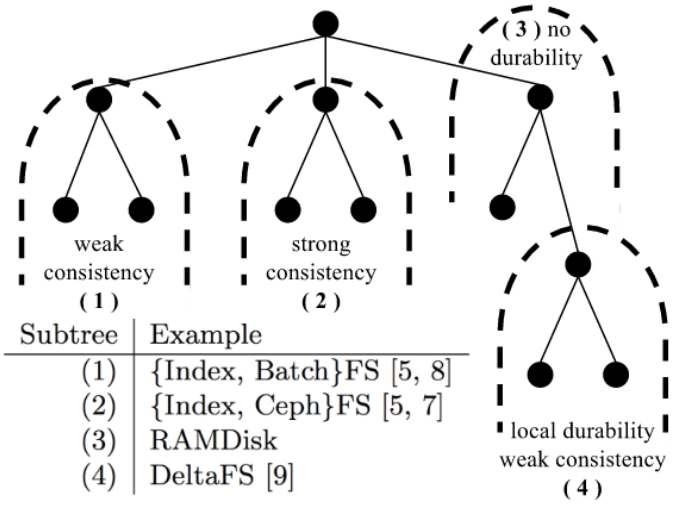
CudeleFS: Programmable Consistency and Durability in a Global Namespace

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 unnecessary – 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 specify their consistency/durability requirements and assign them to subtrees in the namespace, allowing users 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 maintaining strong consistency can cause about a 100× slow down compared to relaxed consistency and no durability. Comparatively, merging updates after a period of relaxed consistency (less than a 10× slow down) and maintaining durability (about a 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),* **??** 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: With CudeleFS, users can assign weaker consistency and durability policies to subtrees to get the same performance benefits of state-of-the-art HPC architectures, such as those shown in the embedded table, while applications that require the stronger guarantees of POSIX 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 [**?**] or scanning the file system [**?**], on large data sets lead to contention for the same directories and inodes (*e.g.*, path traversal). Applications perform better with dedicated metadata servers [**? ?**] 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 [**?**]; 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 [**?**] and DeltaFS [**?**] do more client-side processing and merge updates when the job is done. HPC developers are turning to these non-POSIX 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).

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 [**? ? ? ? ?**]. 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 interfering 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 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 propose a subtree policy API that lets developers dynamically control the consistency and durability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00

DOI: 10.1145/nnnnnnn.nnnnnnn

guarantees for subtrees in the file system namespace. For example, one subtree can adopt weaker consistency semantics while another subtree can retain the rigidity of POSIX's strong consistency. The subtrees in Figure **??** show a setup in our proposed system where a single global namespace has subtrees for applications optimized with techniques from different state-of-the-art HPC architectures.

In this work, we present CudeleFS, a prototype programmable file system that supports different degrees of consistency and durability in a global namespace. CudeleFS achieves this by exposing "mechanisms" that developers use to specify their preferred semantics. CudeleFS supports 3 forms of consistency (invisible, weak, and strong) and 3 degrees of durability (none, local, and global) giving the user a wide range of policies and optimizations that can be custom fit to an application. We make the following contributions:

(1) a prototype that lets developers choose from a range of consistency and durability semantics (9 permutations), allowing them to dynamically custom fit the storage system to the application.
(2) an API for selecting consistency/durability policies and assigning them to subtrees in the file system namespace.
(3) an apples-to-apples comparison of the strategies used in recently proposed research systems against previously unexplored metadata designs, all implemented using CudeleFS.

Our results confirm the assertions of "clean-state" research systems that decouple namespaces; specifically that the technique drastically improves 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. In the remainder of the paper, Section **??** quantifies the cost of POSIX consistency and system-defined durability and Section **??** presents the CudeleFS prototype and API. Section **??** describes the CudeleFS mechanisms and shows how re-using internal subsystems results in an implementation of less than 500 lines of code. The evaluation in Section **??** quantifies the overheads and performance gains of explored and previously unexplored metadata designs. Section **??** places CudeleFS in the context of other related work.

2 POSIX OVERHEADS

In our examination 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. During this process we discovered, based on the analysis and breakdown of costs, that durability and consistency have high overhead.

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 underlying storage system the most. Figure **??** 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.

