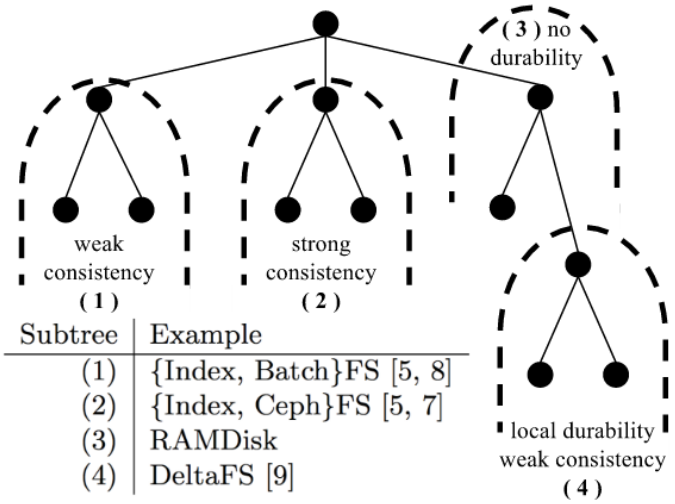
CudeleFS: Programmable Consistency and Durability in a Global Namespace

Paper 382

ABSTRACT

HPC developers are abandoning POSIX because the synchronization and serialization overheads of providing strong consistency and durability are too costly – and often uneccessary – for their applications. Unfortunately, designing near-POSIX file systems excludes applications that rely on strong consistency or durability, forcing developers to re-write their applications or deploy them on a different system. We present a file system and API that lets clients to specify their consistency/durability requirements and assign them to subtrees in the namespace, allowing administrators to optimize subtrees within the same namespace for different workloads. We draw conclusions about the performance impact of unexplored consistency/durability metadata designs and show that strong consistency can cause a 104× slow down while merging updates (7× slow down) and maintaining durability (10× slow down) have more reasonable costs.

ACM Reference format:
Paper 382. 2017. CudeleFS: Programmable Consistency and Durability in a Global Namespace. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn



Subtree	Example
(1)	{Index, Batch}FS [5, 8]
(2)	{Index, Ceph}FS [5, 7]
(3)	RAMDisk
(4)	DeltaFS [9]

Figure 1: Administrators can assign weaker consistency and durability policies to subtrees to get the same performance benefits of state-of-the-art HPC architectures. Applications that need stronger guarantees can still reside in the same namespace.

1 INTRODUCTION

File system metadata services in HPC have scalability problems. It has been shown that HPC workloads are metadata resource intensive because administrative tasks, like checkpointing [4] or scanning the file system [21], on large data sets leads to contention for the same directories and inodes (*e.g.*, path traversal). Applications perform better with dedicated metadata servers [13, 17] but provisioning a metadata server for every client is unreasonable. This problem is exacerbated by current trends in HPC, where 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]; this puts more pressure on data access because more requests end up hitting the same layer and latencies cannot be hidden while data migrates across tiers.

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, batch style jobs often do not need the strong consistency that the file system provides, so BatchFS [21] and DeltaFS [22] do more client side processing and merge updates when the job is done. HPC developers are turning to this non-POSIX solution 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 they wreak havoc on file systems designed for general-purpose workloads (*e.g.*, checkpoint-restart's N-N and N-1 create patterns).

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 [6, 7, 13, 21, 22]. Then the file system can change its internal structure to optimize performance. For example, the file system could enter a mode that prevents other clients from interferring with the decoupled directory. This delayed merge (*i.e.* a form of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of RPCs, synchronization, false sharing, and serialization. The consistency and

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

durability semantics for these systems is shown in the table in Figure 1 . While the performance benefits are obvious for these users, 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.

We propose subtree policies, an interface that lets future programmers control the consistency and durability for subtrees in the file system namespace. For performance, one subtree can adopt weaker consistency semantics while another subtree can retain the rigidity of POSIX's strong consistency. Figure 1 shows an example setup where a single global namespace has directories for applications designed for different, state-of-the-art HPC architectures. We present Cudele, a prototype programmable file system that supports different degrees of consistency and durability by exposing mechanisms used within the file system as a client library. 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. Our contributions:

(1) a prototype that lets administrators program a range of consistency and durability semantics (9 permutations), allowing them to custom fit the storage system to the application.
(2) an API for programming consistency/durability policies and assigning them to subtrees in the file system namespace.
(3) a comparison of the strategies used in recently proposed research systems against previously unexplored metadata designs.

