DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers

Qing Zheng[1], Kai Ren[1], Garth Gibson[1], Bradley W. Settlemyer[2], Gary Grider[2]
[1]Carnegie Mellon University
[2]Los Alamos National Laboratory
Email: {zhengq, kair, garth} @ cs.cmu.edu
{bws, ggrider} @ lanl.gov

ABSTRACT

High performance computing fault tolerance depends on scalable parallel file system performance. For more than a decade scalable bandwidth has been available from the object storage systems that underlie modern parallel file systems, and recently we have seen demonstrations of scalable parallel metadata using dynamic partitioning of the namespace over multiple metadata servers. But even these scalable parallel file systems require significant numbers of dedicated servers, and some workloads still experience bottlenecks. We envision exascale parallel file systems that do not have any dedicated server machines. Instead a parallel job instantiates a file system namespace service in client middleware that operates on only scalable object storage and communicates with other jobs by sharing or publishing namespace snapshots. Experiments shows that our serverless file system design, DeltaFS, performs metadata operations orders of magnitude faster than traditional file system architectures.

1. INTRODUCTION

HPC clusters are traditionally tiered as separate sets of *compute* and *storage* nodes, providing massive numbers of CPU cores, low-latency interconnects, as well as fast concurrent data bandwidth typically managed by an underlying parallel file system[1]. Because of the long-standing I/O challenges imposed by the checkpoint-based fault-tolerance commonly adopted by batch applications, an additional layer of *burst-buffer* nodes are being deployed in new HPC clusters to capture and forward bursty checkpoint data in high-end flash devices[2–4]. Most HPC clusters also include a small set of *head* nodes to serve as dedicated resources for the metadata path of an underlying parallel file system, usually Lustre[5], GPFS[6], PanFS[7], or PVFS[8]. In this paper, we refer to this classic HPC setting as the *"Lustre"* model.

With only a single or a few metadata servers, HPC applications running on massive-scale computing clusters[2, 9] often experience significant bottlenecks when accessing the file system namespace, therefore wasting a large number of

© 2015 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
PDSW2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4008-3/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2834976.2834984.

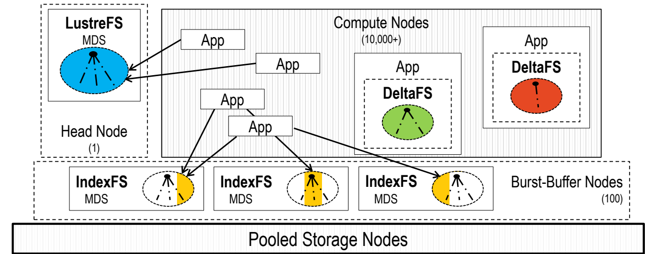


Figure 1: *A typical HPC cluster consisting of compute, storage, head, and burst-buffer nodes under different system models: centralized LustreFS server, partitioned IndexFS servers, or fully decentralized DeltaFS. While Lustre and IndexFS requires dedicated metadata servers, DeltaFS allows each application to instantiate a private namespace on compute nodes and self-manage its metadata on storage nodes.*

CPU cycles while blocking on file system metadata operations[10, 11]. Trying to alleviate this metadata bottleneck, a couple of recent file systems have demonstrated scalable performance in metadata through dynamic namespace partitioning over a large collection of metadata servers[12–15]. As an example from our previous work, IndexFS [15] can be deployed on top of a subset of burst-buffer nodes with each node serving as a dedicated machine holding a piece of the global file system namespace. Unfortunately, with this *"IndexFS"* model, a potentially large number of machines must constantly be reserved in order for the system to be ready for an envisioned peak metadata demand. Even if the right number of metadata servers can be known before-hand, dedicated machines can easily isolate a considerable amount of hardware resources that could be better utilized running user jobs. This increases the cost of building HPC clusters.

