Cudele: Programmable Consistency and Fault Tolerance

Blind Author
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
blindauthor@corporation.com

ABSTRACT

Today's large and highly-parallel workloads are bottlenecked by metadata services because many processes end up accessing the same shared resoure. In HPC, state-of-the-art file systems are abandoning POSIX because the sychronization and serializiation overheads are too costly – and sometimes even unnceccessary – for some of their applications. While the performance benefits are plain for these users, other applications that rely on stronger consistency must be re-written or deployed on a different system. We present Cudele, a programmable file system that supports different degrees of consistency and fault tolerance within the same namespace. First, we take a POSIX compliant file system and relax the consistency constraints by implementing two metadata designs: delayed metadata update merge into the global namespace and client local views of metadata updates. Second, we store the consistency and fault tolerance semantics in inodes so that subtrees within the same namespace can be optimized for different workloads; the inodes are programmable so that clients understand how to access the metadata in a subtree. Third, we present a theoretical framework for a metadata service and a working implementation that provides the lowest common denominator needed to implement a wide range of consistency/fault tolerance semantics that can be benchmarked all on the same system.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Blind Author. 1997. Cudele: Programmable Consistency and Fault Tolerance. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 10 pages.
DOI: 10.475/123_4

WOODSTOCK'97, El Paso, Texas USA

DOI: 10.475/123_4

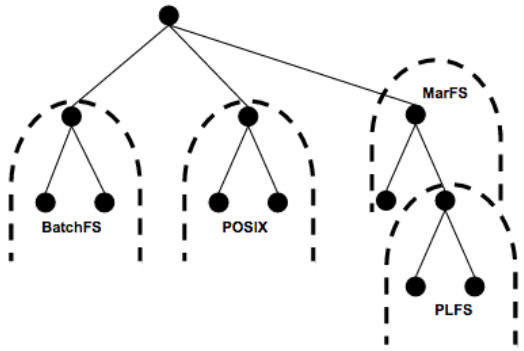


Figure 1: Administrators can assign consistency and fault tolerance policies to subtrees to get the benefits of some of the state-of-the-art HPC architectures.

1 INTRODUCTION

Today's client-server based file system metadata services have scalability problems. It takes a lot of resources to server POSIX metadata requests and applications perform better with dedicated metadata servers [1, 2]. This is fine for small workloads and file systems but as the system scales provisioning a metadata server for every client is expensive and complicated.

Current hardware evolution and the rise of software-defined storage storage, which uses techniqes like erasure coding, replication, and partitioning, have ushered a new era of HPC computing; architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to a two layer stack with just a burst buffer and object store [**?**]. This trend exacerbates the metadata scalability problem and has given rise to the serverless metadata services.

```
def hello():
```

HPC workloads are so metadata intensive that new management techniques are advocating reducing synchronization and serialization overheads by transferring these responsibilities to the client. We call this approach decoupling the namespace and the semantics of consistency differ amongst systems.

We propose subtree policies, an interface that lets future programmers control how the system manages different parts of the namespace. For performance one subtree can adopt weaker consistency semantics while another subtree can retain

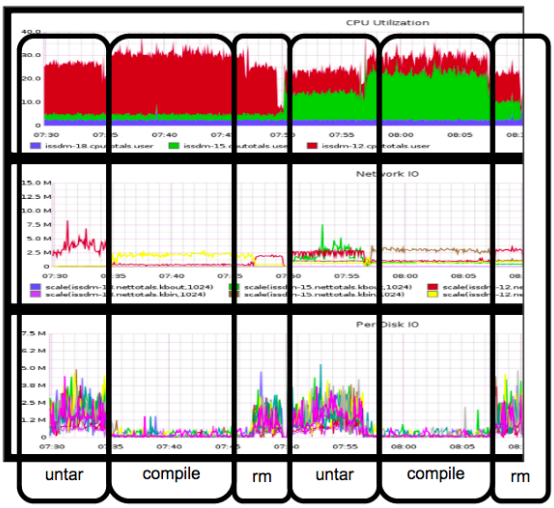


Figure 2: Create-heavy workloads (untar) incurr the highest disk, network, and CPU utilization because the metadata server is managing consistency and fault tolerance.

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. Our system supports 3 forms of consistency and 2 forms of fault tolerance giving the administrator a wide range of policies and optimizations depending on the application's needs.

2 POSIX OVERHEADS

In our examiniation of the overheads of POSIX we benchmark and analyze CephFS, the file system that uses the RADOS object store to store its data and metadata. We choose CephFS because it is an open-source production quality system. CephFS made one set of design decisions and we not asserting that the design decisions that were made are superior but instead highlight the effect those decisions have on performance.

To show how CephFS behaves under high metadata load we use a create-heavy workload because these types of workloads incurr high CPU, network, and disk usage. Figure 2 shows a compilation of the Linux kernel, which has a download, untar, compile, and remove phase. The untar phase is characterized by many creates and has the highest resource usage, indicating that it is stressing the consistency and journalling subsystems of the metadata server. Also of note: a create-heavy workload does not help for caching indoes.

