SwitchFS: Asynchronous Metadata Updates for Distributed Filesystems with In-Network Coordination

Jingwei Xu, Mingkai Dong, Qiulin Tian, Ziyi Tian, Tong Xin, and Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

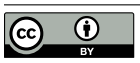
Abstract

Distributed filesystem metadata updates are typically synchronous. This creates inherent challenges for access efficiency, load balancing, and directory contention, especially under dynamic and skewed workloads. This paper argues that synchronous updates are overly conservative. We propose SwitchFS with *asynchronous metadata updates* that allow operations to return early and defer directory updates until reads, both hiding latency and amortizing overhead. The key challenge lies in efficiently maintaining the synchronous POSIX semantics of metadata updates. To address this, SwitchFS is co-designed with a programmable switch, leveraging the limited on-switch resources to track directory states with negligible overhead. This allows SwitchFS to aggregate and apply delayed updates efficiently, using batching and consolidation before directory reads. Evaluation shows that SwitchFS achieves up to 13.34× and 3.85× higher throughput, and 61.6% and 57.3% lower latency than two state-of-the-art distributed filesystems, Emulated-InfiniFS and Emulated-CFS, respectively, under skewed workloads. For real-world workloads, SwitchFS improves end-to-end throughput by 21.1×, 1.1×, and 0.3× over CephFS, Emulated-InfiniFS, and Emulated-CFS, respectively.

CCS Concepts: • **Information systems → Distributed storage**; • **Social and professional topics → File systems management**; • **Networks → In-network processing**; **Programmable networks**.

Keywords: Distributed file system, Metadata management, Programmable switches

ACM Reference Format:
Jingwei Xu, Mingkai Dong, Qiulin Tian, Ziyi Tian, Tong Xin, and Haibo Chen, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University . 2026. SwitchFS: Asynchronous Metadata Updates for Distributed Filesystems with In-Network Coordination. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3767295.3769349



EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769349

1 Introduction

Metadata performance is a critical factor in the efficiency of distributed filesystems (DFSs) deployed in modern data centers. To manage the filesystem namespace and file attributes, modern DFSs [15, 38, 45, 51, 59, 66, 75, 76] typically employ a dedicated metadata service. Before accessing file data, clients must first contact this service to check permissions and obtain the data location. Because files in data centers are often small [3, 5, 17, 38, 61, 74, 75, 81, 82], the relative overhead of metadata operations becomes significant. For instance, traces from Baidu AI Cloud [75] report that metadata operations account for 67%–96% of all filesystem requests and dominate I/O completion time. Unlike data access bandwidth, which scales nearly linearly with the number of data servers, metadata throughput is difficult to scale due to hierarchical dependencies [38, 45, 59, 75]. As a result, metadata frequently becomes the performance bottleneck. In large-scale applications such as deep learning model training [34, 77] and data analytics [34], this bottleneck can lead to underutilized data bandwidth and compute resources, ultimately degrading end-to-end performance [34, 74, 77].

A series of studies [32, 38, 45, 47, 51, 54, 55, 57, 59, 72, 75, 76] have been made on scaling DFS metadata performance, yet there are still challenges remaining. First, while DFSs typically partition the directory tree across multiple metadata servers to scale out, selecting partitioning granularity presents an inherent trade-off between load balance and operation overhead, making it hard to achieve both [46]. Coarse-grained partitioning [15, 45, 47, 51, 59, 76] groups file inodes with their parent directory, making large directories easily become hotspots, especially in skewed workloads [1, 74]. In contrast, fine-grained partitioning [14, 38, 75] evenly shuffles file inodes across metadata servers, achieving good load balance but breaking the colocation between inodes and parent directories; as metadata is partitioned across different servers, metadata operations need to use (costly) distributed transactions to update them consistently. Second, state-of-the-art distributed metadata services often suffer from contention on directory metadata. Specifically, parent-related operations (e.g., *mkdir* and *create*) contend on the parent directory, resulting in degraded scalability [16, 75].

A fundamental reason for the aforementioned challenges is synchronous metadata updates. As filesystem semantics require durable visibility (the effect of an operation is visible to subsequent operations) and atomicity (no partial updates), state-of-the-art DFSs typically use synchronous transactions.

Unfortunately, this approach incurs cross-server coordination overhead and metadata contention on the critical path, leading to high latency and throughput bottlenecks.

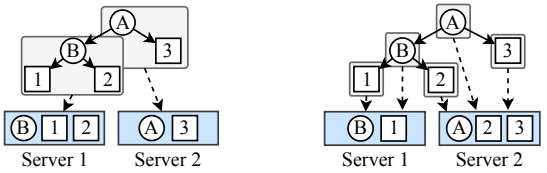
In this paper, we argue that *synchronous updates are overly conservative*. We observe that *directories are typically not read immediately after being updated*. It provides an opportunity for *asynchronous updates*, deferring directory updates until the next read, to hide update latency and to batch updates.

However, to preserve the durable visibility of metadata updates, a mechanism is required to identify directories with pending updates to avoid stale reads. A straightforward approach would be to dedicate a server on the critical path of metadata operations to track directory updates and to notify directory reads. It introduces an additional RTT, and the dedicated server can become a bottleneck. In contrast, we find that the programmable switch — naturally positioned on the critical path of metadata operations and capable of extremely high packet processing throughput — is an ideal candidate to coordinate asynchronous updates and reads.

Based on these insights, we present **SwitchFS**, an asynchronous distributed filesystem co-designed with a programmable switch for operation coordination. SwitchFS is POSIX-compliant, with the goal of achieving good load balance, low latency, and high scalability under skewed workloads at the same time. SwitchFS has the following design features.

First, we propose an efficient protocol for asynchronous metadata updates, offering two benefits: (a) it enables most metadata operations to complete in a single RTT while using fine-grained partitioning for load balance, resolving the trade-off faced by previous DFSs; (b) it merges contending metadata updates before applying them, mitigating metadata contention. Specifically, when an operation updates objects across multiple servers (typically a file and its parent directory), SwitchFS logs the directory update on the server hosting the file's inode and marks the directory as *scattered* in the programmable switch. When reading a directory, the switch signals if the directory is scattered. If so, SwitchFS *aggregates* the deferred updates from other servers before accessing the directory. To further expedite the aggregation process and reduce contention on directory metadata, SwitchFS consolidates metadata updates to the same directory before aggregation, leveraging the inherent commutativity of updates to directory attributes [2, 7, 33, 69, 71, 75, 78].

Second, we leverage a programmable switch to efficiently track scattered directories within the network. Our insight is that, thanks to its high packet processing capability and central position in the network, the programmable switch serves as a superior coordinator compared to a standard server. Given the limited on-switch memory and compute resources, we design a compact in-network dirty set with high memory utilization and low conflict rate by organizing the switch registers in a set-associative manner. By inspecting specific headers in packets, the dirty set can insert, query, and remove directory *fingerprints* at line rate, and redirect packets as needed, enabling SwitchFS to query and update directory states at negligible overhead.



(a) P/C grouping **(b)** P/C separation

Fig. 1: **Metadata partitioning approaches.**

	P/C Grouping	P/C Separation
Single-inode Op.	Local	Local
Double-inode Op.	Local/Cross-server	Cross-server
Load Balancing	Directory hotspots	Balanced
Example FS	CephFS [76], InfiniFS [45]	GlusterFS [14], CFS [75]

Tab. 1: **Trade-offs among partitioning approaches. "P/C" stands for "parent-children".** With P/C grouping, *mkdir*, *rmdir* are cross-server. With P/C separation, *mkdir*, *rmdir*, *create*, *delete* are cross-server.

