Lunule: An Agile and Judicious Metadata Load Balancer for CephFS

Yiduo Wang
University of Science and Technology of China
Hefei, Anhui, China
duo@mail.ustc.edu.cn

Cheng Li
Anhui Province Key Laboratory of High Performance Computing, USTC
Hefei, Anhui, China
chengli7@ustc.edu.cn

Xinyang Shao
University of Science and Technology of China
Hefei, Anhui, China
sxy799@mail.ustc.edu.cn

Youxu Chen
University of Science and Technology of China
Hefei, Anhui, China
cyx1227@mail.ustc.edu.cn

Feng Yan
University of Nevada, Reno
Reno, Nevada, USA
fyan@unr.edu

Yinlong Xu
Anhui Province Key Laboratory of High Performance Computing, USTC
Hefei, Anhui, China
ylxu@ustc.edu.cn

ABSTRACT

For a decade, the Ceph distributed file system (CephFS) has been widely used to serve the ever-growing big data in many key fields ranging from Internet services to AI computing. To scale out the massive metadata access, CephFS adopts a *dynamic subtree partitioning* method, splitting the hierarchical namespace and distributing *subtrees* across multiple metadata servers. However, this method suffers from a severe imbalance problem that may result in poor performance due to its inaccurate imbalance prediction, ignorance of workload characteristics, and unnecessary/invalid migration activities. To eliminate these inefficiencies, we propose Lunule, a novel CephFS metadata load balancer, which employs an *imbalance factor model* for accurately determining *when* to trigger re-balance and tolerate benign imbalanced situations. Lunule further adopts a *workload-aware migration planner* to appropriately select subtree migration candidates. Compared to baselines, Lunule achieves better load balance, increases the metadata throughput by up to 315.8%, and shortens the tail job completion time by up to 64.6% for five real-world workloads and their mixture, respectively. Besides, Lunule is capable of handling the metadata cluster expansion and the client workload growth, and scales linearly on a cluster of 16 MDSs.

ACM Reference Format:
Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. 2021. Lunule: An Agile and Judicious Metadata Load Balancer for CephFS. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476196

1 INTRODUCTION

CephFS is a widely-adopted, open-source, POSIX-compliant distributed file system (CephFS) [17]. It aims for high performance, large data storage, and maximum compatibility with a variety of applications, including shared home directories [32], cloud applications [6], HPC workloads [37, 40], and scientific or AI computing [8, 36]. Recently, CephFS has also become a research hotspot [4, 14, 35, 42–44].

In CephFS, metadata, mainly referring to the namespace hierarchy, is managed separately from data. This decoupling enables the independent scaling of both metadata and data. Within such architecture, metadata must be first obtained prior to the actual data access. Recent studies reveal that many file system workloads are metadata-intensive, i.e., more than 50% (up to 92.8% in our study) of file system operations are concentrating on metadata [1, 5, 21]. Furthermore, the vast majority of files are small [49], while the data path overhead has been greatly improved by the emerging fast storage devices such as NVMe SSDs [13]. All these trends make the metadata performance of critical importance.

For a decade, CephFS followed the conventional wisdom of managing a cluster of metadata servers (MDSs). This is a preferred choice for most modern distributed file systems since it allows them to balance load across MDSs for caching as much metadata in memory as possible and parallelizing metadata request processing [22, 24, 41, 43]. However, scaling the performance of the MDS cluster is more challenging, compared to the data cluster, mainly because metadata contains file system structural information and exhibits a higher degree of interdependence [45]. To improve metadata scalability, CephFS adopts *dynamic subtree partitioning*, where the hierarchical namespace is split into smaller subtrees, and those subtrees will be periodically migrated among MDSs according to the workload intensity level of each MDS [43, 45].

Additionally, Lustre[24], PVFSv2[19] and SkyFS[47] adopt a hash-based mapping, and pair files' metadata and the authorized MDSs by computing the hash value of the corresponding filename or pathname. Despite of the even metadata distribution among MDSs, it destroys spatial locality, which is considered to be important in real world [14, 35]. However, it is challenging for its static partitioning to adapt to dynamics, e.g., the MDS cluster expansion.

Compared to the hash-based mapping, the dynamic subtree partitioning mechanism in CephFS offers better flexibility, and enables to strike a balance between locality and load distribution.

Though substantial engineering efforts have been made, there still exists a severe metadata load imbalance problem within the CephFS dynamic subtree management, which leads to poor metadata performance and resource wasting [27, 35]. In Section 2.2, we assess the performance of the CephFS' built-in metadata load balancer with four representative workloads in different fields, ranging from the traditional CephFS use cases to AI computing. Our main finding is that the metadata accesses were not evenly distributed across multiple MDSs, and in the worst case, the volume of metadata requests processed by the most loaded MDS is 220× higher than that of the least loaded one, even with frequent and active migration activities in the background.

The unexpected performance is primarily contributed by 1) inaccurate load model with benign imbalance oblivious: the load monitoring module fails to accurately model each MDS' load and the cluster load, while not tolerating benign imbalance, where the load imbalance level stays at the safe interval; and 2) invalid subtree migration candidate selection: the migration decision module inappropriately chooses subtree migration candidates, with no consideration of the access patterns of and the future load variance on those subtrees. These root causes make it hard to trigger metadata migration at the right time and choose the appropriate amount of subtree candidates for being migrated, which is either too aggressive or not adapting to the workload-specific demands.

Research on improving the performance of the dynamic subtree partitioning metadata management in CephFS has not received enough attention. Recently, Mantle [35] decouples the load stats collection and migration decision making steps from the rest of CephFS' metadata management and offers programmable APIs to allow users to specify functions to determine when and how much to migrate. However, the APIs are limited and do not cover the important subtree selection feature. More importantly, deriving an accurate load model and reasonable heuristics for metadata migration and load balancing remains a challenge.

To overcome the above challenges, we propose Lunule, a novel metadata load balancer based on *dynamic subtree partitioning* for CephFS. First, to make re-balance when needed, Lunule is driven by an *analytical* model, which accurately captures the whole MDS cluster's workload intensity level. Rather than using average load statistics, this model uses the *Coefficient of Variation* to compute real-time imbalance factor of the MDS cluster to minimize the negative impacts of noises on the migration decision. Additionally, we introduce an *urgency* parameter to quantify whether the imbalanced situations are safe or harmful for further reducing unnecessary migrations. Based on the model, Lunule determines *exporter* and *importer* MDSs. *Exporter* MDSs have stressed metadata loads and need to migrate some loads to other peers. *Importer* MDSs have spare capacities to accommodate incoming loads and compute *how much* data should be migrated between two different MDSs roles by taking into account of the future load variance on MDSs.

Next, Lunule chooses which set of subtrees on each exporter MDS to move to fulfill the above migration decisions. The subtree selection also plays a key role in achieving good load balance, since invalid subtree migrations would not help smooth the

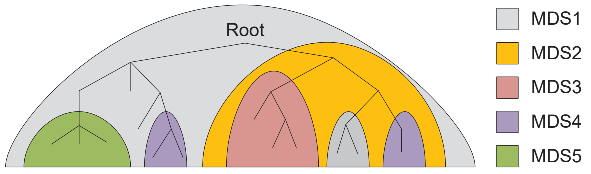


Figure 1: An exemplified metadata distribution using subtree partitioning.

skewed MDS cluster load. Therefore, it is critical to accurately predict the future visiting frequencies of different subtrees, assigned as their *migration indices*, and choose the migration subtree candidates with higher values. To cope with various workloads, we propose a unified formula to estimate the effects of the temporal and spatial locality of past visiting activities of subtrees on their future loads. For the temporal locality impacts, we consider measuring the recurrence of metadata visits in the most recent time interval, rather than relying on a simple accumulated popularity counter used in the current CephFS. With regard to the spatial locality impacts, neglected by the current solution, we take into account the even distribution of metadata accesses of a target subtree, and also consider the access correlations between sibling subtrees.

Finally, we incorporate all design choices into CephFS and build Lunule by extending CephFS's metadata service, with the following major technical contributions:

- We conduct a comprehensive study of discovering the load imbalanced phenomenon in the CephFS dynamic subtree partitioning mechanism and identifying the root causes of their inefficiencies for migrating loads in the MDS cluster.
- We invent a new metadata load balancer Lunule, which accurately models both the MDS load distribution and the urgency of imbalance, and dynamically adapts migration plans to different workloads, in terms of the migration amount determination and subtree selection.
- We conduct an in-depth evaluation of Lunule on five real-world workloads and their mixture, and the results show that Lunule achieves significantly better metadata performance than CephFS-Vanilla and GreedySpill, e.g., up to 315.77% increases in the aggregated metadata throughput, up to 57.14% reduction in the tail job completion time, and up to 93.42% better load balance. With data access enabled, Lunule introduces a 1.20-2.81× speedup of the end-to-end file system throughput, compared to the two baselines. Finally, Lunule scales linearly on a cluster of 16 MDSs.

