Accelerating Metadata Management of DFS via Speculative Permission Checking

Yiduo Wang
China Telecom Cloud Computing Research Institute
Beijing, China
wangyd22@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
jiewu@temple.edu

***Abstract*—Modern distributed file systems face a critical metadata bottleneck as cloud computing and large-scale machine learning applications generate massive datasets with billions of files. Traditional metadata managements suffer from expensive path resolution: each operation requires $O(N)$ remote procedure calls and database queries to recursively check permissions and locate inodes across hierarchical namespaces. This paper revisits real-world namespace structures and reveals that the vast majority of metadata exhibits a *permission descending pattern*, enabling safe shortcuts for permission checking. We leverage this insight to design SPMeta, a scalable metadata management that integrates two key techniques: (1) *Speculative Permission Checking*, which bypasses expensive permission checking while preserving POSIX semantics, and (2) *MART*, a metadata-optimized trie structure that accelerates both permission checking and inode indexing. By combining these, SPMeta reduces the number of remote procedure calls (RPCs) and database queries for most metadata operations from $O(N)$ to $O(1)$. Evaluation in five real-world namespaces shows that SPMeta achieves superior metadata performance in read-only and mixed workloads, improving aggregate throughput by 1.44× to 2.86× over state-of-the-art baselines while scaling to distributed file systems with tens of billions of files.**

***Index Terms*—File system, Metadata, Distributed Storage.**

I. Introduction

With the explosive growth of cloud computing and large-scale machine learning applications, modern data centers are experiencing an unprecedented surge in massive data [1]–[3]. This growth has created an urgent demand for storage systems that can deliver high performance, massive capacity, and robust reliability at scale. Distributed file system (DFS), renowned for their compatibility and ease of deployment, have emerged as critical infrastructure components for cloud computing, large-scale model training, and inference workloads [4]–[9].

With the ubiquity of high-speed SSDs and NICs in modern data centers [10], [11], metadata processing has overtaken data transfer as the primary bottleneck in large-scale DFSs [12]. Recent production traces for cloud and AI applications [13], [14] further reveal that modern DFSs need to scale to tens of billions of files, while metadata operations account for over 2/3 of the total workload, and 96% of data I/O requests are smaller than 32 KB. This shift has attracted considerable research attention, as metadata operations, including path resolution, permission checking, and inode lookups, now dominate the critical path for file system performance. The challenge is particularly acute in namespace-intensive workloads, where applications perform millions of metadata operations per second across deeply nested directory hierarchies [14]. Traditional approaches to metadata management have struggled to meet the scalability demands of modern applications. Centralized designs such as HDFS [15] suffer from single point of failure and limited throughput. Subtree-based partitioning schemes employed by systems like CephFS [16], IndexFS [17] and HopsFS [18] preserve locality but face load balancing challenges and complex metadata migration overhead [19], [20].

To address scalability limitations and simplify metadata management, recent systems have adopted storage-compute disaggregation architectures and store metadata in distributed databases or Key-Value (KV) storage [21]–[23]. Hash-based partitioning approaches such as CFS [13] and InfiniFS [12] distribute metadata across metadata servers (MDSs) to achieve better load balance, but fundamentally compromise path resolution performance. For each metadata operation that involves a path with N directories, these systems require $O(N)$ RPCs, permission checks, and database I/Os, as each component of the path must be resolved through separate RPCs to potentially different MDSs and underlying storage systems.

Recent systems have adopted *Namespace-Centric* approaches that prioritize metadata management efficiency. However, existing solutions face fundamental trade-offs: namespace replication systems like FalconFS [2] achieve low latency at the cost of substantial storage overhead and synchronization complexity, while centralized approaches like LocoFS [24] reduce network overhead but create scalability bottlenecks due to extensive permission checking and index query requirements.

To overcome the metadata bottleneck, we aim to achieve three competing goals: (1) *High Scalability*—supporting tens of billions of files and directories with consistent performance, (2) *Strong Locality*—maintaining efficient path resolution across the hierarchical namespace, and (3) *Minimal Overhead*—avoiding the prohibitive costs of distributed coordination and redundant storage. This paper presents SPMeta, which achieves these goals through the following key contributions:

- **Speculative Permission Checking.** We target large-scale data center environments (e.g., HPC and AI clusters) where deep directory hierarchies make recursive permission checks a critical bottleneck. In these scenarios, we observe that real-

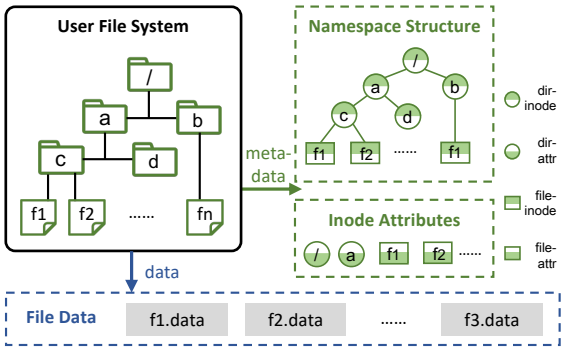

Fig. 1: Components of User File System.

world file systems exhibit a *permission descending pattern* on directory execute permissions, where `user` execute permissions are typically equal to or greater than `group` permissions, which in turn are greater than or equal to `other` permissions (i.e., user.x $\geq$ group.x $\geq$ other.x). We exploit this pattern to design speculative permission checking, which pre-computes directory reachability for different user types, thereby bypassing expensive recursive permission checks. This reduces the permission checking overhead from $O(N)$ to $O(1)$ for the vast majority of operations without compromising POSIX semantics.

- **MART: Metadata-Oriented ART.** We design MART, an efficient trie structure optimized for metadata operations based on the classical ART (Adaptive Radix Tree) [25]. MART provides efficient indexing for both permission reachability checking and inode lookups, allowing SPMeta to scale to namespaces with tens of billions of files while maintaining low memory cost.
- **Hybrid Metadata Organization.** We propose a hybrid metadata management design that decouples namespace structure from file metadata. By caching the namespace structure and reachability in a dedicated Path Node and distributing inode attribute metadata across MDSs, SPMeta achieves both access locality and scalability.

We implement and evaluate SPMeta against state-of-the-art metadata management systems in real-world namespaces. Our evaluation demonstrates that SPMeta achieves significant performance improvements while maintaining full POSIX compliance and scalability. Specifically, the speculative permission checking mechanism reduces path resolution overhead by up to 87.5% in real-world namespaces. Compared to state-of-the-art systems, SPMeta improves metadata aggregation throughput by 1.42× to over 10× across diverse workloads, demonstrating substantial performance gains in large-scale namespace operations while scaling well.

II. Background and Motivation

A. Distributed File System and Metadata

As shown in the upper left of Fig. 1 , the user file system is typically organized as a tree structure, where directories serve as intermediate nodes and files serve as leaf nodes. The file system primarily consists of data and metadata, which include:

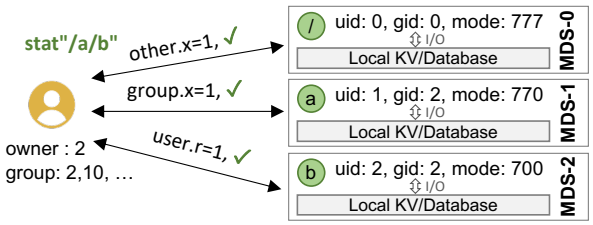

Fig. 2: Path resolution workflow in disaggregated architecture.

- *Namespace Structure*: the hierarchical directory tree structure of the file system, represented as a directed acyclic graph (DAG). This forms the core component of metadata, encompassing file names, inodes, hierarchical relationships, and permissions. Most file system operations require traversing each parent directory and the target file along the specified path. This process involves reading structural information to verify both the existence and permissions necessary for resolving paths within the file system.
- *Inode Attributes*: the collection of inodes for each file and directory along with their associated attributes and statistics. This component constitutes a flat mapping from the inodes to their attributes, including timestamps, file size, and link counts. These attributes are accessed during metadata operations (e.g., the `stat` operation retrieves attribute information, while operations such as `mkdir` modify timestamps and link counts).
- *File Data*: the unstructured content of files, including binary content, video frames, and other application-specific data.

B. Disaggregated Metadata Management

To scale metadata management, traditional DFSs (e.g., HDFS, CephFS) rely on a dedicated MDS or dynamic subtree partitioning to preserve namespace locality, yet they often struggle with load balancing and limited scalability. Consequently, modern DFSs have shifted toward a disaggregated architecture that offloads metadata persistence to scalable distributed KV stores rather than maintaining it locally.

Fig. 2 illustrates a typical example of metadata operations in a disaggregated architecture. As shown, to complete the `stat("/a/b")` operation, the user must start from the root node "/" and sequentially access all metadata nodes along the path, checking the corresponding permissions: execute permission for the root and "a", and read permission for directory "b". In MDS-0, since the user's uid and gid do not match those of the root directory, the system checks the `other` permissions (i.e. the third digit of the mode). In this example, the root node's other permission is 7 (111 in binary), indicating that `read`, `write`, and `execute` permissions (r, w, x) are all granted, allowing the metadata request to proceed. Similarly, after the user sequentially passes the group execute permission check for directory "a" and the user read permission check for directory "b", the actual metadata read operation is performed. As can be seen, although the disaggregated design significantly simplifies metadata management and improves scalability, it

TABLE I: Comparison of SOTA DFS metadata management methods. N: the depths of file paths; M: the number of MDSs.