1.1 DeltaFS Vision

Because access to a global namespace typically requires centralized serializable coordination with dedicated resources and is all too often a performance bottleneck, designing a file system without a global namespace and without any dedicated metadata servers should, in principle, enable the system to grow to much larger scales in a more cost-effective way. In BatchFS [16], our prior extension to IndexFS, each file system client is able to instantiate a private namespace in a snapshot of the global namespace and obtain capabilities to pre-execute metadata operations directly in that private namespace. At a later point, a client would choose to submit changes to its private namespace to the global namespace to

merge updates through a single bulk insertion operation[16]. We showed that this asynchronous mechanism can afford a batch job the ability to efficiently execute a huge set of concurrent metadata operations with only a small interaction with centralized metadata servers, which is particularly useful for achieving high-performance checkpointing[16–19].

Encouraged by BatchFS, in this paper we propose a new (and more extreme) file system metadata design, DeltaFS, which departs from BatchFS by not defining even an asynchronously updated global namespace. Rather than pushing changes in every private namespace back to a centralized controller to serialize metadata updates and enforce global semantics, DeltaFS uses a registration service to index different materialized file system views and allow applications to publish and lookup namespace snapshots that are individually-consistent, loosely-coupled, with each ready to be combined[1] with other snapshots to form new file system views. DeltaFS is envisioned to have the following features:

- Serverless design featuring zero dedicated metadata servers, no global file system namespace, and no ground truth;
- Client-funded metadata service harnessing compute nodes to handle metadata and achieve highly agile scalability;
- Freedom from unjustified synchronization among HPC applications that do not need to use the file system to communicate;
- Write-optimized LSM-based[20] metadata representation with file system namespace snapshots as the basis of inter-job data sharing and workflow execution;
- A file system as no more than a thin service composed by each application at runtime to provide a temporary view of a private namespace backed by a stack of immutable snapshots and a collection of shared data objects;
- Simplified data center storage consisting of multiple independent underlying object stores, providing flat namespaces of data objects, and oblivious of file system semantics.

DeltaFS is a work-in-progress. In this paper, we present its high-level design and characterize its performance. In Section 2 and 3, we show DeltaFS's metadata architecture and runtime interaction advantages. In Section 4, we show preliminary experimental results. Finally, we discuss related work and draw conclusions in Section 5 and 6.

2. SYSTEM DESIGN

Decoupling data from metadata management[7, 21, 22], DeltaFS is designed as file system metadata middleware created on-demand on top of an underlying storage infrastructure that stores file data and handles I/O operations. All metadata operations are executed directly in DeltaFS, which internally reuses the data path provided by the shared underlying storage to store file system metadata. While one can use a parallel file system to serve as the underlying storage, DeltaFS expects no more than a set of global object stores[2] each exposing a flat namespace of data objects[23]. A single DeltaFS namespace can name objects in different object stores without requiring the stores to know about each other. This enables diverse and changing object store deployments within a single HPC data center.

In order to better harness the computation and interconnect resources on compute nodes, DeltaFS enables batch applications to manage their own namespaces and avoid unnecessary coordination. DeltaFS user library code linked into each batch application process constitutes a private metadata server, which can be viewed as a full-fledged but embedded metadata server capable of executing namespace operations and writing journals of metadata mutations (relative to an initial namespace snapshot) to represent and record updates[3] (§3.1). Batch applications obtain an input namespace by collecting appropriate namespace snapshots from public registration services (possibly the result of an application-informed search predicate), and publish their output results as new snapshots ready to be consumed by future applications. Publication and subsequent collection can be deferred and aggregated by an integrated job scheduler that executes workflows and manages cluster resources (§2.2).

In lieu of the traditional "global namespace", DeltaFS uses one or more external namespace registries to serve as repositories of shared file system snapshots. The insertion, deletion, and selection of snapshots from these registries captures the true communication and synchronization between unrelated applications. DeltaFS is not designed for interactive processes that use the file system as an OLTP database to implement inter-process communication. "Almost-batch" processes in need of some anonymous synchronization should seek a mechanism outside the file system to communicate (e.g. by passing messages or tokens, or using a coordination service[24]). Figure 1 shows our *"DeltaFS"* model, along with the *"IndexFS"* and the *"Lustre"* model discussed earlier.

