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

Migrating resources is a useful tool for balancing load in a distributed system. Today's systems can already virtualize memory and the ability to migrate other resources, such as CPU, disks, and network, is fast approaching. When we finally have the ability to migrate different resources, how do we know when and where to move them? Such migration will depend on the utilization, configuration, and workload, but how will we weight these factors to design robust, guaranteeable systems? In this work, we propose using metadata management as a substrate for exploring different heuristics for resource migration and load balancing. This work will address general load balancing topics in the context of metadata management, such as which metrics are important, how to quantify performance, which metrics should be optimized for, and which heuristics are successful. We use the Ceph file system as a platform for attacking the metadata management problem because it was built with locality in mind and the tools for resource migration and hotspot detection are already implemented.

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 [50], 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 [48, 43, 47, 24, 33].

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 [64] pushes resource management into software runtimes, which must now manage large numbers of heterogeneous devices. The increasing momentum behind disaggregated storage [28, 27], 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 [44]. 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" [50, 59]. 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 [8] 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. My thesis uses the programmable storage approach to embed policy engines into the metadata substrates to control the behavior, performance, and transparency of the entire software stack.

1.1 Research questions

We propose investigating automatic load distribution with the awareness that distribution depends on the workload. In this process, we will answer important load balancing 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 and Related Work

2.1 Global Namespace Scalability

In general, namespaces resolve names to data. Traditionally, namespaces are hierarchical and flat namespaces are outside the scope of this work. Although file system are the most popular example, other instances include DNS, LAN network topologies, and scoping in programming languages. File system namespaces are popular because they fit our mental model as humans and they are part of the POSIX IO standard. The momentum of namespaces as a mental model and the overwhelming amounts of legacy code written for namespaces makes this data model that will be around for a long time.

Whenever a file is created, modified, or deleted, the client must access the file's metadata. Single node file systems look in one location on disk for metadata. Using hte name of the file they index into a data structure that returns the inode number. Armed with that inode number, the client can seek to some location on disk for the

data. Distributed file systems use the same idea but at a larger scale, with clusters, more data, and networking.

In distributed file systems, serving metadata and maintaining a file system namespace is sufficiently challenging because metadata requests result in small, frequent accesses to the underlying storage system [41]. This skewed workload is very different from data I/O workloads. As a result, file system metadata services do not scale for sufficiently large systems in the same way that read and write throughput do [1, 2, 3, 61]. Furthermore, clients expect fast metadata access, making it difficult to apply data compressions and transformations [29]. When developers scale file systems, the metadata service becomes the performance critical component.

Although this important metadata problem was once reserved for high-performance computing (HPC), it has recently found its way into large data centers. For example, Google has acknowledged a strain on their own metadata services because today's workloads often deal with many small files (*e.g.*, log processing) and a large amount of simultaneous clients (*e.g.*, MapReduce jobs) [31]. Metadata inefficiences have also plagued Facebook; they migrated away from file systems for photos [4] and aggressively concatenate and compress many small files so their Hive queries do not impose too many small files on the HDFS namenode [55].

To combat the speed and dynamic nature of these metadata workloads, the community has turned to metadata clusters instead of single metadata servers [36, 61, 62, 53, 66]. A common technique for metadata clusters to improve metadata performance is load balancing across servers. These solutions are perfect for our study of

resource migration because they are some of the only systems that migrate the resources themselves, in the form of directories and directory fragments.

2.2 Approaches and Techniques

2.2.1 Global Semantics: Strong consistency

2.2.2 Leases

2.2.3 Locks

2.2.4 Capabilities

2.2.5 Global Semantics: Durability

2.2.6 Journal of updates

2.2.7 Hierarchical Semantics: Ownership

2.2.8 Path Traversals

2.2.9 Workload Locality: Within Directories

2.2.10 Workload Locality: Create Flash Crowds

2.2.11 Workload Locality: Listing Directories

Chapter 3

Mantle as a File System Metadata Load Balancer

3.1 Introduction

Accessing metadata is a bottleneck for large file systems. Single disk POSIX file systems consult metadata before seeking to the data. First they translate the file name into an inode. Then they use that inode to lookup their metadata in an inode table, which is at some fixed location on disk.

Serving metadata and maintaining a POSIX namespace is challenging for large-scale distributed file systems because accessing metadata imposes small and frequent requests on the underlying storage system [41]. As a result of this skewed workload, serving metadata requests does not scale for sufficiently large systems in the same way that read and write throughput do [2, 3, 61]. Many distributed file systems decouple

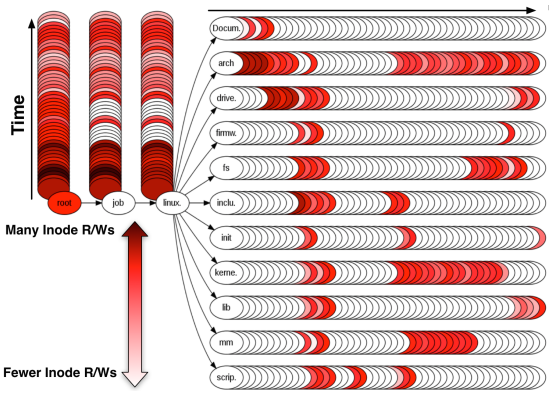


Figure 3.1: Metadata hotspots, represented by different shades of red, have spatial and temporal locality when compiling the Linux source code. The hotspots are calculated using the number of inode reads/writes and smoothed with an exponential decay.

metadata from data access so that data and metadata I/O can scale independently [3, 15, 20, 61, 63, 66]. These "metadata services" manage the namespace hierarchy and metadata requests (*e.g.*, file and directory creates, file and directory renaming, directory listings). File properties that a metadata service manages can include permissions, size, modification times, link count, and data location.

Unfortunately, decoupling metadata and data is insufficient for scaling and many setups require customized application solutions for dealing with metadata intensive workloads. For example, Google has acknowledged a strain on their own metadata services because their workloads involve many small files (*e.g.*, log processing) and simultaneous clients (*e.g.*, MapReduce jobs) [31]. Metadata inefficiencies have also plagued Facebook; they migrated away from file systems for photos [4] and aggressively concatenate and compress small files so that their Hive queries do not overload the HDFS

namenode [55]. The elegance and simplicity of the solutions stem from a thorough understanding of the workloads (*e.g.*, temperature zones at Facebook [34]) and are not applicable for general purpose storage systems.

The most common technique for improving the performance of these metadata services is to balance the load across dedicated metadata server (MDS) nodes [36, 61, 62, 53, 66]. 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 [36, 40, 67, 20, 69], inode path/permission caching [**?**, 30, 66], locality-aware/inter-object transactions [53, 69, 39, 40] and efficient lookup tables [**?**, 69]. Despite having mechanisms for migrating metadata, like locking [53, 45], zero copying and two-phase commits [53], and directory partitioning [66, 36, 40, 61], these systems fail to exploit locality.

File system workloads have locality because the namespace has semantic meaning; data stored in directories is related and is usually accessed together. Figure 3.1 shows the metadata locality when compiling the Linux source code. The "heat" of each directory is calculated with per-directory metadata counters, which are tempered with 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. Exploiting this locality has positive implications for performance because it reduces the number of requests, lowers the communication across MDS nodes, and eases memory pressure. The Ceph [61]

(see also www.ceph.com) file system (CephFS) tries to leverage this spatial, temporal, and request-type locality in metadata intensive workloads using dynamic subtree partitioning, but struggles to find the best degree of locality and balance.

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)

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

and that searching for balance too aggressively increases the standard deviation in runtime.

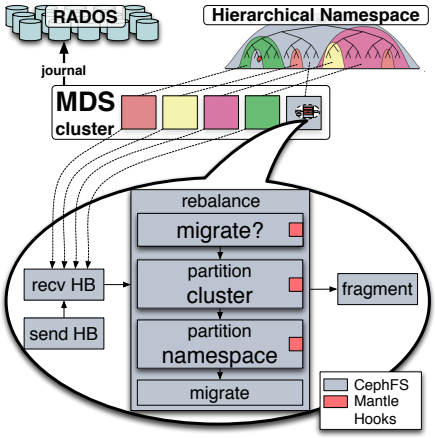


Figure 3.2: The MDS cluster journals to RADOS and exposes a namespace to clients. Each MDS makes decisions by exchanging heartbeats and partitioning the cluster/namespace. Mantle adds code hooks for custom balancing logic.

3.2 Background: Dynamic Subtree Partitioning

We use Ceph [61] to explore the metadata management problem. Ceph 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 by calculating the data placement ("where" should I store my data) and location ("where" did I store my data) using a hash-based algorithm (CRUSH). CephFS is the POSIX-compliant file system that uses RADOS. It decouples metadata and data access, so data IO is done directly with RADOS while all metadata operations go to a separate metadata cluster. The MDS cluster is connected to RADOS so it can periodically flush its state. The hierarchical namespace is kept in the collective memory of the MDS

cluster and acts as a large distributed cache. Directories are stored in RADOS, so if the namespace is larger than memory, parts of it can be swapped out.

The MDS nodes use dynamic subtree partitioning [62] to carve up the namespace and to distribute it across the MDS cluster, as shown in Figure 3.2. 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 3.2. 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 [62, 61]). 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+ [36] 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 §3.2.2.3). 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 §3.2.2.3), 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.

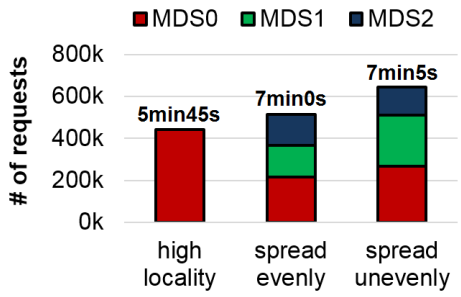
CephFS's Client-Server Metadata Protocols: the mechanisms for migrating metadata, ensuring consistency, enforcing synchronization, and mediating access are discussed at great length in [60] and the Ceph source code. MDS nodes and clients cache a configurable number of inodes so that requests like `getattr` 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.

3.2.1 Advantages of Locality

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 [60, 62], but briefly, improving locality can:

- reduce the number of forwards between MDS nodes (*i.e.* requests for metadata

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



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

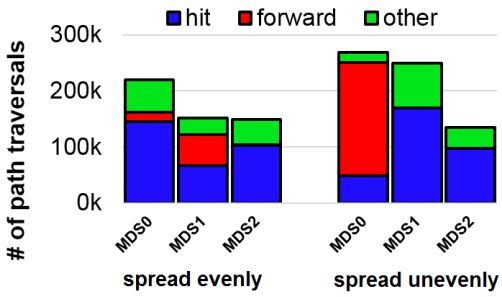


Figure 3.3: Spreading metadata to multiple MDS nodes hurts performance ("spread evenly/unevenly" setups in Figure 3a) when compared to keeping all metadata on one MDS ("high locality" setup in Figure 3a). The times given are the total times of the job (compile, read, write, etc.). Performance is worse when metadata is spread unevenly because it "forwards" more requests (Figure 3b).

