Origami: Efficient ML-Driven Metadata Load Balancing for Distributed File Systems

Yiduo Wang
China Telecom Cloud Computing Research Institute
Beijing, China
wangyd22@chinatelecom.cn

Wenda Tang
China Telecom Cloud Computing Research Institute
Beijing, China
tangwd1@chinatelecom.cn

Linghang Meng
China Telecom Cloud Computing Research Institute
Beijing, China
menglh1@chinatelecom.cn

Liang Li
China Telecom Cloud Computing Research Institute
Beijing, China
lil225@chinatelecom.cn

Jie Wu
China Telecom Cloud Computing Research Institute
Beijing, China
wujie@chinatelecom.cn

Abstract

Modern distributed file systems (DFSs) rely on metadata server clusters to manage large-scale files and achieve scalability. However, the hierarchical namespace structure and dynamic user workloads pose severe challenges for efficient metadata partitioning and load balancing. Existing approaches primarily focus on identifying and redistributing hot metadata to address imbalances. While these load-balancing strategies offer potential benefits, they often reduce metadata locality, ultimately failing to improve the end-to-end job completion time—a key metric prioritized by users. Although recent research reveals that learning-based approaches are effective in predicting hotspots, they have been shown to be less effective in improving metadata performance. We revisit metadata load balancing strategies and propose a learning-based metadata load balance framework *Origami*, which focuses on minimizing end-to-end job completion time rather than equalizing loads. Origami first decomposes the overhead of metadata operations and assesses the impact of migration decisions on user requests, allowing us to compute the benefits of migration decisions for job completion time when future requests are known. Subsequently, Origami propose the Meta-OPT algorithm to determine near-optimal migration decisions. Finally, we implemented OrigamiFS, on which we collected statistical data to train and validate ML-models capable of predicting migration benefits. By predicting the benefits of migration decisions and employing Meta-OPT to quickly explore nearly optimal migration decisions, Origami makes a better trade-off between load balancing and namespace locality. Our evaluation shows that compared to state-of-the-art methods, Origami increases aggregated metadata throughput by 1.12-2.51× across three real-world workloads, and enhances end-to-end throughput by 1.11-2.02×.

CCS Concepts

• **Software and its engineering → Distributed systems organizing principles**; • **Information systems → Distributed storage**; • **Social and professional topics → File systems management**.

Keywords

Distributed file system, Metadata management, Machine learning

1 Introduction

Modern data centers typically deploy large-scale distributed file systems (DFSs) to manage large amounts of files [9, 19, 24, 29]. To enable efficient file indexing and processing, DFSs often rely on dedicated metadata server (MDS) clusters to store metadata and accelerate metadata operations (e.g., file lookups and directory creation) [11, 25, 42]. As the scale of distributed file systems has grown to encompass hundreds of billions of files, distributing metadata across multiple MDSs for parallel processing has become a common practice. This approach, widely adopted by internet and cloud service providers, is crucial to enhancing scalability and performance [7, 21, 22, 31].

File workloads in modern datacenters have exhibited increasingly metadata-intensive characteristics: over 90% of files and I/O requests are smaller than 1MB, resulting in a continuously rising proportion of metadata operations [26, 39]. Currently, metadata has emerged as the primary bottleneck in DFS, which requires careful partitioning to deliver high-performance metadata services. Unlike storage systems with flat namespaces, such as key-value stores or object stores, balancing the metadata load in hierarchical namespaces(i.e., a directory tree structured as a directed acyclic graph) should be more cautious. First, real-world workloads are diverse and dynamic, often leading to hotspots within hierarchical namespaces, which necessitate timely migration of subtrees. Second, partitioning or migrating metadata across multiple nodes introduces additional overhead for metadata operations: (1) path resolution requires multiple RPCs (remote process calls) to traverse the entire path when nodes along a file path are stored on different servers [26, 28]; (2) metadata operations involving concurrent access across multiple nodes (e.g., directory reads or namespace structure mutations) also incur higher costs [10, 39].

Prior work has explored metadata partitioning from various perspectives. CephFS [43] and HopsFS [28] adopt coarse-grained partitioning to maintain namespace locality, while InfiniFS [26] and CFS [39] focus on minimizing the overhead of fine-grained partitioning. Moreover, Mantle [33] introduces programmability to improve metadata load balancing, and Lunule [38] leverages both temporal and spatial locality to enhance load balancing. These studies have primarily focused on the *popularity* [43] of directories or files (i.e.,

their load levels) and enabling migration between MDSs to achieve balance. More recently, some works have introduced machine learning (ML) to predict future popularity and improve metadata load balancing [8, 40, 41]. In general, existing approaches rely on heuristic algorithms or ML-based methods to predict hotspots, and then use this information to guide metadata partitioning.

However, load balancing in hierarchical file system namespaces is significantly more complex than in flat namespaces. In particular, simply predicting or classifying the metadata as "hot" or "cold" is insufficient to achieve optimal load balancing. We argue that ML methods should be leveraged to identify efficient migration decisions in DFS, but this requires solving the following unique challenges: (1) *trade-offs between the benefits and costs of load balancing*. Metadata migration can disrupt the namespace locality, introducing additional overhead with complex and operation-dependent impacts that vary significantly between different operations and sequences. (2) *finding the optimal balancing strategy is difficult*. Modern DFSs typically have massive metadata, making it difficult to quickly find the optimal migration strategy, and aggressive migration strategy can even significantly reduce the efficiency of metadata services. (3) Finally, *existing DFSs are not designed to be ML-native*, which hinders the collection of relevant feature data and the execution of the learned strategies, further complicating balanace solutions.

Based on these observations, this paper advocates that *evenly distributing metadata load should NOT be the primary optimization objective*. Instead, we propose a novel framework, *Origami*, for training efficient ML-driven metadata load balancing strategies. Origami shifts the focus from precise metadata load prediction to optimizing MDS cluster efficiency and minimizing end-to-end job completion time. The main contributions of this paper consist of:

- **Estimating metadata operation execution time**: We propose an approach to predict the completion time for metadata operations and job, based on a given namespace partition and user requests sequence, which provides a key metric for evaluating balancing strategies's benefit and costs.
- **Finding near-optimal migration decisions**: Inspired by the classic Bélády's algorithm [44], we design *Meta-OPT*, a mechanism capable of identifying near-optimal migration decisions for a given sequence of metadata operations.
- **Training ML methods to optimize metadata**: We design a framework, Origami, with a ML-native distributed metadata service *OrigamiFS*, that automates feature collection from user workload, applies ML-based algorithms, and evaluates their effects on end-to-end job performance.

We evaluated Origami with 5 MDSs and found that Origami achieves a better trade-off between load balance and namespace locality. Specifically, Origami increases metadata throughput to 3.86× that of a single MDS while incurring only a 3.5% increase in forwarded requests. Compared to subtree- and hash-based partitioning approaches, Origami achieves the best overall performance, boosting the cluster's aggregate metadata throughput and end-to-end file throughput by factors of 1.12-2.51× and 1.11-2.02×, respectively, in 3 real-world workloads.