Our evaluation shows that SwitchFS achieves up to 13.34× higher throughput than Emulated-InfiniFS [45], and 57.3% lower latency than Emulated-CFS [75]. For real-world workloads, SwitchFS improves end-to-end throughput by 21.1×, 1.1×, and 0.3× over CephFS, Emulated-InfiniFS, and Emulated-CFS, respectively.

In summary, this paper makes the following contributions.

- **An analysis of key challenges in scaling DFS metadata performance (§3.2).** We identify that the inherent trade-off in namespace partition granularity and the directory contention makes metadata performance suboptimal.
- **A distributed filesystem, SwitchFS, adopting asynchronous metadata updates (§5).** SwitchFS resolves the trade-off between operation efficiency and load balancing without compromising consistency with asynchronous metadata updates (§5.2), and mitigates contention over directory metadata with change-log compaction (§5.3).
- **In-network dirty set that tracks directory states (§6).** To the best of our knowledge, SwitchFS is the first to incorporate in-network optimization into a DFS.

2 Background

2.1 Distributed Metadata Partitioning

Many DFSs partition the directory tree across a cluster of metadata servers to scale out metadata service [38, 45, 47, 51, 75, 76]. The partitioning approach is an important design choice that impacts operation overhead and load distribution, and ultimately, overall performance and scalability. Existing approaches can be classified into *parent-children grouping* (*P/C grouping*) and *parent-children separation* (*P/C separation*), depending on whether file inodes are colocated with their parent directories. The approaches are illustrated in Fig. 1 and compared in Tab. 1.

Category	Ratio	Detailed Operation Ratio			
Dir. Update	30.76%	create	9.58%	delete	11.88%
		mkdir	0.01%	rmdir	0.01%
		file rename	9.29%		
Dir. Read	4.19%	statdir	0.28%	readdir	3.91%
Others	65.05%	open/close	52.59%	stat	12.35%
		others	0.10%		

Tab. 2: Ratio of metadata operations in workloads from deployed PanguFS instances in Alibaba [45].

Parent-children grouping. *P/C grouping* colocates file inodes with their parent directories, while directory inodes may be distributed independently for scalability, e.g., subtree partitioning [13, 66] and per-directory hashing [15, 45, 47, 51, 59], as illustrated in Fig. 1(a). These studies adopt P/C grouping since it achieves good metadata locality, thus reducing operation overhead. As metadata operations often simultaneously update metadata object and its parent directory, e.g., file *create*, with parent and children colocated, some updates are performed on the same server, avoiding costly cross-server coordination. The downside of P/C grouping is that it places all file inodes within the same directory onto a single server, leading to substantial load imbalance in skewed workloads, thereby undermining scalability.

Parent-children separation. *P/C separation* distributes file inodes independently of the location of their parent directories, e.g., per-file hashing [14, 75], as illustrated in Fig. 1(b). By evenly distributing metadata objects, *P/C separation* achieves good load balance. However, metadata operations suffer from expensive cross-server coordination to update multiple objects across servers consistently.

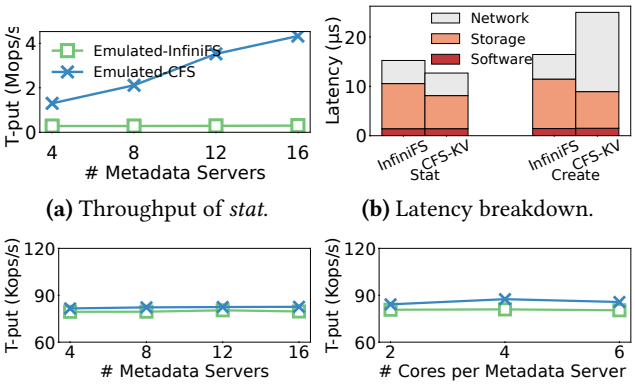
2.2 Programmable Switch

A programmable switch is a packet-switching device equipped with programmable ASICs and stateful memory [6, 21, 49], enabling in-network co-design for distributed systems. It can do user-defined manipulation on packets while forwarding them at extremely high speeds. Compared with manipulating packets with a standard server, the programmable switch has two advantages: (a) its central location in the network grants it a global view of all the communications; (b) the ASIC-based hardware processes packets at line rate, providing high throughput and low latency. We support these statements with our evaluation in §7.3.3.

3 Motivation and Challenges

3.1 Characteristics of Datacenter Workloads

A directory is typically not read immediately after being updated. Tab. 2 presents the ratio of metadata operations in three deployed PanguFS instances in Alibaba [45] under various workloads like data processing, object storage, and block storage. Operations that update directories (i.e., *create*, *delete*, *mkdir*, *rmdir*, and *rename*) constitute 30.76% of all metadata operations, while operations that read directories (i.e., *readdir* and *statidr*) constitute merely 4.19%. By the pigeonhole principle, the disparity implies that at least $(30.76\% - 4.19\%)/30.76\% = 86.3\%$ of the directory updates are not immediately followed by a directory read, satisfying that either (a) the updated directory is never read after the update or (b) the updated directory is updated by at least one other operation before it is read. Thus, applying directory updates in a delayed manner can help hide latency and mitigate contention. Notably, the partial order of dependent updates (e.g., creation and deletion of the same file) should be preserved, which SwitchFS guarantees.



(a) Throughput of *stat*. **(b)** Latency breakdown.

(c) Throughput of *create* with varying numbers of servers. **(d)** Throughput of *create* with varying # of cores per server.

Fig. 2: Throughput scalability and latency of metadata operations. Operations are performed in a shared directory. DFSs have eight servers, each with four cores by default.

Datacenter workload is skewed along multiple dimensions. This skewness challenges metadata management and can hinder performance scaling if the load balance is poor.

- *Unbalanced directory hierarchies.* Analyses of deployed DFSs [10, 58, 59] show that while most directories contain fewer than 128 entries, some directories include over a million entries. Such an unbalanced hierarchy complicates the balancing of inode distribution and metadata load.
- *Directory hotspots.* In addition, applications commonly group related files into directories and access them within short temporal intervals [38, 45, 51]. This creates temporal hotpots, which further complicates load balancing.

3.2 Challenges for Metadata Service

Modern data center applications demand DFS metadata operations with high scalability, high throughput, and low latency. However, there are several challenges in the design.

Challenge #1:. *There is an inherent trade-off between low operation overhead and load balance in the design of directory tree partitioning.*

To reduce operation overhead, it is important to colocate files' inodes with the parent directory. Metadata operations

(e.g., *create* and *delete*) often update a metadata object together with its parent directory's metadata. With the objects colocated, the operation can be executed on a single server, avoiding expensive cross-server coordination [45].

However, as files in the same directory are often related and accessed in a short time, grouping them makes the filesystem vulnerable to hotspots [38, 45, 51]. Therefore, to improve load balance, it is important to scatter files of the same directory across servers.

Unfortunately, colocating inherently conflicts with scattering, making it difficult to have the best of both worlds.

To evaluate how this trade-off affects performance, we compare the performance of Emulated-InfiniFS (E-InfiniFS) [45] and Emulated-CFS (E-CFS) [75], two state-of-the-art DFSs adopting P/C grouping and P/C separation, respectively. In Fig. 2(a), when performing *stat* on uniformly random files in a shared directory, E-CFS's throughput scales linearly since files are evenly distributed across servers, while E-InfiniFS fails to scale because all file inodes reside on the same server, indicating that E-CFS has better load balance. In contrast, in Fig. 2(b), E-CFS suffers higher *create* latency than E-InfiniFS, mainly due to higher network overhead from cross-server coordination, indicating that E-InfiniFS has lower operation overhead.