2 BACKGROUND AND MOTIVATION

2.1 Dynamic subtree partitioning

Overview: With the metadata/data decoupling design, the MDS cluster is responsible for managing the file system namespace (a.k.a hierarchical directory structure) and facilitating client access to file data. To scale out the metadata performance, the MDS nodes adopt dynamic subtree partitioning to split the namespace and distribute its portions across the MDS cluster. In more detail, the load balancer on each MDS carves up the namespace into *subtrees* and *directory fragments* (or short, *dirfrags*). Subtrees are collections of nested directories and files, while dirfrags are partitions of a single directory (whose size exceeds a particular size). These metadata

Table 1: The description of five evaluated workloads from different fields.

Workload	Scenario	Meta_op ratio	Characteristics
CNN image pre-process (CNN)	Machine Learning	78.1%	This workload corresponds to the data preprocessing phase of the common CNN model training tasks. According to the MXNet DNN system, each client needs to scan the whole ImageNet dataset (ILSVRC2012 [34]) to convert the dataset's hierarchical namespace into a metadata list for data shuffling across training epochs, and to create a large sequential record file, consumed by the model training. ImageNet contains 1.28 million images spanning across 1000 directories with an average size of 114.3 KB.
NLP training (NLP)	Machine Learning	92.8%	In this workload, each client trains the THUTC model[38], a Chinese text classifier, by consuming user-defined text classification corpus. The entire dataset consists of 836k news files placed in 14 folders with an average size of 2.8 kilobytes.
Web trace replay (Web)	Traditional	57.2%	We replay a web access trace (used in many other relevant works [49]), which was gathered from a department web server (08/23/2013-03/18/2015) at the Florida State University, and in the Apache access log format[7]. The whole trace contains 302K files and includes 8.06 million HTTP requests, each client gets files in order.
Filebench Zipfian read (Zipf)	Traditional	50.0%	We use Filebench [39] to design a workload for simulating access patterns with strong temporal locality. In this workload, concurrent clients exclusively access to a non-shared directory with 10000 files and randomly read files according to a Zipfian distribution, i.e., 80% of requests are touching 20% of files.
MDtest create (MD)	Traditional	100.0%	We use MDtest [26], a widely used tool for evaluating the metadata performance of POSIX-compliant file systems, to simulate a write-intensive workload. Each MDtest instance operates on a non-shared empty directory and continues to create 100000 empty files into that directory. Note that a similar workload has been used in many related metadata works [22, 30, 33].

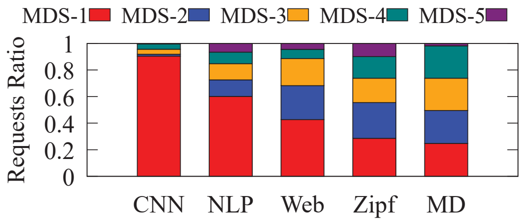
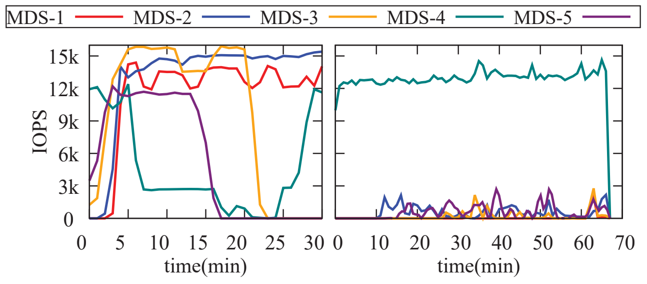


Figure 2: The metadata request distribution of the five-MDS cluster w.r.t different workloads.



(a) Filebench Zipfian (b) CNN pre-porcess

Figure 3: The per-MDS throughput numbers (measured as the number of metadata requests per second) for two selective workloads.

fragments will be migrated between MDSs when workloads vary for achieving the right load balance.

Figure 1 illustrates an exemplified metadata distribution using the dynamic subtree partitioning method. In this example, the hierarchy tree of the target file system is distributed and managed by five MDS servers, e.g., MDS-1 is in charge of the grey partition, while the green part is taken care of by MDS-5. Metadata requests may need to relay from one MDS to another when those requests fall outside the contacted MDS's subtree boundary. For instance, accessing metadata residing in the green subtree on MDS-5 should visit MDS-1 first.

Load balancing: The core feature of *dynamic subtree partitioning* is to dynamically and intelligently re-delegate arbitrary subtrees to different MDS servers based on their usage patterns and the current cluster load. This load balancing procedure can be summarized into four major steps running in a loop as follows. In the first phase, MDS nodes are involved in collecting metadata load statistics to determine if the MDS cluster experiences imbalance and whether the re-balanced should be triggered. If so, then the second step partitions the cluster into *exporters* and *importers*, and figures out the amount of its local load to be shipped from an *exporter* to one of its paired *importers*. Following that, at the third step, each *exporter* MDS selects a set of subtrees or dirfrags and places them into an *export task* queue to fulfill the planned migration load, possibly performing further fine-grained partitions when needed. Finally, the actual migration of those partitions from exporters to importers is performed via a standard two-phase commit protocol.

2.2 Load imbalance phenomenon

To reveal the importance of metadata load balance and the challenges of achieving it, we test CephFS deployed with a five-node MDS cluster by running five representative metadata-heavy workloads, as listed in Table 1. The experimental setup can be found in

Section 4.2. These workloads are from different fields ranging from web server accesses to machine learning. Among these workloads, their metadata operations account for 50.0-100.0% of the whole file system operation spaces. More importantly, the metadata access is crucial to the workload performance, e.g., the ratio of the time spent in the metadata service over that of the entire data path already exceeds half for CNN and NLP workloads.

First, we measure the total number of requests handled by each MDS from the beginning to the completion of each workload. As presented in Figure 2, the metadata operation imbalanced phenomenon exists in all workloads but with different severity. For the most balanced Filebench-Zipfian workload, the accumulated loads on MDS-1 and MDS-2 are almost identical, and they together handled 55.4% of all metadata accesses. Compared to the two busy MDSs, the least loaded MDS-5 only processed 9.9% of metadata accesses. Among the five workloads, CNN exhibits the highest metadata load imbalance. For instance, its MDS-1 received 90.3% of metadata accesses, which is 22 to 220 times higher than the remaining four MDSs. The NLP, Web, and MD workloads sit between the two extreme cases.

In addition to understanding the imbalance in the portion of metadata requests handled by each MDS for a time interval of tens of minutes in Figure 2, we summarize in Figure 3 the instantaneous metadata throughput of each MDS as time evolves, which tells how the metadata load is migrated across the MDS cluster. In the interest of space, we choose two workloads, namely, Zipfian and CNN.

First, for Filebench-Zipfian read, as shown in Figure 3(a), all loads are concentrated on MDS-4 initially, while the other four MDSs start to involve in the metadata request processing. Between 5 and 10 minutes, the load of MDS-4 sharply goes down to only 9% of its maximal capacity, while MDS-3 becomes almost overloaded. More interestingly, between 15 to 23 minutes, both MDS-4 and MDS-5 become idle, while the other three peers remain with heavy loads. Following that, the load on MDS-3 suddenly disappears, while MDS-4 is assigned workloads again, and its load returns to a high level accordingly.

Second, compared to the Filebench-Zipfian workload, the CNN one also experiences a significant metadata load imbalanced problem at an extreme. As shown in Figure 3(b), regardless of time moving, almost no workloads are balanced to MDSs except MDS-4. This implies that only one MDS is actively working, and at the same time the resources allocated for the other peers are wasted.