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 3.3 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%. 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 3.3b 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.

3.2.2 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

[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.

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.

3.2.2.1 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 3.4 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

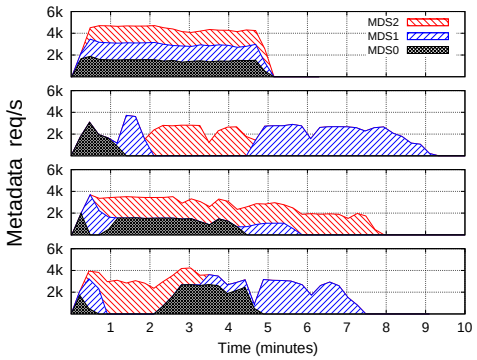


Figure 3.4: The same create-intensive workload has different throughput (y axis; curves are stacked) because of how CephFS maintains state and sets policies.

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 performance and unpredictability.

3.2.2.2 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 measurement 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.

3.2.2.3 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. Tun-

able 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. 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.

The CephFS Policies

The CephFS policies, shown in Table 3.1, 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 3.1.

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

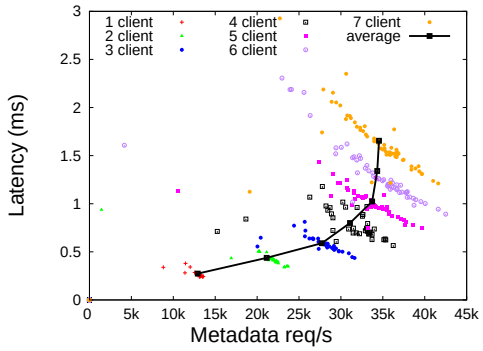


Figure 3.5: For the create heavy workload, the throughput (x axis) stops improving and the latency (y axis) continues to increase with 5, 6, or 7 clients. The standard deviation also increases for latency (up to 3×) and throughput (up to 2.3×).

move inodes; it migrates load when it thinks that it has more load than the other MDS nodes ("when" row of Table 3.1) and it tries to migrate enough of its load to make the load even across the MDS cluster ("where" row of Table 3.1). While this approach is scalable, it reduces the efficiency of the cluster if the job could have been completed with less MDS nodes. Figure 3.5 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 3.1). 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.

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 3.1: 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.

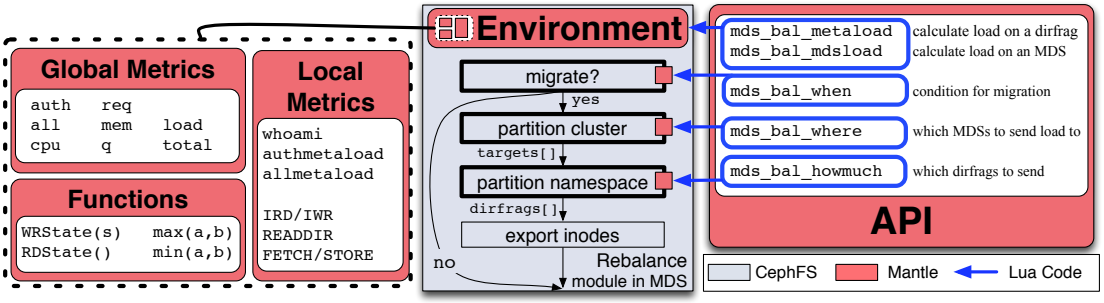


Figure 3.6: Designers set policies using the Mantle API. The injectable code uses the metrics/functions in the environment.

3.3 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 [62], they do not work for others (as demonstrated in Figure 3.4), 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 [32] 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 3.6. 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.

3.3.1 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 [16]. 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,

shown on the left side of Figure 3.6, 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 4.1. The labeled arrows between the phases in Figure 3.6 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.

3.3.2 The Mantle API

Figure 3.6 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 4.1, 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 §3.2.2.3 (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.

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

3.4 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.

3.4.1 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+ [36, 40]. 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

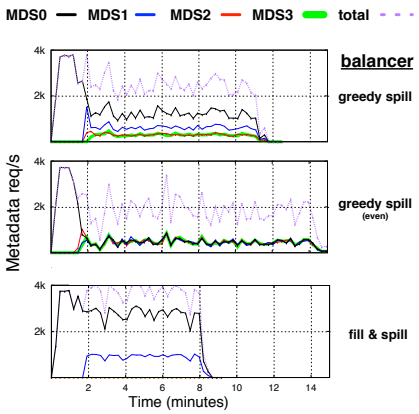


Figure 3.7: With clients creating files in the same directory, spilling load unevenly with Fill & Spill has the highest throughput (curves are not stacked), which can have up to 9% speedup over 1 MDS. Greedy Spill sheds half its metadata immediately while Fill & Spill sheds part of its metadata when overloaded.

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 3.7 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

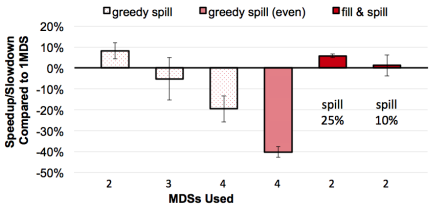


Figure 3.8: The per-client speedup or slowdown shows whether distributing metadata is worthwhile. Spilling load to 3 or 4 MDS nodes degrades performance but spilling to 2 MDS nodes improves performance.

balancing shown in the second graph of Figure 3.7, the balancer is modified according to Listing 2 to partition the cluster when selecting the target MDS.

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 3.8. This agrees with the assessment of the capacity of a single MDS in Section §3.2.2.3; 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

```
-- 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 4.1). Note that all subsequent balancers use the same metadata and MDS loads.

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]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.

```
-- 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.

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.

3.4.2 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, 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 §3.2.2.3. 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 3.8, 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.

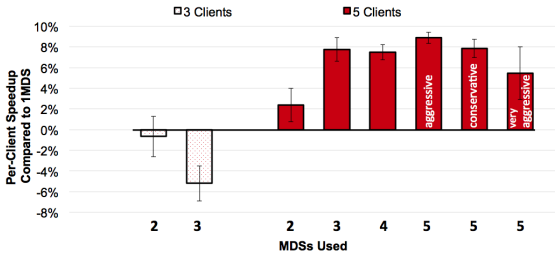


Figure 3.9: For the compile workload, 3 clients do not overload the MDS nodes so distribution is only a penalty. The speedup for distributing metadata with 5 clients suggests that an MDS with 3 clients is slightly overloaded.

3.4.3 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 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

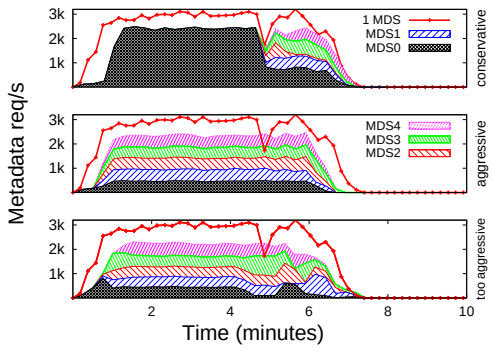


Figure 3.10: With 5 clients compiling code in separate directories, distributing metadata load early helps the cluster handle a flash crowd at the end of the job. Throughput (stacked curves) drops when using 1 MDS (red curve) because the clients shift to linking, which overloads 1 MDS with `readdir`s.

uses a wide range of dirfrag selectors.

Figure 3.9 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 3.10 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 3.10 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.