In this section, we quantify the costs of consistency and fault tolerance in CephFS. If the components that ensure these semantics (i.e. capabilities and journals, respectively) can be mitigated or delayed, then global namespaces perform as well as decoupled namespaces. We run our experiments on a 9 OSD, 3 MDS, 1 MON Ceph cluster. The clients use the Ceph kernel client, which has been in the mainline Linux kernel since TODO. 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. We also turn caching off becuase, as shown in figureX there is little difference, in terms of performance between caching and not caching when using the kernel client.

2.1 Fault Tolerance

As shown in Figure 3 CephFS uses RADOS (1) as a metadata store for all information about files including the hierarchical namespace and (2) as a staging area for the journal of updates before they are applied to the metadata store. (1) is essentially a cache that improves performance by serving metadata from memory and (2) is the mechanisms for achieving fault tolerance.

Fault tolerance means that the client or server can fail and metadata will not be lost. CephFS addresses fault tolerance with a metadata journal that streams into the resilient object store. Similar to LFS [] and WAFL [] the metadata journal can grow to large sizes ensuring (1) sequential writes into RADOS 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. The former is memory bound as larger segments take up more memory but can reduce the time spent journaling and the latter is CPU bound.

Figure 4 shows that journaling metadata updates into the object store has an overhead. Part (a) shows the runtime for different journal segment sizes; the larger the segment size the bigger that the writes into the object store are. The trade-off comes is in terms of memory because larger segment sizes take up more space with their buffers. Parts (b), (c), and (d) show the throughput over time for different segment sizes. Performance suffers when time is spent journaling.

Despite this overhead, we posit that the journal is sufficient to slow down metadata throughput but not so much as to overwhelm RADOS because 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 on 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, there is no ground truth the requires the failed namespace to answer to anyone. On the server side, BatchFS and DeltaFS use IndexFS. Again, IndexFS writes metadata to SSTables

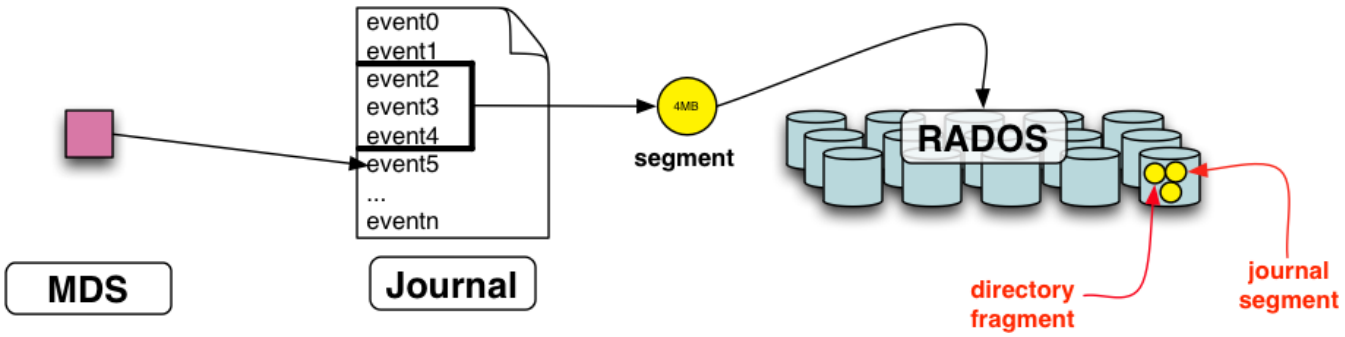


Figure 3: CephFS uses a journal to stage updates and tracks dirty metadata in the collective memory of the MDSs. Each MDS maintains its own journal, which is broken up into 4MB segments. These segments are pushed into RADOS and deleted when that particular segment is trimmed from the end of the log.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

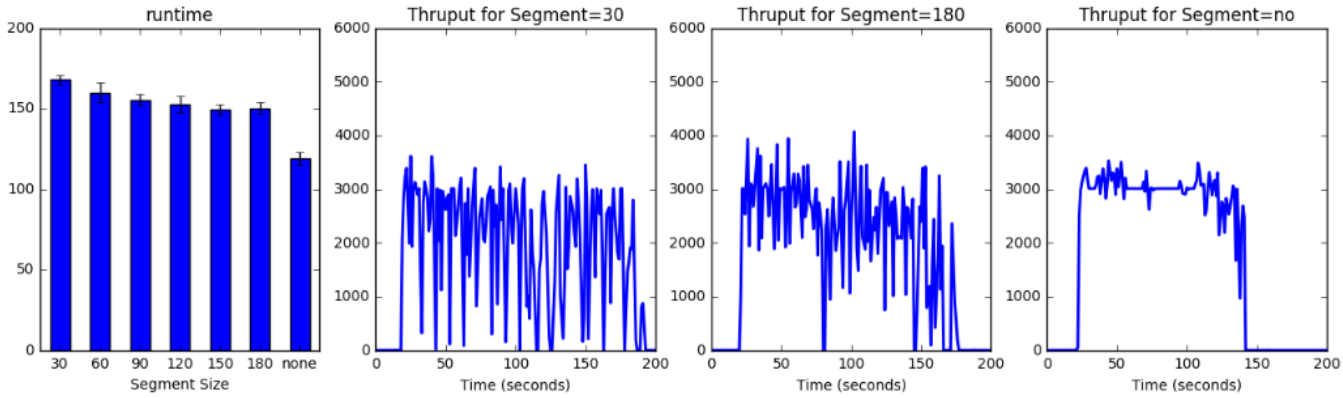


Figure 4: Performance improves with larger journal segments because the metadata server spends less time flushing the journal

but it is not clear whether they ever vacate memory, get written to disk, or are flushed to the object store.

