UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SCALABLE, GLOBAL NAMESPACES WITH PROGRAMMABLE STORAGE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DEFENSE

in

COMPUTER SCIENCE

by

Michael A. Sevilla

June 2014

The Dissertation of Michael A. Sevilla
is approved:

Professor Scott Brandt, Chair

Professor Carlos Maltzahn

Professor Ike Nassi

Dr. Sam Fineberg

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Scalable, Global Namespaces with Programmable Storage

by

Michael A. Sevilla

lorem ipsum

Chapter 1

Introduction

Systems that process and store large amounts of data (petabytes and beyond) are difficult to manage. Computing has reached an era where the data is too large, the software is too complicated, the hardware is too fast, and the events are too frequent for any human to manage. These drastic changes should alter how large systems are designed and deployed. We argue that the most elegant and future-proof solution for improving and maintaining performance in these large systems is to improve the communication between applications and the storage. Our solution is motivated by three trends that lead to more software layers: (1) more data, (2) extreme heterogeneity, and (3) open-source software. The proliferation of large complex stacks composed of many layers makes this thesis an especially timely solution.

The overwhelming volume, velocity, and veracity of today's data shapes modern software. When data grows too large, we scale to larger systems, either by scaling out or up. Focusing on the scale-out model has given birth to stacks, like Apache, that

are used in industry, laboratories, and academia. But the size of these stacks leads to increased complexity, as code bases are larger and there are more layers [74], resulting in reduced performance, redundant code, and longer code paths. The overhead of these stacks are so high that many workloads can be outperformed by a single node with less resources [71, 65, 70, 38, 50].

Extreme heterogeneity in both software and hardware has also lead to larger software stacks that manage resources. Data centers are larger and have faster devices because device and network speeds are scaling much faster than DRAM speeds. The so-called memory wall [93] pushes resource management into software runtimes, which must now manage large numbers of heterogeneous devices. The increasing momentum behind disaggregated storage [44, 43], a model that uses software as the control plane and reduces the CPU requirements of devices, is result of prognoses we are heading towards data centers that need to provision a CPU per storage device [66]. Regardless of where the future leads, the scale and complexity of software will continue to scale with the size of the architectures they manage.

Finally, the last trend that has lead to the explosion of software is open-source software. Open-source software is gaining traction because it helps consumers avoid vendor lock in, it leads to more efficient implementations, and it encourages collaboration. All these advantages are rooted in transparency, as developers can work together to write code that manages the extreme heterogeneity mentioned above, but it also lets developers see the source code for the systems they use "off-the-shelf". In short, open-source software leads to more software because (1) code can come from differ-

ent domains, organizations, and communities, and (2) it easier to write optimizations because functionality is fully exposed.

In light of these trends, our solution is a concept called "programmable storage" [74, 87]. Programmable storage facilitates the re-use and extension of existing storage abstractions provided by the underlying software stack, to enable the creation of new services via composition. This process is faster than reducing layers manually for new architectures with less layers [55] that may break backwards compatibility. We add interfaces *into* a storage system's internal functionality to facilitate application co-design, leading to more efficient implementations that inherit the robustness of the underlying system with less code duplication. This thesis uses the programmable storage approach to embed policy engines into file system metadata substrates to control the behavior, performance, and transparency of the entire software stack.

1.1 Contributions

This thesis argues that the programmable storage approach is the correct model for scaling global namespaces and designing policies effective file system metadata management policies. We design policies for three metadata management techniques: subtree load balancing, subtree semantics, and implied subtrees. The first two expand on a strong foundation of related work while the third is a novel idea.

First, we present Mantle, a programmable file system metadata load balancer. To help decouple policy from mechanism, we introduce a programmable storage sys-

tem that lets the designer inject custom balancing logic. We show the flexibility and transparency of this approach by replicating the strategy of a state-of-the-art metadata balancer and conclude by comparing this strategy to other custom balancers on the same system. We also show how the data management language and policy engine from Mantle turns out to be an effective control plane for managing ZLog sequencers and ParSplice caches.

Second, we present Cudele, an API and framework for programmable consistency and durability in a global namespace. The system lets clients specify their consistency/durability requirements and dynamically assign them to subtrees in the same namespace, allowing users to optimize subtrees over time and space for different workloads. We confirm the performance benefits of techniques presented in related work but also explore new consistency/durability metadata designs, all integrated over the same storage system. By custom fitting a subtree to a create- heavy application, we show 8 speedup and can scale to 2× as many clients when compared to our baseline system.

Third, we present Tintenfisch, a programmable file system that generates namespaces automatically. Clients and servers are injected with a function that describes the size and structure of a subtree and metadata is lazily populated. This reduces the number RPCs as clients/servers need not exchange messages to maintain consistency.

These contributions have mostly been prototyped on Ceph. Mantle was merged into Ceph and funded by the Center for Research in Open Source Software and Los

Alamos National Laboratory. Malacology and Mantle were featured in the Next Platform magazine and the 2017 Lua Workshop. Finally, Malacology and Cudele are the first Popper-compliant conference papers.

1.2 Research questions

Programmable storage attempts to answer answer important data management questions such as "how does the workload affect where/when I move resources" and "how does the system's parameters affect when I move things". This project will address the following general load balancing topics:

Which metrics do we use?

To properly balance load, we have to know which metrics are important and how they represent the state of the system. This requires understanding what metrics are available and how to collect them. The balancer also needs to know how the metrics are related to each other.

How do we quantify performance?

After isolating the important metrics, the balancer needs to understand how the metrics affect the global performance and behavior. This requires understanding how over-utilized resources negatively affect performance and how system events can indicate that the system is performing optimally.

What do we optimize?

Once we have the metrics and understand how they connect to performance, we will identify which metrics to optimize for in our load balancer. To do this, we must understand the trade-offs of optimizing each metric. Specifically, we must quantify how optimizing for one metric affects the other metrics in the system. This will involve solving a multi-objective optimization problem.

Which heuristics are successful?

Finally, when we know what to optimize for, we will apply different heuristics to resolve the optimization trade-offs. This will introduce problems that mirror other distributed systems problems, such as:

- aggressive vs. hesitant migrations
- centralized vs. decentralized knowledge
- small vs. large migration units
- accurate vs. fast decision-making
- learning vs. forgetting rate

Chapter 2

Background: Namespace Scalability

Namespaces resolve names to data. Traditionally, namespaces are hierarchical and allow users to group similar data together. Although file system namespaces are the most well known, other examples include DNS, LAN network topologies, and scoping in programming languages. File system namespaces are popular because they fit our mental organization as humans and are part of the POSIX IO standard. The momentum of namespaces as an abstraction and the overwhelming amount of legacy code written for namespaces makes the data model relatively future proof.

In file systems, whenever a file is created, modified, or deleted, the client must access the file's metadata. File system metadata contains information about the file, like size, links, access times, attributes, permissions/access control lists (ACLs), and ownership. In single disk file systems, clients consult metadata before seeking to data, by translating the file name to an inode and using that inode to lookup metadata in an inode table located at a fixed location on disk. Distributed file systems use a

similar idea; clients look in one spot for their metadata, usually a metadata service, and use that information to find data in a storage cluster. State-of-the-art distributed file systems decouple metadata from data access so that data and metadata I/O can scale independently [7, 25, 33, 89, 91, 95]. Unfortunately, recent trends have shown that separating metadata and data traffic is insufficient for scaling to large systems and identify the metadata service as the performance critical component.

First, we discuss general file system use cases and characterize the resultant metadata workloads. Next, we describe three semantics that users expect from file systems: strong consistency, durability, and a hierarchical organization. For each semantic, we explain why it is problematic for today's metadata workloads and survey optimizations in related work.

2.1 Metadata Workloads

File system metadata is small and highly accessed so the workloads are made up of many small requests [62, 6]. This skewed workload causes scalability issues in file systems because solutions for scaling data IO do not work for metadata IO [62, 5, 7, 89]. Unfortunately, this metadata problem is becoming more common and the same challenges that plagued HPC systems for years are finding their way into the cloud. Jobs that deal with many small files (*e.g.*, log processing and database queries [82]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [48]) are especially problematic.

If the use case is narrow enough, then developers in these domains can build application-specific storage stacks based on a thorough understanding of the workloads (*e.g.*, temperature zones for photos [51], well-defined read/write phases [17, 16], synchronization only needed during certain phases [30, 99], workflows describing computation [96, 24], etc.). Unfortunately, this "clean-slate" approach only works for one type of workload. To build a general-purpose file system, we need a thorough understanding of today's workloads and how they affect metadata services.

In this section, we describe modern applications (*i.e.* standalone programs, compilers, and runtimes) and common user behaviors (*i.e.* how users interact with file systems) that result in metadata-intensive workloads. For each use case, we provide motivation from HPC and cloud workloads; specifically, we look at users using the file system in parallel to run large-scale experiments in HPC and parallel runtimes that use the file system, such as MapReduce [17] (referred to as Hadoop, the open-source counterpart) and Spark [97].

2.1.1 Spatial Locality Within Directories

File system namespaces have semantic meaning; data stored in directories is related and is usually accessed together [89, 90]. Programs, compilers, and runtimes are usually triggered by users so the inputs/outputs to the job are stored within the user's home directory [88]. Hadoop and Spark enforce POSIX IO permissions and ownership to ensure users and bolt-on software packages operate within their assigned directories [4]. User behavior also exhibits locality. Listing directories after jobs is common and accesses

are localized to the user's working directory [62, 6].

A problem in HPC is users unintentionally accessing files in another user's directory. This behavior introduces false sharing and many file systems revoke locks and cached items for all clients to ensure consistency. While HPC tries to avoid these situations with workflows [98, 99], it still happens in distributed file systems when users unintentionally access directories in a shared file system.

2.1.2 Temporal Locality During Flash Crowds

Creates in the same directory is a problem in HPC , mostly due to checkpoint-restart [10]. Flash crowds of checkpoint-restart clients simultaneously open, write, and close files within a directory. But the workload also appears in cloud workloads: Hadoop/Spark use the file system to assign work units to workers and the performance is proportional to the open/create throughput of the underlying file system [94, 76, 77]; Big Data Benchmark jobs examined in [14] have on the order of 15,000 file opens or creates just to start a single Spark query and the Lustre system they tested on did not handle creates well, showing up to a 24× slowdown compared to other metadata operations. Common approaches to solve these types of bottlenecks is to change the application behavior or to design a new file system, like BatchFS [98] or DeltaFS [99], that uses one set of metadata optimizations for the entire namespace.

2.1.3 Listing Directories

As discussed before, this is common for general users (*e.g.*, reading a directory after a job completes), but users also use the file system for its centralized consistency. For example, users often leverage the file system to check the progress of jobs using `ls` even though this operation is notoriously heavy-weight [13, 22]. The number of files or size of the files is indicative of the progress. This practice is not too different from cloud systems that use the file system to manage the progress of jobs; Spark/Hadoop writes to temporary files, renames them when complete, and creates a "DONE" file to indicate to the runtime that the task did not fail and should not be re-scheduled on another node. For example, the browser interface lets Hadoop/Spark users check progress by querying the file system and returning a % of job complete metric.

2.1.4 Performance and Resource Utilization

The metadata workloads discussed in the previous section saturate resources on the metadata servers. Even small scale programs can show the effect; the resource utilization on the metadata server when compiling the Linux source code in a CephFS mount is shown in Figure 2.2. The `untar` phase, which is characterized by many creates, has the highest resource usage (combined CPU, network, and disk) on the metadata server because of the number of RPCs needed for consistency and durability. Many of our benchmarks use a create-heavy workload because it has high resource utilization.

Figure 2.1 shows the metadata locality for this workload. The "heat" of each directory is calculated with per-directory metadata counters, which are tempered with

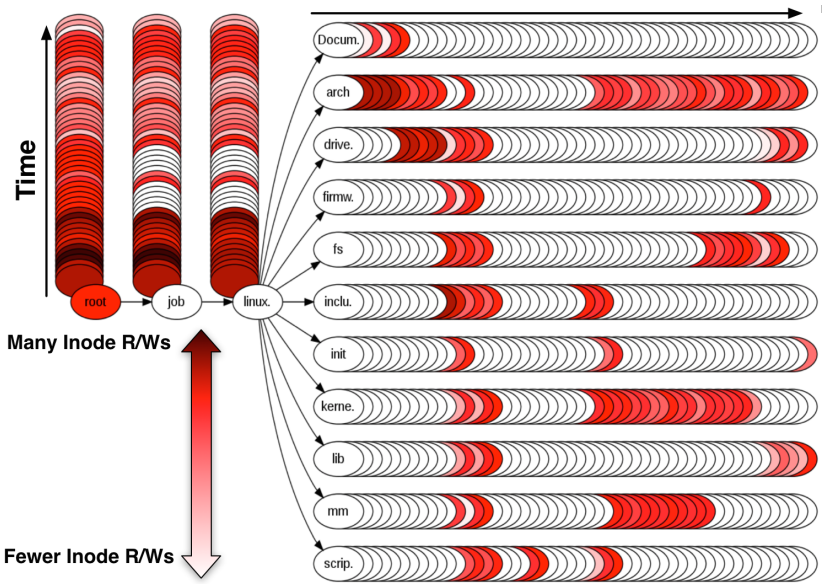


Figure 2.1: Metadata hotspots, represented by different shades of red, have spatial and temporal locality when compiling the Linux source code. The hotspots are calculated using the number of inode reads/writes and smoothed with an exponential decay.

an exponential decay. The hotspots can be correlated with phases of the job: untarring the code has high, sequential metadata load across directories and compiling the code has hotspots in the `arch`, `kernel`, `fs`, and `mm` directories.

2.2 Global Semantics: Strong Consistency

Access to metadata in a POSIX IO-compliant file system is strongly consistent, so reads and writes to the same inode or directory are globally ordered. The benefit of strong consistency is that clients and servers have the same view of the data, which makes state changes easier to reason about. The cost of this "safety" is performance.

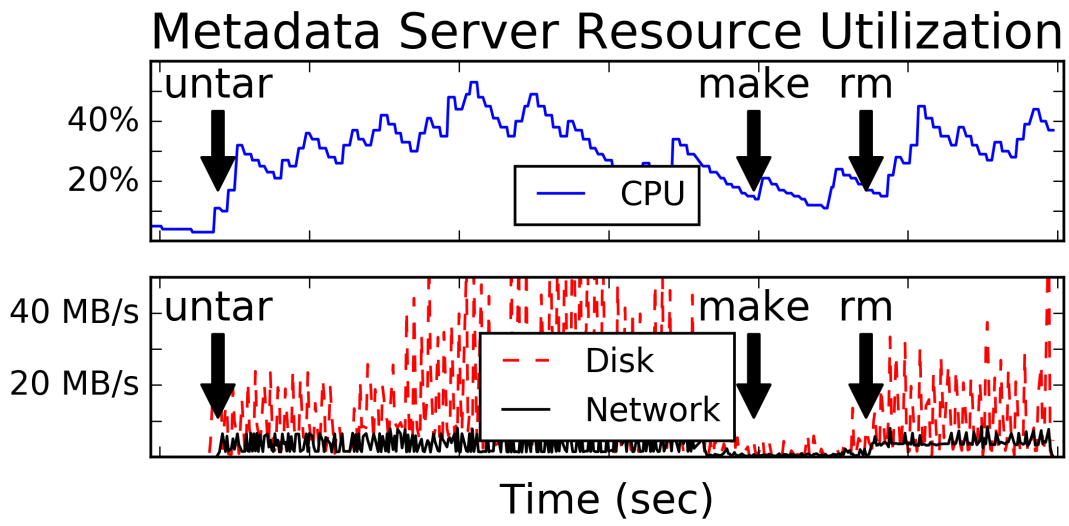


Figure 2.2: [source] **For the CephFS metadata server, create-heavy workloads (*e.g.*, `untar`) incur the highest disk, network, and CPU utilization because of consistency/durability demands.**

The synchronization and serialization machinery needed to ensure that all clients see the same state has high overhead. To make sure that all nodes or processes in the system are seeing the same state, they must come to an agreement. This limits parallelization and metadata performance has been shown to *decrease* with more sockets in Lustre [15]. As a result, and because it is simpler to implement, many distributed file systems limit the number of threads to one for all metadata servers [89, 7, 60].

Coming to an agreement has its own set of performance and accuracy trade-offs. Sophisticated, standalone consensus engines like PAXOS [45], Zookeeper [37], or Chubby [12] are common techniques for maintaining consistent versions of state in groups of processes that may disagree, but putting them in the data path is a large

bottleneck. In fact, PAXOS is used in Ceph and Zookeeper in Apache stacks to maintain cluster state but not for mediating IO.

Many distributed file systems use state machines to guard access to file system metadata. These state machines are stored with traditional file system metadata and they enforce the level of isolation that clients are guaranteed while they are reading or writing a file. CephFS [1, 88] calls the state machines "capabilities" and they are managed by authority metadata servers, GPFS [67] calls the state machines "write locks" and they can be shared, Panasas [92] calls the state machines "locks" and "callbacks", IndexFS [60] calls the state machines "leases" and they are dropped after a timeout, Lustre [69] calls the state machines "locks" and they protect inodes, extents, and file locks with different modes of concurrency [86]. Because this form of consistency is a bottleneck for metadata access, many systems optimize performance by improving locking protocols (Section §2.2.1), caching inodes (Section §2.2.2), and relaxing consistency (Section §2.2.3). We refer to these state machines as "locks" from now.

2.2.1 Lock Management

The global view of locks are read and modified with RPCs from clients. Single node metadata services, such as the Google File System (GFS) [25] and HDFS [77] have the simplest implementations and expose simple lock configurations like timeout thresholds. These implementations do not scale for metadata-heavy workloads so a natural approach to improving performance is to use a cluster to manage locks.

Distributed lock management systems spread the lock request load across a

cluster of servers. One approach is to distribute locks with the data by co-locating metadata servers with storage servers. PVFS2 [20] lets users spin up metadata servers on both storage and non-storage servers but the disadvantage of this approach is resource contention and poor file system metadata locality, respectively. Another approach is to orchestrate a dedicated metadata cluster from a centralized lock manager that accounts for load imbalance and locality. GPFS [67] assigns a process to be the "global lock manager", which is the authority of all locks and synchronizes access to metadata. Local servers become the authority of metadata by contacting the global lock manager, enabling optimizations like reducing RPCs. A decentralized version of this approach is to associate an authority process per inode. For example, Lustre, CephFS, IndexFS, and Panasas servers manage parts of the namespace and respond to client requests for locks. These approaches have more complexity but are flexible enough to service a range of workloads.

2.2.2 Caching Inodes

The discussion above refers to server-server lock exchange, but systems can also optimize client-server lock management. Caching inodes on both the client and server lets clients read/modify metadata locally. This reduces the number of RPCs required to agree on the state of metadata. For example, CephFS caches entire inodes, Lustre caches lookups, IndexFS caches ACLs, PVFS2 maintains a namespace cache and an attribute cache, Panasas lets clients read, cache, and parse directories, GPFS and Panasas cache the results of `stat()` [19], and GFS caches file location/striping

strategies. Some systems, like Ursa Minor [79] and pNFS [33] maintain client caches to reduce the overheads of NFS. These caches improve performance but the cache coherency mechanisms add significant complexity and overhead for some workloads.

2.2.3 Relaxing Consistency

A more disruptive technique is to relax the consistency semantics in the file system. Following the models pioneered by Amazon's eventual consistency [18] and the more fine-grained consistency models defined by Terry et al. [80], these techniques are gaining popularity because maintaining strong consistency has high overhead and because weaker guarantees are sufficient for many target applications.

Batching requests together is one form of relaxing consistency because updates are not seen immediately. PVFS2 batches creates, Panasas combines similar requests (*e.g.*, create and stat) together into one message, and Lustre surfaces configurations that allow users to enable and disable batching. Technically, batching requests is weaker than per-request strong consistency but the technique is often acceptable in POSIX-compliant systems.

More extreme forms of batching "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. One software-based approach is to prevent other clients from interfering with the decoupled directory until the first client commits changes back to the global namespace. This delayed merge (*i.e.* a form

of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of RPCs, synchronization, false sharing, and serialization. BatchFS and DeltaFS clients merge updates when the job is complete to avoid these costs and to encourage client-side processing. Another example approach is to move metadata intensive workloads to more powerful hardware. For example, for high metadata load MarFS [29] uses a cluster of metadata servers and TwoTiers [23] uses SSDs for the metadata server back-end. 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 themselves.

Even more drastic departures from POSIX IO allow writers and readers to interfere with each other. GFS leaves the state of the file undefined rather than consistent, forcing applications to use append rather than seeks and writes; in the cloud, Spark and Hadoop stacks use the Hadoop File System (HDFS) [78], which lets clients ignore this type of consistency completely by letting interfering clients read files opened for writing [30]; and CephFS offers the "Lazy IO" option, which lets clients buffer reads/writes even if other clients have the file open and if the client maintains its own cache coherency [1]. As noted earlier, many of these relaxed consistency semantics are for application-specific optimizations.

2.3 Global Semantics: Durability

While durability is not specified by POSIX IO, users expect that files they reate or modify survive failures. The accepted technique for achieving durability is to append events to a journal of metadata updates. Similar to LFS [63] and WAFL [35] the metadata journal is designed to be large (on the order of MBs) which ensures (1) sequential writes into the storage device (*e.g.*, object store, local disk, etc.) and (2) the ability for daemons to trim redundant or irrelevant journal entries. We refer to metadata updates as a journal, but of course, terminology varies from system to system (*e.g.*, operation log, event list, etc.). Ensuring durability has overhead so many performance optimizations target the file system's journal format and mechanisms.

2.3.1 Journal Format

A big point of contention for distributed file systems is not the technique of journaling metadata updates, rather it is the format of metadata. CephFS employs a custom on-disk metadata format that behaves more like a "pile system" [88]. Alternatively, IndexFS stores its journal in LSM trees for fast insertion and lookup. TableFS [59] lays out the reasoning for using LSM trees: the size of metadata (small) and the number of files (many) fits the LSM model well, where updates are written to the local file system as large objects (*e.g.*, write-ahead logs, SSTables, large files). Panasas separates requests out into separate logs to account for the semantic meaning and overhead of different requests ("op-log" for creates and updates and "cap-log" for capabilities). Many papers

claim that an optimized journal format leads to large performance gains [59, 60, 98] but we have found that the journal safety mechanisms have a much bigger impact on performance [72].

2.3.2 Journal Safety

We define three types of durability: global, local, and none. Global durability means that the client or server can fail at any time and metadata will not be lost because it is "safe" (*i.e.* striped or replicated across a cluster). GFS achieves global durability by replicating its journal from the master local disk to remote nodes and CephFS streams the journal into the object store. Local durability means that metadata can be lost if the client or server stays down after a failure. For example, in BatchFS and DeltaFS unwritten metadata updates are lost if the client (and/or its disk) fails and stays down. None means that metadata is volatile and that the system provides no guarantees when clients or servers fail. None is different than local durability because regardless of the type of failure, metadata will be lost when components die. Storing the journal in a RAMDisk would be an example of a system with a durability level of none.

Implementations of the types of durability vary, ranging from completely software-defined storage to architectures where hardware and software are more tightly-coupled, such as Panasas. Panasas assigns durability components to specific types of hardware. The journal is stored in battery-backed NVRAM and later replicated to both remote peers and metadata on objects. The software that writes the actual operations behaves similar to WAFL/LFS without the cleaner. The system also stores different kinds of

metadata (system vs. user, read vs. write) in different places. For example, directories are mirrored across the cluster using RAID1. This domain-specific mapping to hardware achieves high performance but sacrifices cost flexibility.

2.4 Hierarchical Semantics

Users identify and access file system data with a path name, which is a list of directories completed with a file name. File systems traverse (or resolve) paths to check permissions and to verify that files exist. Files and directories inherit some of the semantics from their parent directories, like ownership groups and permissions. For some attributes, like access and modifications times, parent directories must be updated as well.

To maintain these semantics file systems implement path traversal. Path traversal starts at the root of the file system and checks each path component until reaching the desired file. This process has write and read amplification because accessing lower subtrees in the hierarchy requires RPCs to upper levels. To reduce this amplification, many systems try to leverage the workload's locality; namely that directories at the top of a namespace are accessed more often [60] and files that are close in the namespace spatially are more likely to be accessed together [89, 90].

2.4.1 Caching Paths

To leverage the fact that directories at the top of the namespace are accessed more often, some systems cache "ancestor directories", *i.e.* metadata for entire paths. In

GIGA+ [54], clients contact the parent and traverse down its "partition history" to find which authority metadata server has the data. In the follow-up work, IndexFS, improves lookups and creates by having clients cache permissions instead of attributes. Similarly, Lazy Hybrid [11] hashes the file name to locate metadata but maintains extra per-file metadata to manage permissions. Although these techniques improve performance and scalability, especially for create intensive workloads, they do not leverage the locality inherent in file system workloads. For example, IndexFS's inode cache reduces RPCs by caching ancestor paths for metadata writes but the cache can be thrashed by random reads.

Caching can also be used to exploit locality. Many file systems hash the namespace across metadata servers to distribute load evenly, but this approach sacrifices workload locality. To compensate, systems like IndexFS and SkyFS [95] achieve locality by adding a metadata cache. This approach has a large space overhead, so HBA [100] uses hierarchical bloom filter arrays. Unfortunately, caching inodes is limited by the size of the caches and only performs well for temporal metadata, instead of spatial metadata locality [90, 75, 47]. Furthermore, keeping the caches coherent requires a fair degree of sophistication, which incurs overhead and limits the file system's ability to dynamically adapt to flash crowds.

2.4.2 Metadata Distribution

File systems like GIGA+, CephFS, SkyFS, HBA, and Ursa Minor use active-active metadata clusters instead of single metadata servers. Applications perform better

with dedicated metadata servers [75, 60] but provisioning a metadata server for every client is unreasonable. So finding the right number of metadata servers per client is a challenge. This problem is exacerbated by current hardware and software trends that encourage more clients; for example, HPC architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to more simplified stacks with just a burst buffer and object store [55]. This puts pressure on data access because more requests end up hitting the same layer and old techniques of hiding latencies while data migrates across tiers are no longer applicable.

2.4.2.1 Correctness: Addressing Metadata Inconsistency

Distributing metadata across a cluster requires distributed transactions and cache coherence protocols to ensure strong consistency. For example, file creates are fast in IndexFS because directories are fragmented and directory entries can be written in parallel but reads are subject to cache locality and lease expirations. ShardFS [94] makes the opposite trade-off because metadata reads are fast and resolve with 1 RPC while metadata writes are slow for everyone because they require serialization and multi-server locking. ShardFS achieves this by pessimistically replicating directory state and using optimistic concurrency control for conflicts, where operations fall back to two-phase locking if there is a conflict at verification time.

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 session-

s/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 [2]. These protocols are discussed in more detail in the next section but their inclusion here is a testament to the complexity of migrating metadata.

2.4.2.2 Performance: Leveraging Locality

Approaches that leverage the workload's spatial locality (*i.e.* requests targeted at a subset of directories or files) focus on metadata distribution across a cluster. File systems that hash their namespace spread metadata evenly across the cluster but do not account for spatial locality. IndexFS tries to alleviate this problem by distributing whole directories to different nodes. While this is an improvement, it does not address the fundamental data layout problem. Table-based mapping, done in systems like SkyFS, pNFS, and CalvinFS [81], is another metadata sharding technique, where the mapping of path to inode is done by a centralized server or data structure. These systems are static and while they may be able to exploit locality at system install time, their ability to scale or adapt with the workload is minimal.

Another technique is to assign subtrees of the hierarchical namespace to server nodes. Most systems use a static scheme to partition the namespace at setup, which requires a knowledgeable administrator. Ursa Minor and Farsite [21] traverse the namespace to assign related inode ranges, such as inodes in the same subtree, to servers. This benefits performance because the metadata server nodes can act independently without synchronizing their actions, making it easy to scale for breadth assuming that

incoming data is evenly partitioned. If carefully planned, assigning metadata to servers can achieve both even load distribution and locality, which facilitates multi-object operations and more efficient transactions. Unfortunately, static distribution limits the system's ability to adapt to hotspots/flash crowds and to maintain balance as data is added. Some systems, like Panasas, allow certain degrees of dynamicity by supporting the addition of new subtrees at runtime, but adapting to the current workload is ignored.

2.4.2.3 Performance: 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 [90, 88]. 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 [75].

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. Setting policies for when to partition directories and when to migrate the directory fragments 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.

2.5 Conclusion

This survey suggests that storage systems struggle:

1. **handling general-purpose workloads**. General-purpose file systems are hard to optimize so many users, including their applications (*i.e.* standalone programs,

compilers, and runtimes) and behaviors (*i.e.* how users interact with file systems), resort to application-specific storage stacks.

2. **selecting optimizations**. Optimizations must work together because they are dependent on each other. For example, we have found that for some workloads the metadata protocols in CephFS are inefficient and have a bigger impact on performance and scalability than load balancing. As a result, understanding these protocols improves load balancing and helps developers select metrics that systems 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.).

3. **guiding optimizations with policies**. Policies should be shaped by applications but most policies are hard-coded into the storage system or exposed as confusing configurations. This is exacerbated by software layering and the "skinny waist" to the storage system, which results in feature duplication and long code paths.

We use the programmable storage approach to ease these burdens and to facilitate more scalable namesapces.

Chapter 3

Prototyping Platform: Programmable Storage

3.1 Ceph: A Distributed Storage System

Ceph [89] is a distributed storage platform that stripes and replicates data across a reliable object store, called RADOS. Clients talk directly to object storage daemons (OSDs) on individual disks. This is done by calculating the data's placement ("where should I store my data") and location ("where did I store my data") using a hash-based algorithm (CRUSH). This architecture lets Ceph leverage the resources in the entire system because individual OSDs manage the state of the system.

