Global Namespaces Scale as Well as Decoupled Namespaces

Paper ID: 165

Total Pages: 12

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. In this paper we examine a general purpose file system and describe the overheads of the design decisions that were made; namely strong consistency and fault tolerance. Then, using Cudele, we relax those constraints on specific subtrees in the global namespace and show improved performance without sacrificing the strong consistency needs of other users.

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 [4, 5]. 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 [2]. This trend exacerbates the metadata scalability problem and has given rise to the serverless metadata services.

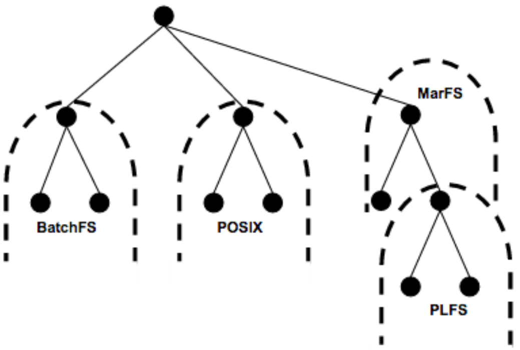

Figure 1. Administrators can assign consistency and fault tolerance policies to subtrees to get the benefits of some of the state-of-the-art HPC architectures.

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 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. Related Work

Decoupling the namespaces has many advantages, including improved scalability, higher resource utilization, and better performance. BatchFS [7] and DeltaFS [8] are near-POSIX filesystems that give clients the ability to decouple subtrees

	decoupled [7, 8]	global [4, 6]	implied [1]
consistency	eventual	capabilities	naming
fault tol.	local	journal	journal

Table 1. Each technique for handling the namespace has its trade-offs. Decoupled namespaces have the highest throughput because they have weaker consistency and fault tolerance semantics.

from the namespace so that the applications can execute metadata operations without synchronization and serialization. These operations are applied to a local snapshot of the file system namespace and conflicts are resolved either by the application or by an external service . Applications link into a metadata server library to reduce resource utilization and code paths (e.g., no daemons and less interprocess communication). Unfortunately, decoupling the namespaces has costs: (1) merging metadata state back into the global namespace is slow; (2) failures are local to the failing node; and (3) the systems are not backwards compatible.

For (1), state-of-the-art systems manage consistency in non-traditional ways: IndexFS maintains the global namespace but blocks operations from other clients until the first client drops the lease, BatchFS does operations on a snapshot of the namespace and merges batches of operations into the global namespace, and DeltaFS never merges back into the global namespace. The merging for BatchFS is done by an auxiliary metadata server running on the client and conflicts are resolved by the application. Although DeltaFS never explicitly merges, applications needing some degree of ground truth can either manage consistency thesmelves on a read or add a bolt-on service to manage the consistency.

For (2), if the client fails and stays down, all metadata operations on the decoupled namespace are lost. If the client recovers, the on-disk structures (for BatchFS and DeltaFS this is the SSTables used in TableFS [3]) can be recovered. In other words, the clients have state that cannot be recovered if the node stays failed and any progress made 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.

Decoupling the namespace delays metadata consistency and sacrifices fault tolerance. As shown in Table 1, metadata consistency is provided by capabilities and fault tolerance is addressed with a journal.

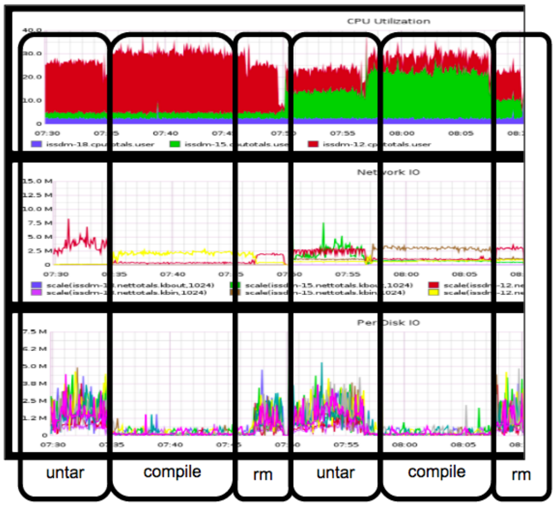

Figure 2. Create-heavy workloads (untar) incurr the highest disk, network, and CPU utilization because the metadata server is managing consistency and fault tolerance.

3. 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 inodes.

3.1 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.