Metadata Organization	Partition Methods	System Names	Scalability	NameSpace Locality	RPC Count	Permission Check	I/O Count	Sync Overhead
Metadata-Data Decoupling	**Centralized**	*HDFS [15]*	Worst	**Best**	$O(1)$	$O(N)$	$O(1)$	**Low**
	Subtree	*Ceph [26], HopsFS [18]*	Worse	**Good**	$O(M)$	$O(N)$	$O(1)$	Fair
Metadata Storage-Compute Disaggregation	**Hash**	*Tectonic [7], CFS [13]*	**Good**	Worse	$O(N)$	$O(N)$	$O(N)$	**Low**
		InfiniFS [12]	Fair	Worse	$O(N)$	$O(N)$	$O(N)$	Fair
	Namespace -Centric	*FalconFS [2]*	Worse	**Good**	$O(1)$	$O(N)$	$O(N)$	High
		LocoFS [24]	Worse	**Good**	$O(1)$	$O(N)$	$O(N)$	Fair
		Mantle [14]	Fair	**Good**	$O(1)$	$O(N)$	$O(N)$	Fair
		SPMeta	**Good**	**Good**	$O(1)$	$O(1)$	$O(1)$	**Low**

compromises namespace locality, leading to increased overhead in path resolution and permission checking.

As illustrated in Table I, hash-based partitioning systems (e.g., CFS, InfiniFS) improve scalability by distributing directory attributes but fundamentally compromise path resolution performance. For a path with k directories, these systems incur linear overhead ($O(k)$ RPCs and I/Os), and parallel resolution mechanisms often prove ineffective under high-concurrency production workloads [14]. To address these limitations, recent research has shifted toward the "Namespace-Centric" approaches (categorized in Table I) that prioritize namespace management efficiency for large-scale small-file workloads.

- *Namespace replication*: FalconFS [2] implements this approach specifically for a static dataset, reducing RPC overhead in path resolution and permission check by trading off substantial storage redundancy and synchronization costs. Although effective in specialized use cases, it lacks generalizability to broader storage scenarios.
- *Centralized namespace storage*: Mantle [14], designed for large-scale object storage, routes metadata operations through a central node to reduce path resolution RPCs, similar to LocoFS's partitioning approach. While this design improves scalability compared to fully distributed approaches, it lacks file system-specific optimizations. The central node must perform extensive permission checking and index lookups for every operation, creating scalability bottlenecks in both capacity and performance.

C. Challenges and Goal

Based on the analysis above of existing metadata management, our aim is to design a *Namespace-Centric* metadata management architecture that simultaneously achieves high scalability, preserves locality, and enables metadata scaling with minimal overhead. To achieve this goal, the following main challenges need to be addressed.

Challenge #1: Expensive recursive permission checking in hierarchical namespaces. In large scale metadata management systems that support hundreds of billions of entries, hierarchical path traversal creates significant overhead. Each directory in the path requires metadata existence and permission checking, leading to substantial RPC and I/O overhead.

Challenge #2: Costly inode indexing in the large scale. Massive-scale systems face prohibitively large memory requirements for complete namespace caching. Existing tiered storage architectures based on key-value stores or databases further compound this issue, suffering from complex database queries and expensive computational overhead, as well as intensive storage I/O operations.

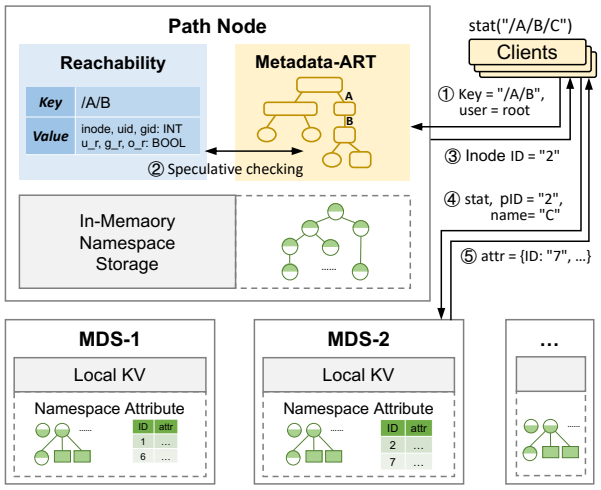

Fig. 3: Overall architecture of SPMeta.

III. System Overview

A. Overall Architecture

This paper proposes SPMeta, a highly scalable Namespace-Centric metadata management architecture that aims to achieve $O(1)$ permission checking and I/O operations, along with fast inode indexing while maintaining minimal memory overhead. As illustrated in Fig. 3, SPMeta employs two different types of metadata nodes to store metadata:

- **Centralized Path Node**: This node maintains the reachability of namespace structure in memory, responsible for handling the path resolution of all metadata operations. When clients execute metadata requests, they first send the complete path (excluding the leaf node) to the Path Node, which performs permission checks and inode lookups (①-③) and returns the permission check results along with the inode ID of the last-level directory.
- **Distributed MDSs**: These nodes persist remaining metadata (directory attributes and file metadata) in local KV stores. Our design strategy incorporates insights from CFS [13] and InfiniFS [12], decoupling inode IDs from their attributes and arranging them in the KV store using the *"parent ID +*

name" format. This ensures that the directory attributes and their child files are stored on the same MDS, thereby reducing additional RPC call overhead. After path resolution, the client constructs the *parent ID + name* key to forward the request to the target MDS (④). The MDS then queries its local KV store and returns the target file attributes (⑤).

B. Key Technologies in SPMeta

As shown in the upper part of Fig. 3, SPMeta maintains an in-memory replica of the namespace. To conserve memory, SPMeta caches only directory inodes (typically $\approx 10\%$ of total entries). However, A naive approach that caches the entire namespace would consume tens of TBs of memory and incurs prohibitive $O(N)$ path resolution overhead. To address these scalability and performance challenges, SPMeta achieves high efficiency through two key innovations:

Speculative Permission Checking Mechanism. We propose a reachability-based permission checking technique that uses a single bit flag for each of the three permission classes (user, group, other) per directory. When a flag is set, it indicates that the corresponding user type definitely has the required access permission along the path. This approach enables SPMeta to achieve $O(1)$ path resolution for more than 99.7% of directories in five real-world namespace datasets. Details are presented in § IV.

Metadata-Oriented Adaptive Radix Tree. To efficiently index the reachability of paths at massive scale, we design MART, a metadata-oriented trie based on the classical ART structure. MART incorporates value-embedded inner nodes for efficient metadata indexing and adopts adaptive tree relocation to efficiently support directory `rename` operations. This design enables MART to index namespaces containing 10 billion files while consuming only hundreds of gigabytes of memory. Details are presented in § V.

IV. Fast Long Path Permission Checking via Speculative Mechanism

A. POSIX Permission Checking Mechanism

The POSIX [27] standard defines a comprehensive set of semantic requirements for file systems, including the widely-used permission checking mechanism. POSIX specifies three types of permission for each file: `read`, `write`, and `execute`. Operations in the file itself require the corresponding `read` or `write` permissions, while path resolution requires `execute` permissions on all directories from the root to the parent directory.

Each file has an associated `uid` and `gid`, representing its owner and group, respectively. Additionally, each file maintains three sets of rwx permissions for the permission classes **user**, **group**, and **other**. During permission checking, the system first determines which permission set to examine based on the accessing user's uid and gid, following these rules: (1) if the user's uid matches the file's uid, check user permissions; (2) if condition (1) is not met but the user's gid matches the file's gid, check group permissions; (3) if neither condition is satisfied, check other permissions. It is crucial to note that these rules follow a strict priority order: Only one permission set is checked per access. For example, if a user's uid matches the file's uid but the corresponding bit in the user permission set is not authorized, access will be denied even if the other permission set grants authorization.

B. Speculative Permission Checking

Pre-computing Directory Reachability. To simplify the complex permission checking mechanism, we first analyze metadata permission characteristics in real-world scenarios. We present a key observation: when a user accesses a target file, the access must fall into exactly one of three classes: (1)owner, the user's `uid` matches the file's `uid`, (2)group, the user belongs to the group corresponding to the file's `gid`, or (3) the user is considered as "others". Our key insights are:

*Insight #1: By precomputing speculative **reachability** (where a set bit guarantees access while a cleared bit triggers a fallback), the complex per-directory execute permission checks can be bypassed for the vast majority of requests.*

A straightforward approach would be to directly compute the reachability for each corresponding scenario. However, this method faces a critical challenge: users may belong to multiple groups simultaneously, which can lead to situations where a directory allows access to most users (i.e., `other.x = 1`) but specifically denies access to users in certain groups (similar to a blacklist mechanism). This complexity makes precomputation significantly more challenging.