CephFS is the POSIX-compliant file system that uses RADOS. CephFS is an important part of the storage ecosystem because it acts as a gateway to the file-based storage for legacy applications. It decouples metadata and data access, so data IO is

done directly with RADOS while all metadata operations go to a separate metadata cluster. This metadata cluster exposes a hierarchical namespace to the user using a technique called dynamic subtree partitioning [90]. In this scheme, each metadata server (MDS) manages a subtree in the namespace.

The MDS cluster is connected to the clients to service metadata operations and to RADOS so it can periodically flush its state. The CephFS components, including RADOS, the MDS cluster, and the logical namespace, are shown Figure 3.1.

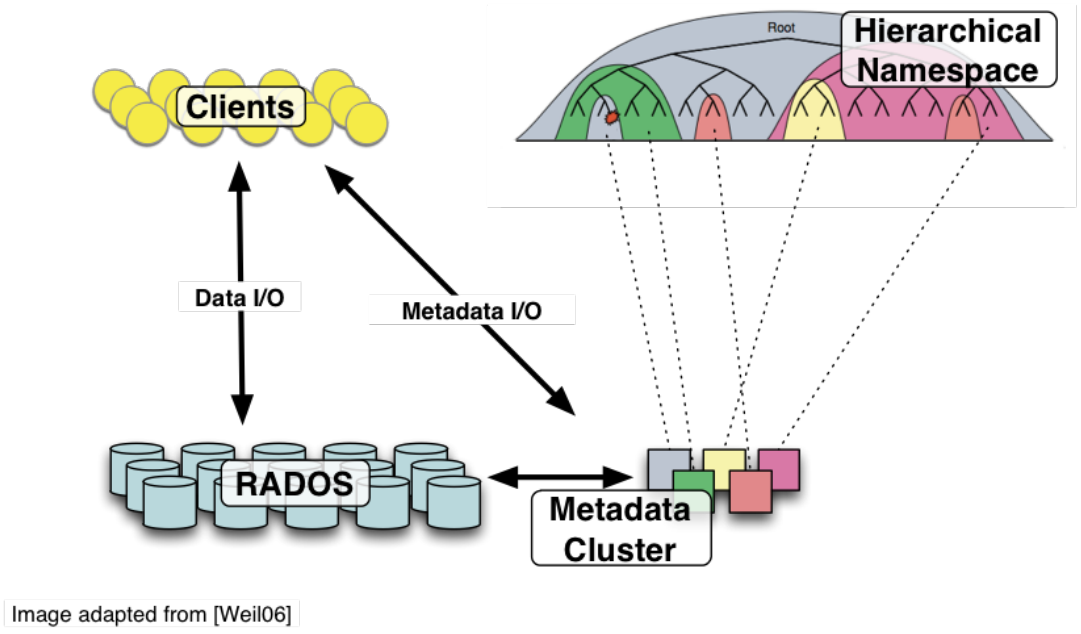


Figure 3.1: MDS cluster: in CephFS, the clients interact with a MDS cluster for all metadata operations. The MDS cluster exposes a hierarchal namespace using a technique called dynamic subtree partitioning, where each MDS manages a subtree in the namespace.

Why Use CephFS?

CephFS has one of the most advanced metadata infrastructures and we use it as a prototyping playform because (1) it was built for locality, (2) the migration tools are already implemented, and (3) the system is highly available.

CephFS accommodates the locality in file systems by packaging related data together. For example, inodes are embedded in directories so that related inodes are fetched on a `readdir`. We predict that migration is fast because directories are small. This is a consequence of having little allocation metadata, since data location is calculated.

CephFS also has many migration features already implemented, most notably directory migration and hotspot detection, as shown in Figure 3.1. Ceph has the ability to move subtrees to different metadata servers dynamically. Traditionally, when many creates/writes are made in the same directory, the filenames are hashed across multiple metadata servers. When many reads/opens are made to the same file, the contents are replicated across different metadata servers. CephFS also other infrastructure already in-place, such as:

- "**soft state**" for locating metadata: each MDS is only aware of the metadata in its own cache so clients hop around the MDS cluster and maintain their own hierarchical boundaries.

- **replication** for distribution/high read availability: distributed cache constraints allow path traversal to start at any node and to be redirected upon encountering

a subtree bound.

- **locking** to maintain consistency: replicas are read-only and all updates are forwarded to the authority for serialization/journaling; each metadata field is protected by a distributed state machine.

- **counters** to identify popularity: each inode and directory fragment maintain a popularity vector to aid in load balancing; MDSs share their measured loads so that they can determine how much to offload and to who to offload to (favors reuniting parents with children).

- "**frag trees**" for throughput (i.e. large directories): interior vertices split by powers of two and directory fragments are stored as a separate objects

- "**traffic control**" for flash crowd distribution (i.e. simultaneous clients): MDS tells clients if metadata is replicated or not so they can either contact the authority or replicas.

Moving the trees entails moving the subtree's cached metadata. It is performed as a two-phase commit: importer journals metadata (Import event), exporter logs event (Export event), importer journals event (ImportFinish).

We use CephFS to explore the metadata management problem. Finally, the Ceph system is accessible, active, and available. The source code is open source under the GNU license and the community is very active. It was also developed here at UC Santa Cruz, so it holds a special place in our heart. Initial interactions with Inktank,

the company delivers and supports Ceph, have been positive, providing useful feedback and guidance.

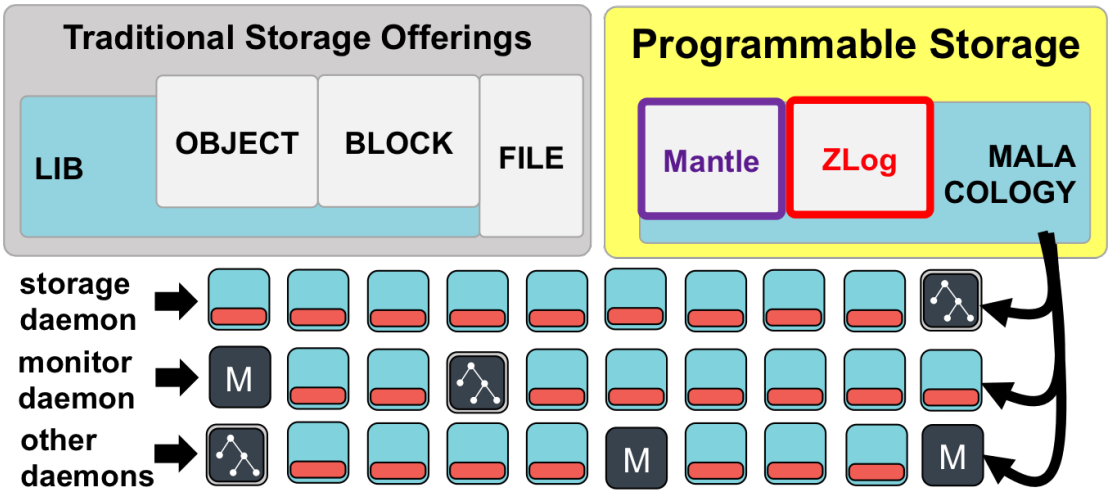


Figure 3.2: Scalable storage systems have storage daemons which store data, monitor daemons (M) that maintain cluster state, and service-specific daemons (e.g., file system metadata servers). Malacology enables the programmability of internal abstractions (bold arrows) to re-use and compose existing subsystems. With Malacology, we built new higher-level services, ZLog and Mantle, that sit alongside traditional user-facing APIs (file, block, object).

3.2 Malacology

We define a programmable storage system to be a storage system that facilitates the re-use and extension of existing storage abstractions provided by the underlying software stack, to enable the creation of new services via composition. A programmable storage system can be realized by exposing existing functionality (such as file system and cluster metadata services and synchronization and monitoring capabilities) as interfaces that can be "glued together" in a variety of ways using a high-level language.

Programmable storage differs from *active storage* [61]—the injection and execution of code within a storage system or storage device—in that the former is applicable to any component of the storage system, while the latter focuses at the data access level. Given this contrast, we can say that active storage is an example of how one internal component (the storage layer) is exposed in a programmable storage system.

To illustrate the benefits and challenges of this approach we have designed and evaluated Malacology, a programmable storage system that facilitates the construction of new services by re-purposing existing subsystem abstractions of the storage stack. We build Malacology in Ceph, a popular open source software storage stack. We choose Ceph to demonstrate the concept of programmable storage because it offers a broad spectrum of existing services, including distributed locking and caching services provided by file system metadata servers, durability and object interfaces provided by the back-end object store, and propagation of consistent cluster state provided by the monitoring service (see Figure 3.2). Malacology is expressive enough to provide the functionality necessary for implementing new services.

Malacology includes a set of interfaces that can be used as building blocks for constructing novel storage abstractions, including:

1. An interface for managing strongly-consistent time-varying **service metadata**.
2. An interface for installing and evolving domain-specific, cluster-wide **data I/O** functionality.
3. An interface for managing access to **shared resources** using a variety of opti-

mization strategies.

4. An interface for **load balancing** resources across the cluster.

5. An interface for **durability** that persists policies using the underlying storage stack's object store.

For more information, see the Malacology paper [74].

Chapter 4

Mantle: Subtree Load Balancing

The most common technique for improving the performance of metadata services is to balance the load across a cluster of metadata server (MDS) nodes [54, 89, 90, 79, 95]. Distributed MDS services focus on parallelizing work and synchronizing access to the metadata. A popular approach is to encourage independent growth and reduce communication, using techniques like lazy client and MDS synchronization [54, 60, 98, 33, 100], inode path/permission caching [**?**, 47, 95], locality-aware/inter-object transactions [79, 100, 59, 60] and efficient lookup tables [**?**, 100]. Despite having mechanisms for migrating metadata, like locking [79, 67], zero copying and two-phase commits [79], and directory partitioning [95, 54, 60, 89], these systems fail to exploit locality.

We envision a general purpose metadata balancer that responds to many types of parallel applications. To get to that balancer, we need to understand the trade-offs of

resource migration and the processing capacity of the MDS nodes. We present Mantle[1], a system built on CephFS that exposes these factors by separating migration policies from the mechanisms. Mantle accepts injectable metadata migration code and helps us make the following contributions:

- a comparison of balancing for locality and balancing for distribution
- a general framework for succinctly expressing different load balancing techniques
- an MDS service that supports simple balancing scripts using this framework

Using Mantle, we can dynamically select different techniques for distributing metadata. We explore the infrastructures for a better understanding of how to balance diverse metadata workloads and ask the question "is it better to spread load aggressively or to first understand the capacity of MDS nodes before splitting load at the right time under the right conditions?". We show how the second option can lead to better performance but at the cost of increased complexity. We find that the cost of migration can sometimes outweigh the benefits of parallelism (up to 40% performance degradation) and that searching for balance too aggressively increases the standard deviation in runtime.

[1]The mantle is the structure behind an octopus's head that protects its organs.

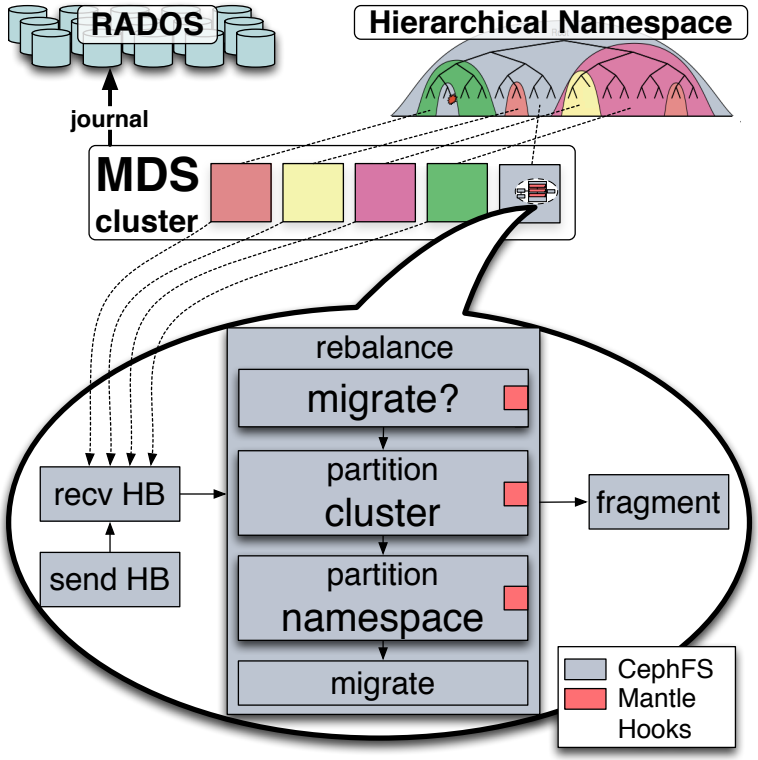


Figure 4.1: The MDS cluster journals to RADOS and exposes a namespace to clients. Each MDS makes decisions by exchanging heartbeats and partitioning the cluster/namespace. Mantle adds code hooks for custom balancing logic.

4.1 Background: Dynamic Subtree Partitioning

In CephFS MDS nodes use dynamic subtree partitioning [90] to carve up the namespace and to distribute it across the MDS cluster, as shown in Figure 4.1. MDS nodes maintain the subtree boundaries and "forward" requests to the authority MDS if a client's request falls outside of its jurisdiction or if the request tries to write to replicated metadata. Each MDS has its own metadata balancer that makes independent decisions,

using the flow in Figure 4.1. Every 10 seconds, each MDS packages up its metrics and sends a heartbeat ("send HB") to every MDS in the cluster. Then the MDS receives the heartbeat ("recv HB") and incoming inodes from the other MDS nodes. Finally, the MDS decides whether to balance load ("rebalance") and/or fragment its own directories ("fragment"). If the balancer decides to rebalance load, it partitions the namespace and cluster and sends inodes ("migrate") to the other MDS nodes. These last 3 phases are discussed below.

Migrate: inode migrations are performed as a two-phase commit, where the importer (MDS node that has the capacity for more load) journals metadata, the exporter (MDS node that wants to shed load) logs the event, and the importer journals the event. Inodes are embedded in directories so that related inodes are fetched on a `readdir` and can be migrated with the directory itself.

Partitioning the Namespace: each MDS node's balancer carves up the namespace into *subtrees* and *directory fragments* (added since [90, 89]). Subtrees are collections of nested directories and files, while directory fragments (*i.e.* dirfrags) are partitions of a single directory; when the directory grows to a certain size, the balancer fragments it into these smaller dirfrags. This directory partitioning mechanism is equivalent to the GIGA+ [54] mechanism, although the policies for moving the dirfrags can differ. These subtrees and dirfrags allow the balancer to partition the namespace into fine- or coarse-grained units.

Each balancer constructs a local view of the load by identifying popular subtrees or dirfrags using metadata counters. These counters are stored in the directories

and are updated by the MDS whenever a namespace operation hits that directory or any of its children. Each balancer uses these counters to calculate a *metadata load* for the subtrees and dirfrags it is in charge of (the exact policy is explained in Section §4.1.2.3). The balancer compares metadata loads for different parts of its namespace to decide which inodes to migrate. Once the balancer figures out which inodes it wants to migrate, it must decide where to move them.

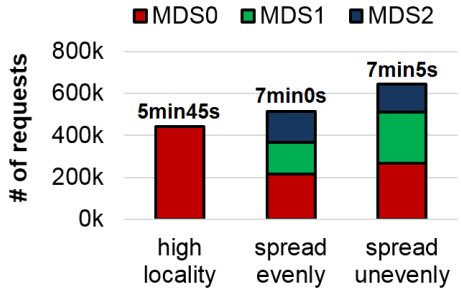
Partitioning the Cluster: each balancer communicates its metadata load and resource metrics to every other MDS in the cluster. Metadata load metrics include the metadata load on the root subtree, the metadata load on all the other subtrees, the request rate/latency, and the queue lengths. Resource metrics include measurements of the CPU utilization and memory usage. The balancer calculates an *MDS load* for all MDS nodes using a weighted sum of these metrics (again, the policy is explained in Section §4.1.2.3), in order to quantify how much work each MDS is doing. With this global view, the balancer can partition the cluster into exporters and importers. These loads also help the balancer figure out which MDS nodes to "target" for exporting and *how much* of its local load to send. The key to this load exchange is the load calculation itself, as an inaccurate view of another MDS or the cluster state can lead to poor decisions.

CephFS's Client-Server Metadata Protocols: the mechanisms for migrating metadata, ensuring consistency, enforcing synchronization, and mediating access are discussed at great length in [88] and the Ceph source code. MDS nodes and clients cache a configurable number of inodes so that requests like `getattr` and `lookup`

can resolve locally. For shared resources, MDS nodes have coherency protocols implemented using scatter-gather processes. These are conducted in sessions and involve halting updates on a directory, sending stats around the cluster, and then waiting for the authoritative MDS to send back new data. As the client receives responses from MDS nodes, it builds up its own mapping of subtrees to MDS nodes.

4.1.1 Advantages of Locality

(a) The number of requests for the compile job.



(b) Path traversals ending in hits (local metadata) and forwards.

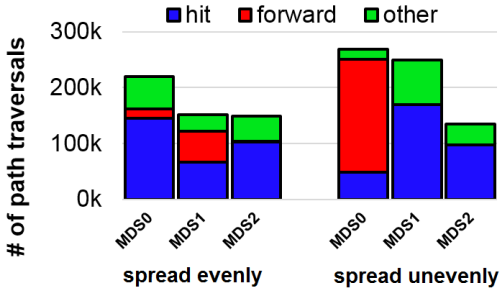


Figure 4.2: Spreading metadata to multiple MDS nodes hurts performance ("spread evenly/unevenly" setups in Figure 3a) when compared to keeping all metadata on one MDS ("high locality" setup in Figure 3a). The times given are the total times of the job (compile, read, write, etc.). Performance is worse when metadata is spread unevenly because it "forwards" more requests (Figure 3b).

Distributing metadata for balance tries to spread metadata evenly across the metadata cluster. The advantage of this approach is that clients can contact different

servers for their metadata in parallel. Many metadata balancers distribute metadata for complete balance by hashing a unique identifier, like the inode or filename; unfortunately, with such fine grain distribution, locality is completely lost. Distributing for locality keeps related metadata on one MDS and can improve performance. The reasons are discussed in [88, 90], but briefly, improving locality can:

- reduce the number of forwards between MDS nodes (*i.e.* requests for metadata outside the MDS node's jurisdiction)
- lower communication for maintaining coherency (*i.e.* requests involving prefix path traversals and permission checking)
- reduce the amount of memory needed to cache path prefixes. If metadata is spread, the MDS cluster replicates parent inode metadata so that path traversals can be resolved locally

Figure 4.2 alters the degree of locality by changing how metadata is distributed for a client compiling code on CephFS; with less locality, the performance gets worse and the number of requests increases. The number of requests (y axis) increases when metadata is distributed: the "high locality" bar is when all metadata is kept on one MDS, the "spread evenly" bar is when hot metadata is correctly distributed, and the "spread unevenly" bar is when hot metadata is incorrectly distributed[2]. For this example, the speedup for keeping all metadata on a single MDS is between 18% and 19%.

[2]To get high locality, all metadata is kept on one MDS. To get different degrees of spread, we change the setup: "spread unevenly" is untarring and compiling with 3 MDS nodes and "spread evenly" is untarring with 1 MDS and compiling with 3 MDS nodes. In the former, metadata is distributed when untarring (many creates) and the workload loses locality.

Although this is a small experiment, where the client clearly does not overload one MDS, it demonstrates how unnecessary distribution can hurt performance.

The number of requests increases when distributing metadata because the MDS nodes need to forward requests for remote metadata in order to perform common file system operations. The worse the distribution and the higher the fragmentation, the higher the number of forwards. Figure 4.2b shows that a high number of path traversals (y axis) end in "forwards" to other MDS nodes when metadata is spread unevenly. When metadata is spread evenly, much more of the path traversals can be resolved by the current MDS (*i.e.* they are cache "hits"). Aggressively caching all inodes and prefixes can reduce the requests between clients and MDS nodes, but CephFS (as well as many other file systems) do not have that design, for a variety of reasons.

4.1.2 Multi-MDS Challenges

Dynamic subtree partitioning achieves varying degrees of locality and distribution by changing the way it carves up the namespace and partitions the cluster. To alleviate load quickly, dynamic subtree partitioning can move different sized resources (inodes) to computation engines with variable capacities (MDS nodes), but this flexibility has a cost. In the sections below, we describe CephFS's current architecture and demonstrate how its complexity limits performance. While this section may seem like an argument against dynamic subtree partitioning, our main conclusion is that the approach has potential and warrants further exploration.

4.1.2.1 Complexity Arising from Flexibility

The complexity of deciding where to migrate resources increases significantly if these resources have different sizes and characteristics. To properly balance load, the balancer must model how components interact. First, the model needs to be able to predict how different decisions will positively impact performance. The model should consider what can be moved and how migration units can be divided or combined. It should also consider how splitting different or related objects affects performance and behavior. Second, the model must quantify the state of the system using available metrics. Third, the model must tie the metrics to the global performance and behavior of the system. It must consider how over-utilized resources negatively affect performance and how system events can indicate that the system is performing optimally. With such a model, the balancer can decide which metrics to optimize for.

Figure 4.3 shows how a 10 node, 3 MDS CephFS system struggles to build an accurate model that addresses the challenges inherent to the metadata management problem. That figure shows the total cluster throughput (y axis) over time (x axis) for 4 runs of the same job: creating 100,000 files in separate directories. The top graph, where the load is split evenly, is what the balancer tries to do. The results and performance profiles of the other 3 runs demonstrate that the balancing behavior is not reproducible, as the finish times vary between 5 and 10 minutes and the load is migrated to different servers at different times in different orders. Below, we discuss the design decisions that CephFS made and we demonstrate how policies with good intentions can lead to poor

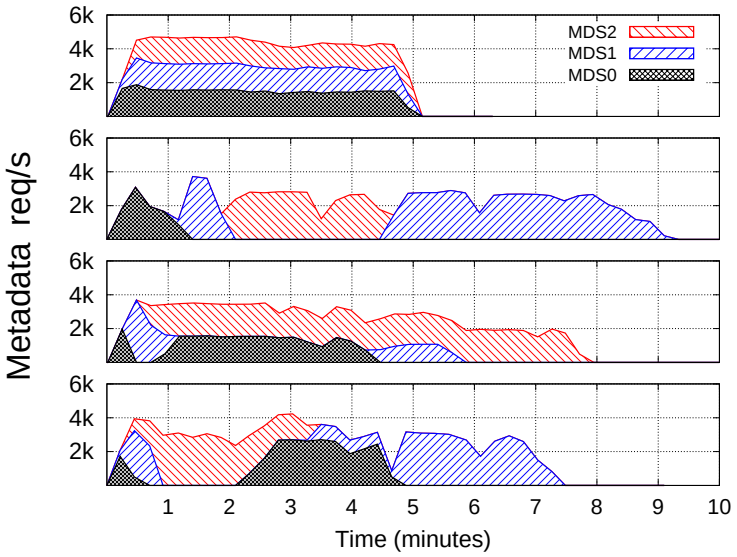


Figure 4.3: The same create-intensive workload has different throughput (y axis; curves are stacked) because of how CephFS maintains state and sets policies.

performance and unpredictability.

4.1.2.2 Maintaining Global & Local State

To make fast decisions, CephFS measures, collects, and communicates small amounts of state. Each MDS runs its balancing logic concurrently - this allows it to construct its own view of the cluster. The design decisions of the current balancer emphasizes speed over accuracy:

1. **Instantaneous measurements**: this makes the balancer sensitive to common system perturbations. The balancer can be configured to use CPU utilization as a metric for making decisions but this metric depends on the instant the measure-

ment is taken and can be influenced by the measurement tool. The balancer dulls this effect by comparing the current measurement against the previous measurement, but in our experiences decisions are still made too aggressively.

2. **Decentralized MDS state**: this makes the balancers reliant on state that is slightly stale. CephFS communicates the load of each MDS around the cluster using heartbeats, which take time to pack, travel across the network, and unpack. As an example, consider the instant MDS0 makes the decision to migrate some of its load; at this time, that MDS considers the aggregate load for the whole cluster by looking at all incoming heartbeats, but by the time MDS0 extracts the loads from all these heartbeats, the other MDS nodes have already moved on to another task. As a result of these inaccurate and stale views of the system, the accuracy of the decisions varies and reproducibility is difficult.

Even if maintaining state was instant and consistent, making the correct migration decisions would still be difficult because the workload itself constantly changes.

4.1.2.3 Setting Policies for Migration Decisions

In complicated systems there are two approaches for setting policies to guide decisions: expose the policies as tunable parameters or tie policies to mechanisms. Tunable parameters, or tunables, are configuration values that let the system administrator adjust the system for a given workload. Unfortunately, these tunable parameters are usually so specific to the system that only an expert can properly tune the system.

Policy	Hard-coded implementation
metaload	`= inode reads + 2*(inode writes)` `  + read dirs + 2*fetches + 4*stores`

MDSload	`= 0.8*(metaload on auth)`
	`  + 0.2*(metaload on all)`
	`  + request rate + 10*(queue length)`

when	`if my load > (total load)/#MDSs`

where	`for each MDS`
	`  if load > target:add MDS to exporters`
	`  else:add MDS to importers`
	`match large importers to large exporters`

how-much	`for each MDS`
accuracy	`  while load already sent < target load` `  export largest dirfrag`

Table 4.1: In the CephFS balancer, the policies are tied to mechanisms: loads quantify the work on a subtree/MDS; when/where policies decide when/where to migrate by assigning target loads to MDS nodes; how-much accuracy is the strategy for sending dirfrags to reach a target load.

For example, Hadoop version 2.7.1 exposes 210 tunables to configure even the simplest MapReduce application. CephFS has similar tunables. For example, the balancer will not send a dirfrag with load below `mds_bal_need_min`. Setting a sensible value for this tunable is almost impossible unless the administrator understands the tunable and has an intimate understanding of how load is calculated.

The other approach for setting policies is to hard-code the policies into the system alongside the mechanisms. This reduces the burden on the system administrator and lets the developer, someone who is very familiar with the system, set the policies.

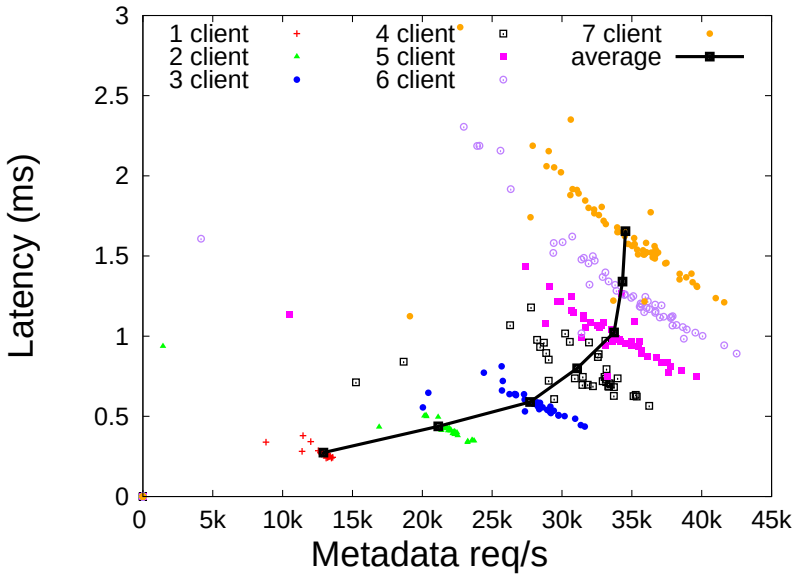


Figure 4.4: For the create heavy workload, the throughput (x axis) stops improving and the latency (y axis) continues to increase with 5, 6, or 7 clients. The standard deviation also increases for latency (up to 3×) and throughput (up to 2.3×).

The CephFS Policies

The CephFS policies, shown in Table 4.1, shape decisions using two techniques: scalarization of logical/physical metrics and hard-coding the logic. Scalarization means collapsing many metrics into a single value, usually with a weighted sum. When partitioning the cluster and the namespace, CephFS calculates metadata and MDS loads by collapsing the logical (*e.g.,* inode reads, inode writes, readdirs, etc.) and physical metrics (*e.g.,* CPU utilization, memory usage, etc.) into a single value. The exact calculations are in the "metaload" and "MDS load" rows of Table 4.1.

The other technique CephFS uses in its policies is to compile the decision logic into the system. The balancer uses one approach for deciding when and where to move inodes; it migrates load when it thinks that it has more load than the other MDS nodes ("when" row of Table 4.1) and it tries to migrate enough of its load to make the load even across the MDS cluster ("where" row of Table 4.1). While this approach is scalable, it reduces the efficiency of the cluster if the job could have been completed with less MDS nodes. Figure 4.4 shows how a single MDS performs as the number of clients is scaled, where each client is creating 100,000 files in separate directories. With an overloaded MDS servicing 5, 6, or 7 clients, throughput stops improving and latency continues to increase. With 1, 2, and 3 clients, the performance variance is small, with a standard deviation for latency between 0.03 and 0.1 ms and for throughput between 103 and 260 requests/second; with 3 or more clients, performance is unpredictable, with a standard deviation for latency between 0.145 and 0.303 ms and for throughput between 406 and 599 requests/second. This indicates that a single MDS can handle up to 4 clients without being overloaded.

