Cover Letter: Cudele (Paper 306)

We thank the reviewers for their hard work and genuinely helpful suggestions. In addition to this cover letter, we have posted a PDF online explicitly showing additions in **blue** and deletions in **red**. Three issues were common across reviewer feedback; Reviewer 1 and 3's comments are addressed in issues II and III, Reviewer 2's comments are addressed in all issues, and Reviewer 4's comments are addressed in issues I and II.

Issue I: Irrelevant and/or Synthetic Use Cases (Reviewers 2 and 4)

We re-wrote Section §V-B to highlight examples from HPC and cloud workloads that would benefit from using Cudele. For HPC we focus on user home directories used for experiments and for the cloud we focus on the Hadoop and Spark runtimes (as suggested by Reviewer 2). We also added **Cudele Setup** headers in Section §V-B to show which subtree semantics accommodate each workload and to demonstrate how Cudele might be able to help in these scenarios. By showing these setups, we hope to demonstrate that different applications can use a file system in parallel with different consistency/durability semantics. Although we do not actually run the workloads on our prototype, hopefully our changes show how the use cases, and the synthetic benchmarks we used to represent them, are indeed relevant to parallel and distributed computing.

To align the paper to the themes of the conference, we changed Section §V-A and Figures 6a, 6b, and 6c to emphasize that decoupled namespaces facilitate client-driven parallelism, where clients detach subtrees form the global namespace and do operations in-parallel to their local disk. We also hope that changes to the use cases above motivate the need for robust distributed storage systems that can handle today's applications. So while the paper is storage-centric, we try to show that the workloads it supports are highly distributed and need a flexible solution like Cudele.

Issue II: Structure and Layout of Evaluation (All Reviewers)

To clarify the evaluation and eliminate some of the confusion due to our cross-referencing, we:

- connected performance gains from the abstract and Section §I to Section §V to clarify speedup/slowdown comparisons and to show how results are derived; we also re-labeled all graphs with the same metric (throughput slowdown/speedup) and aligned Section §V with the graphs in Section §II.
- made experiments self-contained in Sections §II and §V by removing cross-references and adding baseline values so readers can calculate raw numbers from our speedup/slowdown graphs.
- removed "major takeaways" from Section §V and focused on insights into results by analyzing raw numbers in comparison to hardware speeds.
- removed future work from Section §I to make the contributions explicit. We do not attempt to validate or prove any of the statements about the benefits of changing consistency and durability properties of a subtree dynamically, such as saving resource provisioning costs.
- clarified confusing terminology in Section §I: administrator refers to a person that configures subtrees and client refers to a storage client or application that interacts with the metadata server. We avoid using the term "user" except when referring to end-users that interact with home directories in a file system or the Hadoop/Spark runtime itself.

Issue III: No Experimental Setup (Reviewers 1, 2, and 3)

We apologize for the omission of experimental setup and source code details. For the experimental setup, we describe the cluster setup (hardware, software, etc.) in Section §V. We have also made the infrastructure code available and added links after each figure to show exactly how experiments are run. This infrastructure code contains scripts to deploy the system, run experiments, and gather results. This process follows the Popper Convention, which aims to make research reproducible. We also made the source code available online and a link is provided in Section §V. Finally, we add details about which tools and code bases we modified in Section §IV to address Reviewer 1's questions about the framework implementation.

Additional Cosmetic Fixes

For Reviewer 1, we fixed user/client terminology in Section §III, clarified that the cost of dynamically changing semantics is future work in Sections §I and §III, and added source code pointers in Section §V. For Reviewer 2, we removed the major takeaway headings and cross-references in Section §V. For Reviewer 3, we quantified speedups with figure annotations in Section §V. For Reviewers 1 and 3, we added servers, network, and storage setups in Section §V. For Reviewer 4, we added descriptions to each use case in Section §V describing how Cudele would work with Spark.

Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace

Michael A. Sevilla, Ivo Jimenez, Noah Watkins, Jeff LeFevre
Peter Alvaro, Shel Finkelstein, Patrick Donnelly*, Carlos Maltzahn
University of California, Santa Cruz; *Red Hat, Inc.
{msevilla, ivo, jayhawk, jlefevre}@soe.ucsc.edu, {palvaro, shel, carlosm}@ucsc.edu, pdonnell@redhat.com

***Abstract*—HPC and data center scale application developers are abandoning POSIX IO because file system metadata synchronization and serialization overheads of providing strong consistency and durability are too costly – and often unnecessary – for their applications. Unfortunately, designing file systems with weaker consistency or durability semantics excludes applications that rely on stronger guarantees, forcing developers to re-write their applications or deploy them on a different system. We present a framework and API that lets administrators specify their consistency/durability requirements and dynamically assign them to subtrees in the same namespace, allowing administrators to optimize subtrees over time and space for different workloads. We show similar speedups to related work but more importantly, we show performance improvements when we custom fit subtree semantics to applications such as checkpoint-restart (91.7× speedup), user home directories (0.03 standard deviation from optimal), and users checking for partial results (2% overhead).**

I. Introduction

File system metadata services in HPC and large-scale data centers have scalability problems because common tasks, like checkpointing [1] or scanning the file system [2], contend for the same directories and inodes. Applications perform better with dedicated metadata servers [3], [4] but provisioning a metadata server for every client[1] is unreasonable. This problem is exacerbated by current hardware and software trends: parallel runtimes like Hadoop and Spark use storage systems to implement important scheduling and timing semantics and HPC architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to more simplified stacks with just a burst buffer and object store [5]. These types of trends put pressure on data access because more requests from different nodes end up hitting the same layers in parallel.

To address this, developers are relaxing the consistency and durability semantics in the file system because weaker guarantees are sufficient for their applications. For example, many HPC batch jobs do not need the strong consistency that the file system provides, so BatchFS [2] and DeltaFS [6] do more client-side processing and merge updates when the job is done. Developers in these domains are turning to these non-POSIX IO solutions because their applications are well-understood (*e.g.*, well-defined read/write phases, synchronization only needed during certain phases, workflows describing computation, etc.) and because these applications wreak havoc on file systems designed for general-purpose workloads (*e.g.*, checkpoint-restart's N:N and N:1 create patterns [1]).

[1]In this paper, "client" is a storage client or application that interacts with the metadata server, "administrator" is a system administrator that configures the storage, and "end-users" interact with the file system via home directories or runtimes.