Fortunately, through analysis of four real-world filesystem images, we discovered that the vast majority of directories exhibit a consistent characteristic, leading to our second insight:

*Insight #2: Most directories follow a **permission descending pattern (PDP)**: permission bits follow a descending order from owner to group to others (owner $\geq$ group $\geq$ others).*

This observation motivates us to adopt a speculative approach for computing reachability: when the `uid` and `gid` do not match exactly, we can directly use `other.x` as the determining criterion. To this end, we design Algorithm 1 to compute the corresponding reachability for all three classes.

For each file requiring permission checks, we initialize the reachability bits (line 1) for all three classes (U_R, G_R, O_R)—where 0 signifies that the speculative mechanism is uncertain about reachability—and record the file's `uid` and `gid` (line 2). We then traverse all directories from the file to the root (line 3), checking whether any directory denies access, and setting the corresponding reachability bit to 0. First, we check whether the PDP is satisfied. If any directory along the path violates the PDP, we immediately return with all reachability bits set to 0 (lines 4–5), otherwise, we check the reachability as follows:

- *Owner reachability (U_R)*: We compare the target file's uid with the current directory's uid. If they match and the directory's execute permission for the owner is 0 (i.e., `user.x = 0`), access is denied and we set $U_R = 0$, marking it as unreachable (lines 6–7). If the uids do not match, we fall back to checking the execute permission for others (`other.x`). If `other.x = 0`, the directory is unreachable, and we set $U_R = 0$ (lines 8–9).

Algorithm 1: Reachability Determination Algorithm

Input : Full path $P = \{d_1, d_2, ..., d_n\}$ where d_n is the target directory;

Output: Reachability sets for d_n: U_R, G_R, O_R

1 $U_R \leftarrow 1, G_R \leftarrow 1, O_R \leftarrow 1$;
2 $target.uid \leftarrow d_n.uid, target.gid \leftarrow d_n.gid$;
3 **foreach** $d_i \in P$ **do**
4 **if** $\neg PDP(d_i)$ **then**
5 **return** $(U_R, G_R, O_R) \leftarrow (0, 0, 0)$;
6 **if** $target.uid = d_i.uid \wedge d_i.user.x = 0$ **then**
7 $U_R \leftarrow 0$;
8 **else if** $target.uid \neq d_i.uid \wedge d_i.other.x = 0$ **then**
9 $U_R \leftarrow 0$;
10 **if** $target.gid = d_i.gid \wedge d_i.group.x = 0$ **then**
11 $G_R \leftarrow 0$;
12 **else if** $target.gid \neq d_i.gid \wedge d_i.other.x = 0$ **then**
13 $G_R \leftarrow 0$;
14 **if** $d_i.other.x = 0$ **then**
15 $O_R \leftarrow 0$;
16 **return** U_R, G_R, O_R;

- *Group reachability (G_R)*: We compare the target file's gid with the current directory's gid. If they match and the directory's execute permission for group is 0 (i.e., `group.x = 0`), access is denied and we set $G_R = 0$, marking it as unreachable (lines 10–11). If the gids do not match, we fall back to checking the execute permission for others (`other.x`). If `other.x = 0`, the directory is unreachable, and we set $G_R = 0$ (lines 12–13).
- *Other reachability (O_R)*: We check the execute permission of the current directory for others (`other.x`). If `other.x = 0`, the access is denied and we set $O_R = 0$, marking it as unreachable (lines 14–15).

Corner Case Handling. To ensure correctness, Algorithm 1 (lines 4-5) explicitly leverages the PDP to filter out corner cases where speculative checking might fail. Consider a user with `uid=1` who belongs to group `gid=2`, attempting to access a directory owned by `uid=0` and assigned to group `gid=2`. Suppose that the directory's execute permissions are set to 1 (owner), 0 (group), and 1 (other), respectively. In this scenario, without strict validation, a naive speculative method would incorrectly grant execution permission by using the "other" permission bit (which is 1), while the correct behavior should deny access based on the group permission (which is 0). Fortunately, such errors arise only when permissions violate the PDP. Therefore, lines 4-5 detect this violation ($\neg$PDP(d_i)) and immediately invalidate the speculative result by returning $(0, 0, 0)$, enforcing a fallback to standard checks.

C. Correctness Proof

The discussion above intuitively explains the mechanism's correctness and corner case handling. We now formalize this guarantee and provide a rigorous proof.

For a given request r to access path P, the POSIX permission (PP) checking method performs recursive authorization verification. We denote its result by $\mathrm{PP}(r, P)$. In contrast, the proposed method performs speculative permission checking using the reachability bits. We denote its result by $\mathrm{SP}(r, P)$. Our main correctness guarantee is stated as follows.

Theorem 1. *Under PDP, for any request r to path P,*

$$\mathrm{SP}(r, P) = 1 \Rightarrow \mathrm{PP}(r, P) = 1.$$

Proof. We use u_r to denote the request user ID and $\mathcal{G}_r$ to denote the set of groups to which the request user belongs. The access path is written as $P = (d_0, \ldots, d_{n-1}, d_n)$, where each d_i represents a directory and d_n the target file. For any object (file or directory) f, let $U(f)$ and $G(f)$ denote its user and group ownership, respectively. Let $x_u(f), x_g(f), x_o(f) \in \{0, 1\}$ represent the executable bits for the user, group, and others classes of the object. According to the POSIX matching rules, the system checks whether a request r matches an object f as a user, a group member, or other, in order. We use $c_r(f)$ to denote the first matching class:

$$c_r(f) = \begin{cases} U, & u_r = U(f); \\ G, & u_r \neq U(f), G(f) \in \mathcal{G}_r; \\ O, & \text{otherwise.} \end{cases}$$

Let $x_c(f) \in \{0, 1\}$ represent the accessibility bit of f under class $c \in \{U, G, O\}$.

The canonical evaluation process can be formulated as $\mathrm{PP}(r, P) = \bigwedge_{i=0}^{k-1} x_{c_r(d_i)}(d_i)$. The proposed additional reachability bits are defined as $O_R(d_n) = \bigwedge_{i=0}^{k-1} x_o(d_i), G_R(d_n) = \bigwedge_{i=0}^{k-1}([G(d_i) = G(d_n) \wedge x_g(d_i)] \vee [G(d_i) \neq G(d_n) \wedge x_o(d_i)]), U_R(d_n) = \bigwedge_{i=0}^{k-1}([U(d_i) = U(d_n) \wedge x_u(d_i)] \vee [U(d_i) \neq U(d_n) \wedge x_o(d_i)])$. The fast evaluation result is

$$\mathrm{SP}(r, P) = \begin{cases} 1, & c_r(d_n) = U,\ U_R(d_n) = 1; \\ 1, & c_r(d_n) = G,\ G_R(d_n) = 1; \\ 1, & c_r(d_n) = O,\ O_R(d_n) = 1; \\ \mathrm{PP}(r, P), & \text{otherwise.} \end{cases}$$

If the access path does not satisfy PDP, then $U_R = G_R = O_R = 0$, and hence SP = PP. Theorem 1 is trivially valid. Therefore, in the following, we assume that the entire path satisfies the PDP and discuss the cases according to $c_r(d_n)$.

Case 1: $c_r(d_n) = O$ and $\mathrm{SP}(r, P) = 1$. By definition, $\forall i,\ x_o(d_i) = 1$. Under the PDP condition, this implies that $\forall i,\ x_{c_r(d_i)}(d_i) = 1$. Therefore, $\mathrm{PP}(r, P) = 1$.

Case 2: $c_r(d_n) = G$ and $\mathrm{SP}(r, P) = 1$. Since $c_r(d_n) = G$, we have $G(d_n) \in \mathcal{G}_r$. For any directory level d_i, if $G(d_i) = G(d_n)$ and $x_g(d_i) = 1$, there are two subcases.

1) If $u_r = U(d_i)$, then $c_r(d_i) = U$. By the PDP condition, $x_u(d_i) \geq x_g(d_i) = 1$, hence $x_{c_r(d_i)}(d_i) = 1$.
2) If $u_r \neq U(d_i)$, then $c_r(d_i) = G$ and $G(d_n) = G(d_i)$, so $x_{c_r(d_i)}(d_i) = x_g(d_i) = 1$.

On the other hand, if $G(d_i) \neq G(d_n)$ and $x_o(d_i) = 1$, the PDP condition directly implies $x_{c_r(d_i)}(d_i) = 1$. Therefore, $\mathrm{PP}(r, P) = 1$.

TABLE II: Real-world metadata permission analysis.

Dataset	# Dir	Depth	PDP	Reachability		
				User	Group	Other
Users	1.6M	9.22	99.90%	99.22%	11.75%	0.00%
Annoy	3.9M	11.52	99.96%	97.59%	33.29%	4.94%
SCR4	5.0M	10.27	99.71%	99.26%	21.68%	0.02%
Projs	14.0M	13.65	99.98%	50.56%	52.01%	19.17%

Case 3: $c_r(d_n) = U$ and $\mathrm{SP}(r, P) = 1$. We have $u_r = U(d_n)$. For each directory level d_i, if $U(d_i) = U(d_n)$ and $x_u(d_i) = 1$, then $u_r = U(d_i)$, so $c_r(d_i) = U$ and $x_{c_r(d_i)}(d_i) = x_u(d_i) = 1$. If $U(d_i) \neq U(d_n)$ and $x_o(d_i) = 1$, then by the PDP condition $x_{c_r(d_i)}(d_i) = 1$. Hence, $\forall i,\ x_{c_r(d_i)}(d_i) = 1$, and consequently $\mathrm{PP}(r, P) = 1$.