Each balancer also sets policies for shedding load from its own namespace. While partitioning the cluster, each balancer assigns each MDS a target load, which is the load the balancer wants to send to that particular MDS. The balancer starts at its root subtrees and continuously sends the largest subtree or dirfrag until reaching this target load ("how-much accuracy" row of Table 4.1). If the target is not reached, the balancer "drills" down into the hierarchy. This heuristic can lead to poor decisions. For example, in one of our create heavy runs we had 2 MDS nodes, where MDS0 had 8

"hot" directory fragments with metadata loads: 12.7, 13.3, 13.3, 14.6, 15.7, 13.5, 13.7, 14.6. The balancer on MDS0 tried to ship off half the load by assigning MDS1 a target load of: $\frac{\text{total load}}{\text{\# MDSs}} = 55.6$. To account for the noise in load measurements, the balancer also scaled the target load by 0.8 (the value of the `mds_bal_need_min` tunable). As a result, the balancer only shipped off 3 dirfrags, 15.7 + 14.6 + 14.6, instead of half the dirfrags.

It is not the case that the balancer cannot decide how much load to send; it is that the balancer is limited to one heuristic (biggest first) to send off dirfrags. We can see why this policy is chosen; it is a fast heuristic to address the bin-packing problem (packing dirfrags onto MDS nodes), which is a combinatorial NP-Hard problem. This approach optimizes the speed of the calculation instead of accuracy and, while it may work for large directories with millions of entries, it struggles with simpler and smaller namespaces because of the noise in the load measurements and calculations.

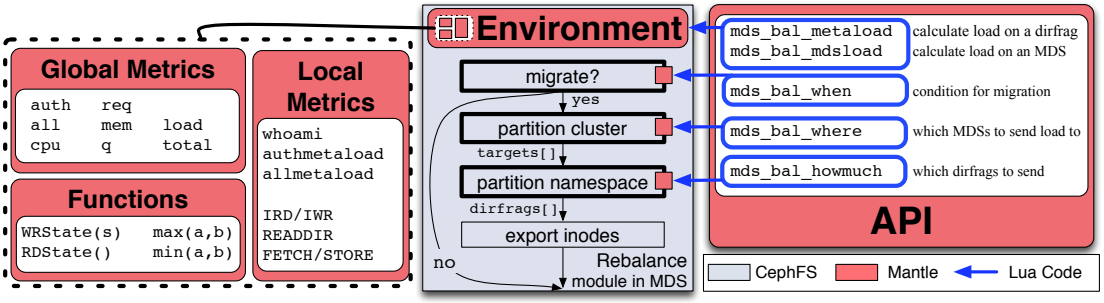


Figure 4.5: Designers set policies using the Mantle API. The injectable code uses the metrics/functions in the environment.

4.2 Mantle Implementation

The CephFS policies shape the decision making to be decentralized, aggressive, fast, and slightly forgetful. While these policies work for some workloads, including the workloads used to benchmark CephFS [90], they do not work for others (as demonstrated in Figure 4.3), they underutilize MDS nodes by spreading load to all MDS nodes even if the job could be finished with a subset, they destroy locality by distributing metadata without considering the workload, and they make it harder to coalesce the metadata back to one server after the flash crowd. We emphasize that the problem is that the policies are hardwired into the system, not the policies themselves.

Decoupling the policies from the mechanisms has many advantages: it gives future designers the flexibility to explore the trade-offs of different policies without fear of breaking the system, it keeps the robustness of well-understood implementations intact when exploring new policies, and it allows policies to evolve with new technologies and hardware. For example, McKusick [49] made the observation that when designing

the block allocation mechanism in the Fast File System (FFS), decoupling policy from mechanism greatly enhanced the usability, efficiency, and effectiveness of the system. The low-level allocation mechanism in the FFS has not changed since 1982, but now the developer can try many different policies, even the worst policy imaginable, and the mechanism will never curdle the file system, by doing things like double allocating.

Mantle builds on the implementations and data structures in the CephFS balancer, as shown in Figure 4.5. The mechanisms for dynamic subtree partitioning, including directory fragmentation, moving inodes from one MDS to another, and the exchange of heartbeats, are left unmodified. While this is a standard technique, applying it to a new problem can still be novel, particularly where nobody previously realized they were separable or has tried to separate them.

4.2.1 The Mantle Environment

Mantle decouples policy from mechanism by letting the designer inject code to control 4 policies: load calculation, "when" to move load, "where" to send load, and the accuracy of the decisions. Mantle balancers are written in Lua because Lua is fast (the LuaJIT virtual machine achieves near native performance) and it runs well as modules in other languages [26]. The balancing policies are injected at run time with Ceph's command line tool, *e.g.,* `ceph tell mds.0 injectargs mds_bal_metaload IWR`. This command means "tell MDS 0 to calculate load on a dirfrag by the number of inode writes".

Mantle provides a general environment with global variables and functions,

Current MDS metrics	Description
`whoami`	current MDS
`authmetaload`	metadata load on authority subtree
`allmetaload`	metadata load on all subtrees
`IRD,IWR`	# inode reads/writes (with a decay)
`READDIR,FETCH,STORE`	# read directories, fetches, stores

Metrics on MDS i	Description
`MDSs[i]["auth"]`	metadata load on authority subtree
`MDSs[i]["all"]`	metadata load on all subtrees
`MDSs[i]["cpu"]`	% of total CPU utilization
`MDSs[i]["mem"]`	% of memory utilization
`MDSs[i]["q"]`	# of requests in queue
`MDSs[i]["req"]`	request rate, in req/sec
`MDSs[i]["load"]`	result of `mds_bal_mdsload`
`total`	sum of the load on each MDS

Global Functions	Description
`WRstate(s)`	save state `s`
`RDstate()`	read state left by previous decision
`max(a,b),min(a,b)`	get the max, min of two numbers

Table 4.2: The Mantle environment.

shown on the left side of Figure 4.5, that injectable code can use. Local metrics are the current values for the metadata loads and are usually used to account for the difference between the stale global load and the local load. The library extracts the per-MDS metrics from the MDS heartbeats and puts the global metrics into an MDSs array. The injected code accesses the metric for MDS i using MDSs[i]["metric"]. The metrics and functions are described in detail in Table 5.1. The labeled arrows between the phases in Figure 4.5 are the inputs and outputs to the phases; inputs can be used and outputs must be filled by the end of the phase.

The `WRstate` and `RDstate` functions help the balancer "remember" decisions

from the past. For example, in one of the balancers, we wanted to make migration decisions more conservative, so we used `WRstate` and `RDstate` to trigger migrations only if the MDS is overloaded for 3 straight iterations. These are implemented using temporary files but future work will store them in RADOS objects to improve scalability.

4.2.2 The Mantle API

Figure 4.5 shows where the injected code fits into CephFS: the load calculations and "when" code is used in the "migrate?" decision, the "where" decision is used when partitioning the cluster, and the "howmuch" decision is used when partitioning the namespace for deciding the accuracy of sending dirfrags. To introduce the API we use the original CephFS balancer as an example.

Metadata/MDS Loads: these load calculations quantify the work on a subtree/dirfrag and MDS. Mantle runs these calculations and stuffs the results in the `auth/all` and `load` variables of Table 5.1, respectively. To mimic the scalarizations in the original CephFS balancer, one would set `mds_bal_metaload` to:

```
IRD + 2*IWR + READDIR + 2*FETCH + 4*STORE
```

and `mds_bal_mdsload` to:

```
0.8*MDSs[i]["auth"] + 0.2*MDSs[i]["all"]
+ MDSs[i]["req"] + 10*MDSs[i]["q"]
```

The metadata load calculation values inode reads (IRD) less than the writes (IWR), fetches and stores, and the MDS load emphasizes the queue length as a signal that the MDS is overloaded, more than the request rate and metadata loads.

When: this hook is specified as an "if" statement. If the condition evaluates to true, then migration decisions will be made and inodes may be migrated. If the condition is false, then the balancer exits immediately. To implement the original balancer, set `mds_bal_when` to:

```
if MDSs[whoami]["load"] > total/#MDSs then
```

This forces the MDS to migrate inodes if the load on itself is larger than the average cluster load. This policy is dynamic because it will continually shed load if it senses cluster imbalance, but it also has the potential to thrash load around the cluster if the balancer makes poor decisions.

Where: the designer specifies where to send load by populating the `targets` array. The index is the MDS number and the value is set to how much load to send. For example, to send off half the load to the next server, round robin, set `mds_bal_where` to:

```
targets[i] = MDSs[whoami + 1]["load"]/2
```

The user can also inject large pieces of code. The original CephFS "where" balancer can be implemented in 20 lines of Lua code (not shown).

How Much: recall that the original balancer sheds load by traversing down the namespace and shedding load until reaching the target load for each of the remote MDS nodes. Mantle traverses the namespace in the same way, but exposes the policy for how much to move at each level. Every time Mantle considers a list of dirfrags or subtrees in a directory, it transfers control to an external Lua file with a list of

strategies called dirfrag selectors. The dirfrag selectors choose the dirfrags to ship to a remote MDS, given the target load. The "howmuch" injectable argument accepts a list of dirfrag selectors and the balancer runs all the strategies, selecting the dirfrag selector that gets closest to the target load. We list some of the Mantle example dirfrag selectors below:

1. `big_first`: biggest dirfrags until reaching target
2. `small_first`: smallest dirfrags until reaching target
3. `big_small`: alternate sending big and small dirfrags
4. `half`: send the first half of the dirfrags

If these dirfrag selectors were running for the problematic dirfrag loads in Section §4.1.2.3 (12.7, 13.3, 13.3, 14.6, 15.7, 13.5, 13.7, 14.6), Mantle would choose the `big_small` dirfrag selector because the distance between the target load (55.6) and the load actually shipped is the smallest (0.5). To use the same strategy as the original balancer, set `mds_bal_howmuch` to: {`"big_first"`}

This hook does not control which subtrees are actually selected during namespace traversal (*i.e.* "which part"). Letting the administrator select specific directories would not scale with the namespace and could be achieved with separate mount points. Mantle uses one approach for traversing the namespace because starting at the root and drilling down into directories ensures the highest spatial and temporal locality, since subtrees are divided and migrated only if their ancestors are too popular to migrate. Policies that influence decisions for dividing, coalescing, or migrating specific

subtrees based on other types of locality (*e.g.,* request type) are left as future work.

4.2.3 Mantle on Programmable Storage

The original implementation is "hard-coded" into Ceph and lacked robustness (no versioning, durability, or policy distribution). Re-implemented using Malacology, Mantle now enjoys (1) the versioning provided by Ceph's monitor daemons and (2) the durability and distribution provided by Ceph's reliable object store. Re-using the internal abstractions with Malacology resulted in a 2× reduction in source code compared to the original implementation.

4.2.3.1 Versioning Balancer Policies

Ensuring that the version of the current load balancer is consistent across the physical servers in the metadata cluster was not addressed in the original implementation. The user had to set the version on each individual server and it was trivial to make the versions inconsistent. Maintaining consistent versions is important for cooperative balancing policies, where local decisions are made assuming properties about other instances in the cluster.

With Malacology, Mantle stores the version of the current load balancer in the Service Metadata interface. The version of the load balancer corresponds to an object name in the balancing policy. Using the Service Metadata interface means Mantle inherits the consistency of Ceph's internal monitor daemons. The user changes the version of the load balancer using a new CLI command.

4.2.3.2 Making Balancer Policies Durable

The load balancer version described above corresponds to the name of an object in RADOS that holds the actual Lua balancing code. When metadata server nodes start balancing load, they first check the latest version from the metadata server map and compare it to the balancer they have loaded. If the version has changed, they dereference the pointer to the balancer version by reading the corresponding object in RADOS. This is in contrast to the original Mantle implementation which stored load balancer code on the local file system – a technique which is unreliable and may result in silent corruption.

The balancer pulls the Lua code from RADOS synchronously; asynchronous reads are not possible because of the architecture of the metadata server. The synchronous behavior is not the default behavior for RADOS operations, so we achieve this with a timeout: if the asynchronous read does not come back within half the balancing tick interval the operation is canceled and a Connection Timeout error is returned. By default, the balancing tick interval is 10 seconds, so Mantle will use a 5 second second timeout.

This design allows Mantle to immediately return an error if anything RADOS-related goes wrong. We use this implementation because we do not want to do a blocking object storage daemon read from inside the global metadata server lock. Doing so would bring down the metadata server cluster if any of the object storage daemons are not responsive.

Storing the balancers in RADOS is simplified by the use of an interpreted language for writing balancer code. If we used a language that needs to be compiled, like the C++ object classes in the object storage daemon, we would need to ensure binary compatibility, which is complicated by different operating systems, distributions, and compilers.

4.2.3.3 Logging, Debugging, and Warnings

In the original implementation, Mantle would log all errors, warnings, and debug messages to a log stored locally on each metadata server. To get the simplest status messages or to debug problems, the user would have to log into each metadata server individually, look at the logs, and reason about causality and ordering.

With Malacology, Mantle re-uses the centralized logging features of the monitoring service. Important errors, warnings, and info messages are collected by the monitoring subsystem and appear in the monitor cluster log so instead of users going to each node, they can watch messages appear at the monitor daemon. Messages are logged sparingly, so as not to overload the monitor with frivolous debugging but important events, like balancer version changes or failed subsystems, show up in the centralized log.

4.3 Evaluation

All experiments are run on a 10 node cluster with 18 object storage daemons (OSDs), 1 monitor node (MON), and up to 5 MDS nodes. Each node is running Ubuntu 12.04.4 (kernel version 3.2.0-63) and they have 2 dual core 2GHz processors and 8GB of RAM. There are 3 OSDs per physical server and each OSD has its own disk formatted with XFS for data and an SSD partition for its journal. We use Ceph version 0.91-365-g2da2311. Before each experiment, the cluster is torn down and re-initialized and the kernel caches on all OSDs, MDS nodes, and clients are dropped.

Performance numbers are specific to CephFS but our contribution is the balancing API/framework that allows users to study different strategies *on the same storage system*. Furthermore, we are not arguing that Mantle is more scalable or better performing than GIGA+, rather, we want to highlight its strategy in comparison to other strategies using Mantle. While it is natural to compare raw performance numbers, we feel (and not just because GIGA+ outperforms Mantle) that we are attacking an orthogonal issue by providing a system for which we can test the strategies of the systems, rather than the systems themselves.

Workloads: we use a small number of workloads to show a comprehensive view of how load is split across MDS nodes. We use file-create workloads because they stress the system, are the focus of other state-of-the-art metadata systems, and they are a common HPC problem (checkpoint/restart). We use compiling code as the other workload because it has different metadata request types/frequencies and because

users plan to use CephFS as a shared file system [**?**]. Initial experiments with 1 client compiling with 1 MDS are, admittedly, not interesting, but we use it as a baseline for comparing against setups with more clients.

Metrics: Mantle pulls out metrics that could be important so that the administrator can freely explore them. The metrics we use are instantaneous CPU utilization and metadata writes, but future balancers will use metrics that better indicate load and that have less variability. In this paper, the high variance in the measurements influences the results of our experiments.

Balancing Heuristics: we use Mantle to explore techniques from related work: "Greedy Spill" is from GIGA+, "Fill & Spill" is a variation of LARD [**?**], and the "Adaptable Balancer" is the original CephFS policy. These heuristics are just starting points and we are not ready to make grandiose statements about which is best.

4.3.1 Greedy Spill Balancer

This balancer, shown in Listing 1, aggressively sheds load to all MDS nodes and works well for many clients creating files in the same directory. This balancing strategy mimics the uniform hashing strategy of GIGA+ [54, 60]. In these experiments, we use 4 clients each creating 100,000 files in the same directory. When the directory reaches 50,000 directory entries, it is fragmented (the first iteration fragments into $2^3 = 8$ dirfrags) and the balancer migrates half of its dirfrags to an "underutilized" neighbor.

The metadata load for the subtrees/dirfrags in the namespace is calculated using just the number of inode writes; we focus on create-intensive workloads, so inode

```
-- Metadata load
metaload = IWR
-- Metadata server load
mdsload = MDSs[i]["all"]
-- When policy
if MDSs[whoami]["load"]>.01 and
   MDSs[whoami+1]["load"]<.01 then
-- Where policy
targets[whoami+1]=allmetaload/2
-- Howmuch policy
{"half"}
```

Listing 1: Greedy Spill Balancer using the Mantle environment (listed in Table 5.1). Note that all subsequent balancers use the same metadata and MDS loads.

reads are not considered. The MDS load for each MDS is based solely on the metadata load. The balancer migrates load ("when") if two conditions are satisfied: the current MDS has load to migrate and the neighbor MDS does not have any load. If the balancer decides to migrate, it sheds half of the load to its neighbor ("where"). Finally, to ensure that exactly half of the load is sent at each iteration, we employ a custom fragment selector that sends half the dirfrags ("howmuch").

The first graph in Figure 4.6 shows the instantaneous throughput (y axis) of this balancer over time (x axis). The MDS nodes spill half their load as soon as they can - this splits load evenly for 2 MDS nodes, but with 4 MDS nodes the load splits unevenly because each MDS spills less load than its predecessor MDS. To get the even balancing shown in the second graph of Figure 4.6, the balancer is modified according to Listing 2 to partition the cluster when selecting the target MDS.

```
-- When policy
t=((#MDSs-whoami+1)/2)+whoami
if t>#MDSs then t=whoami end
while t~=whoami and MDSs[t]<.01 do t=t-1 end
if MDSs[whoami]["load"]>.01 and
   MDSs[t]["load"]<.01 then
-- Where policy
targets[t]=MDSs[whoami]["load"]/2
```

Listing 2: Greedy Spill Evenly Balancer.

This change makes the balancer search for an underloaded MDS in the cluster. It splits the cluster in half and iterates over a subset of the MDS nodes in its search for an underutilized MDS. If it reaches itself or an undefined MDS, then it has nowhere to migrate its load and it does not do any migrations. The "where" decision uses the target, `t`, discovered in the "when" search. With this modification, load is split evenly across all 4 MDS nodes.

The balancer with the most speedup is the 2 MDS configuration, as shown in Figure 4.7. This agrees with the assessment of the capacity of a single MDS in Section §4.1.2.3; at 4 clients, a single MDS is only slightly overloaded, so splitting load to two MDS nodes only improves the performance by 10%. Spilling unevenly to 3 and 4 MDS nodes degrades performance by 5% and 20% because the cost of synchronizing across multiple MDS nodes penalizes the balancer enough to make migration inefficient. Spilling evenly with 4 MDSs degrades performance up to 40% but has the lowest standard deviation because the MDS nodes are underutilized.

The difference in performance is dependent on the number of flushes to client

sessions. Client sessions ensure coherency and consistency in the file system (*e.g.,* permissions, capabilities, etc.) and are flushed when slave MDS nodes rename or migrate directories[3]: 157 sessions for 1 MDS, 323 session for 2 MDS nodes, 458 sessions for 3 MDS nodes, 788 sessions for 4 MDS nodes spilled unevenly, and 936 sessions for 4 MDS nodes with even metadata distribution. There are more sessions when metadata is distributed because each client contacts MDS nodes round robin for each create. This design decision stems from CephFS's desire to be a general purpose file system, with coherency and consistency for shared resources.

Performance: migration can have such large overhead that the parallelism benefits of distribution are not worthwhile.

Stability: distribution lowers standard deviations because MDS nodes are not as overloaded.

4.3.2 Fill and Spill Balancer

This balancer, shown in Listing 3, encourages MDS nodes to offload inodes *only* when overloaded. Ideally, the first MDS handles as many clients as possible before shedding load, increasing locality and reducing the number of forwarded requests. Figuring out when an MDS is overloaded is a crucial policy for this balancer. In our implementation, we use the MDS's instantaneous CPU utilization as our load metric,

[3]The cause of the latency could be from a scatter-gather process used to exchange statistics with the authoritative MDS. This requires each MDS to halt updates on that directory, send the statistics to the authoritative MDS, and then wait for a response with updates.

although we envision a more sophisticated metric built from a statistical model for future work. To figure out a good threshold, we look at the CPU utilization from the scaling experiment in Section §4.1.2.3. We use the CPU utilization when the MDS has 3 clients, about 48%, since 5, 6, and 7 clients appear to overload the MDS.

```
-- When policy
wait=RDState(); go = 0;
if MDSs[whoami]["cpu"]>48 then
  if wait>0 then WRState(wait-1)
  else WRState(2); go=1; end
else WRState(2) end
if go==1 then
-- Where policy
targets[whoami+1] = MDSs[whoami]["load"]/4
```

Listing 3: Fill and Spill Balancer.

The injectable code for both the metadata load and MDS load is based solely on the inode reads and writes. The "when" code forces the balancer to spill when the CPU load is higher than 48% for more than 3 straight iterations. We added the "3 straight iterations" condition to make the balancer more conservative after it had already sent load; in early runs the balancer would send load, then would receive the remote MDS's heartbeat (which is a little stale) and think that the remote MDS is *still underloaded*, prompting the balancer to send more load. Finally, the "where" code tries to spill small load units, just to see if that alleviates load enough to get the CPU utilization back down to 48%.

This balancer has a speedup of 6% over 1 MDS, as shown in Figure 4.7, and only uses a subset of the MDS nodes. With 4 available MDS nodes, the balancer only

uses 2 of them to complete the job, which minimizes the migrations and the number of sessions. The experiments also show how the amount of spilled load affects performance. Spilling 10% has a longer runtime, indicating that MDS0 is slightly overloaded when running at 48% utilization and would be better served if the balancer had shed a little more load. In our experiments, spilling 25% of the load has the best performance.

Performance: knowing the capacity of an MDS increases performance using only a subset of the MDS nodes.

Stability: the standard deviation of the runtime increases if the balancer compensates for poor migration decisions.

4.3.3 Adaptable Balancer

This balancer, shown in Listing 4, migrates load frequently to try and alleviate hotspots. It works well for dynamic workloads, like compiling code, because it can adapt to the spatial and temporal locality of the requests. The adaptable balancer uses a simplified version of the adaptable load sharing technique of the original balancer.

Again, the metadata and MDS loads are set to be the inode writes (not shown). The "when" condition only lets the balancer migrate load if the current MDS has more than half the load in the cluster and if it has the most load. This restricts the cluster to only one exporter at a time and only lets that exporter migrate if it has the majority of the load. This makes the migrations more conservative, as the balancer will only react if there is a single MDS that is severely overloaded. The "where" code scales the

```
-- Metadata load
metaload = IWR + IRD
-- When policy
max=0
for i=1,#MDSs do
  max = max(MDSs[i]["load"], max)
end
myLoad = MDSs[whoami]["load"]
if myLoad>total/2 and myLoad>=max then
-- Balancer where policy
targetLoad=total/#MDSs
for i=1,#MDSs do
  if MDSs[i]["load"]<targetLoad then
    targets[i]=targetLoad-MDSs[i]["load"]
  end
end
-- Howmuch policy
{"half","small","big","big_small"}
```

Listing 4: Adaptable Balancer.

amount of load the current MDS sends according to how much load the remote MDS has. Finally, the balancer tries to be as accurate as possible for all its decisions, so it uses a wide range of dirfrag selectors.

Figure 4.8 shows how Mantle can spread load across MDS nodes in different ways. That figure shows the overall performance for 5 clients compiling the Linux source code in separate directories. The balancer immediately moves the large subtrees, in this case the root directory of each client, and then stops migrating because no single MDS has the majority of the load. We conclude that 3 clients do not saturate the system enough to make distribution worthwhile and 5 clients with 3 MDS nodes is just as efficient as 4 or 5 MDS nodes.

The performance profile for the 5 MDS setups in Figure 4.9 shows how the aggressiveness of the balancer affects performance. The bold red curve is the metadata throughput for the compile job with 1 MDS and the stacked throughput curves correspond to the same job with 5 MDS nodes. The top balancer sets a minimum offload number, so it behaves conservatively by keeping all metadata on one MDS until a metadata load spike at 5 minutes forces distribution. The middle balancer is aggressive and distributes metadata load immediately. The flash crowd that triggers the migration in the top graph does not affect the throughput of the aggressive balancer, suggesting that the flash crowd requests metadata that the single MDS setup cannot satisfy fast enough; metadata is subsequently distributed but the flash crowd is already gone. The bottom balancer is far too aggressive and it tries to achieve perfect balance by constantly moving subtrees/dirfrags. As a result, performance is worse (60× as many forwards as the middle balancer), and the standard deviation for the runtime is much higher.

Performance: adapting the system to the workload can improve performance dramatically, but aggressively searching for the perfect balance hurts performance.

Stability: a fragmented namespace destroys locality and influences the standard deviation dramatically.

Overhead: the gap between the 1 MDS curve and the MDS0 curve in the top graph in Figure 4.9 is the overhead of the balancing logic, which includes the migration decisions, sending heartbeats, and fragmenting directories. The effect is significant, costing almost 500 requests per second, but should be dulled with more MDS nodes if they

make decisions independently.

4.3.4 Discussion and Future Work

In this paper we only show how certain policies can improve or degrade performance and instead focus on how the API is flexible enough to express many strategies. While we do not come up with a solution that is better than state-of-the-art systems optimized for file creates (*e.g.*, GIGA+), we do present a framework that allows users to study the emergent behavior of different strategies, both in research and in the classroom. In the immediate future, we hope to quantify the effect that policies have on performance by running a suite of workloads over different balancers. Other future endeavors will focus on:

Analyzing Scalability: our MDS cluster is small, but today's production systems use metadata services with a small number of nodes (often less than 5) [**?**]. Our balancers are robust until 20 nodes, at which point there is increased variability in client performance for reasons that we are still investigating. We expect to encounter problems with CephFS's architecture (*e.g.*, n-way communication and memory pressure with many files), but we are optimistic that we can try other techniques using Mantle, like GIGA+'s autonomous load splitting, because Mantle MDS nodes independently make decisions.

Adding Complex Balancers: the biggest reason for designing Mantle is to be able to test more complex balancers. Mantle's ability to save state should accommodate balancers that use request cost and statistical modeling, control feedback loops,

and machine learning.

Analyzing Security and Safety: in the current prototype, there is little safety - the administrator can inject bad policies (*e.g.*, `while 1`) that brings the whole system down. We wrote a simulator that checks the logic before injecting policies in the running cluster, but this still needs to be integrated into the prototype.

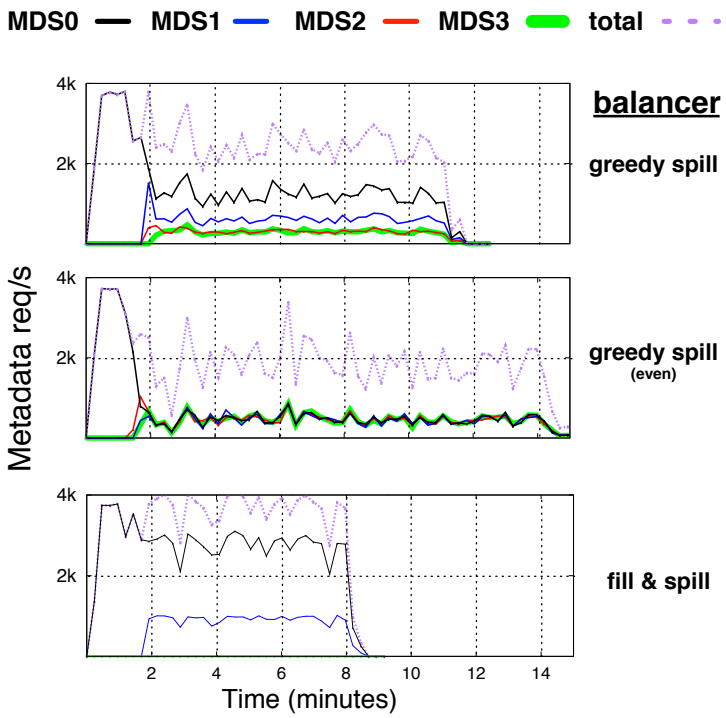


Figure 4.6: With clients creating files in the same directory, spilling load unevenly with Fill & Spill has the highest throughput (curves are not stacked), which can have up to 9% speedup over 1 MDS. Greedy Spill sheds half its metadata immediately while Fill & Spill sheds part of its metadata when overloaded.

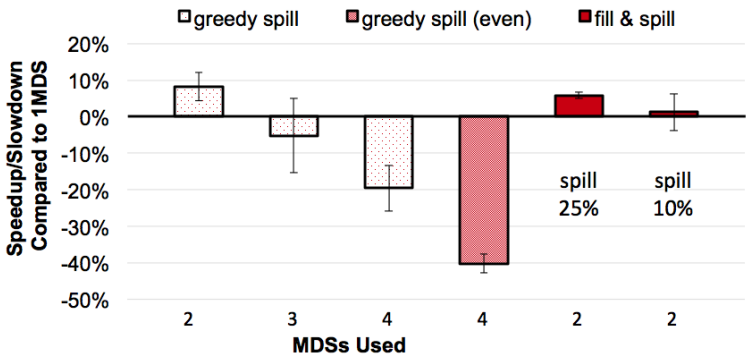


Figure 4.7: The per-client speedup or slowdown shows whether distributing metadata is worthwhile. Spilling load to 3 or 4 MDS nodes degrades performance but spilling to 2 MDS nodes improves performance.

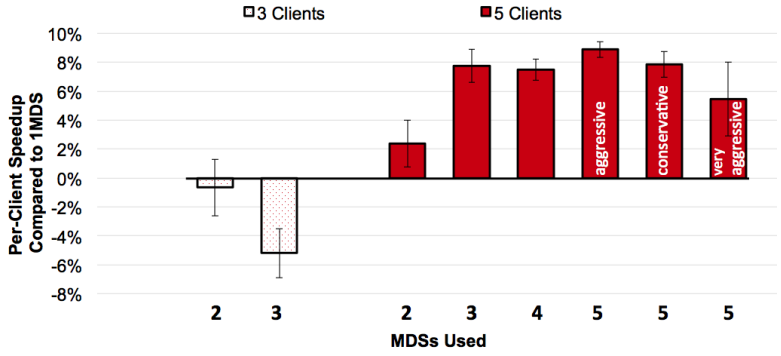


Figure 4.8: For the compile workload, 3 clients do not overload the MDS nodes so distribution is only a penalty. The speedup for distributing metadata with 5 clients suggests that an MDS with 3 clients is slightly overloaded.

