SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server

Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/atc23/presentation/guo

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

SINGULARFS: A Billion-Scale Distributed File System Using a Single Metadata Server

Hao Guo Youyou Lu* Wenhao Lv Xiaojian Liao Shaoxun Zeng Jiwu Shu
Tsinghua University

Abstract

Billion-scale distributed file systems play an important role in modern datacenters, and it is desirable and possible to support these file systems with a single metadata server. However, fully exploiting its performance faces unique challenges, including crash consistency overhead, lock contention in a shared directory, and NUMA locality.

This paper presents SINGULARFS, a billion-scale distributed file system using a single metadata server. It includes three key techniques. First, SINGULARFS proposes *log-free metadata operations* to eliminate additional crash consistency overheads for most metadata operations. Second, SINGULARFS designs *hierarchical concurrency control* to maximize the parallelism of metadata operations. Third, SINGULARFS introduces *hybrid inode partition* to reduce inter-NUMA access and intra-NUMA lock contention. Our extensive evaluation shows that SINGULARFS consistently provides high performance for metadata operations on both private and shared directories, and has a steadily high throughput for the billion-scale directory tree.

1 Introduction

In modern datacenters, the vast majority of distributed file systems are within billions of files, and we call them billion-scale distributed file systems [33]. It is possible to support these file systems with one single metadata server, which typically has enough capacity to hold terabytes of metadata. However, the performance of a single metadata server requires further attention, as metadata operations account for more than half of all file system operations [16, 17, 26].

We find it challenging to store billions of files without sacrificing performance in a single metadata server, so most distributed file systems support billions of files by scaling metadata servers [17, 19, 22, 25, 31]. In this paper, we explore the design space of a billion-scale distributed file system that achieves high performance in a single metadata server.

Remote direct memory access (RDMA) and persistent memory (PM) provide new opportunities for the performance of metadata servers. However, existing solutions fail to fully exploit them and provide desirable metadata performance. Our experiments show that both local PM file systems and distributed file systems have performance limitations in both private and shared directories. Specifically, for the state-of-the-art local PM file system, NOVA [32], its file `create` throughput in a shared directory drops to only 0.14× of its throughput in private directories, even for the million-scale directory tree without the impact of networking.

Exploiting the performance of a single metadata server brings several challenges to the design of file systems. First, the overhead to ensure crash consistency is extremely heavy for a single metadata server that aims to support billions of files. Second, operations in a shared directory, which are common in distributed file systems, suffer from limited parallelism and low performance caused by severe lock contention on the directory `dirents` and inode. Third, The NUMA architecture is under-exploited for file systems, especially for metadata operations in a single metadata server.

This paper presents SINGULARFS, a billion-scale distributed file system using a single metadata server. With only one metadata server, SINGULARFS achieves 8.36M/18.80M IOPS for file `create/stat` operations, which outperforms InfiniFS with 32 metadata servers reported in its paper [19], without sacrificing multi-server scalability. To address the challenges mentioned above, SINGULARFS optimizes the directory tree and the metadata operations with the following designs.

First, we propose *log-free metadata operations* to remove the additional crash consistency overheads from most metadata operations. The key idea is to identify possible metadata inconsistency by leveraging both the single-object update atomicity of the key-value storage backend and the metadata semantic dependency of the parent inode and child inodes.

Second, we design *hierarchical concurrency control* to maximize the parallelism of metadata operations in a shared directory with less synchronization overhead. The key idea of this protocol is to synchronize inode operations in a more fine-grained way. Specifically, inode *writer* uses the per-inode read-write lock to synchronize with other operations, and

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

inode timestamp *updater* and *reader* use a lock-free protocol to do extra synchronization between themselves.

Third, we introduce *hybrid inode partition* to reduce inter-NUMA access and intra-NUMA lock contention. The key idea is to separate the timestamps from the directory inode and group it with the directory's child inodes to the same NUMA node, thus ensuring NUMA locality of file operations. SingularFS further partitions the intra-NUMA data structure to reduce its lock contention.

In summary, this paper makes the following contributions:

- We identify the challenges to fully exploiting the performance of a single metadata server.
- We propose SingularFS, an efficient distributed file system using a single metadata server, featured with log-free metadata operations (§3.2), hierarchical concurrency control (§3.3), and hybrid inode partition (§3.4).
- We implement and evaluate SingularFS to demonstrate that SingularFS outperforms existing distributed file systems in latency and throughput of metadata operations, has comparable latency and throughput with local PM file systems in file operations, provides high throughput scalability in a shared directory, and maintains a steadily high throughput for a billion-scale directory tree (§5).

2 Background and Motivation

2.1 Background

Billion-scale file systems are fundamental building blocks for cloud service vendors and smaller datacenters. Even in some huge datacenters such as Alibaba [19], massive files are managed with small storage clusters, which are within billion-scale. One single metadata server is sufficient to contain all the metadata at this scale, and it has the following benefits compared to using more metadata servers:

- *Implementation and performance.* Distributed transactions and load balancing across metadata servers are avoided, which simplifies the implementation and improves the performance.
- *TCO reduction.* The installation, maintenance, and daily cost of a single metadata server are cheaper than multiple metadata servers.

New network and storage hardware, such as RDMA and PM, provides new opportunities for metadata performance. NVIDIA's latest generation NIC, ConnectX-6, exhibits a speed of 215Mpps for small packets [4]. RedN [23] also illustrates that ConnectX-6 shows a 112M verbs/s throughput in 64B RDMA writes. The only available PM product, Intel Optane DIMM, shows at least 29.06Mops/s and 8.75Mops/s read/write throughput with the access granularity of less than 256B [2] while maintaining memory semantics and data persistence. As inodes are typically tiny in file systems (e.g., 128B in Ext4), achieving high small-granularity access performance allows us to build a high-performance distributed file system with a single metadata server.

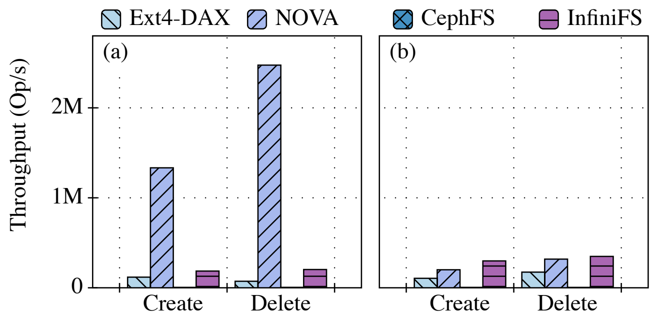

Figure 1: Per-NUMA file `create` and `delete` throughput of local PM file systems and distributed file systems (a) in a private directory for each client, (b) in a shared directory.

2.2 Analysis of Existing Solutions

In this section, we analyze the limitations of existing solutions that discourage them to support a billion-scale distributed file system with a single metadata server.

There are mainly two categories of existing solutions, local PM file systems, and distributed file systems. For each type, we select two typical file systems for comparison, namely Ext4-DAX and NOVA [32] for local PM file systems, and CephFS [31] and InfiniFS [19] for distributed file systems. We gradually increase the number of client threads until each file system achieves the peak throughput. The testbed and configuration details are further described in §5.1.