Figure 3 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

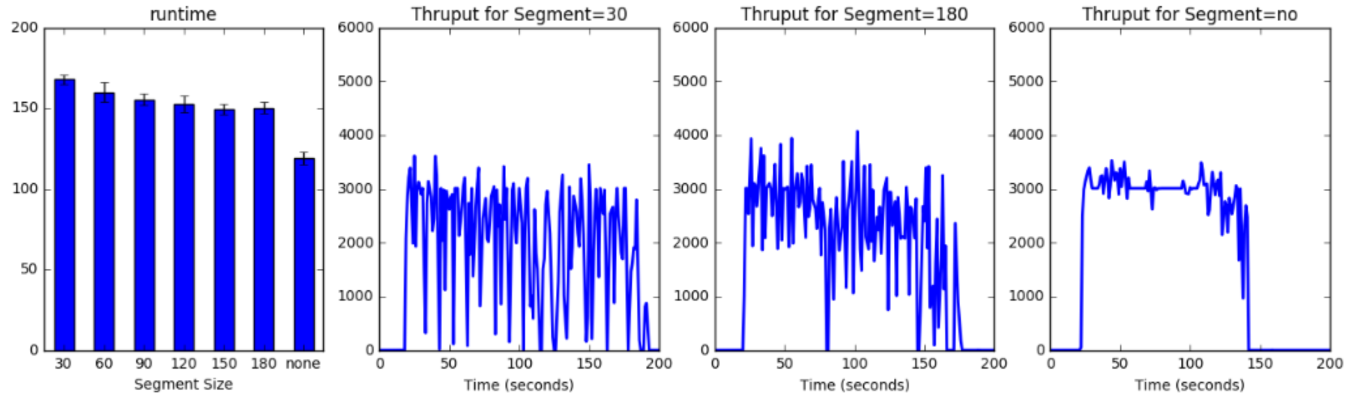

Figure 3. Performance improves with larger journal segments because the metadata server spends less time flushing the journal

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

3.2 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, 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 6(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 6(b) but the runtime is slower because the isolated client scenario incurrs less requests. Figures 6(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 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

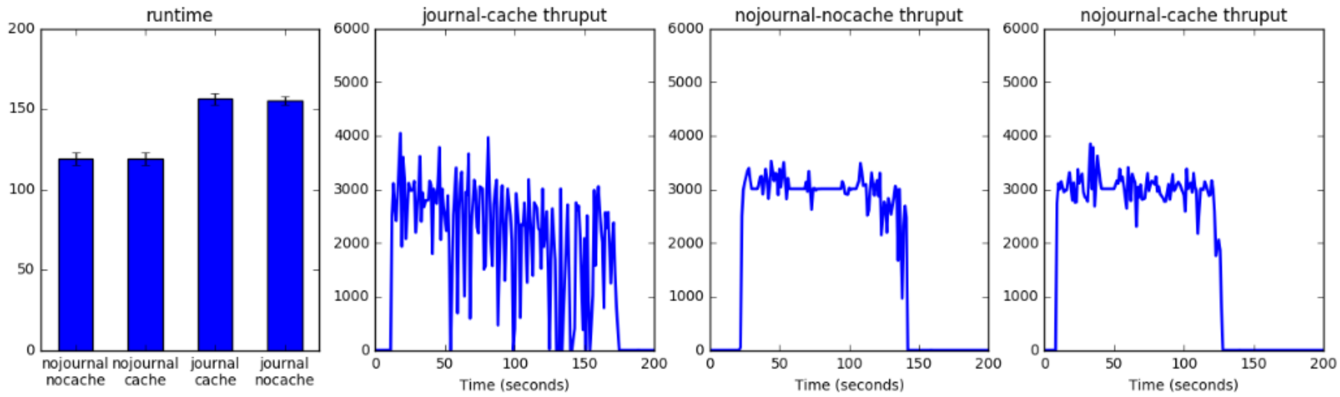

Figure 4. Journalling metadata updates has a bigger overhead than maintaining the inode cache. For create-heavy workloads the inode cache offers no performance benefits.