3.4.4 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.

```
-- 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.

3.5 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 [20] and SkyFS [66] hash the filename to locate the authority for metadata. CalvinFS [54] 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.

To further enhance scalability, many hashing schemes employ dynamic load balancing. [30] presented dynamic balancing formulas to account for a forgetting factor, access information, and the number of MDS nodes in elastic clusters. [66] used a master-slave architecture to detect low resource usage and migrated metadata using a consistent hashing-based load balancer. GPFS [45] elects MDS nodes to manage metadata 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+ [36]

alleviates hotspots and "flash crowds" by allowing unsynchronized directory growth for create intensive workloads. Clients contact the parent and traverse down its "partition history" to find which authority MDS has the data. The follow-up work, IndexFS [36], distributes whole directories to different nodes. To improve lookups and creates, clients cache paths/permissions and metadata logs are stored in a log-structured merge tree for fast insertion and lookup. Although these techniques improve performance and scalability, especially for create intensive workloads, they do not leverage the locality inherent in file system workloads and they ignore the advantages of keeping the required number of servers to a minimum.

Many hashing systems achieve locality by adding a metadata cache [30, 66, 69]. For example, Lazy Hybrid [**?**] hashes the filename to locate metadata but maintains extra per-file metadata to manage permissions. Caching popular inodes can help improve locality, but this technique is limited by the size of the caches and only performs well for temporal metadata, instead of spatial metadata locality. Furthermore, cache coherence requires a fair degree of sophistication, limiting its ability to dynamically adapt to the flash crowds.

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 [23] distributes inode ranges round robin to all servers and HBA [69] distributes metadata randomly to each server and uses bloom filters to speedup the table lookups. These techniques also ignore locality.

Traverse it - Subtree Partitioning: this technique assigns subtrees of the

hierarchal namespace to MDS nodes and most systems use a static scheme to partition the namespace at setup, which requires an administrator. Ursa Minor [53] and Farsite [12] traverse the namespace to assign related inode ranges, such as inodes in the same subtree, to servers. This benefits performance because the MDS nodes can act independently without synchronizing their actions, making it easy to scale for breadth assuming that the incoming data is evenly partitioned. Subtree partitioning also gets good locality, making multi-object operations and transactions more efficient. If carefully planned, the metadata distributions can achieve both locality and even load distribution, but their static distribution limits their ability to adapt to hotspots/flash crowds and to maintain balance as data is added. Some systems, like Panasas [63], allow certain degrees of dynamicity by supporting the addition of new subtrees at runtime, but adapting to the current workload is ignored.

3.6 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 4

Mantle as a Data Management Control Plane

4.1 Implementing Mantle on Programmable Storage

Mantle [51] is a programmable load balancer that separates the metadata balancing policies from their mechanisms. Administrators inject code to change how the metadata cluster distributes metadata. Our previous work showed how to use Mantle to implement a single node metadata service, a distributed metadata service with hashing, and a distributed metadata service with dynamic subtree partitioning.

The original implementation was "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.

4.1.0.1 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.

4.1.0.2 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.

4.1.0.3 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.

4.2 Evaluation

Our evaluation demonstrates the feasibility of building new service abstractions atop programmable storage, focusing on the performance of the internal abstractions exposed by Malacology and used to construct the Mantle and ZLog services. We also discuss latent capabilities we discovered in this process that let us navigate different trade-offs within the services themselves. First, we benchmark scenarios with high sequencer contention by examining the interfaces used to map ZLog onto Malacology; specifically, we describe the sequencer implementation and the propagation of object and data interfaces interfaces. Next, 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 the programmability of 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.

4.2.1 Load Balancing ZLog Sequencers with Mantle

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

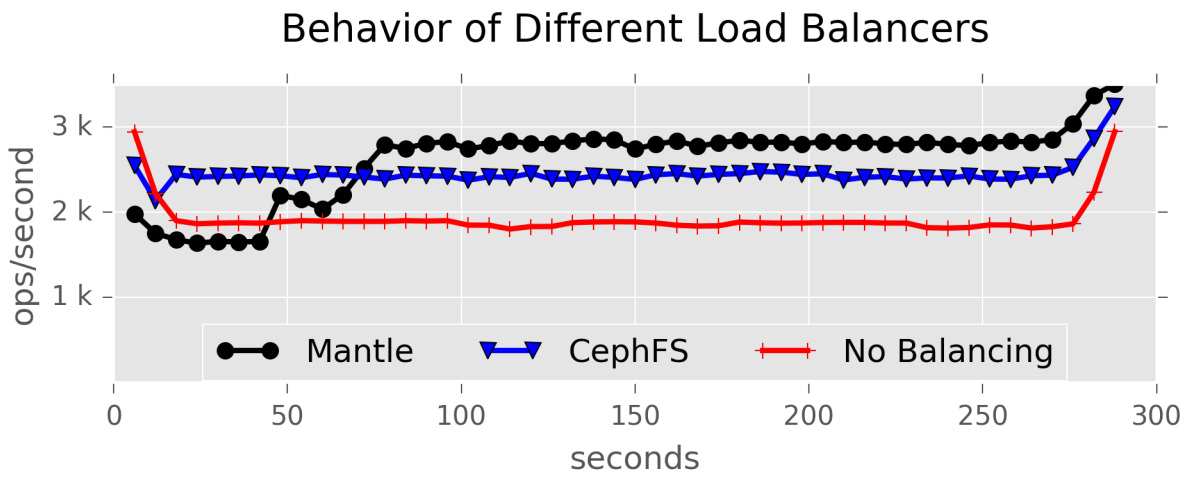


Figure 4.1: [source] CephFS/Mantle load balancing have better throughput than co-locating all sequencers on the same server. Sections 4.2.1.1 and 4.2.1.2 quantify this improvement; Section 4.2.1.3 examines the migration at 0-60 seconds.

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.

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

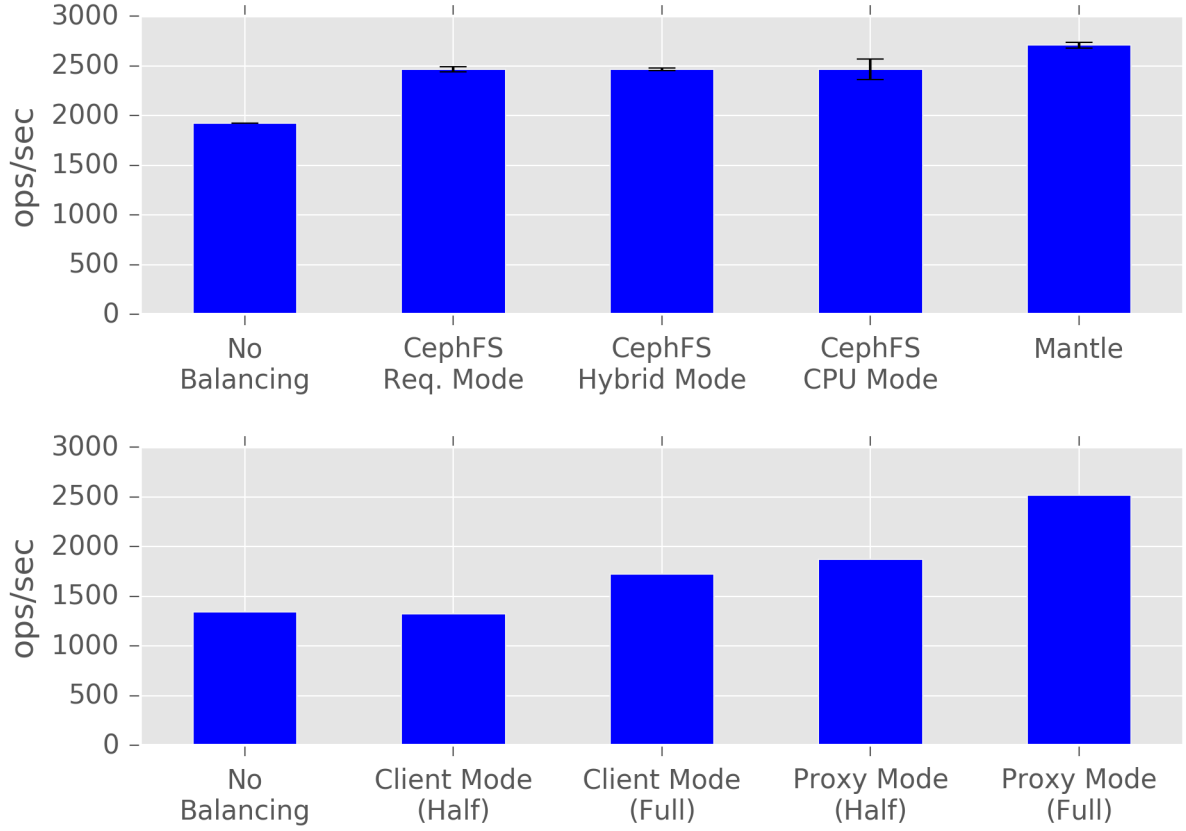


Figure 4.2: [source, source] In (a) all CephFS balancing modes have the same performance; Mantle uses a balancer designed for sequencers. In (b) the best combination of mode and migration units can have up to a 2× improvement.

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 4.1 shows the throughput over time of different load balancers as they migrate 3 sequencers (with 4 clients) around the cluster; "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 4.1 also demonstrates features that we will explore in the rest of this section. Sections 4.2.1.1 and 4.2.1.2 quantify the differences in performance when the cluster stabilizes at time 100 seconds and Section 4.2.1.3 examines the slope and start time of the re-balancing phase between 0 and 60 seconds by comparing the aggressiveness of the balancers.

4.2.1.1 Feature: Balancing Modes

Next, we quantify the performance benefits shown in Figure 4.1. 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 4.1, 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 4.2 (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 4.2 (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 4.3. "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 4.1. 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 4.1;

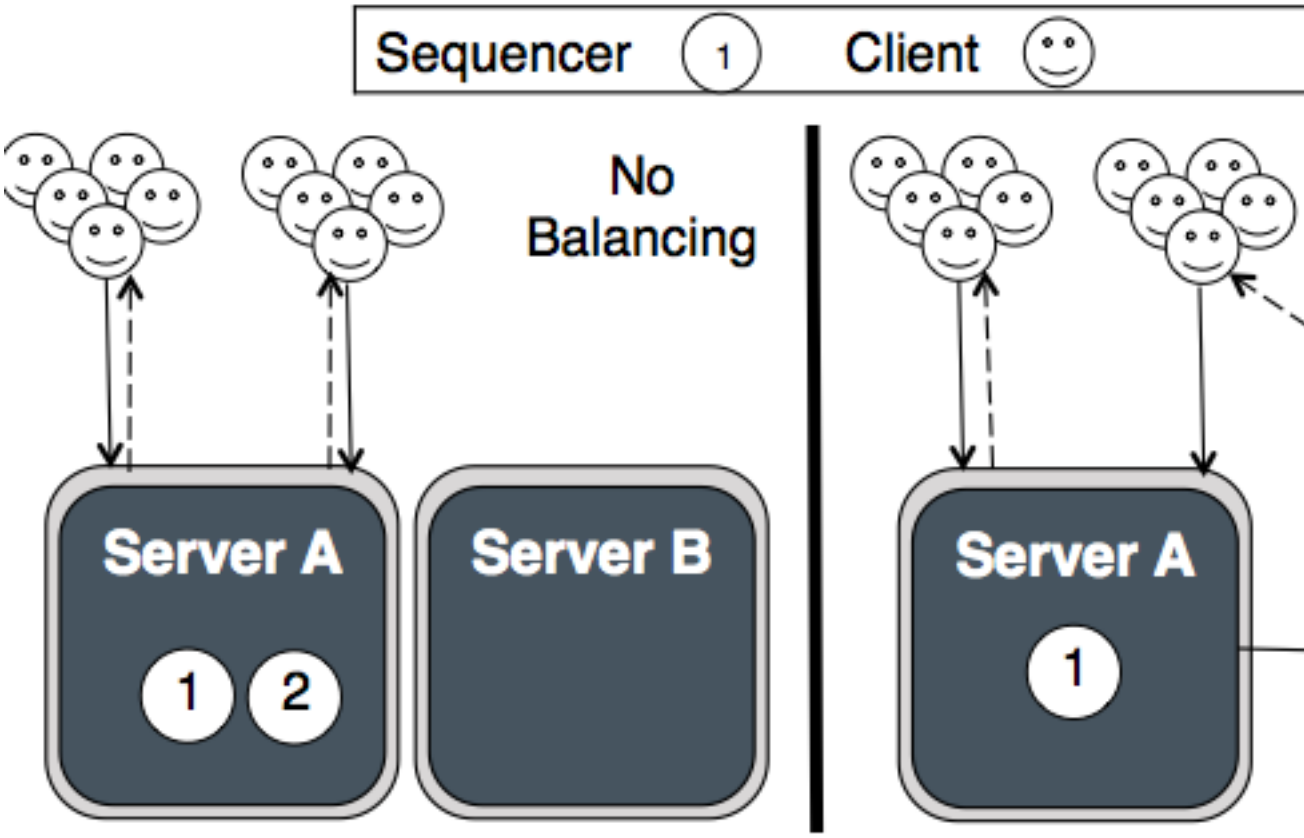