The above counter-intuitive results lead us to further explore how the built-in CephFS metadata load balancing mechanism works and why it is not suiting the two workloads in depth. To do so, we plot the number of migrated inodes in Figure 4. Considering the Filebench-Zipfian read workload (Figure 4(a)), between 0 and 5 minutes, it is as expected that the number of migrated inodes keeps increasing (those inodes are migrated from MDS-4 to the other four) so that the loads on the four importer MDSs quickly go up. However, between 5 and 20 minutes, even though the loads among five MDSs are ill-balanced (Figure 3(a)), no migration activities are observed to react to such a situation, i.e., the number of migrated inodes remains stable in Figure 4(a). In such a case, the five MDSs' load is 13530, 14567, 15625, 11610, and 2692, respectively, and the average is 11604. The built-in balancer mistakenly decides not to re-balance since it believes that the busiest MDS's

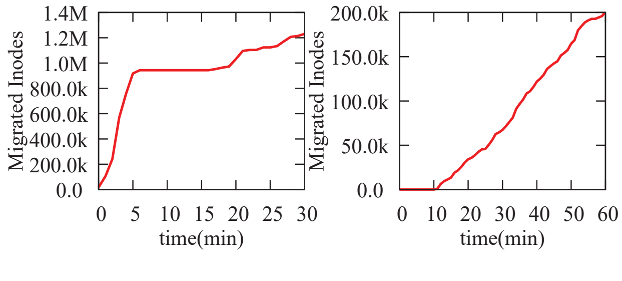


Figure 4: The total number of migrated inodes

load is close to the average and ignores the load gap between the heavy and light MDSs. Furthermore, migration even occurs when the cluster load is moderate, but the MDS with the highest load (small absolute value) is relatively far from the average. This observation reveals the first inefficiency: *the built-in load balancing mechanism fails to draw an accurate view of the cluster state to discover imbalanced situations timely and trigger the re-balance when needed.*

We also zoom in on the decisions on the amount of migrated metadata generated in two periods of the Filebench-Zipfian read workload experiment, namely, 0-5 minutes and 20-30 minutes. During the first interval, nearly 98% of inodes managed by MDS-4 is moved to other MDSs, which results in a very light load at MDS-4 after this migration. In the second interval, MDS-4 acts as an importer and receives 88% of inodes shipped from MDS-3. However, when this migration completes, the loads on MDS-3 and MDS-4 are just swapped, and thus the migration does not make much sense while incurring performance cost. We call this a *ping-pong effect.* This is because the heavy-loaded MDS would plan to migrate metadata as much as possible to other peers, often exceeding the maximal migration capacity during one re-balance time interval. For instance, we observe there were 15 subtrees in the migration task queue, but only 2 were successfully migrated. More importantly, when determining the amount of migrated inodes, the lag effects of metadata migration have not been taken into consideration, leading to over-migration. Upon the completion of migration, the importer MDS sharply becomes hot while the exporter may be completely idle. This result reveals the second inefficiency: *the aggressive migration decision would offset the benefits of load balancing and artificially introduce new imbalanced situations.*

Finally, we look into the migration activities taking place when running the CNN pre-processing workload. In Figure 4(b), metadata are continuously migrated among MDSs as time evolves. Unfortunately, this eager migration trend contradicts the fact that the load never moves from the busy MDS-4 to other idle ones. We analyze all migrated inodes and find that the vast majority of them are never visited after their migration. This is a direct consequence of the temporal locality-based balancing strategy, i.e., selecting hotspots as migration candidates. Nevertheless, this workload plus the NLP one do not re-visit scanned files, and thus invalidate the built-in load balancing mechanism's assumption. We summarize this as the third inefficiency: *the one-size-fits-all candidate selection policy does not consider the unique access patterns of various workloads.*

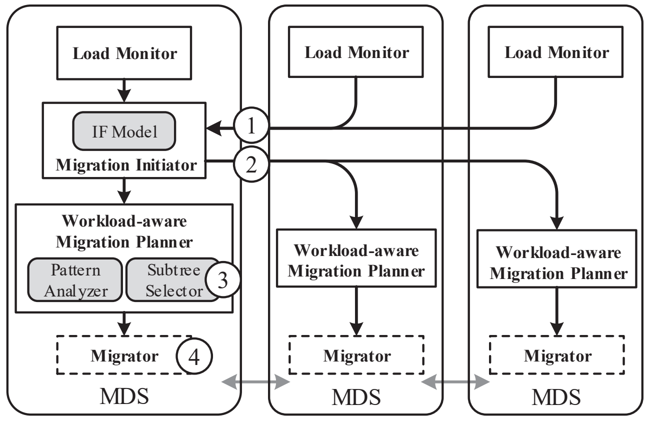


Figure 5: Lunule architecture. The dashed-line boxes are existing components of the CephFS built-in load balancer, while all the solid-line boxes are new components introduced by Lunule.

2.3 Design considerations

In this paper, we plan to address the aforementioned inefficiencies of existing metadata load balancing mechanism. To fulfill this goal, we identify three critical factors determining the effectiveness and efficiency of such a dynamic subtree partitioning mechanism as follows. First, considering migration introduces background traffics that may contend resources with foreground requests, one has to accurately monitor the cluster load and determine when to trigger the re-balance procedure. Second, the target balancer should make the right decisions on how much load to be migrated for reducing the migration frequencies and avoiding long-run migration jobs. Third, one has to appropriately find a set of directories as the migration candidates for invalid migration avoidance. All these principles motivate the design of Lunule as follows.

3 DESIGN OF LUNULE

Based on the problems identified, we propose a novel load balancing service for CephFS, Lunule, with the following two critical missions. First, Lunule needs to accurately monitor and report the imbalance intensity level of the whole MDS cluster as time goes. Second, relying on the real-time collected statistics, Lunule should be able to make suitable plans for various workloads to select a reasonable amount of subtree partitions as candidates to migrate among a carefully chosen set of MDSs.

3.1 Overview

Figure 5 gives an architecture view of Lunule, which augments the existing metadata service with three key components. First, we deploy a *Load Monitor* at each MDS to monitor the load pressure and collect the number of metadata requests processed per second by the corresponding MDS. Second, there is a *Migration Initiator* sitting on one of the MDSs to make real-time decisions on when the migration should take place and how much metadata should be exchanged among metadata servers. The decisions made are dependent on the IF values, which summarize the load dispersion among MDSs, and are computed by applying stats collected from all *Load Monitor* to the residing analytical model. Though the *Migration Initiator* is a centralized component, it will not be a performance bottleneck since the migration procedure takes place in the background, runs every time epoch (configurable, 10 seconds by default), and consumes little resources such as CPU, memory, and network bandwidth.

The third component is a *Workload-aware Migration Planner*, which is also deployed at each MDS and functions independently. It is further split into two pillars. First, there is a *Pattern Analyzer*, which learns the I/O patterns of different workloads concentrating on subtrees the corresponding MDS manages, and computes their probability for being migrated by understanding the past workloads' impacts and predicting their future metadata access variances. Second, when re-balance is triggered, on an *exporter* MDS, its *Subtree Selector* chooses an appropriate set of subtrees for migration, which meets the amount computed by *Migration Initiator* and suits for the future visits by the target workload.

Workflow. Lunule makes migration decisions in a fixed time interval, called *Epoch*. In each epoch, every *Load Monitor* sends the observed metadata throughput numbers to *Migration Initiator* (①). When the metadata cluster's imbalance degree (indicated by the *IF* value) exceeds a pre-defined threshold, *Migration Initiator* triggers the load re-balance procedure, and generates the migration plans, which assign the exporter and importer roles to MDSs and pair the exporters' demands and importers' capacities. Then, the *Workload-aware Migration Planner* on each exporter MDS will be notified of its assigned migration tasks (②). Upon the arrival of migration tasks, the *Subtree Selector* chooses a list of suitable subtrees partitions (③), which will then be fed into the existing *Migrator* and relocated from MDSs with heavy loads to the light ones (④).

3.2 IF model driven re-balance

We develop an analytical model that takes the collected metadata load stats as input and predicts the *Imbalance Factor* values of the whole metadata cluster, denoted as *IF*, and representing the intensity level of metadata load imbalance in each epoch. The key challenge and novelty of the model is to address the prediction inaccuracy achieved by the used linear model, and to identify benign imbalanced situations.

First, we abandon the linear metadata load modeling method used in CephFS and Mantle's case studies, relying on the simple comparison against the average numbers of collected loads. This is because the linear model likely results in skewed loads on some MDSs even with heavy migration activities. Instead, we adopt the *Coefficient of Variation* (CoV) [20, 46] as the basic building block for our model. CoV is a statistical measure of the dispersion of data points around the mean, and is commonly used to compare the data dispersion between distinct series of data. In our context, given a metadata cluster consisting of n servers, its load *CoV* value can be computed as follows:

$$CoV = \frac{\sigma(l)}{\bar{l}} = \frac{\sqrt{\sum_{i=1}^{n}(l_i - \bar{l})^2/(n-1)}}{\sum_{i=1}^{n} l_i/n}, \tag{1}$$

where l_i represents the current load of the i^{th} MDS, i.e., the number of metadata requests served by that MDS per second (*or short*, IOPS), while $\bar{l}$ and $\sigma(l)$ are the average load across all MDSs and the corresponding corrected sample standard deviation. We choose to use IOPS as the major metric to estimate the degree of the busyness of the target MDS cluster as it has a natural reflection of the immediate load of each MDS.

However, it is not ideal to directly use Equation 1 to quantify the imbalance level due to the following two issues. First, the basic equation has an unfixed range $(0, \sqrt{n}]$, but we expect CoV to change in a fixed range so that it is possible to compare it against a pre-defined threshold, above which re-balance must be triggered. To address this, we normalize CoV to its maximum value $\sqrt{n}$, which corresponds to the most imbalanced situation, where only one MDS is handling metadata requests and the others are idle.