Challenge #2:. *Concurrent operations in the same directory suffer low inter- and intra-server parallelism due to contention on the parent directory, hindering performance scaling.*

Regardless of the partitioning approach, the contention on directory metadata often becomes a performance bottleneck. For example, when concurrently creating files in a single directory (e.g., during data preparation [77], compiling, decompression, etc.), each *create* operation updates the attributes and entry list of the parent directory. Although the creation of individual files could be parallelized across multiple servers (under P/C separation), updates to the parent directory are serialized at the server that stores it, resulting in low *inter-server parallelism*. Also, as conventional approaches rely on transactions and locks to guarantee the atomicity of metadata update, the server cannot leverage multicore parallelism, resulting in low *intra-server parallelism*.

In Fig. 2(c) and Fig. 2(d), both DFSs' throughput does not scale with the number of servers and of cores per server. It indicates that both inter-server and intra-server parallelism are restricted by the contention on the parent directory.

4 Overview

To overcome the inherent limitations in §3.2, SwitchFS coordinates asynchronous metadata operations in the network to promote parallelism and to hide latency. We overview the design rationale and architecture in this section, then present the workflow in §5 and the switch data plane in §6.

4.1 Design Rationale

Key idea: asynchronous directory updates. Our key idea is to *leverage asynchronous updates to hide cross-server coordination overhead and exploit metadata operation parallelism*. File-system operations (except for *rename*, discussed in §5.2) update at most two metadata objects: the target object and its parent directory [16]; thus, by deferring updates to the remote parent directory, SwitchFS can execute operations locally on a single server. This approach hides cross-server overhead and enables concurrent updates to a parent directory to execute in parallel.

Performance enabler: programmable switch. To ensure that a delayed update is visible to subsequent accesses, SwitchFS leverages a centralized programmable switch to store directory state at line rate speed and sub-RTT latency.

However, due to limited on-chip memory and computational resources, it is non-trivial to maintain all directories' states for an entire filesystem. We observe that, *although the total number of directories can be huge, the set of directories whose metadata has been modified in a recent time interval is small*. For example, consider a filesystem capable of creating or deleting one million files per second. The number of directories modified per 100 milliseconds is 100 thousand at most, small enough to fit into the switch's on-chip memory. Therefore, by periodically applying delayed updates and clearing the on-switch data structure, on-switch state tracking is feasible.

4.2 Overall Architecture

As depicted in Fig. 4, SwitchFS consists of three kinds of components: clients, metadata servers, and programmable switches. Without loss of generality, we assume a single programmable switch for clarity, and defer the discussion of scaling to multiple switches to §6.4.

Clients. SwitchFS provides a user-space library (LibFS) for clients to communicate with servers. The client maintains a metadata cache to accelerate path resolution, which caches only directory metadata.

Metadata servers. SwitchFS distributes metadata objects across metadata servers with fine-grained partitioning. A server stores its metadata in a *key-value store* (i.e., RocksDB[60]). The server tracks deferred modifications to remote directories in a per-directory *change-log*, and tracks recently deleted directories in its *invalidation list*. The server uses a *write-ahead log* (WAL) for crash recovery.

In-network dirty set. The programmable switch monitors network traffic to maintain a centralized *dirty set* of delayed updates. To efficiently utilize limited on-switch memory, the dirty set is stored in a multi-slot hash table structure.

	Key	Value	Partition By
Dir Metadata	*pid, name*	*id, timestamps, perm., etc.*	the key
Dir Entry	*pid, name*	*file type, permissions*	N/A
File Metadata	*pid, name*	*full regular file metadata*	the key

Tab. 3: **Metadata schema.** Each directory has a 256-bit *id*. *pid* refers to the *id* of the parent directory. A directory's entry list consists of multiple *Dir Entry* key-value pairs, which are stored on the same server as the directory.

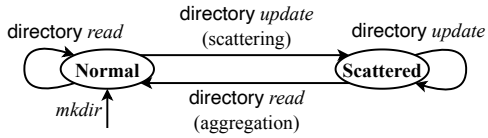


Fig. 3: **Directory state transition.**

4.3 Metadata Schema

Tab. 3 presents the metadata schema of SwitchFS. Each directory is assigned a unique 256-bit identifier *id* upon creation. Metadata objects (i.e., inodes and dentries) are stored as key-value pairs, where the key is the concatenation of the parent directory's *id* (*pid*) and the file/directory *name*, and the value is the metadata attributes.

SwitchFS employs the P/C separation approach for load balance, partitioning file/directory metadata by hashing the *(pid, name)* pair. A directory's entry list is stored as multiple *Dir Entry* key-value pairs, all of which are stored on the same server as the directory's inode.

Though not shown in the table, SwitchFS generates a 49-bit *fingerprint* for each directory by hashing *(pid, directory name)*. The fingerprint, which serves as the directory identifier within the switch, is designed to fit within the switch's limited register width. Since multiple directories may have the same fingerprint, SwitchFS ensures that all directories with the same fingerprint (called a *fingerprint group*) are partitioned to the same server, to simplify conflict handling.

5 SwitchFS Design

In this section, we present how SwitchFS works, including asynchronous metadata operations (§5.1 and §5.2), change-log compaction (§5.3) and fault handling (§5.4). We discuss the correctness of crash recovery (§A.1) and the consistency properties (§A.2) in the appendix.

5.1 Directory State and Transition

State definition. In SwitchFS, a directory is in one of two states: *normal state*, indicating that all returned updates to the directory have been applied to its inode, or *scattered state*, indicating that there are not-yet-applied updates in one or more change-logs. The on-switch dirty set tracks directory states, and metadata operations can query or update it.

State transition. Fig. 3 shows the transitions between directory states. A directory is created in normal state, where its inode is up-to-date. When SwitchFS makes an asynchronous update, the directory transitions to scattered state, indicating that there is one or more delayed updates not yet applied to the inode. When SwitchFS reads a directory in scattered state, it aggregates and applies the delayed updates from other servers, returning the directory to normal state.

Transition granularity. As mentioned in §4.3, the switch identifies a directory by its fingerprint. Therefore, the granularity of state transition is fingerprint group, i.e., all directories with a specific fingerprint. A *metadata aggregation* aggregates all directories in the target fingerprint group. Since SwitchFS places all directories in the same fingerprint group to the same server, this does not complicate aggregation.

5.2 Asynchronous Metadata Operations

Overview. Metadata operations can be classified into three categories according to the number of inodes they access [16]. We summarize how SwitchFS handles them below.

- *Double-inode operations*, including *create*, *delete*, *mkdir*, *rmdir*, access inodes of the target object and its parent directory. These operations defer the update to the parent directory and finalize execution on a single metadata server; we elaborate upon them in §5.2.1 and §5.2.3.
- *Single-inode operations* access the inode of a file (e.g., *open*, *close*, *stat*) or a directory (e.g., *statdir*, *readdir*). The former are performed synchronously as in traditional DFSs like InfiniFS [45] and CFS [75]. We omit their explanation for brevity. The latter read directory attributes, and need to check inode staleness and possibly trigger an aggregation; we discuss them in §5.2.2.
- *Rename* is the most complex metadata operation, as it updates up to four inodes. SwitchFS ensures *rename* consistency through distributed transactions, and avoids *orphaned loops* [4, 45, 75] (i.e., two directories become each other's ancestor) by using a centralized rename coordinator, as in previous DFSs [45, 75]. Notably, if the source is a directory, SwitchFS initiates an aggregation at the beginning of *rename* to apply all delayed updates.

Protocol intuitions. The intuition of SwitchFS's asynchronous metadata operation is to split a double-inode operation into two halves: the *local half*, which updates the target object's inode and persists the update to the parent directory in a change-log, and the *remote half*, which aggregates and applies the change-log entries to the parent directory's inode when it is read. This effectively defers the directory update until read, allowing operations to return early and to hide cross-server overhead.