Figure 4.3: In client mode clients sending requests to the server that houses their sequencer. In proxy mode clients continue sending their requests to the first server.

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 4.3, shows how clients for sequencer 2 contact server B without a redirect from server A.

Figure 4.4 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

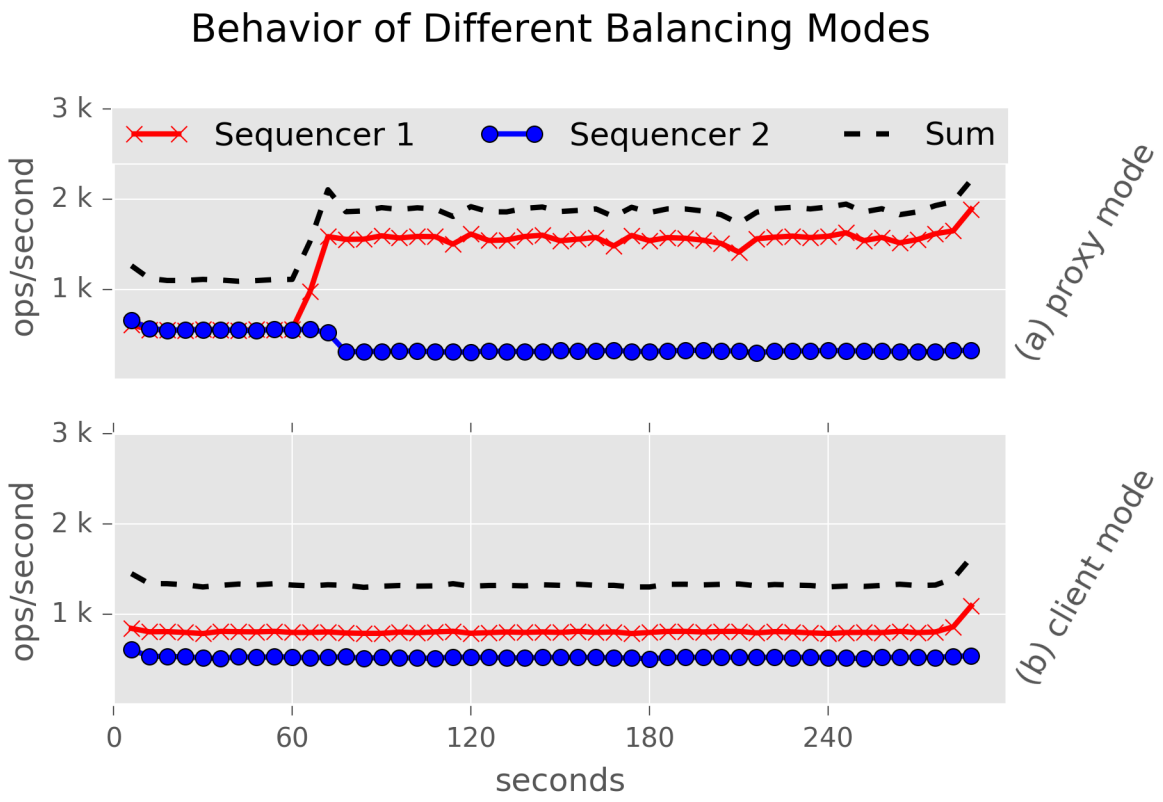


Figure 4.4: [source] The performance of proxy mode achieves the highest throughput but at the cost of lower throughput for one of the sequencers. Client mode is more fair but results in lower cluster throughput.

for both sequencers in Figure 4.4(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 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 [57].

Cluster throughput improves at the cost of decreased throughput for Sequencer 2. Figure 4.4(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.

4.2.1.2 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 Balancing 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 4.3, 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 [51] 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 4.2 (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.

4.2.1.3 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 4.1 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 4.1 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 4.1 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.

4.3 Introduction

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 [40, 36, 62]. 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 [65, 9, 51, 62, 61]. 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 [37].

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

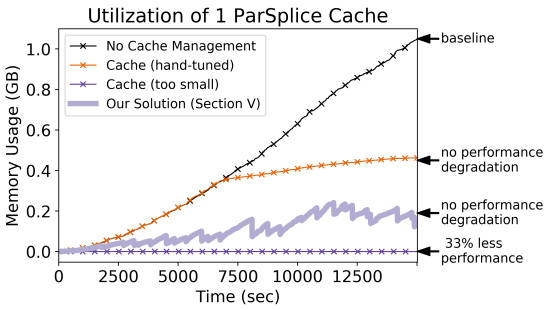


Figure 4.5: Using our data management language and policy engine, we design a dynamically sized caching policy (thick line) for ParSplice. Compared to existing configurations (thin lines with ×'s), our solution saves the most memory without sacrificing performance and works for a variety of inputs.

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 4.5, 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

[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.

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 4.5 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 4.5, 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. 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 [51], 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. 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, turn out to also be effective for cache management.

This last contribution is explored in Sections §4.6 and §4.7, where we try a range of policies from different disciplines; but more importantly, in Section §4.8, 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 [61]) suggesting that there is potential for automatically adapting and generating policies dynamically.

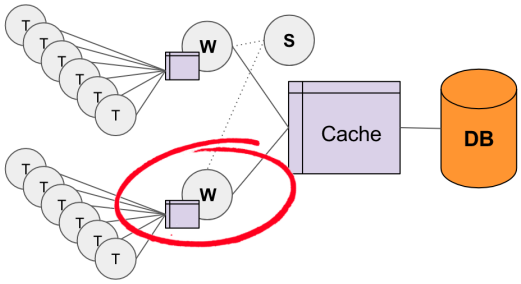


Figure 4.6: The ParSplice architecture has a storage hierarchy of caches (boxes) and a dedicated cache process (large box) backed by a persistent database (DB). A splicer (S) tells workers (W) to generate segments and workers employ tasks (T) for more parallelization. We focus on the worker's cache (circled), which facilitates communication and segment exchange between the worker and its tasks.

4.4 ParSplice Keyspace Analysis

ParSplice [37] 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.

4.4.1 Background

As shown in Figure 4.6, the phases are:

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

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

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.

4.4.2 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 4.6.

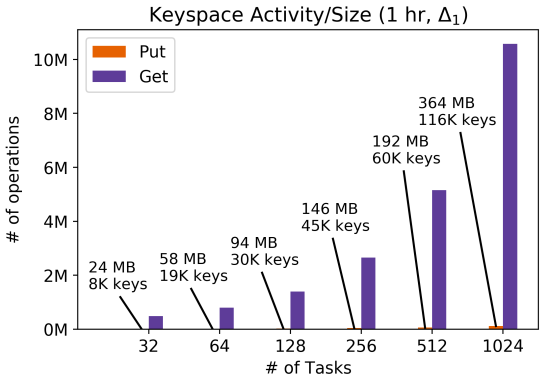


Figure 4.7: The keyspace is small but must satisfy many reads as workers calculate segments. Memory usage scales linearly, so it is likely that we will need more than one node to manage segment coordinates when we scale the system or jobs up.

4.4.3 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.

4.4.3.1 Scalability

Figure 4.7 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.

4.4.3.2 An active but small keyspace

The bars in Figure 4.7 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.

4.4.3.3 Initial conditions influence key activity

Figure 4.8 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

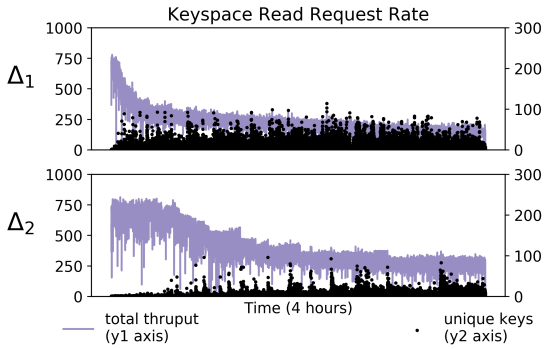


Figure 4.8: Key activity for ParSplice starts with many reads to a small set of keys and progresses to less reads to a larger set of keys. The line shows the rate that EOM minima values are retrieved from the key-value store ($y1$ axis) and the points along the bottom show the number of unique keys accessed in a 1 second sliding window ($y2$ axis). Despite having different growth rates (Δ), the structure and behavior of the key activities are similar.

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. 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.

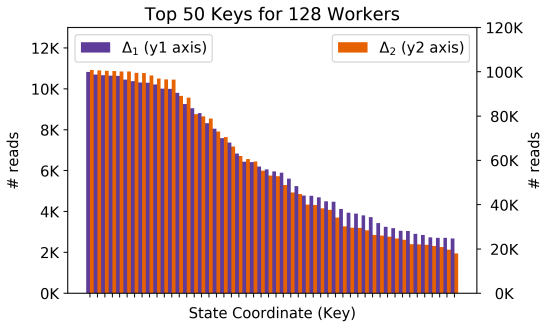


Figure 4.9: Over time, tasks start to access a larger set of keys resulting in some keys being more popular than others. Despite different growth rates (Δ), the spatial locality of key accesses is similar between the two runs. (*e.g.*, some keys are still read 5 times as many times others).

4.4.3.4 Entropy increases over time

The reads per second in Figure 4.8 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 Section §4.6). The resulting key access imbalance for the two growth rates in Figure 4.8 are shown in Figure 4.9, 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 4.9 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).

4.5 Methodology

To explore software-defined cache management, we use the data management language and policy engine presented in [51]. 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.*, $2 \times \texttt{cpu} + 3 \times \texttt{memory usage}$).

The succinctness of the API lets users inject multiple, possibly dynamic, policies. 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.

4.5.1 Extracting Mantle as a Library

We extracted Mantle as a library and Figure 4.10 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

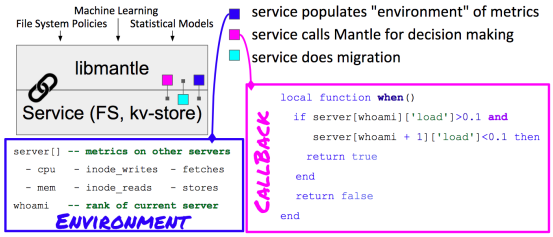


Figure 4.10: Extracting Mantle as library.

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 4.10 and described below.

4.5.1.1 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 4.1 shows how these metrics are accessed from the policies written by administrators.

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

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

is a resource name. Metrics used for file system metadata load balancing are shown by the "environment" box in Figure 4.10. 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
```

