InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems

Wenhao Lv and Youyou Lu, *Department of Computer Science and Technology, BNRist, Tsinghua University;* Yiming Zhang, *School of Informatics, Xiamen University;* Peile Duan, *Alibaba Group;* Jiwu Shu, *Department of Computer Science and Technology, BNRist, Tsinghua University and School of Informatics, Xiamen University*

https://www.usenix.org/conference/fast22/presentation/lv

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems

Wenhao Lv[†] Youyou Lu[†] Yiming Zhang[‡] Peile Duan[↟] Jiwu Shu[*†‡]

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University*
[‡]*School of Informatics, Xiamen University* [↟]*Alibaba Group*

Abstract

Modern datacenters prefer one single filesystem instance that spans the entire datacenter and supports billions of files. The maintenance of filesystem metadata in such scenarios faces unique challenges, including load balancing while preserving locality, long path resolution, and near-root hotspots.

To solve these challenges, we propose INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. It includes three key techniques. First, INFINIFS decouples the *access* and *content* metadata of directories, so that the directory tree can be partitioned with both metadata locality and load balancing. Second, INFINIFS designs the *speculative path resolution* to traverse the path in parallel, which substantially reduces the latency of metadata operations. Third, INFINIFS introduces the *optimistic access metadata cache* on the client-side, to alleviate the near-root hotspot problem, which effectively improves the throughput of metadata operations. The extensive evaluation shows that INFINIFS outperforms state-of-the-art distributed filesystem metadata services in both latency and throughput, and provides stable performance for extremely large-scale directory trees with up to 100 billion files.

1 Introduction

Modern datacenters for fast-expanding businesses often contain huge numbers of files, which can easily exceed the capacity of one single instance of current distributed filesystems [23, 27, 35, 36, 39, 42]. Currently, a datacenter is often divided into relatively smaller clusters, each of which runs a distributed filesystem instance separately. However, it is more desirable to manage the entire datacenter with one single filesystem instance, which provides global data sharing, high resource utilization, and low operational complexity. For example, Facebook introduced the Tectonic distributed filesystem to consolidate small storage clusters into one single instance that contains billions of files [28].

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

Scalable and efficient metadata service is crucial for distributed filesystems [14, 22, 23, 25, 28, 31–33, 36, 41]. As modern datacenters often contain tens or even hundreds of billions of files, using one extremely large-scale filesystem to manage all the files brings severe challenges to the metadata service. First, directory tree partitioning is challenging to achieve both high metadata locality and good load balancing, as the directory tree expands and the workloads are diverse. Second, the latency of path resolution could be high, as the file depths are deep in extremely large-scale filesystems. Third, the overhead of coherence maintenance for client-side metadata cache becomes overwhelming, as extremely large-scale filesystems usually need to serve a large number of concurrent clients.

This paper presents INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. In order to address the challenges mentioned above, INFINIFS distributes the filesystem directory tree and accelerates metadata operations with the following designs.

First, we propose an *access-content decoupled partitioning* method to achieve both high metadata locality and good load balancing. The key idea is to decouple the access metadata (name, ID, and permissions) and content metadata (entry list and timestamps) of the directory, and further partition these metadata objects at a fine-grained level. Specifically, we first group each directory's access metadata with its parent, and content metadata with its children, thereby achieving high metadata locality. Then, we partition these fine-grained groups to different metadata servers with consistent hashing on directory IDs, thereby ensuring good load balancing.

Second, we design a *speculative path resolution* to traverse the directory tree in parallel, which substantially reduces the latency of metadata operations. The key idea is to assign a predictable ID to each directory, so that clients can speculate on the IDs of all intermediate directories, then send lookups for multi-component paths in parallel.

Third, we introduce an *optimistic access metadata cache* to alleviate the near-root hotspots, which achieves scalable path resolution. The key idea is to cache directory access metadata on the client-side to absorb the frequent lookups on near-

root directories, and to invalidate cache entries lazily on the metadata servers with low overhead. Specifically, the directory `rename` and directory `set_permission` operations will send the cache invalidation notification to metadata servers instead of numerous clients, so that each server can validate the cache staleness lazily when processing client metadata requests.

In summary, this paper makes the following contributions:

- We identify the challenges that impair the performance of metadata services in the large-scale scenario (§2).
- We propose a scalable and efficient distributed metadata service, INFINIFS, featured with access-content decoupled partitioning (§3.2), speculative pathname resolving (§3.3), and optimistic access metadata cache (§3.4).
- We implement and evaluate INFINIFS to demonstrate that INFINIFS outperforms the state-of-the-art distributed filesystems in both latency and throughput of metadata operations, and provides stable performance for extremely large-scale directory trees with up to 100 billion files (§5).

2 Background and Motivation

In this section, we first explain why having one single filesystem instance that spans the entire datacenter is desirable (§ 2.1). Then, we discuss the unique challenges for efficient metadata services in such scenarios (beyond billions of files) (§ 2.2). Finally, we analyze the characteristics of metadata access in real datacenter workloads (§ 2.3).

2.1 Large-Scale Filesystem

The filesystem typically provides users with a hierarchical namespace (i.e., a directory tree) to manage files. In the directory tree, each file/directory possesses metadata information. Metadata operations typically involve two critical steps, i.e., path resolution and metadata processing. When a user accesses a file with a pathname, e.g., `/home/Alice/paper.tex`, metadata is accessed as follows: First, the filesystem executes path resolution to locate the target file and check whether the user has the proper permissions; then, the filesystem executes metadata processing to update corresponding metadata objects atomically.

The metadata service is the scalability bottleneck for large-scale distributed filesystems [22, 32, 36]. The distributed filesystem is an important infrastructure component of datacenters [2, 12, 35, 39]. As the number of files inside a datacenter grows rapidly, the metadata service becomes the scalability bottleneck of the distributed filesystems. Currently, datacenters usually consist of a constellation of filesystem clusters. For example, the Alibaba Cloud maintains nearly thousands of Pangu distributed filesystems to collectively support up to tens of billions of files in the datacenter [2]. Facebook also needs many HDFS clusters to store datasets in one single datacenter [28], because each HDFS cluster supports at most 100 million files due to the metadata limitation [36].

However, a large-scale filesystem spanning the entire datacenter is more desirable. For example, Facebook introduced the Tectonic distributed filesystem to consolidate small storage clusters into one single instance [28]. One single large-scale filesystem per datacenter outperforms a constellation of small filesystem clusters in the following aspects:

- *Global data sharing.* One single large-scale filesystem provides a global namespace, enabling better data sharing across the datacenter. In contrast, storing different datasets in separate clusters is inefficient, requiring dedicated data placement and causing data movement. It also complicates the logic of computing service, because related data might be split among separate filesystems.
- *High resource utilization.* Global data sharing can eliminate duplicated data among separate clusters, thus improving disk capacity utilization. Further, one single filesystem allows better resource sharing. In contrast, in the constellation approach, the idle resources in one filesystem cluster could not be reallocated to other clusters.
- *Low operational complexity.* One single large-scale filesystem can significantly reduce the operational complexity, as there is only one system to maintain. In contrast, maintaining thousands of filesystem clusters in a large datacenter (such as Alibaba Cloud) is labor-intensive and error-prone.

2.2 Challenges of Scalable Metadata

One single large-scale filesystem spanning the entire datacenter needs to support billions of files and serve a large number of clients. This brings severe challenges for the metadata service of distributed filesystems, as discussed below.

Challenge 1: *Directory tree partitioning is challenging to achieve both high metadata locality and good load balancing, as the directory tree expands and workloads are diverse.*

Metadata locality is important for efficient metadata processing. A filesystem operation often processes multiple metadata objects. For example, a file creation first locks the parent directory to serialize with directory listing operations [25], then updates three metadata objects atomically, including the file metadata, the entry list, and the directory timestamps. With metadata locality, we can avoid distributed locks and distributed transactions, thus achieving low-latency and high-throughput metadata operations.