To ensure that asynchronous operations do not lose updates, SwitchFS uses the on-switch dirty set to coordinate directory reads and writes. We show that metadata operations are serializable and preserve real-time order in §A.2.

Data structures on metadata servers. A metadata server maintains the following data structures.

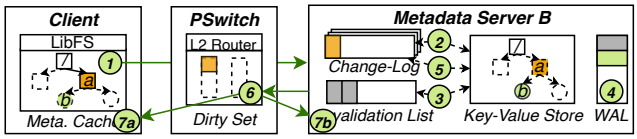


Fig. 4: Workflow of *mkdir, create* and *delete*.

- The *change-log* is a per-server, per-directory FIFO queue that records the committed but not-yet-applied asynchronous updates to a directory.
- The *invalidation list* is used for *lazily* invalidating stale client metadata caches, as in InfiniFS [45]. When a directory is removed, renamed, or changes permission, the server that hosts the directory's inode broadcasts a *message* to all servers, and the receivers append the directory's information to their invalidation lists. When contacting a server, a client consults the invalidation list and invalidates any stale cache entries accordingly.
- The *key-value store* stores inodes and directory entries.
- The *write-ahead-log* (WAL) is a persistent structure for crash recovery. It records the sequence of committed operations, and marks whether the corresponding asynchronous updates have been applied to remote servers.

The *change-log*, *invalidation list*, and *key-value store* are kept in memory for performance. §5.4.2 and §A.1 discuss how to recover them after crashes.

5.2.1 Workflow of *mkdir, create* and *delete*. These operations asynchronously update the parent directory's metadata. Fig. 4 illustrates an example of *create(/a/b)*. Other operations are handled similarly.

Path resolution. First, the client looks up each intermediate directory (i.e., */* and */a*) in its cache (①). On a cache miss, the client sends a *lookup* request to the owner server of the directory's inode and fills the cache with inode information. After the path resolution, the client sends the request to Server B, the owner server of */a/b*'s inode, which is determined by hashing */a/b*'s pid and name.

Locking and checking. Server B acquires a write lock on directory */a*'s change-log and a write lock on */a/b*'s inode in the key-value store (②). It then checks whether any path component of */a/b* appears in the invalidation list, and whether file */a/b* already exists (③). If invalid, Server B asks the client to invalidate stale cache entries and to retry the entire operation. If the checks pass, the operation is persistently logged in the WAL and commits(④).

Execution. Server B inserts the inode of file */a/b* into the key-value store and appends an entry to the change-log of directory */a* to record the *create(/a/b)* operation (⑤).

Dirty set update, reply, and unlocking. Server B sends to the switch a packet that contains the fingerprint of the parent directory */a*. Upon receiving the packet, the switch

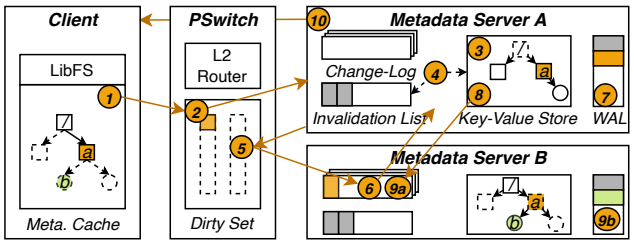


Fig. 5: Workflow of *statdir*.

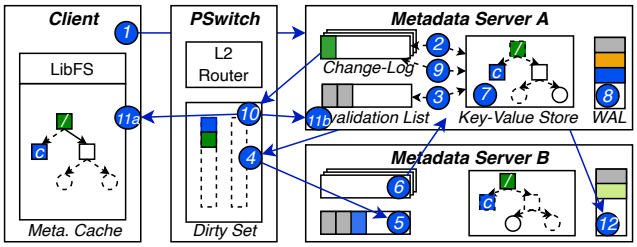


Fig. 6: Workflow of *rmdir*.

inserts */a*'s fingerprint into the dirty set (⑥). If insertion succeeds, the switch multicasts the packet to both the client and Server B, notifying the client of the operation's completion (⑦a) and signaling Server B to release the locks (⑦b). If the insertion fails (e.g., due to overflow), the operation falls back to synchronous metadata update: the switch forwards the packet to the server that owns the parent directory's inode, which then updates the parent directory synchronously.

Discussion. Notably, multiple servers may update the same directory and log their updates independently. These updates do not conflict because updates to directory attributes are commutative, and the next *statdir* or *readdir* operation on the directory will aggregate all these updates (§5.2.2 and §5.3).

5.2.2 Workflow of *statdir* and *readdir*. These operations read directory attributes and entry lists, triggering aggregation if the directory is in *scattered state*. Fig. 5 illustrates an example of *statdir(/a)*. *readdir* is handled similarly.

Path resolution. The client performs path resolution as in §5.2.1 (①), then sends the request packet to directory */a*'s owner server (i.e., Server A). This packet contains the fingerprint of */a*.

Dirty set query. When the switch forwards the packet, it queries */a*'s state and attaches the result to the packet (②).

Locking and checking. Upon receiving the packet, Server A first acquires a read lock on */a*'s inode (③) and checks cache validation and inode existence as in §5.2.1 (④). Then, the server examines the dirty set query result. If */a* is in normal state, the server directly returns the directory's inode to the client. Otherwise, Server A issues a metadata aggregation to renew the directory's inode before replying to the client.

Aggregation and reply. To issue an aggregation, Server A blocks *statdir* and *readdir* for all directories in */a*'s fingerprint group, and sends an aggregation request containing */a*'s fingerprint to the switch. When the switch receives the request, it removes */a*'s *fingerprint* from the dirty set and multicasts the request to all other servers (⑤). Upon receipt of the request, each other server acquires a read lock on */a*'s change-log, then sends all the change-log entries in */a*'s fingerprint group to Server A (⑥). When Server A receives these change-log entries, it logs each entry in the WAL (⑦) and updates */a*'s metadata in the key-value store accordingly (⑧). After receiving entries from all servers and applying them, Server A multicasts an acknowledgment to the other servers, releases local locks, and responds to the client (⑩). When other servers receive the acknowledgment, they unlock the change-logs (9a) and mark the change-log entries as "applied" in the WAL (9b). The entries in the change-logs are reclaimed in the background.

5.2.3 Workflow of *rmdir*. Fig. 6 illustrate an example of *rmdir(/c)*. The first few steps, i.e., path resolution (①), locking (②), validation, and inode existence checks (③) are the same as *create*. After these steps, Server A (i.e., the owner server of */c*) must collect the latest updates to */c* to determine whether the directory is empty, and prevent any stale client-side cache of */c* from being reused in future operations. To do so, Server A sends an aggregation request to the switch. The switch removes */a*'s fingerprint from the dirty set and multicasts the request to all other servers (④). Upon receiving the request, each other server (for example, Server B) inserts */c* into its invalidation list (⑤), so that clients' stale cache of */c* will be invalidated during clients' next operations. Then, Server B acquires a read lock on */c*'s change-log and sends the entries to Server A (⑥). Server A applies all received change-log entries and then checks the emptiness of */c* based on the aggregated metadata (⑦). If */c* is empty, Server A logs the *rmdir(/c)* operation in the WAL (⑧) and the operation commits. Otherwise, the *rmdir(/c)* fails with an *ENOTEMPTY* error. The remaining steps (⑨ ⑩, 11a, 11b) are similar to *create*. At the end, Server A sends a packet notifying other servers to release the locks and to mark the change-log entries they sent as "applied" in their WAL (12).