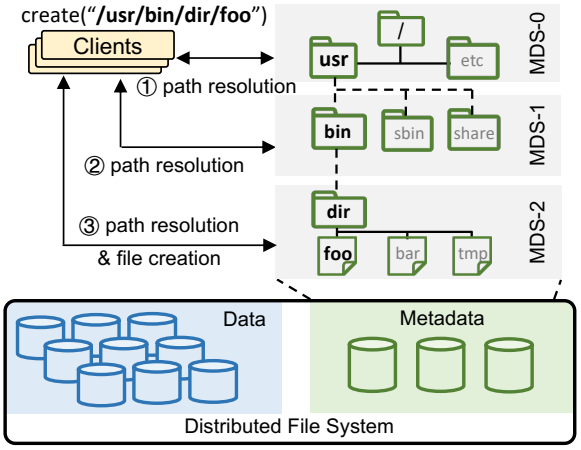


Figure 1: Modern DFS architecture and user workflow.

2 Background and Motivation

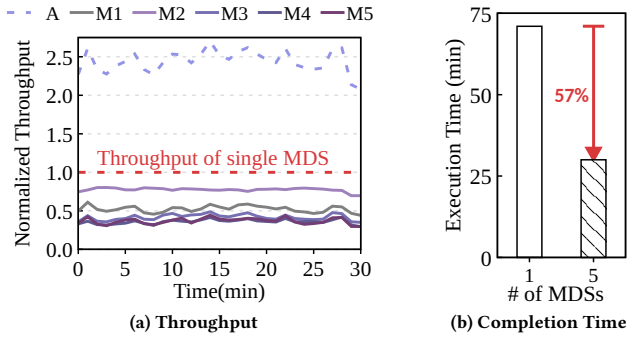
2.1 Distributed Metadata Management

GFS [9] and HDFS [11] introduced a key technique in the evolution of DFS: decoupling metadata from file data. By centralizing metadata management on a dedicated MDS and distributing file data evenly across numerous data servers, DFSs achieve improved scalability. However, as the average file size has decreased from GBs to MBs and the number of files managed by a single DFS has grown to hundreds of billions, a single metadata server is no longer sufficient to meet the demands of processing speed and storage capacity [1, 3, 18, 34]. Recent studies have shown that workloads in the cloud exhibit significant metadata-intensive characteristics, with metadata operations often accounting for more than two-thirds of the workload in most cases, and in some scenarios, even exceeding 90% [39], whic has become a major bottleneck in DFS.

To overcome these limitations, modern DFSs partition the namespace into multiple metadata shards and distribute them across MDSs, enabling parallel processing and further enhancing scalability. Figure 1 illustrates the architecture of modern DFSs, consisting of three components: a metadata cluster for namespace management, a data cluster for file storage, and clients that initiating user requests. In this setup, clients must access metadata before retrieving file data. As shown in the upper half of this figure, a file creation (e.g., "/foo" under "/usr/bin/dir/") requires sequential *path resolution* (i.e., traverse the metadata along the path) to verify directory existence and permissions before execute metadata mutation. This design provides scalability, POSIX [37] compliance, and user programs compatibility but increases metadata overhead due to additional remote procedure calls (RPCs) [2, 13, 45].

2.2 Even Partitioning Considered Harmful

The hierarchical namespace and path resolution process presents significant challenges for metadata load balancing. Unlike many storage systems with flat namespaces (e.g., object stores [5]), evenly distribute metadata in DFS can result in notable performance degradation. To analyze the potential performance degradation caused



(a) Throughput (b) Completion Time

Figure 2: (a) Normalized metadata throughput across 5 MDSs (M1-M5: Per MDS, A: Aggregated) (b) Job completion time with 1 MDS vs. 5 MDSs.

by balancing, we deployed a 5-MDS cluster of CephFS, an open source and widely used DFS, and ran 50 clients to saturate MDSs.

In alignment with prior studies, we replayed a web access workload [4], disabled the data path, and evenly distributed metadata per directory via the built-in CephFS function [6, 38]. Figure 2 compares the performance of a single MDS set-up with a configuration of 5 MDSs. As shown in Figure 2a, the throughput of each individual MDS (5 solid lines) in the 5-MDS configuration is significantly lower than that of a single MDS (red dashed line). Note that this inefficiency is not caused by an insufficient client request load; rather, it is due to the additional execution overhead, which limits the processing capacity of each MDS (see details in § 5.5). Interestingly, even after adding 4 MDSs, the aggregated metadata throughput (purple dashed line) increased by only about 1.4 times. Meanwhile, as shown in Figure 2b, the corresponding job completion time was reduced by merely 57%.

This inefficiency is primarily due to the resolution and processing overhead associated with load balancing, as described in Section 2.1. This suggests that we must approach load balancing with caution, focusing not just on evenly distributing the load, but also on minimizing the associated overhead. It should be noted that our experiments incorporated metadata caching and other optimization techniques to mitigate these issues, but significant inefficiencies persist (see details in § 5.4).

2.3 Limitations of Metadata Partitioning

Recent data centers and clouds statistics reveal that metadata has surpassed data, becoming the main bottleneck for scaling DFS [18, 39]. To address that, modern DFSs have designed various metadata partitioning strategies over the past decade. However, mainstream methods still struggle to effectively balance locality and scalability.

First, *Coarse-grained partitioning* is hard to scale. For example, CephFS's dynamic subtree partitioning [43], a well-known coarse-grained approach, aims to preserve the locality of the namespace by migrating subtrees to other MDS only when load imbalance occurs. Coarse-grained partitioning has been widely embraced by numerous clouds and data centers due to its ability to maintain locality [21, 22, 28]. Although this method effectively reduces the additional metadata overhead, it often leads to metadata hotspots under dynamic workloads, ultimately limiting the overall scalability of the system [33, 38].

Second, *Fine-grained partitioning* increases the metadata overhead. Per-directory partitioning uses hash-based algorithms to evenly distribute metadata across MDSs [23, 25, 29, 35], achieving better load balancing. However, disrupting namespace locality can introduce additional overhead for metadata operations, mainly due to RPC forwarding and distributed coordination [39]. Cloud vendors report that this can lead to latency increasing almost linearly with directory depth, rendering undesirable latency [7, 20, 26].

Recent *ML-based partitioning* strategies aim to predict the future load and migrate files or folders by learning from historical load data. Compared to heuristic strategies, ML-based strategies show potential in flexibility and efficiency. However, existing learned strategies overlook the hierarchical structure of metadata, suffer from low prediction efficiency when handling dynamic loads, and rely heavily on manual optimization, which constrains overall performance improvements [8, 40, 41].

2.4 Challenges of ML-based Balancing

To address the metadata load balancing problem, our goal is to develop an efficient ML-driven metadata load balancing mechanism, building upon existing dynamic subtree and machine learning-based strategies. To achieve this objective, several key challenges must be addressed.

Challenges #1: Measuring appropriate metrics. Simply relying on the number of inodes or the balance of QPS across different MDS is not an appropriate metric, as these measures neither account for the disruption of locality due to load balancing nor can they be mapped to the completion times of user tasks.

Challenges #2: Finding effective migration decisions. The hierarchical structure of the namespace poses significant challenges in the search for optimal migration strategies. The file system namespace is not only deeply layered, exceeding ten levels, which hinders rapid searches, but migration strategies between parent and child nodes can also interfere with each other, and excessive migrations can significantly affect system performance.

Challenges #3: Collecting statistics and validating models. Existing distributed filesystems are not native to ML, making it difficult to collect effective training data and validate the efficacy of migration decisions.

