Mantle: A Programmable Metadata Load Balancer for the Ceph File System

Michael Sevilla[1], Noah Watkins[1], Carlos Maltzahn[1], Ike Nassi[1], Scott Brandt[1], Sage Weil[2], Greg Farnum[2], Sam Fineberg[3]

[1]UC Santa Cruz, {msevilla, jayhawk, carlosm, inassi, scott}@soe.ucsc.edu

[2]Red Hat, {sage, gfarnum}@redhat.com

[3]HP Storage, fineberg@hp.com

ABSTRACT

Migrating resources is a useful tool for balancing load in a distributed system, but it is difficult to determine when to move resources, where to move resources, and how much of them to move. We look at resource migration for file system metadata and show how CephFS's dynamic subtree partitioning approach can exploit varying degrees of locality and balance because it can partition the namespace into variable sized units. Unfortunately, the current metadata balancer is complicated and difficult to control because it struggles to address many of the general resource migration challenges inherent to the metadata management problem. To help decouple policy from mechanism, we introduce a programmable storage system that lets the designer inject custom balancing logic. We show the flexibility of this approach by replicating the strategy of a state-of-the-art metadata balancer and conclude by comparing this strategy to other custom balancers on the same system. Our solution increases performance by up to 9% for a mixed workload and demonstrates that distributing metadata load evenly can degrade performance by 40%.

1. INTRODUCTION

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 [18]. 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 [1, 2, 3, 23]. Many distributed file systems decouple metadata from data access so that data and metadata I/O can scale independently [3, 7, 9, 23, 25, 26]. 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.

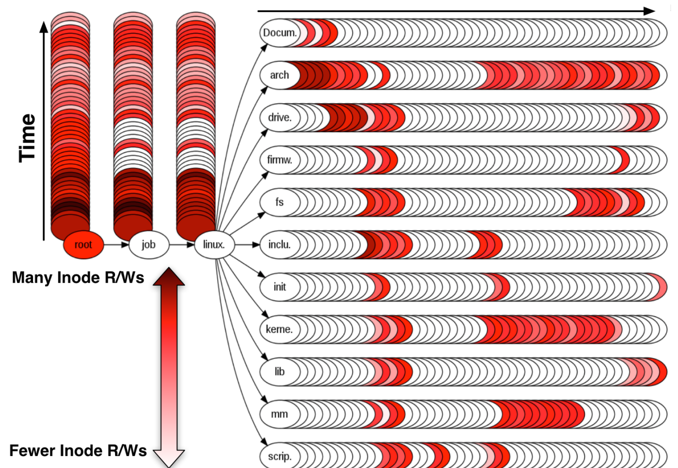


Figure 1: Metadata hotspots, represented by different shades of red, have spatial and temporal locality when compiling the Linux kernel. The hotspots are calculated using the number of inode reads/writes and smoothed with an exponential decay.

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) [12]. 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 [22]. The elegance and simplicity of the solutions stem from a thorough understanding of the workloads (*e.g.*, temperature zones at Facebook [14]) 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 servers (MDS) nodes [15, 23, 24, 20, 26]. 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 [15, 17, 27, 9, 28], inode path/permission caching [5, 11,

26], locality-aware/inter-object transactions [20, 28, 16, 17] and efficient lookup tables [5, 28]. Despite having mechanisms for migrating metadata, like locking [20, 19], zero copying and two-phase commits [20], and directory partitioning [26, 15, 17, 23], these systems fail to exploit locality.

File system workloads have a great deal of locality because the namespace has semantic meaning; data stored in directories is related and is usually accessed together. Figure 1 shows the metadata locality when compiling the Linux src 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 [23] (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, a system built on CephFS that exposes these factors by separating migration policies from the mechanisms. Mantle[1] accepts injectable metadata migration code and helps us make the following contributions:

- a comparison of balancing for locality with subtree partitioning and balancing for distribution with hashing
- 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 find that the cost of migration can sometimes outweigh the benefits of parallelism, resulting in a 40% degradation in performance. We also find that finding a good balance can improve performance but searching for that balance aggressively can drop the speedup by 4% with an increased standard deviation in runtime. We also show that balancers that distribute metadata prepare the MDS cluster better for flash crowds, resulting in 9% performance improvement. The most important insight that Mantle gives us is that the balancing techniques are heavily dependent on the workload, but can be accurately predicted given the proper metrics.