Figure 1 shows the per-NUMA file `create` and `delete` throughput of the compared file systems with clients operating on private directories and a shared directory, respectively. We make the following observations:

1) The overhead of crash consistency limits the throughput of multi-inode metadata update operations. Ext4-DAX uses write-ahead-logging (WAL) to guarantee its crash consistency, while InfiniFS and CephFS leverage the transaction support of their storage backend. These methods all introduce extra costs for the multi-inode metadata update operations. NOVA reduces the crash consistency cost by using the log-structured metadata architecture, making its file `create/delete` throughput 7.17×/12.19× higher than other file systems with private directories. However, it cannot completely get rid of the journaling overhead when coordinating multi-inode update operations. Also, it introduces extra garbage collection overhead.

2) The lock contention limits the throughput of metadata operations in a shared directory. For file `create` and `delete` operations, NOVA shows 0.17×/0.13× throughput in a shared directory than in private directories. This is because NOVA directly acquires the shared directory's write lock when performing these operations, which aggravates lock contention. The other three file systems show no significant performance

degradation. This is because they either directly use the journal for concurrency control (e.g., Ext4-DAX) or leverage the transaction support of the storage backend (e.g., CephFS and InfiniFS). These methods coordinate all the metadata operations similarly no matter they are in private directories or a shared directory, limiting the performance in both scenarios.

3) File systems fail to scale to multiple NUMA nodes while maintaining NUMA locality of metadata operations. For the compared file systems, NOVA doesn't provide support for multiple NUMA nodes. Although Ext4-DAX can be mounted to a RAID 0 device spanning across the PM DIMMs on all NUMA nodes, the simple striped layout of RAID 0 can not ensure NUMA locality for metadata operations. For InfiniFS and CephFS, they scatter the metadata objects to all the PM DIMMs without specific NUMA-aware partition rules, resulting in low NUMA locality as well.

2.3 Challenges

Based on the three observations in §2.2, we find three challenges to fully exploiting the performance of a single metadata server in distributed file systems, as discussed below.

Challenge 1. *The overhead to ensure crash consistency is extremely heavy for a single metadata server that aims to support billions of files.*

File systems use journaling or the log-structured design to provide crash consistency. In the journaling approach, data is written twice and checkpointed in order. In the log-structured approach, data is written in newly-allocated places, while leaving the old places as garbage. Unfortunately, garbage collection causes high overhead. Intensive prior research aims to reduce the overhead of the crash consistency mechanisms [14, 18, 21]. However, we need to further reduce this overhead to exploit the potential of a single server to support billions of files.

Challenge 2. *Operations in a shared directory, which are common in distributed file systems, suffer from limited parallelism and low performance caused by severe lock contention on the directory* `dirents` *and inode.*

In HPC and big data applications, it is common for the workloads to concurrently access a large shared directory, such as N-N checkpointing [9] and image processing [1]. The concurrency of metadata operations in a shared directory significantly influences the overall performance of these applications. However, concurrent inode `create` and `delete` operations in a shared directory need to update their common parent's directory entries (`dirents`) and inode, which causes high lock contention. Such contention limits the parallelism and performance as demonstrated in §2.2.

The previously-proposed solutions are not suitable for the case of a single metadata server. Previous works use partition strategy either inside `dirents` [34] or between metadata server daemons [22, 31] to increase parallelism. However, partitioning inside `dirents` can't reduce the lock contention of the common parent inode, and partitioning the super directories between metadata server daemons can not be used for one metadata server.

Challenge 3. *The NUMA architecture is under-exploited for file systems, especially for metadata operations in a single metadata server.*

NUMA-aware design is important for metadata performance when a server is used only for metadata storage and processing. On the one hand, NUMA locality is of great importance for fully utilizing the performance potentials of PM. Previous works illustrate that remote PM access introduces a significant performance drop in bandwidth and small-granularity throughput, especially for writes [28, 35], which makes metadata operations more vulnerable compared to data operations. On the other hand, our analysis in §2.2 shows that file systems fail to scale to multiple NUMA nodes while keeping NUMA locality of metadata operations, which is because of their coarse-grained partition strategy such as striping.

Existing methods are not desirable for overcoming this challenge. Randomly scattering the inodes into multiple NUMA nodes can't avoid this issue, as some operations like inode `create` and `delete` need to update multiple inodes, which causes inter-NUMA metadata access. Other methods like in-DRAM cache [28], thread migration [30], or thread delegation [35] all waste extra resources for coordination or data structure maintenance.

3 Design and Implementation

With the overall goal of exploiting the performance of a single metadata server, we design SingularFS with three key design principles as below:

- **Guaranteeing crash consistency without logs.** SingularFS performs most metadata operations without extra crash consistency mechanisms such as logs. This reduces their overheads while still maintaining POSIX semantics.
- **Maximizing parallelism in a shared directory.** By utilizing hierarchical concurrency control, SingularFS maximizes the parallelism of metadata operations in a shared directory with less synchronization overhead.
- **Reducing inter-NUMA access and intra-NUMA lock contention.** SingularFS takes a hybrid approach to metadata partition. This ensures NUMA locality of file operations and reduces lock contention within data structures to further increase the parallelism in a shared directory.

3.1 Overview

SingularFS is a billion-scale distributed file system using a single metadata server, exploiting the single-server performance while not sacrificing multi-server scalability. Figure 2

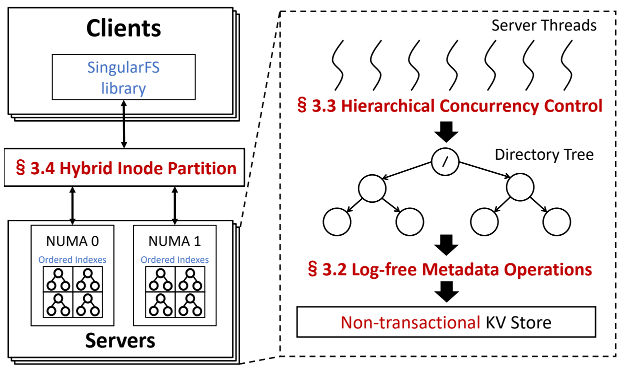
Clients
SingularFS library
§ 3.4 Hybrid Inode Partition
NUMA 0
Ordered Indexes
NUMA 1
Ordered Indexes
Servers
Server Threads
§ 3.3 Hierarchical Concurrency Control
Directory Tree
/
§ 3.2 Log-free Metadata Operations
Non-transactional KV Store

Figure 2: SINGULARFS architecture.

presents the architecture of SINGULARFS, which contains two components, clients and servers. Servers maintain a global file system directory tree inside PM. Clients perform file system operations through POSIX-like interfaces offered by the user-space library. Servers and clients are equipped with RDMA NICs for network communication.

Storage backend. SINGULARFS uses a generic key-value store (KV Store) as its storage backend. The KV Store should be able to execute point queries and prefix matching. Besides, it should guarantee the atomicity of single-object operations at a low runtime cost.

SINGULARFS differs from other file systems with KV Store backends [17, 19, 24, 25] in two ways. First, instead of relying on the transaction provided by the KV Store backends (Non-transactional KV Store in Figure 2), SINGULARFS uses a lightweight approach for metadata operations (§3.2, §3.3). Second, instead of using one ordered KV Store per metadata server, SINGULARFS leverages hybrid inode partition to convert the shared index into multiple ordered indexes scattered to NUMA nodes, aiming at reducing inter-NUMA access and intra-NUMA lock contention (§3.4).

Regular metadata operations. We use this term to refer to metadata operations other than `rename`. Leveraging the timestamp dependency of modified inodes, SINGULARFS uses ordered metadata update to guarantee their crash consistency without logs (§3.2). For concurrency control, SINGULARFS takes a hierarchical method by utilizing both the per-inode read-write lock and lock-free timestamp update (§3.3).