Discussion. SwitchFS prevents clients from accessing stale directory metadata through lazy invalidation. Specifically, *rmdir(/c)* appends */c* to the invalidation lists on all servers (⑤) before actually removing the directory (⑨). Any operation under */c* that checks path validity after the append fails the check and issues a *lookup(/c)* request. Since *rmdir(/c)* holds a write lock on */c*'s inode (②), the lookup waits until *rmdir(/c)* completes, observing the latest metadata.

5.3 Change-Log Compaction and Applying

Asynchronous metadata updates enable parallel and local logging of directory modifications, hiding directory update overhead. SwitchFS further amortizes the update overhead by compacting change-logs before applying them to directory inodes, alleviating the metadata contention bottleneck.

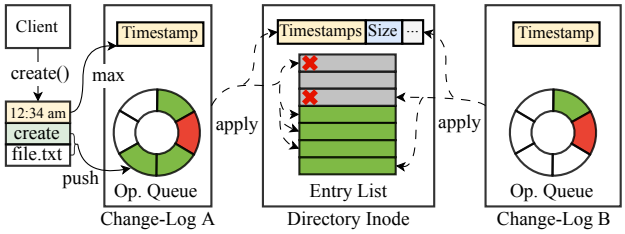


Fig. 7: Change-log compaction and application.

Change-log structure. In SwitchFS, a server maintains a local change-log for every *scattered* directory. A change-log entry represents a delayed directory update, containing the timestamp, operation type (*create*, *delete*, etc.), and filename, as depicted in Fig. 7.

Opportunities for change-log compaction. Updates to the same directory are conditionally commutative, allowing merging multiple updates into a single one before applying them, mitigating update contention [2, 7, 33, 69, 71, 75, 78]. Specifically, a directory update involves three types of actions:

(a) apply a delta to attributes such as *size*,
(b) overwrite timestamps such as *atime* and *mtime*, and
(c) insert or remove an entry in the entry list.

For type (a), applying the deltas in any order results in the same final state. For type (b), only the largest timestamp remains. For type (c), insertions and removals of different entries do not conflict and may be applied in any order, whereas repeated insertions and removals of the same name must be applied in their commit order (e.g., *mkdir(/a), rmdir(/a)*). In SwitchFS, this order is preserved by the change-log's FIFO structure, given that insertions and removals of the same entry are always logged by the same server.

Change-log compaction workflow. Therefore, SwitchFS proposes *change-log compaction* to decrease the cost of directory update. When directory updates are appended to a change-log, the change-log keeps the maximum timestamp of all entries, and stores the operation type and the filename. Upon receiving an aggregation request, the server transmits the relevant change-log to the directory's owner server. The owner server then traverses the operation queue, updating the directory entry list, and updates the inode's attributes (e.g., timestamps and size) atomically.

Change-log compaction mitigates directory update contention in two ways. First, asynchronous metadata updates allow all servers to consolidate updates to remote parent directories locally in their change-logs, enhancing inter-server parallelism. Second, the change-log compaction consolidates timestamps and size deltas beforehand to minimize the number of key-value store's *put()* operations.

Proactive aggregation. To prevent a prolonged aggregation stall of the first directory read after a sequence of directory updates, a server proactively pushes its change-log entries to the directory's owner server (1) when the accumulated entries are sufficient to fill a maximum transmission unit (MTU), or (2) when no new entries have been appended to the change-log within a configured interval. Upon receiving a change-log push, the directory's owner server starts a timer. If no new entry pushes arrive within a configured period, the owner server proactively initiates an aggregation. This proactive aggregation turns the directory back to normal state, allowing subsequent directory reads to complete without triggering an aggregation.

5.4 Fault Tolerance

5.4.1 Unreliable Network. SwitchFS protocol is based on UDP, which can experience packet loss and reordering. SwitchFS handles packet loss with timeout and resending. The reordering of in-flight packets causes no issue, as packets belonging to the same operation are sent and received one by one, while packets belonging to different operations are independent, and SwitchFS does not assume their order.

Resending causes packet duplication. Servers and the switch tolerate it with different mechanisms. A server detects and drops duplicate packets by examining the (*sender server, sequence number*) tuple attached to each packet by the sender server. The switch handles three types of dirty set operations encapsulated in packets: *fingerprint query*, *insert*, and *remove*. The switch executes duplicate queries and inserts without special treatment since they do not affect correctness. Specifically, queries have no side effects, and redundant inserts may trigger unnecessary aggregations, but do not compromise correctness. Only duplicate *remove*s can cause inconsistency, in the case where a duplicate *remove* request sent by an aggregation reaches the switch after the aggregation completes, removing fingerprints inserted by subsequent operations. To avoid this, each *remove* request has a sequence number (distinct from the packet's sequence number), which increments with every new request or resend. The switch records the highest *remove* sequence number received from each server and processes a *remove* request only if its sequence number exceeds all previously received ones from the sending server. After resending a *remove* request, the server waits for responses corresponding to the last resend. In this way, no duplicate *remove* requests take effect after the aggregation completes, ensuring consistency.

5.4.2 Crash Recovery.

Server failure. The servers maintain data structures in DRAM and use the write-ahead log (WAL) to recover lost progress after server failures, as production deployments (e.g., PanguFS and HDFS) typically do [45]. To recover from failures, the server first redoes operations in the WAL to rebuild its local key-value store and change-log entries that haven't been marked as "applied". Note that we don't need to rebuild change-log entries marked as "applied", as they have been persisted in the WAL of the directory inode's server. Then, the server proactively aggregates all directories it owns to ensure any interrupted aggregations it issued before the crash are executed to completion. Finally, the server recovers its invalidation list by cloning it from other servers. During recovery, the server does not serve normal requests.

Switch failure. After a switch failure, all state in the switch are lost. SwitchFS initializes an empty dirty set in the switch and notifies all servers to aggregate all directories. Once all aggregations are complete, all change-log entries have been sent to and applied by their directory inode's server. Thus, all directories in SwitchFS return to normal state, consistent with the empty dirty set. During recovery, all servers stop serving normal requests.

5.5 Discussion

Cluster reconfiguration. SwitchFS uses consistent hashing [25] to map inodes to servers. When servers are added or removed, a small portion of metadata needs to be migrated. SwitchFS performs migration in a stop-the-world manner: all metadata servers stop serving requests, aggregate all directories, and then migrate the metadata to the new server via two-phase commit. Since the hash function resides on the clients and servers, no changes are needed on the switch. We discuss the fault tolerance of cluster reconfiguration in §A.3.

Support of hard links. To support hard links, the file metadata in Tab. 3 needs to be split into two KV objects:
- the *reference*: (pid, name) → (server_id, file_id), and
- the *attributes*: file_id → (ref_count, size, timestamps, ...).

On file creation, both objects are created on the server determined by hashing *(pid, name)*. When a hard link is created, only a new reference is created at the hashing location, pointing to the original file's attributes, which may reside on a different server. The attributes' *ref_count* tracks the number of references, and the attributes are deleted when it drops to zero. For consistency and atomicity, both objects are protected by two-phase locking during metadata access, and two-phase commit coordinates updates across servers.

6 Data Plane Design

In this section, we introduce the format of SwitchFS packets (§6.1), the switch data plane layout (§6.2), and the dirty set design (§6.3). We also discuss scaling to multiple racks (§6.4) and the adaptability issues (§6.5).

6.1 SwitchFS Packet Format

SwitchFS employs UDP as the transport layer protocol for lightweight networking. As illustrated in Fig. 9, the UDP payload begins with an optional header that encapsulates a dirty-set operation, which can be parsed by the switch.