2.2 Strong Consistency

Access to POSIX metadata is strongly consistent, so reads and writes are globally ordered. The synchronization and serialization machinery needed to ensure that all clients see the same state has high overhead. CephFS uses capabilities to keep metadata strongly consistent. To reduce the number of RPCs needed for consistency, clients can obtain capabilities for reading, reading and updating, reads caching, 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 multiple RPCs to the metadata server to (1) determine if the file exists and (2) do the actual create. Unless the client immediately reads all the inodes in the cache, the inode cache is less useful for create-heavy workloads because the cached inodes are unused.

The benefits of caching the directory inode when creating files is shown in Figure 7(a). If only one client is creating files in a directory ("isolated" curve) 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) then the directory inode transitions out of read caching and the first client must send lookups to the metadata server. When other clients interfere the request throughput is higher Figure 7(b) but the runtime is slower because the isolated client scenario incurrs less requests. Figures 7(c) and (d) plot the creates and lookups, repsectively, over time; creates slow down and lookups dominate the request load when the directory inode is shared in the interferring client scenario.

The drawbacks of the inode cache are shown in Figure **??**. Figure **??**(a) shows the throughput in metadata requests per second for a single client creating 200 thousand files. Until

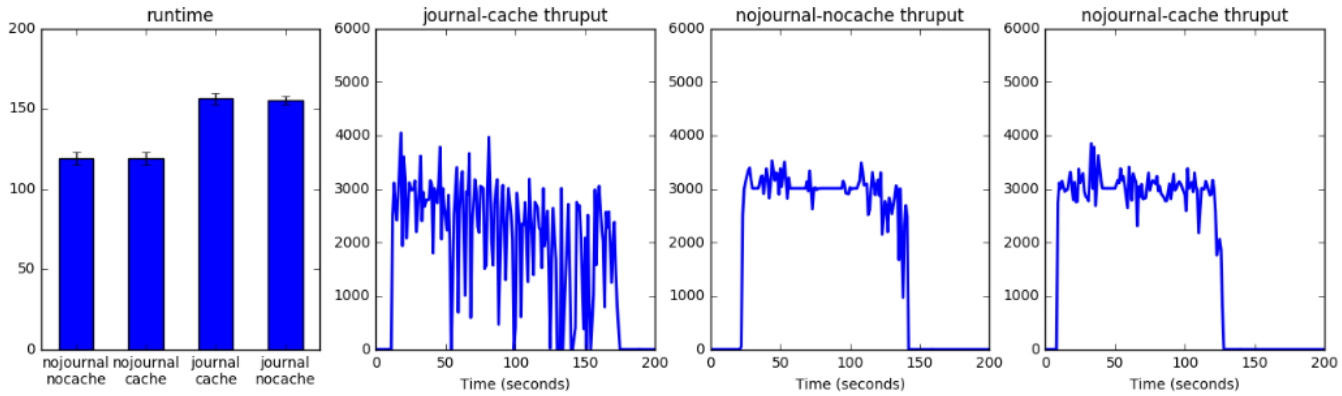


Figure 5: Journalling metadata updates has a bigger overhead than maintaining the inode cache. For create-heavy workloads the inode cache offers no performance benefits.