In conclusion, under PDP, every successful SP case necessarily implies the success of PP. □

D. Efficiency in Real-World Namespace

We conducted a reachability analysis on real-world namespaces, specifically derived from large-scale AI and HPC workloads [28], with results shown in Table II. Our findings reveal that over 99.7% of paths satisfy the permission descending pattern (column 4), demonstrating that our algorithm can effectively mark reachability for the vast majority of directories. As shown in the Reachability column, most directories allow access by the user themselves, while group user access is more restrictive, and most directories deny access to other users—a pattern that aligns with our intuitive understanding of file system permissions. Since actual file system accesses are typically authorized operations, we can reduce the average number of permission checks from 9.22-13.65 (column 3) to nearly 1, achieving a reduction of up to 92.6%.

E. Workflow of Fast Path Resolution

Building upon the reachability design and its correctness proof described above, we implement an in-memory tree structure within each Path Node (implementation details in § V) to facilitate fast permission checking and path resolution. During metadata service initialization, SPMeta constructs the in-memory tree by traversing the filesystem and storing paths along with their corresponding reachability values. To reduce storage overhead on Path Nodes, this tree stores only essential metadata for directories—specifically the inode, uid, gid, and reachability information—while excluding file entries. This design ensures that Path Nodes can maintain all necessary information in memory for efficient lookup operations.

Based on the above design, Path Nodes now use speculative permission checking for fast path resolution, and handle metadata operations following the workflow shown in Fig. 4:

- *Path resolution:* When a client initiates a metadata request, it is decomposed into a lookup request for the parent directory and the actual metadata operation. The client first sends (1.1) a lookup request containing the parent directory's path and the user's uid to the Path Node. The Path Node traverses the in-memory tree to resolve the path and retrieve the corresponding inode, uid, gid, and reachability information, then determines which reachability type to use based on the user's uid. If the reachability check succeeds, it returns the parent's inode ID and the corresponding MDS location; otherwise, it falls back to the original lookup path (1.2).

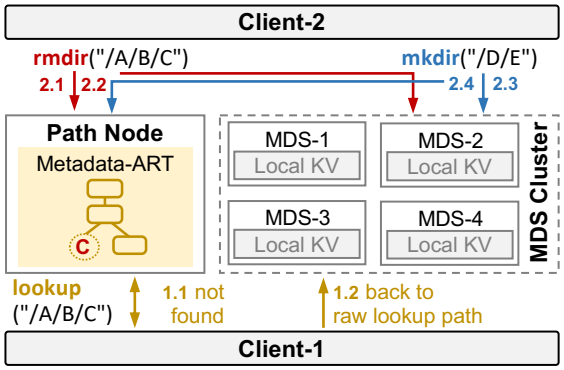

Fig. 4: Workflow of metadata operations.

- *Metadata mutation:* Since Path Nodes essentially serve as an index for metadata, we must carefully design the interaction workflow to ensure consistency in the distributed file system. We adopt a metadata update strategy inspired by prior work on distributed indices: *eagerly updating the index during deletions and lazily updating it during creations*. For deletion operations (e.g., `rmdir`), we first remove the reachability information from the Path Node (2.1) before executing the actual directory deletion on the MDS (2.2). For creation operations (e.g., `mkdir`), we first perform the actual metadata mutation on the MDS (2.3), then update reachability on the Path Node (2.4). This design ensures that any metadata visible on Path Nodes is also present on MDSs, maintaining a conservative consistency model.
  Under this design, if a client cannot successfully resolve a path via the Path Node due to concurrent in-flight metadata operations, the system falls back to the original path resolution workflow through the MDS, thereby avoiding inconsistencies. Taking Fig. 4 as an example, if client1's `lookup` operation occurs while client2's `rmdir` operation is in flight (after step 2.1 but before step 2.2), client1 will encounter a reachability check failure and fall back to the MDSs and use original step-by-step path resolution, ensuring it observes a consistent file system state.
- *Tree-based operations* Distributed file systems must also address the challenges of directory `rename` operations, which represent complex metadata operations that require both correctness and performance guarantees. Specifically, `rename` operations must prevent concurrent orphan loops while avoiding extreme tail latency. To this end, we employ an in-memory locking mechanism to ensure cycle-free concurrent `dirrename`. Additionally, we extend MART to support `dirrename`, which will be detailed in § V-B.

F. Concurrency Control

Since the Reachability bits act as a cached state derived from the directory hierarchy, it simplifies permission checking but

introduces dependencies: a permission change on a directory (e.g., `chmod`) invalidates the cached reachability states of its entire subtree. Consequently, such operations, along with subtree movements like `dirrename`, must be strictly serialized to prevent inconsistencies.

To prevent this, drawing inspiration from state-of-the-art designs [12]–[14], SPMeta first records in-flight range mutations (e.g., `dirrename`) within the PathNode to detect and serialize overlapping subtree operations. Once serialized, the execution follows a 2PC-like workflow to ensure atomicity and isolation: First, SPMeta acquires a write lock on the target metadata stored in the MDS's local key-value store, ensuring isolation from concurrent deletions or updates. Second, to prevent inconsistent access during the mutation, we mark the subtree root as "dirty" in MART to force speculative lookups to fall back to step-by-step path resolution, and then commit the modification to the MDS.

While subtree operations can incur high overhead when manipulating large subtrees, this overhead is not a bottleneck for SPMeta. First, such mutations are extremely rare in production environments ($< 0.01\%$ of workloads [12]). Second, by employing *Adaptive Tree Relocation* for renames and *Lazy Range Invalidation* for permission changes, we decouple the mutation latency from the subtree size (see details in § V-C). This ensures that reachability updates remain efficient even for billion-scale directories.

V. Efficient Reachability Indexing via Metadata-oriented ART

The speculative permission checking mechanism described above enables single-RPC permission checks in most cases by caching file paths along with reachability in Path Nodes. This speculative index has two key properties that eliminate the need for persistence: (1) it represents only a compact subset of the file system metadata, and (2) it can safely fall back to the original path resolution workflow upon data loss or inconsistency. In this section, we present the design of an efficient in-memory data structure that maintains this speculative index by storing mappings from file paths to inodes and permission information. This data structure must satisfy three key requirements:

- Lookup efficiency: It must provide fast lookups to support high-throughput path resolution requests from thousands of concurrent clients.
- Semantic support: It must efficiently support complex metadata operations such as `rename` (which moves subtrees of metadata) and `chmod` (which performs range updates on reachability).
- Memory efficiency: The data structure must use a compact representation to minimize memory footprint while storing metadata for millions of files.

A. Data Structure Selection

Probabilistic or Deterministic? A key insight is that reachability checking tolerates false negatives—incorrectly reporting unreachability is harmless because the system can fall back to the original lookup path. This property makes probabilistic data structures, particularly Bloom filters [29], appealing candidates, as they can achieve better lookup performance and memory efficiency than deterministic approaches at the cost of occasional false negatives. Motivated by their excellent space-time tradeoffs, we initially explored Bloom filters for reachability indexing.

TABLE III: Hardness of string datasets for learned indexes. ✓/✗ denote if learned indexes outperform tries. * denotes real-world file system metadata. (R=Read, W=Write)

Dataset	GPKL	R	W	Dataset	GPKL	R	W
reddit	8.24	✓	✓	url	47.61	✗	✗
geoname	10.36	✓	✓	Web*	82.14	✗	✗
imdb	10.51	✓	✓	User*	105.47	✗	✗
address	12.61	✓	✓	SCR4*	118.76	✗	✗
wiki	14.32	✓	✓	Annoy*	120.64	✗	✗
dblp	20.79	✗	✓	Projs*	146.92	✗	✗

However, we ultimately rejected this approach for two fundamental reasons. First, Bloom filters only answer membership queries and cannot store associated metadata. This would require a separate data structure to maintain the mapping from paths to inodes and their corresponding MDS addresses. While existing work [12] proposes predictive mechanisms to estimate inode locations, adopting such approaches would require extensive modifications to the system architecture. Second, and more critically, Bloom filters fundamentally lack support for efficient directory `rename` operations. Renaming a directory requires updating all entries in the affected subtree, which would require numerous metadata lookups and batch modifications on the MDS, introducing significant tail latency and lock contention.

To Learn or Not? Recent advances in learned indexes have demonstrated significant improvements in lookup performance and memory efficiency by leveraging machine learning to optimize index structures [30]. While early work primarily focused on numerical data, recent studies have successfully extended learned indexes to string-based scenarios [31], demonstrating their effectiveness. Furthermore, LITS [32] proposes a methodology to quantify the hardness of different string datasets for learned indexing algorithms, providing insight into their applicability. Motivated by these advances, we explored learned indexes for reachability indexing. They are particularly appealing because they offer high performance, memory efficiency, and support for range queries, which could be useful for prefix-based operations on file system paths, such as directory `rename`.