Rename. SINGULARFS uses journaling to guarantee the crash consistency of `rename`. As an optimization, ordered metadata update (§3.2) is adopted to guarantee the crash consistency of the two parent directories involved in `rename`, reducing the number of logged inodes from 4 to 2. SINGULARFS uses strict two-phase locking for concurrency control of `rename`.

Inode. SINGULARFS adds two fields, the born time *btime* and the death time *dtime*, to the inode to identify metadata inconsistencies leveraging the timestamp dependency (§3.2). Besides, SINGULARFS partitions the directory inode to timestamp metadata (`atime`, `ctime`, and `mtime`) and access meta-

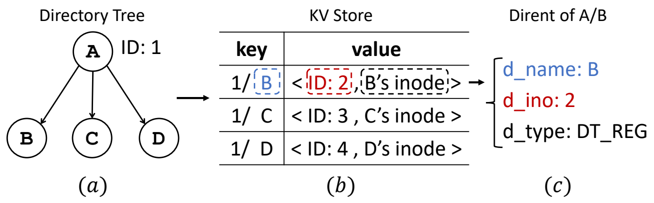

Figure 3: (a) An example of the directory tree. (b) The format of directory A's child inodes in KV Store. For the inode with ID = *i*, the keys of its child inodes share the same prefix *i*. (c) The way to reassemble `dirents` in a `readdir` operation. SINGULARFS first does prefix matching with the target directory's ID to get the child KV pairs, then uses the keys to generate `d_name` and reads the values to get `d_ino` and `d_type`.

data (inode ID, permission, *btime*, *dtime*, etc.), and places the timestamp metadata of the parent directory with its child inodes into the same NUMA node to ensure NUMA locality of file operations (§3.4).

Inodes are all stored in the KV Store backend. Directory access metadata and file inodes are indexed by key <`parent inode ID+name`>, and directory timestamp metadata is indexed by key <`inode ID`>. The root directory has a unique inode ID 0. Path resolution is done by recursively fetching the directory access metadata from the root to the target inode.

Dirent. Directory entries (`dirents`) are omitted from directory metadata blocks but co-located with the keys and values of child inode objects. Thus, `dirent` update and inode update are fused into one key-value update operation. As shown in Figure 3, in a `readdir` operation, SINGULARFS reassembles the `dirents` of the target directory by adopting prefix matching provided by the KV Store backend.

Data management. SINGULARFS employs a decoupled structure for metadata and data. Therefore, existing approaches for data management can be directly adopted by SINGULARFS. Currently, SINGULARFS uses the object store [31] to manage data. It indexes data blocks with key <`inode ID+block no`>.

3.2 Log-free Metadata Operations

In this section, we first analyze and classify the metadata write operations. Then, we demonstrate the timestamp dependency of the parent directory and child inodes and propose the core of log-free metadata operations, i.e., ordered metadata update.

3.2.1 Analysis of Metadata Write Operations

As illustrated in Table 1, metadata write operations in SINGULARFS can be classified into three categories according to the number of modified inodes.

1. *Single-node operations.* These operations only update the target inode itself (e.g., file `open/close/read/write`).
2. *Double-node operations.* These operations update the target inode, as well as the timestamps of its parent directory (e.g., file `create/delete`). Note that in SINGULARFS,

Operation Type	Metadata Write Operations	Modified Inodes		
		Target Inode	Parent Directory	Other Inodes
Single-node	`open/close/read/write/chmod/chown/utimens`	●		
Double-node	`mkdir/rmdir/create/delete`	●	●	
Rename	`rename`	●	●	●

Table 1: Classification of metadata write operations according to the modified inodes.

parent `dirents` are co-located with the child inode objects (§3.1), so there is no need to update the `dirents` separately for these operations.

3. *Rename operation.* This operation updates the original inode, the new inode, and their parent directories.

According to the real workloads shown in InfiniFS [19] and HopsFS [20], regular metadata operations, which include read operations, *single-node operations*, and *double-node operations*, account for more than 90% of all file system operations. Based on this observation, we design log-free metadata operations to guarantee the crash consistency of regular metadata operations.

Since the crash consistency of *single-node operations* can be directly guaranteed by the KV Store backend, in this section, we seek to guarantee the crash consistency of *double-node operations* without incurring additional overheads.

3.2.2 Ordered Metadata Update

As *double-node operations* set the parent directory's `ctime` to the target inode's *btime* or *dtime*, we observe that the timestamps of the parent directory and child inodes have the following dependency:

For an inode d, d.`ctime` $\geq \max(c.btime, c.dtime)$*, where c is any of d's child inodes.*

Based on this observation, we update the metadata in order, to guarantee the crash consistency of *double-node operations* without incurring the logging overhead.

inode creation. As shown in Figure 4(a), inode creation includes two atomic steps. First, we insert the inode with *btime* = t_0, where t_0 is the current timestamp. Second, we set the `ctime` and `mtime` of its parent directory to t_0.

inode deletion. As shown in Figure 4(b), inode deletion includes three atomic steps. First, we invalidate the target inode and set its *dtime* to the current timestamp t_0. Second, we set the `ctime` and `mtime` of its parent directory to t_0. Finally, we physically remove the target inode from KV Store.

Crash recovery. When a crash occurs between the two steps in inode creation, or between step 1 and step 2 in inode deletion, the inconsistency can be identified by checking if the maximum *btime* and *dtime* of child inodes > `ctime` of the parent directory, and fixed by setting the `ctime` and `mtime` of the parent directory to the maximum value. When a crash occurs between step 2 and step 3 in inode deletion, the inconsistency can be identified by checking if the target inode is invalid and fixed by physically removing the invalid inode.

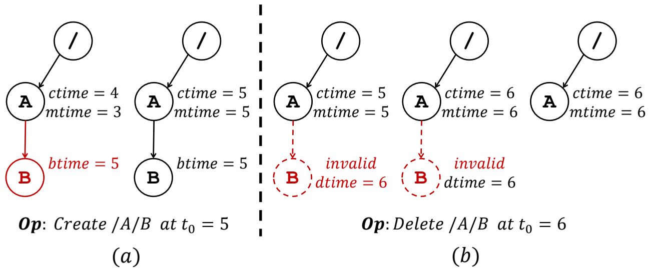

Figure 4: The process of inode creation and inode deletion in log-free metadata operations. (a) Inode creation includes two steps. First, insert the target inode with *btime* = current timestamp t_0. Then, update the timestamps of the parent directory to t_0. (b) Inode deletion includes three steps. First, mark the target inode as invalid and set its *dtime* to the current timestamp t_0. Then, update the timestamps of the parent directory to t_0. Finally, physically remove the target inode. The inconsistency of the directory tree is marked with red lines.

In order to detect and fix all the inconsistencies mentioned above, the most straightforward approach is to scan the whole directory tree. However, this process can be costly. SingularFS detects and fixes the inconsistency lazily when the inconsistent directory is accessed, as described in §4.

3.3 Hierarchical Concurrency Control

In traditional file systems, it is challenging to maximize the parallelism of *double-node operations* in a shared directory, as they cause contention over the parent `dirent` and timestamps. In SingularFS, the update of the parent `dirent` is co-located with the update of the child inode, whose concurrency is guaranteed by the KV Store backend. Therefore, the remaining challenge lies in the concurrent timestamp update.

We observe that *double-node operations* only modify the parent directory's `ctime` and `mtime`, whose size is 16B in total. Therefore, the concurrency control of the timestamp updates could be executed in a lock-free manner by leveraging the 16B atomic compare-and-swap instruction.