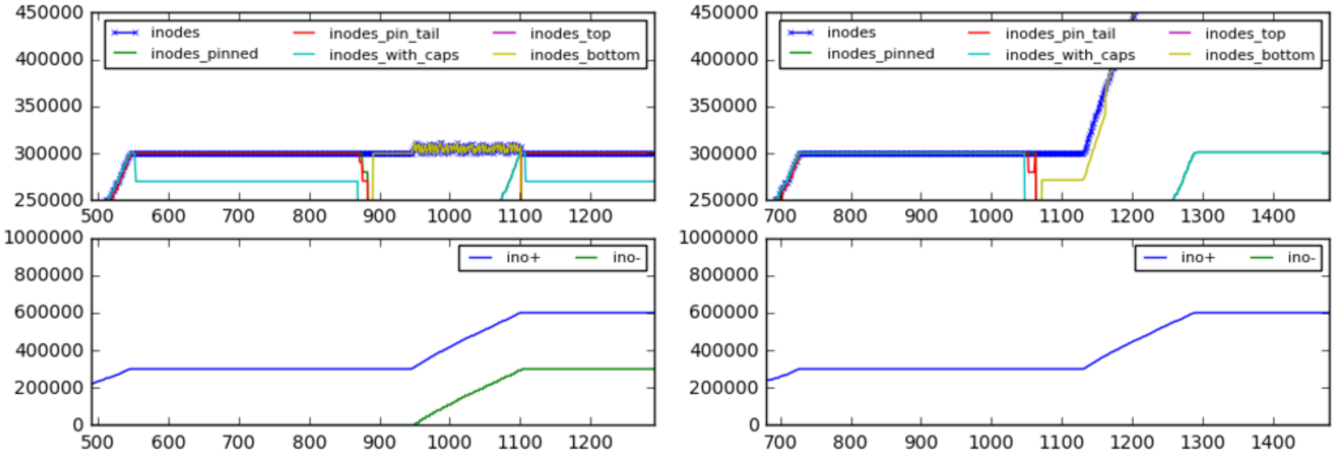

Figure 5. The inode cache improves metadata read performance but for our create-heavy workload it is only an overhead. Most of the time maintaining the cache is spent evicting and adding inodes.

(ino−) indicating that cache eviction is taking more of the metadata servers time.

3.3 Consistency Overhead

Figure 7 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 7(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 7(b) compares the baseline the other client doing the touch or stat To explain Figure 7(a) we use Figure **??**(b) to show why the runtime of creating 100 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 7(b) we use Figure 7(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 7(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

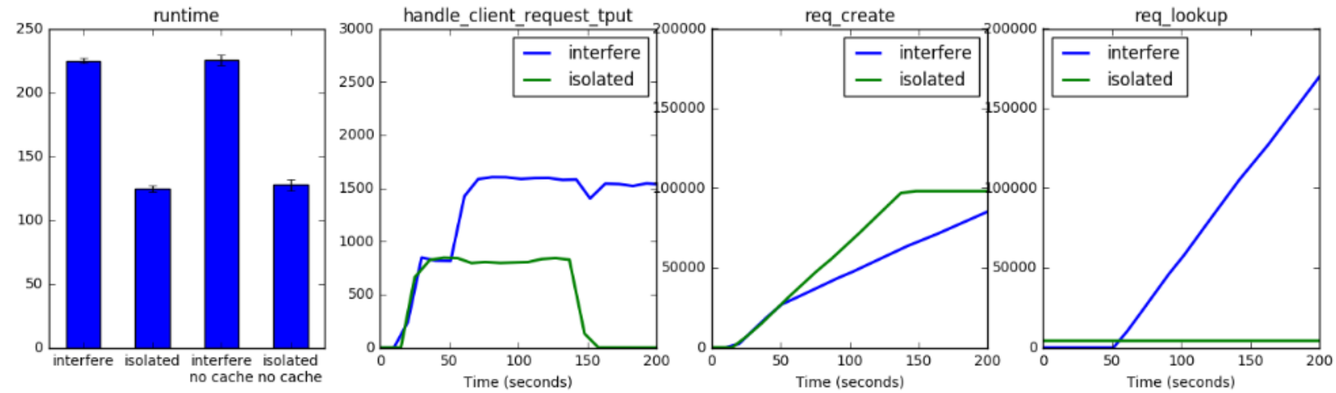

Figure 6. When a client create stream is "isolated" then lookups resolve locally but when a second client "interferes" by creating in the same directory, the directory inode capability is revoked forcing all clients to centralize lookups at the metadata server.