3 Potential Benefits of Balancing

The observations and analysis mentioned above suggest that, in DFS, *even partitioning is NOT suit for metadata load balancing*. To effectively manage metadata, it is crucial to prioritize reducing the end-to-end *Job Completion Time* (JCT) during migration and to train models aimed at minimizing JCT. Determining the JCT prior to job execution is impractical. Therefore, we relax this goal by computing the JCT for a given metadata request and namespace partition, which subsequently guides the learned model to predict the migration benefits. Specifically, our approach, inspired by the Bélády's algorithm and learned cache [44], addresses two key challenges:

Address Challenge #1: Measure the completion time of the user request. We leverage the *request completion time* (RCT) and the corresponding migration benefits, both derived from a given

metadata access sequence and namespace partition. Specifically, we decompose the components of RCT and compute its value for a given namespace partition (§ 3.1).

Address Challenge #2: Find near-optimal decisions quickly. Using the calculated RCT, we proceed to determine nearly optimal metadata migration choices to reduce the total job completion time, which then informs our partitioning strategies developed(§ 3.2).

3.1 Measure RCT Instead of Balance

Dive into metadata overhead. The main challenge in computing RCT is that metadata overhead depends on both network/load conditions and the hierarchical namespace structure. Under different partitions, the same request may access varying numbers of MDSs. Fortunately, metadata operations exhibit fixed patterns, enabling us to decompose the overhead. Specifically, for a metadata request with a path length of k that is distributed among m distinct metadata partitions, we assume that the processing overhead (e.g, time of resolving path, creating file and updating parent attributes) of metadata is T_{meta}, the queuing time on each partition is Q, and the network request access time is RTT. Consequently, RCT can be expressed using the following formula:

$$RCT = T_{meta} + m \cdot RTT + \sum_{i=1}^{m} Q_i \tag{1}$$

For metadata requests, both the queueing time and the network request time can be computed based on the average network latency and the machine load level, whereas the composition of T_{meta} varies depending on the type of operation. In addition, apart from the path resolution time T_{path}, T_{meta} is also influenced by the metadata partitioning strategy, which makes it difficult to predict.

Calculate additional overhead. Fortunately, we have found that primary metadata requests can be categorized into three types, with their execution times discussed separately. Specifically, T_{meta} has a fixed baseline cost, and the additional variable overhead (due to metadata partitioning) requires a case-by-case analysis. We categorize metadata into three types: list directory (short in lsdir), namespace mutation (e.g. `create`, `rmdir`, short in ns-m), and others unaffected operations. For `lsdir`, migrating the sub-files/directories to i other MDSs introduces an additional $i \cdot RTT$. Furthermore, distributing the parent directory and the target file/directory across different MDSs incurs additional distributed coordination overhead for namespace mutation operations. Therefore, let each inode's read time be T_{inode}, the operation execution time be T_{exec}, and the additional coordination time for distributed transactions be T_{coor}, $\mathbb{I}$ mean the indicator function, then we can calculate the execution time of metadata requests after migration:

$$T_{meta} = T_{inode} \cdot (m+k) + T_{exec} + \begin{cases} RTT \cdot i, & \texttt{lsdir} \\ T_{coor} \cdot \mathbb{I}(i>0), & \text{ns-m} \\ 0, & \text{others} \end{cases} \tag{2}$$

3.2 Estimating the Benefits of Migration

In this section, we first explore how to compute JCT and evaluate the benefits of migration decisions. Based on this, we design a Meta-OPT algorithm to identify near optimal migration strategies.

Algorithm 1: Meta-OPT Algorithm

Input : A sequence of metadata operations N;
A list of MDS $\vec{M} = \{m_i\}$; A threshold Δ ;
Output : A list of migration decisions $\vec{D}$

1 **repeat**
2 $T \leftarrow JCT(N, \vec{M})$;
3 $max\_benefit \leftarrow 0$; $best\_decision \leftarrow \emptyset$;
4 **foreach** $m_i \in \vec{M}$ **do**
5 **foreach** *subtree* $s \in m_i$ **do**
6 **foreach** $m_k \in \vec{M} \setminus \{m_i\}$ **do**
7 $\vec{M'} = \vec{M}.migrate(s, m_i, m_k)$;
8 $T' = JCT(N, \vec{M'})$;
9 **if** $T' < T \& (m'_k.rct - m'_i.rct) < \Delta$ **then**
10 $benefit \leftarrow T - T'$;
11 **if** $benefit > max\_benefit$ **then**
12 $max\_benefit \leftarrow benefit$;
13 $best\_decision \leftarrow (s, m_i, m_k)$;
14 $\vec{M} = \vec{M}.migrate(best\_decision)$;
15 $\vec{D}.push(best\_decision)$;
16 **until** $max\_benefit < threshold$;
17 **return** $\vec{D}$;

Estimate the job completion time. Based on the previous analysis, we can compute the total cost and the distribution of RCT across different metadata partitions. Furthermore, as demonstrated in Lunule [38], load balancing yields substantial benefits primarily under high-load, allowing us to focus on scenarios where the most loaded MDS nears its capacity limit. Consequently, JCT can be approximated as a bin-packing problem: treating MDSs as multiple bins, with the JCT estimated by the capacity of the largest bin. Building on this, given the access sequence N, we can estimate the JCT for the entire task in a way that, while not entirely precise, remains simple and efficient: (1) calculate the total costs of requests processed by each MDS (denoted as $m.rct$) in the given request sequence[1]. (2) sum the RCTs of requests processed by each MDS, record the highest value as JCT.

Seek approximately optimal decisions. Here, we present the Meta-OPT algorithm, designed to efficiently find an approximately optimal migration decision given a known future metadata operations sequence N and the current MDS state $\vec{M}$. Details are shown in Algorithm 1. The algorithm iteratively finds a list of migration decisions that maximize the benefit and minimize the overall completion time T of future metadata operations $\vec{N}$ (lines 2-3). This process continues until the benefit drops below a predefined threshold (line 16). Specifically, it traverses all subtrees within each MDS (lines 4-5) and calculates the completion time T' if the subtree is migrated to another MDS (lines 6-8). If T' is smaller than T, indicating a reduced overall completion time, the difference is recorded as $benefit$ (lines 9-10). It should be noted that in order to prevent

[1] Origami will estimating $Tqueue$ and $Tcoor$ via historical sampling data.

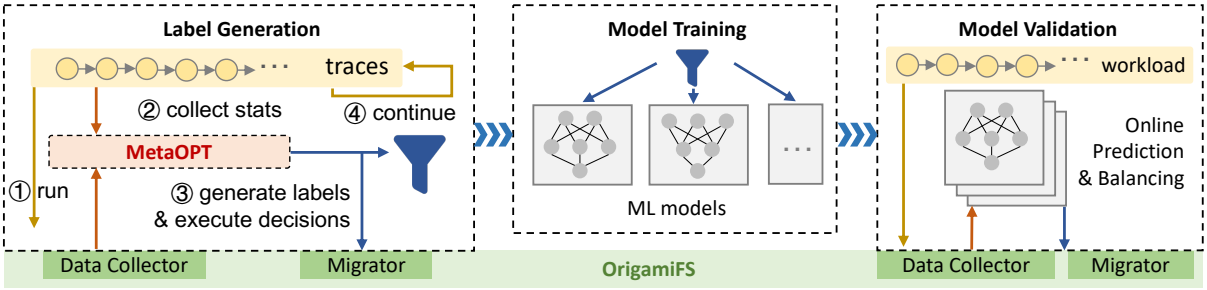


Figure 3: The architecture of Origami for training efficient ML-based metadata balancing models.

new imbalance caused by the migration, we also require the imbalance after migration is less than a specified threshold Δ in line 9. During each iteration, the decision with the maximum *benefit* is selected (lines 11-13). After completion of the iteration, the selected decision is executed, added to the migration decision list $\vec{D}$, and the algorithm proceeds to the next iteration (lines 14-15).