4.5.1.2 Policies Written as Callbacks

the "callback" box in Figure 4.10 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.

4.5.2 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 §4.6 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 §4.7, 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 §4.7 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.

4.6 Cache Management Using 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 §4.4.3. 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 4.11. "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

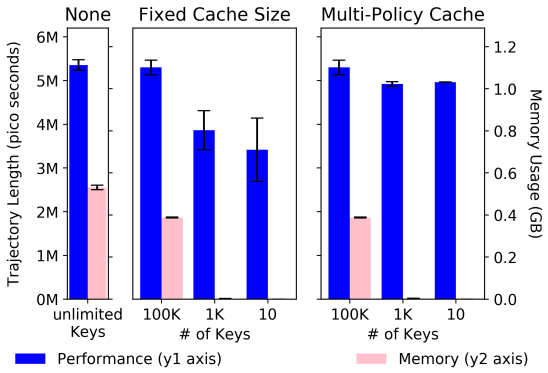


Figure 4.11: Policy performance/utilization shows the trade-offs of different sized caches (x axis). "None" is ParSplice unmodified, "Fixed Sized Cache" evicts keys using LRU, and "Multi-Policy Cache" switches to fixed sized cache after absorbing the workload's initial burstiness. This parameter sweep 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 4.8 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 4.8 are per second and are not representative of the actual active keyspace; the number of active keys is larger than 100, as some keys may be accessed

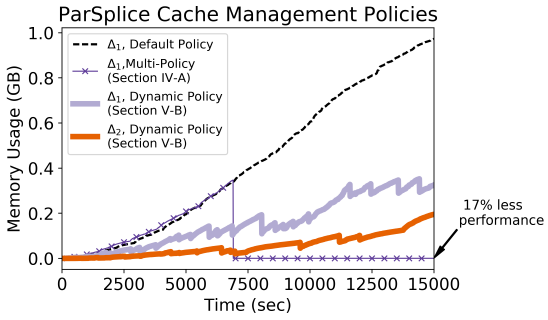


Figure 4.12: Memory utilization for "No Cache Management" (unlimited cache growth), "Multi-Policy" (absorbs initial burstiness of workload), and "Dynamic Policy" (sizes cache according to key access patterns). The dynamic policies saves the most memory without sacrificing performance.

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:

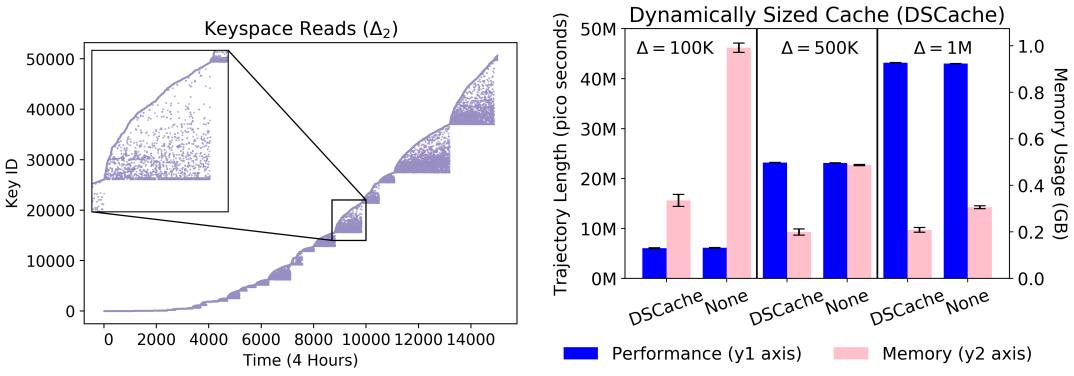
- unlimited growth policy: cache increases on every write
- n key limit policy: cache constrained to n keys

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 §4.8 in Figure 4.15a. The

plot on the right side of Figure 4.11 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 4.11 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 4.11 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 4.12 but in Figure 4.11 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 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).



(a) 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 §4.7.2 and the results are in Figure 4.13b.

(b) 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× as much memory in the best case.

Figure 4.13: Different cache management policies tested over the Mantle policy engine.

4.7 Cache Management Using 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 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 4.13a 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 4.13a, 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 4.13a 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

4.7.1 Failed Strategies

To detect the access patterns in Figure 4.13a, 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.

4.7.2 Dynamically Sized Cache: Access Pattern Detection

After trying these techniques we found that the basic O(n) algorithm in Figure 4.14 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 4.13a 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 4.13b. 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 4.12, where the behavior resembles the key access patterns in Figure 4.13a[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 4.13a 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 4.14: 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 4.13b and the memory usage over time is in Figure 4.12.

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

(a) 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 4.11.

```
local function when()
  if servers[whoami]["load"] > target then
    overloaded = RDstate() + 1
    WRstate(overloaded)
    if overloaded > 2 then
      return true
    end
  end
  else then WRstate(0) end
  return false
end
```

(b) CephFS file system metadata load balancer, designed in 2004 in [62], reimplemented in Lua in [51]. This policy has many similarities to the ParSplice cache management policy.

Figure 4.15: ParSplice's cache management policy has the same components as CephFS's load balancing policy.

4.8 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 [51] with the ones we designed and evaluated above

for cache management in ParSplice.

4.8.1 Using Load Balancing Policies for Cache Management

From a high-level the cache management policy we designed in Figure 4.15a 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 4.15b, which was presented in [51]. 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. The two policies have the following in common:

Condition for "Overloaded" (Fig. 4.15a: Line 2; Fig. 4.15b: 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. 4.15a: Lines 4,6; Fig 4.15b: 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. 4.15a: Line 6; Fig. 4.15b: Line 5) - after deter-

mining 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.

4.8.2 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 4.16, 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 4.13a.

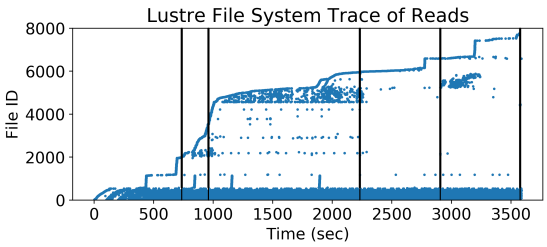


Figure 4.16: File system metadata reads for a Lustre trace collected at LANL. The vertical lines are the access patterns detected by the ParSplice cache management policy from Section §4.7. A file system that load balances metadata across a cluster of servers could use the same pattern detection to make 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.

Although other access pattern detection algorithms are possible, we use the one designed for cache management in Section §4.7.2 with slight modifications based on our knowledge of file systems[4]. The vertical lines in Figure 4.16 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".

4.8.3 Other Use Cases

Storage systems have many other data management techniques that would benefit from the caching policies developed in Sections §4.6 and §4.7. 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 [38] 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 [21] "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 [40] 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 [56].

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

4.9 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 [25]. 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 [50] which seeks to provide fundamental services from within the storage system (*e.g.*, consensus) to the application. Our plan is to use MDHIM [17] 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 [40] partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal. ShardFS [65] takes the replication approach to the extreme by copying all directory state to all nodes. CephFS [62, 61] 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+ [35]

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 [6, 5], big data systems systems [19], and databases [46]. 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 [6]. We cannot drop these techniques into ParSplice because the magnitude and speed of the workload hotspots/flash crowds makes existing approaches less applicable.

4.10 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 5

Cudele for Programmable File System Consistency and Durability

5.1 Introduction

File system metadata services in HPC and large-scale data centers have scalability problems because common tasks, like checkpointing [7] or scanning the file system [67], contend for the same directories and inodes. Applications perform better with dedicated metadata servers [51, 40] but provisioning a metadata server for every client 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 [8]. These types of trends put pressure on data access because more requests end up hitting the same layer and latencies cannot be hidden

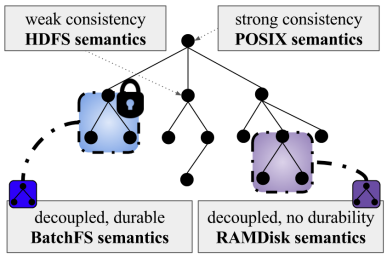


Figure 5.1: Illustration of subtrees with different semantics co-existing in a global namespace. For performance, clients can relax consistency on their subtree (HDFS) or decouple the subtree and move it locally (BatchFS, RAMDisk). Decoupled subtrees can relax durability for even better performance.

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 batch style jobs do not need the strong consistency that the file system provides, so BatchFS [67] and DeltaFS [68] 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 [7]).

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 [18, 68, 67, 40, 14]. Then the file system can change its internal structure to optimize performance. For example, the file system could enter a mode that prevents other clients from interfering with the decoupled directory. This delayed merge (*i.e.* a form of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of remote procedure calls (RPCs), synchronization, false sharing, and serialization. While the performance benefits of decoupling the namespace are obvious, applications that rely on the file system's guarantees must be deployed on an entirely different system or re-written to coordinate strong consistency/durability themselves.

To address this problem, we present an API and framework that lets developers dynamically control the consistency and durability guarantees for subtrees in the file system namespace. Figure 5.1 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 HDFS[1] subtree has weaker than strong consistency because it lets clients read files opened for writing [26], which means that not all updates are immediately seen by all clients; the BatchFS and RAMDisk subtrees are decoupled from the global namespace and have similar consistency/durability behavior to those systems; and the POSIX IO subtree retains the rigidity of POSIX IO's strong consistency. Subtrees without policies inherit the consistency/durability semantics of

[1]HDFS itself is not directly evaluated in this paper, although the semantics and their performance is explored in §5.6.2.

the parent and future work will examine embeddable or inheritable policies.

Our prototype system, Cudele, achieves this by exposing "mechanisms" that users 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 user a wide range of policies and optimizations that can be custom fit to an application. We make the following contributions:

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

2. This framework lets subtrees with different semantics co-exist in a global namespace. We show how this scales further and performs better than systems that use one strategy for the entire namespace .

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

The last contribution lays the groundwork for future work on our prototype. It is more scalable than the current practice of mounting different storage systems in a global namespace because there is no need to provision dedicated storage clusters to applications or move data between these systems. For example, the results of a Hadoop job do not need to be migrated into a Ceph file system (CephFS) for other processing; instead the user can change the semantics of the HDFS subtree into a CephFS subtree. This may cause metadata/data movement to make things strongly consistent again but this is superior to moving all data across file system boundaries. Our prototype enables

studies that adjust these semantics over *time and space*, where subtrees can change their semantics and migrate around the cluster without ever moving the data they reference.

The results in this paper pave the way for such a system and confirm the assertions of "clean-state" research that decouple namespaces; specifically that these techniques drastically improve performance (we see 104× speed up). We go a step further by quantifying the costs of merging updates (7× slow down) and maintaining durability (10× slow down). In our prototype, we get an 8× speedup and can scale to twice as many clients when we assign a more relaxed form of consistency and durability to a subtree with a create-heavy workload. We use Ceph as a prototyping platform because it is used in cloud-based and data center systems and has a presence in HPC [58].