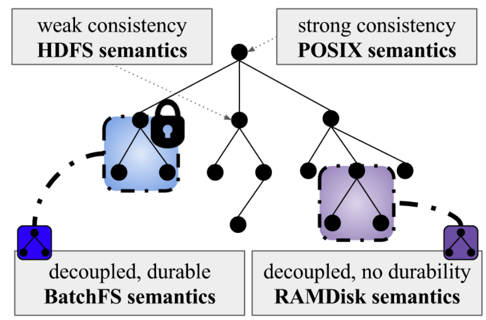


Fig. 1: Illustration of subtrees with different semantics co-existing in a global namespace. For performance, clients relax consistency/durability on their subtree (*e.g.*, HDFS) or decouple the subtree and move it locally (*e.g.*, BatchFS, RAMDisk).

One popular approach for relaxing consistency and durability is to "decouple the namespace", where clients lock the subtree they want exclusive access to as a way to tell the file system that the subtree is important or may cause resource contention in the near-future [2], [4], [6]–[8]. Then the file system can change its internal structure to optimize performance. For example, the file system could enter a mode where clients with decoupled directories perform operations locally and bulk merge their updates at completion. This delayed merge (*i.e.* a form of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of remote procedure calls (RPCs), synchronization, false sharing, and serialization. While the performance benefits of decoupling the namespace are obvious, applications that rely on the file system's guarantees must be deployed on an entirely different system or re-written to coordinate strong consistency/durability themselves.

To address this problem, we present an API and framework that lets developers dynamically control the consistency and durability guarantees for subtrees in the file system namespace. Figure 1 shows a potential setup in our proposed system where a single global namespace has subtrees for applications optimized with techniques from different state-of-the-art architectures. The BatchFS and RAMDisk subtrees are

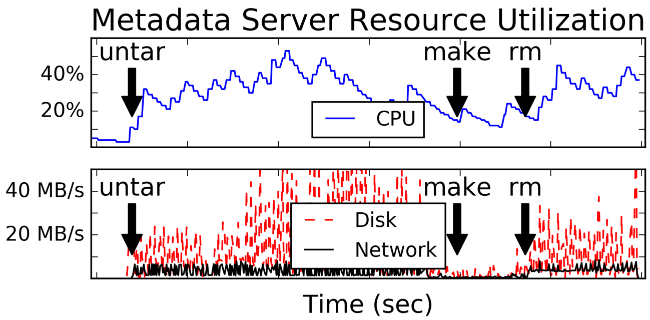


Fig. 2: [source] For the CephFS metadata server, create-heavy workloads (*e.g.*, `untar`) incur the highest disk, network, and CPU utilization because of consistency/durability demands.

decoupled from the global namespace and have similar consistency/durability behavior to those systems; the HDFS subtree has weaker than strong consistency because it lets clients read files opened for writing [9], which means that not all updates are immediately seen by all clients; and the POSIX IO subtree retains the rigidity of POSIX IO's strong consistency. Subtrees without policies inherit the consistency/durability semantics of the parent and future work will examine embeddable or inheritable policies.

Our prototype system, Cudele, achieves this by exposing "mechanisms" that administrators combine to specify their preferred semantics. Cudele supports 3 forms of consistency (invisible, weak, and strong) and 3 degrees of durability (none, local, and global) giving the administrator a wide range of policies and optimizations that can be custom fit to an application. We make the following contributions:

1) A framework/API for assigning consistency/durability policies to subtrees in a global namespace; this lets administrators navigate trade-offs of different metadata protocols on the same storage system.
2) We show that letting different semantics co-exist in a global namespace scales further and performs better than systems that use one strategy.
3) A prototype that lets administrators custom fit subtrees to applications dynamically.

The results in this paper confirm the assertions of "clean-slate" research of decoupled namespaces; specifically that these techniques drastically improve performance. We go a step further by quantifying the costs of traditional file system approaches to maintaining consistency (3.37× slowdown) and durability (2.4× slowdown). In our prototype, we also show the benefits of assigning subtree semantics to certain applications such as checkpoint-restart (91.7× speedup if consistency is fully relaxed), user home directories (within a 0.03 standard deviation from optimal), and end-users checking for partial results (only a 2% overhead). We use Ceph as a prototyping platform because it is used in cloud-based and data center systems and has a presence in HPC [10].

II. POSIX IO Overheads

In our examination of the overheads of POSIX IO we benchmark and analyze CephFS, the file system that uses Ceph's object store (RADOS) to store its data/metadata and a metadata server cluster to service client requests more quickly. During this process we discovered, based on the analysis and breakdown of costs, that durability and consistency have high overhead but we urge the reader to keep in mind that this file system is an implementation of one set of design decisions and our goal here is to highlight the effect that those decisions have on performance. At the end of each subsection we compare the approach to "decoupled namespaces", the technique in related work that detaches subtrees from the global namespace to relax consistency/durability guarantees.

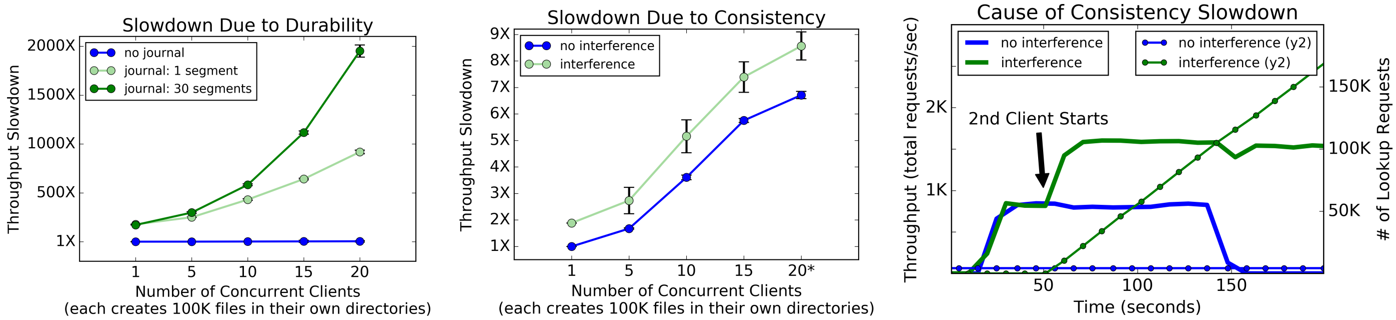
We use a create-heavy workload for this study because it has high resource utilization, as shown by the trace of compiling the Linux kernel in a CephFS mount in Figure 2. The `untar` phase, which is characterized by many creates, has the highest resource usage (combined CPU, network, and disk) on the metadata server because of the number of RPCs needed for consistency and durability.

A. Durability

While durability is not specified by POSIX IO, administrators expect that files they create or modify survive failures. We define three types of durability: global, local, and none. Global durability means that the client or server can fail at any time and metadata will not be lost because it is "safe" (*i.e.* striped or replicated across a cluster). Local durability means that metadata can be lost if the client or server stays down after a failure. None means that metadata is volatile and that the system provides no guarantees when clients or servers fail. None is different than local durability because regardless of the type of failure, metadata will be lost when components die in a None configuration.

