Accelerating Distributed Filesystem Metadata Service via Decoupling Directory Semantics from Metadata Indexing

Wenhao Lv
Tsinghua University
Haidian Qu, Beijing Shi, China
lvwh19@mails.tsinghua.edu.cn

Hao Guo
Tsinghua University
Haidian Qu, Beijing Shi, China
gh23@mails.tsinghua.edu.cn

Qing Wang
Tsinghua University
Haidian Qu, Beijing Shi, China
wq1997@tsinghua.edu.cn

Youyou Lu
Tsinghua University
Haidian Qu, Beijing Shi, China
luyouyou@tsinghua.edu.cn

Jiwu Shu*
Tsinghua University
Haidian Qu, Beijing Shi, China
shujw@tsinghua.edu.cn

Abstract

Existing distributed filesystem metadata services fundamentally rely on ordered metadata indexing, significantly limiting their performance under high-speed networks and persistent memory (PM). In this paper, we propose HMFS, an efficient distributed metadata service that decouples directory listing semantics from the underlying metadata indexing, thus eliminating the need for metadata indexing to organize metadata in an ordered but high-overhead manner. HMFS incorporates three key techniques. First, it proposes *index-decoupled metadata organization*, which indexes metadata with unordered indexes and preserves directory semantics with direct inter-object links. Second, it designs *lightweight crash-consistent metadata updates* to efficiently update these two independent data structures coherently with crash consistency. Third, it employs *parallel directory listing* to accelerate directory traversal. Experiments show that, compared to CephFS and InfiniFS (both deployed on PM) and SingularFS (a PM-optimized metadata service), HMFS reduces metadata operation latency by 92%, 62%, and 39%, and improves throughput by 20.5×, 7.5×, and 1.8×, respectively. At a scale of tens of billions of files, HMFS maintains scalable performance, sustaining 13.4 Mops/s for file `create` operations and 34.7 Mops/s for file `stat` operations.

CCS Concepts

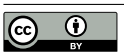
• **Software and its engineering → File systems management**.

Keywords

Distributed file systems, Metadata management, Persistent memory

ACM Reference Format:
Wenhao Lv, Hao Guo, Qing Wang, Youyou Lu, and Jiwu Shu. 2025. Accelerating Distributed Filesystem Metadata Service via Decoupling Directory Semantics from Metadata Indexing. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772237

*Corresponding author.



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772237

1 Introduction

Distributed filesystems are fundamental to modern cloud infrastructure [40, 59]. Metadata management and file data storage in modern distributed filesystems are typically decoupled to enable independent scaling of the two components [22, 47, 56, 60]. An efficient metadata service is particularly critical, as modern clouds and datacenters often manage tens of billions of files, placing significant demands on metadata performance [40, 48, 59, 66]. To address this challenge, many distributed filesystems leverage ordered key-value (KV) stores, such as RocksDB [19], to manage metadata. This design pattern is widely adopted in state-of-the-art distributed filesystems, including both industrial systems [2, 4, 25, 29, 48, 59] and academic research [23, 35, 40, 51, 52, 64, 67, 68].

Recent advances in networking and storage hardware shift the performance bottlenecks of distributed filesystem metadata services from hardware limitations to software inefficiencies. Modern RDMA networks achieve latencies of a few microseconds and bandwidths exceeding 100 Gbps [3, 46], and emerging persistent memory (PM) technologies [12, 15, 53] deliver byte-addressability and near-DRAM storage performance. However, our evaluation reveals that existing distributed filesystem metadata services fail to effectively exploit these advanced hardware capabilities (§2.3). Specifically, under high-speed networks and PM storage, state-of-the-art designs, such as RocksDB-based InfiniFS [40] and PM-optimized ordered-index-based SingularFS [23], exhibit substantial metadata storage overhead, accounting for over 76% and 53% of the total metadata operation latency, respectively. Due to inherent design mismatches between RocksDB and PM, InfiniFS achieves only 18% of the throughput of SingularFS. Meanwhile, SingularFS experiences a notable throughput degradation by 36% as the metadata volume increases from 64 K to 100 million files. These results underscore the significant performance limitations of state-of-the-art approaches when deployed on advanced hardware.

After an in-depth analysis, we identified the root cause of these performance limitations: the inherent reliance on ordered metadata indexing in existing distributed filesystems. This reliance stems from the tight coupling between directory listing semantics and metadata indexing. Specifically, existing designs rely on the range query capabilities of underlying ordered indexes to support directory listing operations. This forces all metadata operations to incur the overhead of maintaining a total order among metadata objects.

Therefore, metadata operations suffer from increased latency and cannot scale effectively with growing metadata volume.

To address this problem, our key idea is to ***decouple directory semantics from the underlying metadata indexing***, where range-based directory listings and all other metadata operations (referred to as point-based operations) are served by independent and specialized data structures. With this design, point-based operations, such as file `create` and `stat`, can be efficiently handled using unordered indexes (e.g., hash tables), while directory listing semantics are preserved through explicit inter-object links. By doing so, point-based operations can achieve constant-time metadata access, regardless of the total volume of metadata. This is beneficial for real-world workloads (e.g., Alibaba Cloud [40], Baidu AI Cloud [59], and Spotify [45]), where point-based operations dominate, accounting for 95.3% of all metadata requests (see §2.1 and Table 1).

Building upon this core idea, we design and implement an efficient distributed filesystem metadata service named HMFS, which is optimized for modern high-speed networks and byte-addressable PM storage. HMFS achieves superior metadata performance by incorporating the following key design:

1) Index-decoupled metadata organization. HMFS indexes metadata objects and serves directory listings with two independent data structures. Specifically, the filesystem hierarchical directory tree is split across metadata servers in a per-directory scheme. Each server indexes metadata objects using a PM-optimized hash table, and preserves directory semantics via per-directory linked lists embedded within the metadata objects. By doing so, point-based metadata operations can be served with $O(1)$ PM read/writes. Moreover, we adopt a unified, cacheline-aligned metadata memory layout, which simplifies allocation and enhances cache efficiency.

2) Lightweight crash-consistent metadata updates. Based on our insight into the timestamp ordering and careful co-location of related metadata fields, common-case metadata operations can efficiently update those two independent data structures coherently without logging, achieving low overhead. Specifically, all single-point operations update metadata using a single atomic write (8-byte or 16-byte); all double-point operations update metadata with minimal atomic writes in a specific order. Crashes may cause inconsistent metadata, but HMFS makes it tolerable: by leveraging inter-metadata relationships, we can correctly detect and recover inconsistent objects in an on-demand manner. In this way, HMFS can quickly provide metadata service upon crash.

3) Parallel directory listing. HMFS pipelines and parallelizes the directory listing process to reduce the high latency of traversing linked lists of large directories. Specifically, HMFS pipelines network requests with PM reads to hide latency, and simultaneously parallelizes directory traversal by dividing the linked list into segments. We further compress directory entries to minimize network transmission overhead, based on our observation of redundancy within directory entry fields.

We evaluate HMFS with Mellanox ConnectX-5 RNIC and Intel Optane PM. Experimental results demonstrate that HMFS significantly outperforms state-of-the-art solutions. Specifically, compared to CephFS and InfiniFS (both deployed on PM) and SingularFS (a PM-optimized distributed metadata service), HMFS reduces metadata operation latency by up to 92%, 62%, and 39%, and improves metadata operation throughput by up to 20.5×, 7.5×, and 1.8×, respectively. Directory listing operations are also highly efficient in HMFS; listing a large directory (containing one million files) completes in 42 ms, representing only 1%, 8%, and 15% of the latency of CephFS, InfiniFS, and SingularFS, respectively. Moreover, HMFS demonstrates excellent scalability: at a scale of tens of billions of files, it sustains stable throughput of 13.4 Mops/s for file `create` operations and 34.7 Mops/s for file `stat` operations.

Metadata operations	Baidu	Alibaba	Spotify	Avg.
Single/Double-point	97.8%	86.8%	89.7%	91.4%
Multi-point	1.2%	9.3%	1.3%	3.9%
Range	1.0%	3.9%	9.0%	4.7%

Table 1: Metadata operation ratios in real-world workloads.

2 Background and Motivation

2.1 Metadata Operations

Metadata operations in distributed filesystems can be broadly classified into following categories based on their access patterns:

1. *Point-based metadata operations.* These operations use point queries to access target metadata, which can be further categorized by the number of inodes accessed:
   - *Single-point operations* access only the target inode, such as `stat`, `access`, `chown`, `chmod`, `read`, `write`, and `truncate`.
   - *Double-point operations* access both the target inode and its parent directory's inode, including `create`, `delete`[1], `mkdir`, `rmdir`, and `symlink` operations.
   - *Multi-point operations* include `rename`, `link`, and `unlink` operations. These operations access three inodes: the target inode, the inode of the source parent directory, and the inode of the target parent directory.
2. *Range-based metadata operations.* These operations rely on range queries to access the target metadata, which include only directory listing (i.e., `readdir`) operations.

To understand the relative frequency of metadata operations in real-world cloud workloads, we analyze the published workload characteristics of three representative production systems, including CFS in Baidu AI Cloud [59], Pangu in Alibaba Cloud [40], and HDFS in Spotify [45]. From the results shown in Table 1, we can observe that: ***Point-based metadata operations are far more frequent than range-based operations.***

2.2 Metadata Management

State-of-the-art distributed filesystems commonly use ordered KV stores to manage filesystem metadata [2, 4, 23, 29, 35, 40, 48, 51, 52, 59, 67]. These systems typically store metadata objects in an ordered tree-based indexing structure, which aligns naturally with the directory semantics of filesystems. In particular, they rely on the ordering guarantees of the index to implement efficient directory listing operations via range queries.