Our results confirm the assertions of "clean-state" research systems that decouple namespaces; specifically that the technique drastically improve performance (104× speed up) but we go a step further by quantifying the costs of merging updates (7× slow down) and maintaining durability (10× slow down). We also show the effect of having a metadata specific file format in systems that are based on in-memory data structures. Section 6 places Cudele in the context of other related work. Section 2 quantifies the cost of POSIX consistency and system-defined durability and Section 3 presents the Cudele prototype and API. Section 4 describes Cudele's mechanisms and shows how re-using internal subsystems results in an implementation of less than 500 lines of code. The evaluation in Section 5 quantifies the overheads and performance gains of explored and previously unexplored metadata designs.

2 POSIX OVERHEADS

In our examiniation of the overheads of POSIX we benchmark and analyze CephFS, the file system that uses the Ceph's object store (*i.e.* RADOS) to store its data and metadata. We choose CephFS because it is an open-source production quality system. This file system is an implementation of one set of design decisions and our goal is to highlight the effect that those decisions have on performance.

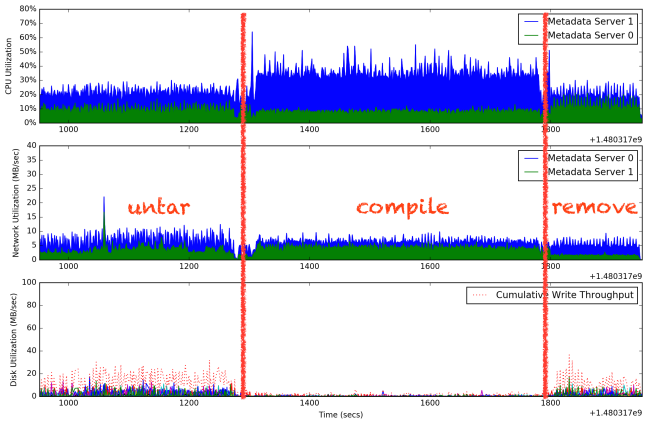


Figure 2: Create-heavy workloads (untar) incurr the highest disk, network, and CPU utilization because of the consistency and durability demands of CephFS.

To show how the file system behaves under high metadata load we use a create-heavy workload. Create-heavy workloads are studied the most in HPC research because of the checkpoint/restart use case but they also happen to stress the underyling storage system the most. Figure 2 shows the resource utilization of compiling the Linux kernel. The untar phase, which is characterized by many creates, has the highest resource usage which suggests that it is stressing the consistency and journaling subsystems of the metadata server the most. Traditional file system techniques for improving performance, such as caching inodes, do not help for create-heavy workloads.

In this section, we quantify the costs of strong consistency and global durability in CephFS. 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 the kernel client so that we can find the true create speed of the server; our experiments show a low CPU utilziation for the clients which indicates that we are stressing the servers more.

2.1 Durability

While durability is not specified by POSIX, users 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. 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 failure, metadata will be lost when components die in a None configuration.

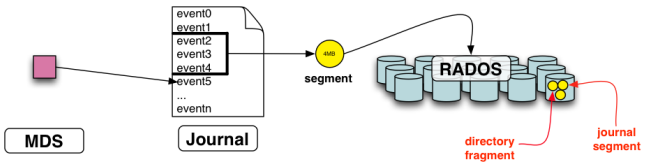


Figure 3: CephFS uses a journal to stage updates and tracks dirty metadata in the collective memory of the metadata servers. Each metadata server maintains its own journal, which is broken up into 4MB segments. These segments are pushed into RADOS and deleted when that particular segment is trimmed from the end of the log. In addition to journal segments, RADOS also stores per-directory objects.

CephFS Design: a journal of metadata updates that streams into the resilient object store. Similar to LFS [15] and WAFL [8] the metadata journal is designed to grow to large which ensures (1) sequential writes into the object store and (2) the ability for daemons to trim redundant or irrelevent journal entries. The journal is striped over objects where multiple journal updates can reside on the same object. There are two tunables for controlling the journal: the segment size and the number of parallel segments that can be written in parallel. Unless the journal saturates memory or CPU resources, larger values for these tunables results in better performance.

As shown in Figure 3, 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.

Figure 4 shows that journaling metadata updates into the object store has an overhead. Figure 4a shows the effect of journaling of different journal segment sizes; the larger the segment size the bigger that the writes into the object store are. The trade-off for better performance is memory consumption because larger segment sizes take up more space with their buffers. Figure 4b shows how the metadata server periodically stops serving requests to flush (*i.e.* apply journal updates to) the metadata store. The journal overhead is sufficient enough to slow down metadata throughput but not so much as to overwhelm the bandidth of the object store. We measured our peak bandwidth to be 100MB/s, which is the speed of our network link.