2.1 Metadata Representation

DeltaFS represents namespace metadata as ordered key-value pairs that are managed by LevelDB[25] using LSM-trees implemented on storage as *SSTables* (Sorted String Tables). Each SSTable stores a range of immutable and indexed key-value entries and serves as the physical format for metadata migration and aggregation[15, 16]. SSTables are ordered with entries in newer tables superseding entries in older ones. In DeltaFS, each file system snapshot is directly represented as a set of SSTables. Snapshots are essentially "copy-on-write" data structures with each one built on top of prior tables using a set of new SSTables to hold overriding entries – the "delta" of the two. Every DeltaFS application uses an initial snapshot to bootstrap, and will typically generate a new snapshot as a child before terminating. As such, snapshots can be thought of as nodes in a *snapshot tree.*

2.2 Namespace Propagation

Instead of synchronizing with a centralized metadata service implementing serializable transactions on a single file system namespace, unrelated applications in DeltaFS communicate with each other through published file system snapshots (generally only at the beginning and end of each application or workflow run). Allowing applications to explicitly control namespace visibility and timings for communication effectively confines the synchronization scope to a complete application run and avoids the unnecessary resource

[1] With all conflicts reconciled on demand (§2.3).

[2] DeltaFS today requires object stores that provide "multi-writes" or "non-transactional appends"; however, given only a "single-write" or "PUT" interface, we could emulate "appends" with deep buffers and a naming convention.

[3] Snapshots are essentially aggregations of metadata journals generated by a series of previous applications (§2.1).

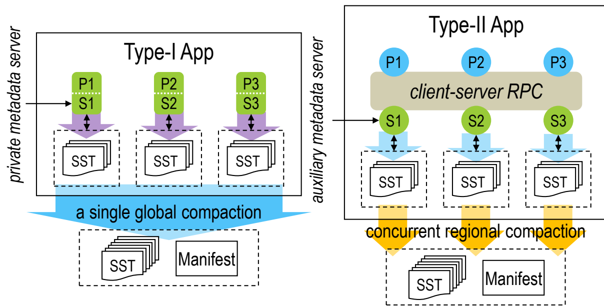


Figure 2: *A DeltaFS app runs either with private metadata servers generating a set of overlapping outputs (left), or auxiliary metadata servers holding partitioned outputs (right).*

contention associated with shared coordinators. Instead of implementing a global catalog holding all registered snapshots, DeltaFS employs multiple available, fault-tolerant, scalable registries to serve as external indices for snapshpts. These registries can be implemented as key-value stores[26, 27], replicated databases[24, 28], directory services[29, 30], or simply as command-line arguments that directly specify which snapshots to use. This avoids introducing another centralized service unnecessarily, and enables flexible system integration. For example, batch jobs may take advantage of an integrated scheduler to automate workflow execution, resource allocation, and namespace propagation.

2.3 No Ground Truth

Unlike traditional file systems that expose a single namespace with all its metadata owned by dedicated servers to provide the *"ground truth"*, DeltaFS offers no ultimate truth[4], but only a collection of file system snapshots that applications could use as "facts" to construct their own namespace views. In doing so, each application loads and merges a set of snapshots[5], detects and resolves conflicts on-demand, and creates a new namespace view that is consistent with its own reconciliation policy[31], such as the "last-writer-wins" rule.