5.2 Related Work

The bottlenecks associated with accessing POSIX IO file system metadata are not limited to HPC workloads and the same challenges that plagued these systems for years are finding their way into the cloud. Workloads that deal with many small files (*e.g.*, log processing and database queries [55]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [31]), are subject to the scalability of the metadata service. The biggest challenge is that whenever a file is touched the client must access the file's metadata and maintaining a file system namespace imposes small, frequent accesses on the underlying storage system [41]. Unfortunately, scaling file system metadata is a well-known problem and solutions for scaling data IO do not work for metadata IO [41, 2, 3, 61].

POSIX IO workloads require strong consistency and many file systems improve performance by reducing the number of remote calls per operation (*i.e.* RPC amplification). As discussed in the previous section, caching with leases and replication are popular approaches to reducing the overheads of path traversals but their performance is subject to cache locality and the amount of available resources, respectively; for random workloads larger than the cache extra RPCs hurt performance [40, 62] and for write heavy workloads with more resources the RPCs for invalidations are harmful. Another approach to reducing RPCs is to use leases or capabilities.

High performance computing has unique requirements for file systems (*e.g.*, fast creates) and well-defined workloads (*e.g.*, workflows) that make relaxing POSIX IO

sensible. BatchFS assumes the application coordinates accesses to the namespace, so the clients can batch local operations and merge with a global namespace image lazily. Similarly, DeltaFS eliminates RPC traffic using subtree snapshots for non-conflicting workloads and middleware for conflicting workloads. MarFS gives users the ability to lock "project directories" and allocate GPFS clusters for demanding metadata workloads. TwoTiers eliminates high-latencies by storing metadata in a flash tier; applications lock the namespace so that metadata can be accessed more quickly. Unfortunately, decoupling the namespace has costs: (1) merging metadata state back into the global namespace is slow; (2) failures are local to the failing node; and (3) the systems are not backwards compatible.

For (1), state-of-the-art systems manage consistency in non-traditional ways: IndexFS maintains the global namespace but blocks operations from other clients until the first client drops the lease, BatchFS does operations on a snapshot of the namespace and merges batches of operations into the global namespace, and DeltaFS never merges back into the global namespace. The merging for BatchFS is done by an auxiliary metadata server running on the client and conflicts are resolved by the application. Although DeltaFS never explicitly merges, applications needing some degree of ground truth can either manage consistency themselves on a read or add a bolt-on service to manage the consistency. For (2), if the client fails and stays down, all metadata operations on the decoupled namespace are lost. If the client recovers, the on-disk structures (for BatchFS and DeltaFS this is the SSTables used in TableFS) can be recovered. In other words, the clients have state that cannot be recovered if the node

stays failed and any progress will be lost. This scenario is a disaster for checkpoint-restart where missed cycles may cause the checkpoint to bleed over into computation time. For (3), decoupled namespace approaches sacrifice POSIX IO going as far as requiring the application to link against the systems they want to talk to. In today's world of software defined caching, this can be a problem for large data centers with many types and tiers of storage. Despite well-known performance problems POSIX IO and REST are the dominant APIs for data transfer.

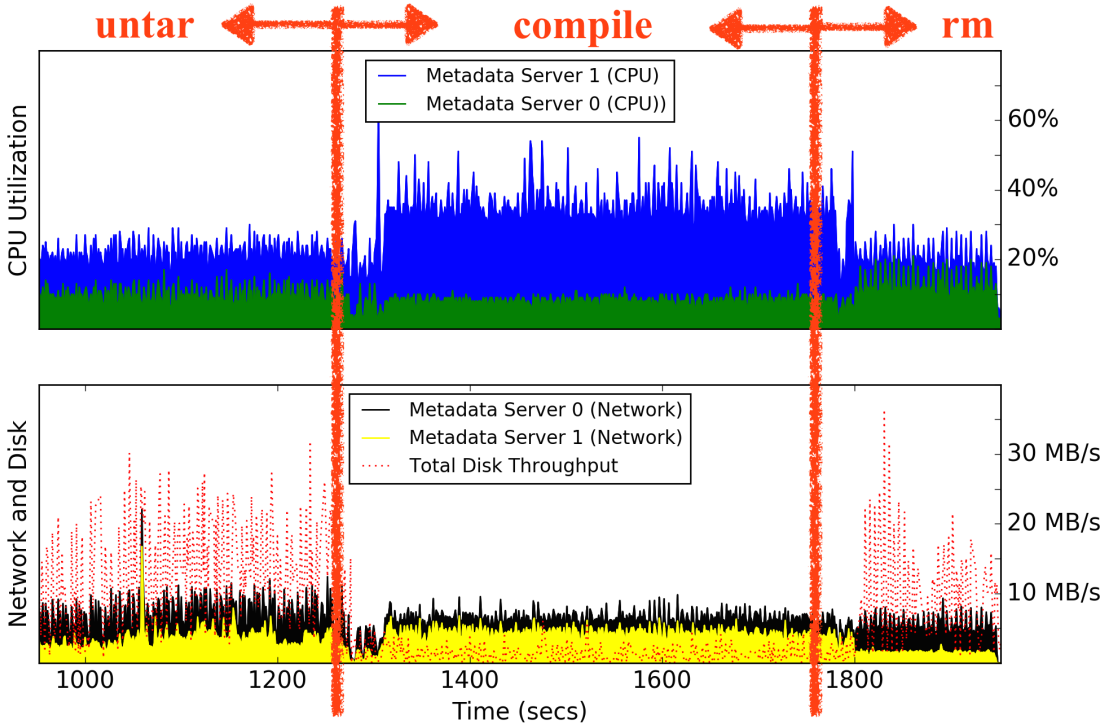


Figure 5.2: Create-heavy workloads (such as `untar`) incur the highest disk, network, and CPU utilization because of the consistency and durability demands of CephFS.

5.3 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 (*i.e.* RADOS) to store its data and metadata. To show how the file system behaves under high metadata load we use a create-heavy workload. 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

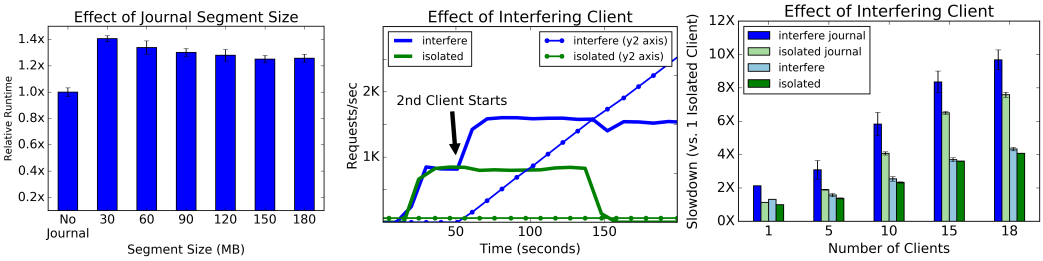
performance.

Figure 5.2 shows the resource utilization of compiling the Linux kernel in a CephFS mount. The `untar` phase, which is characterized by many creates, has the highest resource usage (combined CPU, network, and disk) because of the number of RPCs needed for consistency and durability. The RPCs are also serialized because the metadata server is single threaded; although naïve, this design is common because of the complexity of multi-threaded metadata servers [11]. In this section, we quantify the costs of consistency and durability in CephFS. At the end of each subsection we compare the approach to "decoupled namespaces", the technique in related work that detaches subtrees from the global namespace to relax consistency/durability guarantees.

5.3.1 Durability

While durability is not specified by POSIX IO, users expect that files they create or modify survive failures. We define three types of durability: global, local, and none. Global durability means that the client or server can fail at any time and metadata will not be lost 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 re-



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

Figure 5.3: The overhead of durability and strong consistency in CephFS. (a) shows the effect of different journal segment sizes, which are streamed into the object store for fault tolerance. (b) and (c) show that when a second client "interferes", capabilities are revoked and metadata servers do more work.

silient object store. Similar to LFS [42] and WAFL [22] 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 for controlling the journal: the segment size and the number of segments that can be written in parallel. Unless the journal saturates memory or CPU resources, larger values for these tunables results 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 5.3a shows the effect of journaling of with different segment sizes; the larger the segment size the bigger that the writes into the object store are. The trade-off for better performance is memory consumption because larger segments take up more space for buffering. When journaling is on, the metadata server periodically stops serving requests to flush (*i.e.* apply journal updates) to the metadata store. The journal overhead is sufficient enough to slow down metadata throughput but not so much as to overwhelm the bandwidth of the object store.

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 [39]) 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 [40]. IndexFS writes metadata to SSTables, which initially reside in memory but are later flushed to the underlying distributed file system.

5.3.2 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.

The benefits of caching the directory inode when creating files is shown in Figure 5.3b. The colors show the behavior of the client for two different runs. If only one client is creating files in a directory ("isolated" curve on $y1$ axis) then that client can lookup the existence of new files locally before issuing a create request to the metadata server. If another client starts creating files in the same directory then the directory inode transitions out of read caching and the first client must send `lookup()`s to the metadata server ("interfere" curve on $y2$ axis). These extra requests increase the throughput of the "interfere" curve on the $y1$ axis because the metadata server can handle the extra load but performance suffers. This degradation is shown in Figure 5.3c, where we scale the number of clients and show the slowdown of the slowest client. The results are normalized to a single isolated client without a metadata server journal and the error bars are the standard deviations of all client runtimes. For the "interfere"

bars, each client creates files in private directories and at 30 seconds we launch another process that creates files in those directories. 18 is the maximum number of clients the metadata server can handle for this metadata-intensive workload; at higher client load, the metadata server complains about laggy and unresponsive requests.

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.