It is important to note that, to find the optimal migration solution, we need to enumerate all possible subsets of subtrees and identify the set that yields the maximum benefit. However, exhaustively evaluating all possible combinations of subtrees is computationally infeasible. Therefore, Algorithm 1 adopts a greedy method. It makes a sequence of local decisions, each time selecting the subtree whose migration offers the greatest immediate benefit. Under this approach, once a subtree s is selected and migrated, any subtrees nested within s are no longer considered for migration. If there exists a set of disjoint subtrees $k_1, k_2, \ldots, k_N$ within s whose total migration benefit exceeds that of migrating s as a whole, then the solution provided by Algorithm 1 is suboptimal. Nevertheless, we can prove that the gap between Algorithm 1 and the optimal solution is less than Δ, as shown in the following theorem.

Theorem 1. *Let b_0 denote the benefit of migrating a subtree s. Assume that subtrees $k_1, k_2, \ldots, k_N$ are N disjoint subtrees nested within subtree s, and let b_1 be the benefit of migrating $k_1, k_2, \ldots, k_N$. Under the conditions specified in Algorithm 1, we have $b_0 - b_1 > -\Delta$.*

The proof of this theorem is provided in Appendix A. This theorem shows that even under suboptimal conditions, Algorithm 1 maintains performance within a controlled margin of error.

4 Toward Efficient ML Balancing

Address Challenge #3: Building a framework for training efficient metadata balancing models. In this section, we initially outline Origami, the system framework crafted for training efficient learned metadata balancing models (§ 4.1). After that, we present the architecture of OrigamiFS in (§ 4.2), the prototype metadata service within Origami, and detail the complete workflow for training metadata balancing models in (§ 4.3).

4.1 System Overview

Origami is supported by two key components: the Meta-OPT (§ 3.2) algorithm and a lightweight distributed metadata service OrigamiFS. Meta-OPT, an implementation of Algorithm 1, guides the ML model in making migration decisions. OrigamiFS, implemented as a prototype distributed metadata service, generates training features and evaluates the effectiveness of models. As illustrated at the bottom of Figure 3, we introduced two components for each MDS to further support the Meta-OPT algorithm and ML models in optimizing migration policies:

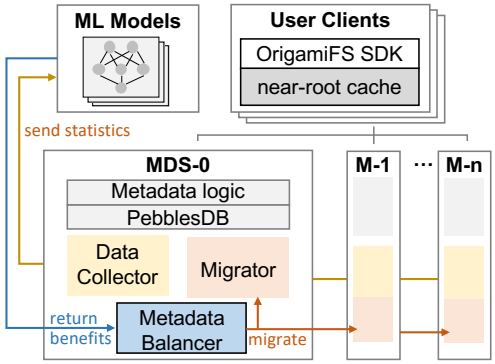


Figure 4: The architecture of OrigamiFS.

- **Data Collector to dump the runtime namespace state.** This component outputs the metadata partition and attribute statistics. Unlike snapshots, it focuses solely on feature data for ML training (e.g., load in last epoch, depth, `links`) and metadata required for Meta-OPT calculations. In addition, modern DFSs use directories as the basic unit for load balancing, allowing us to omit file-level metadata and significantly reduce the data collection overhead.
- **Migrator to execute external migration decisions.** Modern DFS often integrate metadata load balancing as built-in logic or pluggable functions [33]. To better support ML models, Migrator enables external algorithms (e.g., Meta-OPT and ML models), to provide migration decisions (e.g., path, source MDS, destination MDS) in a pipeline manner.

4.2 OrigamiFS Architecture

We implemented OrigamiFS in the Go programming language, which consists of the following parts:

Metadata Cluster. As Figure 4 shows, the metadata service within OrigamiFS is composed of MDS units ranging from 0 to n. Each

MDS stores its individual inodes as key-value pairs in a local PebblesDB [30]. Using the inode number of the parent directory combined with the file name as an index, aligned with state-of-the-art studies [12, 26, 39].

In the initial state, OrigamiFS stores all metadata on the MDS numbered 0. During each epoch, all MDSs send metadata statistics to the ML model through *Data Collectors*. The ML model will send the predicted benefits to MDS-0's Metadata Balancer, which employs the same load monitoring and rebalancing trigger mechanism as Lunule, but with a different rebalancing algorithm. Traditional migration algorithms generally require using bin-packing-like methods to select subtrees based on load differences between MDSs. In contrast, the rebalancing algorithm of OrigamiFS is much more intuitive: MDS-0 simply greedily selects the subtree with the highest benefit and uses the *Migrator* to migrate it to the lightly loaded MDS, repeating this process until the migration benefits of all subtrees fall below a specified threshold.

Clients. We designed a OrigamiFS SDK that enables clients to convert file system calls into corresponding metadata operations directed to the MDSs. Similar to the workflow illustrated in Figure 1, the client first resolves the path recursively and then issues the corresponding metadata operations. To address the well-known near-root hotspot problem and reduce path parsing overhead, we introduced a configurable near-root metadata cache, which stores metadata entries whose depth is less than a predefined threshold. Although the near-root metadata cache is a straightforward design, it is highly effective: since near-root metadata typically constitutes less than 1% of the entire file system [26], this approach substantially mitigates the near-root hotspot issue while avoiding the significant consistency overhead associated with cache synchronization or lease management.

4.3 Origami Workflow

As shown in Figure 3, we train and validate efficient ML-based load balancing models through the following process:

Label generation. We begin by collect the access traces from real-world workloads, and replay the metadata operations on OrigamiFS (①). After each epoch[2] , the Data Collector dumps metadata statistics (②) from OrigamiFS. Next, Meta-OPT extracts features from these statistics, performs normalization, calculates the migration benefit for each metadata subtree, and uses these benefits as training labels step by step. Migration decisions with high estimated benefits are then applied to rebalance OrigamiFS (③). The above process is repeated iteratively to progressively enrich the training dataset (④).

Model training. After extracting feature and label data using OrigamiFS and MetaOPT, we trained multiple models offline. As Table 1 shows, for each directory, OrigamiFS outputs two types of metadata statistics: (1) *namespace structure statistics of this directory*, including directory depth, number of sub-files, and number of sub-directories; (2) *metadata access history in last epoch of this subtree*, including the total number of metadata read operations (e.g., `open()`, `stat()`.) and metadata write operations (e.g., `create()`, `mkdir()`.) in the previous epoch. Note that we refer to the access history of the subtrees instead of that of the directory itself, since migration is conducted on a subtree level. For namespace structure statistics, we normalize using the maximum value of the corresponding items. For metadata history, we normalize by taking the total count of metadata operations from the previous epoch. Furthermore, we incorporate additional features, such as the ratio of subdirectories to subfiles and the proportion of write meta-operations to the total number of meta-operations.

[2]In the experiments of this paper, epoch is set to 10 seconds.

Table 1: Training features and the Gini importance rank obtained after training with LightGBM.