Specifically, these systems typically manage filesystem metadata as illustrated in Figure 1(a). The logical hierarchical directory tree of filesystems is decomposed into a collection of file and directory

[1]In this paper, we use the term `delete` to denote `unlink` operations on files with no extra hard links (i.e., link count equals one), for clarity of presentation.

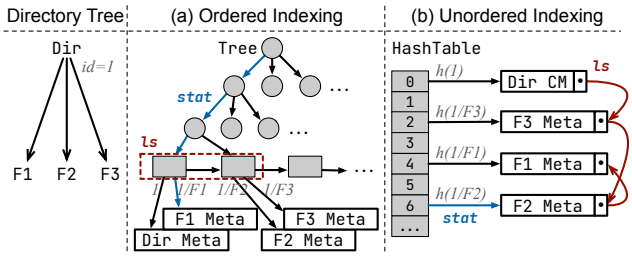


Figure 1: (a) State-of-the-art distributed filesystems maintain directory listing semantics implicitly via the range query capabilities of ordered KV stores. (b) We decouple directory semantics from metadata indexing, enabling the use of unordered indexes for metadata management, while preserving directory structure via explicit inter-object links.

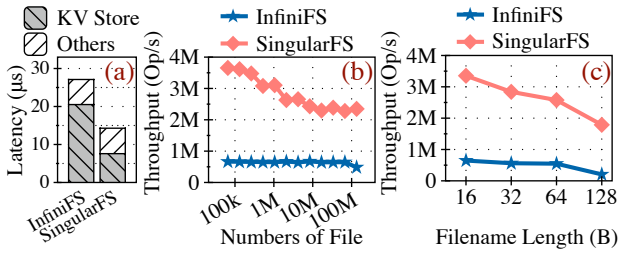


Figure 2: (a) Latency breakdown of `create`. (b) Throughput of `create` under increasing metadata volume. (c) Throughput of `create` under increasing filename length.

metadata objects, each stored as a key-value pair in an ordered tree-based index. The key is typically encoded with the parent directory's inode number and the name string, while the value holds the metadata of the corresponding file or directory. Point-based metadata operations—such as `stat`—are implemented with point queries which traverse multiple levels of internal tree nodes to locate the target metadata object. The search key is formed by concatenating the inode number of the parent directory with the filename string. As for range-based operations—primarily directory listings—distributed filesystems typically adopt a different design from local filesystems. In local filesystems such as Ext4 [43], directory entries (*dirent*) are explicitly stored within the directory's data blocks, directly supporting listings through block reads. However, this approach is unsuitable for distributed scenarios, as metadata operations frequently require updating these extra dentry blocks, incurring substantial cross-server coordination overhead. Therefore, modern distributed filesystems do not materialize directory entries in data blocks, but instead leverage the range query capabilities of ordered indexes to implement directory listing operations. Specifically, all metadata objects keyed with a common prefix—the inode number of the target directory—are retrieved via a range query, reconstructing the directory entries at runtime.

2.3 Revisiting Metadata Management with Fast Networks and Storage

Recent advances in networking and storage hardware have shifted the performance bottlenecks in distributed filesystem metadata services from hardware limitations to software inefficiencies. In particular, high-speed RDMA networks [3, 46] and byte-addressable persistent memory [12, 15, 53] have drastically reduced communication and storage latency. RDMA networks deliver round-trip latencies of a few microseconds and bandwidths exceeding 100 Gbps, enabling low-latency, high-throughput metadata access. At the same time, persistent memory, as a non-volatile storage medium connected to the memory bus, offers persistence, large capacity, byte-addressability, and high performance.

We find that ordered metadata indexing becomes the new performance bottleneck under fast network and storage hardware. To understand this, we evaluate two representative systems: InfiniFS [40] using Pmem-RocksDB [13], representing the majority of distributed metadata services built upon RocksDB, and SingularFS [23] using P-Masstree [34], a representative PM-optimized ordered key-value store. The experiments are conducted on a cluster equipped with Intel Optane DC Persistent Memory and Mellanox ConnectX-5 NICs (see §5.1 for details).

First, we evaluate the latency of file `create` with 128 million files pre-loaded into the filesystem, and break it down into two parts: the ordered KV store in the metadata server, and all other overheads such as request parsing, RPC, and concurrency control. As shown in Figure 2(a), the ordered KV store accounts for 76% and 53% of the total latency in InfiniFS and SingularFS, respectively. These results highlight that *the overhead of ordered key-value stores is significant* for distributed metadata services.

Then, we evaluate the throughput of `create` operation under increasing number of files in the filesystem. As shown in Figure 2(b), InfiniFS performs poorly across all scenarios, highlighting that the underlying ordered LSM-tree used by RocksDB is suboptimal for PM. SingularFS outperforms InfiniFS by leveraging P-Masstree [34], a PM-optimized tree-based index. Compared to traditional B^+-trees, P-Masstree achieves superior performance [34] through its trie-of-trees design, where B^+-trees are organized by key slices, improving cache locality and reducing pointer-chasing overhead.

However, despite these improvements, its ordered metadata organization based on the tree-structured index still incurs high overhead. First, SingularFS's B^+-tree characteristic causes 36% throughput drop as metadata volume increases from 64K to 100 million files (Figure 2(b)). Second, SingularFS's trie-tree characteristic makes it highly sensitive to the naming pattern and length of filenames. As multiple filenames sharing a same prefix string is a common naming pattern in real-world scenarios [1, 37, 49], we evaluate the sensitivity using this naming pattern. Specifically, we measure the throughput of file `create` with filename lengths ranging from 16 to 128 bytes, where every 16 files share a common prefix. As shown in Figure 2(c), SingularFS's throughput drops significantly by 47% as the length of filename increases from 16 to 128 bytes.

Unfortunately, existing distributed filesystem metadata services heavily rely on ordered KV stores. *This reliance stems from a fundamental coupling between directory semantics and metadata indexing.* Specifically, directory listing semantics are supported via the range query capability of ordered indexes, which forces all metadata operations—point or range—to incur the overhead of maintaining a total ordering, even though point-based metadata operations are far more frequent in real-world workloads (Table 1).

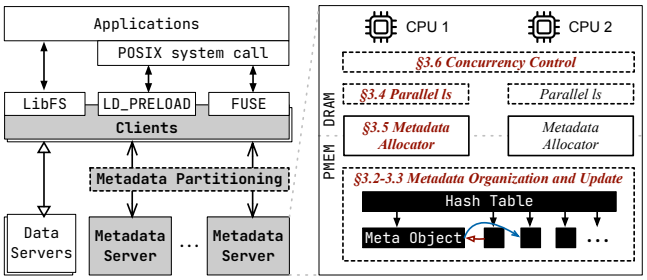


Figure 3: HMFS architecture.

Key idea. The analysis above reveals that existing distributed filesystem metadata services are inherently inefficient on modern hardware due to their reliance on ordered metadata indexing. To address this problem, **we propose to decouple directory semantics from metadata indexing**, where metadata lookup and directory listing are served using independent data structures. By doing so, point-based metadata operations, which dominate real-world workloads, can be efficiently handled using unordered indexes (e.g., hash table in Figure 1(b)). This not only improves performance for common-case operations, but also ensures constant-time access regardless of the metadata volume. Meanwhile, by leveraging the byte-addressability of PM, we can preserve directory listing semantics through explicit inter-object links among metadata objects within the same directory, without relying on ordered indexes.

3 Design and Implementation

3.1 Overview

Following the key idea of decoupling directory semantics from metadata indexing, we propose HMFS, an efficient distributed filesystem metadata service designed for modern high-performance hardware. Figure 3 presents an overview of the HMFS architecture.

Clients. HMFS clients provide POSIX-compliant interfaces to applications through three methods: a user-space library (LibFS), system call hooking via a LD_PRELOAD library, and a FUSE-based filesystem interface. Applications can statically link against LibFS at compile time to access HMFS directly. Alternatively, system call hooking allows unmodified applications to use HMFS by setting the LD_PRELOAD environment variable, which intercepts libc POSIX calls and redirects them to HMFS clients [16, 44]. Also, HMFS offers a FUSE interface, enabling applications to mount it as a local filesystem. Both LibFS and system call hooking enable applications to bypass the kernel, achieving optimal performance, while FUSE is often criticized for the significant overhead it incurs [5, 8]. Similar to previous works [35, 40, 52, 54], HMFS clients cache the directory's name, inode number, and permission, using this cached information exclusively to accelerate path resolution.

Metadata partitioning. HMFS partitions the hierarchical directory tree across metadata servers in a per-directory scheme, as shown in Figure 2. Each directory's metadata is split into two objects: a directory content metadata object, which contains fields related to the child; and a directory access metadata object, which contains all other fields (Figure 4). Then, file and directory metadata objects are distributed across servers using consistent hashing on their partition key. By doing so, for each directory, the content metadata of the directory itself, the access metadata of its child directories, and the metadata of its child files are colocated on the same metadata server, achieving both scalability and inner-directory locality.

Metadata Object	Partition Key	Hash Key
Directory access metadata	*pID*	*pID, name*
Directory content metadata	*ID*	*ID*
File metadata	*pID*	*pID, name*

Table 2: Per-directory partitioning scheme. *ID* denotes the inode number; *pID* refers to the *id* of the parent directory.