2. RESOURCE MANAGEMENT WITH DYNAMIC SUBTREE PARTITIONING

We use Ceph [23] 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.

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

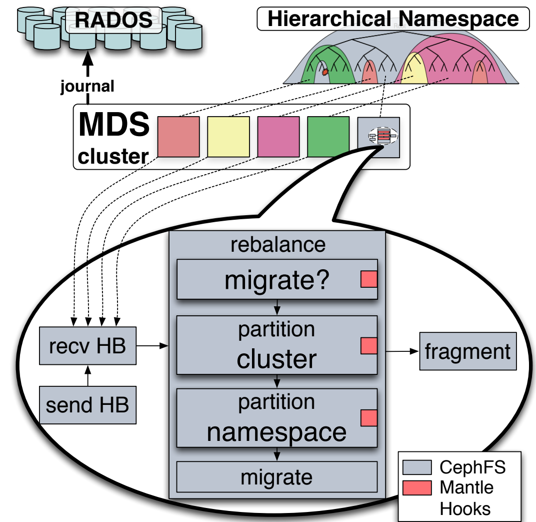


Figure 2: The MDS cluster journals to RADOS and exposes a namespace to clients. Each MDS makes decisions by exchanging heartbeats and partitioning the cluster/namespace. Mantle adds injectable code hooks for custom balancing logic.

CephFS is the gateway to file-based storage for legacy applications. It decouples metadata and data access, so data IO is done directly with RADOS while all metadata operations go to a separate metadata cluster. The MDS cluster is connected to RADOS so it can periodically flush its state and to the clients so it can service metadata operations. The hierarchical namespace is kept in the collective memory of the MDS cluster and acts as a large distributed cache. The MDS nodes use dynamic subtree partitioning [24] to carve up the namespace into subtrees and assign them to MDS nodes, as shown in Figure 2.

MDS nodes can dynamically move *different sized pieces* of their workload to other MDS nodes. Each MDS has its own metadata balancer that makes independent decisions, using the flow in Figure 2. Every 10 second, every 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 migrate, it partitions the namespace and cluster and sends inodes ("migrate") to the other MDS nodes. In the following sections, we focus on the mechanisms that the balancer uses to dynamically migrate load in the "migrate", "partition namespace", and "partition cluster" modules of Figure 2.

2.1 Mechanisms for Migrating Inodes

CephFS has mechanisms for migrating metadata, ensuring consistency, enforcing synchronization, and mediating access. Inodes are embedded in directories so that related

inodes are fetched on a `readdir` and can be migrated with the directory itself. The migrations are performed as a two-phase commit, where the importer journals metadata, the exporter logs the event, and the importer journals the event. Other infrastructure includes:

- "soft state": in addition to its metadata, each MDS is only aware of metadata that borders its subtrees (*i.e.* parent directories know which MDS remote children reside on)
- replication: cache coherence allow reads and path traversals on replicated metadata
- locking: updates are forwarded to the only MDS with metadata write permissions, called the authority, for serialization, journaling, and consistency; each metadata field is protected by a distributed state machine
- "traffic control": MDS nodes redirects clients and clients cache the namespace

2.2 Partitioning the Namespace

Each MDS's balancer carves up the namespace using *directory fragments* (added since [24, 23]) and *subtrees*. Directory fragments (*i.e.* dirfrags) are partitions of a directory; when the directory grows to a certain size, the balancer fragments the directory into these smaller dirfrags. This directory partitioning mechanism is equivalent to the GIGA+ mechanism, although the policies for moving the dirfrags can differ. Subtrees are collections of nested directories and files. 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 cluster by identifying popular subtrees or dirfrags using metadata counters. The metadata counters are stored in the directories and are updated whenever a namespace operation hits that directory or any of its children. With this scheme, the parent directories should reflect the popularity of all 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 §3.3). When searching its namespace for inodes to migrate, the balancer uses this metadata load calculation to quantify the popularity on a subtree or dirfrag. Dynamic subtree partitioning allows the balancer to move different-sized units of the namespace, but this means that it needs to know the load levels of each of the other MDS nodes.