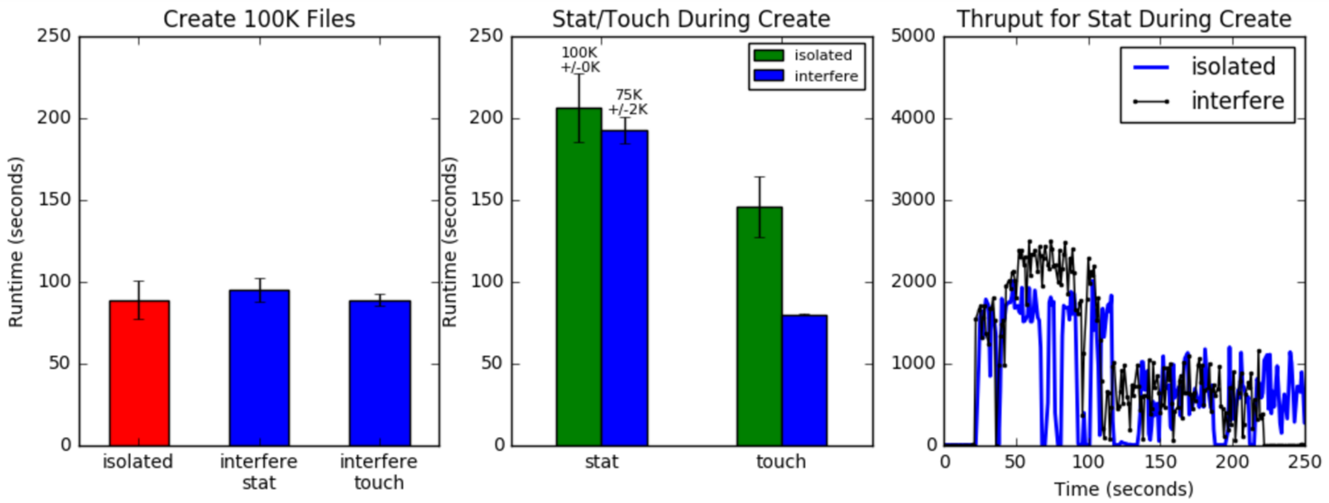

Figure 7. An underloaded metadata server adaquately services conflicting clients; the create speeds are similar while the interferring operations are limited by the cost of RPCs.

thousand 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:

- the metadata server is not overloaded because it can handle both workloads
- 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 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.

4. Methodology

4.1 Decoupled Namespaces in CephFS

Recall that 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. 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

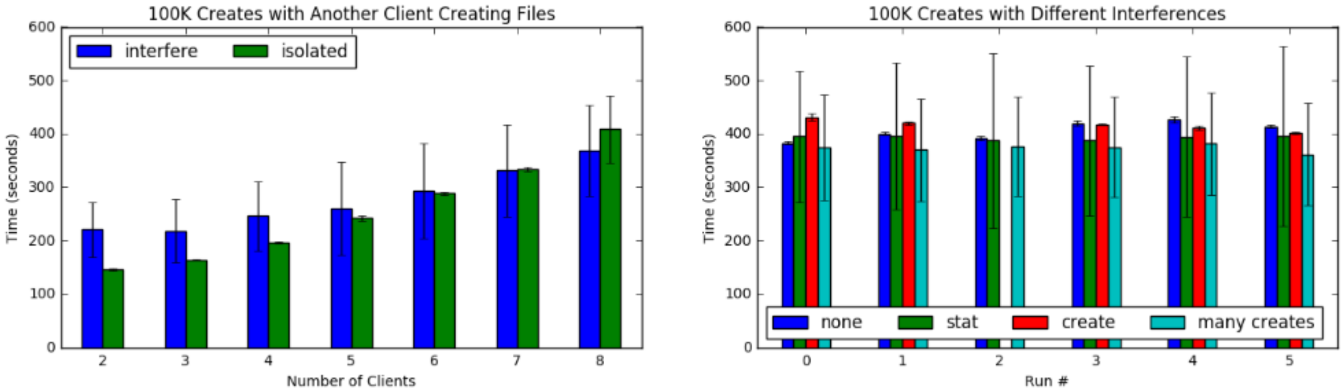

Figure 8. Scaling clients shows increased variability when another client interferes; zooming in on runs iwth 7 clients we see that different types of interferring operations have different effects on performance variability and predictability.

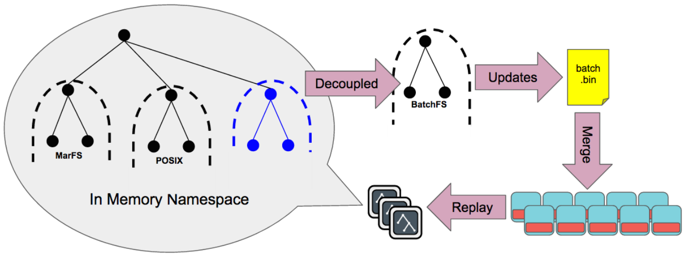

Figure 9. Applications that can tolerate weaker consistency can decouple the namespace, write updates to a local file, and delay metadata updates. The merge and replay steps are optional if there is no need for a global namespace.

the journal to the metadata store. To apply journal updates to the metadata store, the journal tool reads the journal segments from RADOS objects and applies the update to the metadata store (which are also stored as RADOS objects).

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.