Metadata server. Within each metadata server, metadata objects are stored as key-value pairs, where keys are defined in Figure 2, and values are the corresponding metadata objects. Following the key idea of decoupling directory semantics from metadata indexing, HMFS indexes metadata objects using a hash table and maintains directory structure through per-directory linked lists. However, this design may incur high overhead due to the necessity of coherently updating two separate data structures. To address this issue, HMFS introduces a lightweight metadata update protocol (§3.3) that efficiently updates metadata in-place without logging. Also, maintaining directory structure using linked lists may result in slow directory listing operations. To alleviate this, HMFS reduces the latency of directory traversal through pipelining and parallelization (§3.4). Additionally, HMFS includes a metadata-oriented per-CPU PM allocator that leverages the limited variability of metadata objects to achieve fast allocation and instant post-crash availability (§3.5). Furthermore, HMFS provides fine-grained concurrency via CAS primitives and an in-DRAM lock table (§3.6), supporting concurrent creation and deletion operations under a shared directory. Rename and hard links are also efficiently supported (§3.7).

Data management. HMFS separates file data storage from metadata management, delegating the former to existing object storage like RADOS [62]. Specifically, HMFS splits files into blocks, which are stored as objects keyed by `<inode_number, block_number>`, and placed across data servers via the CRUSH algorithm [61]. For crash consistency, HMFS guarantees only metadata consistency [7]. This model is POSIX-compliant [31] and adopted by filesystems like Ext4 and CephFS [60]. Metadata is atomically updated after data is written to the object store. The filesystem will recover to a consistent state after a crash with respect to its metadata.

3.2 Index-Decoupled Metadata Organization

In this section, we first introduce how HMFS organizes metadata in two separate data structures. Then, we show the metadata layouts of different kinds of metadata objects.

Hash-based metadata indexing for constant-time metadata access. Within each metadata server, HMFS indexes metadata objects using a PM-based hash table rather than ordered indexes. This significantly improves the performance of all point-based metadata operations, which are much more frequent than range-based operations in real-world workloads, as described in §2.1. For each entry in the hash table, the key is a pointer to either the metadata object's identifier or a combination of its parent identifier and its name, and the value is a pointer to the metadata object itself (see Figure 4). In our current implementation, we use DASH [39] as the backend index, which is a PM-optimized concurrent bucket-based hash table.

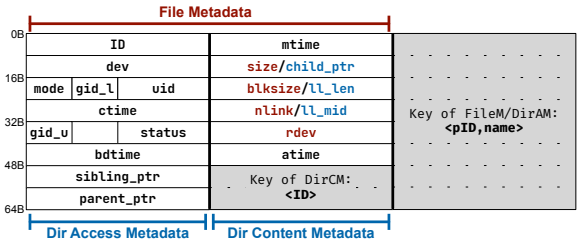


Figure 4: Metadata layout in HMFS.

Insertion and deletion operations in the hash table are performed atomically in a single atomic write.

Link-based directory management to support directory listing. As shown in Figure 1(b), HMFS decouples directory listing semantics from metadata indexing by maintaining directory structures through direct links, thereby enabling the use of efficient unordered indexing for metadata objects. Specifically, the directory content metadata maintains a pointer (*child_ptr*) to the linked list of its child metadata objects. The file metadata or directory access metadata contains a pointer (*sibling_ptr*) to its next sibling. Note that, based on the partitioning scheme, all metadata required for listing a directory is co-located on the same server. Therefore, HMFS can maintain directory structures simply with a local linked list, and serve directory listing operations by list traversing.

Locality-optimized cache-friendly metadata layout. HMFS adopts a unified, cacheline-aligned memory layout for both file and directory metadata (Figure 4), to enhance cache efficiency and simplify memory allocation. Directory metadata mirrors file metadata structure but replaces file-specific metadata fields with directory-specific ones. To improve locality, HMFS stores each hash table key (containing filename) adjacently with its corresponding metadata object in memory. Metadata fields are also carefully organized to exploit common access locality patterns. For example, the *mtime* and *child_ptr* fields of the directory metadata are placed adjacently, as they are updated together during frequent file `create` and `delete` operations. Such locality-optimized metadata layout not only improves runtime performance but also facilitates efficient crash-consistent metadata updates, as detailed in Section 3.3.

3.3 Lightweight Crash-Consistent Updates

In this section, we introduce how HMFS supports metadata updates. We first introduce our key observations: *timestamp ordering* and *atomic 16-byte write*. Then, based on these, we show how to achieve crash-consistent in-place updates without journaling.

O1: Timestamp ordering. Most metadata operations update timestamps upon completion, specifically the modification time (`mtime`) indicating the time of last data modification, and the change time (`ctime`) indicating the time the file status last changed, as summarized in Table 3. Based on the POSIX standard [26] and our analysis, we identify a fundamental relationship among these timestamps: `ctime` *is monotonically increasing and not less than the corresponding* `mtime`. Consequently, whenever an operation simultaneously modifies `mtime` and `ctime`—such as file writes, creations, or deletions—it suffices to update only the `mtime` explicitly. Any subsequent access can dynamically infer the correct `ctime` as *max*(`ctime`, `mtime`).

Metadata Operations	Target Metadata		Parent Metadata	
	mtime	*ctime*	*mtime*	*ctime*
`create,mkdir`	•	•	•	•
`delete,rmdir`			•	•
`link,unlink`		•	•	•
`rename`		•	•	•
`write,truncate`	•	•		
`chown,chmod`		•		

Table 3: Impact of metadata operations on the `mtime` and `ctime` of the target file and its parent directory.

O2: Atomic 16-byte write. Modern CPUs support atomic 16-byte aligned memory operations. For instance, Intel specifications [28] guarantee atomicity for 16-byte aligned store using `movdqa` and compare-and-swap using `cmpxchg16b`. As the persistence granularity for PM is per cacheline, these operations simultaneously provide atomic update and persistence for aligned 16-byte data.

3.3.1 *Atomic Single-Write Updates.*

For all single-point metadata operations, HMFS updates related metadata atomically with a single 8-byte or 16-byte write.

`Write` and `truncate`. `Write` can be categorized into two types: *overwrite* and *append*. An *overwrite* operation requires updating the file's `mtime` and `ctime` fields, while an *append* additionally updates the file's `size`. For overwrite, updating `mtime` using an 8-byte atomic write suffices (O1). For append, we seek to update `mtime` and `size` using a 16-byte atomic write (O2). Therefore, we place these two fields within an aligned 16-byte persistent memory block. `Truncate` updates `mtime`, `ctime`, and `size`, similar to *append*. Thus, we use a 16-byte atomic write for it.

`Read` and `readdir`. These operations only update the `atime` field, which is conducted using a single 8-byte atomic write.

`Chown` and `chmod`. The `chown` and `chmod` operations update the ownership metadata (`uid`, `gid`) and the permission metadata (`mode`), respectively, along with the status change time (`ctime`). These fields account for 18 bytes in total (4 bytes `uid` and `gid`, 2 bytes `mode`, and 8 bytes `ctime`), larger than the 16-byte write in O2.

To support updating them using a single 16-byte write, we split the `gid` into separate upper and lower 2-byte segments and place its lower segment with other required fields into an aligned 16-byte PM region. Thus, `chown` and `chmod` can update these fields using a single 16-byte write, as long as `gid` is less than 2^{16} (i.e., 65536).

When `gid` is greater than 65536, we fall back to the standard journaling mechanisms. However, given that the `gid` is automatically assigned to new groups in numerical order starting at 1000 [24], this fallback should be rare in typical usage scenarios.

3.3.2 *Recoverable Log-Free Updates.*

In general, double-point metadata operations involve writes to both the target metadata and its parent directory's metadata. To ensure crash consistency, HMFS updates corresponding metadata fields in a specific order, eliminating journaling overhead.

`Create/mkdir`. For file creation (Figure 5), HMFS ❶ Inserts the new file metadata object atomically into the metadata hash table with a single 8-byte atomic write. The inserted file metadata contains a status initialized as `StatusCreating`, a birth timestamp (`bdtime`), and a parent pointer referencing its parent directory content metadata. ❷ (Commit Point) Updates the parent directory's child pointer (head of the linked list for its children) and `mtime`, using a single

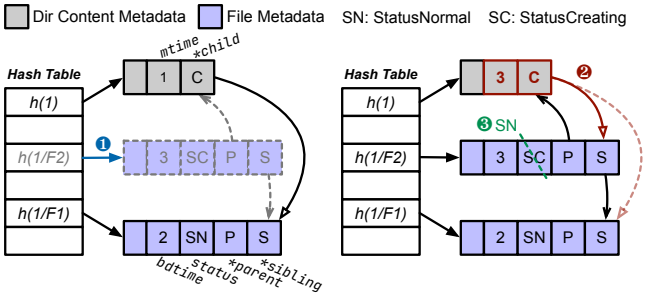


Figure 5: Workflow of creating file F2 under directory (ID=1).