2.3 Partitioning the Cluster

Each balancer exposes metrics for quantifying the local metadata load and resource utilization on each MDS, such as CPU load and memory utilization. There are also MDS-specific metrics, such as the metadata load on the authority subtree, the metadata load on all the other subtees, the request rate/latency, and the queue length, that can be used to make migration decisions. The balancer calculates an *MDS load* for all MDS nodes using a weighted sum (again, the policy is explained in Section §3.3).

With MDS loads the balancer is able to compare how much work each MDS is doing. With this global view, the balancer can partition the cluster into exporters (MDS nodes that want to shed load) and importers (MDS nodes that have the capacity for more load). 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 exhange is the load calculation itself, as an inaccurate view of another MDS or the cluster state can lead to erroneous decisions.

2.4 Metadata Locality vs. Distribution

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. This can improve performance by reducing the amount of migrating inodes and the number of requests in the metadata cluster.

Figure 3a shows how the overheads of distribution can hurt performance for a single client compiling code, in this case the Linux src, in a CephFS directory. 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 "good balance" bar is when hot metadata is correctly distributed, and the "bad balance" 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%.

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, like path prefix traversals and permission checking. The worse the distribution and the higher the fragmentation, the higher the number of forwards. Figure 3b shows a high number of path traversals that end in forward requests (y axis) when metadata is distributed poorly. In addition to the extra metadata requests, the overhead of distributing metadata is related to memory pressure, the cost of moving inodes, and the number of migrations.

3. MULTI-MDS CHALLENGES

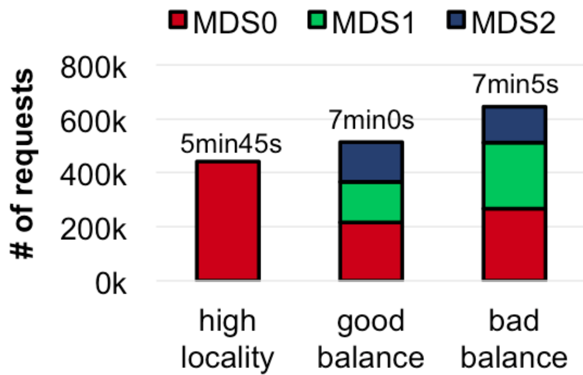
Dynamic subtree partitioning achieves varying degrees of locality and distribution by changing the way it carves up the namespace and partitions the cluster. To alleviate load quickly, dynamic subtree partitioning can move different sized resources (inodes) to computation engines with variable capacities (MDS nodes), but this flexibility has a cost.

3.1 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

[2]To get high locality, all metadata is kept on one MDS. To get different degrees of balance, we change the setup: "bad balance" is untarring and compiling with 3 MDS nodes and "good balance" 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.

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



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

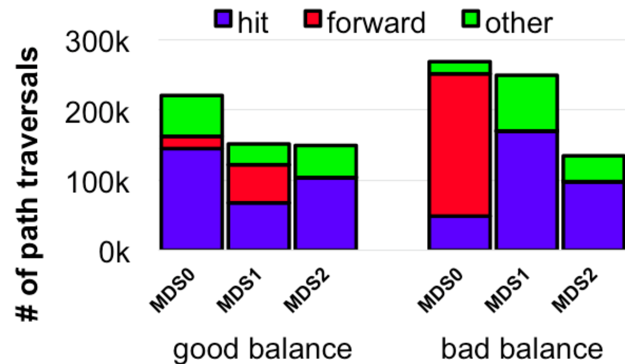


Figure 3: Distributing metadata to multiple MDS nodes hurts performance (left) by reducing locality and increasing the number of "forwarded" requests (right). Reproducibility challenges (see Section §3) make it difficult to provide error bars.