The second problem is that *not all imbalanced situations need to perform the re-balance procedure*, e.g., all MDSs are much under their maximum throughput despite their load differences. To address this, we additionally introduce an urgency parameter U to describe if the current imbalance is harmful. The higher the value, the more urgent the migration needs to be performed. We model U as a logistic function, and U's growth is limited by the load on the most massive loaded MDS. Formally, U is computed as follows:

$$U = (1 + e^{\frac{1-2u}{S}})^{-1}, \quad u = l_{max}/C \tag{2}$$

In Equation 2, l_{max} denotes the maximal throughput delivered among all MDSs in the corresponding epoch, while C is a pre-defined parameter representing the maximal IOPS that a single MDS theoretically could achieve[1]. Therefore, u means how the most massive loaded MDS looks like, compared to the maximal MDS capacity. U's curve is S-shaped and its smoothness is controlled by a knob S with a range of $(0, 1)$. In our evaluation, we set it to 0.2.

Finally, we combine the above two equations and the proposed normalization together into the following equation, which is used to compute the *Imbalanced Factor* of the whole metadata cluster.

$$IF = \frac{CoV}{\sqrt{n}} \cdot U \tag{3}$$

Migration Initiator applies the above model to compute the IF value of the MDS cluster in each epoch and move on to the role determination phase if the load imbalance degree is no longer tolerable, i.e., exceeding a pre-defined threshold. In this phase, as illustrated in Algorithm 1, *Role Decider* takes the load stats gathered from each MDS and computes a matrix $E = \{E_{ij}\}$, where E_{ij} corresponds to the number of loads that need to shipped from MDS-i to MDS-j. As motivated in Section 2, the *amount* value plays a key role in maximizing the benefits of migration and avoiding negative impacts of over-migration. Compared to the existing solutions we studied before, where they compute that value only from the perspectives of exporters, we take a more comprehensive approach and introduce the following two novelties.

First, considering the overhead imposed by migration, we put an upper limit on the exporting/importing demand (eld or ild) for each exporter/importer MDS, and set it to its maximal capacity during one epoch (line 8, 12). This capacity is a constant value and can be computed as the maximal number of inodes one MDS can theoretically send out or receive. Second, for each importer, we need to

[1]We assume all MDSs are allocated the same physical resources and deliver the same capacity. Handling heterogeneous settings is orthogonal to our solutions.

Algorithm 1: Role and migration amount determination

```
Input  : A list of MDS load stats $\vec{M} = \{m_i\}(i = 0, 1, ..., n-1)$;
         Load threshold $L$; Capacity $Cap$;
Output : Export decision: An $n \times n$ matrix $\vec{E}$ (initially all zeros);
1  $Importers \leftarrow \emptyset$; $Exporters \leftarrow \emptyset$;
2  $\bar{ld} \leftarrow Average(m_i.cld)$;
3  foreach $m_i \in \vec{M}$ do
4      $\Delta ld \leftarrow |m_i.cld - \bar{ld}|$;
5      if $(\Delta ld/\bar{ld})^2 > L$ then
6          if $m_i.cld > \bar{ld}$ then
7              $Exporters.push(m_i)$;
8              $m_i.eld \leftarrow min(Cap, \Delta ld)$;
9          else
10             if $m_i.fld - m_i.cld < \Delta ld$ then
11                 $Importers.push(m_i)$;
12                 $m_i.ild \leftarrow min(Cap, \Delta ld - (m_i.fld - m_i.cld))$;
13 foreach $i \in Exporters$ do
14     foreach $j \in Importers$ do
15         if $m_i.eld > 0$ && $m_j.ild > 0$ then
16             $E_{ij} \leftarrow min(m_i.eld, m_j.ild)$ ;
17             $m_i.eld- = E_{ij}$;
18             $m_j.ild- = E_{ij}0$;
```

further take into account the impact of its future load change for over-migration avoidance. To do so, we could apply a linear regression model against the collected historical load stats (cld) to predict the possible load in the next epoch (fld). The importer role can be assigned if the future load increase cannot fulfill the gap, and we also compute its anticipated importing amount (lines 10-12). Third, we consider the bidirectional demands from both exporter and importer for generating the migration plan. For an exporter-importer pair, we check if the exporter has the migration demand, and meanwhile the importer has capacity to accommodate such migration (line 19). If so, we make the migration amount equal to the minimum value of the migration demand and importing capacity (line 20). We then subtract the determined amount from the exporter's eld and importer's ild (lines 21-22). Finally, this algorithm runs iteratively and finishes when all pairs are checked.

3.3 Workload-aware subtree selection

Once the migration decisions are made, the following step will be choosing and moving an appropriate set of subtrees from exporters to importers. The study in Section 2 reveals that performing subtree selection determines whether the target balance could be achieved after the corresponding migration. The built-in migration policy in the widely adopted CephFS relies on the heat of file metadata access measured at runtime. This "one-size-fits-all" solution fails to deliver good performance for various scenarios, and the reasons are two-fold. First, the heat-based solution neglects the impact of spatial locality, which is more influential in AI and big data analytic scenarios than temporal locality. Second, though the heat of subtrees can be accumulated and decayed, it still fails to model the significant variances of future loads.

In Lunule, we advance this migration candidate selection phase to be workload-aware with the following innovations. First, instead of maintaining heat counters for subtrees, we assign them with a *migration index* (*or short, mIndex*), which corresponds to the predicted future load on the target subtree over time. The migration index is larger, the probability for the corresponding subtree to be migrated is higher. Migrating subtrees with higher migration indices will likely ship superfluous workloads on a busy MDS to a less loaded one, thus eliminating invalid but expensive migrations.

The computation of migration indices needs to take a joint consideration of the impacts of both temporal locality and spatial locality exhibited in workloads. To do so, we introduce α and β as the impact factors of temporal and spatial locality, respectively, indicating the inclination of the recent workloads on subtrees to either of the two access patterns. To estimate the values of the two variables, on every MDS, we maintain the history metadata access trace and break it into fixed-size short sequences (a.k.a *cutting windows*). Periodically, we compute the recurrent visit ratio of the most recent cutting windows, which is equal to the division of the number of recurrently visited inodes and the total visited inodes. This ratio is computed on per subtree basis and assigned to the corresponding α. Unlike this, for β, we keep track of the unvisited inodes in each subtree, and assign to β the ratio of unvisited inodes over the total number of metadata visits in the most recent cutting windows.

In addition to the estimation of the temporal and spatial locality inclination, we have to predict the number of future visits falling into the two categories, which are summarized by l_t and l_s. We calculate the value of l_t by counting the number of metadata visits concentrating on the corresponding subtree in the last N cutting windows. Departing from the l_t estimation, for a given subtree, we will increase its l_s by 1 if one of its unvisited inodes is accessed in the current cutting window. Finally, we observe that there exist strong access correlations between sibling subtrees when workloads exhibit spatial locality. Therefore, to respect to that observation, we will also select one of its sibling subtrees with a certain probability, and increment l_s of the selected subtree by 1.

Finally, we provide a unified view of the temporal and spatial locality impact exploration in the following formula:

$$mIndex = \alpha \cdot l_t + \beta \cdot l_s \tag{4}$$

Subtree selection. Each MDS ranks the set of subtrees it manages by their migration indices in descending order. For each migration decision ⟨*exporter, importer, amount*⟩, the exporter MDS first scans its ranked subtree list and tries to find a set of subtrees, whose aggregated load number matches *amount*. For each encountered subtree, we place it into the candidate bag if its *migration index* is equal to or below *amount*, and then accordingly decrement *amount*. Otherwise, we have to split the scanned subtree. There are two cases that we should consider. First, if the metadata accesses are concentrating on that subtree itself, then we divide it into two subtrees, with one's migration index matching *amount*. Second, if some of its descendant subtrees are hotspot, then we remove them from their ancestor according to the left *amount*.

3.4 Discussions

Generality of Lunule. The main limitation of Lunule is that Lunule is tightly coupled with CephFS and the dynamic subtree partitioning-based metadata management. It is beneficial to extend Lunule to work with other metadata services like IndexFS [33], which adopts the hash-based metadata management, to enable a wider range of distributed file systems. It is straightforward to apply the IF model to these scenarios since assessing the load imbalance level of the target MDS cluster is a general assumption. However, it is challenging to directly use the subtree selection method presented in Section 3.3 to file systems that use different metadata organizations than dynamic subtree partitioning. To solve this challenge, we envision to design a generic framework that is similar to but more powerful than Mantle [35], to support various migration candidate selection policies. We will explore these opportunities in future.
Overhead. Lunule introduces extra resource consumption for exchanging and keeping tracking of load information of MDSs, as well as computing the migration plans when needed. However, we have measured and concluded that the extra overhead imposed on the CPU, memory, and network usages is negligible, compared to the original CephFS metadata load balancer. For instance, within each epoch, all MDSs except the primary MDS where *Migration Initiator* resides observe only a 0.94 KB increase in the out-bound network usage for reporting the metadata load information to the primary MDS. In a 16-MDS cluster, the primary MDS incurs only 14.1 KB extra in-bound network usage for receiving metadata stats from other peers per epoch. In addition, when setting the epoch time to 10 seconds, each MDS consumes only 1.37% more memory space to maintain data structures to store load information. Finally, there is no visible CPU utilization variance when enabling Lunule, thanks to Lunule's lightweight design.