Comparison to decoupled namespaces: In BatchFS and DeltaFS, as far as we can tell, when a client or server fails there is no recovery scheme. For BatchFS, if a client fails when it is writing to the local log-structured merged tree (implemented as an SSTable) then those batched 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. IndexFS writes metadata to SSTables, which initially reside in memory but are later to flushed to the underlying distributed file system.

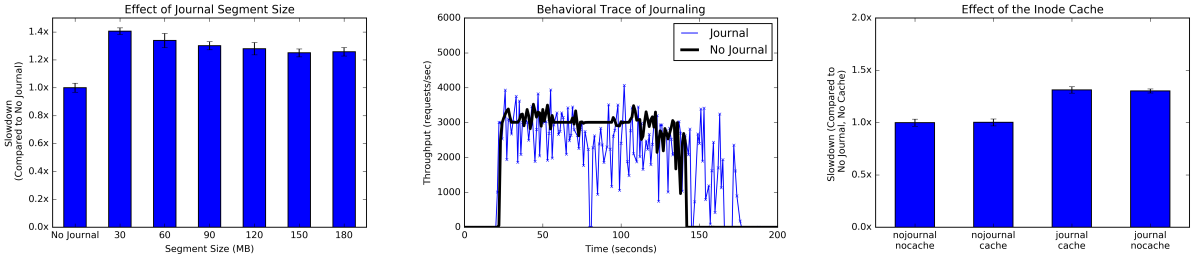
2.2 Strong Consistency

Access to metadata in a POSIX-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, reading and updating, caching reads, writing, buffering writes, changing the file size, and performing lazy IO.

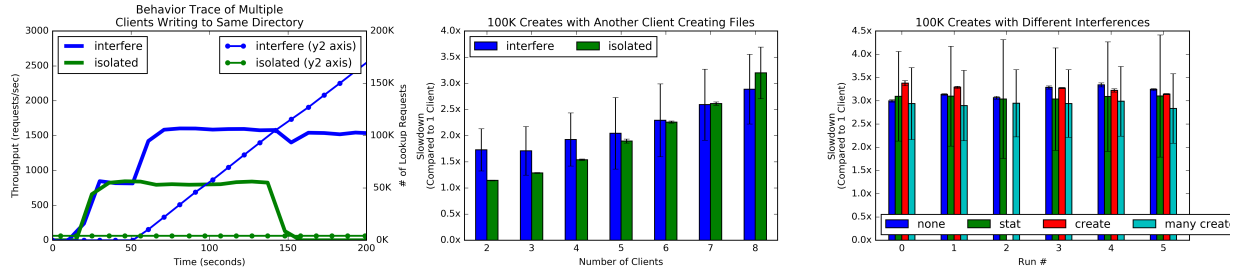
To keep track of the read caching and write buffering capabilities, the clients and metadata servers agree on the state of each inode using an inode cache. If a client has the directory inode cached it can do metadata writes (*e.g.*, create) with a single RPC. If the client is not caching the directory 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.

The benefits of caching the directory inode when creating files is shown in Figure 5a. If only one client is creating files in a directory ("isolated" 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 ("interfere" curve on $y1$ axis) then the directory inode transitions out of read caching and the first client must send `lookup()`s to the metadata server ("interfere" curve on $y2$ axis). These extra requests increase the throughput of the "interfere" curve because the metadata server can handle the extra load but the overall performance decreases. This degradation is shown in Figure 5b, where we scale the number of clients and show increased runtime and variability. The performance is compared to a single isolated client and the error bar is the standard deviation of the runtime of all clients. For the "interfere" bars, each client creates files in private directories and at 30 seconds we launch another process that creates files in those directories. For less than 7 clients, the runtime and deviation is worse when clients interfere. At 7 clients, the metadata server is overloaded so the directory caching in the "isolated" bars has no benefit. Zooming in on runs with 7 clients in Figure 5c we see how different interferring operations affect performance, again when compared to the performance of a single isolate client. "create" has little effect but "stat" and "create many" cause noticeable slowdowns because of the extra work they impose on the metadata server.



(a) Journal Segment Size **(b) 180MB Segment vs. No Journal** **(c) Inode Cache**

Figure 4: The overhead of the file system metadata journal. The segment size is the threshold that the metadata server starts trimming.