CephFS Design: A journal of metadata updates that streams into the resilient object store. Similar to LFS [11] and WAFL [12] the metadata journal is designed to be large (on the order of MBs) which ensures (1) sequential writes into the object store and (2) the ability for daemons to trim redundant or irrelevant journal entries. The journal is striped over objects where multiple journal updates can reside on the same object. There are two tunables, related to groups of journal events called segments, for controlling the journal: the segment size and the dispatch size (*i.e.* the number of segments that can be dispatched at once). Unless the journal saturates memory or CPU resources, larger values for these tunables result in better performance.

In addition to the metadata journal, CephFS also represents metadata in RADOS as a metadata store, where directories and their file inodes are stored as objects. The metadata server applies the updates in the journal to the metadata store when the journal reaches a certain size. The metadata store is optimized for recovery (*i.e.* reading) while the metadata journal is write-optimized.



(a) [source] effect of journal (b) [source] interference hurts variability (c) [source] interference increases RPCs

Fig. 3: The durability and strong consistency slowdown of the existing CephFS implementation increases as the number of clients scales. Results in (a) and (b) are normalized to 1 client that creates 100K files in isolation. (a) shows the effect of journaling metadata updates; "segment(s)" is the number of journal segments dispatched to disk at once. (b) shows the slowdown when another client interferes by creating files in all directories and (c) highlights the cause: when another client interferes, capabilities are revoked and metadata servers do more work.

Figure 3a shows the effect of journaling with different dispatch sizes, normalized to 1 client that creates 100K files with journaling off (about 654 creates/sec). In this case a dispatch size of 30 degrades performance the most because the metadata server is overloaded with requests and cannot spare cycles to manage concurrent segments. Tuning and parameter sweeps show that a dispatch size of 10 is the worst and that larger sizes approach a dispatch size of 1; for all future journal experiments we use a dispatch size of 40 which is a more realistic configuration. Although the "no journal" curve appears flat, it is actually a slowdown of about $0.3\times$ per concurrent client; this slowdown is a result of the peak throughput of a single metadata server, which we found to be about 3000 operations per second. The trade-off for better performance is memory consumption because a larger dispatch size uses more space for buffering.

Comparison to decoupled namespaces: For BatchFS, if a client fails when it is writing to the local log-structured merge tree (implemented as an SSTable [13]) then unwritten metadata operations are lost. For DeltaFS, if the client fails then, on restart, the computation does the work again – since the snapshots of the namespace are never globally consistent and there is no ground truth. On the server side, BatchFS and DeltaFS use IndexFS [4]. IndexFS writes metadata to SSTables, which initially reside in memory but are later flushed to the underlying distributed file system.

B. Strong Consistency

Access to metadata in a POSIX IO-compliant file system is strongly consistent, so reads and writes to the same inode or directory are globally ordered. The synchronization and serialization machinery needed to ensure that all clients see the same state has high overhead.

CephFS Design: Capabilities keep metadata strongly consistent. To reduce the number of RPCs needed for consistency, clients can obtain capabilities for reading and writing inodes, as well as caching reads, buffering writes, changing the file size, and performing lazy IO. To keep track of the read caching and write buffering capabilities, the clients and metadata servers agree on the state of each inode using an inode cache. If a client has the directory inode cached it can do metadata writes (*e.g.*, create) with a single RPC. If the client is not caching the directory inode then it must do an extra RPC to determine if the file exists. Unless the client immediately reads all the inodes in the cache (*i.e.* `ls -alR`), the inode cache is less useful for create-heavy workloads.

Figure 3b shows the slowdown of maintaining strong consistency when scaling the number of clients. We plot the slowdown of the slowest client, normalized to 1 client that creates 100K files (about 513 creates/sec because the journal is turned back on). For the "interference" curve, each client creates files in private directories and at 30 seconds we launch another process that creates files in those directories. 20 clients has an asterisk because the maximum number of clients the metadata server can handle for this metadata-intensive workload is actually 18; at higher client load, the metadata server complains about laggy and unresponsive requests.

The cause for this slowdown is shown in Figure 3c. The colors show the behavior of the client for two different runs. If only one client is creating files in a directory ("no interference" curve on $y1$ axis) then that client can lookup the existence of new files locally before issuing a create request to the metadata server. If another client starts creating files in the same directory then the directory inode transitions out of read caching and the first client must send `lookup()`s to the metadata server ("interference" curve on $y2$ axis). These extra requests increase the throughput of the "interference" curve on the $y1$ axis because the metadata server can handle the extra load but performance suffers.

Comparison to decoupled namespaces: Decoupled namespaces merge batches of metadata operations into the global namespaces when the job completes. In BatchFS, the merge is delayed by the application using an API to switch between asynchronous and synchronous mode. The merge itself is explicitly managed by the application but future work looks at more automated methodologies. In DeltaFS, snapshots of

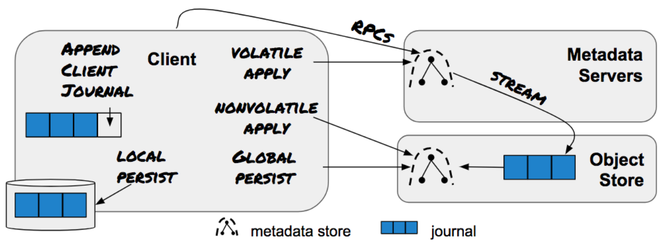


Fig. 4: Illustration of the **mechanisms** used by applications to build consistency/durability semantics. Descriptions are provided by the underlined words in Section §III-A.

C → D ↓	invisible	weak	strong
none	append client journal	append client journal +volatile apply	RPCs
local	append client journal +local persist	append client journal +local persist +volatile apply	RPCs +local persist
global	append client journal +global persist	append client journal +global persist +volatile apply	RPCs +stream

TABLE I: Users can explore the consistency (C) and durability (D) spectrum by composing Cudele mechanisms.

the metadata subtrees stays on the client machines; there is no ground truth and consistent namespaces are constructed and resolved at application read time or when a 3rd party system (*e.g.*, middleware, scheduler, etc.) needs a view of the metadata. As a result, all the overheads of maintaining consistency that we showed above are delayed until the merge phase.

III. Methodology: Global Namespace, Subtree Consistency/Durability

In this section we describe our API and framework that lets administrators assign consistency and durability semantics to subtrees in the global namespace. A **mechanism** is an abstraction and basic building block for constructing consistency and durability guarantees. The administrator composes these mechanisms together to construct **policies**. These policies are assigned to subtrees and they dictate how the file system should handle operations within that subtree. Below, we describe the mechanisms (which are underlined), the policies, and the API for assigning policies to subtrees.