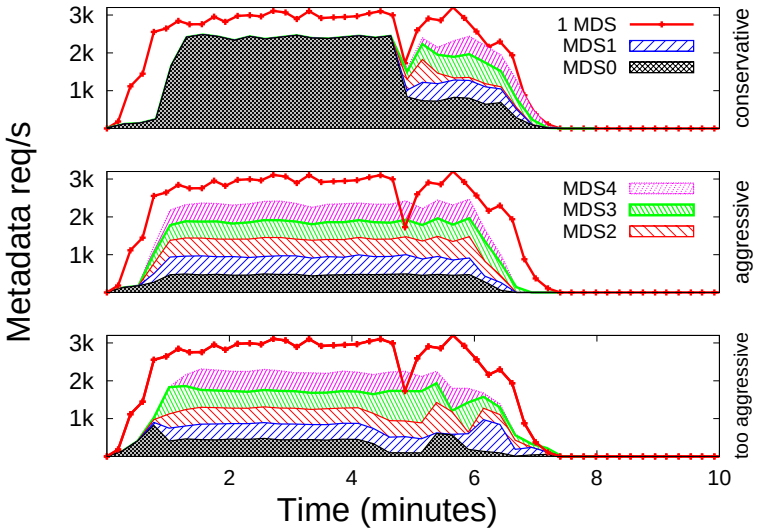


Figure 4.9: With 5 clients compiling code in separate directories, distributing metadata load early helps the cluster handle a flash crowd at the end of the job. Throughput (stacked curves) drops when using 1 MDS (red curve) because the clients shift to linking, which overloads 1 MDS with `readdir`s.

4.4 Related Work

Mantle decouples policy from mechanism in the metadata service to stabilize decision making. Much of the related work does not focus on the migration policies themselves and instead focuses on mechanisms for moving metadata.

Compute it - Hashing: this distributes metadata evenly across MDS nodes and clients find the MDS in charge of the metadata by applying a function to a file identifier. PVFSv2 [33] and SkyFS [95] hash the filename to locate the authority for metadata. CalvinFS [81] hashes the pathname to find a database shard on a server. It handles many small files and fully linearizable random writes using the feature rich Calvin database, which has support for WAN/LAN replication, OLLP for mid-commit commits, and a sophisticated logging subsystem.

Look it up - Table-based Mapping: this is a form of hashing, where indices are either managed by a centralized server or the clients. For example, IBRIX [36] distributes inode ranges round robin to all servers and HBA [100] distributes metadata randomly to each server and uses bloom filters to speedup the table lookups. These techniques also ignore locality.

To further enhance scalability, many hashing schemes employ dynamic load balancing. [47] presented dynamic balancing formulas to account for a forgetting factor, access information, and the number of MDS nodes in elastic clusters. [95] used a master-slave architecture to detect low resource usage and migrated metadata using a consistent hashing-based load balancer. GPFS [67] elects MDS nodes to manage meta-

data for different objects. Operations for different objects can operate in parallel and operations to the same object are synchronized. While this approach improves metadata parallelism, delegating management to different servers remains centralized at a token manager. This token manager can be overloaded with requests and large file system sizes - in fact, GPFS actively revokes tokens if the system gets too big. GIGA+ [54] alleviates hotspots and "flash crowds" by allowing unsynchronized directory growth for create intensive workloads.

4.5 Conclusion

The flexibility of dynamic subtree partitioning introduces significant complexity and many of the challenges that the original balancer tries to address are general, distributed systems problems. In this paper, we present Mantle, a programmable metadata balancer for CephFS that decouples policies from the mechanisms for migration by exposing a general "balancing" API. We explore the locality vs. distribution space and make important conclusions about the performance and stability implications of migrating load. The key takeaway from using Mantle is that distributing metadata can negatively both performance and stability. With Mantle, we are able to compare the strategies for metadata distribution instead of the underlying systems. With this general framework, broad distributed systems concepts can be explored in depth to gain insights into the true bottlenecks that we face with modern workloads.

Chapter 5

Mantle Beyond Ceph

Mantle provides a control plane that injects policies into a running storage system, such as a file system or key-value store. While Mantle was originally designed for file system metadata load balancing [75], we find that it works surprisingly well for specifying cache management policies without requiring users to possess extensive knowledge about the internals of storage systems.

To explore software-defined cache management, we use the data management language and policy engine presented in [75]. The prototype in that paper, Mantle, was built on CephFS and lets administrators control file system metadata load balancing policies. We now refer to Mantle as a policy engine that supports our data management language. The basic premise is that data management policies can be expressed with a simple API consisting of "when", "where", and "how much". The "when" policy controls how aggressive or conservative the decisions are; "where" controls how distributed or concentrated the data should be; and "how much" controls the amount of data that

should be sent. There is also a "load" policy that lets administrators specify how to collapse many metrics into a single load metric (*e.g.*, $2 \times$ `cpu` $+ 3 \times$ `memory usage`). The succinctness of the API lets users inject multiple, possibly dynamic, policies.

5.1 Extracting Mantle as a Library

We extracted Mantle as a library and Figure 5.1 shows how it is linked into a storage system service. Administrators write policies in Lua from whatever domain they choose (*e.g.*, statistics, machine learning, storage system) and the policies are embedded into the runtime by Mantle. We chose Lua for simplicity, performance, and portability; it is a scripting language with simple syntax, which allows administrators to focus on the policies themselves; it was designed as an embeddable language, so it is lightweight and does less type checking; and it interfaces nicely with C/C++. When the storage system makes decisions it executes the administrator-defined policies for when/where/how much and returns a decision. To do this, the storage system needs to be modified to (1) provide an environment of metrics and (2) identify where policies are set. These modification points are shown by the colored boxes in Figure 5.1 and described below.

5.1.0.1 Environment of Metrics

storage systems expose **cluster** metrics for describing resource utilizations and **time series** metrics for describing accesses to some data structure over time. Table 5.1 shows how these metrics are accessed from the policies written by administrators.

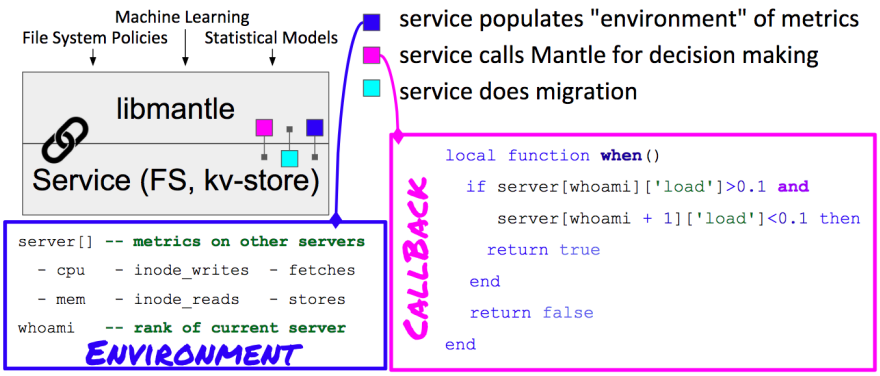


Figure 5.1: Extracting Mantle as library.

Metrics	Data Structure	Description
Cluster	{server → {metric → val}}	resource util. for servers
Time Series	[(ts, val), ..., (ts, val)]	accesses by timestamp (ts)
	Storage System	Example
Cluster	File Systems	CPU util., Inode reads
	ParSplice	CPU util., Cache size
Time Series	File Systems	Accesses to directory
	ParSplice	Accesses to key in DB

Table 5.1: Types of metrics exposed by the storage system to the policy engine using Mantle.

For cluster metrics, the storage system passes a dictionary to Mantle. Policies access the cluster metric values by indexing into a Lua table using `server` and `metric`, where server is a node identifier (*e.g.*, MPI Rank, metadata server name) and metric is a resource name. Metrics used for file system metadata load balancing are shown by the "environment" box in Figure 5.1. The measurements and exchange of metrics between servers is done by the storage system; Mantle in CephFS leverages metrics from other servers collected using CephFS's heartbeats. For example, a policy written for an MPI-based storage system can access the CPU utilization of the first rank in a

communication group using:

```
load = servers[0]['cpu']
```

For time series metrics, the storage system passes an array of `(timestamp, value)` pairs to Mantle and the policies can iterate over the values. The storage system uses a pointer to the time series to facilitate time series with many values, like accesses to a database or directory in the file system namespace. This decision limits the time series metrics to only include values from the *current* node, although this is not a limitation of Mantle itself. For example, a policy that uses accesses to a directory in a file system as a metric for load collects that information using:

```
d = timeseries()        -- d(ata) from storage system
for i=1,d:size() do     -- iterate over timeseries
  ts, value = d:get(i) -- index into timeseries
  if value == 'mydirectory' then
    count = count + 1
  end
end
```

5.1.0.2 Policies Written as Callbacks

the "callback" box in Figure 5.1 shows an example policy for "when()", where the current server (`whoami`) migrates load if it is has load (`>0.1`) and if its neighbor server (`whoami + 1`) does not have load (`<0.1`). The load is calculated using the metrics provided by the environment.

Mantle also provides functions for persisting state across decisions. `WRState(s)`

saves state `s`, which can be a number or boolean value, and `RDState()` returns the state saved by a previous iteration. For example, a "when" policy can avoid trimming a cache or migrating data if it had performed that operation in the previous decision.

5.2 Domain 1: Load Balancing for ZLog

In practice, a storage system implementing CORFU will support a multiplicity of independent totally-ordered logs for each application. For this scenario co-locating sequencers on the same physical node is not ideal but building a load balancer that can migrate the shared resource (e.g., the resource that mediates access to the tail of the log) is a time-consuming, non-trivial task. It requires building subsystems for migrating resources, monitoring the workloads, collecting metrics that describe the utilization on the physical nodes, partitioning resources, maintaining cache coherence, and managing multiple sequencers. The following experiments demonstrate the feasibility of using the mechanisms of the Malacology Load Balancing interface to inherit these features and to alleviate load from overloaded servers.

We also discuss latent capabilities we discovered in this process that let us navigate different trade-offs within the services themselves. We benchmark scenarios in which the storage system manages multiple logs by using Mantle to balance sequencers across a cluster. Since this work focuses on Mantle atop Malacology the goal of this section is to show that the components and subsystems that support the Malacology interfaces provide reasonable relative performance, as well as to give examples of the flexibility that Malacology provides to programmers. This section uses a principled approach for evaluating tunables of the interfaces and the trade-offs we discuss should be acknowledged when building higher-level services.

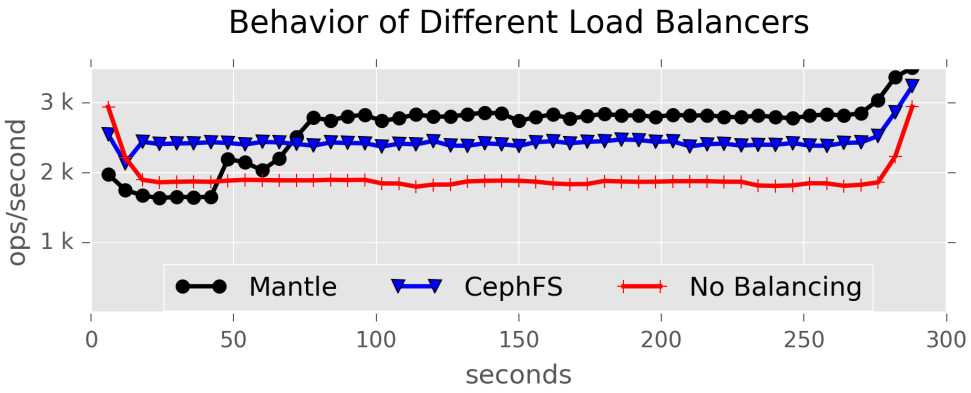


Figure 5.2: [source] CephFS/Mantle load balancing have better throughput than co-locating all sequencers on the same server. Sections 5.3.1 and 5.3.2 quantify this improvement; Section 5.3.3 examines the migration at 0-60 seconds.

5.3 Domain 1: Evaluation

The experiments are run on a cluster with 10 nodes to store objects, one node to monitor the cluster, and 3 nodes that can accommodate sequencers. Instead of measuring contention at the clients like Section **??**, these experiments measure contention at the sequencers by forcing clients to make round-trips for every request. We implement this using the Shared Resource interface that forces round-trips. Because the sequencer's only function is to hand out positions for the tail of the log, the workload is read-heavy.

First, we show how the ZLog service can orchestrate multiple sequencers using the Malacology Load Balancing interface. Figure 5.2 shows the throughput over time of different load balancers as they migrate 3 sequencers (with 4 clients) around the cluster;

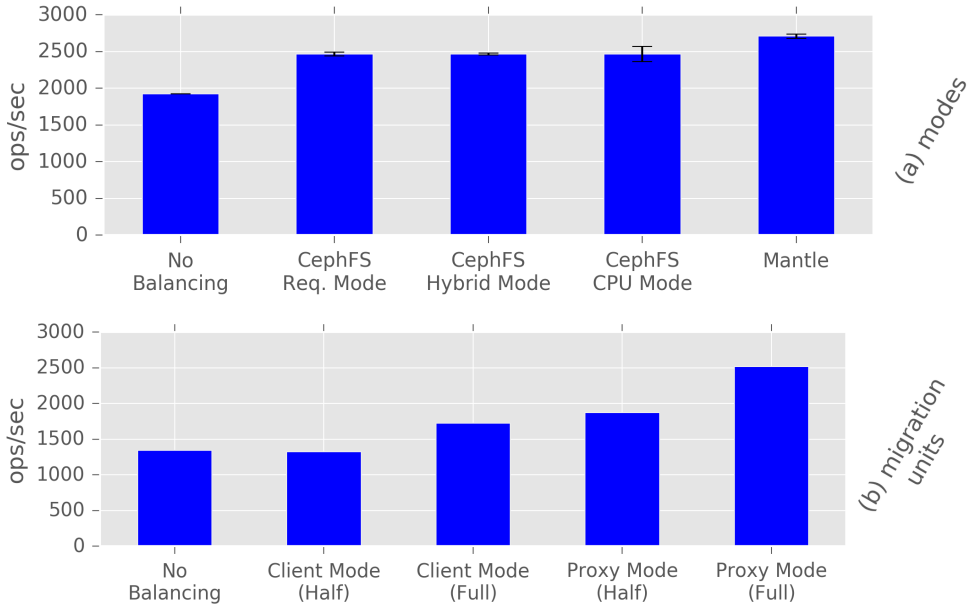


Figure 5.3: [source, source] In (a) all CephFS balancing modes have the same performance; Mantle uses a balancer designed for sequencers. In (b) the best combination of mode and migration units can have up to a 2× improvement.

"No Balancing" keeps all sequencers on one server, "CephFS" migrates sequencers using the hard-coded CephFS load balancers, and "Mantle" uses a custom load balancer we wrote specifically for sequencers. The increased throughput for the CephFS and Mantle curves between 0 and 60 seconds are a result of migrating the sequencer(s) off overloaded servers.

In addition to showing that migrating sequencers improves performance, Figure 5.2 also demonstrates features that we will explore in the rest of this section. Sections 5.3.1 and 5.3.2 quantify the differences in performance when the cluster stabilizes at

time 100 seconds and Section 5.3.3 examines the slope and start time of the re-balancing phase between 0 and 60 seconds by comparing the aggressiveness of the balancers.

5.3.1 Feature: Balancing Modes

Next, we quantify the performance benefits shown in Figure 5.2. To understand why load balancers perform differently we need to explain the different balancing modes that the load balancer service uses and how they stress the subsystems that receive and forward client requests in different ways. In Figure 5.2, the CephFS curve shows the performance of the balancing mode that CephFS falls into *most of the time*. CephFS currently has 3 modes for balancing load: CPU mode, workload mode, and hybrid mode. All three have the same structure for making migration decisions but vary based on the metric used to calculate load. For this sequencer workload the 3 different modes all have the same performance, shown in Figure 5.3 (a), because the load balancer falls into the same mode a majority of the time. The high variation in performance for the CephFS CPU Mode bar reflects the uncertainty of using something as dynamic and unpredictable as CPU utilization to make migration decisions. In addition to the suboptimal performance and unpredictability, another problem is that all the CephFS balancers behave the same. This prevents administrators from properly exploring the balancing state space.

Mantle gives the administrator more control over balancing policies; for the Mantle bar in Figure 5.3 (a) we use the Load Balancing interface to program logic for balancing read-heavy workloads, resulting in better throughput and stability. When we

did this we also identified two balancing modes relevant for making migration decisions for sequencers.

Using Mantle, the administrator can put the load balancer into "proxy mode" or "client mode". In proxy mode one server receives all requests and farms off the requests to slave servers; the slave servers do the actual tail finding operation. In client mode, clients interact directly with the server that has their sequencer. These modes are illustrated in Figure 5.4. "No Balancing" is when all sequencers are co-located on one physical server – performance for that mode is shown by the "No Balancing" curve in Figure 5.2. In "Proxy Mode", clients continue sending requests to server A even though some of the sequencers have been migrated to another server. Server A redirects client requests for sequencer 2 to server B. "Proxy Mode (Half)" is shown in Figure 5.2; in this scenario, half of the sequencers have migrated off the first server. Alternatively, "Proxy Mode (Full)", which is not pictured, is when all the sequencers migrate off the first server. "Client Mode", shown on the far right of Figure 5.4, shows how clients for sequencer 2 contact server B without a redirect from server A.

Figure 5.5 shows the throughput over time of the two different modes for an environment with only 2 sequencers (again 4 clients each) and 2 servers. The curves for both sequencers in Figure 5.5(a) start at less than 1000 ops/second and at time 60 seconds Mantle migrates Sequencer 1 to the slave server. Performance of Sequencer 2 decreases because it stayed on the proxy which now processes requests for Sequencer 2, and forwards requests for Sequencer 1. The performance of Sequencer 1 improves dramatically because distributing the sequencers in this way separates (1) the handling

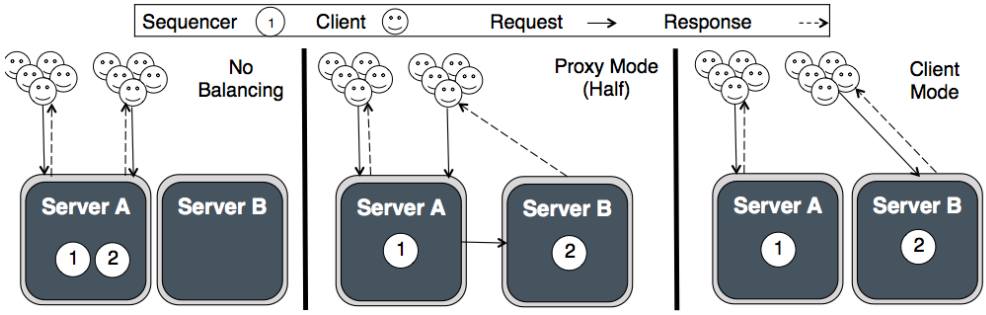


Figure 5.4: In client mode clients sending requests to the server that houses their sequencer. In proxy mode clients continue sending their requests to the first server.

of the client requests and (2) finding the tail of the log and responding to clients. Doing both steps is too heavy weight for one server and sequencers on slave nodes can go faster if work is split up; this phenomenon is not uncommon and has been observed in chain replication [84].

Cluster throughput improves at the cost of decreased throughput for Sequencer 2. Figure 5.5(b) is set to sequencer mode manually (no balancing phase) and shows that the cluster throughput is worse than the cluster throughput of proxy mode. That graph also shows that Sequencer 2 has less throughput than Sequencer 1. In this case, the scatter-gather process used for cache coherence in the metadata protocols causes strain on the server housing Sequencer 2 resulting in this uneven performance.

5.3.2 Feature: Migration Units

Another factor that affects performance in this environment is how much load is on each server; these experiments quantify that effect by programming the Load Balanc-

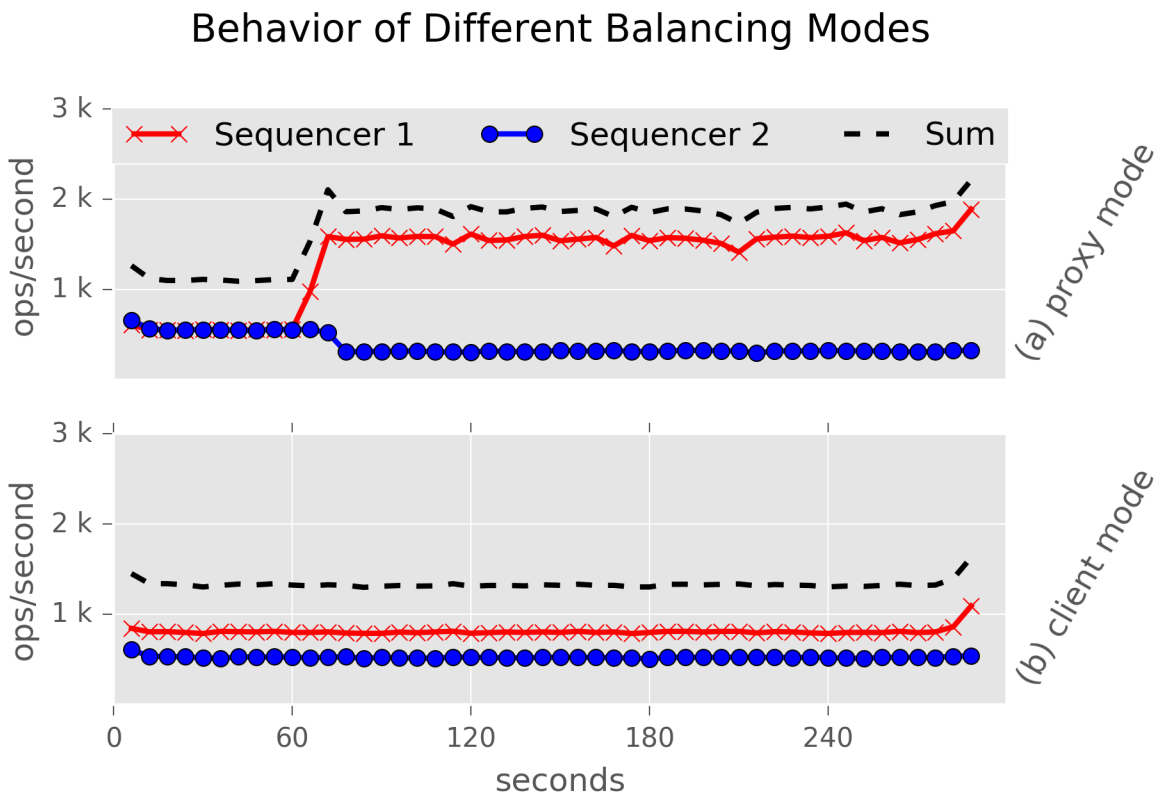


Figure 5.5: [source] The performance of proxy mode achieves the highest throughput but at the cost of lower throughput for one of the sequencers. Client mode is more fair but results in lower cluster throughput.

ing interface to control the amount of load to migrate. We call this metric a "migration unit". Expressing this heuristic is not easily achievable with outward facing tunable parameters (i.e. system knobs) but with Mantle's programmable interface, we can force the load balancer to change its migration units. To force the balancer into the Proxy Mode (Half) scenario in Figure 5.4, which uses migration units equal to half the load on the current server, we can use: `targets[whoami+1] = mds[whoami]["load"]/2`

.

This code snippet uses globally defined variables and tables from the Mantle API [75] to send half of the load on the current server (whoami) to the next ranked server (whoami + 1); the `targets` array is a globally defined table that the balancer uses to do the migrations. Alternatively, to migrate all load a time step, we can remove the division by 2.

Figure 5.3 (b) shows the performance of the modes using different migration units. Recall that this setup only has 2 sequencers and 2 servers, so performance may be different at scale. Even so, it is clear that client mode does not perform as well for read-heavy workloads. We even see a throughput improvement when migrating all load off the first server, leaving the first server to do administrative tasks (this is common in the metadata cluster because the first server does a lot of the cache coherence work) while the second server does all the processing. Proxy mode does the best in both cases and shows large performance gains when completely decoupling client request handling and operation processing in Proxy Mode (Full). The parameter that controls the migration units helps the administrator control the sequencer co-location or distribution across the cluster. This trade-off was explored extensively in the Mantle paper but the experiments we present here are indicative of an even richer set of states to explore.

5.3.3 Feature: Backoff

Tuning the aggressiveness of the load balancer decision making is also a trade-off that administrators can control and explore. The balancing phase from 0 to 60 seconds in Figure 5.2 shows different degrees of aggressiveness in making migration

decisions; CephFS makes a decision 10 seconds into the run and throughput jumps to 2500 ops/second while Mantle takes more time to stabilize. This conservative behavior is controlled by programming the balancer to (1) use different conditions for when to migrate and (2) using a threshold for sustained overload.

We control the conditions for when to migrate using `when()`, a callback in the Mantle API. For the Mantle curve in Figure 5.2 we program `when()` to wait for load on the receiving server to fall below a threshold. This makes the balancer more conservative because it takes 60 seconds for cache coherence messages to settle. The Mantle curve in Figure 5.2 also takes longer to reach peak throughput because we want the policy to wait to see how migrations affect the system before proceeding; the balancer does a migration right before 50 seconds, realizes that there is a third underloaded server, and does another migration.

The other way to change aggressiveness of the decision making is to program into the balancer a threshold for sustained overload. This forces the balancer to wait a certain number of iterations after a migration before proceeding. In Mantle, the policy would use the save state function to do a countdown after a migration. Behavior graphs and performance numbers for this backoff feature is omitted for space considerations, but our experiments confirm that the more conservative the approach the less overall throughput.

Malacology pulls the load balancing service out of the storage system to balance sequencers across a cluster. This latent capability also gives future programmers the ability to explore the different load balancing trade-offs including: load balancing

modes to control forwarding vs. client redirection, load migration units to control sequencer distribution vs. co-location, and backoffs to control conservative vs. aggressive decision making.

5.4 Domain 2: Cache Management for ParSplice

Storage systems use software-based caches to improve performance but the policies that guide what data to evict and when to evict vary with the use case. For example, caching file system metadata on clients and servers reduces the number of remote procedure calls and improves the performance of create-heavy workloads common in HPC [60, 54, 90]. But the policies for what data to evict and when to evict are specific to the application's behavior and the hardware configuration so a new workload may prove to be a poor match for the selected caching policy [94, 11, 75, 90, 89]. We evaluate a variety of caching policies using our data management language/policy engine and arrive at a customized policy that works well for our example application, ParSplice [56].

The ParSplice molecular dynamics simulation is representative of an important class of HPC applications with similar working set behaviors that extensively use software-based caches. It uses a hierarchy of caches and a single persistent key-value store to store both observed minima across a molecule's equation of motion (EOM) and the hundreds or thousands of partial trajectories calculated each second during a parallel job. This workload is pervasive across simulations that (1) rely on a mesh-based decomposition of a physical region and (2) result in millions or billions of mesh cells, where each cell contains materials, pressures, temperatures and other characteristics that are required to accurately simulate phenomena of interest. The fine-grained data annotation capabilities provided by key-value storage is a natural match for these types

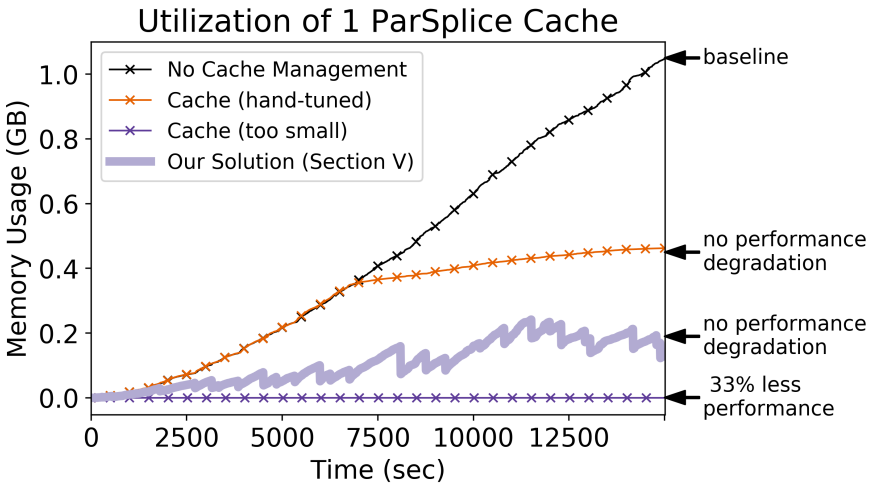


Figure 5.6: Using our data management language and policy engine, we design a dynamically sized caching policy (thick line) for ParSplice. Compared to existing configurations (thin lines with ×'s), our solution saves the most memory without sacrificing performance and works for a variety of inputs.

of scientific simulations. Unfortunately, simulations of this size saturate the capacity and bandwidth capabilities of a single node so we need more effective data management techniques.

The biggest challenge for ParSplice is properly sizing the caches in the storage hierarchy. The memory usage for a single cache that stores molecule coordinates is shown in Figure 5.6, where the thin solid lines marked with ×'s are the existing configurations in ParSplice. The default configuration uses an unlimited sized cache, shown by the "No Cache Management" line, but using this much memory for one cache is unacceptable for HPC environments, where a common goal is to keep memory for

such data structures below 3%[1]. Furthermore, ParSplice deploys a cache per 300 worker processes, so large simulations need more caches and will use even more memory. Users can configure ParSplice to evict data when the cache reaches a threshold but this solution requires tuning and parameter sweeps; the "Cache (too small)" curve in Figure 5.6 shows how a poorly configured cache can save memory but at the cost of performance, which is shown by the text annotation to the right. Even worse, this threshold changes with different initial configurations and cluster setups so tuning needs to be done for all system permutations. Our dynamically sized cache, shown by the thick line in Figure 5.6, detects key access patterns and re-sizes the cache accordingly. Without tuning or parameter sweeps, our solution saves more memory than a hand-tuned cache without any performance degradation, works for a variety of initial conditions, and could generalize to similar applications.