4 EVALUATION

In this section, we first introduce the implementation of Lunule, and then evaluate Lunule against state-of-the-art baselines with several real-world workloads. Throughout all experiments, we focus on the following questions: (1) can Lunule eliminate the load imbalanced situations reported by other baselines for a cluster of MDSs? (2) What are the performance implications (throughput, latency, and job completion time) of achieving good metadata load balance enabled by Lunule? and (3) How does Lunule adapt to changes in workloads and the cluster scale-out?

4.1 Implementation details

We put all components together into a novel metadata load balancer Lunule, which is implemented atop CephFS [43] (Ceph version 12.2.5) and consists of 1800 lines of C++ code[2].We reuse the original migration code but perform heavy changes to the MDS and message modules of CephFS in the following parts:
Stats recording: Original *popularity* counter is removed entirely and replaced by *mIndex*. Each inode has a boolean queue of n-length, recording whether accessed in the last n epochs. Whenever the metadata is accessed, its parent directory checks the queue and modifies its l_t or l_s value, following the rules described in Section 3.3. To reduce the CPU overhead, the value of *mIndex*, α, and β

[2]Code is available at https://github.com/mdbal-lunule/lunule

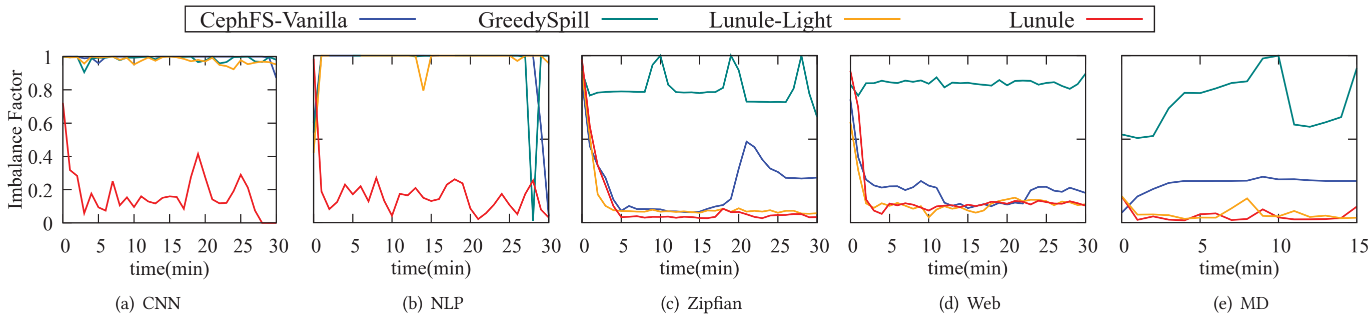


Figure 6: The imbalance factors of different workloads using different load balancer (lower is better). Note that the MD experiments ended around 15 minutes, much shorter than the other four. This is because in the MD workloads, concurrent clients continuously create new files and lead the MDSs to run out of memory beyond 15 minutes.

is updated only once per epoch rather than immediately after each request handling.

Stats collection: To reduce the communication overhead in large-scale conditions, we replace the original decentralized, N-to-N stats collection module with a centralized, N-to-1 *Load Monitor* module, as described in Section 3.1. We assign *Migration Initiator* to the MDS with the lowest rank, e.g., MDS-0, in most cases. Other MDSs first send their states to the initiator, and the latter communicates with all other servers after making a decision. To this end, a new message type, *Imbalance State* message, which consists of MDS rank ID and metadata requests rate, is introduced into the CephFS to replace the original *Heartbeat* message.

Migration trigger and assignment: Corresponding to the above part, the *Migration Initiator* calculates the imbalance factor after cluster stats are collected and then decides whether to migrate. Lunule brings in another message type, *Migration Decision* to dispatch decisions. Each decision message specifies the *amount* of loads of one exporter that need to be migrated to the importers. The *Migration Initiator* sends the assignments of exporting load to the exporters, and the exporters then go ahead to the next step of selecting subtrees upon they receive the decisions.

Subtree selection: Lunule uses a simple recursive algorithm to select the migration subtrees starting from the root directory via the following three searching paths: (1) determines whether there is a subtree with *mIndex* that is approximately equal to the migrating *amount*, allowing a 10% difference; (2) searches the subtree whose *mIndex* is larger than the migrate amount and splits it into two partitions, where the resulting *mIndex* of one partition is close to *amount*; (3) selects a minimal set of subtrees, the sum of whose *mIndex* values roughly meets the migration demand. This method is similar to CephFS's, and we plan to extend it in future work by implementing a dynamic strategy of the subtree selection.

4.2 Experimental setup

Test platform. Our experiments run on a local cluster with 16 bare-metal servers, connected via a 56Gb/s IPoIB network. Each server has 2 Intel(R) Xeon(R) E5-2650 V4 CPUs, 64 GB memory and 1.6TB NVMe SSD (Intel P4610), running CentOS version 7.3.10.0-862.14.4.el7.x86_64. We use five physical servers dedicated for running MDS daemons. We use the other ten servers for deploying a single cluster Monitor daemon for CephFS, OSD daemons for hosting the data and client emulators for generating workloads, respectively. Unless otherwise pointed out, we configure a cluster of 5 MDS daemons and 6 OSD daemons, and evenly distribute 100 clients across the remaining non-MDS servers for generating workloads. For the dynamism and scalability experiments, we increase either the number of MDSs or clients for introducing more computing resources or more workloads. With the increase in the metadata stress, we also add more OSD daemons accordingly until the maximum of 27 is reached.

Baselines and configurations. The CephFS built-in load balancer is our natural baseline, denoted as "Vanilla". We additionally run the GreedySpill metadata load balancer (denoted as "GreedySpill"), which is originally from GIGA+ [30] and implemented and integrated into CephFS via the Mantle metadata framework [35]. The GreedySpill balancer aggressively sheds metadata loads to all MDSs by triggering migration when some MDSs do not have any load, and moving half of the load from the loaded MDS to its idle neighbor MDSs during migration. Its code is at [12].

To explore the benefits of various system components, we configure Lunule with two variants, "Lunule-Light" and "Lunule", with the workload-ware migration optimization switched off/on.

Workloads and metrics. We run five workloads (listed in Table 1), namely, CNN image pre-processing (CNN), NLP training (NLP), Filebench Zipfian read (Zipf), Web trace replay (Web), and MDtest create (MD). Among the five workloads, CNN, NLP, and Web are real-world workloads, while Zipf and MD are benchmark workloads. The MD workload is write-only, while the remaining are read-only.

To make sure load re-balance can be triggered, we stress the target system by launching 100 concurrent clients simultaneously, with each client running its own workload. Alongside the experiments with five individual workloads, we also test how the system behaves when we mix all of them. For the mixture workload, we split clients into four groups, each group of clients running the same type of workload. Since our primary goal is to improve the metadata performance, unless otherwise stated, in most experiments, we skip the data path and only exercise the metadata retrieval. In addition, we enable the data access when testing the end-to-end performance. To understand the system behaviors, we measure the following metrics, namely, imbalance factors, clustered metadata throughput, and job completion time.

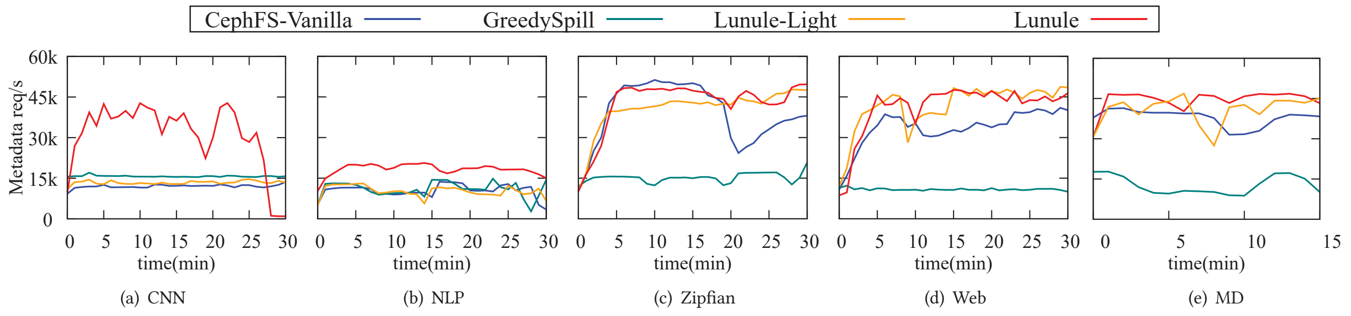


Figure 7: The peak throughput of the MDS cluster with combinations of four workloads and four balancers (higher is better).