3. APPLICATION EXECUTION

A DeltaFS cluster consists of one or more object storage platforms running on storage nodes and a large collection of compute nodes hosting one or more unrelated batch applications. Unrelated applications coordinate their namespace metadata with assistance from a public registration service that provides snapshot propagation. With shared access to file system metadata stored as snapshots in the underlying storage infrastructure, a DeltaFS application is able to retrieve existing file system objects and perform metadata operations independent of and in parallel with metadata activities concurrently performed by other applications. With two alternative namespace partitioning strategies and their corresponding metadata key distributions, a DeltaFS application either links to a user library that serves as a *private* metadata service (§3.1), or initializes a set of *auxiliary* metadata servers to provide application-wise metadata coordination with dynamic namespace partitioning (§3.2).

3.1 Overlapping Sub-namespaces (Type-I)

As is illustrated in Figure 2, a Type-I application is made up of a set of non-communicating (but related) processes using their own embedded private metadata service to independently execute metadata operations. Each process reads from a set of input snapshots, and performs metadata mutations on them as part of the computation. This yields a collection of mutually-isolated sub-namespaces that the application programmer constructs to have no metadata key collisions. The snapshot generated by a Type-I application can be seen as an union of all its sub-namespace snapshots, with each stored as a different set of SSTables with possibly overlapping key ranges. While enabling very fast write-intensive workloads, a later reader of a Type-I application snapshot may have to search many SSTables in order to find any specific metadata key. To avoid this *read amplification*, a "compaction" job[25, 27] could be launched to merge, shuffle, and repartition metadata records, yielding a new set of SSTables with globally non-overlapping key ranges. This ensures a single SSTable access for each metadata key lookup.

3.2 Partitioned Sub-namespaces (Type-II)

Unlike Type-I applications, each process in a Type-II application continuously maintains a consistent, partitioned view of the entire application namespace, typically because each process sometimes requires immediate access to metadata produced by other processes in the middle of a run[32]. Therefore, Type-II applications are assisted by temporary metadata machinery (auxiliary metadata service[15]) serving the same function as traditional IndexFS servers: providing scalable and synchronous metadata access (exclusively to the processes of that application) through dynamic namespace partitioning. As can be seen in Figure 2, every Type-II application process communicates with a private set of auxiliary metadata servers to execute namespace operations[6].

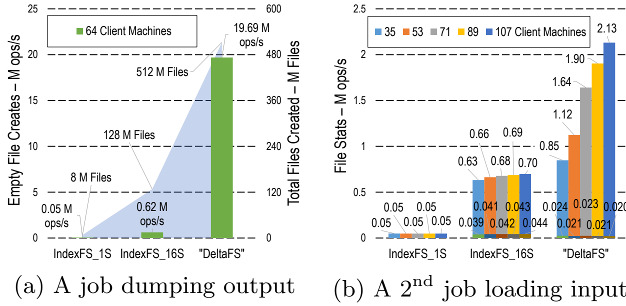
Through dynamic namespace partitioning, the final snapshots generated by Type-II applications are each comprised of sub-namespaces featuring pre-partitioned keys. As such, each metadata key can only appear in the SSTables of a single sub-namespace, whose key ranges do not overlap other sub-namespaces. To eliminate *read amplification* altogether, a local "compaction" can be launched concurrently on each sub-namespace, producing a new set of SSTables whose non-overlapping key ranges are global to the entire application.

4. MEASUREMENTS

In this section, we report experiments done on a DeltaFS extension of IndexFS [15] to show the promise of our serverless design. All our experiments were performed on a 125-machine cluster consisting of one head node, 16 storage nodes, and 108 compute nodes. Each machine was equipped with 4 quad-core CPUs with 2GB of RAM per core, one 20Gb/s Infiniband link, and one HDD disk holding a Linux OS. All resources were part of the Nome[33] testbed operated by NSF PRObE[34]. The shared underlying storage supporting IndexFS was implemented by Ceph RADOS[23] as 64 OSDs (*object storage devices*) on 16 storage nodes[7]. Each OSD was built on a local file system (XFS) mounted

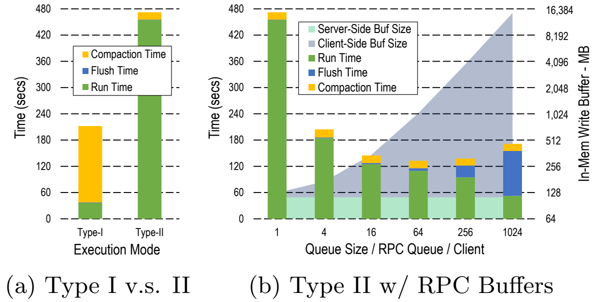
[4] And there are no dedicated services to enforce such truth.
[5] See our BatchFS [16] paper for a discussion on security.