A. Mechanisms: Building Guarantees

Figure 4 shows the mechanisms (labeled arrows) in Cudele and which daemon(s) they are performed by. Decoupled clients use the Append Client Journal mechanism to append metadata updates to a local, in-memory journal. Clients do not need to check for consistency when writing events and the metadata server blindly applies the updates because it assumes the events were already checked for consistency. The trade-off here is fast performance; it is a dangerous approach but could be implemented safely if the clients or metadata server are configured to check the validity of events before writing them. Once the job is complete, the system calls mechanisms to achieve the desired consistency/durability semantics. Cudele provides a library for clients to link into and all operations are performed by the client.

1) Mechanisms Used for Consistency: RPCs send remote procedure calls for every metadata operation from the client to the metadata server, assuming the request cannot be satisfied by the inode cache. This mechanism is part of the default CephFS implementation and is the strongest form of consistency because clients see metadata updates right away. Nonvolatile Apply replays the client's in-memory journal into the object store and restarts the metadata servers. When the metadata servers re-initialize, they notice new journal updates in the object store and replay the events onto their in-memory metadata stores. Volatile Apply takes the client's in-memory journal on the client and applies the updates directly to the in-memory metadata store maintained by the metadata servers. We say volatile because – in exchange for peak performance – Cudele makes no consistency or durability guarantees while Volatile Apply is executing. If a concurrent update from a client occurs there is no rule for resolving conflicts and if the client or metadata server crashes there may be no way to recover.

The biggest difference between Volatile Apply and Nonvolatile Apply is the medium they use to communicate. Volatile Apply applies updates directly to the metadata servers' metadata store while Nonvolatile Apply uses the object store to communicate the journal of updates from the client to the metadata servers. Nonvolatile Apply is safer but has a large performance overhead because objects in the metadata store need to be read from and written back to the object store.

2) Mechanisms Used for Durability: Stream, the default setting in CephFS, saves a journal of metadata updates in the object store. Using existing configuration settings we can turn Stream on and off. For Local Persist, clients write serialized log events to a file on local disk and for Global Persist, clients push the journal into the object store. The overheads for both Local Persist and Global Persist is the write bandwidth of the local disk and object store, respectively. These persist mechanisms are part of the library that links into the client.

B. Defining Policies in Cudele

The spectrum of consistency and durability guarantees that administrators can construct is shown in Table I. The columns are the different consistency semantics and the rows cover the spectrum of durability guarantees. For consistency: "invisible" means the system does not handle merging updates into a global namespace and it is assumed that middleware or the application manages consistency lazily; "weak" merges updates at some time in the future (*e.g.*, when the system has time, when the number of updates reaches a certain threshold, when the client is done writing, etc.); and updates in "strong" consistency are seen immediately by all clients. For durability, "none" means that updates are volatile and will be lost on a failure. Stronger guarantees are made with "local", which means updates will be retained if the client node recovers and reads the updates from local storage, and "global", where all updates are always recoverable.

Existing, state-of-the-art systems in HPC can be represented by the cells in Table I. POSIX IO-compliant systems like CephFS and IndexFS have global consistency and durability[2]; DeltaFS uses "invisible" consistency and "local" durability and BatchFS uses "weak" consistency and "local" durability. These systems have other features that could push them into different semantics but we assign labels here based on the points emphasized in the papers. To compose the mechanisms administrators inject which mechanisms to run and which to use in parallel using a domain specific language. Although we can achieve all permutations of the different guarantees in Table I, not all of them make sense. For example, it makes little sense to do `append client journal+RPCs` since both mechanisms do the same thing or `stream+local persist` since "global" durability is stronger and has more overhead than "local" durability. The cost of each mechanism and the semantics described above are quantified in Sections §V-A.

In our prototype, the consistency and durability properties in Table I are not guaranteed until all mechanisms in the cell are complete. The compositions should be considered atomic and there are no guarantees while transitioning between policies. For example, updates are not deemed to have "global" durability until they are safely saved in the object store. If a failure occurs during Global Persist or if we inject a new policy that changes a subtree from Local Persist to Global Persist, Cudele makes no guarantee until the mechanisms are complete. Despite this, production systems that use Cudele should state up-front what the transition guarantees are for subtrees. This is not a limitation of our approach; it just lead to the simplest implementation.

C. Cudele Namespace API

Users control consistency and durability for subtrees by contacting a daemon in the system called a monitor, which manages cluster state changes. Users present a directory path and a policies configuration that gets distributed and versioned by the monitor to all daemons in the system. For example, (msevilla/mydir, policies.yml) would decouple the path "msevilla/mydir" and would apply the policies in "policies.yml".

The policies file supports the following parameters (default values are in parenthesis): which consistency model to use (`RPCs`), which durability model to use (`stream`), number of inodes to provision to the decoupled namespace (100), and which interfere policy to use, *i.e.* how to handle a request from another client targeted at this subtree (`allow`). The "Consistency" and "Durability" parameters are compositions of mechanisms; they can be serialized (+) or run in parallel (||). "Allocated Inodes" is a way for the application to specify how many files it intends to create. It is a contract so that the file system can provision enough resources for the incumbent merge and so it can give valid inodes to other clients. The inodes can be used anywhere within the decoupled namespace (*i.e.* at any depth in the subtree). "Interfere Policy" has two settings: `block` and `allow`. For `block`, any requests to this part of the namespace returns with "Device is busy", which will spare the metadata server from wasting resources for updates that may get overwritten. If the application does not mind losing updates, for example it wants approximations for results that take too long to compute, it can select `allow`. In this case, metadata from the interfering client will be written and the computation from the decoupled namespace will take priority at merge time because the results are more accurate. Given these default values decoupling the namespace with an empty policies file would give the application 100 inodes but the subtree would behave like the existing CephFS implementation.

IV. Implementation

We use a programmable storage approach [14] to design Cudele; namely, we try to leverage components inside CephFS to inherit the robustness and correctness of the internal subsystems. Using this "dirty-slate" approach, we only had to implement 4 of the 6 mechanisms from Figure 4 and just 1 required changes to the underlying storage system itself. In this section, we first describe a CephFS internal subsystem or component and then we show how we use it in Cudele.

A. Metadata Store

In CephFS, the metadata store is a data structure that represents the file system namespace. This data structure is stored in two places: in memory (*i.e.* in the collective memory of the metadata server cluster) and as objects in the object store. In the object store, directories and their inodes are stored together in objects to improve the performance of scans. The metadata store data structure is structured as a tree of directory fragments making it easier to read and traverse. **In Cudele,** the RPCs mechanism uses the in-memory metadata store to service requests. Using code designed for recovery, Volatile Apply and Nonvolatile Apply replay updates onto the metadata store in memory and in the object store, respectively.