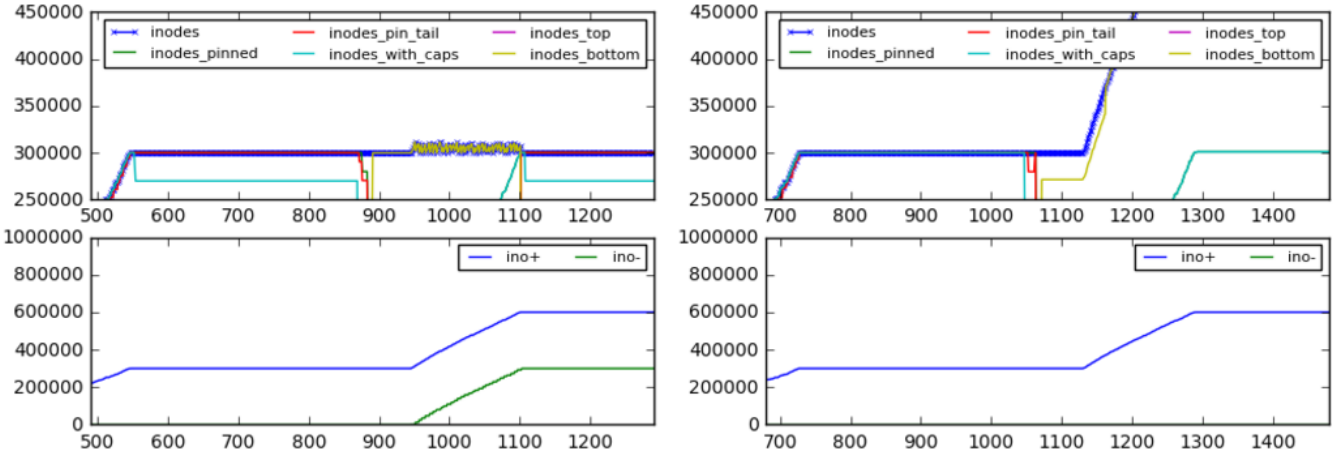


Figure 6: The inode cache improves metadata read performance but for our create-heavy workload it is only an overhead. Most of the time maintaining the cache is spent evicting and adding inodes.

time 950 the throughput is steady at just under 2000 ops/sec and the CPU utilization is at about 30%. At time 950 seconds throughput degrades which corresponds to more inodes in the cache (Figure **??**(c)). The values in Figure **??**(c) are:

- inodes: total number of elemnts in the cache
- inodes pinned:
- inodes pinned tail
- inodes with caps
- inodes top
- inodes bottom

CephFS tries to keep the cache at 100 thousand inodes so the degradation in performance indicates that the metadata server cannot keep up with the workload. Figure **??**(d) shows inodes getting added at the same rate as before 950 seconds (ino+) but the rate at which inodes get removed is less stable (ino−) indicating that cache eviction is taking more of the metadata servers time.

2.3 Consistency Overhead

Figure 8 is a baseline showing that the metadata server can service multiple clients when it is underloaded. One client creates files in the same directory and another does a touch or stat 15 seconds into the run. Figure 8(a) shows the runtime of the client doing the creates: "isolated" is when the create client is the only workload in the cluster, "interfere stat" is the runtime of the create client when another client does a stat, and "interfere touch" is the runtime of the create client when another client does a touch. There is only a minor performance degradation. Figure 8(b) compares the baseline the other client doing the touch or stat To explain Figure 8(a) we use Figure **??**(b) to show why the runtime of creating 100

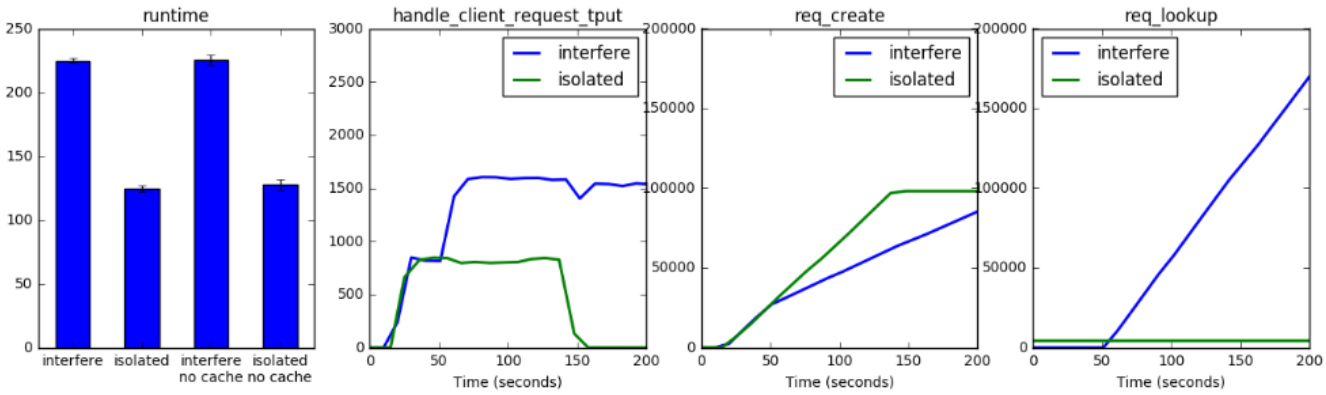


Figure 7: When a client create stream is "isolated" then lookups resolve locally but when a second client "interferes" by creating in the same directory, the directory inode capability is revoked forcing all clients to centralize lookups at the metadata server.