(a) Inteference forces `lookup()`s **(b) Interference hurts performance** **(c) Different interferring requests**

Figure 5: When a client create stream is "isolated" then lookups resolve locally but when a second client "interferes" by creating in the same directory, the directory inode capability is revoked forcing all clients to centralize lookups at the metadata server.

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 to sychronous mode. The merge itself is explicitly managed by the application but future work looks at more automated methdologies. In DeltaFS snapshots of 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.

3 METHODOLOGY: GLOBAL NAMESPACE WITH PER-SUBTREE CONSISTENCY/DURABILITY

In this section we describe CudeleFS, our file system 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. CudeleFS exposes these mechanisms and the administrator composes them 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, the policies, and the API for assigning policies to subtrees.

3.1 Cudele's Mechanisms

Figure 6 shows the mechanisms (labeled arrows) in Cudele and which daemon(s) they are performed by. Table 1 has a description of what each mechanism does. Of the 6 mechanisms in Figure 6 only 4 had to implemented and just 1 required changes to the underlying storage system itself.

3.1.1 No Changes. "RPCs" and "Stream" are part of the default CephFS implementation; they are used to get strong consistency and global durability, respectively. Using existing configuration settings in Ceph we can turn "Stream" on and off. If it is off, then the metadata servers will not save journals in the object store and the daemons that apply the journal to the metadata store will never run.

3.1.2 Library. for "Create", "Nonvolatile Apply", "Local/Global Persist", CudeleFS provides a library for clients to link into and all operations are performed by the client. Decoupled clients use the "Create" mechanism to append metadata updates to a local, in-memory journal. For "Local Persist", clients write serialized log events to a file on local disk and for "Global Persist", clients push the journal into

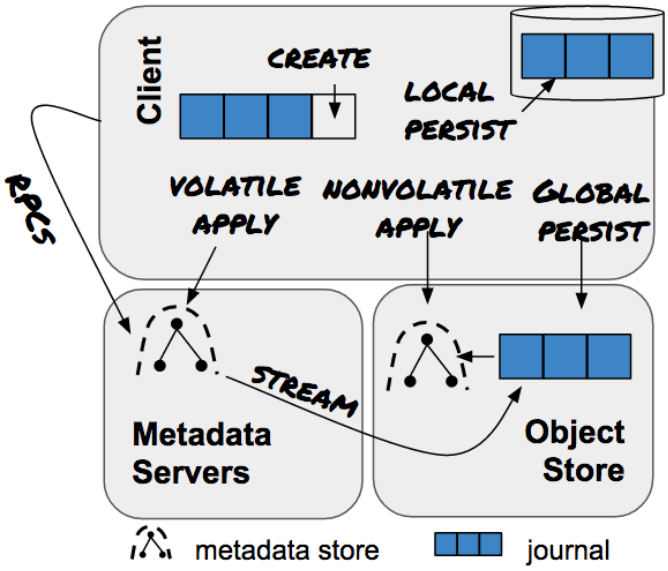


Figure 6: Applications decouple the namespace, write updates to a local journal, and delay metadata updates using the CudeleFS Mechanisms

the objects store. The overheads for both "Local Persist" and "Global Persist" is the write bandwidth of the local disk and object store, respectively. "Nonvolatile Apply" replays the client's journal onto the metadata cluster's metadata store. The client's in-memory journal is written into the object store and the metadata servers are restarted. 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. These implementations required no changes to CephFS because the metadata servers know how to read the events the library is writing into the object store. 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 [12]), and the functions for replaying events onto the internal namespace data structures.

3.1.3 Changes to Metadata Server. the "Volatile Apply" mechanism takes an in-memory journal on the client and applies the updates directly to the in-memory namespace 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.

In contrast, "Nonvolatile Apply" uses the the object store to apply the journal of updates from the client onto the metadata servers' metadata store. "Nonvolatile Apply" is safer but has a performance overhead because objects in the metadata store need to be read from and written back to the object store.

Mechanism	Description
RPCs	round trip remote procedure calls
Stream	stream journal into object store
Create	events appended to in-memory journal
Volatile Apply	apply to metadata store in obj store
Nonvolatile Apply	apply to metadata store in memory
Local Persist	journal saved to client's disk
Global Persist	journal saved in object store

Table 1: Cudele's mechanisms are composed together to form consistency and durability semantics.

Table 2: Future programmers can explore the consistency (C) and durability (D) spectrums by composing Cudele mechanisms.