However, our evaluation reveals that existing learned string indexes are ill-suited for file system metadata. We applied LITS's evaluation methodology [32] to various string datasets under both read-only and write-only workloads (Table III). The GPKL (Group Partial Key Length) metric measures learning difficulty, where higher values indicate harder-to-learn datasets. ✓ and ✗ indicate whether the learned indexes outperform or underperform radix trees, respectively. Although learned indexes work well for most LITS benchmark datasets

(upper half of the table), they consistently underperform on real file system metadata (lower half). The file system metadata exhibits extensive prefix sharing, resulting in high GPKL values that make them exceptionally difficult to learn. Consequently, learned indexes do not offer no performance advantage over traditional radix tree-based structures for our workload, leading us to abandon this approach.

Why Trie-Based Structures? Unlike Bloom filters and learned indexes, trie-based structures such as ART [25] and HOT (Height Optimized Trie) [33] avoid the aforementioned limitations and are naturally well-suited for namespace indexing. File system paths exhibit significant prefix sharing, making them highly amenable to trie-based compression. Moreover, tries can support metadata-specific operations through tree manipulations, such as range modifications for `chmod` and subtree relocations for `rename` directories.

We chose to build upon ART [25] due to its effective prefix compression, high lookup performance, and support for insertions, deletions, and range queries. However, ART faces limitations for metadata management. First, deep path hierarchies require excessive pointer traversals, degrading lookup performance. Second, ART does not natively support `rename` operations—renaming a directory requires scanning and updating all descendant entries individually, which is prohibitively expensive. To address these challenges, we designed MART (Metadata-oriented Adaptive Radix Tree) with two key enhancements: (1) optimized node structures to reduce pointer traversal overhead and (2) efficient subtree relocation for fast `rename` operations. The detailed design and implementation are presented in the remainder of this section.

B. Adapting ART for Metadata Management

To address ART's limitations for metadata management, we designed MART with two key enhancements. First, we introduce *value-embedded inner nodes* to accelerate lookups for non-leaf directories and reduce memory overhead. Second, we design an *adaptive subtree relocation* mechanism that enables efficient `rename` operations by relocating subtrees without traversing individual entries. We describe these two techniques in detail below.

Value-Embedded Inner Nodes. Compared to common string datasets, file system metadata exhibit a distinctive characteristic: many directories simultaneously serve as both leaf nodes and common prefixes. For example, if a directory `/user/local/bin/` exists, its ancestors (`/`, `/user/`, and `/user/local/`) must also exist and serve as prefix nodes. In standard ART, this implies that `/`, `/user/`, and `/user/local/` must all exist as inner nodes. However, since the inner nodes in ART cannot store values, each of these directories requires a separate leaf node to store its metadata. This leaf node has an empty key and shares the same prefix as its parent inner node, introducing two inefficiencies: (1) lookups for non-leaf directories require an additional indirection to the leaf node, increasing query latency, and (2) the redundant leaf nodes waste memory.

To eliminate redundant leaf nodes, we extend ART to allow inner nodes to store values directly. Specifically, we modify all four node types (Node4, Node16, Node48, Node256) to include an optional value field at the end of the node structure, with the same type as the value field in leaf nodes.

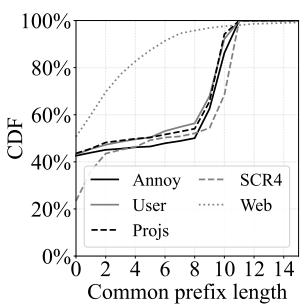

Fig. 5: Common prefix lengths in 5 namespace.

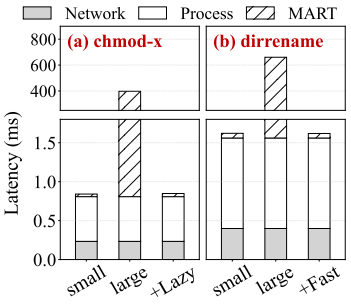

Fig. 6: Overhead analysis of subtree mutations.

When inserting a new key whose prefix matches an existing leaf node's key, MART creates a new inner node to replace the leaf node, embeds the leaf's value in the inner node's value field, and adds the new key as a child leaf node. For example, when inserting `/usr/local/bin/` (value=V_2) into a tree containing a leaf node `/usr/local/` (value=V_1), MART creates an inner node with the prefix `/usr/local/` and value V_1, and adds a child leaf node with the key `bin/` and value V_2. This eliminates the need for a separate leaf node with an empty key to store V_1. For deletion, if an inner node contains a value, we retain it as an inner even when all its children are deleted, avoiding the overhead of repeatedly converting between leaf and inner nodes as subdirectories are added or removed.

Adaptive Tree Relocation. ART lacks native support for subtree relocation, which is essential for efficient directory `rename` operations in file systems. A naive approach would traverse the entire subtree, delete all entries, and re-insert them with updated keys. However, subtrees in file systems can be quite large, and this approach incurs significant overhead proportional to the size of the subtree.

Ideally, subtree relocation could be implemented by updating the parent pointer and modifying the prefix of the root inner node, achieving constant-time complexity with respect to subtree size. However, ART's path compression mechanism poses a challenge: each inner node can store at most `MAX_LEN` bytes of prefix. When the common prefix exceeds this limit, the full prefix is implicitly stored in the leaf nodes' complete keys. In such cases, simply updating the inner node's prefix would create an inconsistency between the inner node's prefix and the actual paths stored in the leaf nodes.

To understand the feasibility of fast relocation, we analyzed the distribution of common prefix lengths in 5 real-world namespaces. As shown in Fig. 5, we found that more than 99% of common prefixes are shorter than 15 bytes in all namespaces. This observation motivates our adaptive tree relocation strategy. Specifically, MART handles relocation based on the node type and prefix length:

- *Leaf nodes:* using simple delete-and-insert methods.
- *Inner nodes with short prefixes:* if the inner node's prefix does not exceed `MAX_LEN`, we perform fast relocation: (1) create a new inner node at the target location, (2) transfer all child pointers and the value field from the source node to the new node, (3) update the new node's prefix to reflect the new path, and (4) remove the source node from its parent. The overhead of fast path is independent with the subtree size, ensuring performance even with large directory.
- *Inner nodes with long prefixes:* if the inner node's prefix exceeds `MAX_LEN`, we recursively delete and re-insert all entries in the subtree.

By setting `MAX_LEN` to 15 bytes, MART achieves fast relocation for most renames while gracefully handling rare cases with longer prefixes.

Tree Construction. To support flexible deployment and rapid recovery, SPMeta offers two mechanisms for constructing the reachability tree in memory. *Bulk Load* is optimized for system initialization, reconstructing the entire directory tree by ingesting an offline namespace image (paths, inodes, permissions). By bypassing the transactional write path, it achieves extreme throughput. *Lazy Load* is designed for crash recovery: when reachability information is missing (a cache miss), the MDS performs standard step-by-step path resolution to validate the request and asynchronously backfills the `PathNode` state. This ensures immediate system availability without waiting for a full state rebuild.

C. Overhead Analysis

We evaluate the performance impact of subtree update operations in Fig. 6. For small directories with 10 children, the MART manipulation overhead for `chmod` or `rename` remains in the microsecond range (tens of μs), negligible compared to I/O. However, as the directory size grows to 1 million nodes, the baseline overhead spikes to hundreds of milliseconds. Crucially, our optimizations (annotated as *+Lazy* and *+Fast*) effectively decouple latency from subtree size, preventing bottlenecks. We also verify the efficiency of state construction on NVMe SSDs. SPMeta completes a *Bulk Load* of a 10B namespace (with over 1B directories) in approximately 20 seconds. Furthermore, even under worst-case cold-start scenarios using *Lazy Load*, the path resolution overhead remains comparable to hash-based partitioning schemes like Tectonic [7], ensuring negligible impact on availability.

VI. Evaluation

A. Experiment Setup

Hardware Configurations. Experiment are conducted on a cloud platform, using 5 client nodes and 5 server nodes, each node is equiped with Intle Xeon platinum 2.6 GHZ 32cores CPU, 128GB memory, 12 Gps network bandwidth and 1TB SSD.

Compared Systems. We compare SPMeta against four state-of-the-art DFSs representing distinct metadata management paradigms: CephFS [16] (dynamic subtree partitioning), Tectonic [7] (hash partitioning), InfiniFS [12] (hash partitioning with parallel path resolution), and LocoFS [24] (hash partitioning with centralized directory management server). We do not include CFS [13] and Mantle [14] in our comparison because their optimizations heavily rely on proprietary distributed coordination components (e.g., TafDB [14]); however, their metadata partition methods and path resolution patterns closely resemble those of Tectonic and LocoFS, respectively.

We implement SPMeta as a prototype system in Go, using PebblesDB [34] for metadata storage and gRPC [35] for network communication. As InfiniFS, Tectonic, and LocoFS are not publicly available, we implement them based on their published designs using the same underlying components as SPMeta to ensure a fair comparison. For deployment, CephFS, InfiniFS, and Tectonic use five nodes as metadata servers. LocoFS and SPMeta both use one centralized path resolution server and four metadata servers, ensuring all systems utilize the same hardware resources.