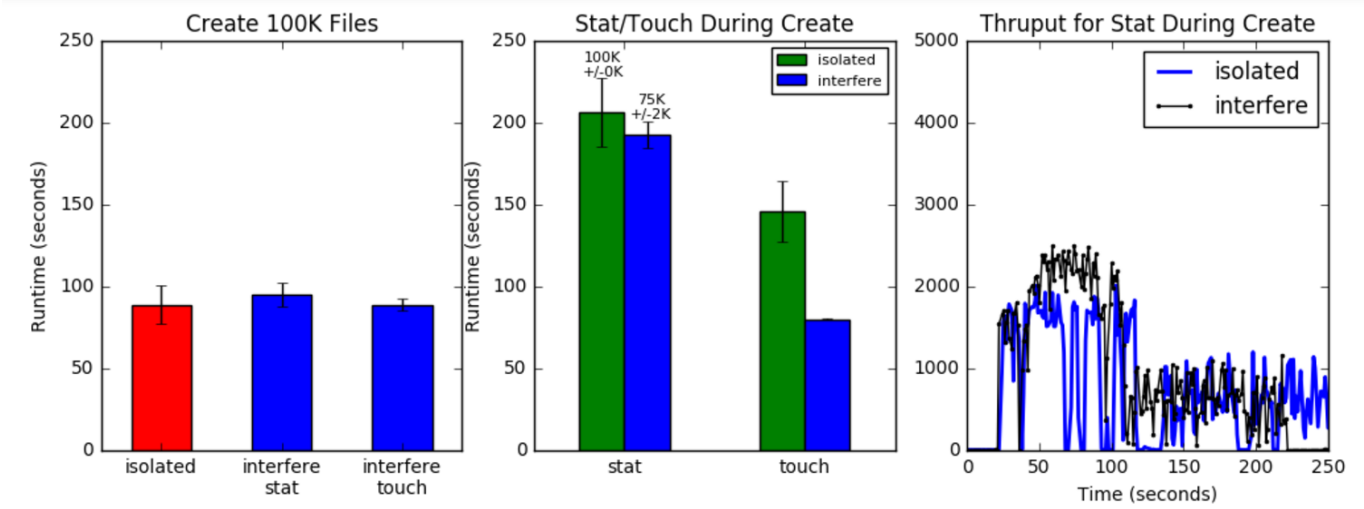


Figure 8: An underloaded metadata server adaquately services conflicting clients; the create speeds are similar while the interferring operations are limited by the cost of RPCs.

thousand files is unaffected. Compared to the throughput without an interferring stat ("isolated" curve) the throughput of the metadata server with an interferring stat ("inteferring stat" curve) is higher. This means the metadata server is doing more operations suggesting that it can adequately handle the demands of both workloads.

To explain Figure 8(b) we use Figure 8(a) to show that operations are bounded by the number of files; more files incurr more requests since both operations lookup every file. The number of files shown in Figure **??**(a) indicates that the local stat ("client 0" bar) is interacting with less files than the remote stat ("client 1" bar). This graph also explains why the isolated stats and touches take the longest from the remote clients; out of all setups, the isolated operations are doing the most RPCs. The difference between local ("client 0 interfere" bar) and remote ("client 1 interfere" bar) in Figure 8(b) is the overhead of doing RPCs. Isolated is the slowest because it requests the most files. On average the local interferring client ("client 0" bar in Figure **??**(a)) requests 40 thousand less files than the remote interferring call ("client 1" bar in Figure **??**(a)). This reflects how fast the local client can get into the queue of requests, presumably because it bypasses capability checks.

From this experiment we make 2 conclusions: (1) the metadata server is not overloaded because it can handle both workloads and (2) the metadata server needs to get global consistency from only 1 client

Comparison to decoupled namespaces: Decoupled namespaces merge batches of metadata operations into the global namespaces when the job completes. In BatchFS the

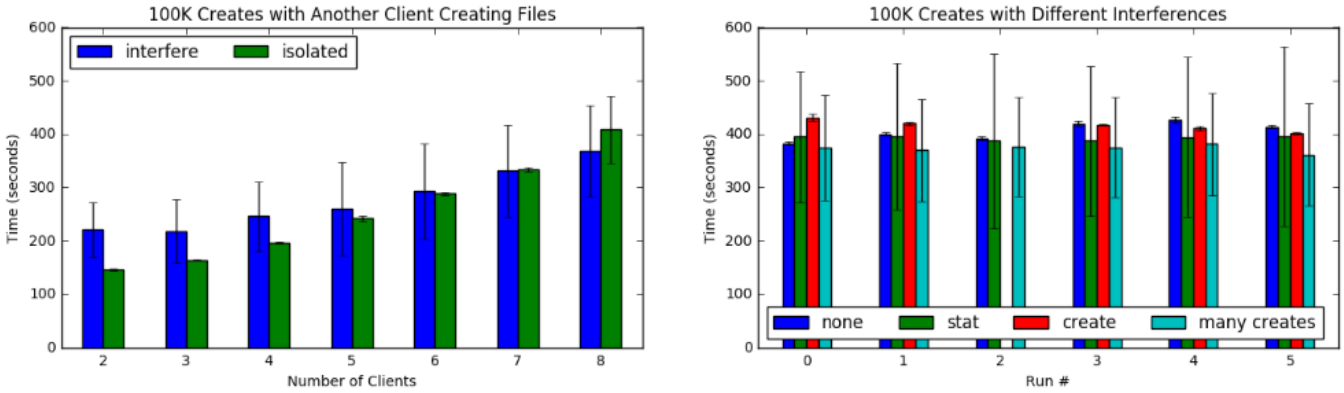


Figure 9: Scaling clients shows increased variability when another client interferes; zooming in on runs with 7 clients we see that different types of interferring operations have different effects on performance variability and predictability.

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.