Load balancing is important to achieve high scalability. Metadata operations often cause load imbalance in the directory tree. This is particularly true for real-world datacenter workloads where related files are grouped into subtrees [4, 41]. Files and directories from the continuous subtree may be heavily accessed in a short period, causing a performance bottleneck on the metadata server that stores the subtree.

Existing partitioning strategies fail to achieve both high locality and good load balancing in the extremely large-scale scenarios. Managing all files in the datacenter causes the directory tree to expand rapidly in both depth and breadth. Further,

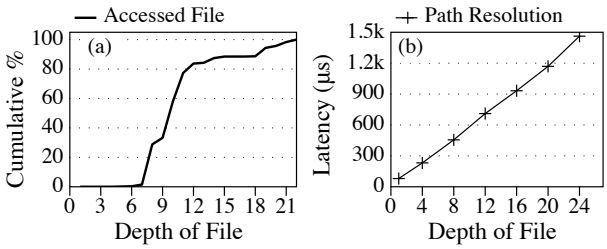

Figure 1: (a) The depth distribution of accessed files in a real-world distributed filesystem workload. (b) The latency of path resolution increases rapidly as the depth of files grows.

File Op	95.8%	Directory Op	4.2%
open/close	54.9%	readdir	93.3%
stat	12.9%	statdir	6.6%
create	10.0%	mkdir	0.1%
delete	12.4%	rmdir	0.1%
rename	9.7%	rename	0.0%
set_permission	0.1%	set_permission	0.0%

Table 1: Ratios of different operations in the real-world workloads from deployed systems. We show the relative ratios of file operations and directory operations separately.

the filesystem faces various workloads with different characteristics, since it backs all datacenter services. This is challenging for the existing directory tree partitioning strategies. Fine-grained partitioning, such as directly hashing metadata objects to servers [25, 37], can achieve load balancing. However, it sacrifices locality and frequently causes distributed locking, introducing expensive coordination overhead and leading to high latency and low throughput [8, 20]. Coarse-grained partitioning, such as grouping a continuous subtree onto the same server [4, 41], preserves locality and avoids cross-server operations. However, it is susceptible to skewed workloads, which lead to the load imbalance.

Challenge 2: *The latency of path resolution could be high, as the file depths are deep in extremely large-scale filesystems.*

The file depth becomes increasingly deeper in extremely large-scale filesystems. Previous filesystems usually assume that the depth of most files in the directory tree is less than 10 [7, 10]. However, we find that the depth of files rapidly increases when consolidating all services into one single filesystem. Figure 1(a) presents the depth distribution of accessed files in a real-world workload (§2.3). We can observe that almost half of the accessed files have a depth of more than 10.

The deep directory hierarchy has a high impact on the filesystem performance. We implement a naive path resolution mechanism based on the design of Tectonic [28], and evaluate the latency of path resolution as the depth grows. Figure 1(b) shows that the latency of path resolution increases linearly with the depth of files. Tectonic partitions directories to different metadata servers based on the directory ID, and consequently, resolving a path at a depth of N requires resolving the $N-1$ intermediate directories, which leads to $N-1$ sequential network requests.

Challenge 3: *The overhead of coherence maintenance for client-side metadata cache becomes overwhelming, as extremely large-scale filesystems usually need to serve a large number of concurrent clients.*

The path resolution needs to traverse the directory tree from the root and check the permissions of all intermediate directories inside the path sequentially. This causes the near-root directories to be read heavily, even for a balanced metadata operation workload. The filesystem throughput will then be bounded by the server that stores the near-root directories. We call this near-root hotspot in this paper. Many distributed filesystems depend on the client-side metadata cache to mitigate the near-root hotspot [13, 23, 30, 31].

We observe that previous client-side cache mechanisms do not work well in large-scale scenarios with numerous clients. For example, the lease-based mechanism grants a lease to every cache entry that will expire after a fixed duration. When the lease expires, corresponding cache entries become invalidated automatically. The lease mechanism is widely used by the NFS v4 [30], PVFS [13], LocoFS [23], and IndexFS [31]. However, the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. This is because all clients have to repeatedly renew their cache entries of the near-root directories for path resolution. As the number of clients increases, such load imbalance at the near-root directories will eventually become a performance bottleneck and impair the overall throughput.

2.3 Characteristics of Real-World Workloads

To understand the characteristics of an extremely large-scale distributed filesystem, we analyze the relative frequency of metadata operations in a real-world workload. We trace metadata operations in production deployments from the Alibaba Cloud, one of the largest cloud providers. We capture workloads from three Pangu filesystem instances that support different services: data processing and analyzing service, object storage service, and block storage service. We merge the workloads from these different services to represent the workload of a large-scale filesystem that spans the entire datacenter. The relative frequencies of metadata operations are shown in Table 1, from which we can observe that:

- File operations account for ~ 95.8% of all operations.
- The directory `readdir` is the most frequent directory operation, accounting for ~ 93.3% of all directory operations.
- Directory `rename` and directory `set_permission` operations rarely occur, accounting for only ~ 0.0083% of all metadata operations.

These insights also refine our design of InfiniFS.

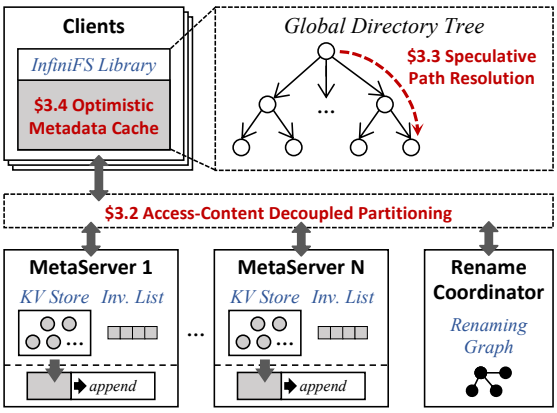

Figure 2: Architecture of INFINIFS.

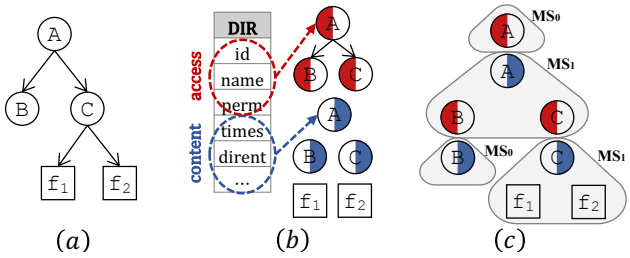

Figure 3: (a) Filesystem directory tree consists of the directory metadata (○) and the file metadata (□). (b) Directory metadata is decoupled into two parts: the access metadata (◐) which contains the name, ID, and permissions, and the content metadata (◑) which contains the timestamps, entry list (i.e., dirent), etc. (c) Directory access metadata is grouped with the parent. Directory content metadata is grouped with the children. These per-directory groups are partitioned to metadata servers (MS_n) by hashing directory IDs.

3 Design and Implementation

We design INFINIFS with three key ideas as below:

- **Decoupling directory metadata.** INFINIFS divides directory metadata into the access state (*name, ID, and permissions*) of the directory itself, and the content state (*entry list and timestamps*) related to the children, so it can be partitioned on a fine grain for load balancing, while still retaining good locality for the metadata processing of common operations such as `create`, `delete`, and `readdir`.
- **Speculating directory IDs in path resolution.** INFINIFS uses a predictable ID for each directory based on the cryptographic hash on the parent ID, the name, and a version number. It enables clients to speculate on directory IDs and launch lookups for multi-component paths in parallel.
- **Invalidating client-side cache lazily.** INFINIFS caches directory access metadata on the client-side to avoid hotspots near the root, thus achieving scalable path resolution. The client uses cache entries for path resolution, agnostic about their staleness. The metadata server lazily validates cache staleness when processing the client metadata requests.