[6] While we expect many jobs to be Type-I, it may be worthwhile for a Type-I job to execute as a Type-II, especially if a read-intensive phase is known to be needed soon (§4.2).
[7] To run IndexFS on Ceph RADOS (or other object stores), we have implemented a shim layer to translate POSIX file system calls invoked from LevelDB to RADOS requests.



(a) A job dumping output (b) A 2nd job loading input

Figure 3: *Performance of three IndexFS deployments creating and stating files within a single directory. Fig. (b) shows both aggregated and per-node throughput. DeltaFS auxiliary metadata servers appear to be less efficient because of over-provisioning relative to the number of requesting clients.*



(a) Type I v.s. II (b) Type II w/ RPC Buffers

Figure 4: *Performance of DeltaFS apps inserting empty files into a single directory, using either Type-I or II executions, with scaling RPC queue sizes and in-memory buffer.*

on a RAM disk so as to emulate the performance of a fast backend flash or disk array. In total, IndexFS was provided with 256GB for SSTable storage, with each SSTable snappy-compressed[25] and stored as a single RADOS object.

4.1 Traditional v.s. Serverless Architecture

We used three differently configured IndexFS deployments to demonstrate the scalability of our "DeltaFS" design as compared to the widely-used "Lustre" design (a single dedicated metadata server) and the recently published "IndexFS" design (a large number of dedicated metadata servers). We ran eight IndexFS metadata processes on one head node to emulate a non-partitioned "Lustre" metadata service running on a single machine. To evaluate our "IndexFS" model, we started 128 IndexFS metadata processes evenly spread across 16 storage nodes. To implement our "DeltaFS" model, we only ran a single IndexFS metadata process on the head node since all metadata operations would be handled by "private metadata servers" embedded inside each client process. In all cases, we had 512 IndexFS client processes evenly distributed on 64 compute nodes. For "DeltaFS", we enabled IndexFS bulk insertion[8] at all clients so that they would behave as simplified DeltaFS private metadata servers.

Figure 3a shows the performance of the three IndexFS deployments performing empty file creations under a shared directory. IndexFS with 16 metadata server machines was an order of magnitude faster than IndexFS running on a single machine, mostly by consuming an order of magnitude more dedicated resources. While more files were actually created, "DeltaFS" managed to deliver another two orders of magnitude of throughput because 1) it enjoyed a shorter metadata path involving only private metadata servers that avoided the use of RPC; 2) its serverless design effectively moved metadata serving from dedicated servers to client machines so more hardware resources were used; and 3) its metadata output was left un-partitioned and un-compacted with an assumption that a later compaction job would "fix" the output in a single pass (this is discussed further in §4.2).

Figure 3b shows a performance comparison among five distinct-sized jobs (running on 35-107 client machines) randomly accessing metadata generated by the preceding job in the first experiment (running on 64 client machines). For both IndexFS setups, throughput was essentially a function of dedicated server resources, since each namespace was constantly served by a fixed set of server daemons on a given set of nodes, leading to a waste of resources if clients were too few, or a bottleneck if clients were too many.

Not having any dedicated servers, each "DeltaFS" job read the metadata by setting up a cluster of auxiliary metadata servers to dynamically load the provided snapshot (as a collection of pre-compacted SSTables). Compared with the two IndexFS jobs, the "DeltaFS" jobs showed higher throughput since they were effectively leveraging a larger set of hardware resources to serve metadata. Another important benefit of this DeltaFS approach is the freedom of each job to deploy its own metadata server tier. This symmetric and highly flexible metadata architecture allows metadata performance to scale agilely with individual job allocations[9], rather than as a global number set by cluster administrators.