We divide operations on a target inode into three categories, *updater*, *writer*, and *reader*. *updater* contains inode update operations that involve only the target inode's `ctime` and `mtime`, and *writer* contains all other update operations. *reader* contains all inode read operations. For example, *double-node operations* like file `create/delete` are both *writers* of the target inode and *updaters* of the parent directory, and file `stat` is a *reader* of the target inode.

```
1  struct Inode {
2      ...
3      uint64 ctime; # aligned at a 16B boundary.
4      uint64 mtime;
5      ...
6  }
7
8  def writer(this: Inode *):
9      this->write_lock() # sync with other operations.
10     modify_inode(this)
11     this->write_unlock()
12
13 def updater(this: Inode *, timestamp: uint64):
14     this->read_lock() # sync with writer
15     while True:
16         # acquire timestamp snapshot.
17         cur = { this->ctime, this->mtime }
18         nxt = { timestamp, timestamp }
19         # update ctime and mtime atomically.
20         if (cur[0] >= timestamp or
21             cmpxchg16b(&this->ctime, cur, nxt)):
22             break
23     this->read_unlock()
24
25 def reader(this: Inode *):
26     this->read_lock() # sync with writer
27     while True:
28         # use ctime as the version number.
29         last_ctime = this->ctime
30         compiler_barrier()
31         inode = *this
32         compiler_barrier()
33         # OCC-like method.
34         cur_ctime = this->ctime
35         if cur_ctime == last_ctime:
36             break
37     this->read_unlock()
38     return inode
```

Algorithm 1: **Hierarchical concurrency control algorithm.**

Algorithm 1 shows the hierarchical concurrency control algorithm among *writer*, *updater*, and *reader*.

Writer-other synchronization. Synchronization between *writers* and other operations is handled by the per-inode read-write locks. The *writer* acquires the write lock to guarantee exclusive access to the target inode (line 9). *Updaters* and *readers* acquire the read lock to avoid concurrent *writer* doing inode update or remove operations (line 14, line 26).

Updater-updater synchronization. With the purpose of minimizing the length of the critical section, *updaters* use atomic instructions to synchronize between themselves in a lock-free manner. As the *updater* only updates the ctime and mtime of the target inode, they are placed in a 16B-aligned block (line 3), and the *updater* uses `cmpxchg16b` to atomically update `ctime` and `mtime` to the maximum of the timestamp parameter and the original value (lines 15-22). Specifically, the *updater* first acquires the current snapshot of `ctime` and `mtime` (line 17). If the current `ctime` is not less than the timestamp parameter, there is no need to update the timestamps as they have been updated by another *updater* with a later timestamp (line 20). If the current `ctime` is less than the timestamp parameter and there is no concurrent *updater* in the critical area (line 21), then update the timestamps atomically.

Updater-reader synchronization. We observe that `ctime` monotonically increases when *updaters* modify the inode, so it has the same semantic as a version number. Based on this observation, we adopt optimistic concurrency control (OCC) to synchronize between *readers* and *updaters* without locks. Specifically, `ctime` is fetched by the *reader* before and after getting the whole inode (line 29, line 34). The *reader* validates the inode by comparing the two `ctimes` (line 35).

3.4 Hybrid Inode Partition

In this section, we first show how to partition the inodes among NUMA nodes and execute multi-object directory operations introduced by the partition. Then, we propose the intra-NUMA data structure.

3.4.1 Inter-NUMA Inode Partition

Partition for NUMA locality of file operations. As file operations account for the majority of all metadata operations [19], we seek to guarantee their NUMA locality.

For *single-node* file operations, we delegate metadata requests to the corresponding NUMA node to ensure their NUMA locality. However, this does not work for *double-node* file operations (`create/delete`), as the two related inodes may reside in different NUMA nodes. Fortunately, in SingularFS, these operations only modify the parent directory's `ctime` and `mtime`, so NUMA locality of them can be guaranteed by grouping the parent directory's `ctime` and `mtime` with the child file inodes to the same NUMA node.

Therefore, SingularFS partitions the directory inode into timestamp metadata (`atime`, `ctime`, and `mtime`) and access metadata (inode ID, permission, *btime*, *dtime*, etc.). For each directory, SingularFS aggregates the directory timestamp metadata, its child file inodes, as well as its child directories' access metadata into a metadata group. The objects within a metadata group are placed in the same NUMA node. SingularFS uses consistent hashing to distribute the groups to NUMA nodes of the metadata server, thus achieving NUMA locality of file operations.

Multi-object directory operations. As the directory metadata is partitioned into two parts (i.e., two objects in the KV Store backend), the KV Store backend cannot intrinsically guarantee crash consistency for updates on the two parts. To solve this problem without incurring extra cost, SingularFS performs these operations in certain orders.

1. *Directory* `mkdir` *and* `rmdir`. Three objects are involved in these operations, specifically the target directory's access metadata and timestamp metadata, as well as the parent directory's timestamp metadata. Note that when a directory is created, all its timestamps are set to its *btime*, which is inside the access metadata of the target directory. Leveraging this, SingularFS creates the target directory's timestamp metadata after creating its access metadata. After a crash, the target directory's timestamp metadata is re-generated with its access metadata.

2. *Directory `set_permission`.* Two objects are involved in this operation, specifically directory access metadata and timestamp metadata. To guarantee the crash consistency, we expand the semantic of *btime* to the maximum of born time and last `set_permission` time. When executing `set_permission`, SingularFS first updates the target directory's permission fields and its *btime* at the same time. Then, SingularFS updates its `ctime`. On crash recovery, SingularFS identifies whether the directory's *btime* is greater than its `ctime` and fixes them if needed.

3.4.2 Intra-NUMA Inode Partition

Partition for less lock contention. To generate `dirents` of a particular directory, SingularFS relies on the range query operation, which is supported by the ordered index. However, typical B+-tree-based ordered indexes suffer from high lock contention caused by traversing and node splits when updates prevail in workloads. To reduce such intra-NUMA lock contention while keeping the functionality of the range query, inside each NUMA node, SingularFS uses the hash algorithm to scatter all metadata uniformly to *n* ordered indexes, where *n* is a configurable parameter.

When executing point queries, SingularFS queries the value from the key's corresponding index calculated by its hash. When performing range queries, such as in directory `readdir` operation, SingularFS makes range queries in all the indexes and merges the results.

Optimization for removal. Directory `rmdir` requires determining whether the target directory is empty. As the `dirents` are co-located with the child inodes, this process can be implemented using prefix matching. However, with intra-NUMA inode partition, we would have to perform prefix matching in all *n* ordered indexes, which could be costly.

We optimize this process by adding a `num_dents` variable to the metadata of each directory, identifying the number of `dirents`. The crash consistency of this variable is easily guaranteed because it can be recovered by executing a complete prefix matching for the directory.

`num_dents` should meet two requirements for concurrency safety. First, it must allow concurrent updates from *updaters*. To achieve this goal, directory *updaters* use `fetch_and_add` and `fetch_and_sub` to synchronize with each other. Second, its value must be correct when a `rmdir` occurs. Since `rmdir` is a *writer* of the target inode, there can be no concurrent *updaters* when SingularFS is executing `rmdir`. Therefore, `num_dents` will keep unchanged during `rmdir` and it is safe for us to use its value to judge the directory's emptiness.