3.1 Overview

INFINIFS is an efficient metadata service for extremely large-scale distributed filesystems. Figure 2 presents the architecture of INFINIFS, which contains the following components:

- *Clients.* INFINIFS provides a global filesystem directory tree that is shared by clients. Clients contact INFINIFS through the user-space library or the FUSE user-level filesystem. They traverse the directory tree via *speculative path resolution* (§3.3), which minimizes latency by predicting directory IDs and parallelizing lookups. Clients use the *optimistic metadata cache* (§3.4) during path resolution to mitigate the excessive read load on near-root directories.
- *Metadata Servers.* The filesystem directory tree is distributed across metadata servers via the *access-content decoupled partitioning* (§3.2), which achieves both high metadata locality and good load balancing. Each server manages metadata objects in the local *key-value store* (i.e., *KV Store*), which typically caches metadata in memory and logs updates in NVMe SSDs for high performance. Metadata servers use the *invalidation list* (i.e., *Inv. List*) to validate client metadata requests lazily.
- *Rename Coordinator.* A central *rename coordinator* is used to process the directory `rename` and directory `set_permission` operations. It checks concurrent directory renames with the *renaming graph* to prevent orphaned loops (§4.1), and broadcasts modification information to invalidation lists of metadata servers.

3.2 Access-Content Decoupled Partitioning

In this section, we first explain why to decouple access and content metadata. Then, we show how to achieve metadata locality via grouping and load balancing via partitioning. Finally, we show how to store metadata in key-value pairs.

Decoupling directory metadata. As discussed in §2.2, previous fine-grained and coarse-grained partitioning failed to achieve both high metadata locality and good load balancing at the same time. Essentially, the root cause is that they treat the directory metadata as a whole. Therefore, when partitioning the directory tree, they have to split the directory either from its parent or its children to different servers, which unintentionally breaks the locality of related metadata.

We analyze the composition of directory metadata and find that directory metadata consists of two independent parts: access and content. As shown in Figure 3(b), *access metadata* contains the directory name, ID, and permissions, which are used to access the directory tree. *Content metadata* contains the entry list, timestamps, etc, which are related to the children. Therefore, we propose to decouple directory metadata into

Metadata Objects	Key	Value	Partitioned by
Directory Access Metadata	*pid, name*	*id, permission*	*pid*
Directory Content Metadata	*id*	*entry list, timestamps, etc.*	*id*
File Metadata	*pid, name*	*file metadata*	*pid*

Table 2: INFINIFS stores metadata objects as key-value pairs. *pid: ID of the parent directory. id: ID of the directory.*

access and content, so as to group and partition two parts independently for both metadata locality and load balancing.

Grouping for locality. We group related metadata objects to the same metadata server to achieve high locality for the metadata processing phase. We first analyze the metadata requirements of all kinds of metadata operations, to determine the related metadata objects during metadata processing. We classify metadata operations into three categories as below:

1. Operations that only process the metadata of the target file/directory, such as `open`, `close`, and `stat`. For example, a file stat will only read the metadata of the target file.
2. Operations that process the metadata of the target file/directory and its parent, such as `create`, `delete`, and `readdir`. For example, a file creation will first insert the file metadata, then lock and update the entry list and timestamps of the parent directory.
3. `Rename` operation is special, as it processes the metadata of two files/directories and their parents.

We observe that most metadata operations (category 1 and 2) require metadata within the target file/directory and the parent during metadata processing. With decoupled directory metadata, we group each directory's content metadata with its subdirectories' access metadata and its files' metadata, as shown in Figure 3(c). In this way, we split the directory tree into independent per-directory groups for later partitioning, while retaining metadata locality for directory `readdir` and file `create/delete/open/close/stat/set_permission` operations. Based on the relative frequency of metadata operations (Table 1), these operations account for ~ 90% of all operations. Thus, INFINIFS achieves high locality for most metadata operations.

Partitioning for load balancing. We further partition the directory tree at a fine-grained level for good load balancing. Based on the locality-aware metadata grouping, we split the directory tree into independent per-directory groups, and then partition these groups to different metadata servers by hashing the directory ID. Such fine-grained hash partitioning effectively load-balances metadata operations [28].

We illustrate the partitioning in Figure 3(c). The metadata group, which contains content metadata of `C` and metadata of `f1` and `f2`, is partitioned to the metadata server 1. In this way, a file `create` under `C` only needs a local transaction in metadata server 1 to insert the new file metadata, then update the entry list and timestamps of `C`. And a `readdir` on `C` only involves metadata server 1 to first lock the directory entry list for isolation, then read the file names from the entry list.

INFINIFS also leverages consistent hashing [29] to map these fine-grained metadata groups to servers, so as to minimize the migration during cluster expanding or shrinking.

Storing. We implement access-content decoupled partitioning with the KV store as the backend storage. The key-value indexing schema is detailed in Table 2, which consists of three kinds of key-value pairs, two for directory access and content metadata, and one for file metadata. To resolve `/A/B/file` (let the IDs of `/`, `A`, and `B` be 0, 1, and 2), we first use $\langle 0, A \rangle$ as the key to get `A`'s access metadata, and find that `A`'s ID equals 1. Then, we use $\langle 1, B \rangle$ to get the `B`'s access metadata, and find that `B`'s ID equals 2. Finally, we use $\langle 2 \rangle$ to get the `B`'s content metadata, and $\langle 2, file \rangle$ to get the file's metadata.

3.3 Speculative Path Resolution

In this section, we first introduce how to generate predictable directory identifiers (§ 3.3.1). Then, we describe how to parallelize path resolution with speculation (§ 3.3.2).

3.3.1 Predictable Directory ID

Here, we present how INFINIFS generates and maintains directory IDs that can be predicted from pathnames later.

1) *Creating.* When creating a new directory, we generate the directory's ID by hashing its *birth triple*: ⟨❶ parent ID, ❷ directory name, ❸ name version⟩, as shown in Figure 4(a). The parent where a directory is created is referred to as the directory's *birth parent*. We use the version to guarantee the universal uniqueness of the birth triple.

2) *Renaming.* When renaming a directory to another location, only the key of its access metadata needs to be updated; its content metadata and ID remain unmodified, thus all descendants' metadata under the directory remain intact.

When a directory is renamed for the first time since creation, its birth parent will record a *rename-list* (RL): ⟨❶ directory name, ❷ name version⟩, and the directory itself will record a *back-pointer* (BP): ⟨❶ birth parent's ID, ❷ name version⟩. The RL of a directory records the subdirectories that were born in that directory but have been moved elsewhere. The BP of a renamed directory (i.e., a directory that has been moved elsewhere from its birth parent) points to its birth

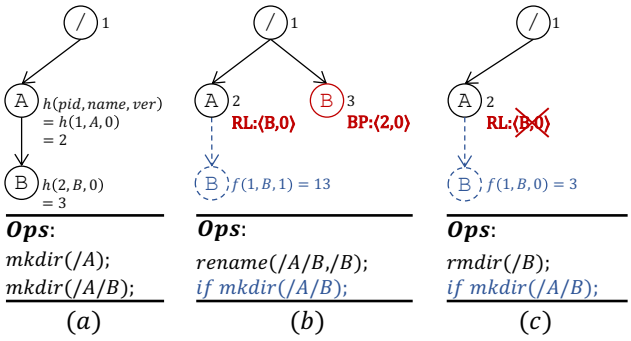

Figure 4: (a) Creating a directory generates its ID by hashing the parent directory ID, the directory name, and the name version. (b) Renaming a directory creates a back-pointer (BP) in itself and an entry in the rename-list (RL) of its birth parent. BP and RL are used for the uniqueness of the directory ID. (c) Deleting the renamed directory erases RL and BP.

parent. We determine the name version with the parent's RL in directory creation. For example, Figure 4(b) shows the `rename(/A/B,/B)` when `B` is renamed for the first time. The key to `B`'s access metadata is updated from `2:B` to `1:B`, while the ID of `B` remains unmodified. `A` records the RL $\langle B,0\rangle$ on the same server as its content metadata. `B` records the BP $\langle 2,0\rangle$ in its access metadata. At this moment, if creating a new `B` under `/A`, its name version should be 1, as the RL of `/A` indicates that there exists a renamed `B` that was born here.