Type	Feature	Normalization	GI Rank
Namespace Structure	depth	by the max value	7
	# sub-files		1
	# sub-dirs		4
Metadata History	# read	by # total access in last epoch	6
	# write		2
Derived Feature	read-write ratio	raw	6
	dir-file ratio		2

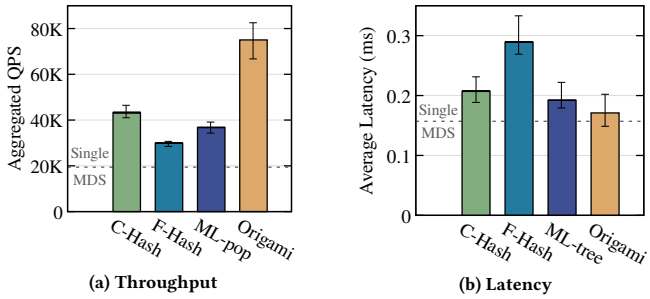
We developed a Python module and trained regression models to predict benefits. We compared LightGBM [15], GBDT [16], and a MLP [36] with 4 hidden layers. Interestingly, we found that although there were slight differences in prediction accuracy among the three models, the migration decisions produced when the predicted results were fed into the metaOPT algorithm were remarkably similar. This occurred because each model succeeded in pinpointing subtrees with notably higher migration benefits, which is crucial for the execution of the migration algorithm, while the migration algorithm filtered out subtrees with lower benefits, thus minimizing the effect of prediction inaccuracies. Consequently, we chose to implement the lightGBM model due to its minimal prediction overhead, with 400 rounds of boosting and 32 leaves. Table 1 lists the specific indicators used for the training, the corresponding normalization methods, and the Gini importance [27] obtained after training.

Model validation. A key challenge of learning-based metadata balancing is that the accuracy of the model does not directly represent the improvement in system performance. Fortunately, OrigamiFS enables the online validation of different models. During runtime, OrigamiFS asynchronously outputs metadata via `Data Collector`, which the trained models use as feature input to predict migration benefits. The high benefit migration decisions are then applied to OrigamiFS through the `Migrator`, enabling online metadata rebalancing. The above work allows us to evaluate the overall performance optimization of metadata cluster directly, rather than relying solely on accuracy metrics.

5 Evaluation

5.1 Experiment Setup

Hardware configurations. We validated Origami by developing a prototype and implementing OrigamiFS with 5 MDS nodes, running 5 client nodes to saturate the capacity of MDS. The experiments were conducted on a 10 node Kubernetes cluster, where each node was equipped with 8 CPU cores, 64GB of RAM, and a 2TB NVMe SSD for metadata processing and storage.



(a) Throughput **(b) Latency**

Figure 5: Aggregate throughput under high load and average latency under single thread of different balancing methods.

Baseline methods. We implemented state-of-the-art load balancing strategies in Origami for comparison, covering two categories:

- Hash-based partitioning: We reproduced two widely used hash-based strategy in recent research and production systems: The coarse-grained approach, akin to *HopsFS* [12], applies hashing only to the upper levels of the namespace, which we label as **C-Hash**. Conversely, the fine-grained approach hashes all directory, which we denote as **F-Hash**, used in *Tectonic* and *InfiniFS* [26, 29].
- Popularity-based ML methods: We also reproduced the latest ML-driven metadata load balancing method [41], using subtrees as the basic granularity, and using the LightGBM model to predict access popularity and guide load balancing, named **ML-tree**.

For hash-based methods, we partitioned the metadata before conducting the evaluation. For both ML-tree and Origami, statistics were collected after each epoch, using Lunule's algorithm [38] to trigger load rebalancing. To ensure a fair comparison, the *near-root cache* was activated for all strategies. A standalone MDS was used as the baseline for performance measurement.

Workload configurations. We selected the following 3 real-world workloads that have been used in recent metadata studies:

(1) *Trace-RW*: A large compilation task consisting of numerous complex metadata operations [33].
(2) *Trace-RO*: A web application access trace, which only includes read-type operations, exhibits a significant skew and extends to a considerable depth [38].
(3) *Trace-WI*: A write-intensive trace from a distributed file system on the cloud, which we reproduced based on the characteristics described in the paper [39].

For comparative analysis purposes, we use Trace-RW with only the metadata function active, allowing us to evaluate and examine the metadata performance and load balancing of various baselines from § 5.2 to § 5.5. Finally, in § 5.6, we activated the data path and compared all methods with three real-world workloads.

5.2 Overall Performance

We begin our analysis by evaluating the overall metadata performance using Trace-RW. For each approach, we initiate 50 client threads to fully utilize the metadata service and activate the load balancing mechanism. Subsequently, we measure the average aggregated metadata throughput post-rebalancing. Following this, we rerun the workload with a single thread to compare the changes in latency under different methods, thereby evaluating the extent to which each balancing approach disrupts namespace locality.

Aggregated throughout under high load. Figure 5a shows the aggregate metadata throughput using different balancing strategies. When a single MDS processes the metadata operations for Trace-RW, OrigamiFS achieves a metadata throughput of 19.4k/s. By distributing the metadata across multiple MDSs, C-Hash can utilize multiple MDSs in parallel, increasing the throughput by 2.23×, which is consistent with our observations in § 2.2. However, fine-grained hashing does not yield additional performance gains: we found that F-Hash's throughput decreased by 31.0% compared to C-Hash. This is because, although F-Hash enables a more balanced distribution of load across multiple MDSs, its significant disruption to namespace locality results in overhead that outweighs the benefits of load balancing in terms of metadata throughput. The results for ML-tree fall between the two, achieving 1.89× the throughput of a single MDS. We found that although ML-tree can predict hot directories, it tends to overlooks the negative impact of migration operations. Moreover, popularity-based balancing strategies often make aggressive migration decisions [38], which can hinder the full utilization of cluster resource. Ultimately, Origami is able to accurately identify subtrees with higher migration benefits and strikes a better trade-off between metadata load and locality preservation. As a result, it increases metadata throughput to 75.0k with five MDSs, which is 3.86× that of a single MDS and 1.73× as high as the best-performing baseline, C-Hash.

Average latency under single thread. We then re-ran Trace-A using a single thread to quantify the degree of disruption to namespace locality under different strategies. As shown in Figure 5b, single MDS, which does not involve load balancing, achieved the lowest latency since all operations could be completed with a single RPC without additional overhead. In contrast, C-Hash and F-Hash have increased the latency of metadata operations by 43.9% and 89.1%, respectively. This is because as the number of hash operations increases, the average number of forwarding steps required for each metadata operation also increases, which degrades the performance of metadata under low-load conditions. In contrast to hashing methods, ML-tree and Origami do not migrate metadata too aggressively, resulting in latency increases of 29.3% and 24.2% compared to a single MDS.

The overall performance experiments validate that Origami outperforms other methods by precisely predicting migration benefits, *achieving a better trade-off between metadata load balancing and preserving namespace locality.* Origami maximizes metadata throughput during high loads while minimizing performance degradation during low loads.

5.3 Balance Analysis

Furthermore, we evaluate load balancing using the *Imbalance Factor* [38], which ranges from 0 to 1, with higher values indicating greater imbalance. For example, in a 5 MDS cluster, an Imbalance Factor of 1 means that all requests go to a single MDS. To dive into the namespace partition, we further extend the imbalance factor from *QPS* to other metrics: *RPCs* (number of RPCs handled per epoch), *Inodes* (number of metadata entries stored), and *BusyTime* (total metadata processing time per epoch).