4.3 Single workload test

Imbalance elimination. We begin our analysis with the imbalance reduction of our solution, compared with other baselines. Figure 6 summarizes the *imbalanced factor* (IF) values of the five-node MDS cluster when running five workloads with four different load balancers. The lower IF values indicate a better load balance. Among all tested cases, *GreedySpill* performs the worst and its IF value is close to 1, leading to the most imbalanced situation, where almost all loads adhere to a single MDS. This is because the load estimation model *GreedySpill* relies on takes into account little local information rather than global stats and is not accurate to trigger necessary re-balances. Meanwhile, at each time when re-balance is triggered, *GreedySpill* always plans to ship half of the hosting inodes from the busy MDS to its neighbors with light load, imposing high migration cost.

CephFS-Vanilla significantly outperforms *GreedySpill* for the web workload (Figure 6(d)), where it quickly balances the workload to all other MDSs at the beginning and keeps the IF value constantly low since then. This is because the web workload exhibits a high temporal locality, which is well handled by the hotness-based migration policy adopted by the original load balancer. Compared to the web workload, with regard to the Zipfian read (Figure 6(c)), although *CephFS-Vanilla* still performs better than *GreedySpill*, its IF values fluctuate and keep increasing after reaching the lowest around 5 minutes. For the MD workload, *CephFS-Vanilla* achieves a roughly constant IF value around 0.25, corresponding to an imbalance situation, where at least one MDS has almost no load. The trends observed by Zipfian and MD is a direct consequence of the aggressive migration decisions, which does not consider the demands from the importer side and the lag of the migration effects.

Contrary to the Zipfian read and web workloads, *CephFS-Vanilla* achieves the same poor results as *GreedySpill* for the other two workloads, namely, CNN (Figure 6(a)) and NLP training (Figure 6(b)). These results are highly relevant to the unique characteristics of the two workloads, i.e., they all are scanning-type workloads, and each file is rarely re-accessed. Therefore, the hotness-based migration policy employed by *CephFS-Vanilla* fails to predict the future visit pattern. All these results again prove the existence and severity of the load imbalance problems in the existing leading solutions, which are consistent with the findings presented in Section 2.2.

Overall, the two Lunule variants achieve better load balance than both *CephFS-Vanilla* and *GreedySpill*. Lunule performs the best, and produces low IF values at most time, corresponding to the best-balanced situation, where no significant load gaps exist between any pair of MDSs. Moreover, the average IF values of Lunule are 17.9-90.4% lower than the ones achieved by *CephFS-Vanilla* and *GreedySpill* for the five workloads, respectively. In more detail, for the CNN and NLP workloads, the performance gains delivered by *Lunule-Light* are limited, since the accurate IF model itself is not sufficient for good load balance, and the two workloads are more dependent on the migration policy. Unlike *Lunule-Light*, *Lunule* obtains much lower IF values, because it is able to choose the right set of subtree migration candidates based on the identified I/O patterns. In this case, it always balances non-visited inodes to other MDSs for parallelizing future visits. For Zipf, web and MD workloads, both Lunule variants deliver similar results. This is due to the fact that these two workloads are more sensitive to the imbalance factor model, and the default subtree selector *Lunule-Light* relies on generated the same migration plans and made the same subtree selection decisions, as *Lunule* does.

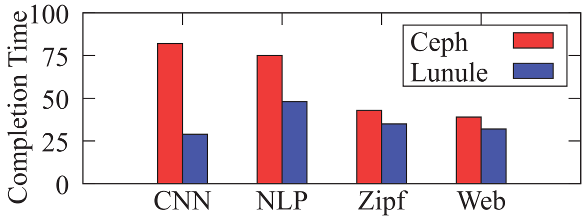


Figure 8: The overall performance of 4 workloads with the original load balancer and Lunule with data access enabled.

Metadata performance improvement. Next, we shift our focus to the implication of the reduction in imbalance factor on the overall performance of the metadata service. Figure 7 presents the throughput of handling metadata requests over time computed by aggregating the IOPS stats reported by each single MDS, for the combinations of five workloads and four load balancers. As expected, there exists a strong negative correlation between the results in Figures 6 and 7, i.e., the lower the IF values, the higher the aggregated IOPS. Overall, because of the accurate sensing model and the workload-aware migration policies, *Lunule* significantly outperforms all other three baselines. For instance, as shown in Figure 7(a), with the CNN workload, *Lunule* introduces a 2.81×, 2.14×, and 2.15× improvement, compared to *CephFS-Vanilla*, *GreedySpill* and *Lunule-Light*, respectively. Similarly, for NLP, another workload exhibiting strong spatial locality, Lunule still outperforms the

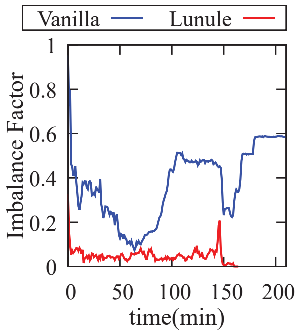


Figure 9: IF values, mixed workload.

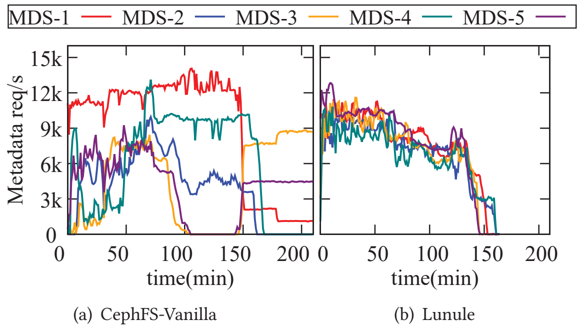


Figure 10: Throughput numbers, two systems, mixed workload.

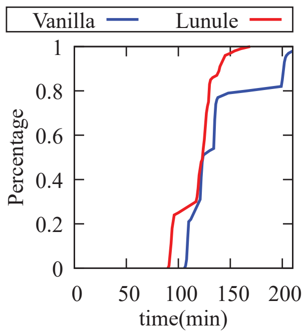


Figure 11: Completion time CDF, mixed workload.

three baselines by 1.76×, 1.65×, and 1.78×. For the three more skewed workloads, Zipf, web, and MD, for which the hotness-based load balancing policy primarily targets, our solution still performs better even though the improvement is not as great as we observe in both CNN and NLP. For instance, with regard to MD, in Figure 7(e), *Lunule* and *Lunule-Light* achieve on average 17.0% and 7.6% higher throughput than the best-performed state-of-the-art baseline CephFS-Vanilla, respectively. All results validate that our design is suitable and reasonable when facing different types of workloads, and capable of eliminating or alleviating the imbalanced problems of the baselines.

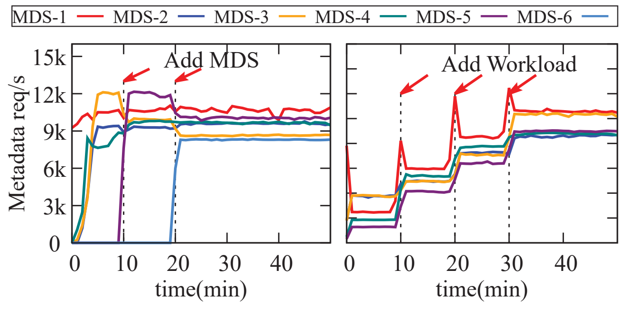
End-to-end performance improvement. Next, we shift our attention to investigating the impact of the good metadata load balance achieved by Lunule on the CephFS overall performance. To do so, we extend the above experiments by enabling data access and we measure the job completion time. We report the job completion time of four workloads except MD in Figure 8, since MDtest is intensively used to solely test the metadata performance and we follow the common practice to not enable data access [22, 33]. For the CNN, NLP and Zipf workloads, Lunule outperforms *CephFS-Vanilla* by 18.6-64.6% in the job completion time. This indicates that the metadata access accounts for a high portion of the whole workloads, and the metadata performance acceleration enabled by *Lunule* contributes to the overall performance improvement. Unlike these workloads, for the Web trace replaying, we observe limited end-to-end performance gains. The reason is two-fold. First, as shown in Figure 7(d), the metadata load imbalanced intensity level of Web is lowest among the four workloads. Second, enabling data path would dilute the benefits of the metadata performance improvement.

4.4 Mixed workload test

To emulate a more realistic setup, we evaluate *Lunule* with a mixed workload, where we partition 100 clients into four groups and each group of clients runs one of the four single workloads presented before. We omit the comparison related to *Lunule-Light* and *GreedySpill* since the former system performs worse than *Lunule* while the latter performs worst.