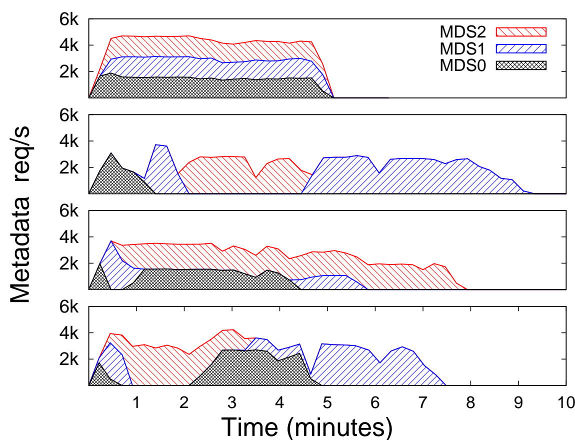


Figure 4: The same create-intensive workload has different throughput (y axis; curves are stacked) because of how CephFS maintains state and sets policies.

impact performance. The model should consider what can be moved and how migration units can be divided or combined. It should also consider how splitting different or related objects affects performance and behavior. Second, the model must quantify the state of the system using available metrics. Third, the model must tie the metrics to the global performance and behavior of the system. It must consider how over-utilized resources negatively affect performance and how system events can indicate that the system is performing optimally. With such a model, the balancer can decide which metrics to optimize for.

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

3.2 Maintaining Global & Local State

To make fast decisions, CephFS measures, collects, and communicates small amounts of state. Its design decisions 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.

3.3 Setting Policies for Migration Decisions

In complicated systems there are two approaches for setting policies to guide decisions: expose the policies as tunable parameters or tie policies to mechanisms. Tunable parameters, or tunables, are configuration values that let the system administrator adjust the system for a given workload. Unfortunately, these tunable parameters are usually so specific to the system that only an expert can properly "tune" the system. For example, Hadoop has 210 tunables to configure

Policy	Hard-coded implementation
metaload	`= inode reads + 2*(inode writes)`
	`+ read dirs + 2*fetches + 4*stores`
MDSload	`= 0.8*(metadataload on auth)`
	`+ 0.2*(metadata load 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 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 MDS nodes target loads; howmuch accuracy is the strategy for sending dirfrags to reach a target load.

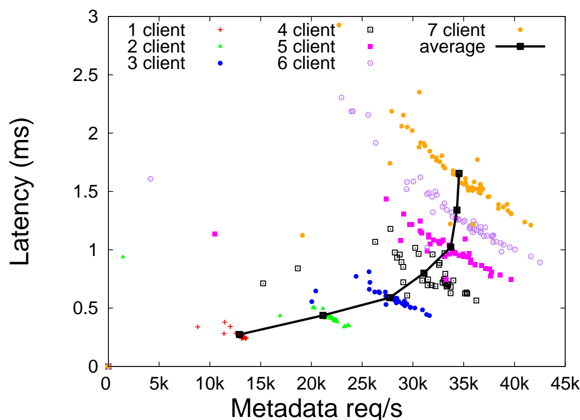


Figure 5: For the create heavy workload, the throughput (x axis) stops improving and the latency (y axis) continues to increase with 5, 6, or 7 clients. The standard deviation also increases for latency (3×) and throughput (2.3×).

a MapReduce application[3]. 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 1, shape decisions using two techniques: scalarization of logical/physical metrics and hard-coding the logic for the balancing decisions.

[3]Hadoop sample XML configuration file: hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml

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 1.

The other technique CephFS uses in its policies is to compile the decision logic into the system. The balancer uses one approach for deciding when and where to move inodes; it migrates load when it thinks that it has more load than the other MDS nodes ("when" row of Table 1) and it tries to migrate enough of its load to make the load even across the MDS cluster ("where" row Table 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 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 currently starts at its root subtrees and sends the largest subtree or dirfrag until reaching this target load ("how-much accuracy" row of Table 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: $\frac{\sum \text{dirfrag_load}}{\#\ \text{MDSs}} = 55.6$. 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 directory fragments, 15.7 + 14.6 + 14.6, instead of half the 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 works for large directories with millions of entries, it struggles with simpler and smaller namespaces because of the noise in the load measurements on each dirfrag.