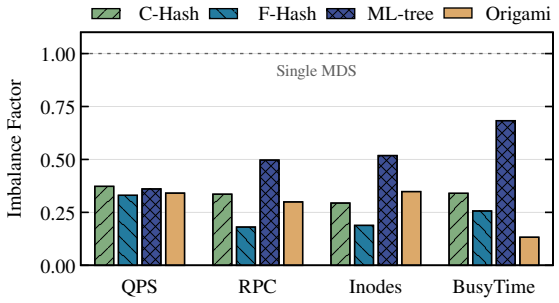


Figure 6: The imbalance factors of different balancing strategies on 4 metrics (lower means better balance).

Table 2: Aggregated metadata throughput and per-request RPC count: comparison with and without metadata cache.

	Throughput		# RPC per request	
	w/o cache	w/ cache	w/o cache	w/ cache
C-Hash	32.8±3.3 k	46.0±3.0 k	2.23±0.03	1.54±0.02
F-Hash	22.5±2.0 k	30.0±1.3 k	2.87 ±0.11	2.27 ±0.07
ML-Tree	26.7±3.7 k	38.6±2.3 k	1.62±0.02	1.17 ±0.02
Origami	39.3±3.7 k	78.9±7.8 k	1.85 ±0.02	1.04 ±0.01

We first analyze the balance of metadata requests using the imbalance factor. As Figure 6 shows, we found that even the most imbalanced C-Hash managed to keep the imbalance factor at a relatively low value. Although F-Hash achieved the best balancing effect, the imbalance factor only decreased from 0.37 to 0.33. The values of the imbalance factor for ML-tree and Origami were intermediate. Furthermore, we dive into the distribution of metadata RPC and Inodes. Similarly to QPS, F-Hash attained the lowest imbalance factor values in both of these metrics, indicating that the hashing method can effectively distribute the metadata evenly, albeit at a considerable performance downgrade. These results validate our conclusion: evenly partitioning metadata is not the optimal strategy, and we should trade-off should be made between namespace locality and load balancing carefully.

To fully understand the load balancing of the MDS cluster, we measured the cumulative time that each MDS was spent processing client metadata requests over every epoch. For the hash-based methods, the balance of BusyTime was similar to previous findings. Interestingly, we found that ML-tree, due to its less aggressive partitioning of metadata, resulted in some MDSs having very low loads, which led to the highest imbalance factor. Surprisingly, Origami exhibited a very low imbalance factor in BusyTime, reducing it by 48.3% compared to F-hash, indicating that all its MDSs utilized the resource to process metadata at a relatively high level. This observation highlights one reason for Origami's superior performance: *Ensuring all MDSs busy is more efficient than evenly partitioning.*

5.4 Metadata Cache Analysis

In this section, we assess the effects of caching on various systems by toggling the near-root cache on and off, and we analyze how their choices differ in the selection of subtrees for migration.

Improving aggregated throughput. Table 2 shows that near-root caching significantly improves performance across all baselines by reducing path resolution overhead with minimal synchronization, thus easing MDS pressure. Without caching, C-Hash, F-Hash, and ML-tree only improve single MDS throughput by 68.8%, 19.9%, and 42.2%, far below the ideal 5× scaling. In contrast, Origami achieves 2.09× throughput without caching, demonstrating more efficient hardware utilization despite root node hotspots. With near-root caching enabled, the throughput of C-Hash, F-Hash, and ML-trees increased by 40. 5%, 33. 3%, and 44. 7%, indicating limited scaling. In contrast, Origami saw a 100.7% throughput increase, as its performance is mainly limited by near-root node overload rather than path resolution or partition imbalance.

Reduce path resolution overhead. We further measured average RPCs per metadata operation with and without caching. Without caching, C-Hash and F-Hash see a significant increase in average RPCs per request, reaching 2.23 and 2.87, respectively. In contrast, the ML-tree approach is more conservative, increasing the RPC forwarding count by only 0.617×, albeit at the cost of less optimal load balancing. Origami strikes a balance between these extremes, increasing the RPC count by 0.85× while still delivering overall performance improvements. After enabling the cache, RPCs per metadata request decreased by 0.68 to 1.17 for the baseline, but high throughput and low forwarding overhead cannot be achieved simultaneously. Surprisingly, Origami 's extra RPC per request fell to just 0.035 after caching, outperforming all baselines.

To understand this advantage, we analyzed Origami's migration decisions and found that it has a particular inclination toward migrating two types of subtree: (1) *subtrees that are near the root node and have a high load*, which can significantly improve the balance of the cluster with just a single migration; (2) *subtrees that are far from the root node and write-intensive*, which migration only impacts a small amount of metadata operations, but yield substantial balancing benefits. Therefore, the near-root cache significantly benefits Origami, as most migrations occur in cached areas. Each metadata operation adds only 0.03 additional RPCs for path resolutions, making the overhead from migrations negligible compared to the benefits.

5.5 Efficiency and Scalability

Higher Efficiency. In Figure 7, we show the efficiency for the first 15 minutes of each strategy. Although hash-based techniques enable parallel processing of metadata from the beginning, their efficiency is considerably worse compared to a single MDS setup. This is due to the high volume of forwarding requests that must be handled and the difficulty in achieving ideal balancing. The other two systems gradually migrate subtrees between MDSs. However, ML-tree faces significant extra overhead to achieve load balancing. In contrast, Origami efficiently and progressively transfers metadata with minimal degradation in efficiency.

Better Scalability. We compare the scalability by measuring the aggregated throughput as the number of MDSs increases from 2 to 5, with all results normalized to the performance of a single MDS. Since balance and efficiency are difficult to trade off, none of the baseline strategies scales effectively. For F-hash with 4 MDS, although hashing improves balance, this benefit is offset by the

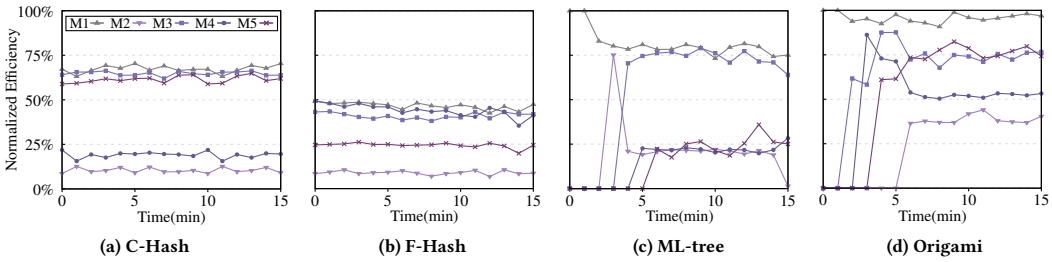


Figure 7: Efficiency comparison, where efficiency refers to the proportion of time each MDS spends processing metadata, normalized to a single MDS setup.

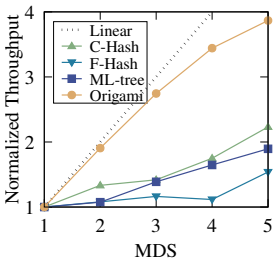


Figure 8: Scalability Comparison.