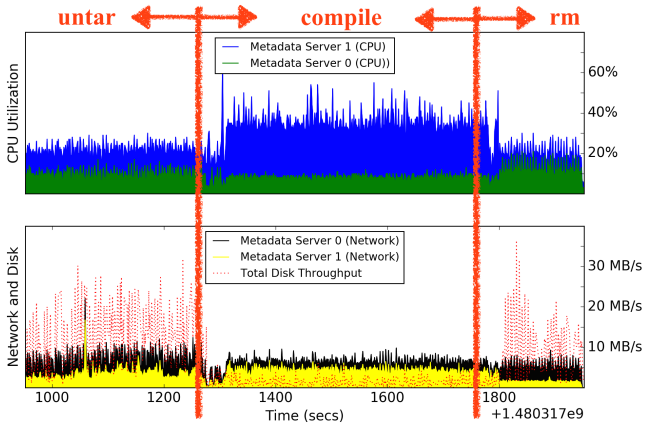


Figure 2: Create-heavy workloads (such as `untar`) incur the highest disk, network, and CPU utilization because of the consistency and durability demands of CephFS.

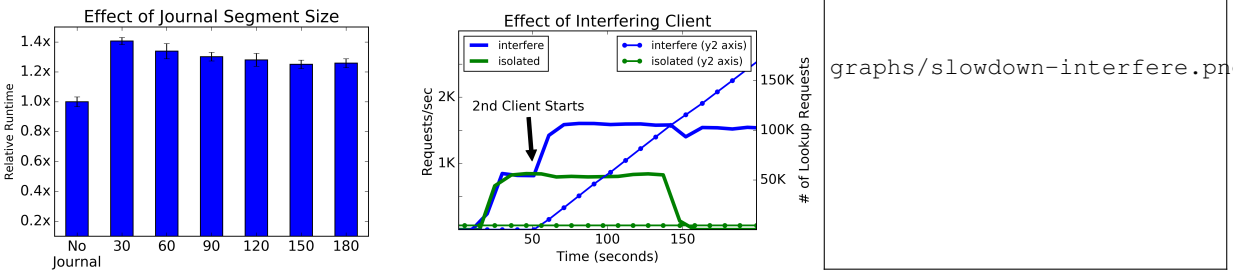
In this section, we quantify the costs of strong consistency and global durability in CephFS. At the end of each subsection we compare the approach to "decoupled namespaces", the technique in related work that detaches subtrees from the global namespace to relax consistency/durability guarantees. We use the kernel client so that we can find the true create speed of the server; our experiments show a low CPU utilization 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.

CephFS Design: a journal of metadata updates that streams into the resilient object store. Similar to LFS [**?**] and WAFL [**?**] 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 irrelevant 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 **??**, 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



(a) Runtime normalized to no journal **(b) Interference forces `lookup()`s** **(c) Interference hurts variability**

Figure 3: The overhead of global durability and strong consistency in CephFS. (a) shows the effect of different journal segment sizes, which are streamed into the object store for fault tolerance. (b) and (c) show that when a second client "interferes", capabilities are revoked and metadata servers do more work.

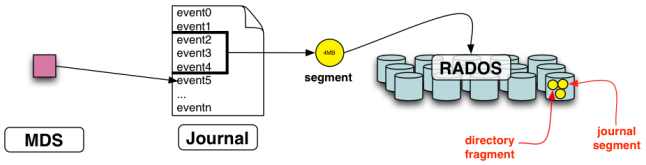


Figure 4: CephFS has two views of the file system namespace: a journal of metadata updates and a metadata store. For fault tolerance, they are stored in the object store as segments and fragments, respectively.

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 **??** 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 segments take up more space for buffering. When journaling is on, 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 bandwidth 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, to the best of our knowledge, 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 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, creating and updating 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.