3.4 Discussion

These 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

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

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

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

Table 2: The Mantle environment.

benchmark CephFS [24], they do not work for others (as demonstrated in Figure 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 [13] 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.

4. MANTLE IMPLEMENTATION

Mantle builds on the implementations and data structures in the CephFS balancer, as shown in Figure 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.

4.1 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 [8]. 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" (see Section §4.2 and Figure 6).

Mantle provides a general environment with global variables and functions, shown on the left side of Figure 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 2. The labeled arrows between the phases in Figure 6 are the inputs and outputs to the phases; inputs can be used and outputs must be filled by the end of the phase.

4.2 The Mantle API

Figure 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 in partitioning the cluster, and the "howmuch" decision for deciding the accuracy of sending dirfrags is used when partitioning the namespace. 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. 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 set `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
```

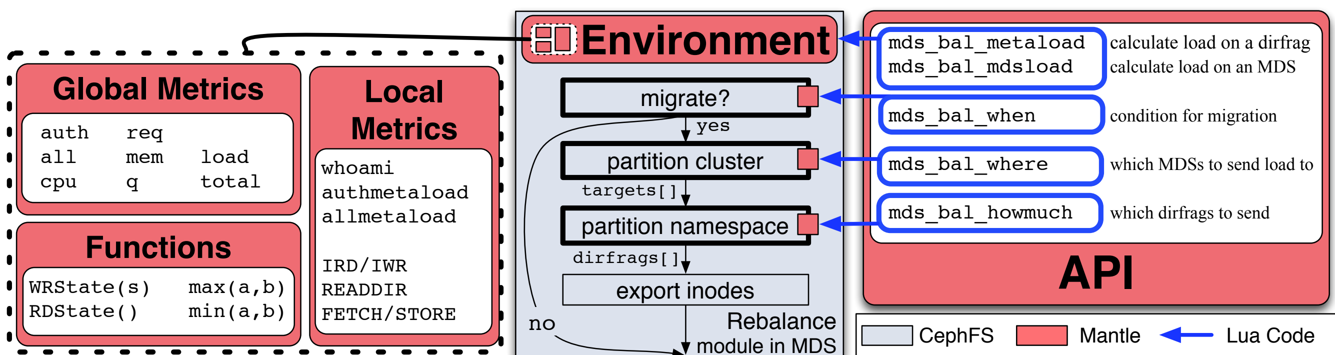


Figure 6: Designers set policies using the Mantle API. The injectable code uses the metrics/functions in the environment.

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, with the values they would calculate had they been running for the problematic dirfrag loads in Section §3.3 (12.7, 13.3, 13.3, 14.6, 15.7, 13.5, 13.7, 14.6). Recall that MDS0 assigns MDS1 the target load of 55.6:

1. `big_first`: biggest dirfrags until reaching target

   $15.5 + 14.6 + 14.6 + 13.7$ MDS0=66.5, MDS1= 44.7
2. `small_first`: smallest dirfrags until reaching target

   $12.7 + 13.3 + 13.3 + 13.5 + 13.7$ MDS0=44.7, MDS1= 66.5
3. `big_small`: alternate sending big and small dirfrags

   $15.5 + 12.7 + 14.6 + 13.3$ MDS0=55.1, MDS1= 56.1
4. `half`: send the first half of the dirfrags

   $12.7 + 13.3 + 13.3 + 14.6$ MDS0=57.3, MDS1= 53.9

For this scenario, Mantle would choose the `big_small` dirfrag selector because the distance between the target load 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.

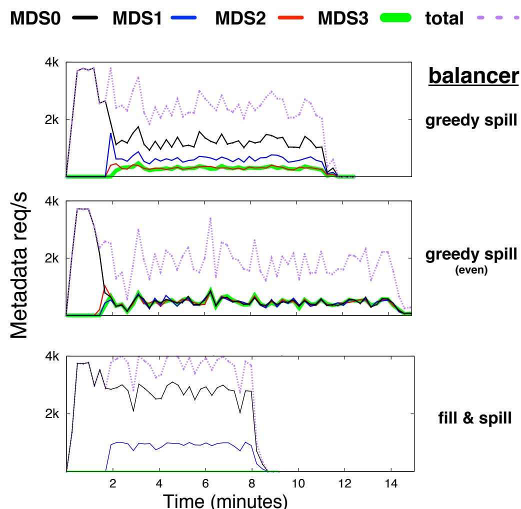


Figure 7: With clients creating files in the same directory, using only a subset of the MDS nodes and spilling load unevenly has the highest throughput (9% speedup). To show the load on each MDS, these throughput curves are not stacked.

5. 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. For the following experiments 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. We evaluate Mantle along three dimensions: API flexibility, performance, and stability.

5.1 API Flexibility

To examine how general the Mantle API is, we implement three balancers: a greedy spill balancer, a fill and spill balancer, and an adaptable balancer.

5.1.1 *Greedy Spill Balancer*

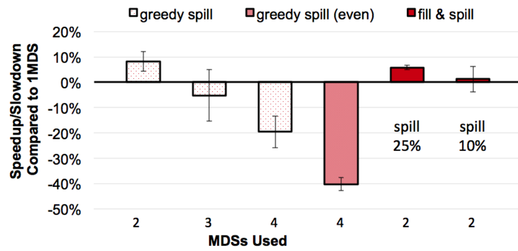


Figure 8: The per-client speedup or slowdown shows whether distributing metadata is worthwhile. Spilling load evenly to 3 or 4 MDS nodes degrades performance but spilling unevenly approach improves performance.

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 strategy of GIGA+ [15, 17]. 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 $8 = 2^3$ dirfrags) and the balancer migrates half of its load to an "underutilized" neighbor.

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

Listing 1: Greedy Spill Balancer using the Mantle environment (listed in Table 2). Note that all subsequent balancers use the same metadata and MDS loads.

The metadata load for the subtrees in the namespace is calculated using just the number of inode writes; we focus on create-intensive workloads, so inode 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 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 balancing shown in the second graph of Figure 7, the balancer is modified according to Listing 2 to partition the cluster when selecting the target MDS.