In this paper we are presenting the successful use of our data management language and Mantle policy engine to control the behavior of ParSplice's caches. We show that our framework:

- decomposes cache management into independent policies that can be dynamically changed, making the problem more manageable and easier to reason about.
- can deploy a variety of cache management strategies ranging from basic algorithms and heuristics to statistical models and machine learning.
- has useful primitives that, while designed for file system metadata load balancing,

[1] Anecdotally, this threshold works well for HPC applications. For reference, a 1GB cache for a distributed file system is too large in LANL deployments.

turn out to also be effective for cache management.

This last contribution is explored in Sections §5.5.1 and §5.5.2, where we try a range of policies from different disciplines; but more importantly, in Section §5.5.3, we conclude that the collection of policies we design for cache management in ParSplice are very similar to the policies used to load balance metadata in the Ceph file system (CephFS [89]) suggesting that there is potential for automatically adapting and generating policies dynamically. In this work we focus on a single node, so the "where" policy is not used. When we move ParSplice to a distributed key-value store back-end, the "where" policy will be used to determine which key-value pairs should be moved to which node.

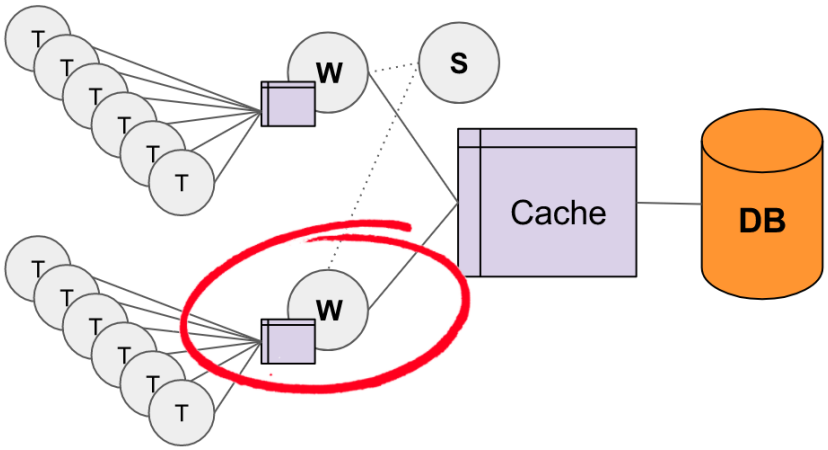


Figure 5.7: The ParSplice architecture has a storage hierarchy of caches (boxes) and a dedicated cache process (large box) backed by a persistent database (DB). A splicer (S) tells workers (W) to generate segments and workers employ tasks (T) for more parallelization. We focus on the worker's cache (circled), which facilitates communication and segment exchange between the worker and its tasks.

5.4.1 ParSplice Keyspace Analysis

ParSplice [56] is an accelerated molecular dynamics (MD) simulation package developed at LANL. It is part of the Exascale Computing Project[2] and is important to LANL's Materials for the Future initiative.

5.4.1.1 Background

As shown in Figure 5.7, the phases are:

[2]http://www.exascale.org/bdec/

1. a splicer tells workers to generate segments (short MD trajectory) for specific states

2. workers read initial coordinates for their assigned segment from data store; the key-value pair is (state ID, coordinate)

3. upon completion, workers insert final coordinates for each segment into data store, and wait for new segment assignment

The computation can be parallelized by adding more workers or by adding tasks to parallelize individual workers. The workers are stateless and read initial coordinates from the data store each time they begin generating segments. Since worker tasks do not maintain their own history, they can end up reading the same coordinates repeatedly. To mitigate the consequences of these repeated reads, ParSplice provisions a hierarchy of caches that sit in front of a single persistent database. Values are written to each tier and reads traverse up the hierarchy until they find the data.

We use ParSplice to simulate the evolution of metallic nanoparticles that grow from the vapor phase. This simulation stresses the storage hierarchy more than other input decks because it uses a cheap potential, has a small number of atoms, and operates in a complex energy landscape with many accessible states. As the run progresses, the energy landscape of the system becomes more complex and more states are visited. Two domain factors control the number of entries in the data store: the growth rate and the temperature. The growth rate controls how quickly new atoms are added to the nanoparticle: fast growth rates lead to non-equilibrium conditions, and hence

increase the number of states that can be visited. However, as the particle grows, the simulation slows down because the calculations become more expensive, limiting the rate at which new states are visited. On the other hand, the temperature controls how easily a trajectory can jump from state to state; higher temperatures lead to more frequent transitions but temperatures that are too high result in meaningless simulations because trajectories have so much energy that they are equally likely to visit any random state.

5.4.1.2 Experimental Setup

We instrumented ParSplice with performance counters and keyspace counters. The performance counters track ParSplice progress while keyspace counters track which keys are being accessed by the ParSplice ranks. Because the keyspace counters have high overhead we only turn them on for the keyspace analysis.

All experiments ran on Trinitite, a Cray XC40 with 32 Intel Haswell 2.3GHz cores per node. Each node has 128GB of RAM and our goal is to limit the size of the cache to 3% of RAM. Note that this is an addition to the 30GB that ParSplice uses to manage other ranks on the same node. The scalability experiment uses 1 splicer, 1 persistent database, 1 cache process, and up to 2 workers. We scale up to 1024 tasks, which spans 32 nodes and disable hyper-threading because we experience unacceptable variability in performance. For the rest of the experiments, we use 8 nodes, 1 splicer, 1 persistent database, 1 cache process, 1 worker, and up to 256 tasks. The keyspace analysis that follows is for the cache on the worker node, which is circled in Figure 5.7.

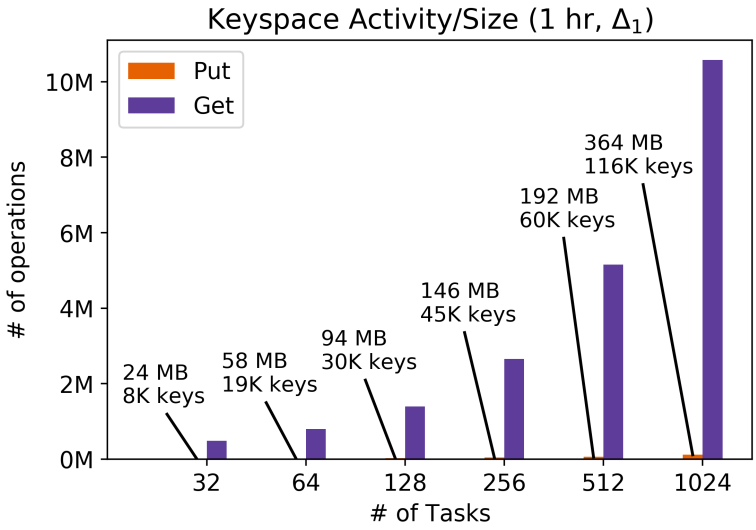


Figure 5.8: The keyspace is small but must satisfy many reads as workers calculate segments. Memory usage scales linearly, so it is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

5.4.1.3 Results and Observations

Our analysis shows that ParSplice accesses keys in a structured and predictable way. The following 4 observations shape the policies we design later in the paper.

Scalability Figure 5.8 shows the keyspace size (text annotations) and request load (bars) after a one hour run with a different number of tasks (x axis). While the keyspace size and capacity is modest the memory usage scales linearly with the number of tasks, which is a problem if we want to scale to Trinitite's 3000 cores. Furthermore, the size of the keyspace also increases linearly with the length of the run. Extrapolating these results puts an 8 hour run across all 100 Trinitite nodes at 8GB for one cache.

This memory utilization easily eclipses the 3% memory usage per node threshold we set earlier, even without factoring in the usage from other workers.

An active but small keyspace

The bars in Figure 5.8 show $50 - 100\times$ as many reads (`get()`) as writes (`put()`). Tasks read the same key for extended periods because the trajectory gets stuck in so-called superbasins composed of tightly connected sets of states. Writes only occur for the final state of segments generated by tasks; their magnitude is smaller than reads because the caches ignore redundant write requests.

Initial conditions influence key activity Figure 5.9 shows how ParSplice tasks read key-value pairs from the worker's cache for two different initial conditions of Δ, which is the rate that new atoms enter the simulation. The line is the read request rate ($y1$ axis) and the dots along the bottom are the number of unique keys accessed ($y2$ axis). The access patterns for different growth rates have temporal locality, as the reads per second for Δ_2 look like the reads per second for Δ_1 stretched out along the time axis. The Δ_1 growth rate adds atoms every 100K microseconds while the Δ_2 growth rate adds atoms every 1 million microseconds. So Δ_2 has a smaller growth rate resulting in hotter keys and a smaller keyspace. Values smaller than Δ_2's growth rate or a temperature of 400 degrees result in very little database activity because state transitions take too long. Similarly, values larger than Δ_1's growth rate or a temperature of 4000 degrees result in an equally meaningless simulation as transitions are unrealistic.

This figure demonstrates that small changes to Δ can have a strong effect on the timing and frequency with which new EOM minima are discovered and referenced.

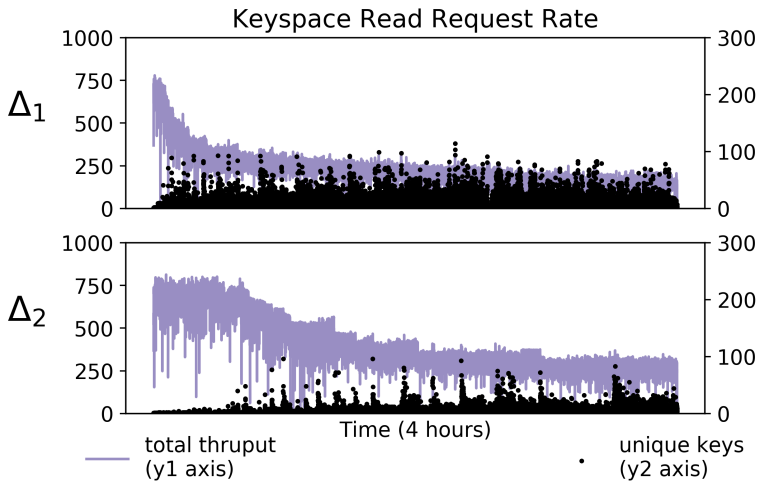


Figure 5.9: Key activity for ParSplice starts with many reads to a small set of keys and progresses to less reads to a larger set of keys. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of unique keys accessed in a 1 second sliding window ($y2$ axis). Despite having different growth rates (Δ), the structure and behavior of the key activities are similar.

Trends also exist for temperature and number of workers but are omitted here for space. This finding suggests that we need a flexible policy language and engine to explore these trade-offs.

Entropy increases over time The reads per second in Figure 5.9 show that the number of requests decreases and the number of active keys increases over time. The number of read and write requests are highest at the beginning of the run when tasks generate segments for the same state, which is computationally cheap (this motivates

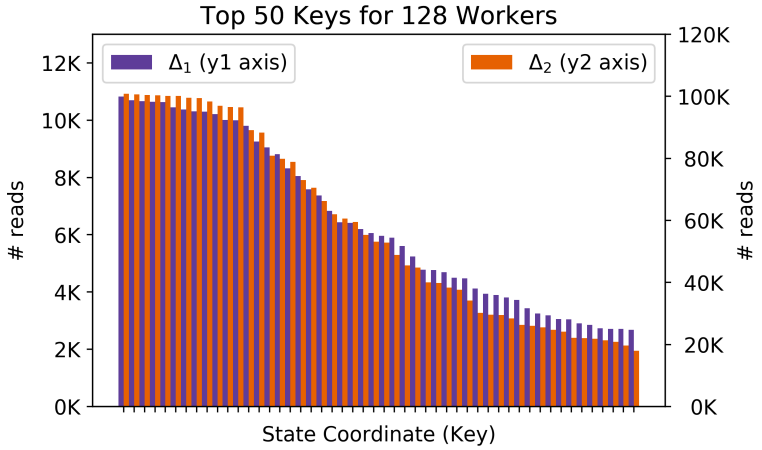


Figure 5.10: Over time, tasks start to access a larger set of keys resulting in some keys being more popular than others. Despite different growth rates (Δ), the spatial locality of key accesses is similar between the two runs. (*e.g.*, some keys are still read 5 times as many times others).

Section §5.5.1). The resulting key access imbalance for the two growth rates in Figure 5.9 are shown in Figure 5.10, where reads are plotted for each unique state, or key, along the x axis. Keys are more popular than others (up to 5×) because worker tasks start generating states with different coordinates later in the run. Figure 5.10 also shows that the number of reads changes with different initial conditions (Δ), but that the spatial locality of key accesses is similar (*e.g.*, some keys are still 5× more popular than others).

5.4.2 Integrating Mantle into ParSplice

Using Mantle cluster metrics, we expose cache size, CPU utilization, and memory pressure of the worker node to the cache management policies. In Section §5.5.1 we only end up using the cache size although the other metrics proved to be valuable debugging tools. Using Mantle time series metrics, we expose accesses to the cache as a list of `timestamp, key` pairs. In Section §5.5.2, we explore a key access pattern detection algorithm that uses this metric.

We link Mantle into all caches in the system and put the "when" and "how much" callbacks alongside code that checks for memory pressure. It is executed right before the worker processes incoming and outgoing put/get transactions to the cache. We only do cache management once every second to avoid maintaining the cache for every request. We expected to have to increase this polling interval to accommodate more complex policies but even our most complicated policy in Section §5.5.2 had a negligible effect on performance when executed every second (within the standard deviation for multiple runs when compared against a policy that returns immediately). This may be because the worker is not overloaded and the bottleneck is somewhere else in the system. As stated previously, we do not use the "where" part of Mantle because we focus on a single node, but this part of the API will be used when we move the caches and storage nodes to a key-values store back-end that uses key load balancing and repartitioning.

5.5 Domain 2: Evaluation

5.5.1 Storage System Architecture Knowledge

Using the Mantle policy engine, we test a variety of cache management algorithms on the worker using the keyspace analysis in Section §5.4.1.3. Our evaluation uses the total "trajectory length" as the goodness metric. This value is the duration of the overall trajectory produced by ParSplice. At ideal efficiency, the trajectory length should increase with the square root of the wall-clock time, since the wall-clock cost of time-stepping the system by one simulation time unit increases in proportion of the total number of atoms. The policy should avoid reducing the trajectory length and be fast enough to run as often as we want to detect key access patterns. First we size the cache according to our system specific knowledge, *i.e.* the hardware and software of the storage hierarchy.

We implement a basic LRU cache using a "when" policy of:

```
server[whoami]['cachesize']>n
```

and a "how much" policy of:

```
servers[whoami]['cachesize']-n
```

The results for different cache sizes for a growth rate of Δ_1 over a 2.5 hour run across 256 workers is shown in Figure 5.11. "Baseline" is the performance of unmodified ParSplice measured in trajectory duration ($y1$ axis) and utilization is measured with memory footprint of just the cache ($y2$ axis). The middle graph labeled "Fixed

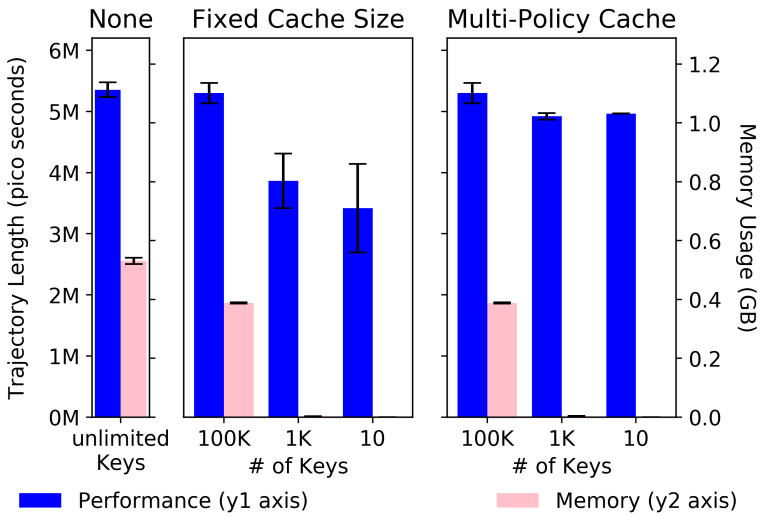


Figure 5.11: Policy performance/utilization shows the trade-offs of different sized caches (x axis). "None" is ParSplice unmodified, "Fixed Sized Cache" evicts keys using LRU, and "Multi-Policy Cache" switches to fixed sized cache after absorbing the workload's initial burstiness. This parameter sweep identifies the "Multi-Policy Cache" of 1K keys as the best solution but this only works for this system setup and initial configurations.

Cache Size" shares the y axes and shows the trade-off of using a basic LRU-style cache of different sizes, where the penalty for a cache miss is retrieving the data from the persistent database. The error bars are the standard deviation of 3 runs. Although the keyspace grows to 150K, a 100K key cache achieves 99% of the performance. Decreasing the cache degrades performance and predictability.

But the top graph in Figure 5.9 suggests that a smaller cache size should suffice, as only 100 keys seem to be active at any one time. It turns out that the unique keys plotted in Figure 5.9 are per second and are not representative of the actual active

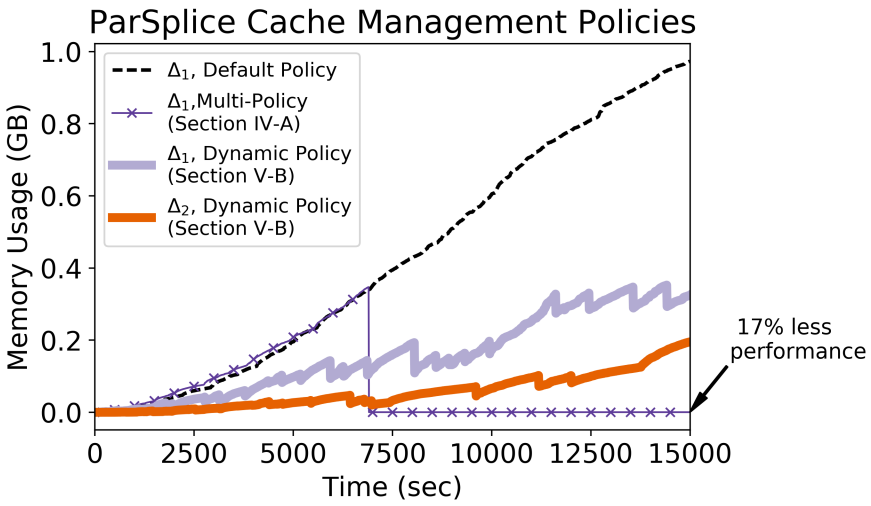


Figure 5.12: Memory utilization for "No Cache Management" (unlimited cache growth), "Multi-Policy" (absorbs initial burstiness of workload), and "Dynamic Policy" (sizes cache according to key access patterns). The dynamic policies saves the most memory without sacrificing performance.

keyspace; the number of active keys is larger than 100, as some keys may be accessed at time t_0, not in t_1, and then again in t_2. Because the cache is too small, reads and writes fall through to the rest of the storage hierarchy and the excessive traffic triggers a LevelDB compaction on the persistent database. To avoid these compactions, which temporarily block operations, we design a multi-policy cache that switches between:

- unlimited growth policy: cache increases on every write
- n key limit policy: cache constrained to n keys

The key observation is that small caches incur too much load on the persistent database at the beginning of the run but should suffice after the initial read flash crowd

passes because the keyspace is far less active. We program Mantle to trigger the policy switch at 100K keys to absorb the flash crowd at the beginning of the run. Once triggered, keys are evicted to bring the size of the cache down to the threshold. The actual policy is shown and described in more detail in Section §5.5.3 in Figure 5.16. The plot on the right side of Figure 5.11 shows the performance/utilization trade-off of the multi-policy cache, where the cache sizes for the n key limit policy are along the x axis. The performance and memory utilization for a 100K key cache size is the same as the 100K bar in the "Fixed Cache Size" graph in Figure 5.11 but the rest reduce the size of the keyspace after the read flash crowd. We see the worst performance when the policy switches to the 10 key limit policy, which achieves 94% of the performance while only using 40KB of memory.

Caveats:

The results in Figure 5.11 are slightly deceiving for two reasons: (1) segments take longer to generate later in the run and (2) the memory footprint is the value at the end of 2.5 hours. For (1), the trajectory length vs. wall-clock time curves down over time; as the nanoparticle grows it takes longer to generate segments so by the time we reach 2 hours, over 90% of the trajectory is already generated. For (2), the memory footprint rises until it reaches the 100K key switch threshold at 0.4GB and then reduces to the final value after switching policies. The memory usage over time for this policy is shown by the "Δ_1, Multi-Policy" curve in Figure 5.12 but in Figure 5.11 we plot the final value. Despite these caveats, the result is still valid: we found a multi-policy cache management strategy that absorbs the cost of a high read throughput on a small

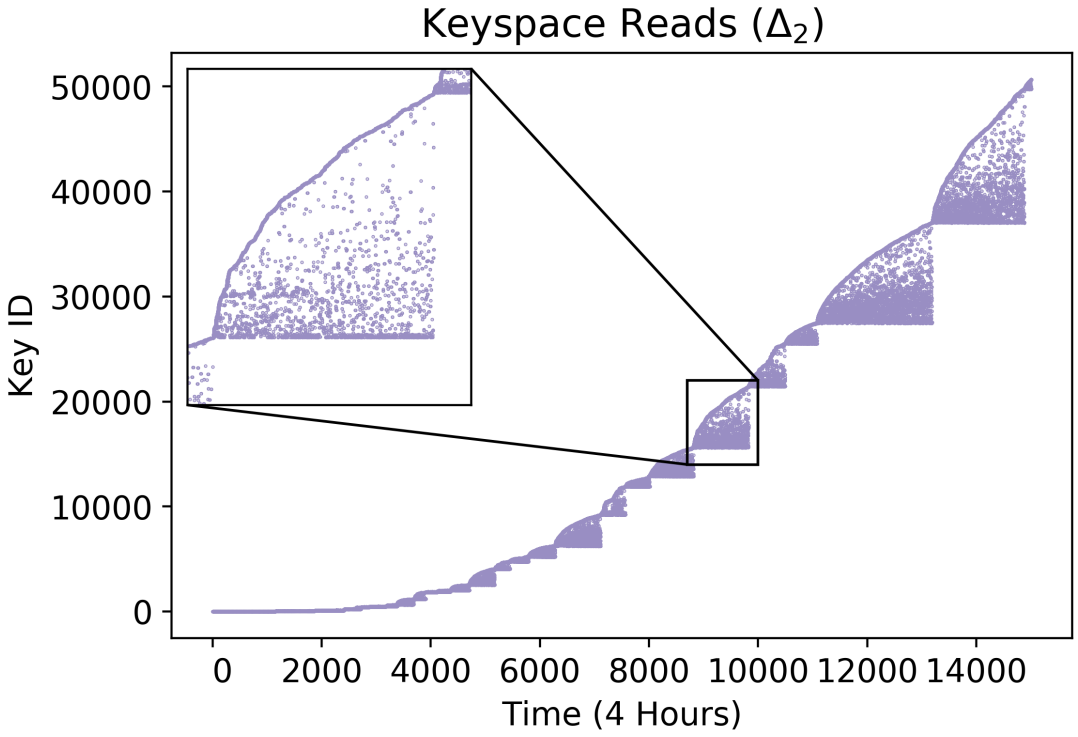


Figure 5.13: Key activity for a 4 hour run shows groups of accesses to the same subset of keys. Detecting these access patterns leads to a more accurate cache management strategy, which is discussed in Section §5.5.2.2 and the results are in Figure 5.14.

keyspace and reduces the memory pressure for a 2.5 hour run. To improve the policy even more, we need a way to identify what thresholds to use for different system setups (*e.g.*, different ParSplice parameters, number of worker tasks, and job lengths).

5.5.2 Application-Specific Knowledge

Feeding application-specific knowledge about ParSplice into a policy leads to a more accurate cache management strategy. The goal of the following section is not

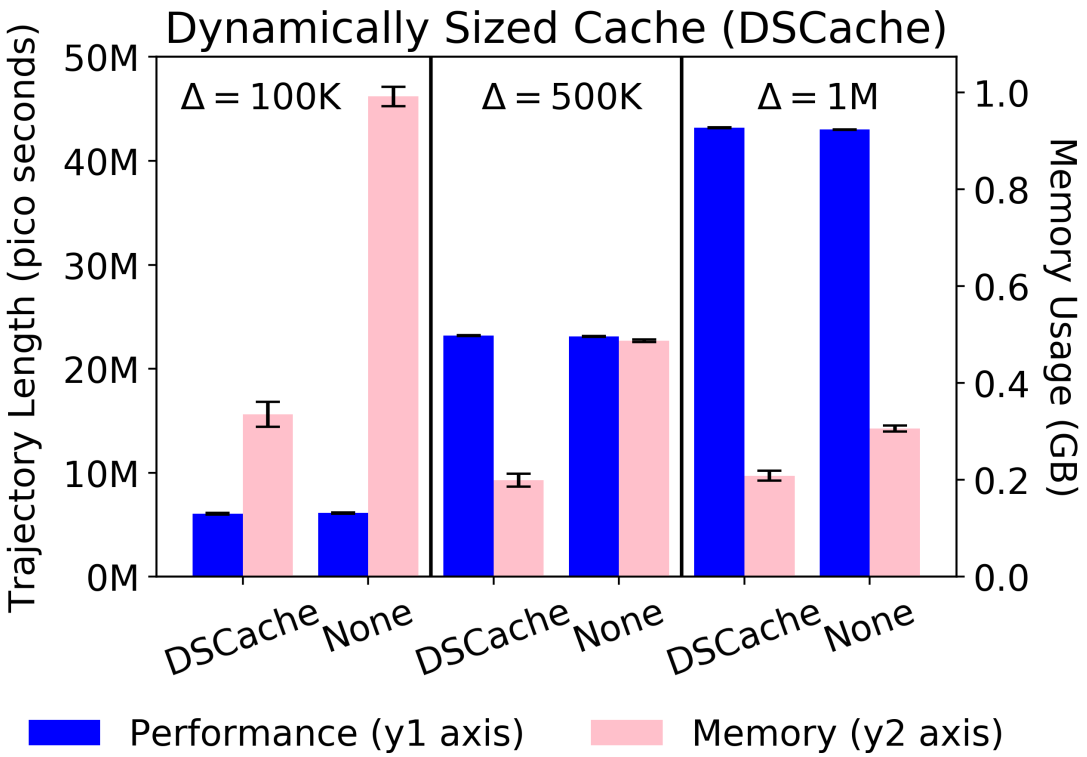


Figure 5.14: The performance/utilization for the dynamically sized cache (DSCache) policy. With negligible performance degradation, DSCache adjusts to different initial configurations (Δ's) and saves 3$\times$ as much memory in the best case.

to find an optimal solution, as this can be done with parameter sweeps for thresholds; rather, we try to find techniques that work for a range of inputs and system setups.

Figure 5.13 shows which keys (y axis) are accessed by tasks over time (x axis). The groups of accesses to a subset of keys occurs because molecules are stuck in deep trajectories. Recall that the cache stores the molecules' EOM minima, which is the smallest effective energy that a molecule observes during its trajectory. So molecules

stuck in deep trajectories explore the same minima until they can escape to a new set of states. This exploration of the same set of states is called a superbasin. In Figure 5.13, superbasins are never re-visited because the simulation only adds molecules; we can never reach a state with less molecules. This is why keys are never re-accessed.

Detecting these superbasins can lead to more effective cache management strategies because the height of the groups of key accesses is "how much" of the cache to evict and the width of the groups of key accesses is "when" to evict values from the cache. The zoomed portion of Figure 5.13 shows how a single superbasin affects the key accesses. Moving along the x axis shows that the number of unique keys accessed over time grows while moving along the y axis shows that early keys are accessed more often. Despite these patterns, the following characteristics of superbasins make them hard to detect:

- superbasin key accesses are random and there is no threshold "minimum distance between key access" that indicates we have moved on to a new superbasin
- superbasins change immediately
- the number of keys a superbasin accesses differs from other superbasins

5.5.2.1 Failed Strategies

To detect the access patterns in Figure 5.13, we try a variety of techniques using Mantle. Unfortunately, we found that the following techniques proliferate more parameters that need to be tuned per hardware/software configuration. Furthermore,

many of the metrics do not signal a new set of key accesses. Below, we indicate with quotes which parameters we need to add for each technique and the value we find to work best, via tuning and parameter sweeps, for one set of initial conditions.

- Statistics: decay on each key counts down until 0; 0-valued keys are evicted. "history-of-key-accesses", set to 10 seconds, to evict keys.
- Calculus: use derivative to strip away magnitudes; use large positive slopes followed by large negative slope as signal for new set of key accesses. "Zero-crossing", set to 40 seconds, for distance between small/large spikes to avoid false positives; "window size", set to 200 seconds, for the size of the moving average.
- K-Means Clustering fails because "K" is not known *a-priori* and groups of key accesses are different size. "K", set to 4, for the number of clusters in the data using the sum of the distances to the centroid.
- DBScan: finds clusters using density as a metric. "Eps", set to 20, for max distance between 2 samples in same neighborhood; "Min", set to 5, for the samples per core.
- Edge Detection: size of the image is too big and bottom edges are not thick enough.