3 METHODOLOGY: DECOUPLED NAMESPACES

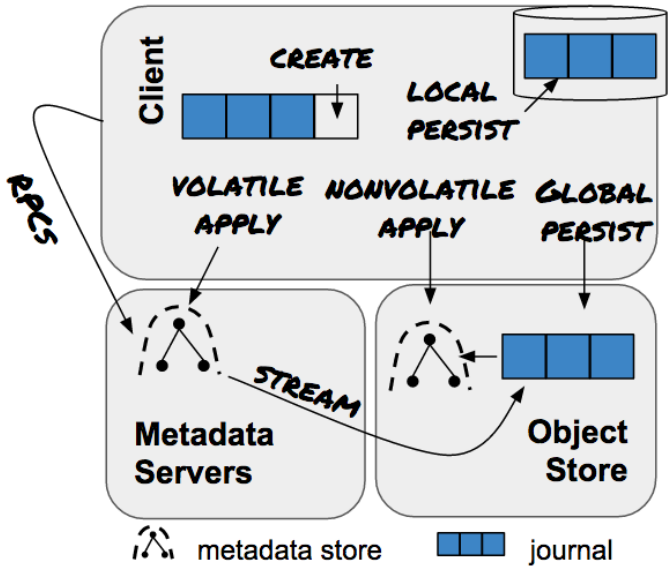
In this section we describe Cudele, our prototype system that lets future programmers compose mechanisms (Section §3.1) to provide the necessary guarantees (Section §3.2) for their application.

3.1 Cudele's Mechanisms

Figure 10 shows the mechanisms (labeled arrows) in Cudele and which entity they are performed by (gray boxes). The metadata store and journal are different ways of representing the namespace. The metadata store represents the namespace as a tree of directory fragments and is easier to read/traverse. On the other hand, the journal represent the namespace as a list of events. It a "pile system"; writes are fast but reads are slow because state must be reconstructed. Specifically, reads are slow because there is more data to read, it is unorganized, and many of the updates may be redundant.

Cudele presents 6 mechanisms: RPCs, Stream, Create, Volatile Apply, Save, and Persist. "RPCs" does round trip remote procedure calls to establish consistency; it is the default implementation for complying with POSIX in CephFS. "Stream" has the metadata servers stream a journal of metadata updates into the object store. "Create" allows clients to append metadata events to an in-memory journal. "Volatile apply" takes the in-memory journal on the client and applies

Figure 10: Applications can decouple the namespace, write updates to a local journal, and delay metadata updates. Table 1 shows how these phases (represented by the arrows) can be combined to provide weaker consistency or fault tolerance semantics.



it directly to the in-memory metadata store of the metadata server cluster. "Save" takes the in-memory journal and writes it to the client's disk. "Persist" saves the journal as a an object in the object store from the client.

Next, we discuss how these mechanisms can be composed to get different consistency and fault tolerance semantics.

Table 1: Future programmers can explore the consistency (C) and durability (D) spectrums by composing Cudele mechanisms. The consistency and durability properties are not guaranteed until all mechanisms in the cell are complete (i.e. the compositions should be considered atomic) and there are no guarantees while tranisitioning between policies.

C → D ↓	none	eventual	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 Setting Policies with Cudele

The spectrum of consistency and fault tolerance guarantees that adminstrators can construct is shown in Table 1. The columns are the different consistency semantics and the rows cover the spectrum of fault tolerance guantees. For consistency: "none" means the system does not handle merging updates into a global namespace and it is assumed that middleware or the application manages consistency lazily; "eventual" merges updates either when the system has time (e.g., by a background daemon) or when the client is done writing; and updates in "global" consistency are seen immediately by all clients. For fault tolerance, "none" means that updates are volatile and will be lost on a crash of any component. 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.

The cells in Table 1 encompass many exisiting storage systems. POSIX systems like CephFS and IndexFS have global consistency and fault tolerance; DeltaFS has consistency and fault tolerance set to "none"; and BatchFS uses "eventual" consistency and "local" fault tolerance. These are just a few of the HPC examples.

To compose the mechanisms administrators inject which steps (described in Section §3.1) to run and which to use in parallel using a domain specific language. For example, to get the semantics of BatchFS, the administrator would inject the following pipeline:

```
create+save+volatile apply
```

Although we can achieve all permutations of the different guarantees in Table 1, not all of them make much sense. For example, it makes little sense to do `creates+RPCs` since both steps do the same thing or `stream+save` since global fault tolerance is stronger and has more overhead than local fault tolerance. Formulaically, valid steps include:

```
<create|RPCs> [+v_apply][+save][+persist][+stream]
```

3.3 Cudele Namespace API: Per-Subtree Policies

To assign consistency and fault tolerance to the subtrees we store 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 data or 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.

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.txt`. 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 `block`)
- `consistency_callback`: which consistency model to use (default `RPCs`)
- `durability_callback`: which durability model to use (default `stream`)

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. Table **??** shows how one would use the Cudele API to implement policies from related work.

For `block`, any requests to this part of the namespace return with "Device is busy", which will spare the MDS 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 because the resutls are more accurate.