4.2 Type-I v.s. Type-II Executions

To compare the performance between partitioned (Type-II) and non-partitioned (Type-I) executions, we ran experiments with a single IndexFS metadata server on the head node and 864 bulk insertion client processes on 108 compute nodes to emulate a Type-I job using a set of private metadata servers, and experiments with 864 IndexFS metadata server processes co-located with 864 regular client processes on the same 108 compute nodes to emulate a Type-II job associated with an auxilary metadata service.

Figure 4a shows the experimental results of the two kinds of jobs each creating 512M empty files into a directory. The green bar shows the time it took for each job to finish all file creates, the blue bar shows the time it took for all servers (either private metadata servers or auxiliary metadata servers) to flush their remaining in-memory write buffers to the underlying storage, and finally the yellow bar shows the time it took for a compaction job to merge-sort SSTables. Compared with Type-I, the Type-II job exhibited a much longer file creation time since each file create operation had to incur at least one RPC to a remote auxiliary metadata server. However, the use of these auxiliary metadata servers permitted a well-partitioned metadata key distribution such that the later compaction of the entire key space could be safely

[8] IndexFS can delegate authority of an empty directory to a set of cooperating clients as leases to drive efficient metadata execution and aggressive client-side batching[15].

[9] In our implementation, there is also no need for the size of a job to divide or be a multiplier of the size of a previous job in order to obtain proportional metadata scalability.

implemented as a set of independent local sub-tasks concurrently executed at each auxiliary metadata process and taking much less time to complete. Nevertheless, RPC overhead dominated, which made Type-I the winner in Figure 4a.

To better characterize RPC overhead, we added write-behind buffering at each IndexFS client to accumulate and coalesce RPC requests before sending them to the destination server as a single batch. Figure 4b shows the effect of different RPC queue sizes, ranging from 1 to 1024 requests per destination auxiliary server process at each client process, collectively capable of holding 764K to 764M pending RPC requests. In Figure 4b, the blue bar now includes the time it took for all clients to flush their RPC queues on test completion. With RPC write-behind buffers, Type-II jobs took much less time to complete, although deep queues can actually do a disservice by failing to keep auxiliary servers busy until the final flush. In general, a storage system performing aggressive buffering should run faster than one with less buffering, though at the same time being more susceptible to data loss during failures. As such, Type-II is generally better if neither larger memory consumption nor an extended window of data loss is a concern. Type-I, however, can be especially useful when the compaction can be delayed through a series of mostly-write jobs until a read-intensive phase sets in, lending itself to more efficient checkpointing[17], where data is not always needed immediately.

5. RELATED WORK

Our initial designs were inspired by PLFS[17, 35], which is a library file system capable of shaping I/O access patterns and aggregating small files. PLFS showed us the efficiency of being a library file system, the importance of decoupling sharing where possible, as well as the influence of metadata compaction on read/write performance[36]. Like PLFS, DeltaFS's metadata plane can also be used to pack small files[37]. In addition, both PLFS and DeltaFS allow user applications to "compact" metadata (either online or offline) to obtain a new metadata organization that is optimized for future read operations. Unlike PLFS, DeltaFS's table-based metadata representation is general purpose and integrated with the file system. Also unlike PLFS, DeltaFS does not assume an underlying parallel file system[38], and can operate on top of one or multiple simple object stores.

Serverless file systems[6, 39] are traditionally characterized by a set of symmetric file servers that are each capable of serving the entire file system namespace. This architecture is reused by both BatchFS [16] and DeltaFS to enable flexible metadata migration and service allocation, and is also extended by DeltaFS to be literally "serverless".