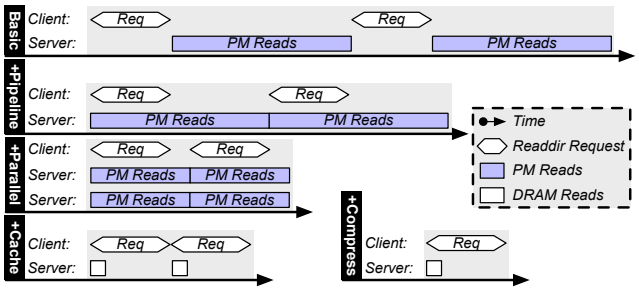


Figure 6: Directory listing workflow in HMFS with incremental optimizations. The initial `opendir` request is omitted.

16-byte write (O2). Note that `ctime` is not required to be updated (O1). ❸ Updates the status of the new metadata to `StatusNormal`.

For directory creation, besides the steps above for directory access metadata, we insert the directory content metadata into the hash table after the commit point (i.e., between steps ❷ and ❸). Note that this step does not require distributed transactions, even when the content metadata resides on a different server, as the content metadata can be correctly generated from access metadata.

`Delete/rmdir`. For file deletion, HMFS ❶ Atomically marks the status of the target file metadata as `StatusDeleting` and updates its birth-deletion timestamp (`bdtime`) using a single 16-byte write (O2). ❷ (Commit Point) Updates the `mtime` of the parent directory's content metadata with an 8-byte atomic write (O1). ❸ Updates the status of the target metadata to *StatusDeleted*. The deleted metadata node is physically removed later during directory listing or removing operation, which will update its sibling's `sibling_ptr`.

For directory deletion, we first check its emptiness by accessing its children list head. Besides the steps above, we also need to delete the content metadata from the hash table right after the commit point (i.e., between steps ❷ and ❸).

On-demand crash recovery. A system crash during these double-point operations will only leave the target metadata in a tolerable and detectable inconsistent state. The recovery is triggered on-demand by a subsequent operation targeting the inconsistent metadata, such as a client-side retry of the failed `create` request, eliminating expensive startup-time consistency scans (e.g., `fsck`).

HMFS can detect and recover the inconsistent metadata object by checking its status and whether it is enlisted in the children list of the parent directory. Specifically, when a metadata object's status is `StatusCreating`, we check if it is in the parent directory's children list: if so, we redo by updating its status to `StatusNormal` and re-creating the content metadata (for directory); if not, we undo by deleting it from the hash table. When the status is `StatusDeleting`, we check if its `bdtime` is greater than its parent's `mtime`: if so, we undo by updating the status to `StatusNormal`; if not, we redo by deleting the content metadata from the metadata hash table (if it is a directory), and updating its status to `StatusDeleted`.

3.4 Parallel Directory Listing

Listing a huge directory requires traversing a long linked list, which triggers pointer chasing, resulting in random PM accesses and high latency. To overcome this, HMFS comes with a suite of optimizations that target different aspects of the workflow, as illustrated in Figure 6: pipelining to hide PM access latency, parallel prefetching to accelerate large directory scans, caching to serve repeated requests from DRAM, and compression to reduce network overhead.

Basic. Directory listings return directory entries containing filenames, file types, and inode numbers. Clients use the POSIX interface: they first open a directory stream with `opendir`, sequentially iterate through entries using `readdir` (which internally retrieves entries from metadata servers in batches, as shown in Figure 6), and finally close the directory stream using `closedir`. In HMFS, directory listing involves dereferencing sibling pointers, causing random PM reads and increased latency.

+Pipeline. Given that reading directory entries from PM involves higher latency compared to network requests, HMFS overlaps these procedures to partially hide PM read latency. Specifically, after `opendir`, a worker thread keeps prefetching directory entries from scattered PM locations into a contiguous DRAM buffer for subsequent `readdir` requests, reducing the overall latency.

+Parallel. For large directories, HMFS partitions the child linked list into multiple segments that can be prefetched concurrently. In our current implementation, we use two segments, which is sufficient to overlap PM read latency with network request latency (see §5.5). To enable this, HMFS maintains both the length (*ll_len*) and the midpoint pointer (*ll_mid*) of the child linked list for directories, as illustrated in Figure 4. Crash consistency is unnecessary for *ll_len* and *ll_mid*, as they are merely performance hints for workload distribution and do not affect the correctness of directory listing.

+Cache. HMFS metadata server caches recently prefetched entries in DRAM to accelerate listing operations on the same directory. Subsequent listings can thus be served directly from DRAM, conserving PM bandwidth for other metadata operations. Upon receiving a directory listing request, HMFS first checks the cache and serves entries directly from DRAM if available; otherwise, it fetches the entries from PM and updates the cache. Cache replacement employs a least recently used (LRU) policy. Creation and deletion requests will update the cache upon completion.

+Compress. Network latency is the bottleneck when serving cached entries. Directory entries often exhibit redundancy in filenames, file types, and inode numbers [1, 37, 49]. Therefore, HMFS compresses directory entries before sending them to clients, to reduce the amount of data transmitted over the network and thereby improve performance. In our current implementation, we utilize LZ4 [9], an extremely fast compression algorithm, to compress entries.

POSIX Compliance. Directory listing with these optimizations, as well as all other metadata operations in HMFS, remains fully POSIX-compliant. As explicitly stated in the POSIX specification [27, 41], directory listing operations are not required to guarantee entry ordering or return an atomic snapshot of the directory.

3.5 Metadata-Oriented PM Allocation

HMFS introduces a PM allocator optimized for filesystem metadata, supporting fast allocation and instant post-crash availability.

Key observation. Filesystem metadata sizes exhibit inherently *limited variability*: inodes consist exclusively of fixed-size fields, and filename lengths are limited to 255 bytes by the POSIX standard.

Therefore, unlike traditional allocators supporting arbitrary size allocations, we only support allocating three fixed sizes: one cache line for directory content metadata, three cache lines for file metadata with filenames up to 64 bytes and directory access metadata with filenames up to 128 bytes, and six cache lines for all others.

Per-CPU metadata allocation. HMFS uses per-CPU allocator for less contention. Each CPU first allocates large memory segments using PMDK [14] (to ensure atomicity) and then allocates the three fixed sizes inside the segments. Each memory segment contains a PM-resident bitmap to record allocation information. HMFS also uses DRAM-resident free lists to locate the free objects efficiently.

Instant post-crash availability. Typically, after a crash, a synchronous recovery is required to rebuild the allocator's DRAM state by scanning the filesystem [18, 23, 30, 65, 69]. This significantly delays filesystem availability. In contrast, HMFS achieves instant allocator availability: After a crash, each CPU immediately allocates fresh memory segments from the global PM pool, allowing immediate service of both allocation and deallocation requests. Concurrently, HMFS asynchronously scans the bitmap sections of allocated segments to collect free spaces and rebuild free lists.

3.6 Fine-Grained Concurrency Control

HMFS manages metadata reader-writer locks in a DRAM-resident lock table. Specifically, to acquire the lock of the target metadata object, we first determine its corresponding entry in the lock table by hashing its index key (Table 2), and then lock this entry. To avoid potential deadlocks when multiple metadata objects need to be locked simultaneously, HMFS acquires locks in an ascending order of the server ID and the position in the lock table.

Single-point metadata operations update metadata with an atomic write, thus requiring only a reader lock of the target metadata to avoid concurrent `delete`. Double-point metadata operations require a writer lock of the target metadata, but only require a reader lock of the parent directory content metadata. HMFS uses the 16-byte CAS [58] to atomically update the parent directory's `child_ptr` and `mtime`. By doing so, HMFS supports concurrent file `create` and `delete` within a shared directory. Range metadata operations also require only a reader lock of the target directory content metadata, based on the POSIX standard [27].

Concurrent 16-byte atomic metadata updates and metadata reads might lead to inconsistency, as the metadata read itself is non-atomic. We use an existing optimistic concurrency control (OCC) protocol for this [23]. Specifically, HMFS validates the consistency of file `stat` by comparing timestamps before and after retrieving the full inode. If timestamps match, the retrieved inode is consistent; otherwise, the operation is retried.

3.7 Hard Link and Rename

HMFS supports hard link and rename operations via indirect metadata objects and this invariant: *a metadata object with multiple hard links (nlink > 1) will be neither freed nor relocated in memory.*

Specifically, when creating a hard link (i.e., a new filename associated with an existing metadata object), HMFS inserts an indirect metadata object into the hash table and the parent directory's linked list. This indirect metadata object contains a reference (server ID and memory address) to the original metadata object. Subsequent accesses to this indirect metadata object will be transparently redirected to the original metadata object. When deleting a hard link, HMFS only removes the indirect metadata object. The original metadata object remains at the same location in PM.

Rename operations in HMFS also adhere to this invariant. When renaming a file across directories, HMFS first removes the old KV pair from the hash table, flags the metadata object to retain it in PM. It then creates an indirect metadata object referencing the original one, and inserts it into the hash table and the target directory's linked list. Renaming files within the same directory is much simpler, as it only involves replacing the old KV pair in the hash table with the new one, without creating indirect objects. This efficiency is particularly beneficial as real-world workloads [59] show that nearly 99% of rename operations occur within the same directory.

Rename and link operations may involve updating multiple metadata objects within a single server or across two servers. Since these metadata operations are infrequent in real-world workloads [40], we adopt standard journaling and two-phase commit protocols to guarantee crash consistency. Additionally, concurrent directory rename operations may introduce orphan loops; this can be resolved using the existing method from InfiniFS [40].

