This manuscript has been approved for unlimited release and has been assigned LA-UR-21-27615. This work has been authored by an employee of Triad National Security, LLC which operates Los Alamos National Lab with the U.S. Department of Energy/National Nuclear Security Administration. We also thank the member companies of the PDL Consortium (Amazon, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung Semiconductor, Seagate, Two Sigma, and Western Digital) for their interest, insights, feedback, and support.

REFERENCES

[1] 2014. OverlayFS. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[2] 2016. APEX Workflows. https://www.nersc.gov/assets/apex-workflows-v2.pdf.

[3] 2018. ISO/IEC 9899:2018 Information technology — Programming languages — C. https://www.iso.org/standard/74528.html.

[4] 2020. IOR/mdtest. https://github.com/hpc/ior.

[5] 2021. Workflows Community Summit: Bringing the Scientific Workflows Community Together. https://doi.org/10.5281/zenodo.4606958 https://arxiv.org/abs/2103.09181.

[6] S. V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (Dec 1996), 66–76. https://doi.org/10.1109/2.546611

[7] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. 2011. Parallel I/O and the Metadata Wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW 11)*. 13–18. https://doi.org/10.1145/2159352.2159356

[8] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the Diversity of Cluster Workloads and Its Impact on Research Results. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC 18)*. 533–546.

[9] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. 1996. Serverless Network File Systems. *ACM Trans. Comput. Syst.* 14, 1 (Feb. 1996), 41–79. https://doi.org/10.1145/225535.225537

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 15)*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[11] John Bent. 2015. PLFS: Software-Defined Storage for HPC. In *High Performance Parallel I/O*, Prabhat and Koziol (Eds.). CRC Press, Chapter 14, 169–176.

[12] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. Article 21, 12 pages. https://doi.org/10.1145/1654059.1654081

[13] John Bent, Brad Settlemyer, and Gary Grider. 2016. Serving Data to the Lunatic Fringe: The Evolution of HPC Storage. *USENIX ;login:* 41, 2 (June 2016).

[14] Andrew D. Birrell and Bruce Jay Nelson. 1983. Implementing Remote Procedure Calls. *SIGOPS Oper. Syst. Rev.* 17, 5 (Oct. 1983). https://doi.org/10.1145/773379.806609

[15] S. A. Brandt, E. L. Miller, D. D. E. Long, and Lan Xue. 2003. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 2003 International Conference on Massive Storage Systems and Technologies (MSST 03)*. 290–298. https://doi.org/10.1109/MASS.2003.1194865

[16] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 335–350.

[17] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD 19)*. 1773–1786. https://doi.org/10.1145/3299869.3314045

[18] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase & Conference (ALS 00)*.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (June 2008), 4:1–4:26. https://doi.org/10.1145/1365815.1365816

[20] Peter F. Corbett and Dror G. Feitelson. 1996. The Vesta Parallel File System. *ACM Trans. Comput. Syst.* 14, 3 (Aug. 1996), 225–264. https://doi.org/10.1145/233557.233558

[21] Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux, and William H. Tetzlaff. 1995. Evaluation of Design Alternative for a Cluster File System. In *Proceedings of the 1995 USENIX Annual Technical Conference (USENIX ATC 95)*.

[22] Patrick Donnelly, Nicholas Hazekamp, and Douglas Thain. 2015. Confuga: Scalable Data Intensive Computing for POSIX Workflows. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster Cloud and Grid Computing (CCGRID 15)*. 392–401. https://doi.org/10.1109/CCGrid.2015.95

[23] John R. Douceur and Jon Howell. 2006. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 321–334.

[24] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 12)*.

[25] Mark Fasheh. 2006. OCFS2: The Oracle Clustered File System, Version 2. In *Proceedings of the 2006 Linux Symposium*. 289–302.

[26] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. 1998. A Cost-effective, High-bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 98)*. 92–103. https://doi.org/10.1145/291069.291029

[27] Google. 2013. LevelDB. https://github.com/google/leveldb/.

[28] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.

[29] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. 2016. Architecture and Design of Cray Datawarp. In *Proceedings of the 2016 Cray User Group (CUG 2016)*. https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf.

[30] W. Daniel Hillis and Lewis W. Tucker. 1993. The CM-5 Connection Machine: A Scalable Supercomputer. *Commun. ACM* 36, 11 (Nov. 1993), 31–40. https://doi.org/10.1145/163359.163361

[31] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 51–81. https://doi.org/10.1145/35037.35059

[32] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. 2009. SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for next-Generation File Systems. In *Proceedings of the 2009 Conference on High Performance Computing, Networking, Storage, and Analysis (SC 09)*. Article 10, 12 pages. https://doi.org/10.1145/1654059.1654070

[33] H. Howie Huang, Nan Zhang, Wei Wang, Gautam Das, and Alexander S. Szalay. 2011. Just-in-Time Analytics on Large File Systems. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 11)*. 16.

[34] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[35] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. 2007. XtreemFS - A Case for Object-Based Storage in Grid Data Management. In *Proceedings of the 2007 VLDB Workshop on Data Management in Grids (DMG 2007)*.

[36] Jeff Inman, Will Vining, Garrett Ransom, and Gary Grider. 2017. MarFS, a Near-POSIX Interface to Cloud Objects. *USENIX ;login:* 42, 1 (Jan. 2017).

[37] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*. 301–315.

[38] Charles Johnson, Kimberly Keeton, Charles B. Morrey, Craig A. N. Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro. 2014. From Research to Practice: Experiences Engineering a Production Metadata Database for a Scale out File System. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. 191–198.