4 Crash Recovery

Algorithm 2 shows the overall crash recovery algorithm for a single directory inode in SingularFS. It mainly consists of three aspects: Parent timestamp recovery (lines 9-14) and invalid inode removal (lines 15-16) in log-free metadata operations, directory timestamp metadata recovery (lines 4-7) in inter-NUMA inode partition, and `num_dents` maintenance (line 2, line 17) in intra-NUMA inode partition.

```
1  def recover(ino: Inode):
2      ino.num_dents = 0
3      # re-create incomplete timestamp metadata
4      recreate_timestamp_meta(ino)
5      # inconsistency caused by set_permission
6      if ino.btime > ino.ctime:
7          ino.ctime = ino.btime
8      for c in ino.child_inodes:
9          if (c.valid and c.btime > ino.ctime) or
10            (!c.valid and c.dtime > ino.ctime):
11             # crash before timestamp update.
12             ino.ctime = max(c.btime, c.dtime)
13             ino.mtime = max(c.btime, c.dtime)
14             redo(c) # create / delete.
15         else if not c.valid:
16             remove(c) # invalid inode
17         ino.num_dents += 1 # maintain num_dents
```

Algorithm 2: **Crash recovery algorithm for inode `ino`.**

SingularFS detects and fixes the inconsistent directory only when it is accessed. Specifically, SingularFS maintains a global `restartCnt` variable, which increases by one at each restart. Inside each directory inode, we keep a local `restartCnt` padded in the directory read-write lock. Locking a directory also sets its local `restartCnt` to the global value. After a crash, an inconsistent directory will contain an outdated `restartCnt`. Such inconsistency will be detected and fixed according to Algorithm 2 on the next access.

5 Evaluation

In this section, we evaluate SingularFS to answer the following questions:

- How does SingularFS compare to local PM file systems and distributed file systems utilizing RDMA and PM on metadata performance? (§5.2)
- How does SingularFS scale on concurrent racing metadata operations in a shared directory? (§5.3)
- How do the techniques employed by SingularFS impact its metadata performance? (§5.4)
- How does SingularFS perform with a billion-scale directory tree? (§5.5)
- What is the performance of `rename` in SingularFS? (§5.6)
- What is the end-to-end performance of SingularFS? (§5.7)
- What is the overhead of crash recovery? (§5.8)

5.1 Experimental Setup

Hardware configuration. In the experiments, unless otherwise stated, we use one server node and one or two client nodes. The metadata server and data server are co-located. Each server node has two Intel Xeon Gold 6330 CPUs, with 28 cores per socket. Hyperthreading is disabled on servers, for it aggravates contention and degrades the overall performance.

Each socket has four 128GB Intel Optane DC Persistent Memory (DCPMM) DIMMs, 256GB DRAM, and one 200Gb/s Mellanox ConnectX-6 NIC. The Optane DIMMs are configured in App Direct mode.

Each client node has two Intel Xeon Platinum 8360Y CPUs, with 36 cores (72 threads) per socket, so as to issue as many requests as possible to saturate the metadata server. Each socket has 256GB DRAM and one 200Gb/s Mellanox ConnectX-6 NIC. All the clients and servers are connected with a Mellanox QM8790 switch. For ConnectX-6 NICs, the driver and firmware versions are OFED 5.5-1.0.3.2 and 20.32.1010.

Compared systems. We use two types of baseline file system in the comparison, specifically local PM file systems and distributed file systems.

For local PM file systems, we choose Ext4-DAX and NOVA [32]. For these file systems, we clear the VFS cache before each directory and file `stat` operation to mitigate its impact and ensure that the acquired performance reflects the actual metadata performance of the file system.

For distributed file systems, we choose CephFS [31] and InfiniFS [19]. For CephFS, we use version 15.2.16 with RDMA enabled. The latency of CephFS is obtained by running it in RDMA mode, while the throughput is obtained by running it in IPoIB mode because we find that CephFS will always crash if run in RDMA mode for a relatively long time. For InfiniFS, we add support for RDMA and multiple NICs with eRPC [13]. Since InfiniFS uses RocksDB [5] storage backend, we run it on Ext4-DAX mounted on top of a RAID 0 device spanning across all the PM DIMMs.

We use P-Masstree [15] as the ordered index of SingularFS and set the per-NUMA index number *n* to 8. For SingularFS and InfiniFS, we use 56 worker threads executing in-line requests, and clients connect to them in a round-robin way. SingularFS stores its data objects in PM for fairness.

Benchmark. We use mdtest v3.3.0 provided by IOR [3] to evaluate the metadata performance of the aforementioned file systems. We use OpenMPI v4.1.2 to generate parallel mdtest client processes, which are placed on the metadata server for local PM file systems and scattered across all the client nodes for distributed file systems. We use the POSIX interface for local PM file systems. For distributed file systems, we use either the intercepted POSIX syscall [6] or the client libraries (e.g., libcephfs of CephFS). Our experiments create files of zero length like the previous works [17, 19, 25] because we focus on the insights into metadata performance. For end-to-end benchmarks, we use Filebench [27] to evaluate the overall performance.

For lack of multi-NUMA support in some of the compared file systems (e.g., NOVA and CephFS), we compare their per-NUMA throughput instead of overall throughput to guarantee the fairness of the comparison. For Ext4-DAX and NOVA, we limit their CPU and PM resources to a single NUMA node and use the results directly as their per-NUMA performance.

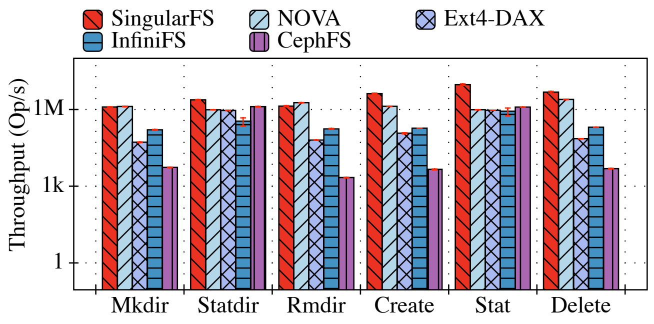

Figure 5: Per-NUMA throughput of metadata operations in private directories. The Y-axis is log-scaled.

For CephFS, we restrict its server-side PM and NIC resources and directly use the results as its per-NUMA performance. For SingularFS and InfiniFS, we do not limit their hardware resources and average their throughput results to get their per-NUMA performance.

5.2 Metadata Performance

In this section, we compare the overall metadata performance of the file systems mentioned above. We use mdtest to measure the performance of directory `mkdir/stat/rmdir` and file `create/stat/delete` operations. Each client handles 2 million directories and 2 million files in its private directory. For CephFS and InfiniFS, we get their peak performance by modestly reducing the directory and file quantity per client process, as they show a relatively low performance at the aforementioned scale.

5.2.1 Throughput

In this section, we evaluate the throughput of metadata operations in different file systems. We gradually increase the number of client processes to achieve the peak per-NUMA throughput for each file system.

Figure 5 shows the per-NUMA metadata throughput of different file systems in private directories. From the figure, we make the following observations:

1) SingularFS outperforms local PM file systems in file `create` and `delete` operations and outperforms the distributed file systems by more than an order of magnitude. SingularFS achieves 3.13×/1.93× and 22.49×/23.57× higher throughput for file `create/delete` operations than NOVA and InfiniFS. This is because the log-free metadata operations in SingularFS removes the extra transaction logic from the critical path of these operations, which saves both PM bandwidth and the CPU cycles spent writing logs and waiting for persistence. Inter-NUMA inode partition also guarantees NUMA locality of file operations, reducing inter-NUMA communication. These two designs make SingularFS fully exploit the single-server performance of file metadata write operations.