Client-funded metadata is a technique initially proposed by BatchFS [16] to advocate the use of abundant client resources to serve file system metadata in an efficient way. This idea is inherited by DeltaFS with each DeltaFS application operating upon a materialized namespace view projected by a provisional metadata service. However, different from BatchFS, each published namespace in DeltaFS is regarded as a 1st-class entity, rather than a temporary write buffer that is eventually merged into a global namespace.

The idea of leveraging client resources can also be applied to data operations, such as storing application data directly on compute nodes using their local storage[40], or buffering checkpointing data inside the local memory of each compute node[18] or at a set of specialized I/O server nodes equipped with fast flash storage[3, 4]. Since most of these techniques focus on speeding up the data path, they are orthogonal and complementary to our work.

Object storage has been increasingly used as an underlying storage infrastructure upon which multiple richer storage abstractions are being built[7, 22, 41–43]. We envision future HPC data centers to be object-storage oriented, with the file system being one of the services layered atop.

Conflict resolution protocols have been widely used in different storage systems to ensure a consistent view of all stored objects[16, 26, 31, 44]. Like many existing implementations, DeltaFS allows conflicts to be resolved using application domain logic[26, 31], and delays resolving conflicts until the data is actually requested[16, 26]. Unlike many existing solutions, DeltaFS does not enforce a single namespace as the ultimate destination into which all updates will eventually get merged. Instead, each conflict resolution creates a new namespace, with the original left intact.

6. CONCLUSION

At exascale, metadata is no longer a trivial step that adds only a tiny latency before data operations. In LANL's new Trinity cluster[2], it takes 256s for every CPU core to create a file in the global Lustre namespace, but only 600s for the entire 2PB of memory to be dumped from compute nodes to burst-buffer nodes. Traditional file systems are unlikely to scale to exascale because: 1) centralized metadata requires either expensive hardware to scale-up or a large number of dedicated machines to scale-out; 2) imposing a single namespace forces applications to frequently synchronize with each other mostly unnecessarily; 3) ensuring metadata integrity and strong consistency over a global namespace demands the use of a dedicated (and easily bottlenecked) coordinator to enforce system invariants; and 4) classic on-disk metadata representation lacks efficient support for fast metadata insertion, migration, redistribution, and aggregation.

Through a serverless design, DeltaFS will not need the dedicated server machines found in traditional parallel file systems. Each application can start from immutable snapshots and manage their own metadata using their own resources. DeltaFS's LSM-based metadata representation will be optimized for write, efficient to share and merge, and can enable appropriate compaction to optimize later retrieval. In addition, DeltaFS advocates the use of object stores to provide the underlying storage, and to assist with security enforcement, garbage collection, as well as administrative data purging. Preliminary experiments demonstrated that DeltaFS can be orders of magnitude faster for metadata than file systems relying on dedicated metadata services.

7. ACKNOWLEDGMENTS

This research was supported in part by the DOE and Los Alamos National Laboratory, under contract number DE-AC52-06NA25396 subcontracts 161465 and 153593 (IRHPIT), the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE, `www.nmc-probe.org`), and Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We also thank the member companies of the PDL Consortium (Actifio, Avago, EMC, Facebook, Google, Hewlett-Packard, Hitachi, Intel, Microsoft, MongoDB, NetApp, Oracle, Samsung, Seagate, Symantec and Western Digital).

References

[1] S. Lang et al. "I/O Performance Challenges at Leadership Scale". In: *SC*. 2009.

[2] *Trinity*. `http://www.lanl.gov/projects/trinity/`.

[3] N. Ali et al. "Scalable I/O forwarding framework for high-performance computing systems". In: *CLUSTER*. 2009.

[4] N. Liu et al. "On the role of burst buffers in leadership-class storage systems". In: *MSST*. 2012.

[5] P. Schwan. "Lustre: Building a file system for 1000-node clusters". In: *Linux Symposium*. 2003.

[6] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". In: *FAST*. 2002.