[39] James J. Kistler and M. Satyanarayanan. 1992. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 3–25. https://doi.org/10.1145/146941.146942

[40] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[41] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. 1986. VAXcluster: A Closely-Coupled Distributed System. *ACM Trans. Comput. Syst.* 4, 2 (May 1986), 130–146. https://doi.org/10.1145/214419.214421

[42] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[43] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sep. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[44] LANL. 2018. Grand Unified File-Index. https://github.com/mar-file-system/GUFI.

[45] LANL, NERSC, SNL. 2018. Crossroads Workflows. https://www.lanl.gov/projects/crossroads/__internal/__blocks/xroads-workflows.pdf.

[46] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. 2009. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST 09)*. 153–166.

[47] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A Loosely-coupled Metadata Service for Distributed File Systems. In *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*. Article 4, 12 pages. https://doi.org/10.1145/3126908.3126928

[48] Yubo Liu, Yutong Lu, Zhiguang Chen, and Ming Zhao. 2020. Pacon: Improving Scalability and Efficiency of Metadata Service through Partial Consistency. In *Proceedings of the 2020 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 20)*. 986–996. https://doi.org/10.1109/IPDPS47924.2020.00105

[49] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*. Article 50, 12 pages. https://doi.org/10.1109/SC.2016.49

[50] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[51] Don Monroe. 2020. Fugaku Takes the Lead. *Commun. ACM* 64, 1 (Dec. 2020), 16–18. https://doi.org/10.1145/3433954

[52] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. 1988. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 134–154. https://doi.org/10.1145/35037.42183

[53] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[54] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. 2020. Job Characteristics on Large-Scale Systems: Long-Term Analysis, Quantification, and Implications. In *Proceedings of the 2020 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 20)*. Article 84, 17 pages. https://doi.org/10.1109/SC41405.2020.00088

[55] Swapnil Patil and Garth Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 11)*.

[56] Arnab K. Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R. Butt. 2020. Efficient Metadata Indexing for HPC Storage Systems. In *Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster Cloud and Internet Computing (CCGrid 20)*. 162–171. https://doi.org/10.1109/CCGrid49817.2020.00-77

[57] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. 1999. A 64-bit, shared disk file system for Linux. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (MASS 99)*. 22–41. https://doi.org/10.1109/MASS.1999.829973

[58] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 145–156.

[59] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *Proc. VLDB Endow.* 10, 13 (Sept. 2017), 2037–2048. https://doi.org/10.14778/3151106.3151108

[60] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*. 237–248. https://doi.org/10.1109/SC.2014.25

[61] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* 17, 7 (July 1974), 365–375. https://doi.org/10.1145/361011.361061

[62] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[63] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Computer Science and Technology* 35, 1, Article 121 (2020), 23 pages. https://doi.org/10.1007/s11390-020-9802-0

[64] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. 1985. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*.

[65] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*. 231–244.

[66] Philip Schwan. 2003. Lustre: Building a File System for 1000-Node Clusters. In *Proceedings of the 2003 Linux Symposium*. 380–386.

[67] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 International Conference on Massive Storage Systems and Technologies (MSST 10)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[68] Hyogi Sim, Youngjae Kim, Sudharshan S. Vazhkudai, Geoffroy R. Vallée, Seung-Hwan Lim, and Ali R. Butt. 2017. Tagit: An Integrated Indexing and Search Service for File Systems. In *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*. Article 5, 12 pages. https://doi.org/10.1145/3126908.3126929

[69] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER 13)*. 1–8. https://doi.org/10.1109/CLUSTER.2013.6702617

[70] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. 2010. BabuDB: Fast and Efficient File System Metadata Storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 10)*. 51–58. https://doi.org/10.1109/SNAPI.2010.14

[71] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 05)*. 2–11. https://doi.org/10.1109/ICDE.2005.1

[72] Osamu Tatebe, Shukuko Moriwake, and Yoshihiro Oyama. 2020. Gfarm/BB — Gfarm File System for Node-Local Burst Buffer. *Journal of Computer Science and Technology* 35, 1 (2020), 61–71. https://doi.org/10.1007/s11390-020-9803-z

[73] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP 97)*. 224–237. https://doi.org/10.1145/268998.266694

[74] Marc-André Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2020. Gekkofs - a temporary burst buffer file system for HPC applications. *Journal of Computer Science and Technology* 35, 1 (2020), 72–91. https://doi.org/10.1007/s11390-020-9797-6

[75] Jeffrey S. Vetter. 2019. *Contemporary High Performance Computing: From Petascale toward Exascale*. Vol. 1-2-3. CRC Press.

[76] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-buffer File System for Scientific Applications. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*. Article 69, 12 pages. https://doi.org/10.1109/SC.2016.68

[77] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 307–320.

[78] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage (PDSW 07)*. 35–44. https://doi.org/10.1145/1374596.1374606

[79] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*. Article 2, 17 pages.

[80] Steven Whitehouse. 2007. The GFS2 Filesystem. In *Proceedings of the 2007 Linux Symposium*. 253–260.

[81] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and Unix Semantics in Namespace Unification. *ACM Trans. Storage* 2, 1 (Feb. 2006), 74–105. https://doi.org/10.1145/1138041.1138045

[82] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. 2015. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*. 236–249. https://doi.org/10.1145/2806777.2806844

[83] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. 2009. Adaptive and Scalable Metadata Management to Support a Trillion Files. In *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. Article 26, 11 pages. https://doi.org/10.1145/1654059.1654086

[84] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. 2014. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Proceedings of the 2014 IEEE International Conference on Big Data (BigData 14)*. 61–70. https://doi.org/10.1109/BigData.2014.7004214

[85] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling Embedded In-situ Indexing with DeltaFS. In *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*. Article 3, 15 pages. https://doi.org/10.1109/SC.2018.00006

[86] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*. 1–6. https://doi.org/10.1109/PDSW.2014.7