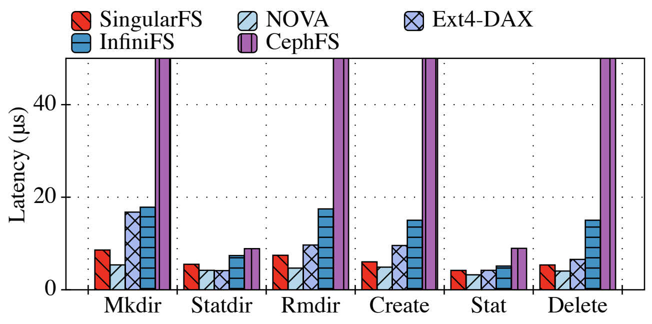

Figure 6: Latency of metadata operations. Note that the result of SingularFS, CephFS, and InfiniFS includes the network delay, while the result of Ext4-DAX and NOVA doesn't.

2) The throughput of directory `mkdir` and `rmdir` is much lower than file `create` and `delete`, but still comparable with local PM file systems (0.96×/0.73× than NOVA) and much higher than distributed file systems (7.82×/7.77× higher than InfiniFS). This is because these operations need to write both the target directory's access metadata and its timestamp metadata, which are not guaranteed to be in the same NUMA node. However, these metadata update operations are transformed into several simple KV writes by utilizing log-free metadata operations, accelerating this process.

3) SingularFS outperforms local PM file systems and distributed file systems in the case of metadata read operations. For file `stat` operation, SingularFS achieves 9.54×/10.10× higher throughput than NOVA/Ext4-DAX, and 7.54×/10.97× higher throughput than CephFS/InfiniFS. This is because by adopting log-free metadata operations and hierarchical concurrency control, SingularFS separates the transaction logic from KV Store. This enables SingularFS to use a lightweight KV Store backend (e.g., P-Masstree [15]) to accelerate the `stat` operation.

4) SingularFS demonstrates high NUMA scalability. Using inter-NUMA inode partition, SingularFS guarantees NUMA locality for file operations. Although it utilizes all the hardware resources rather than just one NUMA node, its per-NUMA throughput of file operations still outperforms that of file systems running on one NUMA node.

5.2.2 Latency

In this section, we evaluate the latency of metadata operations in different file systems. We launch one mdtest client to measure the latency of each metadata operation.

Figure 6 shows the average latency of different metadata operations of the compared file systems. From this figure, we make the following observations:

1) Compared with local PM file systems, SingularFS achieves comparable latency with Ext4-DAX and NOVA in file operations. This is because SingularFS has a shorter server-side critical path for metadata write operations by leveraging log-free metadata operations and inter-NUMA inode partition. SingularFS also has a low latency for read operations by guaranteeing NUMA locality of file metadata. Even though suffering from the µs-level inherent latency of RDMA, SingularFS still achieves comparable latency with local PM file systems for file operations.

2) Compared with distributed file systems, SingularFS achieves lower latency than CephFS and InfiniFS. This is because with the lightweight KV Store and log-free metadata operations, SingularFS has a more lightweight software stack than CephFS and InfiniFS, contributing to its lower latency.

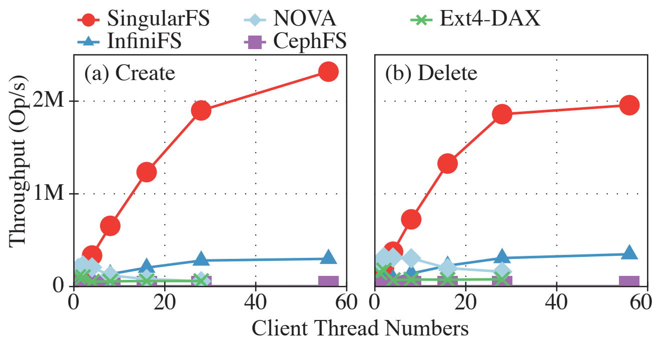

Figure 7: Per-NUMA throughput scalability of file `create` and `delete` in a shared directory.

5.3 Scalability in a Shared Directory

In this section, we evaluate the throughput scalability of concurrent racing metadata operations in a shared directory. In the evaluation process, we gradually increase the number of mdtest clients and get the per-NUMA throughput for file `create` and `delete` operations in a shared directory.

Figure 7 shows the throughput scalability of file `create` and `delete` in a shared directory in different file systems. From the figure, we make the following observations:

1) SingularFS shows much better throughput scalability in a shared directory than other file systems, outperforming them by at least 7.76×/5.61× on file `create/delete` operations. This is because SingularFS leverages hierarchical concurrency control to maximize the parallelism of metadata operations in a shared directory, and adopts intra-NUMA inode partition to reduce lock contention inside the intra-NUMA data structure. These two methods contribute to the shared-directory scalability of SingularFS.

2) SingularFS achieves nearly the theoretical peak performance in a shared directory. The file `create/delete` throughput of SingularFS in a shared directory converges to around 4Mops/s for the whole metadata server, which is close to the per-NUMA peak throughput of SingularFS illustrated in Figure 5. This is because all the inodes in a shared directory are placed in the same NUMA node. Therefore, all the operations are handled in the same NUMA node, and the upper bound for performance is the per-NUMA peak performance of SingularFS.

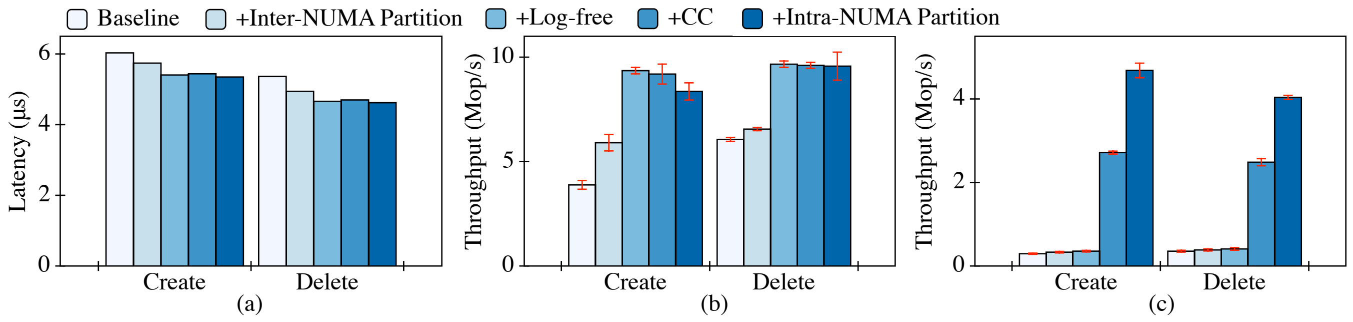

Figure 8: Performance breakdown. (a) Average latency. (b) Throughput in private directories. (c) Throughput in a shared directory. Design techniques are accumulated.

5.4 Factor Analysis

In this section, we analyze how our designs contribute to the latency and throughput by breaking down the performance gap between the *Baseline* and SingularFS. We apply our designs one by one to the *Baseline*, and measure the average latency and throughput of file `create` and `delete`. For the latency breakdown evaluation, we initiate one mdtest client to operate on 2 million files in a directory. For the throughput breakdown evaluation in private directories, we initiate 112 mdtest clients, and each client handles 2 million files in its private directory. For the throughput breakdown evaluation in a shared directory, we gradually increase the client number from 1 to 112 to achieve the peak throughput of each configuration. Each client operates on 2 million files in a shared directory. The results are not averaged to per-NUMA performance.