Namespaces and Workloads. We evaluate all systems using five real-world, publicly available file system namespace snapshots: one from a web server mirror (denoted as *Web*) [19] and four from production data centers (denoted as *Annoy*, *SCR4*, *User*, and *Projs*) [28]. The details of these snapshots are given in Section IV. We use two types of workloads for evaluation: read-only and read-write mixed. The *Web* dataset provides a publicly available read-only workload trace. For the other four snapshots, no available traces. Therefore, we synthesize workload traces based on metadata operation characteristics reported in recent file system studies [13], comprising approximately 5.0% `create`, 4.1% `unlink`, and 1.0% `mkdir`. We implement a custom mdtest-like [36] metadata-only benchmark to replay these traces.

B. Overall Performance on Real-world Workloads

In this section, we evaluate the impact of different metadata management approaches on overall system performance under real-world workloads. We compare SPMeta against all baselines using namespace snapshots from production file systems. Each client spawns 100 threads to saturate the system, and we measure the aggregate throughput of the metadata cluster under both read-only and read-write mixed workloads. Fig. 7(a-e) shows the aggregate throughput under read-only workloads across different trace patterns.

Traditional metadata management exhibits poor scalability. CephFS achieves significantly lower aggregate throughput than all other systems across all workloads. This is due to two fundamental limitations. First, CephFS only decouples metadata from data but does not disaggregate metadata storage and computation, preventing independent scaling of these resources. Second, CephFS partitions metadata using subtree-based sharding, which suffers from severe load imbalance at scale, limiting its ability to scale out effectively.

Hash-based partitioning improves load balance. Unlike CephFS, Tectonic partitions metadata using hash-based sharding across multiple metadata servers, enabling better load balancing and parallelism. This improves aggregate throughput by 127.3% to 155.1% over CephFS.

Concurrent path resolution provides moderate gains. InfiniFS builds on Tectonic by enabling concurrent execution of

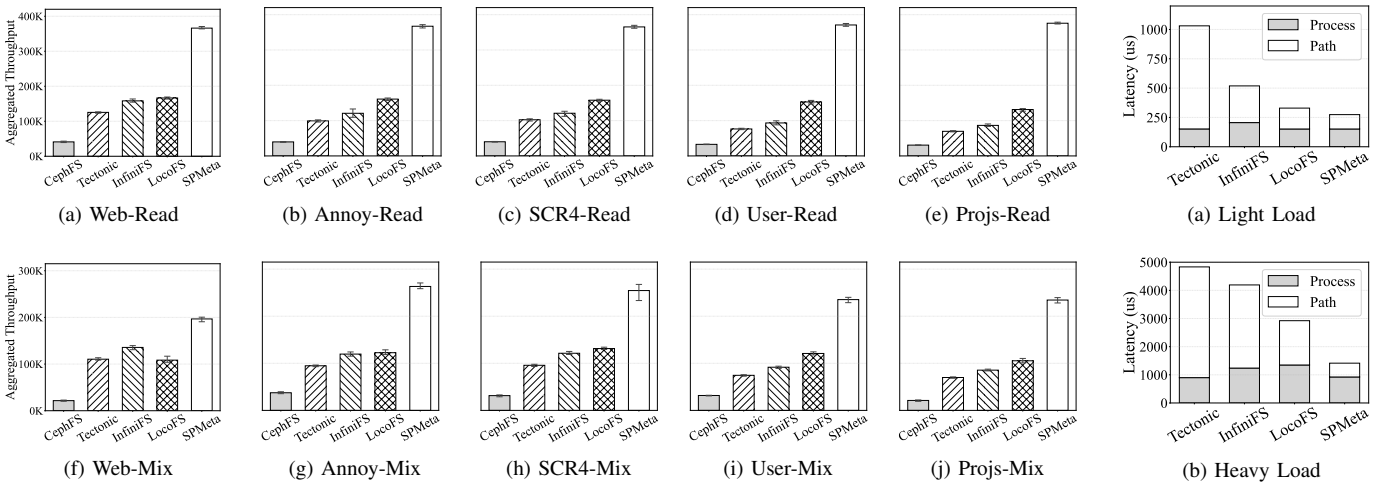

Fig. 7: Aggregate metadata throughput across five namespaces. Top row (a-e): Read-only workloads; Bottom row (f-j): Read-write mixed workloads.

Fig. 8: Latency: (a) light / (b) heavy load.

path resolution operations, further improving throughput by 1.18× to 1.24× over Tectonic. However, under high concurrency, path resolution remains a bottleneck because InfiniFS does not reduce the number of RPCs per request.

Namespace-Centric metadata management reduces path resolution overhead. LocoFS further reduces path resolution overhead, improving throughput by 30.5% to 63.6% over InfiniFS except for the Web workload. Fig. 7a shows that for the Web workload, which has shallow directory hierarchies, path resolution overhead is minimal, resulting in similar performance between InfiniFS and LocoFS.

Speculative permission checking eliminates the path resolution bottleneck. Despite these optimizations, all the above systems still perform step-by-step path resolution, which becomes the dominant bottleneck under high concurrency. In contrast, SPMeta's speculative path resolution drastically reduces this overhead by enabling single-RPC lookups for most requests. This shifts computational resources from path resolution to actual metadata processing. As a result, SPMeta achieves 2.19× to 2.86× higher throughput than LocoFS across different workloads.

Fig. 7(f-j) presents the aggregate throughput under read-write mixed workloads. Compared to the read-only scenarios, all methods exhibit a significant decline in throughput due to the higher processing overhead imposed by metadata write operations. Under these workloads, the relative performance of different systems shifts: InfiniFS and LocoFS achieve nearly identical throughput in the Annoy scenario (Fig. 7g), while in the Web scenario (Fig. 7f), InfiniFS outperforms LocoFS by 25.1%. This is because both LocoFS and SPMeta dedicate one metadata server exclusively to path resolution, reducing the number of servers available for handling actual metadata operations—a trade-off that becomes more pronounced in write-intensive workloads. Nevertheless, SPMeta consistently maintains the highest throughput across most scenarios, achieving improvements of 44.9% to 122.4% over the second-best approach. This demonstrates that optimizing path resolution yields substantial performance benefits even under read-write mixed workloads.

C. Latency Breakdown Analysis

We break down the latency overhead across different metadata processing stages. We excluded CephFS from this comparison in subsequent experiments due to its weaker scalability and different building blocks, instead focusing on the metadata management differences between Tectonic, InfiniFS, LocoFS, and SPMeta. We use the Annoy namespace under read-only workloads with two load levels: low load (1 thread per client node) and high load (100 threads per client node).

Metadata Processing Overhead(Process). Fig. 8a visualizes the latency breakdown, where the gray bars represent the processing overhead. As shown in Fig. 8a under light load, Tectonic, LocoFS, and SPMeta exhibit similar metadata processing overhead at approximately 150 μs, consistent with the cost of a single remote Pebbles read operation. InfiniFS, however, incurs higher overhead because it concurrently issues multiple requests within a single metadata operation. Although this parallelism improves throughput, it increases both the processing burden on metadata servers and network concurrency, thereby inflating the latency of the Process stage.

Path Resolution Overhead(Path). The path resolution stage reveals more significant performance differences. Tectonic suffers from a significant performance bottleneck due to its complete destruction of namespace locality. Resolving a path of length k requires k RPCs and database reads, resulting in millisecond-level latency that substantially exceeds other systems. Although InfiniFS also requires k RPCs, its ability to issue multiple path resolution requests concurrently significantly reduces latency compared to Tectonic. LocoFS achieves lower overhead than InfiniFS's distributed path resolution by performing path resolution on a centralized metadata server.

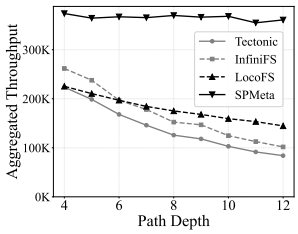

Fig. 9: Comparison under different path depths.

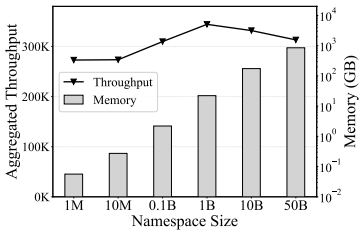

Fig. 10: Scalability with varying namespace sizes.

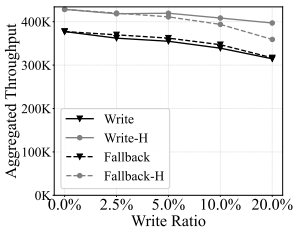

Fig. 11: Thpt. with varying write / fallback ratio.

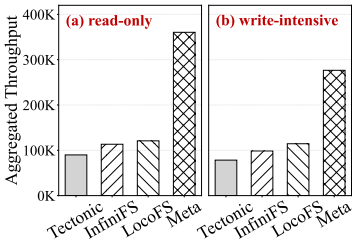

Fig. 12: End-to-end performance with data operations enabled.

SPMeta delivers the best performance by completing path resolution with only a single RPC while bypassing the otherwise cumbersome permission checking process. Compared to baselines, SPMeta reduces path resolution latency by 31.22% to 86.0% and overall metadata latency by 16.8%.