When a directory is renamed again, only the key of its access metadata needs to be updated, while its content metadata, BP, and birth parent's RL remain unmodified. When deleting the birth parent of a renamed directory, the birth parent's access and content metadata are erased, but the birth parent's RL is retained. The RL is only removed through the BP when the renamed directories are deleted.

3) *Deleting.* When deleting a renamed directory, we use the BP of that directory to erase the RL of its birth parent, as shown in Figure 4(c). At this moment, if creating a new `B` under `/A`, its name version returns to the default zero.

ID uniqueness. Here, we first show the predictable directory ID is universally unique, then show hash collisions are very rare, and INFINIFS can detect and handle them properly. A directory ID is generated by hashing the birth triple, so if each birth triple is universally unique, the ID should be universally unique as well, unless a hash collision happens. The filesystem semantic mandates that no two directories inside the same parent have the same name at any time. Without directory renames, ⟨birth parent's ID, directory name⟩ is sufficient to be universally unique. When a directory is renamed elsewhere, a new directory with the same name is allowed to be created within the same parent. We use a version number, as elaborated before, to solve the directory rename problem, thus guaranteeing that each birth triple is universally unique.

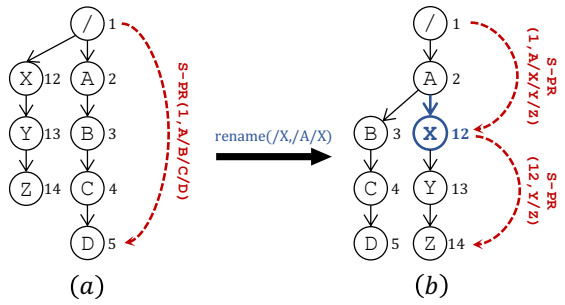

Figure 5: (a) Speculative path resolution (S-PR) reduces the latency of path resolution to nearly one network round-trip time, if correctly predicted. (b) After renaming the directory `/X` to be a child of `A`, resolving `/A/X/Y/Z` requires two rounds, taking about two network round-trip times.

We generate IDs using a cryptographic hash (e.g., SHA-256), so the chance of a collision is very small. As directory ID is the key of each directory's content metadata, we can easily detect a hash collision when inserting the new content metadata during directory creation. We handle a hash collision in the same way as a renaming by using the version number, RL, and BP. In INFINIFS, the collision and rename cases have the same effect on subsequent directory creations and can be distinguished through the RL entry format.

3.3.2 Parallel Path Resolution

Based on the predictable directory ID, a client can conduct path resolution in parallel with the following two steps:

1) *Predict directory IDs.* The client predicts the IDs of all intermediate directories by using the root ID of the path or subpath. It first reconstructs the birth triples with 0 as the version number, then recalculates the hashing results. With the speculated directory IDs, the client reconstructs the keys for all path components.
2) *Lookup in parallel.* The client sends lookup requests for all intermediate directories in parallel. Each lookup request will check the access permissions and compare the speculated ID with the ID stored in the metadata server. If the speculated ID does not match the one on the server, the lookup request returns the true ID to the client.
3) Steps 1) and 2) are repeated until the resolving completes.

Figure 5 shows the speculative path resolution mechanism. If one of the intermediate directories was once renamed to here, the speculated ID of the renamed directory will be wrong (i.e., $h(2,X,0) \neq 12$). However, the lookup request can find `X`'s access metadata using the correct key `2:X`, and returns the directory's true ID to the client. With the true ID of `X`, the client then continues to resolve the subpath under `X`.

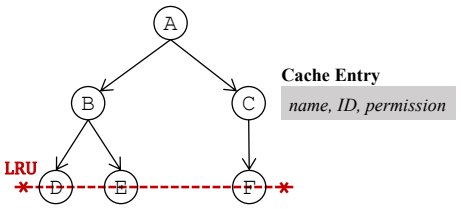

Figure 6: Organization of the access metadata cache on the client-side. Cache entries are organized as a tree. The leaf entries are linked as a least-recently-used (LRU) list.

3.4 Optimistic Access Metadata Cache

In this section, we first show how INFINIFS organizes directory access metadata at the client-side cache. Then, we introduce how clients use cache entries optimistically, and how metadata servers invalidate stale cache entries lazily.

Cache organization. INFINIFS caches only directory access metadata (i.e., directory name, ID, and permissions) on the client side. Cache hits will eliminate lookup requests to near-root directories, thereby avoiding hotspots near the root and ensuring scalable path resolution. As illustrated in Figure 6, INFINIFS organizes cache entries in a tree structure based on the filesystem hierarchy, and links the leaf entries as a least-recently-used (LRU) list. When cache replacement happens, the least recently used leaf entry will be evicted, ensuring that the near-root directories remain cached.

Lazy invalidation. A lot of cache entries become stale after the directory `rename` or directory `set_permission` operation. It is impractical to invalidate stale cache entries on all associated clients during each directory `rename` operation. Because the membership of clients is difficult to manage, and the number of clients can be huge, substantially outnumbering the number of metadata servers.

The lazy invalidation addresses this problem by broadcasting the invalidation information to metadata servers (the number of which is significantly less than clients), so that each server can validate cache staleness lazily when processing client requests. Specifically, a directory `rename` will contact the central rename coordinator to prevent orphaned loops (§4.1), then broadcast the rename information to metadata servers. A single coordination server should be sufficient to handle these infrequent operations, because directory `rename` operations rarely occur (accounting only for ~ 0.0083%) based on the real-world workload studies in §2.3.

As illustrated in Figure 7(a), INFINIFS handles the directory `rename` with the following procedures:

❶ INFINIFS sends the directory rename operation to the rename coordinator, to detect whether this directory rename produces orphaned loops with the in-flight ones. Then, the coordinator assigns each directory rename operation an incremental version number.

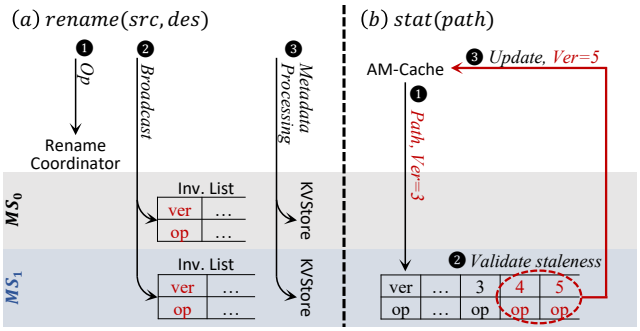

Figure 7: (a) Directory `rename` broadcasts the modification information to all metadata servers (MS_n). (b) Metadata server lazily validates cache staleness when clients issue metadata requests. *ver: version. op: rename operation.*

❷ INFINIFS serializes directory renames with other operations by locking the target directory, so that new accesses to the subtree are blocked until the rename completes. Then, it broadcasts the rename information with its version to metadata servers in parallel, and waits for acknowledgments. The metadata server maintains the rename information in the invalidation list sorted by the version.

❸ INFINIFS moves the directory's access metadata from the source server to the destination server, and updates the RL and BP if the directory is renamed for the first time.

As illustrated in Figure 7(b), INFINIFS validates cache staleness lazily at the server in the following manner:

❶ INFINIFS clients are agnostic about the staleness and optimistically utilize local cache entries during path resolution. Each client has a local version, indicating that its cache has been updated with rename operations before this version. When a client contacts a metadata server, it sends the request along with the pathname and version.

❷ The metadata server validates staleness by comparing the pathname against rename operations in the invalidation list. Only operations between the request's version and the latest version in the invalidation list need to be compared. If the server finds the requested pathname is valid, the request is processed and returned successfully.