Imbalance elimination. Figure 9 shows the comparison of the imbalance factors of the MDS cluster with *Lunule*, and *CephFS-Vanilla* using the mixed workload. Similar to the single workload results, the built-in load balancer's imbalance factor values fluctuate significantly, with the maximum value up to approximately 0.6. Although *CephFS-Vanilla* perfectly balances loads among MDS at the time of 50 minutes, it takes 3.33× longer time than *Lunule* for doing so. This is because the inaccurate migration decision making and ineffective migration target selection cause additional costs. However, this best-balanced situation enabled by *CephFS-Vanilla* does not last for an extended period. Instead, the loads among the five MDSs become skewed again at 85 minutes. The variations with large gaps continue to proceed. This is because the completion of client jobs at different time likely results in new imbalance issues and would lead *CephFS-Vanilla* to frequently trigger its ineffective re-balance procedure. As opposed to *CephFS-Vanilla*, Lunule is able to lead the MDS cluster to nearly reach the best-balanced state during the whole testing period, i.e., the imbalanced factor value is always close to zero. Again, this is because of the accurate model and workload-aware migrations for making a reasonable trade-off between balancing loads and preserving locality. We also observe that the *Lunule*'s curve is much shorter than the one of *CephFS-Vanilla*. This is because Lunule could leverage the most resources for parallelizing metadata request processing, and the workloads always run faster with *Lunule* than *CephFS-Vanilla*.

Overall performance improvement. To understand the impacts of load balancing on the overall system performance, we take a fine-grained view of the throughput numbers of each MDS in Figure 10 for both *Lunule* and *CephFS-Vanilla*. At a high level, the higher imbalanced factor values correspond to skewed loads among the deployed five MDSs. For instance, as shown in Fig.10(a), the load is initially handled by MDS-1, and then shipped to other MDSs between 0 and 150 minutes. During this period, MDS-4 sharply becomes overloaded, and the load on MDS-5 drops by 10.6%. At the 44^{th} minute, MDS-5 receives loads from MDS-2 and becomes busy again. Meanwhile, the loads on MDS-3 and MDS-5 decline to zero at 105 minute and IOPS increase to 7.5k/s and 4.3k/s at 152 minute, respectively. This all indicates that the loads among MDSs managed by *CephFS-Vanilla* are highly skewed, and the ping-pong effects are clearly observed. In contrast to *CephFS-Vanilla*, as shown in Figure 10(b), the throughput numbers achieved by each MDS are more balanced within *Lunule*. This balanced situation does not sacrifice the overall performance, instead, improves evidently it. For instance, during 0 and 50 minutes, *Lunule* delivers 48k IOPS



(a) Dynamic MDS (b) Dynamic Workload

Figure 12: Performance of Lunule when handling dynamics.

as the clustered throughput, which is 1.6× higher than the one of *CephFS-Vanilla*. Around 60 minutes, all MDSs are observing declining throughput numbers, since some workloads have finished.

Job completion time. Finally, we continue to investigate the impact of load balancing on the job completion time under the mixed workload. We plot the CDF curves of job completion time of all 100 clients in Figure 11. Among all clients, 50% could finish their requests before 125 minutes, regardless of using *Lunule* or *CephFS-Vanilla*. However, half of the fastest clients finish in 100 minutes with *Lunule*, 13.1% shorter than the one achieved by *CephFS-Vanilla*. Furthermore, *Lunule* makes nearly 80% of clients complete their jobs before 130 minutes, while *CephFS-Vanilla* delivers 24.8% longer completion time. Again, *Lunule* defeats *CephFS-Vanilla* in the tail completion time reduction, e.g., the 99% job completion time of *Lunule* is 154 minutes, 1.42× better than the baseline. This improvement is a direct consequence of the higher IOPS and better load balance results shown above.

4.5 Dynamic adaptation test

As the load balancing is crucial for adapting to changes in terms of cluster expanding and denser workloads, here, we further stress Lunule with these two types of changes, and the results are summarized in Figure 12. The workload we use is Zipfian (We observe similar trends with the other three workloads and omit them due to space limit).

Expanding MDS cluster size. In Figure 12(a), there are 4 MDSs initially and we add one more MDS at the time of 10 and 20 minutes, respectively. In the first phase, workloads are first concentrating on MDS-1 and then moving to the other 3 MDSs, resulting in 41k IOPS as the aggregated throughput. When adding a new MDS-5, thanks to the load balancing mechanism of Lunule, MDS-5 quickly absorbs migrated loads from other peers and the clustered throughput increases to 51k IOPS. Similarly, when adding MDS-6, Lunule timely transitions from a balanced state to another. In such a state, MDS-3 and MDS-5's throughput drops by 13.1% and 16.4% but the clustered throughput is improved by 10.0%. This implies that Lunule can easily leverage new spare resources for serving metadata requests in parallel among the expanded cluster while minimizing the migration's negative impacts.

Increasing client workloads. In Figure 12(b), we partition the experiment into four phases, wherein the first phase, we launch 10 clients to simulate workloads and then add 10 more clients when each subsequent phase starts. First, at the beginning of each phase, there is a throughput spike for MDS-1, because we always force newly added clients to first contact this MDS. However, the extra load introduced to MDS-1 is immediately balanced to other peers so that we can observe stable sustained throughput for different MDSs. From the first phase towards the last one, the throughput number of each MDS constantly increases, since more workloads are introduced and Lunule is able to coordinate all MDSs for evenly partitioning loads. Moreover, in the first phase, the cluster load is more imbalanced than others, but Lunule does not trigger rebalance after the very first epochs. This is because our imbalance-sensing model knows that this imbalanced situation is not harmful since all MDSs are lightly loaded.

4.6 Other experiments

Scalability test. Figure 13(a) shows the scalability of the MDS cluster by measuring the peak throughput w.r.t. the increasing number of MDSs. For the MD workload, Lunule scales linearly on a cluster of 16 MDSs (the largest MDS cluster that our local environment can support), and delivers a throughput of more than 112k requests per second. The slight deviation between Lunule's curve and the linear scaling curve (the dotted line) is because we run out of the client machines' capacity and the MDS cluster is slightly below the saturation point. We expect Lunule to bring performance benefits with larger scale deployments. In addition to MD, we also observe similar scalability results for workloads like CNN.

Lunule vs. Hash-based solution. Figure 13(b) compares the performance between Lunule and a hash-based solution, denoted by "Dir-Hash". To make a fair comparison, we simulate the hash-based baseline within CephFS by leveraging its existing features [15]. We first split the file system namespace into fine-grained subtrees and then evenly distributing these subtrees to all MDSs by statically pinning them to different MDSs according to their hash values. Results with the Web workload highlight that Lunule outperforms Dir-Hash and CephFS-Vanilla by up to 22.2%. Figure 14 shows the shortcomings of Dir-Hash in detail: Metadata distributes evenly across 5 MDSs as shown in Figure 14(a), while Figure 14(b) indicates that this distribution not only leads to evident request imbalance but also fails to balance load dynamically. In addition, Dir-Hash imposes 98.0% higher number of forwards among MDSs than both Lunule and CephFS-Vanilla. These forward metadata requests are needed to traverse file path because Dir-Hash does not preserve locality and introduces more metadata fragmentation.

5 RELATED WORKS

Decoupling metadata from data makes it easy to scale their performance independently. However, this also introduces challenges to efficiently manage metadata due to its hierarchical namespace and more stringent requirements for scalability and fault tolerance. Upon its emergence, GFS[16] and HDFS[18] assume a single server (a.k.a namenode) to accommodate all metadata. Unfortunately, studies show that the real-world workloads easily overwhelm this single metadata server setting, making it become a performance bottleneck [25, 28].

More recent distributed file systems maintain a cluster of metadata servers for serving ever-increasing demands from clients to

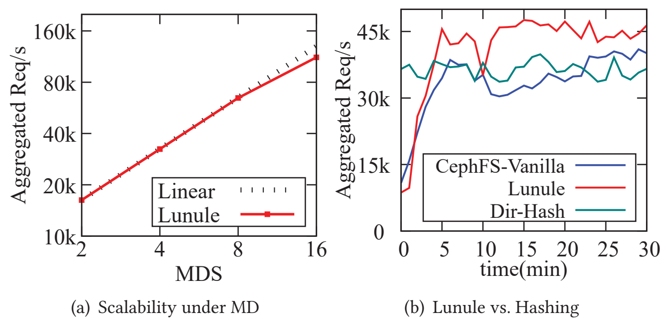


Figure 13: (a) reports Lunule's scalability under the MD workload, while (b) shows the performance comparison between Lunule, CephFS-Vanilla and the hash-base solution under the Web workload.