While Tectonic remains limited by its sequential resolution logic, interesting shifts occur for InfiniFS and LocoFS. For InfiniFS, the advantage of concurrent resolution diminishes; the network throughput limits exacerbate queuing delays, keeping its path resolution latency (white bar) high. For LocoFS, a distinct scalability bottleneck emerges in the processing stage. As indicated by the significantly elongated gray bar in Fig. 8b, the centralized server responsible for path resolution becomes overloaded by complex permission checking, causing processing latency to spike compared to the light-load scenario. In contrast, SPMeta maintains a low path resolution overhead even under high load by leveraging speculative permission checking and the efficient MART data structure, ensuring metadata scalability.

D. Sensitivity and Scalability Analysis

In this subsection, we investigate the sensitivity and scalability of SPMeta through a series of microbenchmarks using synthetic workloads. We systematically evaluate the sensitivity of metadata performance to three critical factors: file path depth, namespace scale, and metadata write / lookup fallback ratios. Finally, we enable data I/O operations to assess the end-to-end performance of SPMeta.

Impact of Path Depth. We first analyze the sensitivity of metadata throughput to file path depth by varying the depth from 4 to 12. As shown in Fig. 9, Tectonic, InfiniFS, and LocoFS exhibit a common characteristic: their aggregate throughput declines linearly as the path depth increases. At a shallow depth of 4, InfiniFS achieves the highest throughput among the baselines due to its superior concurrency. However, as the depth increases, the overhead of recursive resolution accumulates. Both InfiniFS and Tectonic, utilizing hash-based partitioning, suffer similar degradation trends. In contrast, LocoFS experiences a relatively milder drop due to its dedicated directory management nodes, surpassing InfiniFS when the depth exceeds 7. Unlike these baselines, SPMeta's path resolution is extremely lightweight, making its metadata throughput nearly insensitive to path depth variations. Across all experiments, SPMeta consistently maintains high aggregate throughput, achieving improvements ranging from 42.8% to 149.4% over the second-best baseline. These results show that SPMeta's design effectively eliminates the path depth bottleneck that plagues traditional metadata managements.

Scalability with Namespace Size. To evaluate SPMeta's scalability under large-scale namespaces, we employed an additional cloud server (Intel Xeon Platinum 8566C, 1 TB DRAM) to host the Path Node, connected to the 4 MDSs from our baseline cluster. We generated synthetic namespaces with a fan-out of 10, scaling the dataset from 1 million to 50 billion files. Fig. 10 shows the memory consumption and aggregate throughput. SPMeta exhibits linear memory growth but remains efficient, managing 50B files with less than 1 TB of memory. Interestingly, throughput increases by 26.1% as the namespace scales from 1M to 1B. This stems from SPMeta's hash distribution: larger namespaces reduce hash skew, balancing the load across MDSs. From 1B to 50B, throughput declines slightly by 9.0% due to deeper directory levels and larger index overheads. Nevertheless, performance at 50B exceeds the 1M baseline, demonstrating robust scalability even under extreme sizes.

Impact of Metadata Write and Fallback Ratio. We evaluated SPMeta under varying metadata write (e.g., *create*) and speculation fallback ratios (0% to 20%) using 500 and 1000 (labeled as -H in figure) client threads, as shown in Fig. 11. While throughput declines as ratios increase, SPMeta maintains robust performance in both scenarios, albeit with distinct bottlenecks. For write operations, the overhead stems from prolonged MDS processing; consequently, increasing client concurrency (Write-H) effectively masks this latency, significantly recovering throughput. Notably, SPMeta maintains high performance even at a 20% write ratio, which significantly exceeds real-world production averages [14], confirming its robustness across diverse workload mixtures. In contrast, increasing the fallback ratio directly diminishes the fast-path advantage. The decline in throughput is significant and cannot be mitigated by higher concurrency, yet the system maintains robust performance levels. Theoretically, even in the worst case (100% fallback), SPMeta degrades to standard hash-based partitioning, ensuring a safe performance baseline.

End-to-End Performance with Data I/O. Finally, we evaluated end-to-end performance with data operations using two production-derived workloads [13]: a *read-only small-file* workload and a *data-write-intensive* workload. Fig. 12 shows that SPMeta maintains a substantial advantage in the metadata-

TABLE IV: Memory consumption (GB) comparison between raw ART and MART across different datasets.

Dataset	# Entries	Raw ART		MART		
		# nodes	Mem	# nodes	Mem	Mem/B
User	1.6M	2.6M	0.43	2.1M	0.34	211.44
SCR4	5.1M	7.2M	1.24	5.9M	1.00	197.99
Annoy	7.4M	11.6M	2.01	9.3M	1.55	211.74
Projs	14.1M	22.6M	4.28	18.1M	3.32	235.65

bound small-file scenario. In the write-intensive scenario, while throughput declines across all systems due to data overheads, SPMeta still outperforms the nearest competitor by 58.6%. This confirms SPMeta's effectiveness even when metadata operations are interleaved with data I/O.

E. MART Analysis

In this subsection, we compare our Go-based MART implementation against the original ART in the context of namespace management. Table IV presents the memory consumption of raw ART and MART for storing namespace reachability metadata across four real-world, large-scale namespace snapshots. MART's value-embedded inner nodes eliminate a substantial number of nodes, particularly leaf nodes. This optimization directly translates to fewer pointer dereferences during queries and reduced memory consumption, with MART achieving 19.6-22.5% memory savings across all datasets. The rightmost column in Table IV shows the extrapolated memory usage for storing one billion entries. MART consistently requires approximately 200 GB per billion entries across all namespaces. Since directories typically represent only 10% of total file count [12], [14], [28], MART can efficiently cache all reachability information for namespaces with tens of billions of files in memory, enabling fast speculative checking.

VII. Related Work

Metadata Decoupling and Disaggregation. More than two decades ago, pioneering DFSs such as GFS [4], HDFS [15], and MooseFS [37] introduced the design principle of decoupling metadata from data, storing metadata on dedicated MDS for efficient processing. This architectural choice has profoundly influenced subsequent generations of DFS, including CephFS [26], Lustre [38], and CubeFS [39], all of which cache metadata in dedicated nodes to accelerate metadata operations. Based on decoupling, recent systems such as CalvinFS [40], HopsFS [18] and ADLS [9] have further disaggregated metadata storage from metadata computation. These systems leverage specialized distributed database systems [21], [22] to implement stateless metadata services, enabling better scalability and fault tolerance.

Scaling Metadata Out. Metadata has been identified as a primary bottleneck in modern DFSs [41], [42], motivating extensive research on scalable metadata management. Early approaches such as CephFS [16], IndexFS [17], and HopsFS [18] partition metadata across multiple MDSs using subtree-based strategies to preserve namespace locality. However, these methods suffer from load imbalance and hotspots. Subsequent work employs heuristics or learning-based algorithms to improve load balance [19], [20], [43], [44], but scalability remains limited. Hash-based partitioning [7], [45] balances load, while CFS [13] further eliminates its transaction overhead [46]. Though common in large DFSs, it introduces significant path resolution overhead since path components reside on different metadata servers. SPMeta builds upon the insights from prior work, but further eliminates path resolution overhead through speculative permission checking.

Mitigating Path Resolution Overhead. Several approaches have been proposed to reduce path resolution overhead in DFSs. LazyHybrid [47] and Duplex [48] replicate permission metadata to accelerate path resolution, while FileScale [8] stores full paths. However, their significant storage and update overhead poses challenges for highly consistent and available production systems. InfiniFS [12] parallelizes path resolution requests across metadata servers, but its performance degrades under high load. Like SPMeta, LocoFS [24] and Mantle [14] use a centralized path resolution node, which can become a bottleneck. SPMeta avoids this via speculative permission checking, safely bypassing expensive checks with minimal metadata to keep CPU and I/O overhead low.

Other Related Work. Recent work has explored hardware acceleration for metadata management. DFSs like SingularFS [49] and Octopus [50] leverage persistent memory to accelerate metadata operations, while SmartNIC-based approaches [51], [52] offload metadata processing to network cards. Emerging CXL [53] and memory tiering [54] technologies enable massive memory capacities, which have been leveraged to scale metadata management in DFS [55]. These optimizations are orthogonal to SPMeta's design and can be combined to achieve further performance gains.

VIII. Conclusion

Hierarchical namespaces incur expensive recursive path resolution, which has become a major bottleneck that limits the scalability and performance of modern distributed file systems. This paper presents SPMeta, a scalable metadata service that addresses this challenge through two key techniques. First, SPMeta introduces *speculative permission checking*, which safely bypasses expensive path resolution while preserving POSIX permission semantics. Second, SPMeta proposes MART, a metadata-optimized trie structure that efficiently indexes metadata to accelerate metadata indexing. Our evaluation shows that SPMeta effectively alleviates the path resolution bottleneck, consistently delivering superior metadata performance across five real-world namespaces. Future work will extend our speculative reachability approach to distributed object storage. Addressing its trillion-scale namespaces and distinct permission models is critical for accelerating the high-concurrency data loading phases of AI/LLM workloads.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported by National Nature Science Foundation of China under Grant No. U25B2020.