4 IMPLEMENTATION

Each section below corresponds ot the labeled arrows in Figure 10. This implementation decouples policies from mechanisms allowing applciations to choose the consistency and fault tolerance semantics they need.

Of the 6 mechanisms in Figure 10 4 had to implemented and only 1 required changes to the underlying storage system itself. RPCs and stream can be achieved with tunables in configuration files for the storage system (e.g., metadata cache size, logging on/off, etc.). Persist", save, and create are

implemented as library and does not require modifications to the storage system. Volatile apply requires changes to the metadata server to inject updates back into the global namespace.

4.1 No Changes: RPCs, Stream

RPCs is the default behavior of the storage system. Depending on the cache state and configuration each operation incurrs at least RPC. 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.

Re-using the debugging and testing features in the storage system, 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. Performance numbers for enabling and disabling the stream phase are presented back in Section §2.1.

The journal segments are saved as objects in RADOS. The journal has 4 pointers, described in 'osdc/Journaler.h':

- write position: tail of the journal; points to the current session where we are appending events
- unused field: where someone is reading
- expire position: old journal segments
- trimmed position: where daemon is expiring old items

Journal segments in RADDS have a header followed by serialized log events. The log events are read by hopping over objects using the read offset and object size pulled from the journal header. After decoding them, we can examine the metadata (1) about the event (e.g., type, timestamps, etc.) and (2) for inodes that the event touches.

The metadata for inodes that the event touches are called metadata blobs and the ones associated with events are **unordered**; this layout makes writing journal updates fast but the cost is realized when reading the metadata blobs. It makes sense to optimize for writing since reading only occurs on failures. To reconstruct the namespace for the metadata blob, the journal tool iterates over each metadata blob in the events and builds mappings of inodes to directory entries (for directories) and parent inodes to child inodes/directory entries.

4.2 External Library: Create, Save, Persist

For "create", "save", and "persist" Cudele provides a library for clients to link into. Recall that CephFS uses the object store (1) as a metadata store for all information about files including the hierarchical namespace and (2) as a staging area for the journal of updates before they are applied to the metadata store. As discussed previously, the metadata store is optimized for reading while the journal is optimized for writing. Cudele re-uses the journal tool to read/write log events to memory and persistent storage.

The journal tool is used for disaster recovery and lets administrators view and modify the journal of metadata updates; it can read the journal, erase events from the journal, and apply the updates in the journal to the metadata store. To apply journal updates to the metadata store, the journal tool reads the journal segments from object store objects and applies the update to the metadata store (which are also stored as object store objects). For create, clients write to an in-memory journal and updates are merged into the global namespace by replaying them onto the metadata store in the object store.

The journal tool imports journals from binary files stored on disk. First the header of the dump is sanity checked and written into RADOS to the "header" object. The "header" object has metadata about the journal as well as the locations of all the journal pointers (e.g., where the tail of the journal is, where we are currently trimming, etc.). Then the journal events are cleaned (erasing trailing events that are not part of the header) and written as objects into RADOS. Note that while the journal is in RADOS, the metadata servers do do not have the namespace reconstructed in memory so the metadata cluster will not service requests relating to the journal of imported events. To construct the namespace in the collective memory of the metadata servers we need to first construct the namespace in RADOS. The journal tool can explicitly do this by applying the journal to the metadata store in RAODS. This will pull the objects containing journal segments and replay them on the metadata store. Finally, we delete the journal in RADOS and restart the metadata servers so they rebuild their caches.

The journal tool exports journals to binary files stored on disk. First the journal is scanned for the header and then journal is recovered. To recover the journal the "header" object is read off disk and then objects are probed in order and starting from the write position saved in the header. Probing will update the write position if it finds objects with data in them.

When exporting a journal of events, the journal tool first scans the journal to check for corruption. Then it recovers the journal by reading the "header" object out of RADOS. After reading the header, the journal tool can pull journal segments from RADOS because it knows how many objects to pull and how far to seek within those objects.

For save, clients write serialized log events to a file on local disk and for persist, clients store the file in the objects store. The overheads for save and persist is the write bandwidth of the local disk and object store, respectively.

This required no changes to Ceph because the metadata server knows how to read the events we were writing into the object store. The client writes metadata updates locally and merges the updates with the journal tool. The client that decouples the namespace operates without any consistency and any conflicts at the merge are resolved in favor of this client. Updates by other clients (*i.e.* metadata writes to the global namespace) are overwritten. We leverage the journal tool's ability to reconstruct metadata events in memory. The client library is shown in Figure 10. Cudele adopts the following process when an application decouples the namespace:

(1) "**decoupled**": metadata server exports the journal events to a file
(2) "**transfer**": the file is pulled by the client from the metadata server
(3) "**snapshot**": client reads the file and materializes a snapshot of the namespace in memory

By using 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), and the functions for replaying events onto the internal namespace data structures.

4.3 Storage System Changes: Volatile Apply

The volatile apply mechanism (*i.e.* the "v_apply" arrow in Figure 10) 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 Cudele makes no consistency or fault tolerance guarantees. 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. Relaxing these constraints gives volatile apply the best performance.