B. Journal Format and Journal Tool

The journal is the second way that CephFS represents the file system namespace; it is a log of metadata updates that can materialize the namespace when the updates are replayed onto the metadata store. The journal is a "pile system"; writes are fast but reads are slow because state must be reconstructed. Specifically, reads are slow because there is more state to read, it is unorganized, and many of the updates may be redundant. **In Cudele,** the journal format is used by Stream, Append Client Journal, Local Persist, and Global Persist. Stream is the default implementation for achieving global durability in CephFS but the rest of the mechanisms are implemented by writing with the journal format. By writing with the same format, the metadata servers can read and use the recovery code to materialize the updates from a client's decoupled

[2] IndexFS also has bulk merge which is a form of "weak consistency"

namespace (*i.e.* merge). These implementations required no changes to CephFS because the metadata servers know how to read the events the library is writing. By re-using the journal subsystem to implement the namespace decoupling, Cudele leverages the write/read optimized data structures, the formats for persisting events (similar to TableFS's SSTables [13]), and the functions for replaying events onto the internal namespace data structures.

The journal tool is used for disaster recovery and lets administrators view and modify the journal. It can read the journal, export the journal as a file, erase events, and apply updates to the metadata store. To apply journal updates to the metadata store, the journal tool reads the journal from object store objects and replays the updates on the metadata store in the object store. **In Cudele,** the external library the clients link into is based on the journal tool. It already had functions for importing, exporting, and modifying the updates in the journal so we re-purposed that code to implement Append Client Journal, Volatile Apply, and Nonvolatile Apply.

C. Inode Cache and Large Inodes

In CephFS, the inode cache reduces the number of RPCs between clients and metadata servers. Without contention clients can resolve metadata reads locally thus reducing the number of operations (*e.g.*, `lookup()`s). For example, if a client or metadata server is not caching the directory inode, all creates within that directory will result in a lookup and a create request. If the directory inode is cached then only the create needs to be sent. The size of the inode cache is configurable so as not to saturate the memory on the metadata server – inodes in CephFS are about 1400 bytes [15]. The inode cache has code for manipulating inode numbers, such as pre-allocating inodes to clients. **In Cudele,** Nonvolatile Apply uses the internal inode cache code to allocate inodes to clients that decouple parts of the namespace and to skip inodes used by the client at merge time.

In CephFS, inodes already store policies, like how the file is striped across the object store or for managing subtrees for load balancing. These policies adhere to logical partitionings of metadata or data, like Ceph pools and file system namespace subtrees. To implement this, the namespace data structure has the ability to recursively apply policies to subtrees and to isolate subtrees from each other. **In Cudele,** the large inodes also store consistency and durability policies. This approach uses the File Type interface from the Malacology programmable storage system [14] and it tells clients how to access the underlying metadata. The underlying implementation stores executable code in the inode that calls the different Cudele mechanisms. Of course, there are many security and access control aspects of this approach but that is beyond the scope of this paper.

V. Evaluation

Cudele lets administrators construct consistency/durability guarantees using well-established research techniques from other systems; so instead of evaluating the scalability and performance of the techniques themselves against other file systems, we show that (1) the mechanisms we propose are useful for constructing semantics used by real systems and (2) the techniques can work side-by-side in the same namespace for common use cases. These techniques have already proven to be effective and scalable in systems that specialize an entire namespace according to a single optimization strategy (*e.g.*, Lustre, BatchFS, DeltaFS).

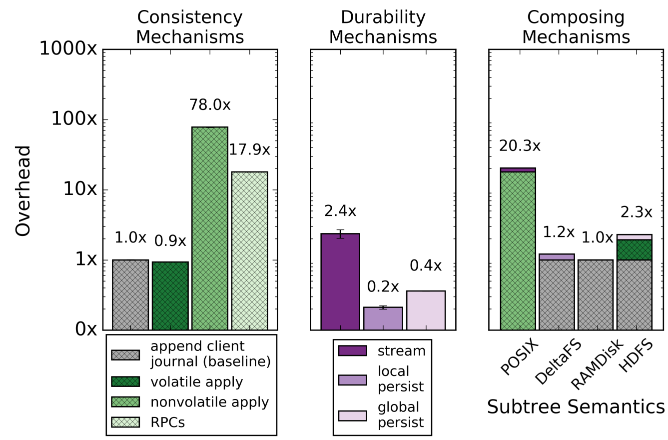


Fig. 5: [source] Overhead of processing 100K create events for each mechanism in Figure 4, normalized to the runtime of writing events to client memory. The far right graph shows the overhead of building semantics of real world systems.

We graph standard deviations for three runs (sometimes error bars are too small to see) and normalize results to make our results more generally applicable to different hardware. We use a CloudLab cluster of 34 nodes connected with 10Gbit ethernet, each with 16 2.4 GHz CPUs, 128GB RAM, and 400GB SSDs. Each node uses Ubuntu 14.04 and we develop on Ceph's Jewel release, version 10.2.1, which was released in May 2016. We use 1 monitor daemon, 3 object storage daemons, 1 metadata server daemon, and up to 20 clients. We scope the evaluation to one metadata server and scale the number of parallel clients each doing 100K operations because 100K is the maximum recommended size of a directory in CephFS. We scale to 20 clients because, as shown in Section §II, 20 clients is enough to saturate the resources of a single metadata server. This type of analysis shows the capacity and performance of a metadata server with superior metadata protocols, which should be used to inform metadata distribution across a cluster. Load balancing across a cluster of metadata servers with partitioning and replication can be explored with something like Mantle [3]. To make our results reproducible, this paper adheres to The Popper Convention [16] so experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure. The source code for Cudele is available on a branch [17] of our Ceph fork.

A. Microbenchmarks

We measure the overhead of each Cudele mechanism by having 1 client create 100K files in a directory for various subtree configurations. Figure 5 shows the time that it takes each Cudele mechanism to process all metadata events. Results are normalized to the time it takes to write updates to the client's in-memory journal (*i.e.* the Append Client Journal mechanism), which is about 11K creates/sec. The first graph groups the consistency mechanisms, the second groups the durability mechanisms, and the third has compositions representing real-world systems.

Poorly Scaling Data Structures: Despite doing the same amount of work, mechanisms that rely on poorly scaling data structures have large slowdowns. For example, RPCs has a $17.9\times$ slowdown because this technique relies on internal directory data structures, which is a well-known problem [4]. Other mechanisms that write events to a journal experience a much less drastic slowdown because the journal data structure does not need to be scanned for every operation. Events are written to the end of the journal without even checking the validity (*e.g.*, if the file already exists for a create), which is another form of relaxed consistency because the file system assumes the application has resolved conflicting updates in a different way.