❸ If the server finds the request is invalid, it aborts the request and returns the information of these new rename operations. The client then updates the cache and version.

4 Consistency

In this section, we introduce how INFINIFS prevents orphaned loops caused by directory `rename` operations (§4.1), and implements transactional metadata operations (§4.2), so as to guarantee the consistency of the filesystem directory tree.

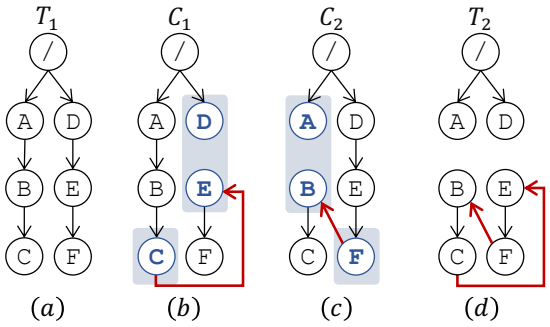

Figure 8: The orphaned loop caused by concurrent directory renames. At time T_1, $Client_1$ tries to rename E to be a child of C, and $Client_2$ tries to rename B to be a child of F, resulting in an orphaned loop at time T_2. The required metadata objects of each rename operation is colored in blue.

4.1 Orphaned Loop

Concurrent directory renames may cause orphaned loops, which lead to server data loss. Figure 8 illustrates such an orphaned loop caused by two directory renames. As shown in Figure 8(a), the filesystem directory tree should be connected and acyclic. In Figure 8(b) and 8(c), $Client_1$ attempts to rename `E` to be the child of `C`, while $Client_2$ attempts to rename `B` to be the child of `F`. The metadata objects required for two renames are completely independent of each other, therefore allowed to execute in parallel. But they lead to an orphaned loop that breaks the directory tree, as shown in Figure 8(d). No client can access any files within the orphaned loop.

InfiniFS addresses the orphaned loop problem by using a central *rename coordinator* to check each directory `rename` operation before its execution. The rename coordinator maintains a *renaming graph*, which tracks the source and destination paths of in-flight directory renames at the present moment. Before allowing a new directory rename operation to proceed, the rename coordinator first verifies whether it leads to an orphaned loop with the in-flight ones. The source and destination paths of a directory rename operation are kept in the renaming graph throughout the renaming procedure, and deleted after the rename operation completes.

Directory `rename` operations rarely occur (accounting for less than ~ 0.0083%) based on the real-world workload studies in §2.3. Therefore, a single rename coordinator should be sufficient to handle directory rename operations.

4.2 Transactional Metadata Operations

Operations in InfiniFS can be classified into three types:

1) *Single-server operations.* These operations include directory `readdir` and file `create/delete/open/close/stat/set_permission`, which are the most frequent operations. They process metadata objects within one single metadata server. Single-server operations leverage the transaction mechanism of the key-value storage backend to guarantee atomicity. When a metadata server restarts after a crash, it recovers the transactions of metadata operations and consults the rename coordinator to update its invalidation list.

2) *Two-server operations.* These operations include directory `mkdir/rmdir/statdir` and file `rename`. They process metadata objects across two metadata servers, requiring distributed transactions to guarantee correct behavior. InfiniFS adopts the two-phase commit protocol [21, 24, 31] for the atomicity of updates across servers. Since clients are unreliable and difficult to track, one of the two metadata servers is chosen to be the coordinator in the transaction. To recover from failures, write-ahead logging is used by both the coordinator and the participant to record the partial state of the transaction.

3) *Directory rename operations.* Directory `rename` and directory `set_permission` operations rarely occur. These operations are delegated to the rename coordinator, which detects orphaned loops, broadcasts the modification to all metadata servers, and processes the target directory metadata across two servers. These operations are implemented with distributed transactions similar to the two-server operations, with the difference that they choose the rename coordinator as the coordinator in the transaction, and broadcast modification at the beginning of the commit phase. If the rename coordinator crashes during the broadcast, it restarts and recovers the transaction by restarting the broadcast to ensure the modification is delivered to all servers at least once.

5 Evaluation

In this section, we use a number of microbenchmarks to evaluate InfiniFS, seeking to answer the following questions:

- How does InfiniFS compare to other distributed filesystems in the metadata performance? (§5.2)
- How do the design features employed in InfiniFS contribute to the overall performance? (§5.3)
- How does InfiniFS perform for extremely large-scale directory trees (up to 100 billion files)? (§5.4)
- How does InfiniFS compare to the lease mechanism in cache efficiency? (§5.5)
- What is the performance of directory `rename` and file `rename` in InfiniFS? (§5.6)
- What is the overhead of speculative path resolution in case of mispredictions? (§5.7)

5.1 Experimental Setup

5.1.1 Hardware Configuration

Experiments are conducted on-premises using 32 client nodes and 32 server nodes with the same hardware configurations,

CPU	Intel Xeon Platinum 2.50GHz, 96 cores
Memory	Micron DDR4 2666MHz 32GB × 16
Storage	RAMdisk
Network	ConnectX-4 Lx Dual-port 25Gbps

Table 3: Hardware configurations.

as shown in Table 3. Each node has two physical ports bonded to a single IP address. All nodes are connected in a network topology with two-level switches. The client nodes can run up to 2048 client processes in parallel, sufficient to saturate the metadata services of tested filesystems.

In our experiments, distributed filesystems are deployed on RAMdisks. Thus, evaluations show the pure performance of different designs, independent of the disk I/O speed. Actually, in production deployments (e.g., Pangu and HDFS), filesystem metadata is typically placed in DRAM and uses a standalone logger to persist metadata updates in NVMe SSDs (with tens of microseconds latency), and thus the latency of metadata operations is mainly affected by the network RPCs (with hundreds of microseconds latency) rather than the local storage devices (NVMe SSDs or RAMdisks).

5.1.2 Software Configuration

Each node runs CentOS 7 with Linux kernel version 4.9.151.

Compared Systems. We choose four state-of-the-art distributed filesystems for comparison, namely, LocoFS [23], IndexFS [31], CephFS [39], and HopsFS [25]. We use CephFS at version 12.2.13, deployed with multiple active MDS daemons, coexisting with OSD daemons on 32 server nodes. We use HopsFS at version 3.2.0.0, deployed with Mysql NDB cluster (version 7.5.3) in the diskless mode on 12 server nodes. Tectonic [28] was not compared because it is closed-source. The throughput of IndexFS is not included, because the metadata servers of IndexFS consistently terminate with errors.

InfiniFS is implemented with the Thrift RPC library [3] for network communication, and RocksDB [5] for the backend KV store. Thrift uses the Linux TCP/IP network stack, leading to a round-trip time of about 60 μs. For InfiniFS, LocoFS, and IndexFS, we set their metadata servers to the Thrift non-blocking server with 8 worker threads, metadata caches to 8MB, and KV stores in asynchronous write mode.

Benchmark. We use the *mdtest* [1] benchmark to evaluate the metadata performance of the aforementioned distributed filesystems. We use OpenMPI at version 3.0.6 to generate parallel mdtest processes across the client nodes. We modify the mdtest benchmark to use the client libraries provided by each distributed filesystem (e.g., libcephfs of CephFS).

Our experiments create files of zero length, like the previous works [16, 23, 25, 31, 43, 46], as we focus on insights into the metadata performance. For full-fledged distributed filesystems such as HopsFS and CephFS, we guarantee the fairness of comparison by ensuring their data paths were not accessed during experiments.