The benefits of caching the directory inode when creating files is shown in Figure **??**. 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 performance suffers. This degradation is shown in Figure **??**, where we scale the number of clients and show increased variability. The results are normalized to a single isolated client and the y axis is the standard deviation of the runtime of all clients. For the "interfere" bars, each client creates

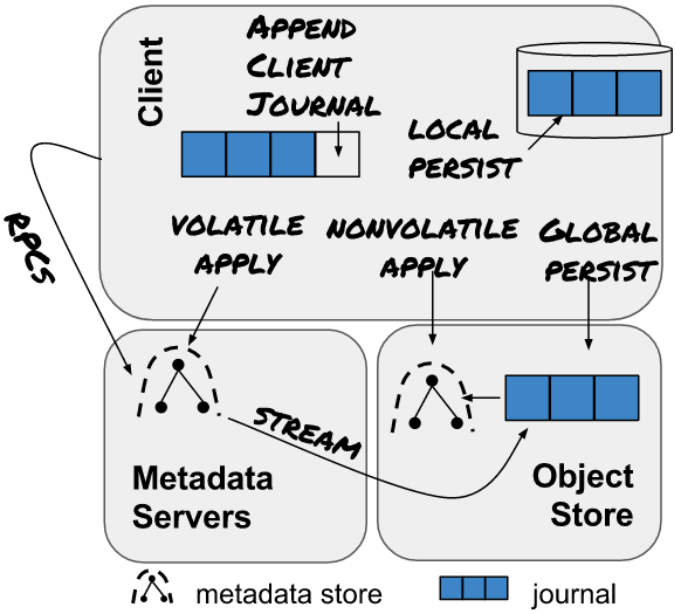


Figure 5: Applications decouple the namespace, write updates to a local journal, and delay metadata updates using the CudeleFS Mechanisms

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.

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 synchronous mode. The merge itself is explicitly managed by the application but future work looks at more automated methodologies. 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 users 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 user 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.

Mechanism	Description
RPCs	round trip remote procedure calls
Stream	stream journal into object store
Append Client Journal	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: The CudeleFS mechanisms are composed together to form consistency and durability semantics.

3.1 The CudeleFS Mechanisms

Figure **??** shows the mechanisms (labeled arrows) in CudeleFS and which daemon(s) they are performed by. Table **??** has a description of what each mechanism does. Decoupled clients use the "Append Client Journal" mechanism to append metadata updates to a local, in-memory journal. Once the job is complete, the system calls CudeleFS mechanisms to achieve the desired consistency/durability semantics. CudeleFS provides a library for clients to link into and all operations are performed by the client.

3.1.1 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 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 reinitialize, 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 – CudeleFS 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.

3.1.2 Mechanisms Used for Durability. "Stream" is one of the mechanisms used by default in CephFS. 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. For "Local Persist", clients write serialized log events to a file on

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 2: Users can explore the consistency (C) and durability (D) spectrums by composing CudeleFS mechanisms.

local disk and for "Global Persist", clients push the journal into the objects 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.

3.2 Defining Policies in CudeleFS

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

Existing, state-of-the-art systems in HPC can be represented by the cells in Table **??**. POSIX-compliant systems like CephFS and IndexFS have global consistency and durability [1]; 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 users 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 **??**, 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+save` since "global" durability is stronger and has more overhead than "local" durability.

The consistency and durability properties in Table **??** 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", CudeleFS makes no guarantee until the mechanisms are complete.

[1] This is the normal case. IndexFS also has bulk merge which would transition the system into "weak consistency"

3.3 CudeleFS Namespace API

Users control consistency and durability for subtrees by contacting the monitor daemon with a directory path and presenting a policies configuration file. 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):

- `consistency`: which consistency model to use (`RPCs`)
- `durability`: which durability model to use (`stream`)
- `allocated_inodes`: the number of inodes to provision to the decoupled namespace (100)
- `interfere_policy`: how to handle a request from another client targeted at this subtree (`allow`))

The "Consistency" and "Durability" parameters are set to the CudeleFS mechanisms; they can be composed (+) 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. To implement DeltaFS on CudeleFS, the user would use the configuration:

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

BatchFS merges updates into the global namespace after the job is complete. This can be achieved with by composing two mechanisms:

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

4 IMPLEMENTATION

We use a programmable storage approach [**?**] to design CudeleFS; namely, we try to leverage components inside Ceph 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 **??** and just 1 required changes to the underlying storage system itself. In this section, we first describe a Ceph internal subsystem or component and then we show how we use it in CudeleFS.

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.

CudeleFS: the "RPCs" mechanism uses the metadata store to service requests. Using code designed for recovery, "Nonvolatile Apply" and "Volatile Apply" replay updates onto the metadata store in memory and in the object store, respectively. 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.

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.

CudeleFS: 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 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, CudeleFS leverages the write/read optimized data structures, the formats for persisting events (similar to TableFS's SSTables [**?**]), and the functions for replaying events onto the internal namespace data structures.