Overhead of Consistency: RPCs is $19.9\times$ slower than Volatile Apply because sending individual metadata updates over the network is costly. While RPCs sends a request for every file create, Volatile Apply writes directly to the in-memory data structures in the metadata server. While communicating the decoupled namespace directly to the metadata server with Volatile Apply is faster, communicating through the object store with Nonvolatile Apply is $78\times$ slower. Nonvolatile Apply was not implemented as part of Cudele – it was a debugging and recovery tool packaged with CephFS. It works by iterating over the updates in the journal and pulling all objects that *may* be affected by the update. This means that two objects are repeatedly pulled, updated, and pushed: the object that houses the experiment directory and the object that contains the root directory (*i.e.* `/`). Nonvolatile Apply ($78\times$) and composing Volatile Apply + Global Persist ($1.3\times$) end up with the same final metadata state but using Nonvolatile Apply is clearly inferior.

Parallelism of the Object Store: Stream, which is an approximation (journal on minus journal off), has the highest slowdown at $2.4\times$ because the overhead of maintaining and streaming the journal is incurred by the metadata server. Comparing Local and Global Persist demonstrates the bandwidth advantages of storing the journal in a distributed object store. The Global Persist performance is only $0.2\times$ slower than Local Persist because Global Persist is leveraging the collective bandwidth of the disks in the cluster. This benefit comes from the object store itself but should be acknowledged when making decisions for the application; the bandwidth of the object store can help mitigate the overheads of globally persisting metadata updates. The storage per journal update is about 2.5KB. So the storage footprint scales linearly with the number of metadata creates and suggests that updates for a million updates in a single journal would be 2.38GB

Composing Mechanisms: The graph on the right of Figure 5 shows how applications can compose mechanisms together to get the consistency/durability guarantees they need in a global namespace. We label the x-axis with systems that employ these semantics, as described in Figure 1. We make no guarantees during execution of the mechanisms or when transitioning semantics – the semantics are guaranteed *once the mechanism completes*. So if servers fail during a mechanism, metadata or data may be lost. This graph shows how we can build application-specific subtrees by composing mechanisms and the performance of coupling well-established techniques to specific applications over the same file system.

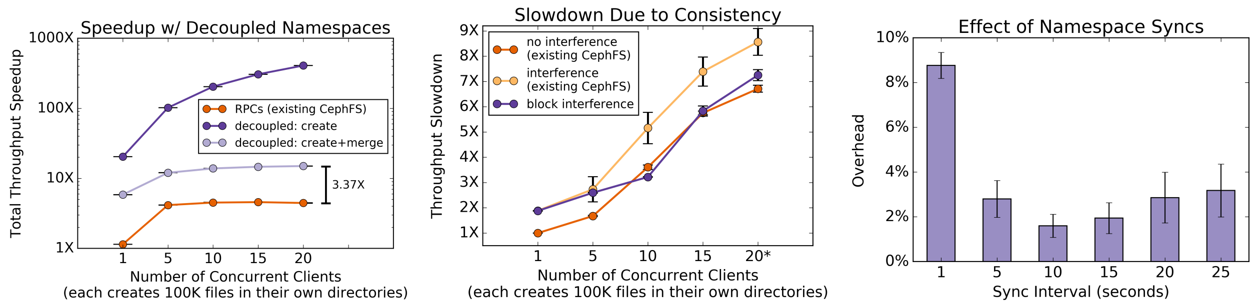
B. Use Cases

Next we present three uses cases: creates in the same directory, interfering clients, and read while writing. For each use case, we provide motivation from HPC and cloud workloads; specifically, we look at users using the file system in parallel to run large-scale experiments in HPC and parallel runtimes that use the file system, such as Hadoop and Spark. The synthetic benchmarks model scenarios where these workloads co-exist in a global namespace and we provide insight into how the workload benefits from Cudele.

1) Creates in the Same Directory: We start with clients creating files in private directories because this workload is heavily studied in HPC [2]–[4], [18], [19], mostly due to checkpoint-restart [1]. A more familiar example is uncompressing an archive (*e.g.*, `tar xzf`), where the file system services a flash crowd of creates across all directories as shown in Figure 2. But the workload also appears in cloud workloads: Hadoop/Spark use the file system to assign work units to workers and the performance is proportional to the open/create throughput of the underlying file system [20]–[22]; Big Data Benchmark jobs examined in [23] have on the order of 15,000 file opens or creates just to start a single Spark query and the Lustre system they tested on did not handle creates well, showing up to a $24\times$ slowdown compared to other metadata operations. Common approaches to solve these types of bottlenecks is to change the application behavior or to design a new file system, like BatchFS or DeltaFS, that uses one set of metadata optimizations for the entire namespace.

Cudele setup: accommodate these workloads in the global namespace by configuring three subtrees with the following semantics: one with strong consistency and global durability (RPCs), one with invisible consistency and local durability (decoupled: create), and one with weak consistency and local durability (decoupled: create + merge).

In Figure 6a we scale the number of clients each doing 100K file creates in their own directories. Results are normalized to 1 client that creates 100K files using RPCs (about 549 creates/sec). As opposed to earlier graphs in Section §II that plotted the throughput of the slowest client, Figure 6a plots the throughput of the total job (*i.e.* from the perspective of



(a) [source] parallel creates on clients (b) [source] block interference (c) [source] syncing to global namespace

Fig. 6: Cudele performance. (a) shows the speedup of decoupled namespaces over RPCs; `create` is the throughput of clients creating files in-parallel and writing updates locally; `create+merge` includes the time to merge updates at the metadata server. Decoupled namespaces scale better than RPCs because there are less messages and consistency/durability code paths are bypassed. (b) shows how the allow/block API isolates directories from interfering clients. (c) is the slowdown of a single client syncing updates to the global namespace. The inflection point is the trade-off of frequent updates vs. larger journal files.

the metadata server). Plotting this way is easier to understand because of how we normalize but the speedups over the RPC approach are the same, whether we look at the slowest client or not.

When the metadata server is operating at peak efficiency at 20 clients, the performance of the "RPCs" and "decoupled: merge + create" subtrees is bottlenecked by the metadata server processing power, so the curves flatten out at a slowdown of $4.5\times$ and $15\times$, respectively. On the other hand, the "decoupled: create" subtree performance scales linearly with the number of concurrent clients because clients operate in parallel and write updates locally. At 20 clients, we observe a $91.7\times$ speedup for "decoupled: create" over RPCs.