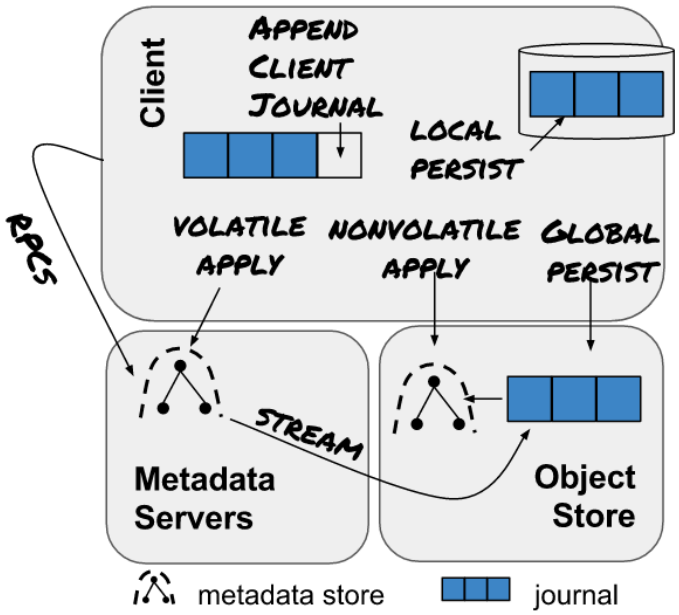


Figure 5.4: Illustration of the mechanisms used by applications to build consistency/durability semantics. Descriptions are provided in Table 5.1 and the underlined words in Section §5.4.1.

5.4 Methodology: Global Namespace, Subtree Consistency/Durability

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

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

Table 5.1: Descriptions of the mechanisms. Example compositions are shown in Table 5.2.

subtrees.

5.4.1 Mechanisms: Building Guarantees

Figure 5.4 shows the mechanisms (labeled arrows) in Cudele and which daemon(s) they are performed by. Table 5.1 has a description of what each mechanism does. Decoupled clients use the Append Client Journal mechanism to append metadata updates to a local, in-memory journal. 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.

5.4.1.1 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.

5.4.1.2 Mechanisms Used for Durability

Stream is one of the mechanisms used by default in CephFS. Using existing configuration settings in Ceph we can turn Stream on and off. If it is off, then the metadata servers will not save journals in the object store. For Local Persist, clients write serialized log events to a file on 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.

5.4.2 Defining Policies in Cudele

The spectrum of consistency and durability guarantees that users can construct is shown in Table 5.2. The columns are the different consistency semantics and the rows cover the spectrum of durability guarantees. For consistency: "invisible" means the system does not handle merging updates into a global namespace and it is assumed that middleware or the application manages consistency lazily; "weak" merges updates at some time in the future (*e.g.*, when the system has time, when 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

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

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 5.2 (we construct the semantics for these systems later in Section §5.6.1). 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

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

them into different semantics but we assign labels here based on the points emphasized in the papers. To compose the mechanisms users inject which mechanisms to run and which to use in parallel using a domain specific language. Although we can achieve all permutations of the different guarantees in Table 5.2, 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 §5.6.1 and §5.6.2.

The consistency and durability properties in Table 5.2 are not guaranteed until all mechanisms in the cell are complete. The compositions should be considered atomic and there are no guarantees while transitioning between policies. For example, updates are not deemed to have "global" durability until they are safely saved in the object store. If a failure occurs during Global Persist or if we inject a new policy that changes a subtree from Local Persist to Global Persist, 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.

5.4.3 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 de-

couple the path "msevilla/mydir" and would apply the policies in "policies.yml". The policies file supports the following parameters (default values are in parenthesis):

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

The "Consistency" and "Durability" parameters are 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.

5.5 Implementation

We use a programmable storage approach [49] to design Cudele; namely, we try to leverage components inside Ceph to inherit the robustness and correctness of the internal subsystems. Using this 'dirty-slate' approach, we only had to implement 4 of the 6 mechanisms from Figure 5.4 and just 1 required changes to the underlying storage system itself. In this section, we first describe a Ceph internal subsystem or component and then we show how we use it in Cudele.

Metadata Store: In CephFS, the metadata store is a data structure that represents the file system namespace. This data structure is stored in two places: in memory (*i.e.* in the collective memory of the metadata server cluster) and as objects in the object store. In the object store, directories and their inodes are stored together in objects to improve the performance of scans. The metadata store data structure is structured as a tree of directory fragments making it easier to read and traverse.

Cudele: the RPCs mechanism uses the metadata store to service requests. Using code designed for recovery, Nonvolatile Apply and Volatile Apply replay updates onto the metadata store in memory and in the object store, respectively. When the clients are ready to merge their updates back into the global namespace, they pass a binary file of metadata updates to the metadata server.

Journal: The journal is the second way that CephFS represents the file system namespace; it is a log of metadata updates that can materialize the namespace when the updates are replayed onto the metadata store. The journal is a "pile system"; writes

are fast but reads are slow because state must be reconstructed. Specifically, reads are slow because there is more state to read, it is unorganized, and many of the updates may be redundant.

Cudele: 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 [39]), and the functions for replaying events onto the internal namespace data structures.

Journal Tool: The journal tool is used for disaster recovery and lets administrators view and modify the journal. It can read the journal, export the journal as a file, erase events, and apply updates to the metadata store. To apply journal updates to the metadata store, the journal tool reads the journal from object store objects and replays the updates on the metadata store in the object store.

Cudele: 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.

Inode Cache: In CephFS, the inode cache reduces the number of RPCs between clients and metadata servers. Without contention clients can resolve metadata reads locally thus reducing the number of operations (*e.g.*, `lookup()`s). For example, if a client or metadata server is not caching the directory inode, all creates within that directory will result in a lookup and a create request. If the directory inode is cached then only the create needs to be sent. The size of the inode cache is configurable so as not to saturate the memory on the metadata server – inodes in CephFS are about 1400 bytes[3]. The inode cache has code for manipulating inode numbers, such as pre-allocating inodes to clients.

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.

Large Inodes: In CephFS, inodes already store policies, like how the file is striped across the object store or for managing subtrees for load balancing. These policies adhere to logical partitionings of metadata or data, like Ceph pools and file system namespace subtrees. To implement this, the namespace data structure has the ability to recursively apply policies to subtrees and to isolate subtrees from each other.

Cudele: uses the large inodes to store consistency and durability policies in the directory inode. This approach uses the File Type interface from the Malacology programmable store system [49] and it tells clients how to access the underlying meta-

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

data. 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.

5.6 Evaluation

Cudele lets users 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, (2) the techniques can work side-by-side in the same namespace, and (3) the combination of these techniques can help the system scale further when subtrees are coupled to the correct type of application. We scope the evaluation to one metadata server and scale the number of parallel clients. This type of analysis shows the capacity and performance of a metadata server with superior metadata protocols, which should be used to inform load balancing across a metadata cluster. Load balancing across a cluster of metadata servers with partitioning and replication can be explored with something like Mantle [51].

5.6.1 Microbenchmarks

Figure 5.5 shows the runtime of the Cudele mechanisms for a single client creating files in the same directory, normalized to the time it takes to write 100K file create updates to the client's in-memory journal (*i.e.* the Append Client Journal mechanism). Stream is an approximation of the overhead and is calculated by subtracting the runtime of the job with the journal turned off from the runtime with the journal turned on. 100K is the maximum recommended size of a directory in CephFS; preliminary

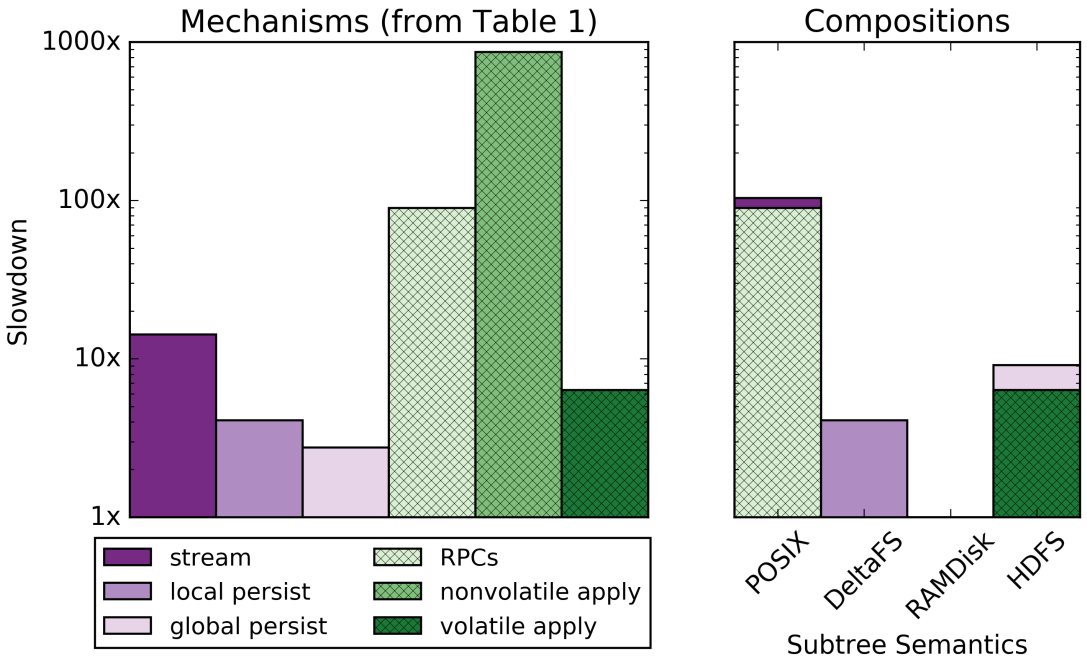


Figure 5.5: Performance of each mechanism (left) and building the consistency/durability semantics of real-world systems (right) for 100K files creates from a single client. Results are normalized to the runtime of writing events to the client's in-memory journal.

experiments with larger directory sizes show memory problems.

Poorly Scaling Data Structures: Despite doing the same amount of work, mechanisms that rely on poorly scaling data structures have large slowdowns for the larger number of creates. For example, RPCs has a 90× slowdown because it relies on the internal CephFS directory data structures. It is a well-known problem that directory data structures do not scale when creating files in the same directory [40] and any mechanism that uses these data structures will experience similar slowdowns. 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 RPCs: RPCs is 66× slower than Volatile Apply because sending individual metadata updates over the network is costly. While RPCs sends a request for every file create, Volatile Apply writes all the updates to the in-memory journal and applies them to the in-memory data structures in the metadata server. While communicating the decoupled namespace directly to the metadata server is faster, communicating through the object store (Nonvolatile Apply) is 10× slower.

Overhead of Nonvolatile Apply: The cost of Nonvolatile Apply is much larger than all the other mechanisms. That mechanism was not implemented as part of Cudele – it was a debugging and recovery tool packaged with CephFS. It works by iterating over the updates in the journal and pulling all objects that *may* be affected by the update. This means that two objects are repeatedly pulled, updated, and pushed: the object that houses the experiment directory and the object that contains the root directory (*i.e.* `/`). The cost of communicating through the object store is shown by comparing the runtime of Volatile Apply + Global Persist to Nonvolatile Apply. These two operations end up with the same final metadata state but using Nonvolatile Apply is clearly inferior.