[7] B. Welch et al. "Scalable Performance of the Panasas Parallel File System". In: *FAST*. 2008.

[8] P. H. Carns et al. "PVFS: A parallel file system for Linux clusters". In: *Linux Showcase and Conference*. 2000.

[9] *Titan*. `https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/`.

[10] S. R. Alam et al. "Parallel I/O and the metadata wall". In: *PDSW*. 2011.

[11] R. Latham, R. Ross, and R. Thakur. "The impact of file systems on MPI-IO scalability". In: *EuroPVM/MPI*. 2004.

[12] S. A. Weil et al. "Dynamic Metadata Management for Petabyte-Scale File Systems". In: *SC*. 2004.

[13] J. Xing et al. "Adaptive and Scalable Metadata Management to Support a Trillion Files". In: *SC*. 2009.

[14] S. Patil and G. Gibson. "Scale and Concurrency of GIGA+: File System Directories with Millions of Files". In: *FAST*. 2011.

[15] K. Ren et al. "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion". In: *SC*. 2014.

[16] Q. Zheng, K. Ren, and G. Gibson. "BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers". In: *PDSW*. 2014.

[17] J. Bent et al. "PLFS: a checkpoint filesystem for parallel applications". In: *SC*. 2009.

[18] R. Rajachandrasekar et al. "A 1 PB/s File System to Checkpoint Three Million MPI Tasks". In: *HPDC*. 2013.

[19] R. Prabhakar et al. "Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines". In: *ICDCS*. 2011.

[20] P. O'Neil et al. "The Log-structured Merge-tree". In: *Acta Inf.* 33.4 (June 1996).

[21] D. Hildebrand and P. Honeyman. "Exporting storage systems in a scalable manner with pNFS". In: *MSST*. 2005.

[22] S. A. Weil et al. "Ceph: A Scalable, High-Performance Distributed System". In: *OSDI*. 2006.

[23] S. A. Weil et al. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters". In: *PDSW*. 2007.

[24] P. Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX ATC*. 2010.

[25] LevelDB. *A fast and lightweight key/value database library.* `https://github.com/google/leveldb/`.

[26] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *SOSP*. 2007.

[27] F. Chang et al. "BigTable: a distributed storage system for structured data". In: *OSDI*. 2006.

[28] M. Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems". In: *OSDI*. 2006.

[29] *OpenLDAP*. `http://www.openldap.org/`.

[30] *AWS Directory Service*. `https://aws.amazon.com/directoryservice/`.

[31] D. B. Terry et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *SOSP*. 1995.

[32] H. Greenberg, J. Bent, and G. Grider. "MDHIM: A Parallel Key/Value Framework for HPC". In: *HotStorage*. 2015.

[33] *Nome*. `http://nome.nmc-probe.org/`.

[34] G. Gibson et al. "PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research". In: *USENIX ;login:* 38.3 (June 2013).

[35] A. Torres and D. Bonnie. *Small File Aggregation with PLFS*. `http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-22024`. 2013.

[36] J. He et al. "Discovering Structure in Unstructured I/O". In: *PDSW*. 2012.

[37] K. Ren and G. Gibson. "TableFS: Enhancing Metadata Efficiency in the Local File System". In: *USENIX ATC*. 2013.

[38] C. Cranor, M. Polte, and G. Gibson. "Structuring PLFS for Extensibility". In: *PDSW*. 2013.

[39] T. E. Anderson et al. "Serverless Network File Systems". In: *SOSP*. 1995.

[40] D. Zhao et al. "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems". In: *Big Data*. 2014.

[41] G. A. Gibson et al. "A Cost-effective, High-bandwidth Storage Architecture". In: *ASPLOS*. 1998.

[42] B. Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *SOSP*. 2011.

[43] J. Chen et al. "Walnut: A Unified Cloud Object Store". In: *SIGMOD*. 2012.

[44] J. J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System". In: *SOSP*. 1991.