```
-- Balancer 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
-- Balancer where policy
targets[t]=MDSs[whoami]["load"]/2
```

Listing 2: Greedy Spill Evenly Balancer.

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

The balancer with the most speedup is the 2 MDS configuration, as shown in Figure 8. This agrees with the assessment of the capacity of a single MDS in Section §2.3; at 4 clients, a single MDS is only slightly overloaded, so splitting load to two MDS nodes only improves the performance by 10%. Spilling unevenly to 3 and 4 MDS nodes degrades performance by 5% and 20% because the cost of synchronizing across multiple MDS nodes penalizes the balancer enough to make migration inefficient. Spilling evenly with 4 MDSs degrades performance up to 40% but has the lowest standard deviation because the MDS nodes are underutilized.

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

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.

5.1.2 Fill and Spill Balancer

The fill and spill balancer, shown in Figure 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

[4]The cause of the latency could be from a scatter-gather process 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.

overloaded is a crucial policy for this balancer. In our implementation, we use the MDS's instantaneous CPU utilization, 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 §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.

```
-- Balancer 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
-- Balancer 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; early runs 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% and only uses a subset of the MDS nodes, as shown in Figure 8. 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.

5.1.3 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

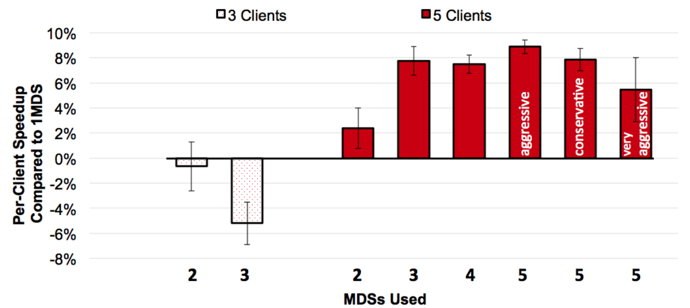


Figure 9: For the compile workload, 3 clients do not overload the MDS nodes so distribution is only a penalty. The speedup for distributing metadata with 5 clients suggests that an MDS with 3 clients is slightly overloaded.