"Decoupled: merge + create" outperforms "RPCs" by $3.37\times$ because "decoupled: merge + create" uses a relaxed form of consistency and leverages bulk updates just like DeltaFS [6]. Decoupled namespaces (1) place no restrictions on the validity of metadata inserted into the journal (*e.g.*, never checking for the existence of files before creating files), (2) avoid touching poorly scaling data structures, and (3) allow clients to batch events into bulk updates. Had we implemented the client to send updates one at a time and to include `lookup()` commands before `open()` requests, we would have seen performance closer to the "RPC" subtree. The "decoupled: merge + create" curve is also pessimistic because it models a scenario in which all client journals arrive at the same time. So for the 20 clients data point, we are measuring the operations per second for 20 client journals that land on the metadata server at the same time. Had we added infrastructure to overlay journal arrivals or time client sync intervals, we could have scaled more closely to "decoupled: create".

2) Interfering Clients: Next we show how Cudele can be programmed to block interfering clients, which lets applications control isolation to get better and more reliable performance. Clients create 100K files in their own directories while another client interferes by creating 1000 files in each directory. The workload introduces false sharing and the metadata server revokes capabilities on directories touched by the interfering client. While HPC tries to avoid these situations with workflows [2], [6], it still happens in distributed file systems when users unintentionally access directories in a shared file system. In the cloud, Spark and Hadoop stacks use HDFS, which lets clients ignore this type of consistency completely by letting interfering clients read files opened for writing [9].

Cudele setup: enable global durability with Stream and strong consistency with RPCs to mirror the setup from the problem presented in Figure 3b. We configure one subtree with an interfere policy of "allow" and another subtree with "block" so `-EBUSY` is returned to interfering clients. The former is the default behavior in file systems and the latter isolates performance from interfering clients.

Figure 6b plots the overhead of the slowest client, normalized to 1 client that creates 100K files in isolation (about 513 creates/sec). "Interference" and "no interference" is the performance with and without an interfering client touching files in every directory, respectively. The goal is to explicitly isolate clients so that performance is similar to the "no interference" curve, which has lower slowdowns (on average, $1.42\times$ per client compared to $1.67\times$ per client for "interference") and less variability (on average, a standard deviation of 0.06 compared to 0.44 for "interference"). "Block interference" uses the Cudele API to block interfering clients and the slowdown ($1.34\times$ per client) and variability (0.09) look very similar to "no interference" for a larger number of clients. For smaller clusters the overhead to reject requests is more evident when the metadata server is underloaded so the slowdowns are similar to "interference". We conclude that administrators can block interfering clients to get the same performance as isolated scenarios but there is a non-negligible overhead for rejecting requests when the metadata server is not operating at peak efficiency.

3) Read while Writing: The final use case shows how the API gives administrators fine-grained control of the consistency semantics to support current practices and scientific workflows in HPC. Users often leverage the file system to check the progress of jobs using `ls` even though this operation is notoriously heavy-weight [24], [25]. The number of files or size of the files is indicative of the progress. This practice is not too different from cloud systems that use the file system to manage the progress of jobs; Spark/Hadoop writes to temporary files, renames them when complete, and creates a "DONE" file to indicate to the runtime that the task did not fail and should not be re-scheduled on another node. So the browser interface lets Hadoop/Spark users check progress by querying the file system and returning a % of job complete metric.

Cudele setup: in this scenario, Cudele end-users will not see the progress of decoupled namespaces since their updates are not globally visible. To provide the performance of decoupled namespaces and to help end-users judge the progress of their jobs, Cudele clients have a "namespace sync" that sends batches of updates back to the global namespace at regular intervals. We configure a subtree as a decoupled namespace with invisible consistency, local durability, and partial updates enabled.

Figure 6c shows the performance degradation of a single client writing 1 million updates to the decoupled namespace and pausing to send updates to the metadata server. We scale the namespace sync interval to show the trade-off of frequently pausing or writing large logs of updates. We use an idle core to log the updates and to do the network transfer. The client only pauses to fork off a background process, which is expensive as the address space needs to be copied. The alternative is to pause the client completely and write the update to disk but since this implementation is limited by the speed of the disk, we choose the memory-to-memory copy of the fork approach.

As expected, syncing namespace updates too frequently has the highest overhead (up to 9% overhead if done every second). The optimal sync interval for performance is 10 seconds, which only incurs 2% overhead, because larger intervals must write more updates to disk and network. For the 25 second interval, the client only pauses 3-4 times but each sync writes about 278 thousand updates at once, which is a journal of size 678MB.

VI. Related Work

The bottlenecks associated with accessing POSIX IO file system metadata are not limited to HPC workloads and the same challenges that plagued these systems for years are finding their way into the cloud. Workloads that deal with many small files (*e.g.*, log processing and database queries [26]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [27]), are subject to the scalability of the metadata service. The biggest challenge is that whenever a file is touched the client must access the file's metadata and maintaining a file system namespace imposes small, frequent accesses on the underlying storage system [28]. Unfortunately, scaling file system metadata is a well-known problem and solutions for scaling data IO do not work for metadata IO [28]–[31].

POSIX IO workloads require strong consistency and many file systems improve performance by reducing the number of remote calls per operation (*i.e.* RPC amplification). As discussed in the previous section, caching with leases and replication are popular approaches to reducing the overheads of path traversals but their performance is subject to cache locality and the amount of available resources, respectively; for random workloads larger than the cache extra RPCs hurt performance [4], [18] and for write heavy workloads with more resources the RPCs for invalidations are harmful. Another approach to reducing RPCs is to use leases or capabilities.

High performance computing has unique requirements for file systems (*e.g.*, fast creates) and well-defined workloads (*e.g.*, workflows) that make relaxing POSIX IO sensible. BatchFS assumes the application coordinates accesses to the namespace, so the clients can batch local operations and merge with a global namespace image lazily. Similarly, DeltaFS eliminates RPC traffic using subtree snapshots for non-conflicting workloads and middleware for conflicting workloads. MarFS gives users the ability to lock "project directories" and allocate GPFS clusters for demanding metadata workloads. TwoTiers eliminates high-latencies by storing metadata in a flash tier; applications lock the namespace so that metadata can be accessed more quickly. Unfortunately, decoupling the namespace has costs: (1) merging metadata state back into the global namespace is slow; (2) failures are local to the failing node; and (3) the systems are not backwards compatible.