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.

CudeleFS: 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".

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 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[2]. The inode cache has code for manipulating inode numbers, such as pre-allocating inodes to clients.

CudeleFS: "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.

4.5 Large Inodes

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.

CudeleFS: 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 [**?**] and it tells clients how to access the underlying metadata. The underlying implementation stores executable code in the inode that calls the different CudeleFS mechanisms. Of course, there are many security and access control aspects of this approach but that is beyond the scope of this paper.

5 EVALUATION

We evaluate CudeleFS 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

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

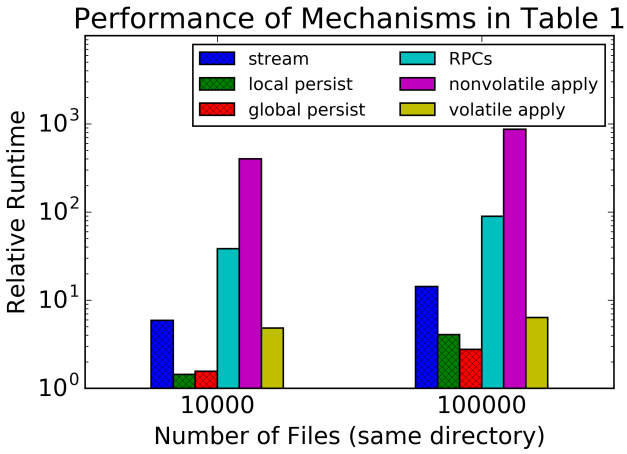


Figure 6: [source] The performance of the CudeleFS mechanisms normalized to the runtime of the "Append Client Journal" mechanism (the runtime of writing n file creates to the client's in-memory journal).

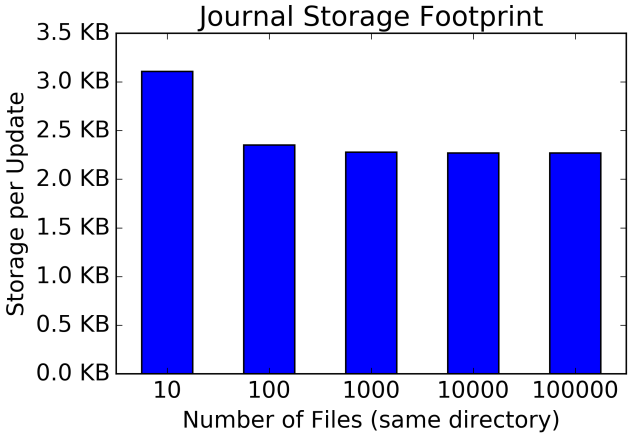


Figure 7: [source] The size of the client's journal scales with the number of updates.

easier to tear down and re-initialize the cluster (*e.g.*, dropping the kernel cache) between experiments.

This paper adheres to The Popper Convention[3] [**?**], so experiments presented here are available in the repository for this article[4]. 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 CudeleFS Mechanism Performance

Figure **??** shows the runtime of the CudeleFS mechanisms, normalized to the time it takes to write n file create updates to the client's in-memory journal (*i.e.* the "Append Client Journal" mechanism). Bars above 10^0 are slower than the "Append Client Journal" mechanism. "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. The largest workload we tested is 100K file creates in the same directory. This is the maximum recommended size of a directory in CephFS; preliminary experiments with larger directory sizes show memory problems.

5.1.1 Poorly Scaling Data Structures. Despite doing the same amount of work, mechanisms that rely on poorly scaling data structures have large slowdowns for the larger number of creates. For example, "RPCs" which relies on the internal CephFS directory structures, goes from a $40\times$ slowdown for 10K files to a $90\times$ slowdown for 100K files. It is a well-known problem that directory data structures do not scale when creating files in the same directory [**?**] and any mechanism that uses these data structures will experience similar slowdowns. Other mechanisms that write events to a journal (*e.g.* the persists, "Volatile Apply") 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.

5.1.2 Overhead of RPCs. "RPCs" is $66\times$ 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 $10\times$ slower.

5.1.3 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 CudeleFS – 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 communicating 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.

5.1.4 Parallelism of the Object Store. Comparing "Local" and "Global persist" demonstrates the bandwidth advantages of storing the journal in a distributed object store. For 100K file creates, the "Global Persist" performance is $1.5\times$ 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.