C → / D ↓	invisible	weak	strong
none	create	create +volatile apply	RPCs
local	create +local persist	create +local persist +volatile apply	RPCs +local persist
global	create +global persist	create +global persist +volatile apply	RPCs +stream

3.2 Defining Policies in Cudele

The spectrum of consistency and durability guarantees that adminstrators can construct is shown in Table 2. 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 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 "global", where all updates are always recoverable.

Existing, state-of-the-art systems in HPC can be represented by the cells in Table 2. POSIX-compliant systems like CephFS and IndexFS have global consistency and durability; DeltaFS uses "invisible" consistency and "local" durability and BatchFS uses "weak" consistency and "local" durability. To compose the mechanisms administrators inject which mechanisms (described in Section §3.1) 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 2, not all of them make sense. For example, it makes little sense to do `creates+RPCs` since both mechanisms do the same thing or `stream+save` since "global" durability is stronger and has more overhead than "local" durability.

The consistency and durability properties in Table 2 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 globally consistent until they are safely saved in the object store. If a failure occurs during "global persist" or if we inject a new policy that chanages a subtree from "local persist" to "global persist", CudeleFS no guarantee until the mechanisms are complete.

3.3 Cudele Namespace API

The interface for setting the subtree policies is with {path, <block|overwrite>, pre-allocated inodes} tuples. 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 values:

- `allocated_inodes`: the number of inodes to allocate to the decoupled namespace (default 100)
- `interfere_callback`: how to handle a request from another client targeted at the now decoupled subtree (default `overwrite`)
- `consistency_callback`: which consistency model to use (default `RPCs`)
- `durability_callback`: which durability model to use (default `stream`)

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 `overwrite`. In this case, metadata 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. To implement DeltaFS on CudeleFS, the user would use the configuration from Listing 1. BatchFS's merging back into the global namespace can be achieved with the configuration in Listing 2.

4 IMPLEMENTATION

A programmable storage system exposes internal subsystem as building blocks for higher level services. This 'dirty-slate' approach limits redundant code and leverages the robustness of the underlying storage system. Cudele uses this approach

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "create"
    "durability": "local_persist"
}
```

Listing 1: Implementing DeltaFS with CudeleFS.

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "create+volatile_apply"
    "durability": "local_persist"
}
```

Listing 2: Implementing BatchFS with CudeleFS.

and re-uses some of the building blocks from the Malacology programmable storage system [16] to great success and requires only:

- 354 lines of library code
- 219 lines of non-destructive metadata server code, which is not used unless it is turned on
- 4 lines of destructive client/server code to check whether a namespace is decoupled

4.1 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.

Cudele: uses the metadata store format to write objects formatted like the journal into the object store. 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 namespace (*i.e.* merge).

4.2 Journal

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.

Cudele: uses the journal subsystems within the metadata server to replay updates onto the in-memory metadata store. When the clients are ready to merge their updates back into the global namespace, they pass a binary file of metadata updates to the metadata server. The metadata server uses its recovery code to "replay" the updates.

4.3 Journal Tool

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.

Cudele: using the journal tool, clients can materialize the journal in memory and append events to their own copy. When they are done they export their journal to a binary file. Clients also use the journal tool to save their updates to disk or to the object store.

4.4 Inode Cache

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 operations like `lookup()`. 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[1]. The inode cache has code for manipulating inode numbers, such as pre-allocating inodes to clients.

Cudele: 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.

4.5 Large Inodes

In CephFS, inodes store policies for 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.

Cudele: uses the large inodes to store consistency and durability policies in the directory inode. This approach uses the File Type interface from the Malacology programmable store system [16] 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.

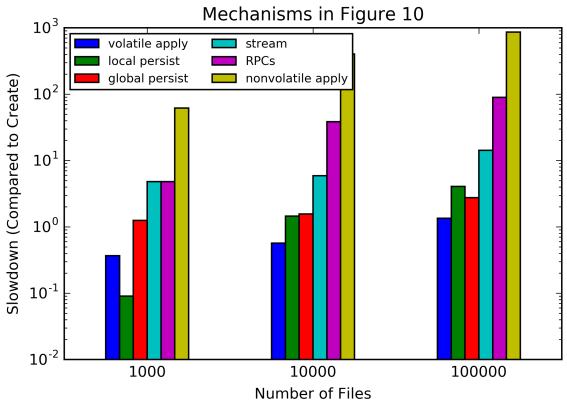


Figure 7: [source] The performance of the Cudele mechanisms normalized to the runtime of the create mechanism. The runtime of the create mechanism is the time it takes to write 100 file creates to the client's in-memory journal of metadata updates.