4 Discussion

PM hardware used by HMFS. The design of HMFS leverages the byte-addressability of persistent memory in general, making it compatible with a range of hardware technologies. This includes not only Intel Optane PM, which was used in our evaluation, but also emerging CXL-based solutions like the Samsung Memory Semantic SSD [53]. The memory-semantic SSD can effectively serve as persistent memory by leveraging CXL interconnect technology along with built-in DRAM caches. Furthermore, UPS-backed DRAM serves as a readily available alternative in cloud data centers [17], when persistent memory is unavailable. Although this option may offer smaller capacity, HMFS effectively mitigates this limitation through its space-efficient metadata management (requiring only ~300 GB to manage 1 billion files) and scale-out architecture (see §5.4). Additionally, we can tier cold metadata to SSDs to further enhance cost-effectiveness.

Large shared directory. This paper primarily focuses on the metadata storage engine rather than the directory partitioning scheme. Therefore, for partitioning, we adopted a straightforward per-directory granularity, similar to prior work like InfiniFS [40]. While generally effective, this approach is often argued to perform poorly under shared directory workloads, where numerous clients

concurrently create files within a single directory. Nevertheless, HMFS achieves superior performance in such scenarios (see evaluation in §5.6), owing to its efficient metadata storage engine. This is accomplished primarily through the following two optimizations. First, our engine requires only a shared lock on the directory metadata (§3.6), enabling concurrent file creations in the same directory. Meanwhile, the critical section is minimized to a single CAS instruction (§3.3.2), drastically reducing contention. Second, contention can be further reduced by batching pending requests, merging multiple directory modifications into a single update. Finally, HMFS can readily integrate existing directory partitioning techniques, such as GIGA+ [50] or CephFS's directory fragmentation [6], to dynamically distribute a large directory across multiple servers.

5 Evaluation

5.1 Experimental Setup

Experiments are conducted on a cluster of 6 machines, all equipped with identical hardware. Each machine has two NUMA nodes, each hosting an Intel Xeon Gold 6240M CPU with 18 cores (36 hyperthreads). Each NUMA node is equipped with three 256 GB Intel Optane DIMMs (totalling 768 GB), three 32 GB DDR4 DRAM modules (totalling 96 GB), and one 100 Gbps Mellanox ConnectX-5 RNIC. All nodes are interconnected via a 100 Gbps Mellanox InfiniBand switch. The operating system is Ubuntu 20.04 with Linux kernel version 5.4.0. The PMDK version is 1.7.

Comparison systems. To ensure a comprehensive evaluation, we compare HMFS with both local PM filesystems and distributed filesystems. We select four representative systems: Ext4-DAX represents local PM filesystems. CephFS [60] represents traditional distributed filesystems. InfiniFS [40] represents distributed filesystem metadata services built upon KV stores. SingularFS [23] represents PM-optimized filesystem metadata services. HMFS, InfiniFS, and SingularFS are all implemented using the same RPC framework, eRPC [33]. InfiniFS's storage backend is replaced from RocksDB [19] to Pmem-RocksDB [13], which is optimized for PM by Intel. CephFS uses PM as raw block devices.

Unless otherwise specified, all distributed filesystems are evaluated using a single metadata server deployed on one NUMA node, which launches 36 worker threads. For the scalability evaluation presented in §5.4, we deploy HMFS with multiple metadata servers, each on a separate NUMA node. For the local PM filesystem (Ext4-DAX), evaluations are conducted within the same NUMA node for better performance, without network involvement.

Benchmark. We utilize the standard IOR mdtest benchmark suite (version 3.3.0) [38] to evaluate filesystem metadata performance. OpenMPI v4.1.2 is used to launch parallel mdtest processes as clients. Unless otherwise specified, each mdtest client runs on a dedicated hyperthread, keeps one request in flight, and operates on 1 million files in its unique working directory. File and directory `stat` operations are performed randomly. Each test runs for 3 iterations to obtain the average performance. To approximate realistic conditions, we extend mdtest to generate filenames with varying lengths and naming patterns. We use this modified mdtest exclusively in the sensitivity analysis (§5.8), and use the standard mdtest for others.

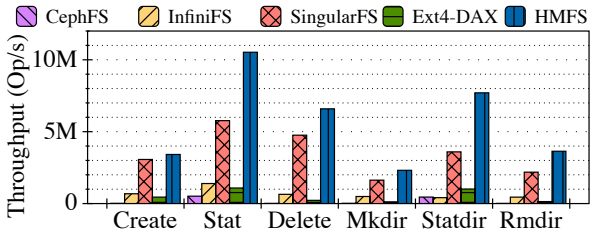


Figure 7: Throughput of metadata operations.

5.2 Throughput

To measure the peak throughput, we tested each filesystem with an increasing number of clients, up to a maximum of 360 (5 machines × 72 hyperthreads). The highest aggregated throughput measured is reported as the peak throughput. Evaluation results are shown in Figure 7. From the results, we have the following observations.

(1) HMFS achieves the highest throughput across metadata operations. For `stat`, HMFS achieves throughput 20.5× that of CephFS, 7.5× InfiniFS, 1.8× SingularFS, and 7.4× ext4-DAX. This is because the decoupling of directory semantics from metadata indexing enables using a hash-based indexing for metadata, thus achieving constant-time complexity for point query. In contrast, SingularFS and InfiniFS rely on tree-based ordered KV Store for directory listing, which imposes higher overhead with logarithmic complexity for all point metadata operations.

(2) For `create`, HMFS achieves throughput 215.7× that of CephFS, 5.0× InfiniFS, 1.1× SingularFS, and 7.4× ext4-DAX. HMFS outperforms SingularFS by 12%, although both create files in a log-free way to avoid the journaling overhead. This is because HMFS updates metadata in-place with lightweight atomic writes, whereas SingularFS relies on atomic KV operations that update metadata out-of-place using copy-on-write. HMFS further optimizes the allocator around the characteristics of metadata objects, leading to faster allocation. Moreover, HMFS manages metadata read-write locks within a DRAM-resident hash table, whereas SingularFS stores locks directly within inode structures on PM and requires persisting lock status synchronously for crash states detection.

(3) The throughput of metadata operations of CephFS and InfiniFS is significantly lower than HMFS. The inferior metadata performance of CephFS on PM primarily stems from kernel-level protocol stack overhead, expensive journaling transactions, and a lack of PM-specific optimizations, as CephFS only uses PM as raw block devices without leveraging its byte-addressability. The inferior performance of InfiniFS mainly arises from two factors. First, InfiniFS relies heavily on transactions for metadata updates, causing significant overhead. Second, it stores metadata in RocksDB, which was originally designed for SSD. Although efforts have been made to optimize RocksDB for PM [13], some of its design features, such as the memtable, the write-ahead log (WAL), software caching, and file-based management are less effective for PM.

5.3 Latency

In this experiment, we launch a single mdtest client to measure the average latency of each type of metadata operation. From the results shown in Figure 8, we have the following observations.

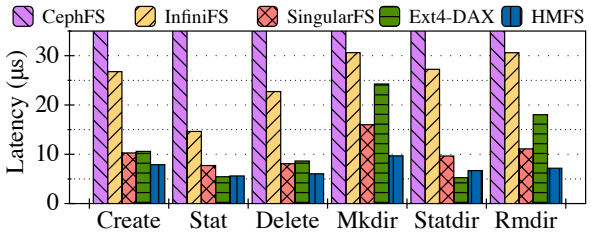


Figure 8: Latency of metadata operations. Note that Ext4-DAX is evaluated at the local node, without network overhead.

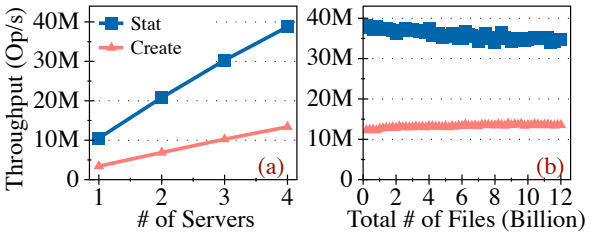


Figure 9: Throughput of `create` and `stat` in HMFS with increasing number of servers (a) and file count in billions (b).

(1) Compared to distributed filesystems, HMFS achieves the lowest average latency across metadata operations. Specifically, for the common file `stat` operation, HMFS substantially reduces latency by 92%, 62%, and 39%, compared to CephFS, InfiniFS, and SingularFS, respectively. The improved performance of HMFS largely derives from its hash-based metadata indexing, lightweight log-free crash-consistency mechanisms, metadata-oriented allocator, and in-memory concurrency control.

(2) Compared to the local filesystems, Ext4-DAX, HMFS achieves lower latency in `create`, `delete`, `mkdir`, and `rmdir`, due to its lightweight crash-consistent metadata updates. Although the Ext4-DAX is evaluated locally without network overhead, HMFS still achieves comparable latency in file and directory `stat` operations.

5.4 Scalability

This section evaluates two key aspects of HMFS's scalability: its ability to scale with an increasing number of metadata servers and its performance stability as the filesystem grows to contain tens of billions of files. First, to assess server scalability, we measure the peak throughput of HMFS with 1 to 4 metadata servers. To saturate the throughput, we use up to 288 mdtest clients (4 machines × 72 hyperthreads), each maintaining 8 requests in flight via coroutines, resulting in a total of 2304 concurrent client requests. Then, to assess performance at a massive metadata scale, we use the 4-server setup (totaling 3 TB of PM) and repeatedly `create` and `stat` 200 million new files until the total number of files reaches tens of billions. Observations from these evaluations are as follows.