[3] http://falsifiable.us
[4] Links have been removed for double-blind submission

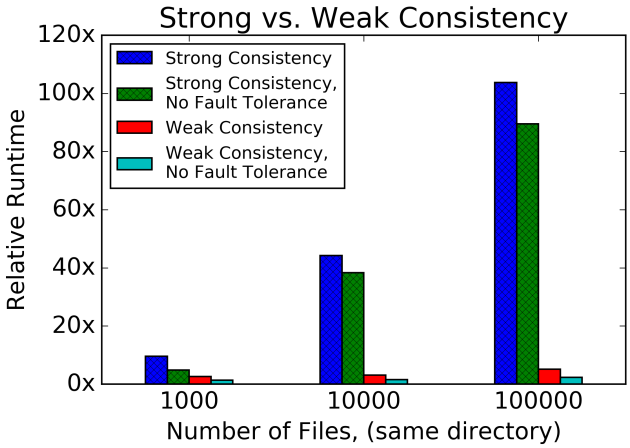


Figure 8: [source] The RPC per metadata update of "Strong Consistency" has a large overhead compared to the decoupled namespace strategy of "Weak Consistency".

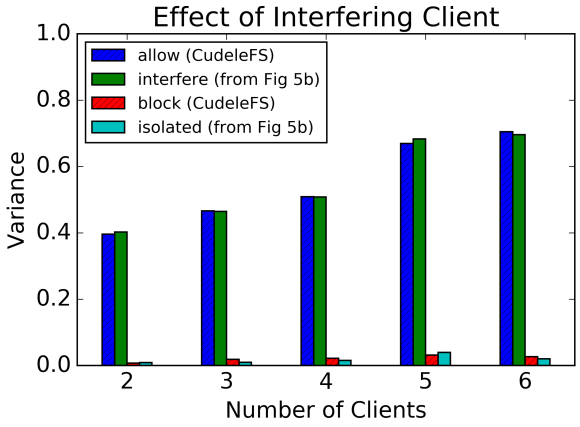


Figure 9: [source] Using the "allow" and "block" API, users can isolate directories from interfering clients. Variance with blocking turned on is the same as "isolated" from Figure ??.

5.1.5 Journal Size. Figure **??** 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 $2.5\text{KB} * 1 \text{ million files} = 2.38\text{GB}$. Transfer times for payloads of this size on an HPC network are reasonable.

Takeaway: the CudeleFS mechanisms have overheads and costs that can differ *by orders of magnitude*. CudeleFS gives users the ability to compose these mechanisms based on their application's correctness requirements and performance goals.

5.2 Weak vs. Strong Consistency

Figure **??** shows the runtime of systems employing weak and strong consistency, normalized to the runtime of the "Append Client Journal" mechanism (again, just creating files in the client's in-memory journal). We use the following compositions from the mechanisms in Table **??**: Strong Consistency = "RPCs" + "Stream"; Strong Consistency, No Fault Tolerance = "RPCs"; Weak Consistency = "Append Client Journal" + "Local Persist"; and Weak Consistency, No Fault Tolerance = "Append Client Journal" + "Local Persist" + "Volatile Apply".

We compare these semantics because the final metadata states are equivalent. CudeleFS 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. Weak consistency uses the decoupled namespace strategy and shows up to a $20\times$ 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 Weak Consistency, which emulates DeltaFS.