For (1), state-of-the-art systems manage consistency in non-traditional ways: IndexFS maintains the global namespace but blocks operations from other clients until the first client drops the lease, BatchFS does operations on a snapshot of the namespace and merges batches of operations into the global namespace, and DeltaFS never merges back into the global namespace. The merging for BatchFS is done by an auxiliary metadata server running on the client and conflicts are resolved by the application. Although DeltaFS never explicitly merges, applications needing some degree of ground truth can either manage consistency themselves on a read or add a bolt-on service to manage the consistency. For (2), if the client fails and stays down, all metadata operations on the decoupled namespace are lost. If the client recovers, the on-disk structures (for BatchFS and DeltaFS this is the SSTables used in TableFS) can be recovered. In other words, the clients have state that cannot be recovered if the node stays failed and any progress will be lost. This scenario is a disaster for checkpoint-restart where missed cycles may cause the checkpoint to bleed over into computation time. For (3), decoupled namespace approaches sacrifice POSIX IO going as far as requiring the application to link against the systems they want to talk to. In today's world of software defined caching, this can be a problem for large data centers with many types and tiers of storage. Despite well-known performance problems POSIX IO and REST are the dominant APIs for data transfer.

VII. Conclusion and Future Work

Relaxing consistency/durability semantics in file systems is a double-edged sword. While the technique performs and scales better, it alienates applications that rely on strong consistency and durability. Cudele lets administrators assign consistency/durability guarantees to subtrees in the global namespace, resulting in custom fit semantics for applications. We show how applications can co-exist and perform well in a global namespace and our prototype enables studies that adjust these semantics over *time and space*, where subtrees can change without ever moving the data they reference.

Cudele prompts many avenues for future work. First is to co-locate HPC workflows with real highly parallel runtimes from the cloud in the same namespace. This setup would show how Cudele reasonably incorporates both programming models (client-driven parallelism and user-defined workflows) at the same time and should show large performance gains. Second is dynamically changing semantics of a subtree from stronger to weaker guarantees (or vice versa). This reduces data movement across storage cluster and file system boundaries so the results of a Hadoop job do not need to be migrated into CephFS for other processing; instead the administrator can change the semantics of the HDFS subtree into a CephFS subtree, which may cause metadata/data movement to ensure strong consistency. Third is embeddable policies, where child subtrees have specialized features but still maintain guarantees of their parent subtrees. For example, a RAMDisk subtree is POSIX IO-compliant but relaxes durability constraints, so it can reside under a POSIX IO subtree alongside a globally durable subtree.

References

[1] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09.

[2] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers," in *Proceedings of the Workshop on Parallel Data Storage*, ser. PDSW '14.

[3] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg, "Mantle: A Programmable Metadata Load Balancer for the Ceph File System," in *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '15.

[4] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[5] J. Bent, B. Settlemyer, and G. Grider, "Serving Data to the Lunatic Fringe."

[6] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers," in *Proceedings of the Workshop on Parallel Data Storage*, ser. PDSW '15, 2015.

[7] G. Grider, D. Montoya, H.-b. Chen, B. Kettering, J. Inman, C. DeJager, A. Torrez, K. Lamb, C. Hoffman, D. Bonnie, R. Croonenberg, M. Broomfield, S. Leffler, P. Fields, J. Kuehn, and J. Bent, "MarFS - A Scalable Near-Posix Metadata File System with Cloud Based Object Backend," in *Work-in-Progress at Proceedings of the Workshop on Parallel Data Storage*, ser. PDSW'15, November 2015.

[8] S. Faibish, J. Bent, U. Gupta, D. Ting, and P. Tzelnic, "Slides: 2 tier storage architecture." [Online]. Available: http://www.pdl.cmu.edu/SDI/2015/slides/Faibish-CMU-PDL-Spring-2015-final.pdf

[9] J. D. Kamal Hakimzadeh, Hooman Peiro Sajjad, "Scaling HDFS with a Strongly Consistent Relational Model for Metadata," in *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, ser. DAIS '14.

[10] F. Wang, M. Nelson, S. Oral, S. Atchley, S. Weil, B. W. Settlemyer, B. Caldwell, and J. Hill, "Performance and Scalability Evaluation of the Ceph Parallel File System," in *Proceedings of the Workshop on Parallel Data Storage Workshop*, ser. PDSW '13.

[11] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," in *ACM Transactions on Computer Systems*, '92.

[12] D. Hitz, J. Lau, and M. Malcolm, "File system design for an nfs file server appliance," in *Proceedings of the USENIX Technical Conference*, ser. WTEC '94.

[13] K. Ren and G. Gibson, "TABLEFS: Enhancing Metadata Efficiency in the Local File System," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC '13, 2013.

[14] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A Programmable Storage System," in *Proceedings of the European Conference on Computer Systems*, ser. Eurosys '17, Belgrade, Serbia.

[15] "Ceph Documentation," http://docs.ceph.com/docs/jewel/dev/mds_internals/data-structures/, December 2017.

[16] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Popper: Making Reproducible Systems Performance Evaluation Practical," UC Santa Cruz, Technical Report UCSC-SOE-16-10, May 2016.

[17] "Cudele Source Code," https://github.com/michaelsevilla/ceph/tree/cudele, December 2017.

[18] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '04.

[19] S. V. Patil and G. A. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files," in *Proceedings of the Conference on File and Storage Technologies*, ser. FAST '11.

[20] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems," in *Proceedings of the Symposium on Cloud Computing*, ser. SoCC '15.

[21] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics," *Proceedings of the VLDB Endowment*.

[22] K. o. V. Shvachko, "HDFS Scalability: The Limits to Growth."

[23] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC Systems," in *Proceedings of the 25th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16.

[24] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file Access in Parallel File Systems," in *Proceedings of the Symposium on Parallel and Distributed Processing*, ser. IPDPS '09.

[25] M. Eshel, R. Haskin, D. Hildebrand, M. Naik, F. Schmuck, and R. Tewari, "Panache: A Parallel File System Cache for Global File Access," in *Proceedings of the Conference on File and Storage Technologies*, ser. FAST '10.

[26] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data Warehousing and Analytics Infrastructure at Facebook," in *Proceedings of the SIGMOD International Conference on Management of Data*, ser. SIGMOD '10.

[27] K. McKusick and S. Quinlan, "GFS: Evolution on Fast-forward," *Commununications ACM*, vol. 53, no. 3, pp. 42–49, Mar. 2010.

[28] D. Roselli, J. R. Lorch, and T. E. Anderson, "A Comparison of File System Workloads," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC '00.

[29] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell, "Metadata Traces and Workload Models for Evaluating Big Storage Systems," in *Proceedings of the International Conference on Utility and Cloud Computing*, ser. UCC '12.

[30] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni, "Parallel I/O and the Metadata Wall," in *Proceedings of the Workshop on Parallel Data Storage*, ser. PDSW '11.

[31] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, ser. OSDI '06.