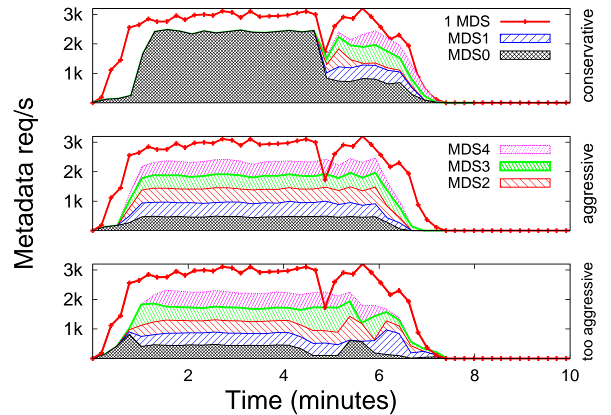


Figure 10: With 5 clients compiling the Linux src code in separate directories, distributing metadata load (stacked throughput curves) early helps the cluster handle a flash crowd at the end of the job. Throughput drops when using 1 MDS (red curve) because the clients shift to linking, which overloads 1 MDS with `readdirs`.

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 uses a wide range of dirfrag selectors.

The overall performance for 5 clients compiling the Linux src code in separate directories is shown in Figure 9. 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. With 3 clients, the MDS nodes are underutilized so the overhead of distribution is only a penalty. With 5 clients, 1 and 2 MDS setups are overloaded because some MDS nodes end up servicing the entire load of 3 clients. The 5 MDS setup that aggressively sheds load has the fastest runtime because each client gets a dedicated MDS, but the performance improvement is small indicating that each MDS could handle up to two clients efficiently.

```
-- Balancer 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
-- Balancer howmuch policy
{"half","small","big","big_small"}
```

Listing 4: Fill and Spill Balancer.

The performance profile for the 5 MDS setups in Figure 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, there are almost 30,000 forwarded requests (60× as many forwards as the middle balancer), and the standard deviation for the runtime is much higher.

Figure 10 also shows 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.

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.

6. 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 [9] and SkyFS [26] hash the filename to locate the authority for metadata. CalvinFS [21] 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. [11] presented dynamic formulas to account for a forgetting factor, access information, and the number of MDS nodes for load balancing in elastic clusters. [26] used a master-slave architecture to detect low resource usage and migrated metadata using a consistent-hashing-based load-balancer. GPFS [19] 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+ [15] 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 [15], 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 [11, 26, 28]. For example, Lazy Hybrid [5] 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.

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. This is a form of hashing, where indices are either managed by a centralized server or the clients. IBRIX [10] distributes inode ranges round robin to all servers. HBA [28] distributes metadata randomly to each server and uses bloom filters to speedup the table lookups. Ursa Minor [20] and Farsite [6] 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 its ability to adapt to hotspots/flash crowds and maintain balancer as data is added. Some systems, like Panasas [25], allow certain degrees of dynamicity by supporting the addition of new subtrees at runtime, but adapting to the current workload is ignored.

7. CONCLUSION

The flexibility of dynamic subtree partitioning introduces significant complexity in the balancing decision making 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 balancing policies from the mechanisms for migration. 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 affects 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.

References

[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 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 125–132, 2012.

[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 6th Workshop on Parallel Data Storage*, PDSW'11, 2011.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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, pages 13–13, 2011.

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

[17] 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 20th ACM/IEEE Conference on Supercomputing*, SC '14, 2014.

[18] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 4–4, 2000.

[19] 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.

[20] 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.

[21] 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)*, pages 1–14, Santa Clara, CA, Feb. 2015. USENIX Association.

[22] 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 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[23] 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 7th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'06, 2006.

[24] 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 17th ACM/IEEE Conference on Supercomputing*, SC'04, 2004.

[25] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, 2008.

[26] 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.

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

[28] 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.