AI-Generated Content Acknowledgement

The authors confirm that all technical contributions, ideas, and experimental designs presented in this paper are original work. We used an LLM (Claude-4.5-Sonnet) to assist with grammar checking, language polishing, and proofreading throughout the manuscript to improve clarity and readability. No AI tools were used to generate technical content, figures, tables, or experimental results. All claims, analysis, and conclusions are the authors' own work.

References

[1] W. An, X. Bi, G. Chen, S. Chen, C. Deng, H. Ding, K. Dong, Q. Du, W. Gao, K. Guan *et al.*, "Fire-flyer ai-hpc: A cost-effective software-hardware co-design for deep learning," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–23.

[2] J. Xu, J. Kang, M. Dong, M. Liu, L. Zhang, S. Guo, Z. Qiu, M. You, Z. Tian, A. Yu, T. Ding, X. Hu, and H. Chen, "Falconfs: Distributed file system for large-scale deep learning pipeline," 2025. [Online]. Available: https://arxiv.org/abs/2507.10367

[3] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell, "Metadata traces and workload models for evaluating big storage systems," in *IEEE 5th International Conference on Utility and Cloud Computing (UCC'12)*, 2012.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[5] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: a proposal for an exascale storage system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE, 2016, pp. 585–596.

[6] Q. Li, L. Chen, X. Wang, S. Huang, Q. Xiang, Y. Dong, W. Yao, M. Huang, P. Yang, S. Liu *et al.*, "Fisc: A large-scale cloud-native-oriented file system," in *21st USENIX Conference on File and Storage Technologies (FAST'23)*, 2023, pp. 231–246.

[7] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, M. Shuey, R. Wareing, M. Gangapuram, G. Cao *et al.*, "Facebook's Tectonic filesystem: Efficiency from exascale," in *19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 217–231.

[8] G. Liao and D. J. Abadi, "FileScale: Fast and elastic metadata management for distributed file systems," in *Proceedings of the 2023 ACM Symposium on Cloud Computing(SoCC'23)*, 2023, pp. 459–474.

[9] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos *et al.*, "Azure Data Lake Store: a hyperscale distributed file service for big data analytics," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, 2017, pp. 51–63.

[10] S. Yi, S. Sun, L. Peng, Y. Sun, M.-C. Yang, Z. Cao, Q. Li, M. Jung, K. Zhou, and J. Zhang, "Biza: Design of self-governing block-interface zns afa for endurance and performance," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 313–329.

[11] S. Yi, X. Pan, Q. Li, Q. Li, C. Wang, B. Mao, M. Jung, and J. Zhang, "ScalaAFA: Constructing User-SpaceAll-Flash array engine with holistic designs," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 141–156.

[12] W. Lv, Y. Lu, Y. Zhang, P. Duan, and J. Shu, "InfiniFS: An efficient metadata service for large-scale distributed filesystems," in *20th USENIX Conference on File and Storage Technologies (FAST'22)*, 2022, pp. 313–328.

[13] Y. Wang, Y. Wu, C. Li, P. Zheng, B. Cao, Y. Sun, F. Zhou, Y. Xu, Y. Wang, and G. Xie, "CFS: Scaling metadata service for distributed file system via pruned scope of critical sections," in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*, 2023, pp. 331–346.

[14] J. Li, B. Cao, J. Jian, C. Li, S. Han, Y. Wang, Y. Wu, K. Chen, Z. Yin, Q. Chen *et al.*, "Mantle: Efficient hierarchical metadata management for cloud object storage services," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025, pp. 928–943.

[15] A. Hadoop, "Hadoop distributed file system," http://hadoop.apache.org, 2006, accessed Jan 15, 2025.

[16] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, 2004.

[17] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[18] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling hierarchical file system metadata using newsql databases," in *15th USENIX Conference on File and Storage Technologies (FAST'17)*, 2017, pp. 89–104.

[19] Y. Wang, C. Li, X. Shao, Y. Chen, F. Yan, and Y. Xu, "Lunule: an agile and judicious metadata load balancer for CephFS," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*, 2021, pp. 1–16.

[20] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg, "Mantle: A programmable metadata load balancer for the Ceph file system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, 2015.

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*, 2012, pp. 1–12.

[22] M. Developer, "MySQL NDB Cluster Reference Guide," https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html, 2022, accessed Jan 15, 2025.

[23] F. O. Source, "RocksDB," https://rocksdb.org, 2012, accessed Jan 15, 2025.

[24] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "LocoFS: a loosely-coupled metadata service for distributed file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*, 2017.

[25] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 2013, pp. 38–49.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. Seattle, WA: USENIX Association, Nov. 2006.

[27] S. R. Walli, "The POSIX family of standards," *ACM Stand.*, vol. 3, no. 1, pp. 11–17, 1995. [Online]. Available: https://doi.org/10.1145/210308.210315

[28] D. Manno, J. Lee, P. Challa, Q. Zheng, D. Bonnie, G. Grider, and B. Settlemyer, "Gufi: fast, secure file system metadata search for both privileged and unprivileged users," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.

[29] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2018.

[30] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 16, no. 8, pp. 1992–2004, 2023.

[31] Y. Wang, C. Tang, Z. Wang, and H. Chen, "Sindex: a scalable learned index for string keys," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 17–24.

[32] Y. Yang and S. Chen, "Lits: An optimized learned index for strings," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3415–3427, 2024.

[33] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "Hot: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 521–534.

[34] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017, pp. 497–514.

[35] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[36] C. J. Morrone, "MDtest," https://github.com/MDTEST-LANL/mdtest, 2003.

[37] MooseFS, "MooseFS – a petabyte distributed file system," https://moosefs.com, 2008, accessed Jan 15, 2025.

[38] Lustre, "Lustre file system," http://www.lustre.org, 2003, accessed Jan 15, 2025.

[39] H. Liu, W. Ding, Y. Chen, W. Guo, S. Liu, T. Li, M. Zhang, J. Zhao, H. Zhu, and Z. Zhu, "CFS: A distributed file system for large scale container platforms," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, 2019, pp. 1729–1742.

[40] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems," in *13th USENIX Conference on File and Storage Technologies (FAST'15)*, 2015, pp. 1–14.

[41] K. McKusick and S. Quinlan, "GFS: evolution on fast-forward," *Communications of the ACM (CACM)*, vol. 53, no. 3, pp. 42–49, 2010.

[42] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebook's warm BLOB storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014, pp. 383–398.

[43] Y. Wang, P. Zhang, F. Yang, K. Zhou, and C. Li, "Loadm: Load-aware directory migration policy in distributed file systems," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[44] Y. Wang, W. Tang, L. Meng, L. Li, and J. Wu, "Origami: Efficient ml-driven metadata load balancing for distributed file systems," in *Proceedings of the 54th International Conference on Parallel Processing*, 2025, pp. 22–32.

[45] Lustre, "Lustre metadata service," https://wiki.lustre.org/Lustre_Metadata_Service_(MDS), 2017, accessed Jan 15, 2025.

[46] J. Nemoto, T. Kambayashi, T. Hoshino, and H. Kawashima, "Oze: Decentralized graph-based concurrency control for long-running update transactions," *Proceedings of the VLDB Endowment*, vol. 18, no. 8, pp. 2321–2333, 2025.

[47] S. A. Brandt, E. L. Miller, D. D. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*. IEEE, 2003, pp. 290–298.

[48] C. Dong, F. Wang, Y. Yang, M. Lei, J. Zhang, and D. Feng, "Low-latency and scalable full-path indexing metadata service for distributed file systems," in *2023 IEEE 41st International Conference on Computer Design (ICCD'23)*. IEEE, 2023, pp. 283–290.

[49] H. Guo, Y. Lu, W. Lv, X. Liao, S. Zeng, and J. Shu, "SingularFS: A billion-scale distributed file system using a single metadata server," in *2023 USENIX Annual Technical Conference (ATC'23)*, 2023, pp. 915–928.

[50] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (ATC'17)*, 2017, pp. 773–785.

[51] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, 2021, pp. 756–771.

[52] J. Xu, M. Dong, Q. Tian, Z. Tian, T. Xin, and H. Chen, "Asyncfs: Metadata updates made asynchronous for distributed filesystems with in-network coordination," *arXiv preprint arXiv:2410.08618*, 2024.

[53] W. Tang, Y. Han, T. Ai, G. Li, B. Yu, and X. Yang, "Yggdrasil: Reducing network i/o tax with (cxl-based) distributed shared memory," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 597–606.

[54] W. Tang, Y. Wang, Y. Wang, and J. Wu, "Leave no one behind: Fair and efficient tiered memory management for multi-applications," in *Proceedings of the 54th International Conference on Parallel Processing*, 2025, pp. 699–709.

[55] X. Xu, X. Xie, X. Qiao, L. Tian, Q. Wu, W. Gu, and L. Xiao, "Bridging metadata service and cxl: A metadata-grained and directory-aware storage engine for distributed storage systems," in *2025 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2025, pp. 1–12.