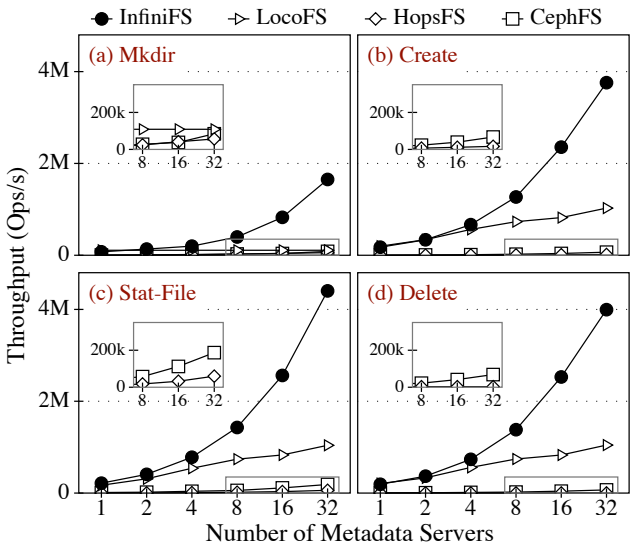

Figure 9: Throughput scalability of metadata operations (`mkdir`, `create`, `stat`, and `delete`). 500 million files.

5.2 Overall Performance

In this section, we compare the overall metadata performance of the aforementioned distributed filesystems. We use mdtest to generate a four-phase workload to measure the performance of directory `mkdir`, file `create`, file `stat`, and file `delete` operations, respectively. These clients operate on a shared directory tree with a depth of 10. Each client handles its own 0.1 million directories and 0.1 million files, which are evenly distributed in the shared directory tree. Clients are initiated at the beginning of each phase.

5.2.1 Throughput

In this section, we evaluate the throughput scalability of metadata operations in different distributed filesystems. We scale the number of metadata servers from 1 to 32, and measure the peak throughput that each filesystem can provide. To obtain the peak throughput, we gradually increase the number of clients until the throughput no longer increases. With 2048 client processes, clients process approximately half a billion files during the experiment.

Figure 9 shows the throughput scalability of directory `mkdir`, file `create`, file `stat`, and file `delete` metadata operations in different distributed filesystems. From the figure, we make the following observations:

1) InfiniFS presents near linear throughput scalability in `mkdir`, `create`, `stat`, and `delete` metadata operations, as the number of metadata servers scales from 1 to 32. InfiniFS achieves high scalability by optimizing the two critical

steps of metadata operations, i.e., path resolution and metadata processing. (a) For the path resolution, INFINIFS caches near-root access metadata at the client-side to absorb the read load on near-root directories, thus the near-root hotspot caused by path resolution will not impair scalability. Besides, INFINIFS partitions file/directory metadata across metadata servers by hashing directory IDs. The fine-grained hash partitioning strategy effectively load-balances metadata accesses, achieving high scalability. (b) For the metadata processing, INFINIFS decouples the directory metadata, then groups the directory access metadata with the parent and the directory content metadata with the children. In this way, metadata processing of file `create`, `stat`, and `delete` only accesses one single server requiring no cross-server coordination, thus, being scalable. Metadata processing of directory `mkdir` requires coordination with only two servers for atomicity, which also scales well with more servers.

2) According to Figure 9(a) and 9(b), the throughput of the `mkdir` operation is much lower than the throughput of the `create` operation in INFINIFS. This is because file creation is implemented using the local transaction protocol with no cross-server coordination, while directory creation requires two-phase locking and two-phase commit protocols. These distributed protocols require expensive coordination between servers, leading to lower throughput.

3) With one metadata server, the throughput of file `create` is 180K ops/sec in INFINIFS, which is slightly lower than LocoFS (200K ops/sec). This is because LocoFS uses a hash-based KV store, which provides higher performance than RocksDB while does not support the scan operation. LocoFS also decouples file metadata into two finer key-value pairs for high throughout. INFINIFS outperforms LocoFS as the number of metadata servers increases. With 32 servers, the directory `mkdir` and file `stat` operations of INFINIFS are 18× and 4× higher than LocoFS. This is because INFINIFS partitions the directory metadata to multiple servers, while LocoFS manages all the directory metadata in one single directory metadata server. When the number of clients increases and the directory tree expands, the single directory metadata server in LocoFS becomes the throughput bottleneck.

4) INFINIFS achieves higher metadata operation throughput than HopsFS and CephFS. For file `create` operations, the throughput of INFINIFS is 73× and 23× higher than that of HopsFS and CephFS, respectively. This is because INFINIFS reduces the latency of metadata operations by resolving paths speculatively in parallel, thus achieving a higher base throughput than HopsFS and CephFS.

5.2.2 Latency

In this section, we evaluate the latency of metadata operations in different distributed filesystems. In the evaluation, we use 32 metadata servers and measure the latency of each metadata operation issued by the client.

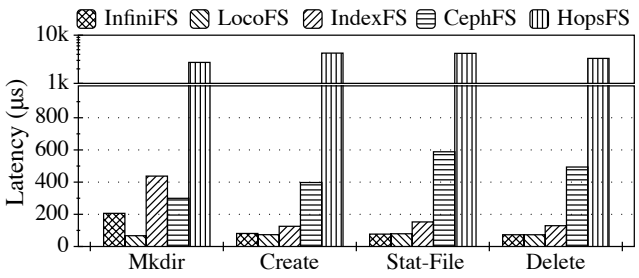

Figure 10: Latency of metadata operations.

Figure 10 shows the average latency of different metadata operations in the evaluated distributed filesystems. From the figure, we make the following observations:

1) For file `create`, file `stat`, and file `delete` operations, INFINIFS achieves comparable low latency with LocoFS, This is because the speculative path resolution and the optimistic access metadata cache in INFINIFS reduce the latency of path resolution, and metadata processing completes within a single server. INFINIFS has higher latency in `mkdir` than LocoFS. This is because INFINIFS partitions directory metadata across metadata servers, causing the directory creation operation to be a distributed transaction. On the contrary, LocoFS manages all directory metadata on one single directory metadata server. Thus, all directory metadata operations can complete in nearly one round-trip time (RTT).

2) INFINIFS achieves lower latency than IndexFS, CephFS, and HopsFS. This is because IndexFS and HopsFS require recursive RPCs to resolve pathnames in case of cache misses, which is slower than our parallelized approach. Besides, CephFS and HopsFS store metadata through external distributed object storage and MySQL NDB cluster, which increases the software stack and results in high latency.

5.3 Factor Analysis

In this section, we analyze how the design features contribute to the latency and throughput by breaking down the performance gap between the Baseline and INFINIFS. We accumulate design features into the Baseline, and measure the latency and throughput of file `create` and directory `mkdir` on 32 metadata servers. For the latency breakdown evaluation, we initiate one mdtest client to create empty files at a depth of 10, and measure the time consumption of path resolution and metadata processing separately. For the throughput breakdown evaluation, we initiate 1024 mdtest clients. Each client creates 0.1 million files or directories that are evenly distributed on a shared directory tree with a depth of 10.

Baseline. We implement the baseline upon the framework of INFINIFS, which partitions the directory tree at the per-directory granularity (like IndexFS, HopsFS, and Tectonic), but without the three design features. As shown in the bars in Figure 11, the path resolution takes 72% of the overall

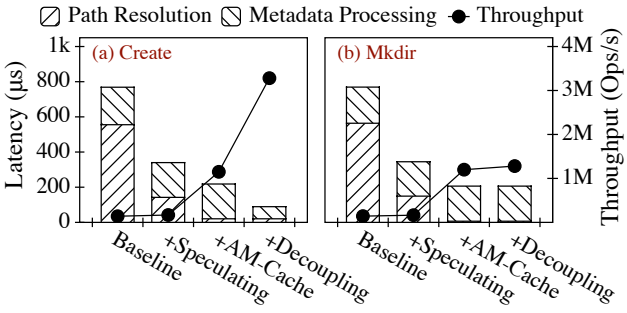

Figure 11: Contributions of design features to the latency (left Y-axis) and throughput (right Y-axis) of INFINIFS. Different segments inside the bar represent the decomposed latency. Design features are accumulated.

metadata operation latency, around 9 RTTs. This is because the path resolution needs to traverse and check permissions of all intermediate directories inside the path sequentially. The metadata processing takes 28% of the latency, around 3 RTTs. This is because both the file and directory creation need to create the target metadata and modify the parent's timestamps and entry list, thus requiring cross-server coordination. As shown in the lines in Figure 11, the metadata operation throughput is very low in the Baseline. This is because all metadata operations will access the near-root directories during path resolution, causing the overall throughput to be limited by the server with the near-root directories.