5.5.2.2 Dynamically Sized Cache: Access Pattern Detection

After trying these techniques we found that the basic O(n) algorithm in Figure 5.15 works best. The algorithm detects groups of key accesses, which we call "fans",

by iterating backwards through the key access trace, finding the lowest key ID, and comparing against the lowest key ID we have seen so far (Line 7). We also maintain the top and bottom of each group of key accesses (Line 13) so we can tell the "how much" policy the number of keys to evict (Line 23). The algorithm is O(n), where n is the number events, but the benefit is that the approach avoids adding new thresholds for key access pattern detection (*e.g.*, space between key accesses, space between key IDs, and window size of consecutive key accesses).

The algorithm iterates backwards over the key access trace because a change in the minimum value signals a new group of key accesses. No signal exists iterating left to right, as the maximum value always increases and the minimum values at the bottom of each group of key accesses are sparse. For example, the maximum distance between values along the bottom edge of the zoomed group of key accesses in Figure 5.13 is 125 seconds, while the maximum distance between minimum values for the group of key accesses before is 0 seconds. As a result of this sparseness, iterating left to right requires a "window size" parameter to determine when we think a minimum value will not show up again.

The performance and memory utilization is shown by the "DSCache" bars in Figure 5.14. Without sacrificing performance (trajectory length), the dynamically sized cache policy uses between 32%-66% less memory than the default ParSplice configuration (no cache management) for the 3 initial conditions we test. The memory usage over time is shown by the "Dynamic Policy" curves in Figure 5.12, where the behavior

resembles the key access patterns in Figure 5.13[3]. We also show a Δ_2 growth rate to demonstrate the dynamic policy's ability to adjust to a different set of initial conditions.

[3]The memory usage is not *exactly* the same because these are two different runs; Figure 5.13 has key activity tracing turned on, which reduces performance.

```
d = timeseries()
ts, id = d:get(d:size())
fan  = {start=nil, finish=ts, top=0, bot=id}
fans = {}
for i=d:size(),1,-1 do      -- iterate backwards
  ts, id = d:get(i)
  if id < fan['bot'] then  -- found a new fan!
    fan['start']  = ts
    fans[#fans+1] = fan
    fan = {start=nil, finish=ts, top=0, bot=id}
  end

  if id > fan['top'] then  -- track top of fan
    fan['top'] = id
  end
end
fan['start'] = 0
fans[#fans+1] = fan

if #fans < 2 then -- do not evict current fan
  return false
else
  WRstate(fans[#fans-1]['top']-fans[1]['bot'])
  return true
end
```

Figure 5.15: The dynamically sized cache policy iterates backwards over timestamp-key pairs and detects when accesses move on to a new subset of keys (*i.e.* "fans"). The performance and total memory usage is in Figure 5.14 and the memory usage over time is in Figure 5.12.

```
function when()
  if server[whoami]['cachesize'] > n then
    if server[whoami]['cachesize'] > 100K then
      WRstate(1)
    end
    if RDstate() == 1 then
      return true
    end
  end
  return false
end
```

Figure 5.16: ParSplice cache management policy that absorbs the burstiness of the workload before switching to a constrained cache. The performance/utilization for different n is in Figure 5.11.

5.5.3 Towards General Data Management Policies

In the previous section, we used our data management language and the Mantle policy engine to design effective cache management strategies for a new application and storage system. In this section, we compare and contrast the policies examined for file system metadata load balancing in [75] with the ones we designed and evaluated above for cache management in ParSplice.

5.5.3.1 Using Load Balancing Policies for Cache Management

From a high-level the cache management policy we designed in Figure 5.16 trims the cache if the cache reaches a certain size *and* if it has already absorbed the initial burstiness of the workload. Much of this implementation was inspired by the CephFS metadata load balancing policy in Figure 5.17, which was presented in [75]. That policy migrates file system metadata if the load is higher than the average load in the cluster *and* the current server has been overloaded for more than two iterations.

```
1  local function when()
2    if servers[whoami]["load"] > target then
3      overloaded = RDstate() + 1
4      WRstate(overloaded)
5      if overloaded > 2 then
6        return true
7      end
8    end
9    else then WRstate(0) end
10   return false
11 end
```

Figure 5.17: CephFS file system metadata load balancer, designed in 2004 in [90], reimplemented in Lua in [75]. This policy has many similarities to the ParSplice cache management policy.

The two policies have the following in common:

Condition for "Overloaded" (Fig. 5.16: Line 2; Fig. 5.17: Line 2) - these lines detect whether the node is overloaded using the load calculated in the load callback (not shown). While the calculations and thresholds are different, the way the loads are used is exactly the same; the ParSplice policy flags the node as overloaded if the cache reaches a certain size while the CephFS policy compares the load to other nodes in the system.

State Persisted Across Decisions (Fig. 5.16: Lines 4,6; Fig 5.17: Lines 3,4,9) - these lines use Mantle to write/read state from previous decisions. For ParSplice, we save a boolean that indicates whether we have absorbed the workload's initial burstiness. For CephFS, we save the number of consecutive instances that the server has been overloaded. We also clear the count (Line 9) if the server is no longer overloaded.

Multi-Policy Strategy (Fig. 5.16: Line 6; Fig. 5.17: Line 5) - after determining that the node is overloaded, these lines add an additional condition before the

policy enters a data management state. ParSplice trims its cache once it eclipses the "absorb" threshold while CephFS allows balancing when overloaded for more than two iterations. The persistent state is essential for both of these policy-switching conditions.

These similarities among effective policies for two very different domains suggest that the heuristics and techniques in other load balancers can be used for cache management. The result supports the notion that concepts and problems that architects grapple with are transcendent across domains and the solutions they design can be re-used in different code bases.

5.5.3.2 Using Cache Management Policies for Load Balancing

The cache management policies we developed earlier can be used by load balancing policies to effectively spread load across a cluster. For example, distributed file systems that load balance file system metadata across a dedicated metadata cluster could use the caching policies to determine what metadata to move and when to move it. To demonstrate this idea, we analyze a 3-day Lustre file system metadata trace, collected at LANL. The trace is anonymized so all file names are replaced with a unique identifier and we do not know which applications are running. We visualize a 1 hour window of the trace in Figure 5.18, where the dots are the file system metadata reads in a 1 hour window. The x axis is time and the y axis is the file ID, listed in the order that file IDs appear in the trace. The groups of accesses look similar to the ParSplice key accesses in Figure 5.13.

Although other access pattern detection algorithms are possible, we use the

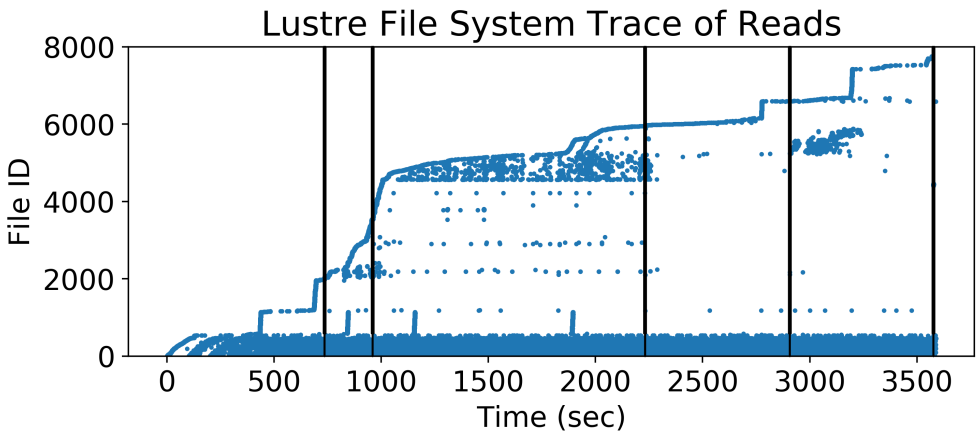


Figure 5.18: File system metadata reads for a Lustre trace collected at LANL. The vertical lines are the access patterns detected by the ParSplice cache management policy from Section §5.5.2. A file system that load balances metadata across a cluster of servers could use the same pattern detection to make migration decisions, such as avoiding migration when the workload is accessing the same subset of keys or keeping groups of accesses local to a server.

one designed for cache management in Section §5.5.2.2 with slight modifications based on our knowledge of file systems[4]. The vertical lines in Figure 5.18 are the groups of accesses identified by the algorithm; it successfully detects the largest group of key accesses that starts at time 1000 seconds and ends at time 2200 seconds. File systems that load balance file system metadata across a cluster would want to keep metadata in that group of key accesses on the same server for locality and would want to avoid migrating metadata to a different server until the group of key accesses completes.

[4]We filtered out requests for key IDs less than 2000, as these are most likely path traversal requests to higher parts of the namespace.

Before we showed how policies designed for load balancing heavily influence our cache management in a different application and storage system. But in this section we show how an *unmodified* cache management policy can be used in a load balancing strategy. This generalization may reduce the work that needs to be done for load balancing as ideas may have already been explored in other domains and could work "out-of-the-box".

5.5.3.3 Other Use Cases

Storage systems have many other data management techniques that would benefit from the caching policies developed in Sections §5.5.1 and §5.5.2. For example, Ceph administrators can use the policies in ParSplice to automatically size and manage cache tiers[5], caching on object storage devices, or in the distributed block devices[6]. Integration with Mantle would be straightforward as it is merged into Ceph's mainline[7] and the three caching subsystems mentioned above already maintain key access traces.

More generally, the similarities between load balancing and cache management show how the "when"/"where"/"how much" abstractions, data management language, and policy engine may be widely applicable to other data management techniques, such as:

- QoS: when to move clients, where to move clients, how much of the reservation to move. We could use Mantle to implement something like the reservation algo-

[5]http://docs.ceph.com/docs/master/rados/operations/cache-tiering/
[6]http://docs.ceph.com/docs/master/rbd/rbd-config-ref/
[7]http://docs.ceph.com/docs/master/cephfs/mantle/

rithms based on utilization and period in Fahrrad [58] to achieve better guarantees without sacrificing performance.

- Scheduling: when to yield computation cycles to another process, how much of a resource to allocate. We could use Mantle to implement the fairness/priority models used in the Mesos [34] "how many" policies.

- Batching: how many operations to group together, when to send large batches of updates. We could use Mantle to implement pathname leases from IndexFS [60] or the capabilities from CephFS[8].

- Prefetching: how much to prefetch, how to select data. We could use Mantle to implement forward/backward/stride detection algorithms for prefetching in RAID arrays or something more complicated, like the time series algorithms for adaptive I/O prefetching from [83].

[8] http://docs.ceph.com/docs/master/cephfs/capabilities/

5.5.4 Related Work

Key-value storage organizations for scientific applications is a field gaining rapid interest. In particular, the analysis of the ParSplice keyspace and the development of an appropriate scheme for load balancing is a direct response to a case study for computation caching in scientific applications [39]. In that work the authors motivated the need for a flexible load balancing *microservice* to efficiently scale a memoization microservice. Our work is also heavily influenced by the Malacology project [74] which seeks to provide fundamental services from within the storage system (*e.g.*, consensus) to the application. Our plan is to use MDHIM [27] as our back-end key-value store because it was designed for HPC and has the proper mechanisms for migration already implemented.

State-of-the-art distributed file systems partition write-heavy workloads and replicate read-heavy workloads, similar to the approach we are advocating here. IndexFS [60] partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal. ShardFS [94] takes the replication approach to the extreme by copying all directory state to all nodes. CephFS [90, 89] employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies are controlled with tunable parameters. These systems still need to be tuned by hand with *ad-hoc* policies designed for specific applications. Setting policies for migrations is arguably more difficult than adding the migration mechanisms themselves. For example, IndexFS/CephFS use the GIGA+ [53]

technique for partitioning directories at a *predefined* threshold. Mantle makes headway in this space by providing a framework for exploring these policies, but does not attempt anything more sophisticated (*e.g.*, machine learning) to create these policies.

Auto-tuning is a well-known technique used in HPC [9, 8], big data systems systems [32], and databases [68]. Like our work, these systems focus on the physical design of the storage (*e.g.* cache size) but since we focused on a relatively small set of parameters (cache size, migration thresholds), we did not need anything as sophisticated as the genetic algorithm used in [9]. We cannot drop these techniques into ParSplice because the magnitude and speed of the workload hotspots/flash crowds makes existing approaches less applicable.

5.5.5 Conclusion

Data management encompasses a wide range of techniques that vary by application and storage system. Yet, the techniques require policies that shape the decision making and finding the best policies is a difficult, multi-dimensional problem. We iterate to a custom solution for our target application that uses workload access patterns to size its caches. Without tuning or parameter sweeps, our solution saves memory without sacrificing performance for a variety of initial conditions, including the scale, duration, configuration, and hardware of the simulation. More importantly, rather than attempting to construct a single, complex policy that works for a variety of scenarios, we instead use the Mantle framework to enable software-defined storage systems to flexibly change policies as the workload changes. We also observe that many of the primitives

and strategies have enough in common with data management in file systems that they both can be expressed with similar semantics.

This lays the foundation for future work, where we will focus on formalizing a collection of general data management policies that can be used across applications and storage systems. The value of such a collection eases the burden of policy development and paves the way for automated solutions such as (1) adaptable policies that switch to new strategies when the current strategy behaves poorly (*e.g.*, thrashing, making no progress, etc.), and (2) policy generation, where new policies are constructed by examining the collection of existing policies. Ultimately, we hope that this automation enables control of policies by machines instead of administrators.

Chapter 6

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

File system metadata services in HPC and large-scale data centers have scalability problems because common tasks, like checkpointing [10] or scanning the file system [98], contend for the same directories and inodes. Applications perform better with dedicated metadata servers [75, 60] but provisioning a metadata server for every client[1] is unreasonable. This problem is exacerbated by current hardware and software trends; for example, HPC architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to more simplified stacks with just a burst buffer and object store [55]. These types of trends put pressure on data

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

access because more requests from different nodes end up hitting the same software layers in parallel 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, many HPC batch jobs do not need the strong consistency that the file system provides, so BatchFS [98] and DeltaFS [99] do more client-side processing and merge updates when the job is done. Developers in these domains are turning to these non-POSIX IO solutions because their applications are well-understood (*e.g.*, well-defined read/write phases, synchronization only needed during certain phases, workflows describing computation, etc.) and because these applications wreak havoc on file systems designed for general-purpose workloads (*e.g.*, checkpoint-restart's N:N and N:1 create patterns [10]).

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 [29, 99, 98, 60, 23]. Then the file system can change its internal structure to optimize performance. For example, the file system could enter a mode where clients with decoupled directories perform operations locally and bulk merge their updates at completion. This delayed merge (*i.e.* a form of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of remote procedure calls (RPCs), synchronization, false sharing, and serialization. While the performance benefits of decoupling the namespace are obvious, applications that

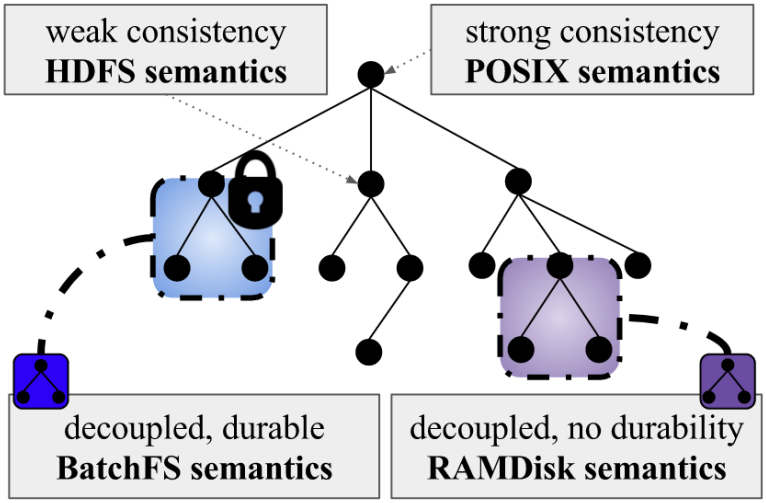


Figure 6.1: Illustration of subtrees with different semantics co-existing in a global namespace. For performance, clients relax consistency/durability on their subtree (*e.g.*, HDFS) or decouple the subtree and move it locally (*e.g.*, BatchFS, RAMDisk).

rely on the file system's guarantees must be deployed on an entirely different system or re-written to coordinate strong consistency/durability themselves.

To address this problem, we present an API and framework that lets administrators dynamically control the consistency and durability guarantees for subtrees in the file system namespace. Figure 6.1 shows a potential setup in our proposed system where a single global namespace has subtrees for applications optimized with techniques from different state-of-the-art architectures. The BatchFS and RAMDisk subtrees are decoupled from the global namespace and have similar consistency/durability behavior to those systems; the HDFS subtree has weaker than strong consistency because it lets clients read files opened for writing [30], which means that not all updates are immedi-

ately seen by all clients; and the POSIX IO subtree retains the rigidity of POSIX IO's strong consistency. Subtrees without policies inherit the consistency/durability semantics of the parent and future work will examine embeddable or inheritable policies.

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

1. A framework/API for assigning consistency/durability policies to subtrees in a global namespace; this lets administrators navigate trade-offs of different metadata protocols on the same storage system.

2. We show that letting different semantics co-exist in a global namespace scales further and performs better than systems that use one strategy.

3. A prototype that lets administrators custom fit subtrees to applications dynamically.

The results in this paper confirm the assertions of "clean-slate" research of decoupled namespaces; specifically that these techniques drastically improve performance. We go a step further by quantifying the costs of traditional file system approaches to maintaining consistency (3.37× slowdown) and durability (2.4× slowdown). In our prototype, we also show the benefits of assigning subtree semantics to certain applications such as checkpoint-restart (91.7× speedup if consistency is fully relaxed), user home

directories (within a 0.03 standard deviation from optimal), and end-users checking for partial results (only a 2% overhead). We use Ceph as a prototyping platform because it is used in cloud-based and data center systems and has a presence in HPC [85].

6.1 POSIX IO Overheads

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

6.1.1 Durability

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

configuration.

CephFS Design: A journal of metadata updates that streams into the resilient object store. Similar to LFS [63] and WAFL [35] the metadata journal is designed to be large (on the order of MBs) which ensures (1) sequential writes into the object store and (2) the ability for daemons to trim redundant or irrelevant journal entries. The journal is striped over objects where multiple journal updates can reside on the same object. There are two tunables, related to groups of journal events called segments, for controlling the journal: the segment size and the dispatch size (*i.e.* the number of segments that can be dispatched at once). Unless the journal saturates memory or CPU resources, larger values for these tunables result in better performance.

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

Figure 6.2 shows the effect of journaling with different dispatch sizes, normalized to 1 client that creates 100K files with journaling off (about 654 creates/sec). In this case a dispatch size of 30 degrades performance the most because the metadata server is overloaded with requests and cannot spare cycles to manage concurrent segments. Tuning and parameter sweeps show that a dispatch size of 10 is the worst and that larger sizes approach a dispatch size of 1; for all future journal experiments we use a dispatch size of 40 which is a more realistic configuration. Although the "no journal"

curve appears flat, it is actually a slowdown of about 0.3× per concurrent client; this slowdown is a result of the peak throughput of a single metadata server, which we found to be about 3000 operations per second. The trade-off for better performance is memory consumption because a larger dispatch size uses more space for buffering.

Comparison to decoupled namespaces: For BatchFS, if a client fails when it is writing to the local log-structured merge tree (implemented as an SSTable [59]) then unwritten metadata operations are lost. For DeltaFS, if the client fails then, on restart, the computation does the work again – since the snapshots of the namespace are never globally consistent and there is no ground truth. On the server side, BatchFS and DeltaFS use IndexFS [60]. IndexFS writes metadata to SSTables, which initially reside in memory but are later flushed to the underlying distributed file system.

6.1.2 Strong Consistency

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

CephFS Design: Capabilities keep metadata strongly consistent. To reduce the number of RPCs needed for consistency, clients can obtain capabilities for reading and writing inodes, as well as caching reads, buffering writes, changing the file size, and performing lazy IO. To keep track of the read caching and write buffering capabilities, the clients and metadata servers agree on the state of each inode using an inode cache.

If a client has the directory inode cached it can do metadata writes (*e.g.*, create) with a single RPC. If the client is not caching the directory inode then it must do an extra RPC to determine if the file exists. Unless the client immediately reads all the inodes in the cache (*i.e.* `ls -alR`), the inode cache is less useful for create-heavy workloads.

Figure 6.3 shows the slowdown of maintaining strong consistency when scaling the number of clients. We plot the slowdown of the slowest client, normalized to 1 client that creates 100K files (about 513 creates/sec because the journal is turned back on). For the "interference" curve, each client creates files in private directories and at 30 seconds we launch another process that creates files in those directories. 20 clients has an asterisk because the maximum number of clients the metadata server can handle for this metadata-intensive workload is actually 18; at higher client load, the metadata server complains about laggy and unresponsive requests.

The cause for this slowdown is shown in Figure 6.4. The colors show the behavior of the client for two different runs. If only one client is creating files in a directory ("no interference" curve on $y1$ axis) then that client can lookup the existence of new files locally before issuing a create request to the metadata server. If another client starts creating files in the same directory then the directory inode transitions out of read caching and the first client must send `lookup()`s to the metadata server ("interference" curve on $y2$ axis). These extra requests increase the throughput of the "interference" curve on the $y1$ axis because the metadata server can handle the extra load but performance suffers.

Comparison to decoupled namespaces: Decoupled namespaces merge

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

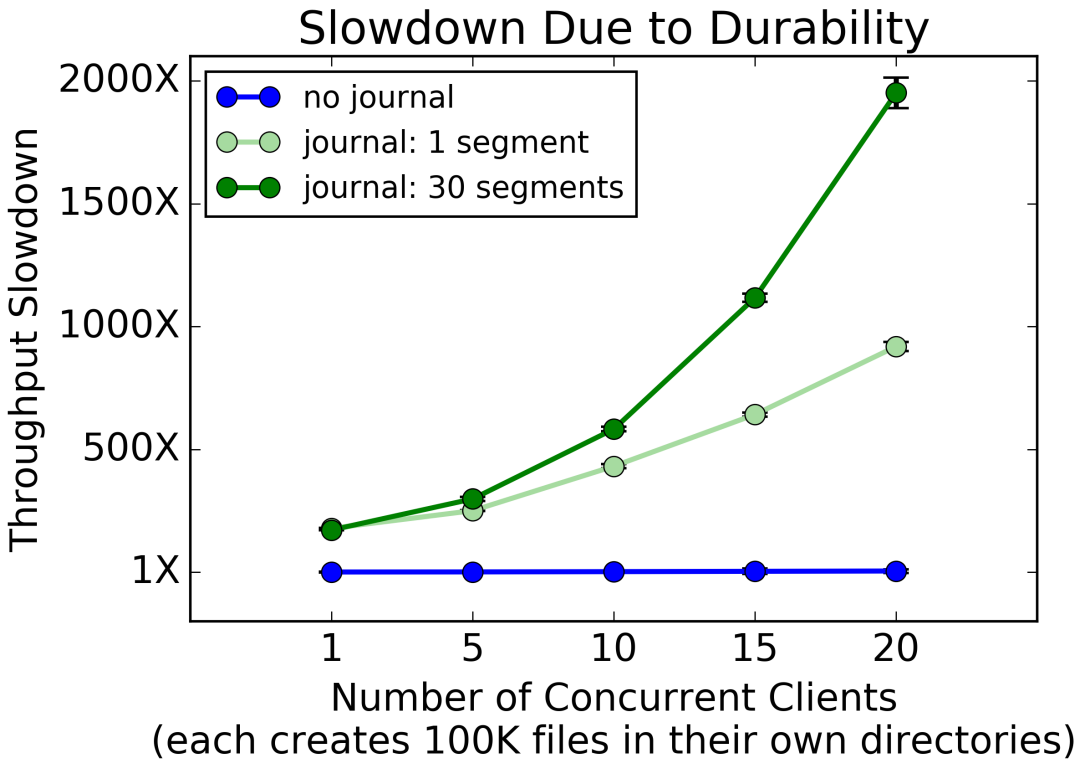


Figure 6.2: [source] **Durability slowdown. The bars show the effect of journaling metadata updates; "segment(s)" is the number of journal segments dispatched to disk at once. The durability slowdown of the existing CephFS implementation increases as the number of clients scales. Results are normalized to 1 client that creates 100K files in isolation.**

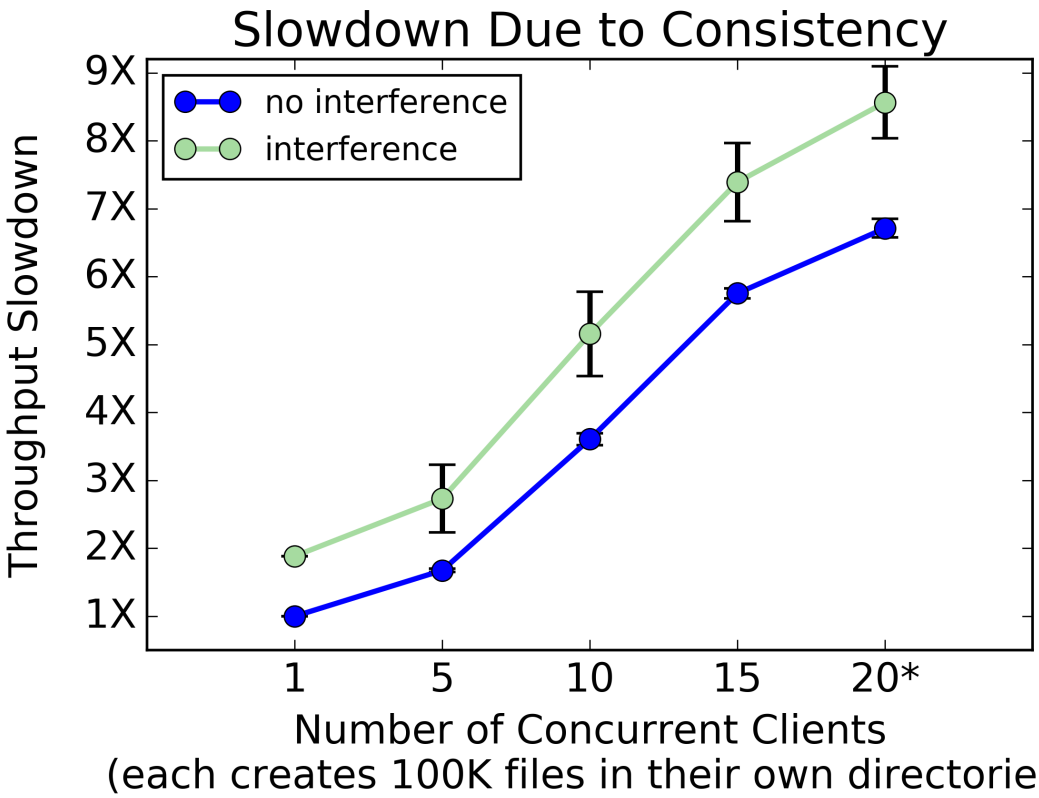


Figure 6.3: [source] **Consistency slowdown. Interference hurts variability; clients slow down when another client interferes by creating files in all directories. Results are normalized to 1 client that creates 100K files in isolation.**

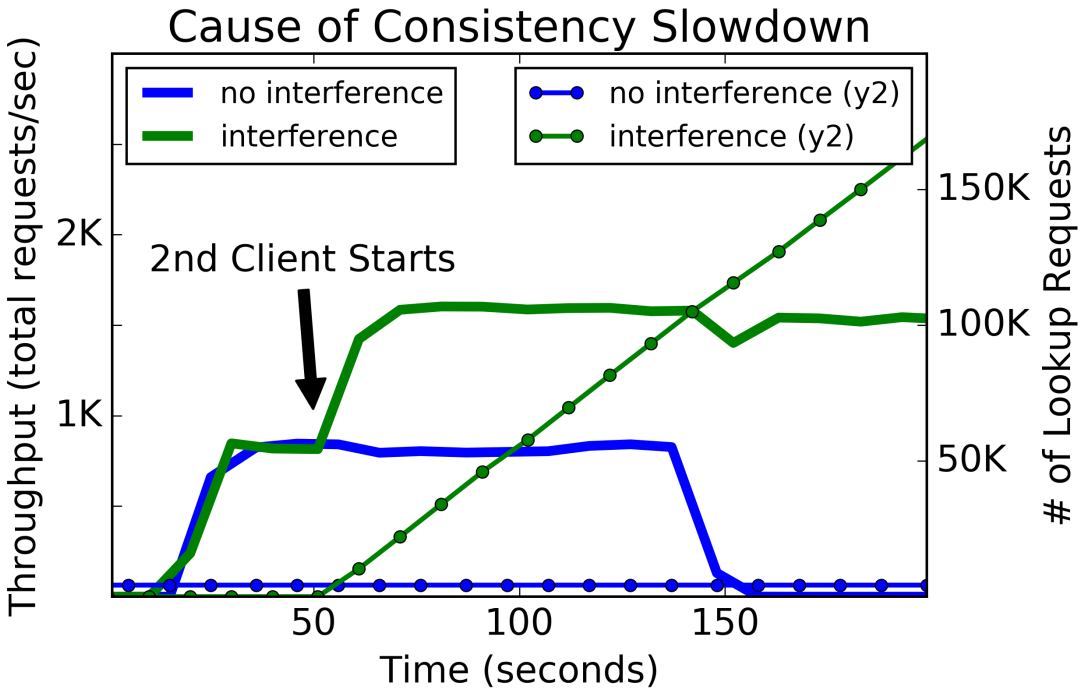


Figure 6.4: [source] **Cause of consistency slowdown. Interference increases RPCs; when another client interferes, capabilities are revoked and metadata servers do more work.**

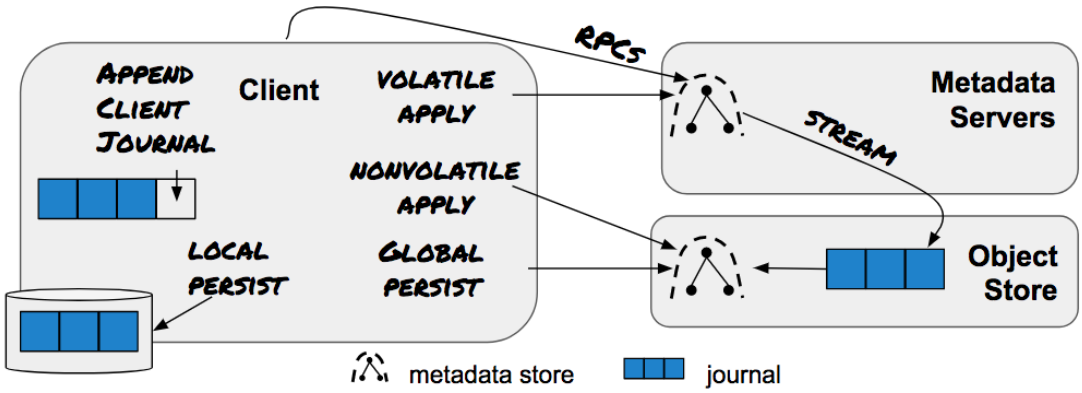


Figure 6.5: Illustration of the mechanisms used by applications to build consistency/durability semantics. Descriptions are provided by the underlined words in Section §6.2.1.