(1) As shown in Figure 9(a), HMFS shows near-linear scalability as the number of metadata servers increases. Specifically, with 4 metadata servers, the throughput of HMFS reaches 38.7 Mops/s and 13.4 Mops/s for file `create` and `stat` operations, respectively, representing 3.7× and 3.9× improvements over the single-server setup. This linear scalability mainly results from HMFS's scalable metadata partitioning strategy (§3.1).

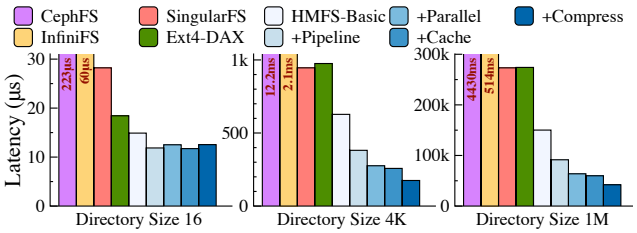


Figure 10: Latency of directory listing operations for directories of varying sizes.

(2) As shown in Figure 9(b), HMFS scales well to tens of billions of files. Specifically, at 12 billion files, the throughput of HMFS sustains 13.4 Mops/s for file `create` operations and 34.7 Mops/s for file `stat` operations. The stable metadata performance arises because HMFS decouples directory semantics from metadata indexing, enabling the use of unordered hash-based metadata indexing, which guarantees constant-time PM access for common point-based metadata operations, regardless of the number of files.

5.5 Directory Listing

In this experiment, we evaluate the latency of directory listing operations for directories of different sizes. Each directory listing operation involves opening the directory stream (`opendir`), iterating through all directory entries (`readdir`), and closing the stream (`closedir`). The measured latency encompasses the entire process. We further analyze how our designs improve directory listing performance by examining the performance gap between HMFS and HMFS-*basic*, which is identical to HMFS without the optimizations for directory listing. We incrementally apply our directory listing optimizations to HMFS-*basic*. Results are shown in Figure 10.

(1) For a large directory containing 1 million files, fully optimized HMFS achieves a directory listing latency of 42 ms, representing only 1% of CephFS, 8% of InfiniFS, 15% of SingularFS, and 15% of Ext4-DAX. The HMFS-*basic* also achieves lower directory listing latency than SingularFS, despite SingularFS relying on efficient range queries provided by the ordered index backend, P-Masstree [34, 42]. This is because, during directory listing, SingularFS traverses P-Masstree's linked leaf nodes and reads linked metadata objects (to obtain the inode number and file type) through pointer chasing. Moreover, the key (containing the filename string) is sliced into 8-byte segments. These segments are stored non-contiguously in the internal tree nodes, requiring additional random accesses, degrading its performance on PM. In contrast, HMFS-*basic* directly traverses inter-object links between child metadata objects during listing. Furthermore, it stores filename string alongside file metadata (containing the inode number and file type) in contiguous cache lines (§3.2), which is cache-friendly for directory listing.

(2) The *+pipeline* and *+parallel* optimizations further reduce directory listing latency by 39% and 30%, respectively, for a large directory containing 1 million files. However, the *+cache* optimization achieves only a modest 6% latency reduction compared to *+parallel*, indicating that splitting the linked list into only two segments is sufficient to shift the latency bottleneck from storage to the network. The *+compression* optimization further enhances performance by reducing the total number of RPCs required for large

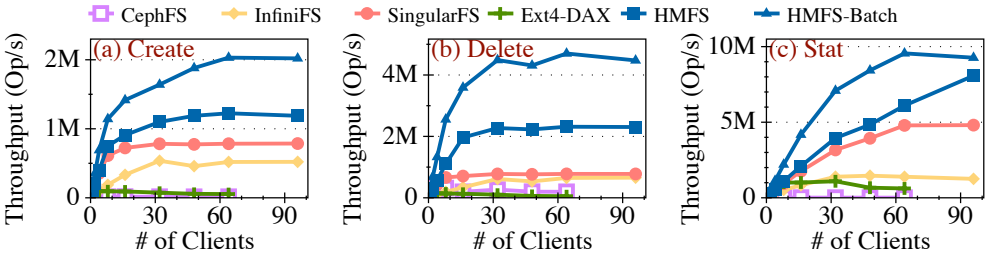


Figure 11: Throughput scalability of concurrent `create` (a), `delete` (b), and `stat` (c) operations in a single shared directory.

directories. Specifically, for directories containing 1 million files, the compression optimization reduces latency by an additional 42%.

(3) The benefits of these optimizations diminish for small directories containing only 16 files. Therefore, at runtime, we use the length of the directory's child linked list (stored as *ll_len* in the directory content metadata) to determine whether to serve directory listings through optimized approaches.

(4) We also measure the end-to-end listing latency including client-side filename sorting (GNU `ls`-style). For a directory with 1 million files, HMFS completes the process in 67 ms, still significantly faster than SingularFS and InfiniFS.

5.6 Shared Directory

In this experiment, we evaluate the throughput scalability of concurrent metadata operations targeting a single shared directory. We incrementally increase the number of mdtest clients and measure the aggregate throughput of file `create`, random `stat`, and `delete` operations. Each client operates on 1 million distinct files within the same shared directory. We also evaluate a variant of HMFS that implements the batching mechanism discussed in §4. In this experiment, the batch size is set to 4, meaning that each metadata server worker thread polls up to four pending metadata requests and processes them together. Evaluation results are shown in Figure 11, from which we have the following observations.

(1) HMFS demonstrates high scalability for `stat` operations but shows limited scalability for `create` and `delete`. Figure 11(c) shows that the throughput of `stat` scales well, approaching HMFS's peak throughput (see Figure 7). This is because each `stat` involves only contention-free, constant-time PM reads from the hash table. In contrast, the throughput of `create` and `delete` saturates at substantially lower levels—approximately 1.2 Mops/s and 2.3 Mops/s, respectively—due to the high contention on updating and persisting the shared directory metadata.

(2) HMFS consistently outperforms other filesystems in handling concurrent operations in a shared directory. Specifically, HMFS achieves throughput improvements of 1.6×, 2.9×, and 1.7× compared to SingularFS for `create`, `delete`, and `stat` operations, respectively. This advantage stems from HMFS's efficient metadata handling. For `create` and `delete`, HMFS requires only a single CAS and a single flush to the directory metadata for updating its `mtime` and `child_ptr`. In contrast, SingularFS requires at least two flushes to the directory metadata: one for acquiring the read lock and another for updating the `mtime` and `ctime`. The lock, containing a local `restartCnt`, must be persisted first, as SingularFS relies

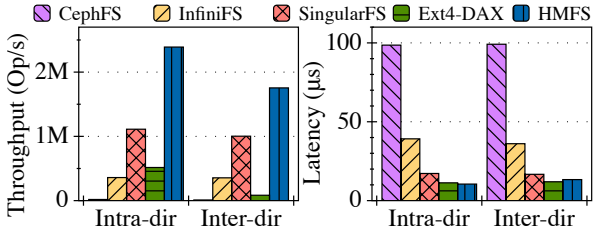


Figure 12: Performance of file rename within the same directory (Intra-dir) and across different directories (Inter-dir).

on it to detect inconsistent metadata after a crash. As noted in previous studies [32], repeated durable writes to the same PM memory location suffer sever performance degradation, due to repeated invalidation of the same cache line. For `stat` operations, HMFS's unordered hash-table indexing is inherently more efficient than SingularFS's ordered tree-based indexing. The other systems lag further behind. InfiniFS is constrained by its transactional metadata updates. Ext4-DAX and CephFS exhibit poor scalability, as their per-directory mutex serializes all concurrent modifications.

(3) HMFS-*batch* notably improves the throughput of concurrent file creation and deletion in a shared directory. As illustrated in Figure 11(a) and (b), HMFS-*batch* achieves throughput improvements of 1.7× and 2.0× over HMFS for `create` and `delete` operations, respectively. In HMFS-*batch*, instead of performing separate CAS and flush operations to the same directory metadata for each file creation/deletion individually, the batching mechanism consolidates multiple updates into a single CAS and flush operation, effectively amortizing the cost of serialization and persistence.

5.7 Rename

In this experiment, we measure the throughput of intra-directory and inter-directory file rename operations using 72 clients, each renaming 1 million files within its unique directory. The latency is measured using a single client. Results are shown in Figure 12.

(1) For intra-directory rename operations, HMFS achieves latency comparable to the local PM filesystem Ext4-DAX, and throughput 134.2× that of CephFS, 6.7× that of InfiniFS, 2.2× that of SingularFS, and 4.6× that of Ext4-DAX. InfiniFS and SingularFS rely on key-value-level transactions to persist at least two entire metadata objects. In contrast, journaling in HMFS persists only required metadata fields, reducing overhead. The rename throughput of Ext4-DAX is severely limited by a global rename lock (`write_seqlock()` on `rename_lock`), which serializes all rename operations.

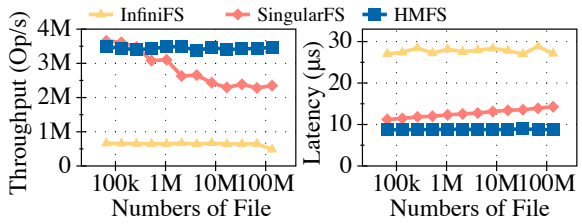


Figure 13: Impact of metadata volume on the throughput and latency of the `create` metadata operations.