Parallelism of the Object Store: Comparing Local and Global Persist demonstrates the bandwidth advantages of storing the journal in a distributed object store. The Global Persist performance is 1.5× faster because the object store is leveraging the

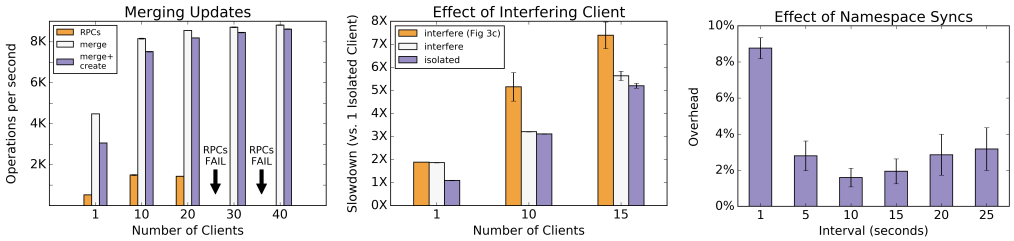
collective bandwidth of the disks in the cluster. This benefit comes from the object store itself but should be acknowledged when making decisions for the application; the bandwidth of the object store can help mitigate the overheads of globally persisting metadata updates.

Journal Size: We measure the amount of storage per journal update to be about 2.5KB. The storage footprint scales linearly with the number of metadata creates and suggests that updates for a million files would be 2.5KB $*$ 1 million files = 2.38GB. Transfer times for payloads of this size in most HPC/data center networks are reasonable.

Takeaway: measuring the mechanisms individually shows that their overheads and costs can differ *by orders of magnitude*. We also show that some mechanisms, like Nonvolatile Apply, are not worthwhile as currently implemented.

5.6.2 Use Case 1: Creates in the Same Directory

First we show how we can build application-specific subtrees by composing mechanisms and that this approach results in more a scalable global namespace. We start with clients creating files in private directories because this workload is heavily studied in HPC [62, 40, 35, 67, 51], mostly due to checkpoint-restart [7]. But the workload also appears in cloud workloads [65], where systems like Hadoop use the file abstraction to exchange work units to workers or to indicate when jobs complete [52]. A more familiar example is uncompressing an archive (*e.g.*, `tar xzf`), where the file system services a flash crowd of creates across all directories as shown in Figure 5.2.



(a) Use Case 1 **(b) Use Case 2** **(c) Use Case 3**

Figure 5.6: The performance and features of Cudele. (a) shows the cost of merging client journals at the metadata server. Shipping and merging journals of updates scales better than RPCs because there are less messages and consistency/durability code paths are bypassed. In (b), the allow/block API isolates directories from interfering clients. (c) is 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.

The workload is clients creating 100K files in private directories in the same global namespace.

The graph on the right of Figure 5.5 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 5.1. Again, the runtime is normalized to creating files in the client's in-memory journal. 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.

Takeaway: we confirm the performance benefits of other well-established research systems but, more importantly, we show that the Cudele mechanisms we propose are useful for building and evaluating different consistency/durability semantics.

To show the contention at the metadata server, in Figure 5.6a we scale the number of clients for the RPC per operation strategy and the decoupled namespace strategy. For RPCs, we increase the number of clients making requests to the metadata server and for decoupled (the merge bars), we increase the number of concurrent journal merge requests. So for the 30 clients data point, we are measuring the operations per second for 30 client journals that land on the metadata server at the same time. The "create & merge" bar adds the time it takes for clients to generate the journal of events in parallel. The experiment is run three times but the standard deviation is too small to see.

This workload scales further with decoupled namespaces because the metadata server is not overwhelmed with RPC requests. Compared to Figure 5.3c we can scale to 40 clients which is more than double the capacity of a single metadata server using the RPC strategy to maintain strong consistency. Related work has pointed to on-disk metadata format as the primary factor for improved scalability; but our results show that the weakened consistency and durability enables the scalability improvements. The on-disk metadata formats in this experiment are the same for all schemes.

The performance of decoupled namespaces is better than RPCs because we place no restrictions on the validity of metadata inserted into the journal and avoid

touching poorly scaling data structures. We can scale linearly if we weaken our consistency by never checking for the existence of files before creating files (*i.e.* only append `open()` requests to the journal). When the updates are merged by the metadata server into the in-memory metadata store they never scan growing data structures. Had we implemented the Append Client Journal mechanism to include `lookup()` commands before `open()` requests, we would have seen the poor scaling that we see with the RPCs mechanism.

Takeaway: the ability to couple well-established techniques to specific applications allows the global namespace to scale further and perform better. In our case, RPCs overwhelm the system but decoupling/shipping a journal of updates improves scalability by 2×.

5.6.3 Use Case 2: Creates with Interfering Client

Next we show how Cudele can be programmed to block interfering clients using the same problematic workload from Figure 5.3b. In this workload, clients create files in private directories and a separate client interferes by creating more files in each directory. This introduces false sharing and the metadata server revokes capabilities on directories touched by the interfering client. While HPC tries to avoid these situations with workflows [67, 68], it still happens in distributed file systems when users unintentionally access directories in a shared file system. We only scale to 15 clients because the performance variability of a nearly overloaded metadata server, in our case 18 clients, is too high.

Figure 5.6b plots the overhead of the slowest client and the error bars are the standard deviation of 3 runs. "Interfere (Fig 3c)" is the result from Figure 5.3b; "Interfere" uses the allow/block API to return `-EBUSY` to interfering clients; and "Isolated" is the baseline performance without an interfering. Each subtree has RPCs and Stream enabled. Results are normalized to the runtime of a single client creating files in a directory. Note that IndexFS has a similar behavior to "interfere" with leases, except clients block.

We draw three conclusions: (1) clients that use the API to block interfering clients get the same performance as isolated clients, (2) there is a negligible effect on performance for the extra work the metadata server does to return `-EBUSY`, and (3) merging updates from the decoupled client has a negligible effect on performance.

Takeaway: the API lets users isolate directories when applications need better and more reliable performance. Blocking updates is an effective way of controlling consistency.

5.6.4 Use Case 3: Read while Writing

The final use case shows how the API gives users fine-grained control of the consistency semantics. The use case is that users often leverage the file system to check the progress of jobs using `ls` even though this operation is notoriously heavy-weight [10, 13] . The number of files or size of the files is indicative of the progress. This practice is not too different from systems that use the file system to manage the progress of jobs; 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. In this scenario, Cudele users will not see the progress of decoupled namespaces since their updates are not globally visible. To help 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.

Figure 5.6c shows the performance degradation of a single client writing updates to a decoupled namespace and pausing to send updates to the metadata server. The error bars are the standard deviations of 3 runs. 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 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.

Takeaway: syncing namespace updates has up to a 9% overhead and can be tuned depending on the users preference but, more importantly, the API gives users fine-grain control of their consistency/durability to support current practices or experimental workflows.

5.6.5 Discussion and Future Work

Cudele and the experiments we present here prompt many avenues for future work. First is quantifying the benefits of dynamically changing the semantics of a sub-

tree from stronger to weaker guarantees (or vice versa), as described in the introduction. Second is embeddable policies, where child subtrees have specialized features but still maintain the guarantees of their parent subtrees. We already show how applications can compose guarantees and control their performance, knobs that we did not have before, but embeddable policies are a way to compose the policies themselves. For example, a RAMDisk subtree is POSIX IO-compliant but relaxes durability constraints, so it can reside under a POSIX IO subtree that is strongly consistent. These embeddable policies should be controlled and enforced by Cudele. Finally, performance prediction benefits from our cost quantification of each mechanism. Applications can use Cudele to microbenchmark their components and software, similar to what we did in Section §5.6.1. Using those results, they can predict how much slower their system will be if they adopt stronger consistency or durability. This is a form of co-design that takes a "dirty-slate" approach but building just the guarantees the application needs from existing implementations. This can also be a verification tool where performance that varies wildly from the predicted performance can be a red flag that something is wrong or that the bottleneck is not in the consistency or durability plane.

5.7 Conclusion

Relaxing consistency/durability semantics in the file system is a double-edged sword. While it performs and scales better, it alienates applications that rely on strong consistency and durability. Mounting other systems to the global namespace is convenient but wastes resources and incurs data movement. Cudele lets users assign consistency/durability guarantees to subtrees in the global namespace, which can be custom fit to the application. Using Cudele, we show how applications can co-exist and perform well in a global namespace and lay the groundwork for new systems that dynamically change consistency/durability guarantees to avoid provisioning dedicated storage clusters and moving large amounts of data.

Bibliography

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

[2] 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.

[3] 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.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design & Implementation*, OSDI '10, 2010.

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

[6] 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*, page 68. ACM, 2013.

[7] 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.

[8] J. Bent, B. Settlemyer, and G. Grider. Serving Data to the Lunatic Fringe.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Faibish, J. Bent, U. Gupta, D. Ting, and P. Tzelnic. Slides: 2 tier storage architecture.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[16] 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.

[17] H. Greenberg, J. Bent, and G. Grider. MDHIM: A Parallel Key/Value Framework for HPC. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[18] 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, November 2015.

[19] 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 *Proc. of the Fifth CIDR Conf.*

[20] 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.

[21] 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 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11.

[22] 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.

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

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

[25] 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*, pages 1309–1316. IEEE Computer Society, 2017.

[26] J. D. Kamal Hakimzadeh, Hooman Peiro Sajjad. Scaling HDFS with a Strongly

Consistent Relational Model for Metadata. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS '14.

[27] 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.

[28] 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.

[29] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, 2008.

[30] 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.

[31] K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *Commununications ACM*, 53(3):42–49, Mar. 2010.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.*

[38] A. Povzner, D. Sawyer, and S. Brandt. Horizon: efficient deadline-driven disk I/O management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10.

[39] K. Ren and G. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local

File System. In *Proceedings of the USENIX Annual Technical Conference*, ATC '13, 2013.

[40] 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.

[41] 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.

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

[43] 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.

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

[45] 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.

[46] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. 2.

[47] 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.

[48] 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 International Workshop on Data Intensive Scalable Computing at SuperComputing '13*, DISCS'13, 2013.

[49] 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, Belgrade, Serbia.

[50] 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 Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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 SIGMOD International Conference on Management of Data*, SIGMOD '10.

[56] N. Tran and D. A. Reed. ARIMA Time Series Modeling and Forecasting for Adaptive I/O Prefetching.

[57] 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.

[58] 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 Sys-

tem. In *Proceedings of the Workshop on Parallel Data Storage Workshop*, PDSW '13.

[59] 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*.

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

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.