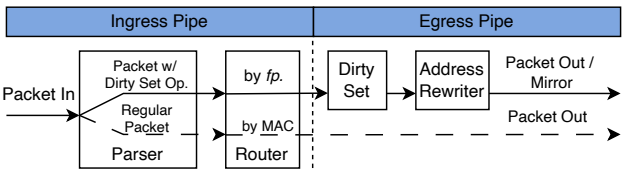


Fig. 8: Logical view of SwitchFS switch data plane.

Following this header, the payload contains a DFS request or response for servers to process. SwitchFS reserves two UDP ports for SwitchFS packets with and without the dirty-set operation header so that the switch can distinguish them.

6.2 Data Plane Layout

Figure 8 presents the layout of the switch data plane, which consists of the following components.

Parser. The *parser* parses the packet headers to extract the fields for subsequent processing.

Router. The *Router* sets the egress port field in the switch's metadata for (a) regular packets, by destination MAC address, and (b) packets with dirty-set operations, by *fingerprint* prefix. The switch forwards the packets accordingly.

Dirty set. The *dirty set* is implemented as a multi-slot hash table that supports three kinds of operations: (1) *insert* a fingerprint into the set; (2) *query* whether a fingerprint exists in the set; (3) *remove* a fingerprint from the set. When a packet arrives, the dirty set executes the operation specified in the *OP* field of the header and writes the result to the *RET* field. *SEQ* is a sequence number for recognizing duplicate *remove* requests, as described in §5.4.1.

Since a switch may have multiple pipes and pipes do not share state, we place the dirty set in the egress pipes in a shared-nothing manner. Each pipe is responsible for fingerprints with different prefixes, and packets are routed accordingly. If a packet accesses a fingerprint managed by a pipe different from its destination port, the switch mirrors it to the destination pipe, as in prior studies [24, 79].

Address rewriter. If a dirty-set insertion fails due to overflow, the switch redirects the packet using the *alternative MAC address* field in the packet header for fallback handling. The *address rewriter* overwrites the L2 destination field with this alternative address, and the switch forwards accordingly.

6.3 In-Network Dirty Set

The on-switch dirty set maintains the set of directories in scattered states. Since its performance and capacity are critical to SwitchFS, we have the following design goals: (1) support line-rate *query*, *insert* and *remove* of fingerprint, and (2) maximize the utilization of scarce on-chip memory.

Structure. Due to programming constraints that prevent the implementation of complex indexing structures, SwitchFS organizes 32-bit registers in a manner similar to a set-associative

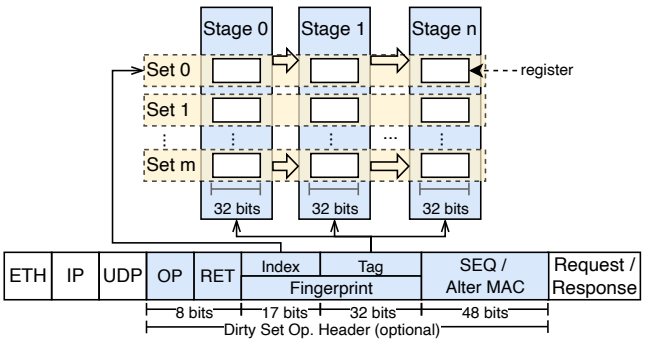


Fig. 9: Data structure of the dirty set.

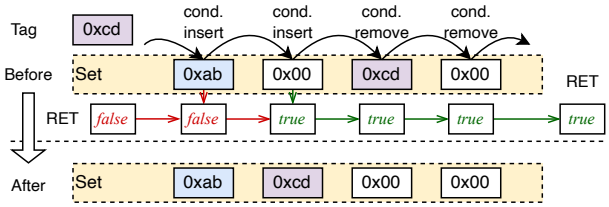


Fig. 10: An example of the dirty-set *insert* operation.

cache to store fingerprints. As Fig. 9 shows, the switch data plane comprises multiple stages, each containing an array of registers. Registers at the same position across stages are horizontally grouped into a set. In our switch configuration, there are ten stages, and each stage allocates 131,072 (2^{17}) registers. This setup allows the switch to store up to 1,310,720 fingerprints. We use the upper 17 bits of the fingerprint as the set *index*, while the remaining 32 bits serve as the *tag* to uniquely identify each fingerprint.

Dirty-set operations. The dirty set supports three operations: *insert*, *query*, and *remove*. Each operation performs a sequence of *register actions* on the registers indexed by the header's index field. We define three register actions: (a) *register query* compares the register's value with *tag* and returns the result; (b) *conditional insert* returns whether the the register's value equals zero or *tag* and writes *tag* into the register if the old value is zero; (c) *conditional remove* writes zero into the register if the old value matches *tag*.

For dirty-set *query*, all stages perform register query sequentially, and it returns *true* if any of the stages return *true*. For dirty-set *remove*, all stages perform conditional remove sequentially. For dirty-set *insert*, stages perform conditional insert one by one until one of them returns *true*, and the following stages perform *conditional remove* to ensure no duplicate *tags* remain in the set. Fig. 10 illustrates an example of dirty set *insert*.

Properties. The switch architecture provides two properties [73]. (a) *Atomicity.* Operations in the same stage are atomic. (b) *Ordered execution.* For any two packets P_A, P_B and two ordered stages S_A, S_B, if P_A reaches S_A before P_B, then P_A reaches S_B before P_B.

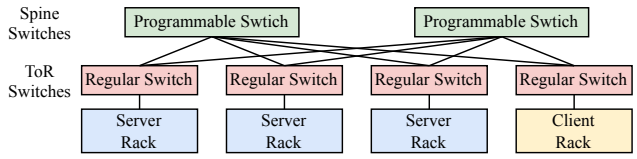


Fig. 11: Topology of the multi-rack deployment.

Based on these properties, our design ensures that the dirty-set operations are (a) *idempotent*, since successive repeated operations have the same effects as a single one, and (b) *linearizable*, since operations on the same *fingerprint* are serialized by the pipeline.

6.4 Scale to Multiple Racks

For single-rack deployment, where all metadata servers reside in the same rack, the dirty set is placed in the top-of-rack (ToR) programmable switch, enabling it to monitor all rack traffic. For multi-rack deployments, SwitchFS uses a leaf-spine topology as shown in Fig. 11. Since ToR switches no longer have a global view, programmable switches are deployed at the spine layer. To scale further, SwitchFS range-partitions the state management of directories across switches by fingerprint, and directs packets involving each subset of directories to the corresponding switch. A previous study [84] has shown that the overhead of the extra network hops between switches is negligible, since it is orders of magnitude smaller than that of operation processing. This approach adapts to skewed workloads since (a) the distribution of directories over switches is uniform due to random fingerprints, and (b) each switch can process all the traffic in the metadata cluster without becoming a bottleneck (§7.3.3).

6.5 Discussion

Adaptability of the programmable switch. SwitchFS's on-switch program comprises 847 lines of P4 code and 303 lines of C code. It consumes 1,310,720 32-bit on-chip registers (5 MiB in total). Initializing it on a Tofino switch [21] takes less than ten seconds.

Limitations. Given the limited on-chip memory of the Tofino switch [21], colocating SwitchFS with state-heavy network functions (e.g., in-network caching [24, 44]) is challenging. However, it can coexist with lightweight functions such as flow control and packet filtering. The programmable switch also constitutes a potential single point of failure, which we leave for future work.