+Speculating. With the speculative path resolution, the latency of the path resolution is reduced down to 26% of the Baseline. The client leverages asynchronous RPCs to parallelize network requests, but the request processing overhead inside the Linux TCP/IP network stack keeps accumulating. This causes the latency of speculative path resolution to be more than one RTT. The speculative path resolution still faces the problem of near-root bottlenecks, so the metadata operation throughput remains nearly the same.

+AM-Cache. With the optimistic access metadata cache, the heavy read load of path resolution on near-root directories can be absorbed by the client-side cache. Thus, the near-root hotpot will not impair the filesystem throughput. This boosts the overall throughput of file `create` and directory `mkdir` operation to more than 1M ops/sec. Besides, cache hits will further speed up the path resolution, reducing the latency of path resolution to less than one RTT.

+Decoupling. The metadata processing of file `create` and directory `mkdir` involves three metadata objects, including the new file/directory metadata, the entry list, and the timestamps of the parent directory. Without the directory metadata decoupling, these metadata objects are typically located on different metadata servers after the directory tree partitioning. Therefore, metadata processing involves expensive cross-server coordination. With the decoupling, we group the entry list and the timestamps of each directory with the files underneath.

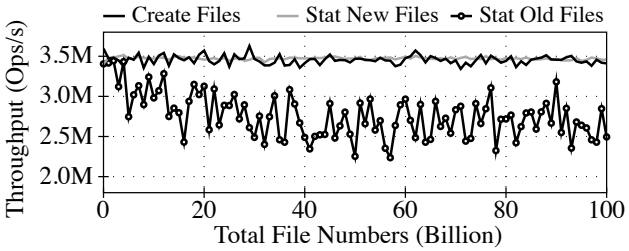

Figure 12: Throughput of INFINIFS with 100 billion files.

In this way, file `create` can process related metadata within a single server with no cross-server coordination. Decoupling boosts the file `create` throughput to 3.3M ops/sec, and reduces its latency to nearly one RTT. However, the throughput and latency of directory `mkdir` remain the same, as it still involves metadata objects across two servers.

5.4 Large-Scale Directory Tree

In this section, we demonstrate that INFINIFS can efficiently support large-scale directory trees. In this experiment, we deploy INFINIFS on 32 metadata servers and initiate 1024 clients. We increase the size of the directory tree up to 100 billion files. Each time we first insert one billion files into the directory tree, then generate a three-phase workload to measure the current performance. Specifically, we measure the throughput of file `create`, file `stat` at the new files, and file `stat` at the old files which are created at the beginning.

Evaluation results are shown in Figure 12. From the figure, we make the following observations:

1) INFINIFS can provide steady performance for file `create` and file `stat` operations (~ 3.5M ops/sec), even when the directory tree expands to a huge size (100 billion files). In real-world datacenters, the Tectonic of Facebook manages 10.7 billion files [28], and the datacenter of Alibaba Cloud maintains up to tens of billions of files. As a result, we believe INFINIFS matches real-world scenarios by supporting 100 billion files with stable performance.

2) The throughput of `stat` old files is lower than that of `stat` new files. This is because INFINIFS stores metadata in RocksDB, which holds key-value pairs in multiple levels of SSTables. Performance drops slightly as the old key-value pairs are mitigated to lower levels. The lower the level, the higher the capacity, and therefore the performance degradation slows down (only ~ 20% at 100 billion files.).

5.5 Cache Efficiency

In this section, we evaluate the efficiency of the lazy invalidation mechanism. To compare with the lease mechanism, we implement a lease version of INFINIFS that uses lease to maintain cache coherence. In the experiments, we deploy

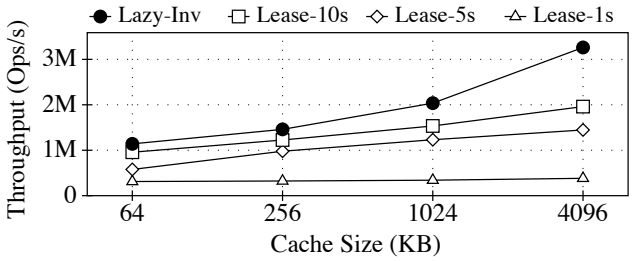

Figure 13: Comparisons of the lazy invalidation with the lease mechanism. The lease expiration time is set to 1s, 5s, and 10s.

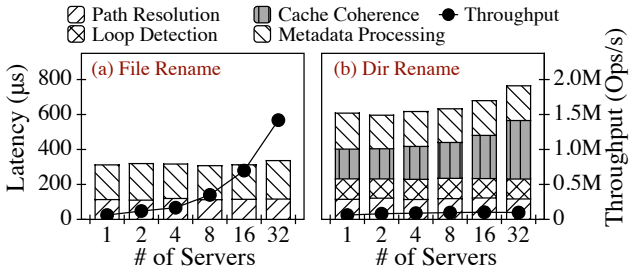

Figure 14: Latency (left Y-axis) and throughput (right Y-axis) of file `rename` and directory `rename`. Different segments inside the bar represent the decomposed latency.

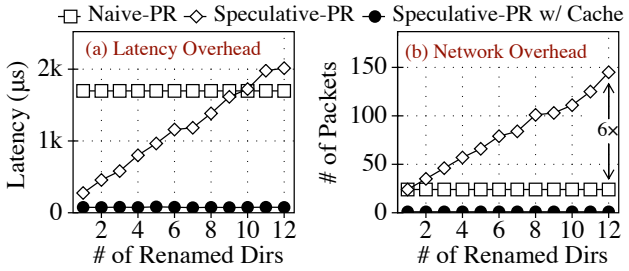

Figure 15: Latency and network packets of three different path resolution approaches. PR: path resolution.

INFINIFS on 32 metadata servers and initiate 2048 clients to `stat` files.

Evaluation results are shown in Figure 13. From the figure, we make the following observations:

1) The lazy invalidation mechanism outperforms the lease mechanism. This is because, in the lease mechanism, cache entries expire periodically regardless of the cache size. Increasing the lease expiration time can improve the throughput, but also increases the latency of all modification operations.

2) As the cache size increases, the throughput increment is more pronounced for the lazy invalidation mechanism than the lease mechanism. This is because the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. All clients have to repeatedly renew their cache entries at the near-root directories for the path resolution procedure. As the number of clients is huge, such load imbalance eventually becomes the performance bottleneck, impairing the overall throughput.

5.6 Rename

In this section, we evaluate the latency and throughput of file `rename` and directory `rename` operations in INFINIFS. In the experiments, we scale the number of metadata servers and measure the peak throughput. We break down `rename` operations to measure the time consumption of each phase.

1) We find that the latency of file `rename` is much lower than directory `rename`. Moreover, as the number of servers increases, the latency of file `rename` remains steady, while the latency of directory `rename` increases slowly. This is because the directory `rename` operation is more complex than the others, involving the following four steps: (1) resolve the source and the destination path, (2) detect whether it leads to orphaned loops, (3) broadcast modification information to metadata servers to maintain cache coherence, and (4) process related metadata across two servers. The latency of directory `rename` grows slowly as the number of servers grows, because the broadcast messages are sent to servers in parallel.

Evaluation results are shown in Figure 14. From the figure, we make the following observations:

2) We find that the throughput of file `rename` scales with the number of servers, while the throughput of directory `rename` does not scale (stays near 10K ops/sec). This is because file `rename` only requires coordination with the source and destination metadata servers for atomicity, thus scales well. However, directory `rename` operations require the central coordination server for orphaned loop detection and broadcasting, thus do not scale. However, these operations rarely occur (accounting for ~ 0.0083%) based on the read-world workload studies in §2.3. Therefore, one single coordination server should be sufficient to handle these infrequent operations.