6.2 Methodology: Global Namespace, Subtree Consistency/Durability

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

6.2.1 Mechanisms: Building Guarantees

Figure 6.5 shows the mechanisms (labeled arrows) in Cudele and which daemon(s) they are performed by. Decoupled clients use the Append Client Journal mechanism to append metadata updates to a local, in-memory journal. Clients do not need to check for consistency when writing events and the metadata server blindly applies the updates because it assumes the events were already checked for consistency. The trade-off here is fast performance; it is a dangerous approach but could be implemented safely if the clients or metadata server are configured to check the validity of events before writing them. Once the job is complete, the system calls mechanisms to achieve the desired consistency/durability semantics. Cudele provides a library for clients to link into and all operations are performed by the client.

6.2.1.1 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 in-memory journal into the object store and restarts the metadata servers. When the metadata servers re-initialize, they notice new journal updates in the object store and replay the events onto their in-memory metadata stores. Volatile Apply takes the client's in-memory journal on the client and applies the updates directly to the in-memory metadata store maintained by the metadata servers. We say volatile

because – in exchange for peak performance – Cudele makes no consistency or durability guarantees while Volatile Apply is executing. If a concurrent update from a client occurs there is no rule for resolving conflicts and if the client or metadata server crashes there may be no way to recover.

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

6.2.1.2 Mechanisms Used for Durability

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

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

6.2.2 Defining Policies in Cudele

The spectrum of consistency and durability guarantees that administrators can construct is shown in Table 6.1. The columns are the different consistency semantics and the rows cover the spectrum of durability guarantees. For consistency: "invisible" means the system does not handle merging updates into a global namespace and it is assumed that middleware or the application manages consistency lazily; "weak" merges updates at some time in the future (*e.g.*, when the system has time, when the number of updates reaches a certain threshold, when the client is done writing, etc.); and updates

in "strong" consistency are seen immediately by all clients. For durability, "none" means that updates are volatile and will be lost on a failure. Stronger guarantees are made with "local", which means updates will be retained if the client node recovers and reads the updates from local storage, and "global", where all updates are always recoverable.

Existing, state-of-the-art systems in HPC can be represented by the cells in Table 6.1. POSIX IO-compliant systems like CephFS and IndexFS have global consistency and durability[2]; DeltaFS uses "invisible" consistency and "local" durability and BatchFS uses "weak" consistency and "local" durability. These systems have other features that could push them into different semantics but we assign labels here based on the points emphasized in the papers. To compose the mechanisms administrators inject which mechanisms to run and which to use in parallel using a domain specific language. Although we can achieve all permutations of the different guarantees in Table 6.1, not all of them make sense. For example, it makes little sense to do `append client journal+RPCs` since both mechanisms do the same thing or `stream+local persist` since "global" durability is stronger and has more overhead than "local" durability. The cost of each mechanism and the semantics described above are quantified in Sections §6.4.1.

In our prototype, the consistency and durability properties in Table 6.1 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

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

saved in the object store. If a failure occurs during Global Persist or if we inject a new policy that changes a subtree from Local Persist to Global Persist, Cudele makes no guarantee until the mechanisms are complete. Despite this, production systems that use Cudele should state up-front what the transition guarantees are for subtrees. This is not a limitation of our approach; it just lead to the simplest implementation.

6.2.3 Cudele Namespace API

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

The policies file supports the following parameters (default values are in parenthesis): which consistency model to use (`RPCs`), which durability model to use (`stream`), number of inodes to provision to the decoupled namespace (100), and which interfere policy to use, *i.e.* how to handle a request from another client targeted at this subtree (`allow`). The "Consistency" and "Durability" parameters are compositions of mechanisms; they can be serialized (+) or run in parallel (||). "Allocated Inodes" is a way for the application to specify how many files it intends to create. It is a contract so that the file system can provision enough resources for the incumbent merge and so it can give valid inodes to other clients. The inodes can be used anywhere within the decoupled namespace (*i.e.* at any depth in the subtree). "Interfere Policy" has two settings:

`block` and `allow`. For `block`, any requests to this part of the namespace returns with "Device is busy", which will spare the metadata server from wasting resources for updates that may get overwritten. If the application does not mind losing updates, for example it wants approximations for results that take too long to compute, it can select `allow`. In this case, metadata from the interfering client will be written and the computation from the decoupled namespace will take priority at merge time because the results are more accurate. Given these default values decoupling the namespace with an empty policies file would give the application 100 inodes but the subtree would behave like the existing CephFS implementation.

6.3 Implementation

We use a programmable storage approach [73] to design Cudele; namely, we try to leverage components inside CephFS to inherit the robustness and correctness of the internal subsystems. Using this "dirty-slate" approach, we only had to implement 4 of the 6 mechanisms from Figure 6.5 and just 1 required changes to the underlying storage system itself. In this section, we first describe a CephFS internal subsystem or component and then we show how we use it in Cudele.

6.3.1 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. **In Cudele,** the RPCs mechanism uses the in-memory metadata store to service requests. Using code designed for recovery, Volatile Apply and Nonvolatile Apply replay updates onto the metadata store in memory and in the object store, respectively.

6.3.2 Journal Format and Journal Tool

The journal is the second way that CephFS represents the file system namespace; it is a log of metadata updates that can materialize the namespace when the

updates are replayed onto the metadata store. The journal is a "pile system"; writes are fast but reads are slow because state must be reconstructed. Specifically, reads are slow because there is more state to read, it is unorganized, and many of the updates may be redundant. **In Cudele,** the journal format is used by Stream, Append Client Journal, Local Persist, and Global Persist. Stream is the default implementation for achieving global durability in CephFS but the rest of the mechanisms are implemented by writing with the journal format. By writing with the same format, the metadata servers can read and use the recovery code to materialize the updates from a client's decoupled namespace (*i.e.* merge). These implementations required no changes to CephFS because the metadata servers know how to read the events the library is writing. By re-using the journal subsystem to implement the namespace decoupling, Cudele leverages the write/read optimized data structures, the formats for persisting events (similar to TableFS's SSTables [59]), and the functions for replaying events onto the internal namespace data structures.

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

6.3.3 Inode Cache and Large Inodes

In CephFS, the inode cache reduces the number of RPCs between clients and metadata servers. Without contention clients can resolve metadata reads locally thus reducing the number of operations (*e.g.*, `lookup()`s). For example, if a client or metadata server is not caching the directory inode, all creates within that directory will result in a lookup and a create request. If the directory inode is cached then only the create needs to be sent. The size of the inode cache is configurable so as not to saturate the memory on the metadata server – inodes in CephFS are about 1400 bytes [2]. The inode cache has code for manipulating inode numbers, such as pre-allocating inodes to clients. **In Cudele,** Nonvolatile Apply uses the internal inode cache code to allocate inodes to clients that decouple parts of the namespace and to skip inodes used by the client at merge time.

In CephFS, inodes already store policies, like how the file is striped across the object store or for managing subtrees for load balancing. These policies adhere to logical partitionings of metadata or data, like Ceph pools and file system namespace subtrees. To implement this, the namespace data structure has the ability to recursively apply policies to subtrees and to isolate subtrees from each other. **In Cudele,** the large inodes also store consistency and durability policies. This approach uses the File Type interface from the Malacology programmable storage system [73] and it tells clients how to access the underlying metadata. The underlying implementation stores executable code in the inode that calls the different Cudele mechanisms. Of course, there are many

security and access control aspects of this approach but that is beyond the scope of this paper.

6.4 Evaluation

Cudele lets administrators construct consistency/durability guarantees using well-established research techniques from other systems; so instead of evaluating the scalability and performance of the techniques themselves against other file systems, we show that (1) the mechanisms we propose are useful for constructing semantics used by real systems and (2) the techniques can work side-by-side in the same namespace for common use cases.

We graph standard deviations for three runs (sometimes error bars are too small to see) and normalize results to make our results more generally applicable to different hardware. We use a CloudLab cluster of 34 nodes connected with 10Gbit ethernet, each with 16 2.4 GHz CPUs, 128GB RAM, and 400GB SSDs. Each node uses Ubuntu 14.04 and we develop on Ceph's Jewel release, version 10.2.1, which was released in May 2016. We use 1 monitor daemon, 3 object storage daemons, 1 metadata server daemon, and up to 20 clients. We scope the evaluation to one metadata server and scale the number of parallel clients each doing 100K operations because 100K is the maximum recommended size of a directory in CephFS. We scale to 20 clients because, as shown in Section §6.1, 20 clients is enough to saturate the resources of a single metadata server. This type of analysis shows the capacity and performance of a metadata server with superior metadata protocols, which should be used to inform metadata distribution across a cluster. Load balancing across a cluster of metadata servers with partitioning and replication can be explored with something like Mantle [75]. To make our results

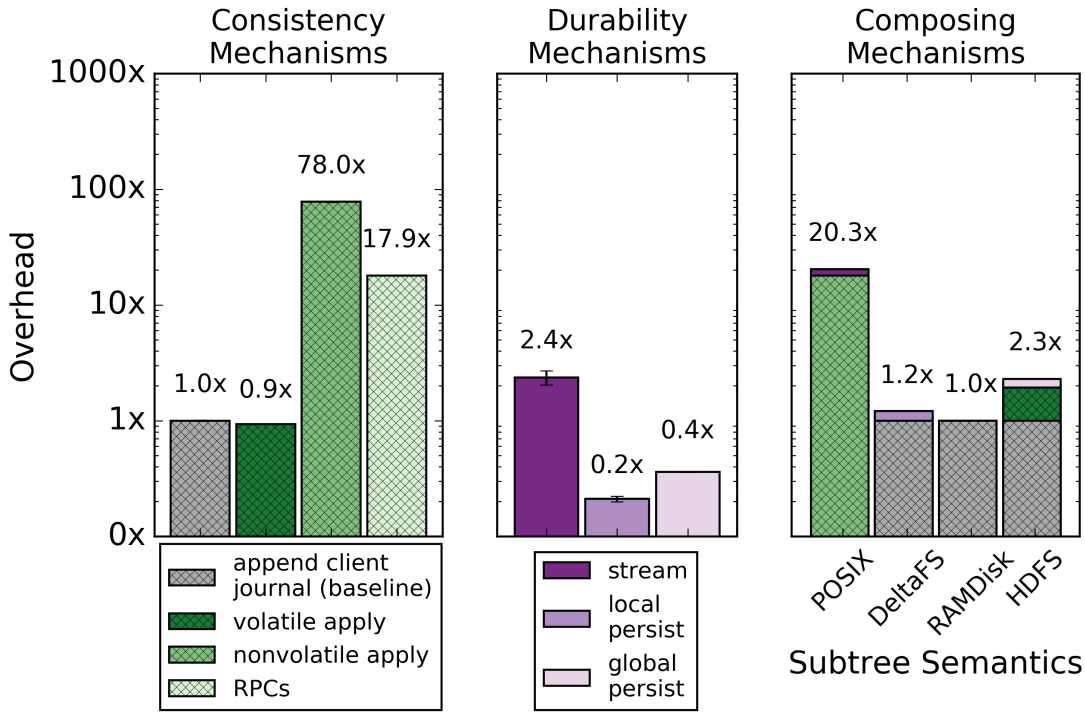


Figure 6.6: [source] Overhead of processing 100K create events for each mechanism in Figure 6.5, normalized to the runtime of writing events to client memory. The far right graph shows the overhead of building semantics of real world systems.

reproducible, this paper adheres to The Popper Convention [40] so experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure. The source code for Cudele is available on a branch [3] of our Ceph fork.

6.4.1 Microbenchmarks

We measure the overhead of each Cudele mechanism by having 1 client create 100K files in a directory for various subtree configurations. Figure 6.6 shows the time

that it takes each Cudele mechanism to process all metadata events. Results are normalized to the time it takes to write updates to the client's in-memory journal (*i.e.* the Append Client Journal mechanism), which is about 11K creates/sec. The first graph groups the consistency mechanisms, the second groups the durability mechanisms, and the third has compositions representing real-world systems.

Poorly Scaling Data Structures: Despite doing the same amount of work, mechanisms that rely on poorly scaling data structures have large slowdowns. For example, RPCs has a 17.9× slowdown because this technique relies on internal directory data structures, which is a well-known problem [60]. Other mechanisms that write events to a journal experience a much less drastic slowdown because the journal data structure does not need to be scanned for every operation. Events are written to the end of the journal without even checking the validity (*e.g.*, if the file already exists for a create), which is another form of relaxed consistency because the file system assumes the application has resolved conflicting updates in a different way.

Overhead of Consistency: RPCs is 19.9× slower than Volatile Apply because sending individual metadata updates over the network is costly. While RPCs sends a request for every file create, Volatile Apply writes directly to the in-memory data structures in the metadata server. While communicating the decoupled namespace directly to the metadata server with Volatile Apply is faster, communicating through the object store with Nonvolatile Apply is 78× slower. Nonvolatile Apply was not implemented as part of Cudele – it was a debugging and recovery tool packaged with CephFS. It works by iterating over the updates in the journal and pulling all objects

that *may* be affected by the update. This means that two objects are repeatedly pulled, updated, and pushed: the object that houses the experiment directory and the object that contains the root directory (*i.e.* /). Nonvolatile Apply (78×) and composing Volatile Apply + Global Persist (1.3×) end up with the same final metadata state but using Nonvolatile Apply is clearly inferior.

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

Composing Mechanisms: The graph on the right of Figure 6.6 shows how applications can compose mechanisms together to get the consistency/durability guarantees they need in a global namespace. We label the x-axis with systems that employ these semantics, as described in Figure 6.1. We make no guarantees during execution of the mechanisms or when transitioning semantics – the semantics are guaranteed *once the*

mechanism completes. So if servers fail during a mechanism, metadata or data may be lost. This graph shows how we can build application-specific subtrees by composing mechanisms and the performance of coupling well-established techniques to specific applications over the same file system.

6.4.2 Use Cases

Next we present three uses cases: creates in the same directory, interfering clients, and read while writing. The synthetic benchmarks model scenarios where these workloads co-exist in a global namespace and we provide insight into how the workload benefits from Cudele.

6.4.2.1 Creates in the Same Directory

We start with clients creating files in private directories because this workload is heavily studied in HPC [90, 60, 53, 98, 75], mostly due to checkpoint-restart [10]. For more use cases from other domains like the cloud, see Section §2.1.2.

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

In Figure 6.7 we scale the number of clients each doing 100K file creates in their own directories. Results are normalized to 1 client that creates 100K files using RPCs (about 549 creates/sec). As opposed to earlier graphs in Section §6.1 that plotted the

throughput of the slowest client, Figure 6.7 plots the throughput of the total job (*i.e.* from the perspective of the metadata server). Plotting this way is easier to understand because of how we normalize but the speedups over the RPC approach are the same, whether we look at the slowest client or not.

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

"Decoupled: merge + create" outperforms "RPCs" by 3.37× because "decoupled: merge + create" uses a relaxed form of consistency and leverages bulk updates just like DeltaFS [99]. Decoupled namespaces (1) place no restrictions on the validity of metadata inserted into the journal (*e.g.*, never checking for the existence of files before creating files), (2) avoid touching poorly scaling data structures, and (3) allow clients to batch events into bulk updates. Had we implemented the client to send updates one at a time and to include `lookup()` commands before `open()` requests, we would have seen performance closer to the "RPC" subtree. The "decoupled: merge + create" curve is also pessimistic because it models a scenario in which all client journals arrive at the same time. So for the 20 clients data point, we are measuring the operations per second for 20 client journals that land on the metadata server at the same time. Had we

added infrastructure to overlay journal arrivals or time client sync intervals, we could have scaled more closely to "decoupled: create".

6.4.2.2 Interfering Clients

Next we show how Cudele can be programmed to block interfering clients, which lets applications control isolation to get better and more reliable performance. Clients create 100K files in their own directories while another client interferes by creating 1000 files in each directory. The workload introduces false sharing and the metadata server revokes capabilities on directories touched by the interfering client. More examples can be found in Section §2.1.1.

Cudele setup: enable global durability with Stream and strong consistency with RPCs to mirror the setup from the problem presented in Figure 6.3. We configure one subtree with an interfere policy of "allow" and another subtree with "block" so `-EBUSY` is returned to interfering clients. The former is the default behavior in file systems and the latter isolates performance from interfering clients.

Figure 6.8 plots the overhead of the slowest client, normalized to 1 client that creates 100K files in isolation (about 513 creates/sec). "Interference" and "no interference" is the performance with and without an interfering client touching files in every directory, respectively. The goal is to explicitly isolate clients so that performance is similar to the "no interference" curve, which has lower slowdowns (on average, 1.42× per client compared to 1.67× per client for "interference") and less variability (on average, a standard deviation of 0.06 compared to 0.44 for "interference"). "Block interference"

uses the Cudele API to block interfering clients and the slowdown (1.34× per client) and variability (0.09) look very similar to "no interference" for a larger number of clients. For smaller clusters the overhead to reject requests is more evident when the metadata server is underloaded so the slowdowns are similar to "interference". We conclude that administrators can block interfering clients to get the same performance as isolated scenarios but there is a non-negligible overhead for rejecting requests when the metadata server is not operating at peak efficiency.

6.4.2.3 Read while Writing

The final use case shows how the API gives administrators fine-grained control of the consistency semantics to support current practices and scientific workflows in HPC. More details for this use case can be found in Section §2.1.3.

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

Figure 6.9 shows the performance degradation of a single client writing 1 million updates to the decoupled namespace and pausing to send updates to the metadata server. We scale the namespace sync interval to show the trade-off of frequently pausing or writing large logs of updates. We use an idle core to log the updates and to do

the network transfer. The client only pauses to fork off a background process, which is expensive as the address space needs to be copied. The alternative is to pause the client completely and write the update to disk but since this implementation is limited by the speed of the disk, we choose the memory-to-memory copy of the fork approach.

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

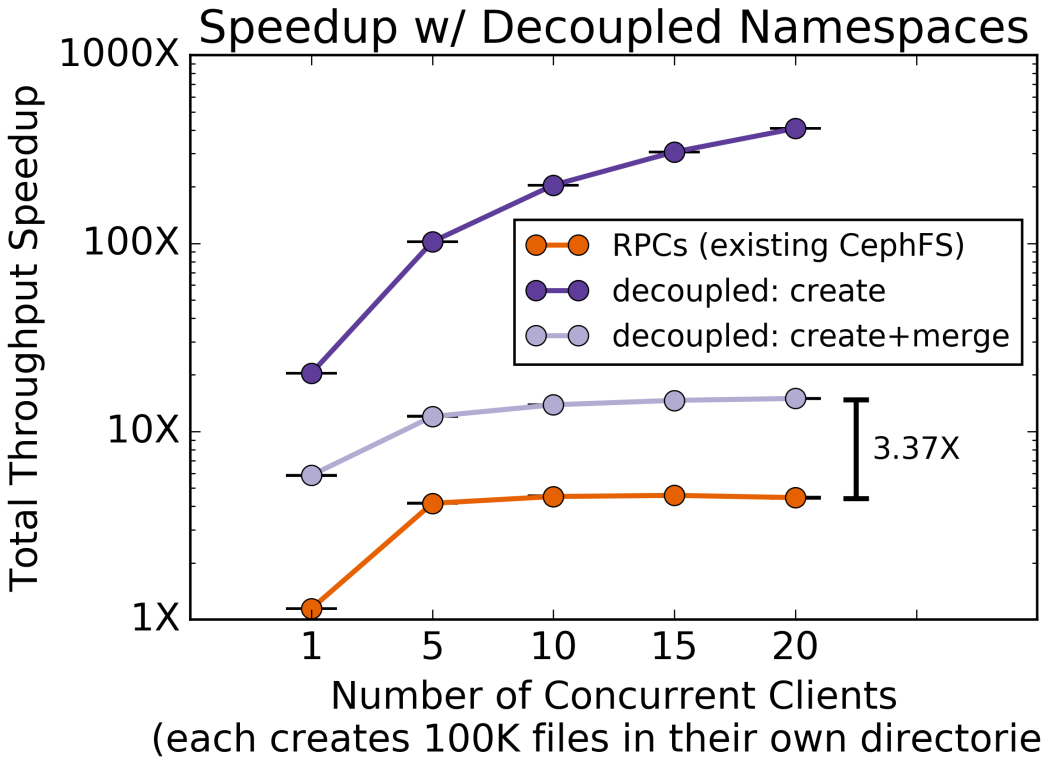


Figure 6.7: [source] **The speedup of decoupled namespaces over RPCs for parallel creates on clients ;** **`create`** **is the throughput of clients creating files in-parallel and writing updates locally;** **`create+merge`** **includes the time to merge updates at the metadata server. Decoupled namespaces scale better than RPCs because there are less messages and consistency/durability code paths are bypassed.**

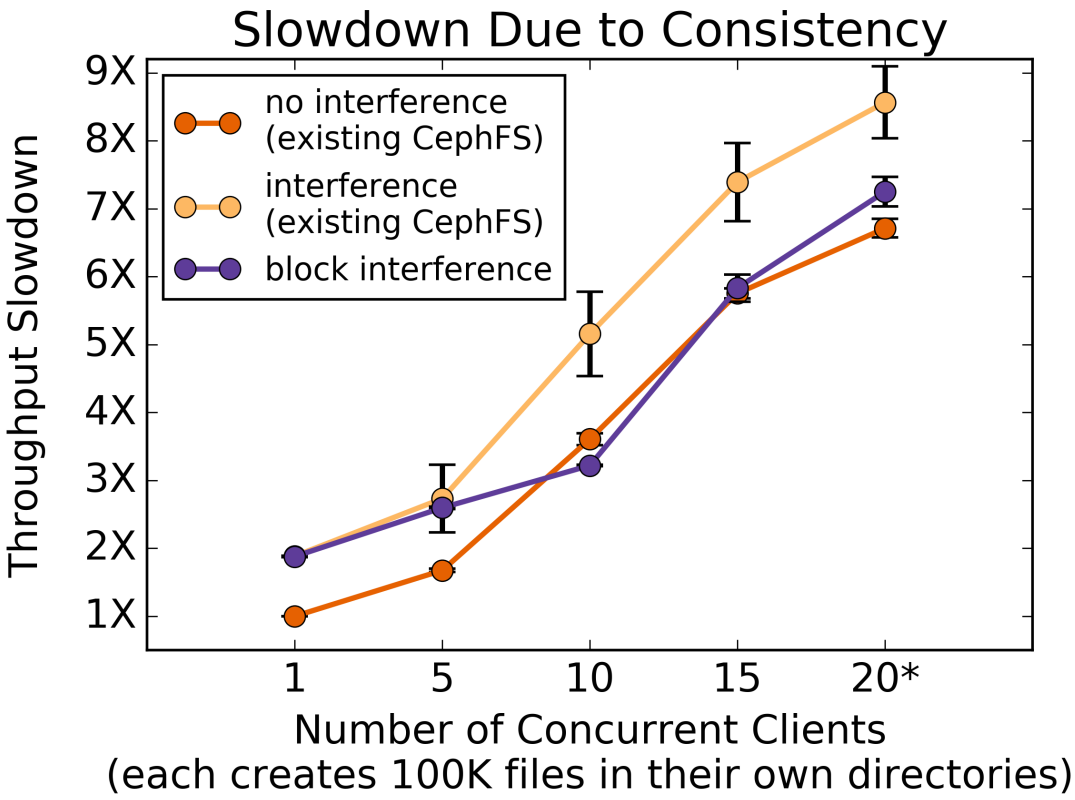


Figure 6.8: [source] **The block/allow interference API isolates directories from interfering clients.**

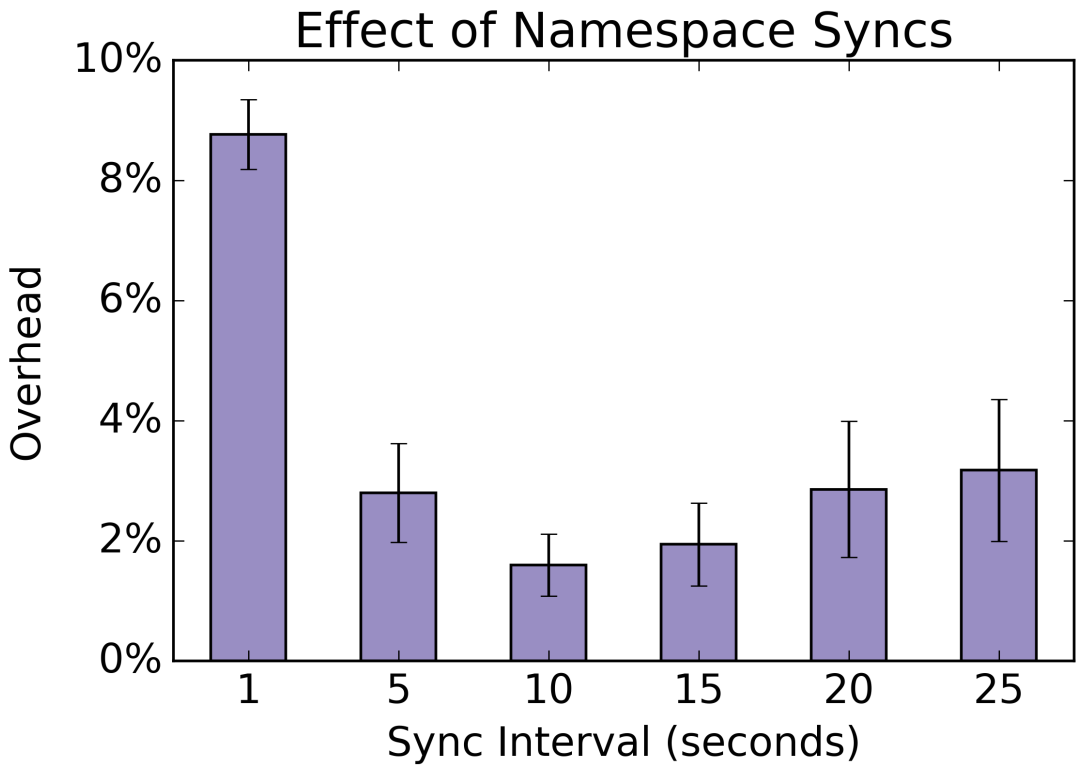


Figure 6.9: [source] **Syncing to the global namespace. The bars show the slowdown of a single client syncing updates to the global namespace. The inflection point is the trade-off of frequent updates vs. larger journal files.**

6.5 Conclusion and Future Work

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

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

Chapter 7

Tintenfisch: File System Namespace Schemas and Generators

The file system metadata service is the scalability bottleneck for many of today's workloads [62, 5, 6, 7, 89]. Common approaches for attacking this "metadata scaling wall" include: caching inodes on clients and servers [19, 79, 33, 20, 91], caching parent inodes for path traversal [54, 60, 11, 90, 60], and dynamic caching policies that exploit workload locality [95, 100, 47]. These caches reduce the number of remote procedure calls (RPCs) but the effectiveness is dependent on the overhead of maintaining cache coherence and the administrator's ability to select the best cache size for the given workloads. Recent work reduces the number of metadata RPCs to 1 without using a cache at all, by letting clients "decouple" the subtrees from the global namespace so that they can do metadata operations locally [99, 72]. *Even with* this technique, we show that file system metadata is still a bottleneck because namespaces for today's

workloads can be very large. The size is problematic for reads because metadata needs to be transferred and materialized.

The management techniques for file system metadata assume that namespaces have no structure but we observe that this is not the case for all workloads. We propose Tintenfisch, a file system that allows users to succinctly express the structure of the metadata they intend to create. If a user can express the structure of the namespace, Tintenfisch clients and servers can (1) compact metadata, (2) modify large namespaces more quickly, and (3) generate only relevant parts of the namespace. This reduces network traffic, storage footprints, and the number of overall metadata operations needed to complete a job.

Figure 7.1 provides an architectural overview: clients first decouple the file system subtree they want to operate on[1] then clients and metadata servers lazily generate subtrees as needed using a "namespace generator". The namespace generator is stored in the root inode of the decoupled subtree and can be used later to efficiently merge new metadata (that was not explicitly stated up front) into the global namespace. The fundamental insight is that the client and server both understand the final structure of the file system metadata. Our contributions:

- observing namespace structure in high performance computing, high energy physics, and large fusion simulations (§7.1)
- based on these observations, we defined namespace schemas for categorizing namespaces and their amenability to compaction and generation (§7.2.1)

[1] This is not a contribution as it was presented in [72].