5 EVALUATION

We evaluate Cudele on a 15 node cluster, partitioned into 8 object storage servers, 3 metadata servers, and 2 monitor servers. The object storage servers double as clients which is fine because clients are CPU and memory bound while object storage servers are disk IO bound. All daemons run as a single process which is the default setting for Ceph and the nodes have 2 dual core 2GHz processors with 8GB of RAM. There are three daemons per object storage server (one for each disk formatted with XFS) and they share an SSD for the journal. The nodes are running Ubuntu 12.04.4, kernel version 3.2.0-63 but all experiments run in Docker containers; this makes it easier to tear down and re-initialize (*e.g.*, dropping the kernel cache) for the cluster between experiments.

This paper adheres to The Popper Convention[2] [9], so experiments presented here are available in the repository for this article[3]. Experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure. That link points to a Jupyter notebook that shows the analysis and source code for that graph, which points to an experiment and its artifacts.

5.1 Cudele Mechanism Performance

Figure 7 shows the runtime of the Cudele mechanisms, normalized to the time it takes to write 100K file create updates

[1] http://docs.ceph.com/docs/jewel/dev/mds_internals/data-structures/

[2] http://falsifiable.us

[3] https://github.com/michaelsevilla/cudele-popper/

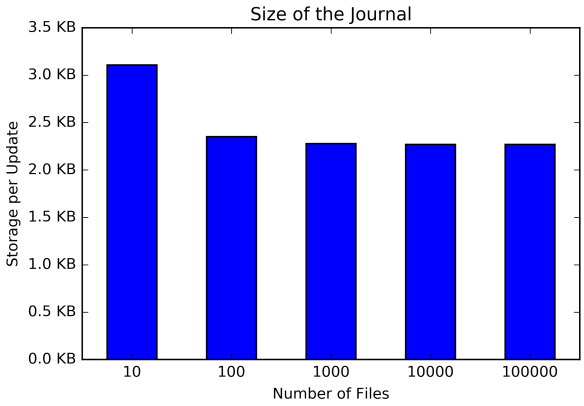


Figure 8: [source] In-memory client journal scales with the number of updates.

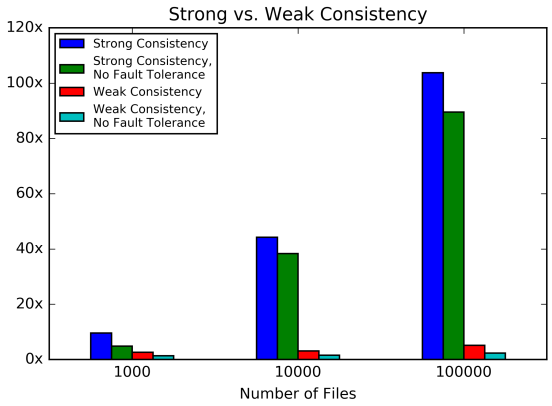


Figure 9: The RPC per metadata update of "Strong Consistency" has a large overhead compared to the decoupled namespace strategy of "Eventual Consistency".

to the client's in-memory journal (*i.e.* the create mechanism). Bars above 10^0 are slower than the create mechanism and bars below are faster. "Stream" is an approximation of the overhead and is calculated by subtracting the runtime of the job with the journal turned off from the runtime with the journal turned on. All the slowdowns and speedups reported are for the 100K file create job, the largest workoad we tested.

5.1.1 Overhead of RPCs. "RPCs" is 66× slower than "volatile apply" because sending individual metadata updates over the network is costly. While "RPCs" sends a request for every file create, "nonvolatile apply" writes all the updates to the in-memory journal and applies them to the in-memory data structures in the metadata server. While communicating the decoupled namespace directly to the metadata server is faster, communicating through the object store ("nonvolatile apply") is 10× slower.

5.1.2 Overhead of "nonvolatile apply". The cost of "nonvolatile apply" is much larger than all the other mechanisms. That mechanism 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.* `/`). The cost of communcating through the object store is shown by comparing the runtime of "volatile apply" + "global persist" to "nonvolatile apply". These two operations end up with the same final metadata state but using "nonvolatile apply" is clearly inferior.

5.1.3 Parallelism of the Object Store. Comparing "local" and "global persist" demonstrates the bandwidth advantages of storing the journal in a distributed object store. As the journal size increases, the "global persist" performance is 1.5× faster because the object store 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 size of the object store can help mitigate the overheads of globally persisting metadata updates.