7 Evaluation

7.1 Experimental Setup

Hardware configuration. We conduct experiments with eight server machines and three client machines. Tab. 4 shows their specifications. All machines are connected via a Wedge100BF-32X programmable switch equipped with a Tofino ASIC [21]. To scale up the evaluation to 16 servers,

Server Cluster	CPU	2 × Intel Xeon Gold 5317 3.00GHz, 12 cores
	Memory	16 × DDR4 2933MHz 16GB
	Storage	Intel Optane Persistent Memory
	Network	2 × ConnectX-5 Single-Port 100GbE
Client Cluster	CPU	2 × Intel Xeon E5-2650 v4 2.20GHz, 12 cores
	Memory	16 × DDR4 2933MHz 16GB
	Network	2 × ConnectX-4 Single-Port 100GbE

Tab. 4: Hardware specifications of clusters.

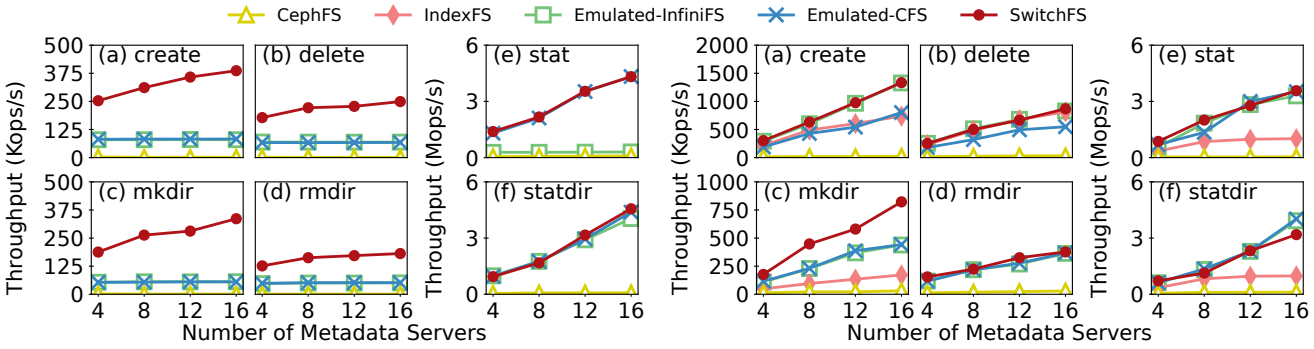
we deploy a metadata server on each socket of the server nodes, utilizing the NUMA-local cores, DRAM, and NIC. We do not evaluate with multiple programmable switches due to hardware limitations.

Baseline systems. We compare SwitchFS with CephFS v12.2.13 [76], IndexFS [59], InfiniFS [45], and CFS [75]. CephFS is a widely used commercial DFS, while the others are state-of-the-art DFSs optimized for metadata performance. Since InfiniFS and CFS are not publicly available, we implemented them from scratch. We emulate InfiniFS by carefully implementing it according to its paper [45] and emulate CFS by replacing InfiniFS's P/C grouping approach (i.e., per-directory hashing) with CFS's P/C separation approach (i.e., per-file hashing). Notably, Emulated-InfiniFS (E-InfiniFS), Emulated-CFS (E-CFS), and SwitchFS share the same storage and networking framework, ensuring a fair comparison. We do not compare with SingularFS [16] and DeltaFS [83], as SingularFS focuses on enhancing per-server performance with novel hardware (i.e., persistent memory and RDMA), which is orthogonal to our work, and DeltaFS is designed for HPC jobs, whose semantics differ from POSIX DFSs.

Configuration details. E-InfiniFS, E-CFS, and SwitchFS use RocksDB in asynchronous write mode for metadata storage, and employ a coroutine-based, non-blocking RPC engine on DPDK 21.11.2 [53] for networking. Unless specified otherwise, each metadata server uses four cores, and clients use LibFS to communicate with servers. For CephFS, we deploy multiple active metadata server daemons (MDSs) colocated with object storage daemons (OSDs). SwitchFS enables proactive change-log aggregation in all experiments, so aggregation overhead is included in the results. No dirty-set overflow occurs in the evaluation.

7.2 Overall Performance

7.2.1 Throughput. We first evaluate how the peak throughput of individual operations scales with the number of servers. We gradually increase the number of concurrent requests issued by clients (up to 512) until the throughput no longer increases. We conduct experiments on two different access patterns, *a single large directory* and *multiple directories*, which reflect DFSs' load balance and operation overhead, respectively. IndexFS's throughput on a single large directory is not included since it consistently crashes with errors.



(a) Single large directory. **(b)** Multiple directories.

Fig. 12: Throughput of metadata operations.

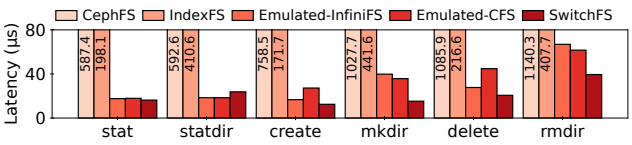


Fig. 13: Metadata operation latency.

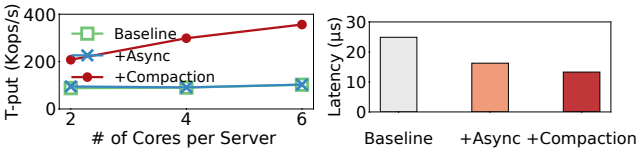


Fig. 14: Contribution breakdown. Performance of file create in a single directory with different techniques enabled.

A single very large directory. In this evaluation, clients perform operations on 10 million files in a single large directory and randomly select files to access. Fig. 12(a) shows the operation throughput as the metadata server increases. We make the following observations.

1) SwitchFS's throughput scales well for double-inode operations including *create*, *delete*, *mkdir* and *rmdir*, as server number increases. The reason is that, (a) SwitchFS's fine-grained partitioning approach distributes loads across servers evenly; (b) the asynchronous metadata update protocol allows double-inode operations to access only one server, avoiding cross-server coordination and hiding directory contention; (c) change-log compaction merges conflicting directory updates, reconciling the directory contention. Note that SwitchFS's *rmdir* throughput is lower than that of *mkdir* since SwitchFS uses multicast to notify the invalidation of the directory during *rmdir*.

2) Despite employing fine-grained partitioning, E-CFS exhibits minimal scalability for *create*, *delete*, *mkdir*, and *rmdir*. This arises because these operations are serialized per directory, leading to severe contention.

3) Both SwitchFS and E-CFS scale linearly for *stat* operations, benefiting from effective load balancing via fine-grained partitioning. In contrast, E-InfiniFS suffers from significant load imbalance due to assigning all files within a large directory to a single server. For *statdir*, all filesystems except CephFS scale linearly, as they partition directories in a balanced manner. SwitchFS's dirty set checking introduces negligible additional overhead to *statdir*.

4) CephFS's throughput is low (below 100 Kops/s) for its heavy software stack and inflexible subtree partitioning.

Multiple directories. In this evaluation, clients randomly access 10 million files uniformly distributed across 1024 directories. Fig. 12(a) shows the operation throughput as the number of metadata servers increases. We make the following observations.

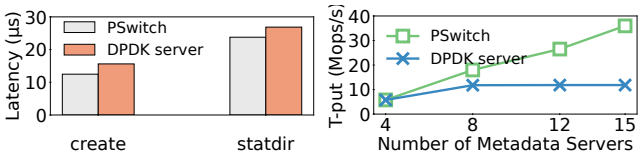
1) In this setting, operations have little contention and DFSs achieve optimal performance. SwitchFS performs the best for all operations, indicating that asynchronous metadata updates have negligible overhead on the common path.

2) For *create* and *delete*, SwitchFS and E-InfiniFS perform similarly and outperform E-CFS. E-InfiniFS performs well as its parent-children grouping enables updating the file and parent directory metadata on the same server. SwitchFS achieves comparable performance since asynchronous metadata updating hides the overhead for updating directory metadata. In contrast, E-CFS performs worse since it separates file and parent directory metadata, requiring cross-server transactions for updates.