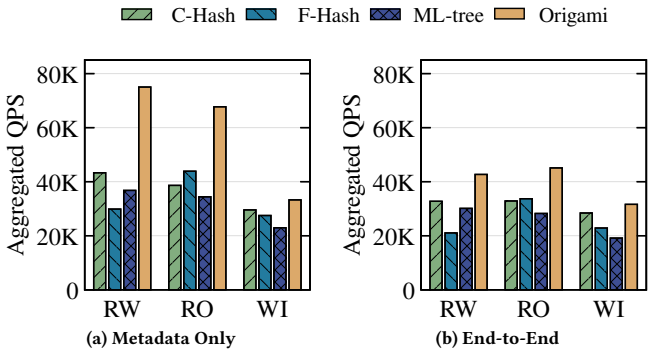


Figure 9: The aggregated throughput for three real-world traces, both w/o and w/ the data path.

overhead from reduced locality. However, Origami demonstrates a distinct performance characteristic, as aggregate throughput with three MDSs reaches 2.7 times that of a single MDS, showing nearly linear scalability. As more MDSs are added, this trend slows slightly due to the increased overhead associated with finer-grained load balancing. In general, Origami achieves near-linear scalability.

5.6 Real-world Workload Results

We replayed traces from 3 real-world workloads with distinct characteristics: Read-Write, Read-Only, and Write-Intensive. We first measured the throughput focus soley on metadata, then enabled the data path and evaluated the end-to-end filesystem throughput.

First, Figure 9a presents a comparison of metadata throughput. Compared to baseline, Origami consistently achieves the highest throughput, with improvements ranging from 12.5% to 102.9%. Although the applicability of different baseline strategies varies between traces, Origami shows improvements of 73. 3%, 54. 3%, and 12. 5% over the second-best baseline, respectively. Origami performs worst on Trace-WI due to the highly dynamic and skewed load, which complicates balancing; however, it still shows a significant improvement over the baseline strategy. Next, we present the end-to-end throughput after enabling the data path, as shown in Figure 9b. Origami still delivered the best performance, increasing the metadata throughput of the second-best baseline from 1.11× to 1.37×. The absolute value of end-to-end data throughput is somewhat lower compared to metadata throughput, as expected. Moreover, if we allocate additional hardware resources to data service–as is common in production systems–it can be anticipated that Origami could further increase the end-to-end performance.

6 Related Work

Metadata partitioning and load balancing. Modern DFSs generally separate metadata from data and distribute metadata across multiple MDSs to achieve scalability. Lustre, InfiniFS, Tectnoic [25, 26, 29] hashes metadata based on identifiers like file and directory names to balance inodes, but the performance overhead is not negligible. The dynamic subtree partitioning employed by systems such as CephFS, IndexFS, HopsFS, and FileScale [21, 28, 32, 42] requires expert tuning to balance metadata. Origami adopts the basic idea of dynamic subtree partitioning but introduces ML techniques to identify subtrees with higher migration benefits, thus striking a balance between load balancing and locality. Moreover, Mantle [33] improves load balancing via programmable interfaces, whereas Lunule [38] incorporates the spatiotemporal aspects of locality, which collectively strengthen Origami.

Machine learning for storage system. DeepHash and LDPP [8, 40] adopt ML to identify the popularity of metadata and improve load balancing. Furthermore, AdaM [14] uses reinforcement learning to adaptively refine load balancing, whereas LoADM [41] concentrates on finding hot directories for migration. The primary concept of Origami is to identify migration benefits instead of just directory popularity, aiming to optimize balance gains while protecting locality. Moreover, inspired by ML-driven caching solutions such as Balleen and GL-cache [44, 47], Origami aims to shorten the user job completion time rather than evenly partition.

Scaling metadata up. Recent research has explored scaling metadata with new hardware like persistent memory and SmartNICs [10, 17, 46], or reducing the coordination overhead of cross-server metadata operations [10, 26, 39]. In contrast, Origami focuses on scaling out metadata, complementing these strategies.

7 Conclusion

The hierarchical namespace of DFS presents significant challenges for effectively balancing metadata. Traditional methods, while effective in evenly distribute metadata, often disrupt namespace locality and fail to optimize job completion time. This paper proposes Origami, a framework designed to train ML-guided metadata load balancing with the aim of maximizing the migration benefit in user

job completion times. Origami utilizes the Meta-OPT algorithm to identify strategies that enhance migration benefits and trains and validates ML models on OrigamiFS. Compared to hash and subtree strategies, Origami can achieve a favorable balance between load balancing and namespace locality, which improves metadata throughput by up to 151. 3%, and increases end-to-end filesystem throughput by 1.11-2.02×.

References

[1] Cristina L Abad, Huong Luu, Nathan Roberts, et al. Metadata traces and workload models for evaluating big storage systems. In *IEEE UCC'12*, 2012.

[2] Michael Abd-El-Malek, William V Courtright II, Chuck Cranor, et al. Ursa Minor: Versatile cluster-based storage. In *FAST'05*, 2005.

[3] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the metadata wall. In *PDSW'11*, 2011.

[4] A.S.Foundation. Log files - apache HTTP server version 2.4, 2020.

[5] AWS. Cloud Object Storage - Amazon S3 . https://aws.amazon.com/s3/, 2006. Accessed May 1, 2025.

[6] Ceph Community. CephFS subtree pinning. https://ceph.io/community/new-luminous-cephfs-subtree-pinning/, 2017. Accessed May 1, 2025.

[7] Chao Dong, Fang Wang, Yuxin Yang, et al. Low-latency and scalable full-path indexing metadata service for distributed file systems. In *ICCD'23*, pages 283–290. IEEE, 2023.

[8] Yuanning Gao, Xiaofeng Gao, Ruisi Zhang, and Guihai Chen. An end-to-end learning-based metadata management approach for distributed file systems. *IEEE TC*, 71(5):1021–1034, 2021.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP'03*, 2003.

[10] Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu. SingularFS: A billion-scale distributed file system using a single metadata server. In *ATC'23*, pages 915–928, 2023.

[11] Apache Hadoop. Hadoop distributed file system. http://hadoop.apache.org, 2006. Accessed May 1, 2025.

[12] Hops Hadoop. Hopsfs 3.2.0.4. https://github.com/hopshadoop/hops/tree/3.2.0.4, 2021. Accessed May 1, 2025.

[13] Jan Heichler. An introduction to BeeGFS, 2014.

[14] Xiuqi Huang, Yuanning Gao, Xinyi Zhou, et al. An adaptive metadata management scheme based on deep reinforcement learning for large-scale distributed file systems. *IEEE/ACM TON*, 31(6):2840–2853, 2023.

[15] Guolin Ke, Qi Meng, Thomas Finley, et al. Lightgbm: A highly efficient gradient boosting decision tree. *NeurIPS'17*, 30, 2017.

[16] Guolin Ke, Zhenhui Xu, Jia Zhang, et al. Deepgbm: A deep learning framework distilled by gbdt for online prediction tasks. In *KDD'19*, pages 384–394, 2019.

[17] Jongyul Kim, Insu Jang, et al. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *SOSP'21*, pages 756–771, 2021.

[18] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, et al. Measurement and analysis of large-scale network file system workloads. In *ATC'08*, 2008.

[19] Qiang Li, Lulu Chen, Xiaoliang Wang, et al. Fisc: A large-scale cloud-native-oriented file system. In *FAST'23*, pages 231–246, 2023.

[20] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. LocoFS: a loosely-coupled metadata service for distributed file systems. In *SC'17*, 2017.

[21] Gang Liao and Daniel J Abadi. FileScale: Fast and elastic metadata management for distributed file systems. In *SoCC'23*, pages 459–474, 2023.

[22] Haifeng Liu, Wei Ding, Yuan Chen, et al. CFS: A distributed file system for large scale container platforms. In *SIGMOD'19*, pages 1729–1742, 2019.