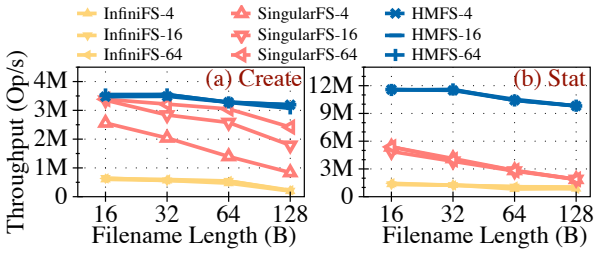


Figure 14: Impact of the pattern and length of filename on the throughput of metadata operations.

(2) Inter-directory rename in HMFS shows higher latency and lower throughput compared to intra-directory rename. This is because an inter-directory rename in HMFS also needs to generate a new indirect metadata object, and updates the target directory's linked list. Also, when source and target directories reside on different metadata servers, HMFS requires a two-phase commit to atomically update metadata across servers.

5.8 Sensitivity Analysis

Here, we extend mdtest to evaluate how metadata volume (i.e., the total number of files in the filesystem), as well as the length and naming pattern of filenames, affect the performance of metadata operations. The throughput is measured using 144 mdtest clients.

Impact of metadata volume. In this experiment, we measure the throughput and latency of file `create` operations under increasing metadata volumes and fixed-length (16-byte) random filenames. As shown in Figure 13, HMFS sustains stable performance as the metadata volume increases from 64K to 100 million files. In contrast, SingularFS experiences a 36% drop in throughput and a 27% increase in latency. This is because SingularFS fundamentally depends on ordered tree-based indexes to support directory listing operations, forcing common point-based metadata operations to incur the overhead of maintaining the global order of metadata objects. The increasing overhead of point operations over a growing number of files causes the performance of SingularFS to degrade notably. InfiniFS performs poorly due to the inefficiency of its LSM-tree-based Pmem-RockDB on PM.

Impact of filename pattern and length. To evaluate the impact of filename naming patterns, we adjust the frequency of common filename prefixes from every 4 files up to every 64 files sharing the same prefix. This is because multiple files sharing the same prefix string is a common naming pattern in real-world workload [1, 37, 49]. To investigate how filename length affects metadata performance, we fix the prefix-sharing pattern to every 16 files and increase the filename length from 16 B to 128 B. From the results shown in Figure 14, we have the following observations.

HMFS exhibits low sensitivity to variations in the naming pattern and length of filenames. The insensitivity of HMFS arises from the internal indexing methods—hash table—which is not affected by the string patterns of keys (contains filename). HMFS's throughput begins to drop as filename length exceeds 64 B. This is because storing longer filenames requires more cache lines in HMFS. Specifically, the throughput of `create` and `stat` in HMFS decrease by 10% and 15% when filenames grow from 16 B to 128 B.

SingularFS exhibits a high sensitivity to the filename pattern. Specifically, under the same filename length, the throughput of file `create` operation in SingularFS varies significantly, suffering up to 65.3% throughput degradation across different naming patterns. SingularFS also exhibits a high sensitivity to the filename length. Specifically, the throughput of SingularFS drops significantly by 47% and 64% for `create` and `stat` operations, respectively, when filenames grow from 16 B to 128 B. This is because SingularFS's trie-of-trees indexing data structure makes it highly sensitive to the distribution and length of keys (containing the filename), causing deeper trees and frequent node splits/merges, which increase random PM accesses and allocation overhead.

Scenario	Launch Time	Consistent stat	Inconsistent stat
Clean Shutdown	53.5ms	5.7µs	-
Crash (100-file dir)	65.7ms	5.7µs	30.5µs
Crash (100K-file dir)	65.7ms	5.7µs	23.4ms

Table 4: Recovery performance of HMFS. *Consistent* `stat`: Latency to access metadata unaffected by a crash. *Inconsistent* `stat`: Latency to access metadata left inconsistent by an interrupted operation, thus triggering on-demand recovery.

5.9 Recovery Overhead

This section evaluates two facets of HMFS's recovery mechanism: the time required to restart the metadata service after a failure and the overhead of accessing metadata left inconsistent by a crash. First, we measure the server launch time following both a normal shutdown and a simulated crash, at a scale of 100 M files. Then, we measure the latency of `stat` operation on inconsistent file metadata whose creation was interrupted by a crash. We test this scenario in both a small directory (100 files) and a large directory (100 K files). From the results in Table 4, we make the following observations.

(1) HMFS recovers rapidly from crashes, requiring only 65.7 ms under 100 M files. Specifically, after a normal shutdown, HMFS completes recovery in 53.5 ms, which involves opening the PM pool, loading free lists from the PM checkpoint concurrently, initializing lock tables, and activating the RPC services. Notably, opening the 720 GB PM pool via PMDK takes 38.2 ms, making it the dominant contributor to the recovery overhead. After a crash, HMFS restores metadata services in 65.7 ms, which is comparable to the recovery overhead after a normal shutdown. This rapid crash recovery is made possible by two key design features of HMFS: on-demand detection and recovery of inconsistent metadata objects (§3.3); and instant post-crash allocator availability (§3.5).

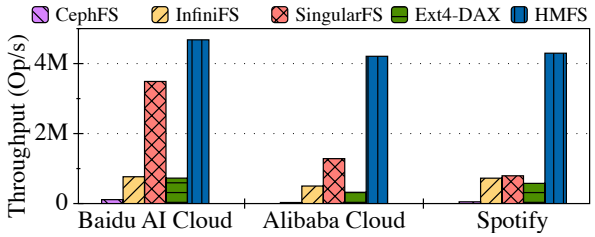


Figure 15: Performance under production workloads.

Workload	Metadata Operation Ratios
Baidu AI Cloud [59]	93.05% stat, 1.44% create, 1.14% delete, 0.08% mkdir, 0.04% rmdir, 0.12% rename, 0.92% list, 3.21% others
Alibaba Cloud [40]	64.95% stat, 0.28% statdir, 9.58% create, 11.88% delete, 0.01% mkdir, 9.29% rename, 3.92% list, 0.09% others
Spotify [45]	81.91% stat, 3.96% statdir, 2.70% create, 0.75% delete, 0.02% mkdir, 1.30% rename, 9.01% list, 0.35% others

Table 5: Ratios of metadata operations in production workloads from Baidu AI Cloud, Alibaba Cloud, and Spotify.

(2) The on-demand recovery latency for an inconsistent metadata object is proportional to its parent directory's size due to child list traversal. For small directories, the recovery time is low: only 5.35× that of a normal stat operation.

5.10 Production Workloads

In this section, we evaluate HMFS's performance under realistic cloud scenarios by emulating production workloads based on metadata operation ratios from three well-known industrial systems [2]. Specifically, we generated synthetic workloads by mirroring the metadata operation ratios reported from the production distributed filesystem clusters of Pangu at Alibaba Cloud [40], CFS at Baidu AI Cloud [59], and HDFS at Spotify [45]. Table 5 summarizes the detailed operation ratios in these workloads.

Experiments are conducted using 1 metadata server and 144 clients. Each client operates on 1 million files according to the metadata operations ratios of each workload. The aggregate throughput under each workload is shown in Figure 15. From the figure, we can observe that HMFS achieves the highest throughput across all three workloads. Specifically, the metadata operation throughput of HMFS outperforms SingularFS, InfiniFS, Ext4-DAX, and CephFS by at least factors of 1.3×, 5.9×, 6.4×, and 41.6×, respectively.

6 Related Work

Metadata organization. Many distributed filesystems leverage ordered KV stores to manage metadata. For example, InfiniFS [40], IndexFS [52], DeltaFS [67], TableFS [51], and ShardFS [64] organize filesystem metadata in RocksDB and rely on its range query capability to reconstruct directory entries during listing. Tectonic [48] uses ZippyDB [20], a distributed KV store, which internally runs RocksDB. Others, like HopsFS [45] and CalvinFS [57], build metadata services on top of full-fledged distributed databases, and depend on the ordered index scan for directory listing. CFS [59] and 3FS [4] also maintain directory semantics with databases, but store file metadata separately in RocksDB for better performance. Early distributed filesystems like GFS [22], HDFS [55], and MooseFS [11] manage the entire namespace in the DRAM of a single, centralized metadata server, which inherently limits scalability. CephFS [60] and DAOS [21] provide filesystem interfaces atop underlying object storage. They maintain directory semantics by encapsulating all file metadata within the same directory into a single object. To accelerate metadata access, CephFS caches metadata in DRAM of metadata servers [63]. DAOS instead uses PM as fast storage for these metadata objects; however, this approach inherently lack support for hardlinks and incurs a long software stack for metadata lookup. Local PM-oriented filesystems, such as Ext4-DAX [43], BPFS [10], PMFS [18], and NOVA [65], explicitly store directory entries in directory data blocks. However, this method incurs frequent cross-node modifications, making it unsuitable for distributed scenarios. In contrast, HMFS combines an unordered index with per-directory linked lists, achieving both O(1) lookup and fast directory listing.