We implement the *Baseline* based on the framework of SingularFS, but without the key design features. It uses P-Masstree running on Ext4-DAX mounted on top of a RAID 0 device built from all the PM DIMMs, with pessimistic two-phase locking (2PL) and WAL for transaction support. In these three figures, *+Log-free* stands for utilizing log-free metadata operations instead of WAL to guarantee crash consistency. *+CC* represents adopting hierarchical concurrency control instead of simply using 2PL to do concurrency control. *+Inter-NUMA partition* and *+Intra-NUMA partition* are the two design parts of hybrid inode partition.

5.4.1 Private Directories

As shown in Figure 8(a) and Figure 8(b), SingularFS has much higher throughput and lower latency against *Baseline*. Specifically, for file `create` operation, SingularFS achieves 2.15× higher throughput with 1.13× lower average latency. For file `delete` operation, SingularFS achieves 1.58× higher throughput with 1.16× lower average latency. Here, we separately analyze all the design techniques in terms of file `create` workload (file `delete` has the same conclusions).

Inter-NUMA inode partition improves the throughput by 1.52× and reduces the average latency by 1.05×, since it guarantees NUMA locality of file operations, thus reducing the frequency of inter-NUMA PM access.

By using log-free metadata operations to guarantee crash consistency, SingularFS gains another 1.58× and 1.06× improvement in terms of throughput and average latency. This is because WAL is replaced in the critical path with log-free metadata operations, reducing the bandwidth waste of PM for logs and saving the CPU cycles of log persistence.

Hierarchical concurrency control and intra-NUMA partition bring a 10% throughput drop and a minor latency increase since they introduce extra synchronization overhead when there are few conflicts. Besides, the intra-NUMA inode partition has a negative effect on the cache friendliness of the overall data structure, accounting for the throughput drop.

5.4.2 Shared Directory

Figure 8(c) shows the throughput of file `create` and file `delete` in a shared directory. Compared with *Baseline*, SingularFS achieves 15.93×/11.40× higher throughput for file `create/delete` operations separately.

Inter-NUMA inode partition and log-free metadata operations contribute to 1.21×/1.15× throughput increase for file `create/delete` operations by reducing the length of the critical area, which shortens the critical area. However, the major bottleneck still lies in the lock contention of the shared directory. Hierarchical concurrency control mitigates this lock contention, thus increasing the throughput by 7.66×/6.09×. This is the peak throughput of P-Masstree in the case of high lock contention brought by the common prefix. The intra-NUMA inode partition mitigates this lock contention and contributes to another 1.72×/1.62× higher throughput.

5.5 Billion-scale Directory Tree

In this section, we demonstrate that SingularFS can efficiently support the billion-scale directory tree. We repeatedly `create` and `stat` 50 million files per NUMA node to increase the directory tree size until the file system is full.

Figure 9 shows the evaluation results. We make the following observations from the figure:

1) SingularFS delivers a steadily high throughput for file `create` and file `stat` operations with the billion-scale direc-

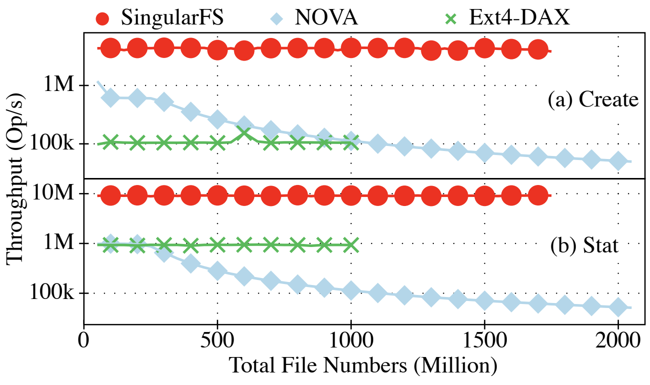

Figure 9: Per-NUMA throughput of file `create` and `stat` for the billion-scale directory tree. The Y-axis is log-scaled.

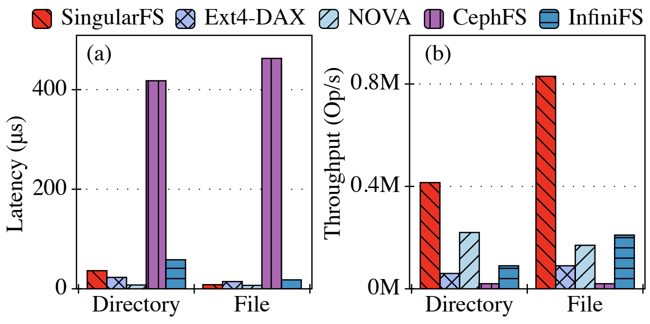

Figure 10: Performance of directory `rename` and file `rename`. (a) Latency. (b) Per-NUMA throughput.

tory tree. Specifically, it shows a steady per-NUMA throughput of ~4.40Mops/s for file `create` and ~9.10Mops/s for file `stat`, which is 41.90×/9.83× higher than Ext4-DAX. Although NOVA shows a relatively high throughput when the directory tree is nearly empty, its throughput is not even as high as Ext4-DAX when there is a billion-scale directory tree.

2) SingularFS shows similar directory tree scalability to local PM file systems. SingularFS supports 1.75 billion files per NUMA node, which is higher than Ext4-DAX (1 billion files) and lower than NOVA (2.05 billion files). The reason why SingularFS supports fewer files than NOVA is that SingularFS uses a KV Store to store the inodes, while NOVA uses per-core linked lists. The index of SingularFS has a higher capacity overhead than NOVA. However, it provides significantly higher performance.

5.6 Rename

In this section, we evaluate the latency and per-NUMA throughput of directory `rename` and file `rename` in SingularFS. In the experiments, each client renames 1 million directories and 1 million files from one directory to another.

Figure 10 shows the overall results. From the figure, we make the following observations:

1) For file `rename`, SingularFS shows comparable latency with local PM file systems and at least 3.95× higher per-NUMA throughput than other file systems. Although SingularFS uses journaling to ensure crash consistency, the journal is lightweight as it only contains the two target KV pairs. Logs of the parent directories are omitted with log-free metadata operations. However, because of the journaling overhead and inevitable inter-NUMA access, the throughput of file `rename` is 0.17×/0.15× of file `create/delete`, lower than a half of their throughput (i.e., theoretical throughput limit).

2) The latency and throughput of directory `rename` are both worse than those of file `rename`. This is because SingularFS adopts the directory metadata cache like the previous works [17, 19, 25]. Directory `rename` causes cache invalidation and path re-resolution, reducing its performance.

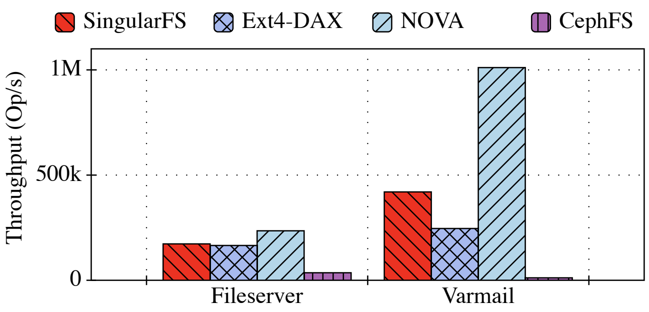

Figure 11: Per-NUMA throughput of Fileserver and Varmail workloads in Filebench.