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "RPCs"
    "durability": "stream"
}
```

5.2.2 Durability << Consistency. The $1.15\times$ overhead of Strong Consistency compared to Strong Consistency, No Fault Tolerance for 100K files is negligible. 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 Weak Consistency because the runtimes are so short that the normalized slowdowns are misleading.

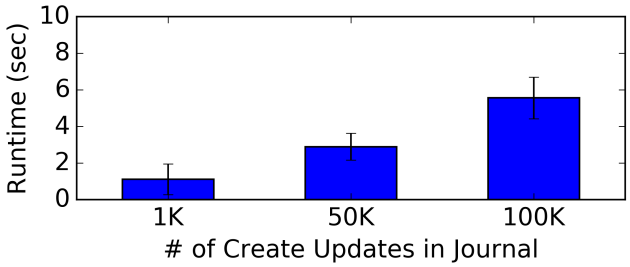
5.2.3 Metadata Formats. Because the metadata formats are the same for all schemes we argue that the performance gain for decoupled namespaces comes from relaxing the consistency guarantees and not from the metadata formats.

Takeaway: CudeleFS shows the true benefit of eventual consistency, where we see over a $100\times$ slowdown for achieving strong consistency, in the worst case.

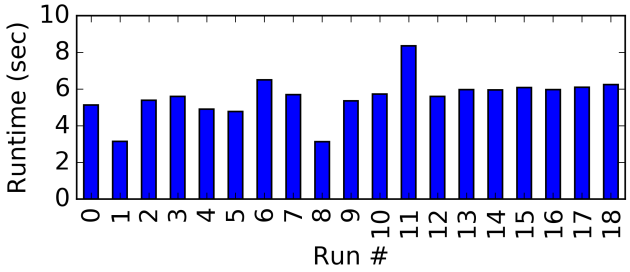
5.3 Benefits of Isolated Subtree Policies

Next we show how CudeleFS can be programmed to block interfering clients. We use the same problematic workload from Figure **??**, where clients write to their own private directories and another client interferes with a stream of creates at 30 seconds. We also have another client write to a decoupled namespace and merge its updates at 90 seconds. We equip each client directory with the configuration:

Each directory functions with "RPCs" and "Stream" enabled – which is the default implementation of CephFS. The only difference is that we enable blocking on each subtree so while the tree behaves like CephFS, all interfering operations will be returned with



(a) Merging journals serially.



(b) Merging a journal w/ 100K create updates.

Figure 10: Merging updates has linear scalability.

`-EBUSY`. Note that IndexFS does a similar operation with leases except clients block.

To show the benefits of this isolations, our results in Figure **??** are plotted alongside the variance bars from Figure **??**. Because "allow"/"interfere and "block"/"isolated" have the same variability we draw the following three conclusions: (1) clients that use the API to block interfering clients get the same performance as isolated clients, (2) there is a negligible effect on performance for the extra work the metadata server does to return `-EBUSY`, and (3) merging updates from the decoupled client has a negligible effect on performance.

Takeaway: the API lets users isolate directories when applications need better and more reliable performance. This is a way of controlling consistency.

5.4 Scaling Serialized Merges

CudeleFS serializes merge requests at the metadata server before applying the updates to the in-memory metadata store. Merging concurrently is an optimization for future work, but merges for the largest journal size we tested were quite fast. Furthermore, since we place no restrictions on the validity of metadata inserted into the journal, we can avoid touching poorly scaling data structures. In other words, we can scale linearly if we weaken our consistency by never checking for the existence of files before creating files.

Figure **??** demonstrates this scalability by showing the runtime (y axis) of merging journals of different sizes (x axis). The variance of the runtime is over 20 serial merges, each hitting a different directory in the namespace. The runtime scales linearly and with low variance because the "Create" mechanism that was used to create the journal only appends `open()` requests to the journal. When the updates are merged by the metadata server into the in-memory metadata store they never scan growing data structures. Had we implemented the "Create" mechanism to include `lookup()` commands before `open()` requests, we would have seen the poor scaling that we see with the "RPCs" mechanism. Figure **??** confirms that we do not touch a poorly scaling data structure because the runtimes do not increase as we add more metadata.

Takeaway: the weakest form of consistency for creating files (*i.e.* not checking data structures for the validity of an update or existence of a file) shows linear scalability and stable performance.

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 [**?**]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [**?**]), 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 [**?**]. Unfortunately, scaling file system metadata is a well-known problem and solutions for scaling data IO do not work for metadata IO [**? ? ? ? ?**]. 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 [**?**] 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 incur 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+ [**?**] 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 requires 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 authoritative metadata server responds. These protocols are discussed 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 [**? ?**] 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. 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 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.

7 CONCLUSION

Relaxing consistency and durability semantics in the file system is a double-edged sword. While it achieves better performance and scalability, it alienates applications that rely on strong consistency and durability. CudeleFS lets users assign consistency and durability guarantees to subtrees in the global namespace, giving users a wide range of policies and optimizations that can be custom fit to the application. Using CudeleFS, we compare related work on the same file system and conclude that strong consistency can cause a $104\times$ slow down while merging updates and maintaining durability only have between a $7 - 10\times$ slow down.