5.1.4 Journal Size. Figure 8 shows the amount of storage per journal update (y axis) for the range of file creates we tested (x axis). The increase in file size is linear with the number of metadata creates and suggests that updates for a million files would be 2.5KB $*$ 1 million files $=$ 2.38GB. Transfer times for files this large on an HPC network are reasonable.

5.2 Eventual vs. Strong Consistency

Figure 9 shows the runtimes of weak and strong consistency based systems implemented on Cudele, normalized to the runtime of the create mechanism (again, just creating files in the client's in-memory journal). We scaled the number of files up to 100K which is the maximum size of a directory by default in CephFS. We use the following compositions from the mechanisms in Table 2:

- Strong Consistency
  `RPCs` + `stream`
- Strong Consistency, No Fault Tolerance
  `RPCs`
- Eventual Consistency
  `creates` + `local persist`
- Eventual Consistency, No Fault Tolerance
  `creates` + `local persist` + `volatile apply`

We compare these semantics because the final metadata states are equivalent. Cudele makes 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.

5.2.1 Speedups of Decoupled Namespaces. Eventual consistency uses the decoupled namespace strategy and shows up to a 20× speedup over the traditional namespaces that use RPCs. Compared to the baseline the slowdown is $5 - 7\times$ for Strong Consistency, which emulates BatchFS and $90 - 104\times$ for Eventual Consistency, which emulates DeltaFS.

5.2.2 Durability << Consistency. The 1.15× overhead of "Strong Consistency" compared to "Strong Consistency, No Fault Tolerance" for 100K files is neglible. It suggests that the overhead of consistency is much larger than the overhead of durability. This conclusion should be stronger as we scale the number of files because the cost of streaming the journal into the object store is constant. We omit the same analysis for "Eventual Consistency" because the runtimes are so short that the normalized slowdowns are misleading.

5.2.3 Metadata Formats. Because the metadata formats are the same for all schemes we argue that the performance gain for decoupled namespaces comes from relaxing the consistency guarantees and not from the metadata formats, as was argued in previous work outlining the benefit of SSTables [12, 13].

5.3 Isolation from EBUSY

In this section, we show the isolation benefits of subtree policies. We repeat the experiment from Figure 5b, where clients write to their own private directories and another client interferes at 30 seconds. We also have 2 clients write to decoupled namespaces and merge their updates 90 seconds. The

5.3.1 Benefits of Isolated Subtree Policies.

5.3.2 Cost of Merging.

5.3.3 Scaling Concurrent Merges.

5.4 Isolation from Global Writes, Cost of Overwrites

5.5 Cost of Nonvolatile Apply

Creating many files in the same directory would touch the same object but the existing implementation results in this object being repeatedly pushed/pulled.

5.6 Partial Directory Listings

6 RELATED WORK

The bottlenecks associated with accessing POSIX 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 [18]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [10]), 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 [14]. Unfortunately, scaling file system metadata is a well-known problem and solutions for scaling data IO do not work for metadata IO [1–3, 14, 19]. There are two approaches for improving the performance of metadata access.

6.1 Metadata Load Balancing

One approach for improving metadata performance and scalability is to alleviate overloaded servers by load balancing metadata IO across a cluster. Common techniques include partitioning metadata when there are many writes and replicating metadata when there are many reads. For example, IndexFS [13] partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal; it does well for strong scaling because servers can keep more inodes in the cache which results in less RPCs. Alternatively, ShardFS replicates directory state so servers do not need to contact peers for path traversal; it does well for read workloads because all file operations only require 1 RPC and for weak scaling because requests will never incurr extra RPCs due to a full cache. CephFS employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies do not change depending on the balancing technique. Despite the performance benefits these techniques add complexity and jeopardize the robustness and performance characteristics of the metadata service because the systems now need (1) policies to guide the migration decisions and (2) mechanisms to address inconsistent states across servers.

Setting policies for migrations is arguably more difficult than adding the migration mechanisms themselves. For example, IndexFS and CephFS use the GIGA+ [11] technique for partitioning directories at a predefined threshold and using lazy synchronization to redirect queries to the server that "owns" the targeted metadata. Determining when to partition directories and when to migrate the directory fragments are policies that vary between systems: GIGA+ partitions directories when the size reaches a certain number of files and migrates directory fragments immediately; CephFS partitions directories when they reach a threshold size or when the write temperature reaches a certain value and migrates directory fragments when the hosting server has more load than the other servers in the metadata cluster. Another policy is when and how to replicate directory state; ShardFS replicates immediately and pessimistically while CephFS replicates only when the read temperature reaches a threshold. There is a wide range of policies and it is difficult to maneuver tunables and hard-coded design decisions.