5.7 Overhead of Misprediction

In this section, we evaluate the overhead of misprediction, including latency overhead and network overhead. A misprediction means INFINIFS mispredicts the ID of a directory during the speculative path resolution. Only the directory `rename` and the hash collision will cause the misprediction. In this experiment, we generate a directory path with a depth of 24, and increase the counts of mispredictions by adding more renamed directories within the path. The renamed directories are evenly distributed inside the pathname. We create 10K files under the path with one client, and measure the average latency and network packets during path resolution.

Evaluation results are shown in Figure 15. From the figure, we make the following observations:

1) Mispredictions increase the latency of the speculative path resolution, but do not affect the naive approach. However,

speculative path resolution still outperforms naive path resolution unless nearly half of the intermediate directories have been renamed, which is rare. Besides, with the optimistic access metadata cache, clients can cache the true IDs of renamed directories, thus avoid mispredictions in the future.

2) Mispredictions increase the number of network packets required for the speculative path resolution. However, trading excess network bandwidth for lower metadata operation latency is worthwhile, as the network bandwidth is not the performance bottleneck. For example, the 25-Gbps CX4 NIC can process 12.3 million packets per second [19]. Thus, 32 metadata servers can process a total of 393.6 million packets per second, which is substantially higher than the peak throughput of metadata operations (~ 4M ops/sec), Besides, INFINIFS comes with the optimistic access metadata cache on the client side, so as to eliminates the extra packets by avoiding mispredictions in the future.

6 Related Work

Efficient distributed filesystems have always been important research topics. Due to the rapidly increasing file quantities, metadata service becomes the performance bottleneck for large-scale distributed filesystems [22, 32, 33, 36].

Directory tree partitioning. Early distributed filesystems, such as GFS [12], HDFS [35], Farsite [11], and QFS [27], distribute file data to multiple data servers while managing all metadata in a single dedicated metadata server. However, they fail in the extremely large-scale scenario with billions of files, because the amount of metadata exceeds the capacity of a single server, and the throughput of metadata operations will be bottlenecked due to the limited resources.

Some distributed filesystems partition the directory tree into subtrees, such as AFS Volumes [15], Sprite Domain [26], and HDFS Federation [9, 35]. Subtree-based metadata partitioning can achieve high metadata locality, but suffers from low scalability due to load imbalance and data migration. Some distributed filesystems partition the directory tree into user-visible partitions and disallow cross-partition renames. As the directory tree expands, organizing and maintaining the static partitioning scheme becomes impractical. CephFS [6, 39–41] partitions metadata into subtrees as well, but when a load imbalance is detected, it migrates hot subtrees across metadata servers. Mantle [34] provides a programmable interface to adjust CephFS's balancing policy for various metadata workloads. However, they suffer from the high overhead of frequent metadata migrations, when workloads are diverse and vary frequently.

Some distributed filesystems partition the directory tree at the per-directory granularity, such as IndexFS [31, 46], HopsFS [25], and Tectonic [28]. Due to the fine-grained partitioning, they can achieve load balancing and good scalability. However, they sacrifice the metadata locality, causing frequent distributed locks and distributed transactions. These distributed protocols impose expensive coordination overhead, resulting in high latency and low throughput [8, 20].

Path resolution. LocoFS [23] stores all directory metadata on a single metadata server, in order to reduce the latency of path resolution, However, it suffers from the single node bottleneck. Some distributed filesystems use the full pathname or the hashing on the full pathname to index files, such as BetrFS [17, 18, 44, 45], Giraffa [37], and CalvinFS [38]. BetrFS uses the full pathname to index files in the local filesystem. Giraffa uses the full pathname as the primary key to the file metadata. CalvinFS locates the file metadata by hashing the full pathname. However, they make the hierarchy semantic to be hard to implement. For example, the directory rename operation becomes prohibitively costly, as it changes the full pathname of all descendants, causing all descendants' metadata must be migrated to new locations.

Client-side metadata caching. HopsFS caches the metadata location information on the server side to parallel path resolution. However, it suffers from the near-root hotspot, as all metadata operations need to read the near-root directories for path traversing and permission checking. LocoFS [23], IndexFS [31], and NFS v4 [30] leverage the lease mechanism to cache both the directory entries and permissions on the client side. However, the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. As the number of clients increases, such load imbalance at the near-root directories will become the performance bottleneck, impairing the overall throughput.

7 Conclusion

This paper presents INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. INFINIFS decouples the directory's access and content metadata, so that the directory tree can be partitioned with both high metadata locality and good load balancing; then parallelizes path resolution with speculation to substantially reduce the latency of metadata operations; and finally, cache access metadata on the client-side optimistically with lazy invalidation. The extensive evaluation shows that INFINIFS provides high-performance metadata operations for large-scale filesystem directory trees.

Acknowledgments

We sincerely thank our shepherd Brent Welch and the anonymous reviewers for their valuable feedback. We also thank Wenhui Yao from the Alibaba Group, Qing Wang, Xiaojian Liao, Youmin Chen, and Zhe Yang from the Tsinghua University for the help on this work. This work is supported by the National Natural Science Foundation of China (Grant No. 61832011, 62022051). This work was also supported by Alibaba Group through Alibaba Innovative Research Program.

References

[1] Mdtest HPC Benchmark. https://sourceforge.net/projects/mdtest/, 2014.

[2] Pangu: The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059, 2018.

[3] Apache Thrift. https://thrift.apache.org/, 2021.

[4] HDFS Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html, 2021.

[5] RocksDB. http://rocksdb.org/, 2021.

[6] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. *ACM Trans. Storage*, 3(3):9–es, oct 2007.

[8] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, Boston, MA, June 2012. USENIX Association.

[9] Dipayan Dev and Ripon Patgiri. Dr. Hadoop: an infinite scalable metadata management for Hadoop—How the baby elephant becomes immortal. *Frontiers of Information Technology & Electronic Engineering*, 17(1):15–31, 2016.

[10] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, page 59–70, New York, NY, USA, 1999. Association for Computing Machinery.

[11] John R. Douceur and Jon Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 321–334, USA, 2006. USENIX Association.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[13] Ibrahim F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux J.*, 2000(80es):5–es, nov 2000.

[14] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[15] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, feb 1988.

[16] Mahmoud Ismail, Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. Scaling HDFS to More than 1 Million Operations per Second with HopsFS. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 683–688. IEEE Press, 2017.

[17] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 301–315, USA, 2015. USENIX Association.

[18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: Write-Optimization in a Kernel File System. *ACM Trans. Storage*, 11(4), nov 2015.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[20] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[21] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander Sandler. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 15–31, USA, 2019. USENIX Association.

[22] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 213–226, USA, 2008. USENIX Association.

[23] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao 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, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[25] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.

[26] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, 1988.

[27] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[28] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[29] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 13, USA, 2011. USENIX Association.

[30] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[31] Kai Ren, Qing Zheng, Swapnil Patil, and Garth 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, page 237–248. IEEE Press, 2014.

[32] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, page 4, USA, 2000. USENIX Association.

[33] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems Are Dead. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 1, USA, 2009. USENIX Association.

[34] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[36] Konstantin V Shvachko. HDFS Scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.

[37] Konstantin V Shvachko and Yuxiang Chen. Scaling Namespace Operations with Giraffa File System. *USENIX; login*, 2017.

[38] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 1–14, USA, 2015. USENIX Association.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[40] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery.

[41] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, page 4, USA, 2004. IEEE Computer Society.

[42] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USA, 2008. USENIX Association.

[43] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. Shardfs vs. indexfs: Replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 236–249, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, page 1–14, USA, 2016. USENIX Association.

[45] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The Full Path to Full-Path Indexing. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 123–138, Oakland, CA, February 2018. USENIX Association.

[46] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, page 1–6. IEEE Press, 2014.