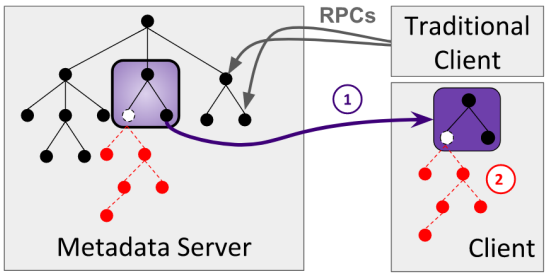
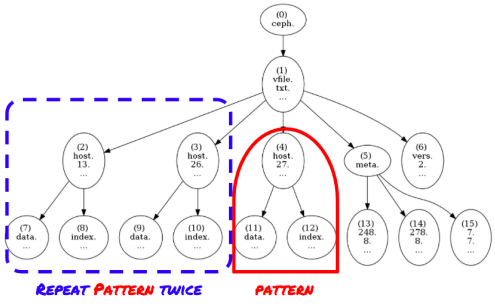
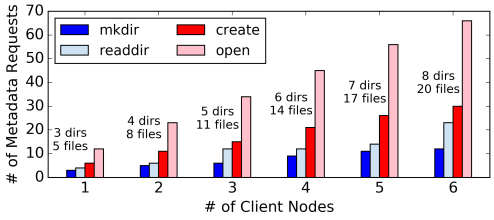


Figure 7.1: In (1), clients decouple file system subtrees and interact with their copies locally. In (2), clients and metadata servers generate subtrees, reducing network/storage usage and the number of metadata operations.

- a generalization of existing file system services to implement namespace generators that efficiently compact and generate metadata (§7.2.2)



(a) PLFS file system tree



(b) [source] PLFS metadata operations.

Figure 7.2: PLFS file system metadata. (a) shows that the namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times. (b) shows that the namespace scales linearly with the number of clients. This makes reading and writing difficult using RPCs so decoupled subtrees must be used to reduce the number of RPCs.

7.1 Motivating Examples

We look at the namespaces for 3 large-scale applications. Each is from a different domain and this list is not meant to be exhaustive. To highlight the scalability challenges for file system metadata management, we focus on large scale systems in high performance computing, high energy physics, and large scale simulations. Large lists represent common problems in each of these domains. To make our results reproducible, this paper adheres to The Popper Convention [41] so experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure.

7.1.1 High Performance Computing: PLFS

Checkpointing performs small writes to a single shared file but because file systems are optimized for large writes, performance is poor. PLFS [10] solved the checkpoint-restart problem by mapping logical files to physical files on the underlying file system. The solution targets N-1 strided checkpoints, where many processes write small IOs to offsets in the same logical file. Each process sequentially writes to its own, unshared data file in the hierarchical file system and records an offset and length in an index file. Reads aggregate index files into a global index file, which it uses as a lookup table for identifying offsets into the logical file.

Namespace Description: when PLFS maps a single logical file to many physical files, it deterministically creates the namespace in the backend file system. For metadata writes, the number of directories is dependent on the number of clients nodes and the number of files is a function of the number of client processes. A directory called a container is created per node and processes write data and index files to the container assigned to their host. So for a write workload (*i.e.* a checkpoint) the underlying file system creates a deep and wide directory hierarchy, as shown in Figure 7.2a. The `host*` directory and `data*/index` files (denoted by the solid red line) are created for every node in the system. The pattern is repeated twice (denoted by the dashed blue line) in the Figure, representing 2 additional hosts each with 1 process.

Namespace Size: Figure 7.2b scales the number of clients and plots the total number of files/directories (text annotations) and the number of metadata operations

needed to write and read a PLFS file. The number of files is 2 × (# of processes). So for 1 million processes each checkpointing a portion of a 3D simulation, the size of the namespace will be 2 million files. RPC-based approaches like IndexFS [60] have been shown to struggle with metadata loads of this size but decoupled subtree approaches like DeltaFS [99] report up to 19.69 million creates per second, so writing checkpoints is largely a solved problem.

For reading a checkpoint, clients must coalesce index files to reconstruct the PLFS file. Figure 7.2b shows that the read metadata requests ("readdir" and "open") outnumber the create requests by a factor of 4×. Metadata read requests are notoriously slow [13, 22], so like create requests, RPCs are probably untenable. If the checkpoint had been written with the decoupled namespace approach, file system metadata would be scattered across clients so metadata would need to be coalesced before restarting the checkpoint. If the metadata had already been coalesced at some point they would still need to be transferred to the client. Regardless, both decoupled subtree scenarios require moving and materializing the file system subtree. Current efforts improve read scalability by reducing the space overhead of the index files themselves [31] and transferring index files after each write [28] but these approaches target the transfer and materialization of the index file data, not the index file metadata.

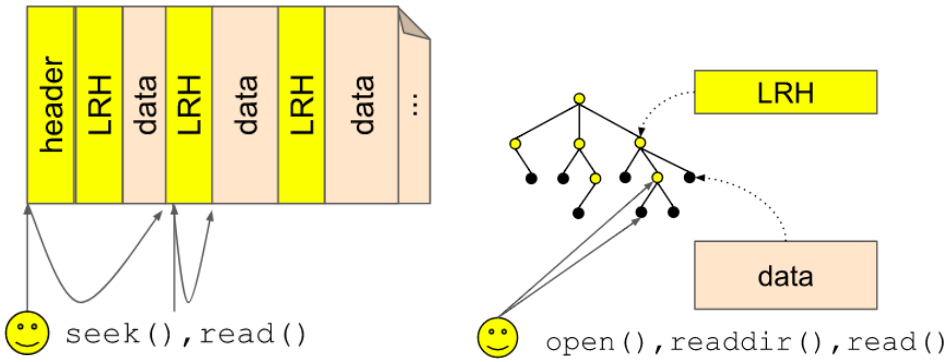
Takeaway: the PLFS namespace scales with the number of client processes so RPCs are not an option for reading or writing. Decoupling the namespace helps writes but then the read performance is limited by the speed of transferring file system metadata across the network to the reading client *in addition* to reading the contents

of the index files themselves.

7.1.2 High Energy Physics: ROOT

The High Energy Physics (HEP) community uses a framework called ROOT to manipulate, manage, and visualize data about proton-proton collisions collected at the large hadron collider (LHC). The data is used to re-simulate phenomena of interest for analysis and there are different types of reconstructions each with various granularities. The data is organized as nested, object oriented event data of arbitrary type (*e.g.*, particle objects, records of low-level detector hit properties, etc.). Physicists analyze the data set by downloading interesting events, which are stored as a list of objects in ROOT files. ROOT file data is accessed by consulting metadata in the header and seeking to a location in the bytestream, as shown in Figure 7.3a. The ROOT file has both data and ROOT-specific metadata called Logical Record Headers (LRH). For this discussion, the following objects are of interest: a "Tree" is a table of a collection of events, listed sequentially and stored in a flat namespace; a "Branch" is a data container representing columns of a Tree; and "Baskets" are byte ranges partitioned by events and indexed by LRHs. Clients request Branches and data is transferred as Baskets; so Branches are the logical view of the data for users and Baskets are the compression, parallelization, and transfer unit. The advantages of the ROOT framework is the ability to (1) read only parts of the data and (2) easily ingest remote data over the network.

Namespace Description: the HEP community is running into scalability problems. The current effort is to integrate the ROOT framework with Ceph. But



(a) file approach **(b) namespace approach**

Figure 7.3: ROOT file system metadata. (a) file approach: stores data in a single ROOT file, where clients read the header and seek to data or metadata (LRH); a ROOT file stored in a distributed file system will have IO read amplification because the striping strategies are not aligned to Baskets. (b) namespace approach: stores Baskets as files so clients read only data they need.

naive approaches such as storing ROOT files as objects in an object store or files in a file system have IO read amplification (*i.e.* read more than is necessary); storing as an object would pull the entire GB-sized blob and storing as a file would pull more data than necessary because the file stripe size is not aligned to Baskets. To reduce IO read amplification the namespace approach [57] views a ROOT file as a namespace of data. Physicists ask for Branches, where each Branch can be made up of multiple sub-Branches (*i.e.* `Events/Branch0/Branch1`), similar to pathname components in a POSIX IO file name. The namespace approach partitions the ROOT file onto a file system namespace, as shown in Figure 7.3b. File system directories hold Branch

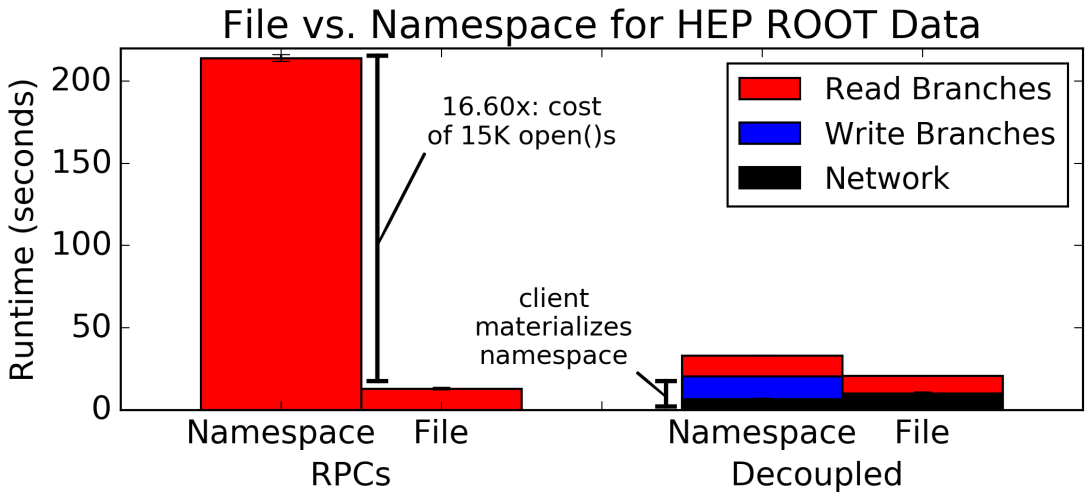


Figure 7.4: [source] **ROOT metadata size and operations**

Figure 7.5: "Namespace" is the runtime of reading a file per Basket and "File" is the runtime of reading a single ROOT file. RPCs are slower because of the metadata load and the overhead of pulling many objects. Decoupling the namespace uses less network (because only metadata and relevant Baskets get transferred) but incurs a metadata materialization overhead.

metadata, files contain Baskets, and clients only pull Baskets they care about.

Namespace Size: storing this metadata in a file system would overwhelm most file systems in two ways: (1) too many inodes and (2) per-file overhead. To quantify (1), consider the Analysis Object Dataset which has a petabyte of data sets made up of a million ROOT files each containing thousands of Branches, corresponding to a billion files in the namespace approach. To quantify (2), the read and write runtime over six runs of replaying a trace of Branch access from the NTupleMaker application is shown in Figure 7.4, where the x-axis is approaches for storing ROOT data. Using

the namespace approach with RPCs is far slower because of the metadata load and because many small objects are pulled over the network. Although the file approach reads more data than is necessary since the stripe size of the file is not aligned to Baskets, the runtime is still 16.6× faster. Decoupling the namespace is much faster for the namespace approach but the cost of materializing file system metadata makes it slower than the file approach. Note that this is one (perhaps pessimistic) example workload; the ROOT file is 1.7GB and 65% of the file is accessed so the namespace approach might be more scalable for workloads that access fewer Baskets.

Takeaway: the ROOT namespace stores billions of files and we show that RPCs overwhelm a centralized metadata server. Decoupling the namespace helps writes but then the read performance is limited by (1) the speed of transferring file system metadata across the network and (2) the cost of materializing parts of the namespace that are not relevant to the workload.

7.1.3 Large Scale Simulations: SIRIUS

SIRIUS [42] is the Exascale storage system being designed for the Storage System and I/O (SSIO) initiative [64]. The core tenant of the project is application hints that allow the storage to reconfigure itself for higher performance using techniques like tiering, management policies, data layout, quality of service, and load balancing. SIRIUS uses a metadata service called EMPRESS [46], which is an SQLite instance that stores user-defined metadata for bounding boxes (*i.e.* a 3-dimensional coordinate space). EMPRESS is designed to be used at any granularity, which is important for

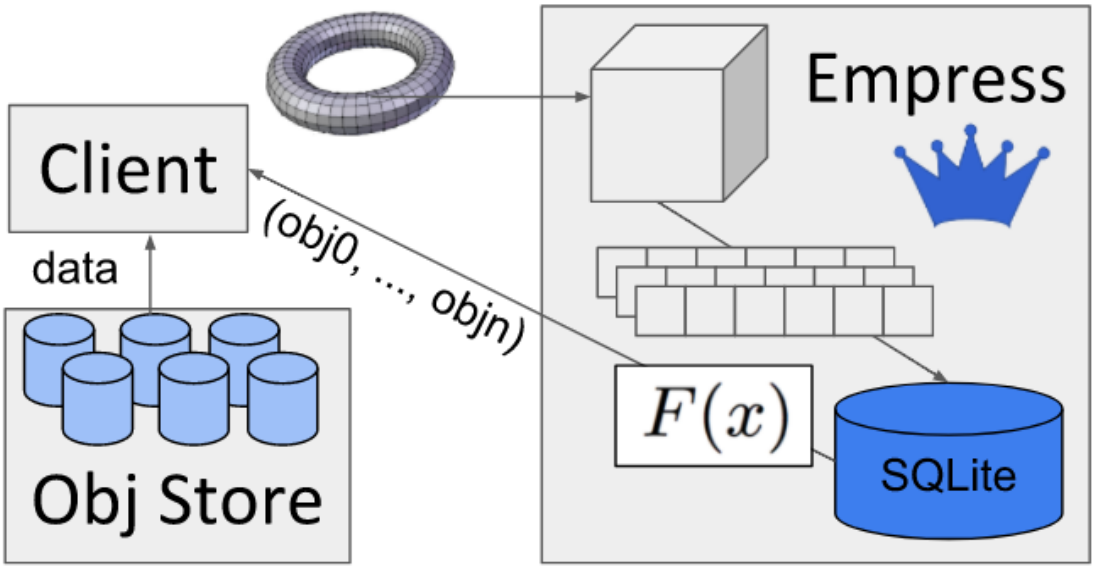


Figure 7.6: One potential EMPRESS design for storing bounding box metadata. Coordinates and user-defined metadata are stored in SQLite while object names are calculated using a partitioning function ($F(x)$) and returned as a list of object names to the client.

a simulation space represented as a 3D mesh. By granularity, we mean that metadata access can be optimized per variable (*e.g.*, temperature, pressure, etc.), per timestamp, per run, or even per set of runs (which may require multiple queries). At this time, EMPRESS is single node but it is designed to scale-out via additional independent instances.

Namespace Description: the global space is partitioned into non-overlapping, regular shaped cells. The EMPRESS database has columns for the application ID, run ID, timestamp, variable name, feature name, and bounding box coordinates for these cells. Users can also add custom-defined metadata. The namespace we are referring

to here is the list of objects containing simulation data associated to a bounding box (or row in the database). Variables affect how the space is partitioned into objects; temperature may be computed for every cell while pressure is computed for every n cells. For most simulations, there are a minimum of 10 variables.

Namespace Size: a back-of-the-envelope calculation for the number of object names for a single run is:

$$\frac{\text{(processes)} \times \text{(data/process)} \times \text{(variables)} \times \text{(timesteps)}}{\text{(object size)}}$$

We calculate $1 * 10^{12}$ (1 trillion) objects for a simulation space of $1\text{K} \times 1\text{K} \times 1\text{K}$ cells containing 8 byte floats. We use 1 million processes, each writing 8GB of data for 10 variables over 100 timesteps and an object size of 8MB (the optimal object size of Ceph's object store). The data per process and number of variables are scaled to be about 1/10 of each process's local storage space, so about 80GB. 100 timesteps is close to 1 timestep every 15 minutes for 24 hours.

As we integrate EMPRESS with a scalable object store, mapping bounding box queries to object names for data sets of this size is a problem. Clients query EMPRESS with bounding box coordinates[2] and EMPRESS must provide the client with a list of object names. One potential design is shown in Figure 7.6; coordinates for variables are stored in the database and object name lists are calculated using the $F(x)$ partitioning function at read time. The problem is that object name lists can be very large when applications query multiple runs each containing trillions of objects, resulting in long transfer times as the metadata is sent back to the client. Even after receiving the object

[2]Users usually track bounding boxes of interest by tagging features at write time.

name list, the client may need to manage and traverse the list, doing things like filtering for object names at the "edge" of the feature of interest.

Takeaway: SIRIUS stores trillions of objects for a single large scale simulation run and applications often access multiple runs. These types of queries return a large list of object names so the bottleneck is managing, transferring, and traversing these lists. The size of RPCs is the problem, not the number. POSIX IO hierarchical namespaces may be a good model for applications to access simulation data but another technique for handling the sheer size of these object name lists is needed.

For n processes on m servers:

```
# of dirs = m × mkdir()

# of file = 2 × n

# of file per dir = n/m
```

Figure 7.7: Function generator for PLFS

```
local box require 'box2d'
for i=_x,_x+x do for j=_y,_y+y do
  if t>30 then
    obj_list.insert(box(x,y,z))
  else
    b0,b1,b2,=box.nsplit(4)
    obj_list.insert(b0,b1,b2)
end end end
return obj_list
```

Figure 7.8: Code generator for SIRIUS

7.2 Methodology: Compact Metadata

Namespace schemas and generators help clients and servers establish an understanding of the final file system metadata shape and size that eliminates the metadata overheads highlighted above.

```
void recurseBranch(TObjArray *o){
  TIter i(o);
  for(TBranch *b=i.Next();
      i.Next()!=0;
      b=i.Next()){
    processBranch(b);
    recurseBranch(b->GetListOfBranches());
  }
}
```

Figure 7.9: Code generator for HEP

7.2.1 Namespace Schemas

Namespace schemas describe the structure of the namespace. A "balanced" namespace means that subtree patterns (files per directory) are repeated and a "bounded" namespace means that the range of file/directory names can be defined *a-priori* (before the job has run but after reading metadata). Traditional shared file systems are designed for general file system workloads, like user home directories, which have an unbalanced and unbounded namespace schema because users can create any number of files in any pattern. PLFS has a balanced and bounded namespace because the distribution of files per directory is fixed (and repeated) and any subtree can be generated using the client hostnames and the number of processes. ROOT and SIRIUS are examples of unbalanced and bounded namespace schemas. The file per directory shape is not repeated (it is determined by application-specific metadata, LRH for ROOT or variables for SIRIUS) but the range of file/directory names can be determined before

the job starts.

7.2.2 Namespace Generators

A namespace generator is a compact representation of a namespace that lets clients/servers generate file system metadata. They can be used for bounded or balanced namespace schemas. Tintenfisch is built on Cudele [72] so a centralized, globally consistent metadata service can decouple subtrees and clients can do metadata IO locally with the consistency/durability semantics they require. This concept is similar to LWFS [52], which supplied a core set of functionality and applications add additional functionality. In Tintenfisch, namespace generators are stored in the directory inode of the decoupled subtree using the "file type" interface from Malacology [74]. Next we discuss 3 example namespace generators.

Formula Generator: takes domain-specific information as input and produces a list of files and directories. For example, PLFS creates files and directories based on the number of clients, so administrators can use the formula in Figure 7.7, which takes as input the number of processes and hosts in the cluster and outputs the number of directories, files, and files per directory. The namespace drawn in Figure 7.2a can be generated using an input of 3 hosts each with 1 process.

Code Generator: gives users the flexiblity to write programs that generate the namespace. This is useful if the logic is too complex to store as a formula or requires external libraries to interpret metadata. For example, SIRIUS constructs the namespace using domain-specific partitioning logic written in Lua. Figure 7.8 shows

how the namespace can be constructed by iterating through bounding box coordinates and checking if a threshold temperature is eclipsed. If it is, extra names are generated using the `box2d` package. Although the partitioning function itself is not realistic, it shows how code generators can accommodate namespaces that are complex and/or require external libraries.

Pointer Generator: references metadata in scalable storage and avoids storing large amounts of metadata in inodes, which is a frowned upon in distributed file system communities [2]. This is useful if there is no formal specification for the namespace. For example, ROOT uses self-describing files so headers and metadata need to be read for each ROOT file. A code generator is insufficient for generating the namespace because all necessary metadata is in objects scattered in the object store. A code generator containing library code for the ROOT framework *and* a pointer generator for referencing the input to the code can be used to describe a ROOT file system namespace. Figure 7.9 shows a code generator example where clients requesting Branches follow the pointer generator (not pictured) to objects containing metadata. An added benefit is that Tintenfisch can lazily construct parts of the namespace as needed, avoiding the inode problem discussed in §7.1.2.

7.3 Conclusion

Namespace schemas and generators solve the read problems from the examples in §7.1 because clients and servers avoid exchanging file system metadata in its entirety. Our examples benefit from *metadata compaction* because it speeds up network transfers and reduces the storage footprint of metadata. Our examples also benefit from the ability to *modify large namespaces*: if a PLFS namespace was constructed with 1 million processes, scaling to 2 million processes only requires sending a new input to the formula generator; ROOT Branches can be added to the namespace by changing the metadata referenced by the pointer generator; and if SIRIUS objects need to be repartitioned, only the logic in the code generator needs to be updated. SIRIUS and ROOT benefit from the ability to *generate relevant parts of the namespace* because only a fraction of the metadata is needed.

Contrary to common belief, global file system namespaces can be scalable if they are given enough domain-specific knowledge. File systems are thought to be robust and general because they have been around for a long time. But we show that today's applications are specialized, so they have regular, large namespaces. As a result, the file system should be changing its internal mechanisms to leverage the bounded and balanced nature of these namespaces to optimize metadata performance.

Bibliography

[1] Ceph Documentation. http://docs.ceph.com/docs/master/cephfs/capabilities/, December 2017.

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

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

[4] Hadoop 3.0 Documentation. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html, February 2018.

[5] C. L. Abad, H. Luu, Y. Lu, and R. Campbell. Metadata Workloads for Testing Big Storage Systems. Technical report, Citeseer, 2012.

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

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

[8] B. Behzad, S. Byna, S. M. Wild, and M. Snir. Improving Parallel i/o Autotuning with Performance Modeling. 2014.

[9] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir, et al. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13.

[10] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09.

[11] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '03, 2003.

[12] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06.

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

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

[15] K. Chasapis, M. F. Dolz, M. Kuhn, and T. Ludwig. Evaluating Lustre's Metadata Server on a Multi-socket Platform. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, 2014.

[16] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operarting Systems Design & Implementation*, OSDI '04, 2004.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[19] B. Depardon, G. Le Mahec, and C. Séguin. Analysis of six distributed file systems. Technical report, French Institute for Research in Computer Science and Automation, 2013.

[20] A. Devulapalli and P. Wyckoff. File creation strategies in a distributed metadata file system. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[21] J. R. Douceur and J. Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design & Implementation*, OSDI '06, 2006.

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

[23] S. Faibish, J. Bent, U. Gupta, D. Ting, and P. Tzelnic. Slides: 2 Tier Storage Architecture.

[24] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceed-*

ings of the 19th ACM Symposium on Operating Systems Principles, SOSP '03, 2003.

[26] M. Grawinkel, T. Sub, G. Best, I. Popov, and A. Brinkmann. Towards Dynamic Scripted pNFS Layouts. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, 2012.

[27] H. Greenberg, J. Bent, and G. Grider. MDHIM: A Parallel Key/Value Framework for HPC. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, HotStorage '15.

[28] G. Grider. If the plfs index is still too large too efficiently read/distributepull out the hammerplfs collectives, reduces plfs index to nearly nothingdnabling strategy for optimizing reads and active analysis. FIXME!

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

[30] K. Hakimzadeh, H. P. Sajjad, and J. Dowling. Scaling HDFS with a Strongly Consistent Relational Model for Metadata. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS '14.

[31] J. He. Io acceleration with pattern detection. FIXME!

[32] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-Tuning System for Big Data Analytics. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '11.

[33] D. Hildebrand and P. Honeyman. Exporting Storage Systems in a Scalable Manner with pNFS. In *Proceedings of the 22Nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, 2005.

[34] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the Conference on Networked Systems Design and Implementation*, NSDI '11.

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

[36] H.-P. D. C. HP. HP Storeall Storage Best Practices. In *HP Product Documentation*, whitepaper '13. http://h20195.www2.hp.com/, 2013.

[37] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10.

[38] G. T. Inc. Scale up vs. scale out. In *Gigaspaces Resource Center*. http://www.gigaspaces.com/WhitePapers, 2011.

[39] J. Jenkins, G. M. Shipman, J. Mohd-Yusof, K. Barros, P. H. Carns, and R. B.

Ross. A Case Study in Computational Caching Microservices for HPC. In *IPDPS Workshops*, 2017.

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

[41] I. Jimenez, M. A. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '17.

[42] S. A. Klasky, H. Abbasi, M. Ainsworth, J. Choi, M. Curry, T. Kurc, Q. Liu, J. Lofstead, C. Maltzahn, M. Parashar, N. Podhorszki, E. Suchyta, F. Wang, M. Wolf, C. S. Chang, M. Churchill, and S. Ethier. Exascale Storage Systems the SIRIUS Way. *Journal of Physics: Conference Series*.

[43] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proceedings of the 11th European Conference on Computer Systems*, Eurosys '16.

[44] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17.

[45] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[46] M. Lawson, C. Ulmer, S. Mukherjee, G. Templet, J. F. Lofstead, S. Levy, P. M. Widener, and T. Kordenbrock. Empress: Extensible Metadata Provider for Extreme-Scale Scientific Simulations. In *Proceedings of the Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW '17.

[47] W. Li, W. Xue, J. Shu, and W. Zheng. Dynamic Hashing: Adaptive Metadata Management for Petabyte-scale Scale Systems. In *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '06, 2006.

[48] K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *login; '10.*

[49] M. K. McKusick. Keynote Address: A Brief History of the BSD Fast Filesystem, February 2015.

[50] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up X Scale-out: A Case Study Using Nutch/Lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[51] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage

System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*, 2014.

[52] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for Scientific Applications. In *International Conference on Cluster Computing*, Cluster Computing '06.

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

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

[55] J. PenisBent, B. Settlemyer, and G. Grider. Serving Data to the Lunatic Fringe. *login; '16*.

[56] D. Perez, E. D. Cubuk, A. Waterland, E. Kaxiras, and A. F. Voter. Long-Time Dynamics Through Parallel Trajectory Splicing. *Journal of Chemical Theory and Computation*.

[57] J. Pivarski. How to make a petabyte ROOT file: proposal for managing data with columnar granularity.

[58] A. Povzner, D. Sawyer, and S. Brandt. Horizon: efficient deadline-driven disk I/O

management for distributed storage systems. In *Proceedings of the International Symposium on High Performance Distributed Computing*, HPDC '10.

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

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

[61] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *24th international Conference on Very Large Databases (VLDB '98)*, New York, NY, 1998.

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

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

[64] R. Ross and et. al. Storage Systems and Input/Output to Support Extreme Scale Science. In *Report of the DOE Workshops on Storage Systems and Input/Output.*

[65] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody

Ever Got Fired for Using Hadoop on a Cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[66] A. Samuels. The Consequences of Infinite Storage Bandwidth. In *OpenStack Summit 2016*, OpenStack '16, 2016.

[67] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, 2002.

[68] K. Schnaitter, N. Polyzotis, and L. Getoor. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. In *Proceedings of the VLDB Endowment*, VLDB '09.

[69] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003.

[70] M. Schwarzkopf, D. G. Murray, and S. Hand. The seven deadly sins of cloud computing research. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[71] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn. A Framework for an In-depth Comparison of Scale-out and Scale-up. In *Proceedings of the 2nd In-*

ternational Workshop on Data Intensive Scalable Computing at SuperComputing '13, DISCS'13, 2013.

[72] M. A. Sevilla, I. Jimenez, N. Watkins, S. Finkelstein, J. LeFevre, P. Alvaro, and C. Maltzahn. Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, 2018.

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

[74] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the European Conference on Computer Systems*, EuroSys '17, 2017.

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

[76] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. In *Proceedings of the VLDB Endowment*, VLDB '15.

[77] K. o. V. Shvachko. HDFS Scalability: The Limits to Growth.

[78] K. V. Shvachko, H. Kuang, S. Radia, and bert Chansler. The hadoop distributed file system. In *MSST2010*, Incline Village, NV, May 3-7 2010.

[79] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger. A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In *USENIX ATC '10*, ATC '10, Boston, MA, June 23-25 2010.

[80] D. Terry. Replicated Data Consistency Explained Through Baseball. 56.

[81] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

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

[83] N. Tran and D. A. Reed. ARIMA Time Series Modeling and Forecasting for Adaptive I/O Prefetching. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, SC '01.

[84] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

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

[86] F. Wang, H. S. Oral, G. M. Shipman, O. Drokin, D. Wang, and H. Huang. Understanding Lustre Internals. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2009.

[87] N. Watkins, M. A. Sevilla, I. Jimenez, K. Dahlgren, P. Alvaro, S. Finkelstein, and C. Maltzahn. DeclStore: Layering Is for the Faint of Heart. In *HotStorage '17*.

[88] S. A. Weil. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, University of California at Santa Cruz, December 2007.

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

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

[91] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhu. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, 2008.

[92] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proccedings of the 6th Conference on File and Storage Technologies*, FAST '08.

[93] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

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

[95] J. Xing, J. Xiong, N. Sun, and J. Ma. Adaptive and Scalable Metadata Management to Support a Trillion Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[96] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science.

[97] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design & Implementation*, NSDI'12, 2012.

[98] Q. Zheng, K. Ren, and G. Gibson. BatchFS: Scaling the File System Control

Plane with Client-funded Metadata Servers. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '14.

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

[100] Y. Zhu, H. Jiang, J. Wang, and F. Xian. HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. *IEEE Transactions on Parallel Distribed Systems*, 19(6), June 2008.