[23] Jiaxi Liu, Renxuan Wang, Xiaofeng Gao, Xiaochun Yang, and Guihai Chen. Anglecut: A ring-based hashing scheme for distributed metadata management. In *DASFAA'17*, pages 71–86. Springer, 2017.

[24] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, et al. DAOS and friends: a proposal for an exascale storage system. In *SC'16*, pages 585–596. IEEE, 2016.

[25] Lustre. Lustre metadata service. https://wiki.lustre.org/Lustre_Metadata_Service_(MDS), 2017. Accessed May 1, 2025.

[26] Wenhao Lv, Youyou Lu, Yiming Zhang, et al. InfiniFS: An efficient metadata service for large-scale distributed filesystems. In *FAST'22*, pages 313–328, 2022.

[27] Stefano Nembrini, Inke R König, and Marvin N Wright. The revival of the gini importance? *Bioinformatics*, 34(21):3711–3718, 2018.

[28] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using newsql databases. In *FAST'17*, pages 89–104, 2017.

[29] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's Tectonic filesystem: Efficiency from exascale. In *FAST'21*, pages 217–231, 2021.

[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 497–514, 2017.

[31] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure Data Lake Store: a hyperscale distributed file service for big data analytics. In *SIGMOD'17*, pages 51–63, 2017.

[32] Kai Ren, Qing Zheng, Swapnil Patil, et al. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14*, 2014.

[33] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, et al. Mantle: A programmable metadata load balancer for the Ceph file system. In *SC'15*, 2015.

[34] Konstantin V. Shvachko. The exabyte club: Linkedin's journey of scaling the hadoop distributed file system. https://engineering.linkedin.com/blog/2021/the-exabyte-club-linkedin-s-journey-of-scaling-the-hadoop-distr, 2021. Accessed May 1, 2025.

[35] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. SoMeta: Scalable object-centric metadata management for high performance computing. In *CLUSTER'17*, pages 359–369. IEEE, 2017.

[36] Hind Taud and Jean-Franccois Mas. Multilayer perceptron (mlp). In *Geomatic approaches for modeling land change scenarios*, pages 451–455. Springer, 2017.

[37] Stephen R. Walli. The POSIX family of standards. *ACM Stand.*, 3(1):11–17, 1995.

[38] Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. Lunule: an agile and judicious metadata load balancer for CephFS. In *SC'21*, pages 1–16, 2021.

[39] Yiduo Wang, Yufei Wu, Cheng Li, et al. CFS: Scaling metadata service for distributed file system via pruned scope of critical sections. In *EuroSys'23*, pages 331–346, 2023.

[40] Yuanzhang Wang, Fengkui Yang, Ji Zhang, Chunhua Li, Ke Zhou, Chong Liu, Zhuo Cheng, Wei Fang, and Jinhu Liu. Ldpp: A learned directory placement policy in distributed file systems. In *ICPP'22*, pages 1–11, 2022.

[41] Yuanzhang Wang, Peng Zhang, Fengkui Yang, Ke Zhou, and Chunhua Li. Loadm: Load-aware directory migration policy in distributed file systems. In *DATE'24*, pages 1–6. IEEE, 2024.

[42] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, et al. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, November 2006. USENIX Association.

[43] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *SC'04*, 2004.

[44] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Baleen:ML admission & prefetching for flash caches. In *FAST'24*, pages 347–371, 2024.

[45] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *SC'09*, pages 1–11. IEEE, 2009.

[46] Jingwei Xu, Mingkai Dong, Qiulin Tian, Ziyi Tian, Tong Xin, and Haibo Chen. Asyncfs: Metadata updates made asynchronous for distributed filesystems with in-network coordination. *arXiv preprint arXiv:2410.08618*, 2024.

[47] Juncheng Yang, Ziming Mao, Yao Yue, et al. GL-Cache: Group-level learning for efficient and high-performance caching. In *FAST'23*, pages 115–134, 2023.

Appendix A Proof of Theorem 1 and Analysis of Algorithm 1

Proof of Theorem 1. **Benefit formulation.** To mitigate the issue of load imbalance, we identify and migrate a subset of metadata from an overloaded machine A to an underloaded machine B. Denote the load difference as $D = A.\text{rct} - B.\text{rct}$. Migrating subtree s from machine A to machine B reduces A's RCT by l_s and increases B's RCT by $l_s + o_s$, where l_s and o_s represents the load and additional overhead associated with subtree s respectively. After the migration, the system's overall performance is the maximum of the two machines' RCTs. Therefore, the benefit of migrating s is

$$b_0 = \begin{cases} l_s, & D \geq 2l_s + o_s \quad \text{(A-1)} \\ D - (l_s + o_s), & D < 2l_s + o_s \quad \text{(A-2)} \end{cases}$$

Benefit of optimal solution. If the optimal solution is to migrate a set of disjoint subtrees contained within subtree s, denoted as

$\{k_1, \ldots, k_N\}$. The corresponding benefit is

$$b_1 = \begin{cases} \sum_{k_n} l_{k_n}, & D \geq \sum_{k_n} (2l_{k_n} + o_{k_n}) \quad \text{(A-3)} \\ D - \sum_{k_n} (l_{k_n} + o_{k_n}), & D < \sum_{k_n} (2l_{k_n} + o_{k_n}) \quad \text{(A-4)} \end{cases}$$

The sub-optimality gap is written as $b_0 - b_1$, which measures the performance of Alg. 1. A lower bound for this gap is desired.

Conditions. Since subtrees $\{k_1, \ldots, k_N\}$ are all nested within subtree s, their cumulative load and overhead must be strictly smaller than those of s. So we have $l_s > \sum_{k_n} l_{k_n}, \quad o_s > \sum_{k_n} o_{k_n}$. Moreover, to prevent the migration from introducing a new imbalance, Alg. 1 imposes an additional constraint (Line 9 in Alg. 1):

$$\Delta > B.\text{rct} + l_s + o_s - (A.\text{rct} - l_s) = 2l_s + o_s - D.$$

Lower bound for the gap. Given the above conditions, we can get that when $D \geq 2l_s + o_s$,

$$b_0 - b_1 = l_s - \sum_{k_n} l_{k_n} > 0.$$

When $\sum_{k_n} (2l_{k_n} + o_{k_n}) \leq D < 2l_s + o_s$,

$$b_0 - b_1 = D - (l_s + o_s) - \sum_{k_n} l_{k_n} > D - 2l_s - o_s > -\Delta.$$

When $D < \sum_{k_n} (2l_{k_n} + o_{k_n})$,

$$b_0 - b_1 = \sum_{k_n} (l_n + o_n) - (l_s + o_s) > -\Delta.$$

This completes the proof. □

The above analysis indicates that when there is a significant imbalance, specifically, when the imbalance D exceeds $2l_s + o_s$, the greedy decision made by Alg. 1 is indeed optimal. On the other hand, when the load difference is relatively small, Alg. 1 may no longer yield the optimal result. In these scenarios, the optimal strategy would involve carefully selecting and migrating a finely tuned set of smaller subtrees to achieve a more balanced state without introducing significant overhead.

Nevertheless, the performance gap between Alg. 1 and the optimal solution remains bounded. The benefit difference between the greedy solution and the optimal fine-grained adjustment is bounded by Δ. This ensures that even in suboptimal conditions, Alg. 1 maintains performance within a controlled margin of error.