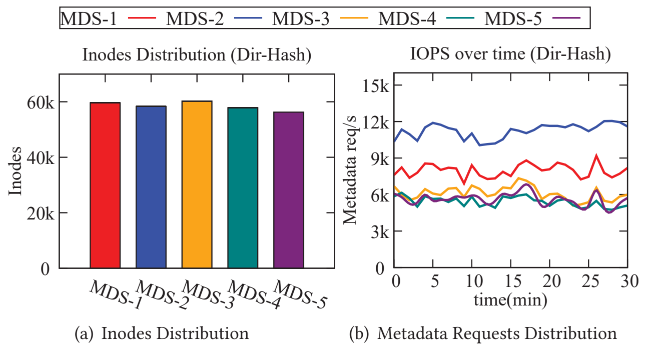


Figure 14: (a) shows the amount of inodes distribution of Web workload by static hashing method, while (b) shows the distribution of runtime metadata requests.

eliminate such bottleneck. There are two major solutions to place metadata on multiple servers and regulate concurrent accesses to them, namely, hash-based mapping and subtree partitioning. Lazy Hybrid [9], PVFSv2 [19], SkyFS [47], CalvinFS [41], and Lustre [24] adopt the first solution and determine the location of metadata by hashing the corresponding file pathname or some other unique identifiers. Although these solutions can evenly distribute metadata across MDSs, they do not preserve locality inherent in file system workloads and fail to adapt dynamics in terms of the MDS cluster expansion and hot-spots of activities in the hierarchical namespace. To benefit a specific workload, where massive metadata creations are passed through a shared directory and the sizes of directories are growing at unprecedented speeds, GIGA+ [30] further improves the performance of hash-based metadata management via the metadata re-distribution and eventual consistency adoption.

To preserve locality while not targeting specific workloads, Ursa Minor [2], NFS [31], and Farsite [3] use subtree partitioning, which assigns the portions of the file system namespace (a.k.a subtrees) to different MDSs. However, due to the static metadata distribution, similar to the aforementioned hash-based mapping, it is challenging for these solutions to dynamically accommodate metadata growth and cluster expansion, since the growth may not keep metadata evenly distributed across MDSs. Furthermore, the static subtree partitioning cannot handle ever-changing client workloads, particularly when load distribution is highly skewed. Handling these dynamics would require manual redistribution of the hierarchical namespace. To address this limitation, CephFS [45] and PanFS[10] employ dynamic subtree partitioning, which improves the static counterpart by periodically migrating subtrees among MDSs according to the load intensity level of the whole MDS cluster. CephFS is now widely used in both academia and industry [4, 11].

Mantle [35] generalizes the metadata load balance problem in DFSs and enables to specify conditions to trigger re-balance. However, without a deep understanding of the significant factors that impact the decisions, it is hard to write suitable policies even with the Mantle framework. We conduct a comprehensive study on the metadata load balance problem and identify the inefficiencies of various load balancers including the one implemented within Mantle. We have shown that our solution significantly outperforms the baselines by timely detecting and reacting to imbalance and making judicious migration decisions, relying on an accurate model.

There is a large body of related work, which consider the organization and storage of metadata. For instance, IndexFS [33] stores file system metadata using LevelDB, and LocoFS[22] further bridges the performance gap between the hierarchical namespace and key-value stores by decoupling metadata dependencies. With the rapid development of CephFS technologies, it has already been incorporated many advanced features, including the ones used in IndexFS such as the dynamic metadata re-balancing and the key-value based out-of-core metadata organization [4]. LocoFS's primary focus is not on improve the metadata load balance. Thus, the goals and design spaces of Lunule are complementary to these work. Along the same lines, CalvinFS[41] and HopsFS [29] leverage relational and NewSQL database for metadata management, respectively. Furthermore, some recent proposals utilize new hardware technology such as RDMA and NVM to either improve the fault tolerance or performance of metadata service [23, 48].

6 CONCLUSION

To address the existing metadata load balance mechanism's inefficiencies, we introduce Lunule, driven by an accurate load monitor module for judging the imbalance degree and its urgency, and enabling workload-aware migration plans. Experimental results show that Lunule outperforms two state-of-the-art baselines with better load balance, higher clustered metadata IOPS, shorter job completion time, and greater end-to-end system performance.

ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their constructive feedback and suggestions. This work was supported by National Nature Science Foundation of China (61772486, 61832011, and 61802358), National Science Foundation (CAREER-2048044, IIS-1838024, and CCF-1756013), and USTC Research Funds of the Double First-Class Initiative (YD2150002006). Cheng Li is the corresponding author.

REFERENCES

[1] Cristina L Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H Campbell. Metadata traces and workload models for evaluating big storage systems. In *UCC*, 2012.

[2] Michael Abd-El-Malek, William V Courtright II, Chuck Cranor, Gregory R Ganger, James Hendricks, Andrew J Klosterman, Michael P Mesnier, Manish Prasad, Brandon Salmon, Raja R Sambasivan, et al. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[3] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 2002.

[4] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *SOSP*, 2019.

[5] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel i/o and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, 2011.

[6] Ambedded Technology. Use cephfs and s3 for medical application. ambedded.com.tw/en/use-case/use-case-03.html, 2020.

[7] A.S.Foundation. Log files - apache http server version 2.4, 2020.

[8] G Borges, S Crosby, and Lucien Boland. Cephfs: a new generation storage platform for australian high energy physics. *Journal of Physics: Conference Series*, 898:062015, 10 2017.

[9] Scott A Brandt, Ethan L Miller, Darrell DE Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *MSST*, 2003.

[10] Welch Brent, Unangst Marc, Abbasi Zainul, G Garth, M Brian, S Jason, Z Jim, and Z Bin. Scalable performance of the panasas parallel file system. In *FAST*, 2008.

[11] Ceph. Ceph user. https://ceph.io/users/.

[12] Ceph. Mantle. https://docs.ceph.com/docs/master/cephfs/mantle/.

[13] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32. USENIX Association, February 2021.

[14] Y. Chen, C. Li, M. Lv, X. Shao, Y. Li, and Y. Xu. Explicit data correlations-directed metadata prefetching method in distributed file systems. *IEEE TPDS*, 2019.

[15] Ceph Community. New in Luminous: CephFS subtree pinning. https://ceph.io/community/new-luminous-cephfs-subtree-pinning/.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

[17] Red Hat. Ceph file system official site. https://ceph.io/ceph-storage/file-system/, 2021.

[18] HDFS. Hadoop file system. http://hadoop.apache.org/.

[19] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *MSST*, 2005.

[20] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 2007.

[21] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC*, 2008.

[22] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. Locofs: A loosely-coupled metadata service for distributed file systems. In *SC*, 2017.

[23] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX ATC*, 2017.

[24] Lustre. Lustre file system. http://www.lustre.org/.

[25] Kirk McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *Commun. ACM*, 2010.

[26] Christopher J. Morrone. Mdtest. https://github.com/MDTEST-LANL/mdtest, 2003.

[27] Theofilos Mouratidis. Ceph MDS balancing issues. https://www.spinics.net/lists/ceph-devel/msg44650.html.

[28] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[29] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *FAST*, 2017.

[30] Swapnil Patil and Garth Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 13, USA, 2011. USENIX Association.

[31] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3: Design and implementation. In *USENIX Summer*, 1994.

[32] Red Hat. The shared file systems service with cephfs through nfs. https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/16.0/html/deploying_the_shared_file_systems_service_with_cephfs_through_nfs/assembly_cephfs-intro, 2020.

[33] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC*, 2014.

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[35] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. Mantle: A programmable metadata load balancer for the ceph file system. In *SC*, 2015.

[36] SoftIron. Ceph comes of age with the explosion of ai and ml data. https://s3.amazonaws.com/cdn-media.softiron.com/doc/SCW+White+Paper_Ceph+comes+of+age.pdf, 2019.

[37] StackHPC Ltd. Stackhpc at the cern ceph day 2019. https://www.stackhpc.com/cern-ceph.html, 2019.

[38] Maosong Sun, Jingyang Li, Zhipeng Guo, Z Yu, Y Zheng, X Si, and Z Liu. Thuctc: an efficient chinese text classifier. *GitHub Repository*, 2016.

[39] V. Tarasov. Filebench. https://github.com/filebench/filebench, 2018.

[40] Telfer, Stig and Garbutt, John. Ad-hoc filesystems for dynamic science workloads. https://indico.cern.ch/event/765214/contributions/3517141/attachments/1908894/3153554/2019-09-17-TelferGarbutt-Ad-hoc-Filesystems.pdf, 2019.

[41] Alexander Thomson and Daniel J Abadi. Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems. In *FAST*, 2015.

[42] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. MAPX: Controlled data migration in the expansion of decentralized object-based storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 1–11, Santa Clara, CA, February 2020. USENIX Association.

[43] Sage A Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California, Santa Cruz, 2007.

[44] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[45] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *SC*, 2004.

[46] Wikipedia contributors. Coefficient of variation. https://en.wikipedia.org/w/index.php?title=Coefficient_of_variation&oldid=981477185, 2020.

[47] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *SC*, 2009.

[48] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *FAST*, 2019.

[49] Shuanglong Zhang, Helen Catanese, and Andy An-I Wang. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *FAST*, 2016.