5.7 End-to-end Performance

In this section, we test the end-to-end performance of SingularFS. Specifically, we run the per-NUMA throughput of Filebench Fileserver and Varmail workloads on SingularFS and baseline systems. We set the file number of Varmail to 100K. The result of InfiniFS is not included as it focuses on metadata service rather than the whole file system.

Figure 11 shows the results. From the figure, we make the following observations:

1) SingularFS outperforms CephFS and Ext4-DAX in both workloads. Specifically, SingularFS outperforms Ext4-DAX by 1.05× in Fileserver and 1.71× in Varmail. With its metadata design, SingularFS gets more performance gains in Varmail, which is metadata-intensive.

2) SingularFS shows 0.74× and 0.41× the throughput of NOVA in Fileserver and Varmail respectively. The lower performance of SingularFS stems from the existing inter-client metadata dependencies (e.g., client 1 creates a file, then client 2 deletes it) within the workload, which makes the high performance of the metadata server hard to be fully exploited. As the metadata server is under-saturated, the latency of metadata operation becomes the primary influence on the overall performance. Because SingularFS is accessed over the network, it has higher metadata operation latency (as discussed in §5.2.2) and thus lower throughput than NOVA.

Method	*Launch*	*Consistent*	*Inconsistent*
Scan	32.1s	5.3µs	-
Lazy	0s	5.3µs	30.8ms

Table 2: Recovery overhead. *Launch*: recovery time during launch. *Consistent*: latency of file `create` that does not require recovery. *Inconsistent*: latency for file `create` to detect and fix the inconsistent parent directory.

5.8 Crash Recovery

In this section, we evaluate the impact of lazy recovery on the latency of metadata operations, compared to scanning the directory tree. In the experiments, the server crash results in one inconsistent directory with 100K files in it. For lazy recovery, SingularFS detects and fixes this inconsistent directory while doing file `create` in it. For scan, the server scans the whole directory tree during launch. The directory tree size is 100M.

Table 2 shows the results. From the table, we make the following observations:

1) The time of lazy recovery is much shorter than scanning the directory tree. This is because SingularFS only scans the 100K files in the inconsistent directory on lazy recovery. However, all the 100M files must be accessed when scanning the whole directory tree, contributing to its high latency. This issue is even more severe in the billion-scale directory tree.

2) The lazy recovery time is still orders of magnitude higher than metadata operations. This is because SingularFS still needs to scan the 100K files in the inconsistent directory during lazy recovery. The recovery time is expected to be lower if the inconsistent directory contains fewer files.

6 Related Work

The efficiency of file systems has always been an interesting and important research topic, both for local PM file systems [10, 11, 32, 35] and distributed file systems [7, 17, 19, 22, 25, 31]. Different from the works mentioned above, SingularFS improves metadata efficiency by optimizing the transactions in metadata operations with ordered updates and improving NUMA locality of metadata operations. In this section, we focus on these two aspects of related work.

Transactions and Ordering in metadata operations. Transactions are a way to guarantee the atomicity and crash consistency of metadata operations. For local PM file systems, BPFS [10] relies on copy-on-write (CoW) and 8-byte in-place atomic updates to provide metadata consistency. PMFS [11], in contrast, uses larger in-place atomic updates with journaling, while NOVA [32] uses a log-structured data structure for metadata consistency. For distributed file systems, HopsFS [20] relies on both row-level locking and the NDB backend for strong metadata consistency semantics, while InfiniFS [19] leverages the transaction mechanism of the key-value storage backend and the two-phase commit protocol separately for local and distributed metadata transactions. CephFS [7, 31] and CFS [29] also leverage their storage backend to guarantee the atomicity and crash consistency of metadata operations. These methods have considerable CPU overhead, and SingularFS mitigates this overhead by using log-free metadata operations to guarantee crash consistency with minimal cost and adopting hierarchical concurrency control to maximize parallelism.

Ordering is another way to support crash consistency. Soft Updates [12] ensures that data and metadata are written to disks in an ordered way, so as to enable recovery after a crash. One of the obstacles to using Soft Updates is the complexity of keeping general orders between different metadata blocks. In comparison, SingularFS only needs to keep the orders of timestamp metadata, and thus is more practical in use.

NUMA-aware file systems. Several recent studies propose approaches for mitigating the NUMA effect in PM file systems. NThread [30] uses thread migration to alleviate the NUMA issues of the PM file systems. Assise [8] uses on-die DMA engines for remote PM writes to bypass hardware cache coherence. OdinFS [35] uses NUMA-aware delegation threads to handle PM access with large granularity. These approaches are mainly for data service in the file system, and can not be easily applied to metadata service, because metadata operations 1) are not more costly than thread migration, 2) have a more complex indexing logic other than simple read/write, 3) have small access granularity. SingularFS uses hybrid inode partition to ensure NUMA locality of file operations while minimizing the extra cost of scheduling.

7 Conclusion

This paper presents SingularFS, a billion-scale distributed file system using a single metadata server. SingularFS uses log-free metadata operations to eliminate additional crash consistency overheads for most metadata operations; then uses hierarchical concurrency control to maximize the parallelism of metadata operations; and finally, takes hybrid inode partition to reduce inter-NUMA access and intra-NUMA lock contention. Our extensive evaluation shows that SingularFS consistently provides high performance for metadata operations on both private and shared directories, and has a steadily high throughput for the billion-scale directory tree.

Acknowledgments

We sincerely thank our shepherd Emmett Witchel for helping us improve the paper. We are also grateful to the anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, U22B2023), and China Postdoctoral Science Foundation (Grant No. 2022M721828).

References

[1] one usage of up to a million files/directory. https://leaf.dragonflybsd.org/mailarchive/kernel/2008-11/msg00070.html, 2008.

[2] Intel® Optane™ Persistent Memory 200 Series Brief. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/achieve-greater-insight-with-intel-optane-persistent-memory-brief.pdf, 2021.

[3] HPC IO Benchmark Repository. https://github.com/hpc/ior, 2022.

[4] NVIDIA Mellanox ConnectX-6 SmartNIC Adapter. https://www.nvidia.com/en-us/networking/ethernet/connectx-6/, 2022.

[5] RocksDB. http://rocksdb.org/, 2022.

[6] syscall_intercept. https://github.com/pmem/syscall_intercept, 2022.

[7] 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.

[8] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[9] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Gregory R Ganger, Marshall Kirk McKusick, Craig AN Soules, and Yale N Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[13] 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.

[14] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-Journal: Scalable Per-Core Journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 893–906, 2021.

[15] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[16] 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.

[17] 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.

[18] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write Dependency Disentanglement with HORAE. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[19] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[20] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 89–103, USA, 2017. USENIX Association.

[21] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, 2017.

[22] 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.

[23] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 71–85, 2022.

[24] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, 2013.

[25] 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.

[26] 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.

[27] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[28] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111, 2021.

[29] Yiduo Wang, Yufei Wu, Cheng Li, Pengfei Zheng, Biao Cao, Yan Sun, Fei Zhou, Yinlong Xu, Yao Wang, and Guangjun 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, page 331–346, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Ying Wang, Dejun Jiang, and Jin Xiong. Numa-aware thread migration for high performance nvmm file systems. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*, 2020.

[31] 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.

[32] Jian Xu and Steven Swanson. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, page 323–338, USA, 2016. USENIX Association.

[33] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. End-to-End I/O Monitoring on Leading Supercomputers. *ACM Trans. Storage*, nov 2022. Just Accepted.

[34] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 17–34, Santa Clara, CA, February 2022. USENIX Association.

[35] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.