Metadata crash consistency. One common strategy is to leverage transactions from the underlying KV store or database. For instance, InfiniFS [40] and IndexFS [52] use RocksDB transactions to update multiple metadata KV pairs atomically, while HopsFS [45] and CFS [59] depend on database transactions. This approach, however, introduces high overhead from locking and coordination protocols. SingularFS [23] replaces transactions with ordered KV operations, but still rely on the atomicity of KV operations which internally update metadata out-of-place via copy-on-write (CoW). Local PM filesystems typically employ logging or CoW techniques to ensure metadata consistency. For example, NOVA [65] adopts a log-structured design; BPFS [10] updates metadata out-of-place using CoW; and PMFS [18] updates metadata in-place using write-ahead journaling. ByteFS [36] also relies on logging to ensure metadata consistency. These methods require additional writes and many flushes, causing high overhead. In contrast, HMFS significantly reduces the consistency overhead to a single atomic write (8-byte or 16-byte), or a few ordered atomic writes.

7 Conclusion

This paper presents HMFS, an efficient distributed metadata service designed for modern high-speed networks and byte-addressable PM, by decoupling directory listing semantics from underlying metadata indexing. Extensive evaluations demonstrate that HMFS significantly outperforms state-of-the-art distributed metadata services, and sustains high-performance for tens of billions of files.

Acknowledgments

We thank all reviewers for their insightful comments. This work is supported by the National Key R&D Program of China (Grant No. 2024YFB4505201), the National Natural Science Foundation of China (Grant No. U22B2023, 62472242), and the Young Elite Scientists Sponsorship Program by CAST (2023QNRC001).

[2] To the best of our knowledge, no open-source metadata operation traces from large-scale production distributed filesystems are available, largely due to commercial confidentiality. While previous studies [40, 45, 59] do not release raw traces, they provide valuable insights by publishing these ratios.

References

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. 2007. A five-year study of file-system metadata. *ACM Trans. Storage* 3, 3 (Oct. 2007), 9–es. doi:10.1145/1288783.1288788

[2] Alluxio, Inc. 2025. Alluxio: Data Orchestration for Analytics and AI. https://www.alluxio.io.

[3] Amazon. 2018. Introducing Amazon EC2 C5n Instances Featuring 100 Gbps of Network Bandwidth. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/.

[4] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, Jianzhong Guo, Yongqiang Guo, Zhe Fu, Ying He, Panpan Huang, Jiashi Li, Wenfeng Liang, Xiaodong Liu, Xin Liu, Yiyuan Liu, Yuxuan Liu, Shanghao Lu, Xuan Lu, Xiaotao Nie, Tian Pei, Junjie Qiu, Hui Qu, Zehui Ren, Zhangli Sha, Xuecheng Su, Xiaowen Sun, Yixuan Tan, Minghui Tang, Shiyu Wang, Yaohui Wang, Yongji Wang, Ziwei Xie, Yiliang Xiong, Yanhong Xu, Shengfeng Ye, Shuiping Yu, Yukun Zha, Liyue Zhang, Haowei Zhang, Mingchuan Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, and Yuheng Zou. 2024. Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 83, 23 pages. doi:10.1109/SC41406.2024.00089

[5] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 121–134. https://www.usenix.org/conference/atc19/presentation/bijlani

[6] Ceph Documentation. 2025. CephFS Directory Fragmentation. https://docs.ceph.com/en/latest/cephfs/dirfrags/.

[7] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 9.

[8] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. 2024. RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 141–157. https://www.usenix.org/conference/fast24/presentation/cho

[9] Yann Collet. 2025. LZ4 – Extremely Fast Compression algorithm. https://github.com/lz4/lz4.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 133–146. doi:10.1145/1629575.1629589

[11] Core Technology. 2008. MooseFS: Open Source Distributed File System. https://moosefs.com/.

[12] Intel Corporation. 2019. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[13] Intel Corporation. 2024. PMEM-RocksDB. https://github.com/pmem/pmem-rocksdb.

[14] Intel Corporation. 2025. Persistent Memory Development Kit (PMDK). https://github.com/pmem/pmdk.

[15] Kioxia Corporation. 2022. XL-FLASH Storage Class Memory Solution. https://www.kioxia.com/en-jp/business/news/2022/20220802-1.html.

[16] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 478–493. doi:10.1145/3341301.3359637

[17] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. doi:10.1145/2815400.2815425

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. doi:10.1145/2592798.2592814

[19] Facebook. 2012. RocksDB. https://rocksdb.org.

[20] Facebook. 2018. ZippyDB: Facebook's Key-Value Storage System. https://engineering.fb.com/2018/05/22/core-data/zippydb/.

[21] The DAOS Foundation. 2025. The Open-Source Storage Platform for AI & HPC. https://daos.io/.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 29–43. doi:10.1145/945445.945450

[23] Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu. 2023. SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 915–928. https://www.usenix.org/conference/atc23/presentation/guo

[24] Red Hat. 2019. Linux sysadmin basics: User account management with UIDs and GIDs. https://www.redhat.com/en/blog/user-account-gid-uid.

[25] Dean Hildebrand and Denis Serenyi. 2021. 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.

[26] The IEEE and The Open Group. 2024. POSIX.1-2024: 4.12 File Times Update. https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap04.html.

[27] The IEEE and The Open Group. 2024. POSIX.1-2024: readdir, readdir_r - read a directory. https://pubs.opengroup.org/onlinepubs/9799919799/functions/readdir.html.

[28] Intel. 2025. Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol 3 , Ch 10.1.1). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[29] Juicedata, Inc. 2025. JuiceFS: High-Performance Cloud-Native Distributed File System. https://juicefs.com/zh-cn/.

[30] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. doi:10.1145/3477132.3483567

[31] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. doi:10.1145/3341301.3359631

[32] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 105–119. doi:10.1145/3419111.3421294

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 1–16.

[34] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. doi:10.1145/3341301.3359635

[35] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A Loosely-Coupled Metadata Service for Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. doi:10.1145/3126908.3126928

[36] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. 2025. ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 116–132. doi:10.1145/3669940.3707250

[37] Seung-Hwan Lim, Hyogi Sim, Raghul Gunasekaran, and Sudharshan S. Vazhkudai. 2017. Scientific user behavior and data-sharing trends in a petascale file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 46, 12 pages. doi:10.1145/3126908.3126924

[38] LLNL and Contributors. 2025. IOR and mdtest: Parallel I/O and Metadata Benchmarking Tools. https://github.com/hpc/ior.

[39] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. doi:10.14778/3389133.3389134

[40] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. 2022. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 313–328. https://www.usenix.org/conference/fast22/presentation/lv

[41] Linux manual page. 2024. readdir(3) — Linux manual page (POSIX.1-2008). https://man7.org/linux/man-pages/man3/readdir.3.html.

[42] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. doi:10.1145/2168836.2168855

[43] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, and Laurent Vivier. 2010. The new ext 4 filesystem : current status and future plans. https://api.semanticscholar.org/CorpusID:267893896

[44] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. 2021. Simurgh: a fully decentralized and secure NVMM user space file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 46, 14 pages. doi:10.1145/3458817.3476180

[45] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 89–104. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi

[46] NVIDIA Corporation. 2024. NVIDIA ConnectX InfiniBand Adapters. https://www.nvidia.com/en-us/networking/infiniband-adapters/.

[47] OpenSFS and EOFS. 2025. Lustre File System. https://www.lustre.org/.

[48] 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. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[49] Aleatha Parker-Wood, Darrell D. E. Long, Ethan Miller, Philippe Rigaux, and Andy Isaacson. 2014. A File By Any Other Name: Managing File Names with Metadata. In *Proceedings of International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR 2014)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2611354.2611367

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

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

[52] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. 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* (New Orleans, Louisana) *(SC '14)*. IEEE Press, 237–248. doi:10.1109/SC.2014.25

[53] Samsung. 2024. Samsung Memory-Semantic CXL SSD. https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.

[54] Jiwu Shu. 2024. *Data Storage Architectures and Technologies*. Springer.

[55] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. doi:10.1109/MSST.2010.5496972

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

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

[58] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), 461–472. https://api.semanticscholar.org/CorpusID:53080051

[59] Yiduo Wang, Yufei Wu, Cheng Li, Pengfei Zheng, Biao Cao, Yan Sun, Fei Zhou, Yinlong Xu, Yao Wang, and Guangjun Xie. 2023. CFS: Scaling Metadata Service for Distributed File System via Pruned Scope of Critical Sections. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 331–346. doi:10.1145/3552326.3587443

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

[61] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (Tampa, Florida) *(SC '06)*. Association for Computing Machinery, New York, NY, USA, 122–es. doi:10.1145/1188455.1188582

[62] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. 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* (Reno, Nevada) *(PDSW '07)*. Association for Computing Machinery, New York, NY, USA, 35–44. doi:10.1145/1374596.1374606

[63] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*. IEEE Computer Society, USA, 4. doi:10.1109/SC.2004.22

[64] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. 2015. 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* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 236–249. doi:10.1145/2806777.2806844

[65] Jian Xu and Steven Swanson. 2016. 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* (Santa Clara, CA) *(FAST'16)*. USENIX Association, USA, 323–338.

[66] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. 2022. End-to-End I/O Monitoring on Leading Supercomputers. *ACM Trans. Storage* (nov 2022). doi:10.1145/3568425 Just Accepted.

[67] Qing Zheng, Charles D. Cranor, Gregory R. Ganger, Garth A. Gibson, George Amvrosiadis, Bradley W. Settlemyer, and Gary A. Grider. 2021. DeltaFS: a scalable no-ground-truth filesystem for massively-parallel computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 48, 15 pages. doi:10.1145/3458817.3476148

[68] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *Proceedings of the 9th Parallel Data Storage Workshop* (New Orleans, Louisiana) *(PDSW '14)*. IEEE Press, 1–6. doi:10.1109/PDSW.2014.7

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