In contrast, apply uses the the object store to merge the journal of updates from the client to the metadata server. Apply is safer than volatile apply but has a performance overhead because objects in the metadata store need to be read from and written back to the object store. Apply was already implemented by the journal tool: it reads the journal from the object store, adds events to the journal, and writes the metadata updates out to the metadata store in the object store. Cudele's volatile apply executes "save" (described in §4.2), transfers the file of metadata updates to the metadata server, and then merges the updates directly into the in-memory metadata store of the metadata cluster.

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 RESULTS

5.1 Microbenchmarks

5.1.1 *Per phase latencies.*

5.2 Journaling Overhead

Journal to RADOS Turn off journaling (large segment) Journal to in-memory OSD

5.3 Macrobenchmarks

updatedb: http://lists.ceph.com/pipermail/ceph-users-ceph.com/2015-July/002768.html

REFERENCES

[1] 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)*.

[2] 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)*.

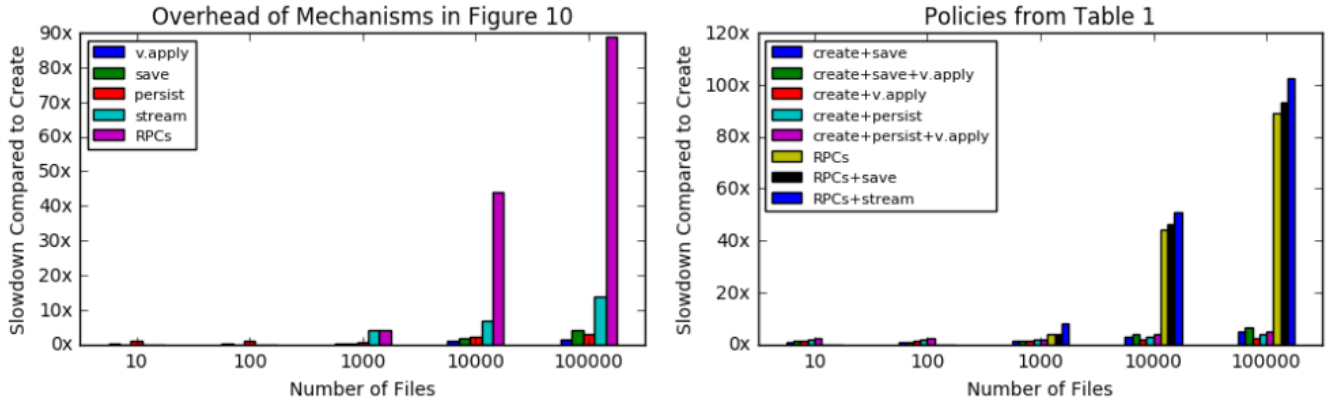


Figure 11: Here's a graph.

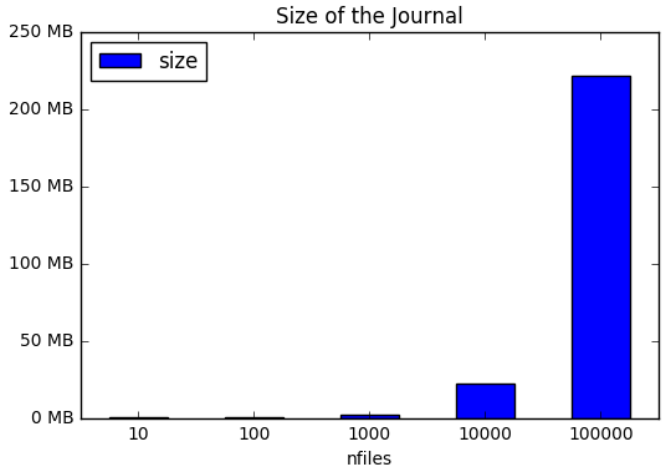


Figure 12: Here's a graph.

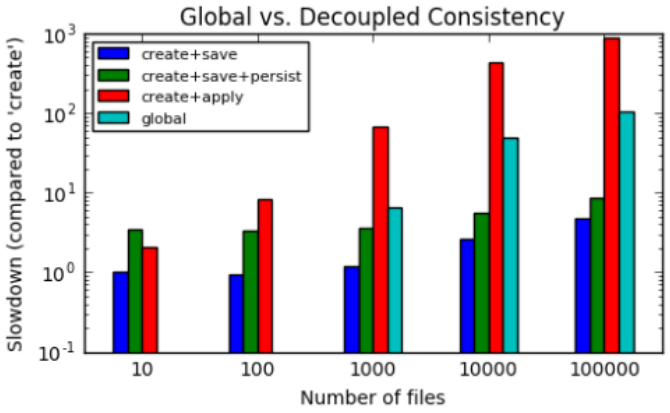


Figure 13: Compared to the `create` phase, saving and persisting updates (`create+save` and `create+save+persist`) experience only a 4.79× and 8.66× slowdown, in the worst case for 100K files. In contrast, maintaining `global` consistency is 905.70× slower. The disadvantage of decoupling the namespace is the merge phase where updates are applied to the metadata store (`create+apply`, resulting in a 905.70× slowdown for 100K files.