In addition to the policies, distributing metadata across a cluster requiress distributed transactions and cache coherence protocols to ensure strong consistency (e.g., POSIX). For example, ShardFS pessimistically replicates directory state and uses optimistic concurrency control for conflicts; namely it does the operation and if there is a conflict at verification time it falls back to two-phase locking. Another example is IndexFS's inode cache which reduces RPCs by caching

ancestor paths – the locality of this cache can be thrashed by random reads but performs well for metadata writes. For consistency, writes to directories in IndexFS block until the lease expires while writes to directories in ShardFS are slow for everyone as it either requires serialization or locking with many servers; reads in IndexFS are subject to cache locality while reads in ShardFS always resolve to 1 RPC. Another example of the overheads of addressing inconsistency is how CephFS maintains client sessions and inode caches for capabilities (which in turn make metadata access faster). When metadata is exchanged between metadata servers these sessions/caches must be flushed and new statistics exchanged with a scatter-gather process; this halts updates on the directories and blocks until the authoratitive metadata server responds. These protocols are discusssed in more detail in the next section but their inclusion here is a testament to the complexity of migrating metadata.

The conclusion we have drawn from this related work is that metadata protocols have a bigger impact on performance and scalability than load balancing. Understanding these protocols helps load balancing and gives us a better understanding of the metrics we should use to make migration decisions (*e.g.*, which operations reflect the state of the system), what types of requests cause the most load, and how an overloaded system reacts (*e.g.*, increasing latencies, lower throughput, etc.).

6.2 Relaxing POSIX

POSIX 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 [13, 20] 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 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. Similarily, DeltaFS eliminates RPC traffic using subtree snapshots for non-conflicting workloads and middleware for conflicting workloads. MarFS gives administrators 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; apps 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 thesmelves 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 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 and REST are the dominant APIs for data transfer.

REFERENCES

[1] Christina L. Abad, Huong Luu, Yi Lu, and R Campbell. 2012. *Metadata Workloads for Testing Big Storage Systems*. Technical Report. Citeseer.

[2] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. 2012. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*. 125–132. http://dx.doi.org/10.1109/UCC.2012.27

[3] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. 2011. Parallel I/O and the Metadata Wall. In *Proceedings of the 6th Workshop on Parallel Data Storage (PDSW'11)*.

[4] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009) *(SC '09)*.

[5] John Bent, Brad Settlemyer, and Gary Grider. Serving Data to the Lunatic Fringe. (????).

[6] Sorin Faibish, John Bent, Uday Gupta, Dennis Ting, and Percy Tzelnic. Slides: 2 Tier Storage Architecture. http://www.pdl.cmu.edu/SDI/2015/slides/Faibish-CMU-PDL-Spring-2015-final.pdf

[7] Gary Grider, Dave Montoya, Hsing-bung Chen, Brett Kettering, Jeff Inman, Chris DeJager, Alfred Torrez, Kyle Lamb, Chris Hoffman, David Bonnie, Ronald Croonenberg, Matthew Broomfield, Sean Leffler, Parks Fields, Jeff Kuehn, and John Bent. 2015. MarFS - A Scalable Near-Posix Metadata File System with Cloud Based Object Backend. In *Work-in-Progress at Proceedings of the 10th Workshop on Parallel Data Storage (PDSW'15)*.

[8] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference (WTEC'94)*.

[9] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. 2016. *Popper: Making Reproducible Systems*

Performance Evaluation Practical, UCSC-SOE-16-10. Technical Report UCSC-SOE-16-10. UC Santa Cruz.

[10] Kirk McKusick and Sean Quinlan. 2010. GFS: Evolution on Fast-forward. *Commununications ACM* 53, 3 (March 2010), 42–49.

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

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

[13] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the 20th ACM/IEEE Conference on Supercomputing (SC '14)*.

[14] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. 4–4.

[15] M. Rosenblum and J.K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. In *ACM Transactions on Computer Systems*.

[16] Michael A. Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the 12th European Conference on Computer Systems (Eurosys '17)*. Belgrade, Serbia. To Appear, preprint: https://www.soe.ucsc.edu/research/technical-reports/UCSC-SOE-17-04.

[17] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[18] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*.

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

[20] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing (SC'04)*.

[21] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Workshop on Parallel Data Storage (PDSW' 14)*.

[22] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Workshop on Parallel Data Storage (PDSW' 15)*.