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. There are two types of objects:

1. journal objects (e.g., 200.000*)
2. metadata store objects (e.g., 1.inode)

We need to update the journal objects with the new events! How do we write out new objects to RADOS (e.g., 200.000*)? These should be the corect sizes (e.g., 4980 bytes) – these log events that we are adding are not making them into the journal objects... do we have to explicitly read them and then re-write them?

TODO: pull the object, read it, append to it, make sure size changes. It looks like we can only read these objects and apply doesn't actually touch them... it only touches the metadata store objects (e.g., 1.inode*).

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.

4.1.1 Leveraging the Journal Tool

We leverage the journal tool's ability to reconstruct the namespace for the metadata blob. First we scan journal and store all events in a map. Next we create a new event and copy the root inode directory lump from another journal event into the event's metadata blob. Finally, we add a new directory lump for our new file or directory.

If decoupled namespaces are turned on, 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. Cudele adopts the following process when an application decouples the namespace:

1. metadata server exports a snapshot of the file system to a binary file
2. client materializes the snapshot in memory
3. client operates on the data structures in memory
4. client exports the new snapshot to a binary file
5. metadata server merges the snapshot into the global namespace

Step 3 is the most complicated and requires understanding how the snapshot is materialized in memory.

4.1.2 Operating on Snapshots

Our first implementation attempted to re-create journal events using the same libraries that the metadata server uses. To construct a `mkdir` we tried to instantiate a Ceph inode and directory entry for the current file/dir and its parent. This is too hard because there are too many moving parts in the metadata server (e.g., a mdlog class, stuff in memory, assumption that we can traverse up and down namespace, etc.). So when I tried add dentries and inodes it was trying to traverse up/down and it would almost always segfault when it was looking for something. These metablobs are supposed to be self container – the problem is I do not know what is supposed to go *inside* them.

Our second idea was to copy the metadata blog and change just what we needed. For example, we would save a binary dump of a generic `mkdir` event on disk. When the application makes a directory, this dump would be loaded and the fields would be changed before being written back to disk. Rather than traversing up and down a namespace in memory of a metadata server, we should traverse up and down the namespace *inside* the metadata blob. This implementation requires disk IO and editing the log event is non-trivial for two reasons:

- methods do not edit events; they just write them
- the metadata that the event touches (e.g., the metablob) is unorganized on disk for performance – it is trade-off for writing data faster serially and reconstructing information slowly since failure is not considered the norm

Faced with these challenges we landed on our final implementation: load the snapshot into the the data structures used to examine and replay journals, edit those data structures, and write them out to disk as binary.

5. notes

Linking clients into our custom libcephfs

Use namespace's recursive data structure to put policies on subtrees - consistency: eventual vs. strong, global vs. local - e.g., BatchFS/DeltaFS: eventual, local - e.g., POSIX: strong, global - e.g., PLFS: no consistency - fault tolerance: global vs. local - e.g., CephFS: global - e.g., BatchFS/DeltaFS: local

Experimental Setup - Ceph: 9 OSDs, 1 MDS, 2 kernel client - Workload limitations: blah

Workload: creates

Baseline: 200K creates in the same directory - throughput: degrades at 950s - CPU utilization: more at 950s - inode cache: eviction dominate - inodes +- to cache: eviction dominate - per-disk throughput: RADOS not bottleneck

Experiment 1: Interference

6. Experiments

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.

Experimental Setup: 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.

6.1 Baseline

Experiment 0: creates in the same directory - setup: why we use caching, we use the kernel client, how we circumvent max fragment size

Experiment 0: creates with a stat - Hypothesis: metadata read pauses creates and requires a snapshot in time - what is more of an overhead: pausing creates and getting a consistent view OR sucking up resources as it reads from RADOS? - can we delay snapshot?

Experiment 1: creates with a readdir - Hypothesis: shows the cost of synchronization because on a write, the first client drops his caps - client0: create 100k, client1: stat at 2 mins

Experiment 2: scale the number of files - See if the open/close spike occurs - Try to see why open/close spike is allowed to happen - Try to disable all caching – metadata writes don't ever re-use the inode – we never ask for it again! - client0: create 100k, client1: touch at 2 mins

Experiment 3: see how fast the cache satisfies a read - client0: create 100k, stat inodes - client0: create 100k, client1: stat inodes

lient 0: creates, client 1 create(s)

6.2 Journaling Overhead

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

6.3 Macrobenchmarks

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

References

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

[2] John Bent, Brad Settlemyer, and Gary Grider. Serving Data to the Lunatic Fringe.

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

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

[5] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 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, 2015.

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

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

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