3) For *mkdir*, SwitchFS outperforms IndexFS, E-InfiniFS, and E-CFS since asynchronous updates hide cross-server coordination overhead for updating two directories' metadata on different servers, while the other DFSs need to use distributed transactions. For *rmdir*, SwitchFS performs similarly to E-InfiniFS and E-CFS, as the multicast overhead offsets the gain of asynchronous updates. IndexFS's implementation of *rmdir* is incomplete, so its results are omitted.

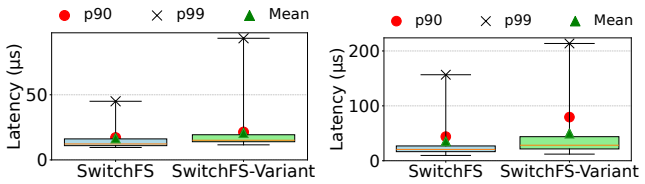
7.2.2 Latency. We evaluate the latency of operations by using a single client to issue requests one by one. DFSs have eight servers. Fig. 13 shows the average latency.

1) SwitchFS significantly reduces the latency of *create*, *delete*, *mkdir*, and *rmdir* compared to other DFSs, because



(a) Latency of *create*. **(b)** Throughput of *statdir*.

Fig. 15: Tracking directory states in a dedicated server.



(a) Under medium 50 Kops/s load. **(b)** Under heavy 120 Kops/s load.

Fig. 16: Latency of *create* operations when directory states are tracked on owner servers. The top and bottom edges of each box represent p75 and p25, respectively, and the line in the box represents the median.

its asynchronous metadata updates hide the overhead for updating the parent directory.

2) SwitchFS's *statdir* latency is 28.6% higher than E-InfiniFS and E-CFS due to additional checks introduced by asynchronous metadata update for correctness.

7.3 Contribution Breakdown

7.3.1 Contribution of Techniques. In this evaluation, we analyze how each design feature contributes to SwitchFS's throughput and latency. Clients create 10 million files in a single directory, and the DFS has eight servers. When evaluating throughput, we further vary the number of cores per server to demonstrate SwitchFS's intra-server parallelism. Fig. 14 shows the result.

Baseline is the framework of SwitchFS, employing fine-grained partitioning and synchronous operations. It suffers from high latency due to cross-server coordination and low throughput caused by parent-directory contention.

+Async adopts asynchronous metadata updates without change-log compaction. Compared to *Baseline*, the average latency is reduced by 34.7%, while the throughput remains unchanged. This is because asynchronous metadata updates hide the latency of updating the parent directory, but do not reduce the updating overhead. During aggregation, updates to the parent directory's metadata are serialized by the key-value store due to contention, resulting in poor parallelism.

+Compaction is the complete SwitchFS design with both asynchronous metadata updates and change-log compaction optimization. Compared to previous settings, the throughput is improved by up to 2.48× and scales as the number of cores per server increases. This is because the change-log compaction merges multiple updates to the directories' attributes (e.g., timestamps) into a single one, and the directory entries are inserted or deleted in parallel. Compared to *+Async*, the average latency is reduced by 18.2%, and the p99 latency drops from 173 μs to 22 μs.

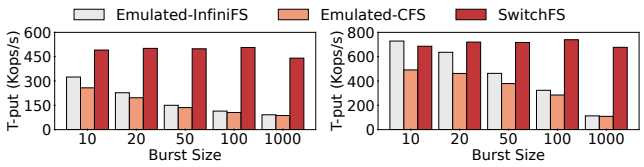
7.3.2 Impact of Dirty-Set Overflow. If insertion into the programmable switch's dirty set fails due to overflow, the operation falls back to server-side synchronous updates (§5.2.1). We force dirty set insertion to always fail and evaluate the performance of *create* using the setup described in §7.3.1. Compared to scenarios where insertions succeed, throughput drops by 69.7% and average latency rises by 0.85×, closely matching the performance of *Baseline* in §7.3.1.

7.3.3 Contribution of the Programmable Switch. The asynchronous update protocol in SwitchFS is not tightly coupled to the programmable switch. This evaluation examines the trade-off of using a programmable switch versus two alternatives: (a) using a dedicated server to maintain the dirty set, and (b) assigning each directory's owner server to track its own dirty state.

Tracking directory state with a dedicated server. Tracking directory state with a dedicated server, rather than a programmable switch, has two disadvantages. First, as shown in Fig. 15(a), operations involving the dirty set incur an additional RTT (~3μs), raising the average latency of *create* and *statdir* by 24.1% and 13.1%, respectively. Second, the server cannot match the throughput of a programmable switch. In Fig. 15(b), we measure throughput by performing *statdir* on 1 million directories, with each server using 12 cores and the dedicated server leveraging DPDK [53] for packet processing. The dedicated server hits a throughput ceiling of 11 Mops/s, whereas the programmable switch scales linearly with the number of metadata servers. Theoretically, a programmable switch can process up to 4.8 Bops/s [22], far exceeding the demand of a metadata cluster. Consequently, a single programmable switch suffices to serve the cluster, while dedicated servers would need to partition fingerprints across multiple machines to meet throughput demands.

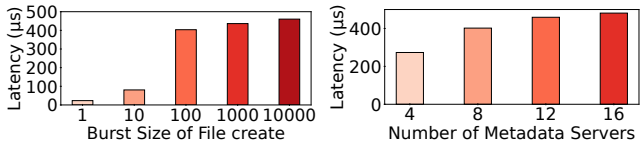
Tracking directory state with the owner server. Maintaining directory state on the owner server instead of a programmable switch reduces the peak throughput of double-inode operations such as *mkdir* and *create* by ~ 10%, as it doubles the number of packets processed per operation, consuming valuable CPU resources. Since metadata performance is CPU-bound, this lowers overall throughput.

Moreover, tracking on owner servers substantially increases operation latency. Fig. 16 shows the latency distribution of *create* under medium and heavy load. Compared with the original SwitchFS, this variant increases median, p90, and p99 latency by 23.2%, 24.9%, and 107.4% under medium load, and by 36.9%, 80.8%, and 36.4% under heavy load. The extra server on the critical path of directory updates introduces additional queuing and head-of-line blocking, amplifying latency — especially under high load.



(a) 32 in-flight requests. **(b)** 256 in-flight requests.

Fig. 17: Throughput of *create* in bursts.



(a) Latency w.r.t burst size on eight servers. **(b)** Latency w.r.t. server number with 100 preceding *create*s.

Fig. 18: Latency of *statdir* after successive *create*.

7.4 Operation Burst Performance

In this evaluation, we evaluate how SwitchFS handles temporal load imbalance; that is, a group of spatially related operations performed by applications within a short time period. We model temporal load imbalance as operation bursts, defined as a group of successive file *create*s in the same directory. Successive operation bursts create files in different directories. SwitchFS has eight servers, and clients issue 32 or 256 in-flight requests to stress the servers. We observe the throughput changes while varying the burst size.

As shown in Fig. 17, both E-InfiniFS and E-CFS are sensitive to operation bursts. With 32 in-flight requests (Fig. 17(a)), compared to burst size 10, the throughput of E-InfiniFS and E-CFS drops by 53.7% and 47.1% at burst size 50, and drops by 71.9% and 66.1% at burst size 1000. In contrast, SwitchFS exhibits stable performance as the burst size increases, because it buffers bursts in the change-log and applies them later. Fig. 17(b) shows results with 256 in-flight requests, which imposes greater pressure on the servers. Even in this scenario, SwitchFS maintains stable performance. These experiments demonstrate SwitchFS's ability to tolerate temporal load imbalance and to maintain stable throughput.

7.5 Directory Aggregation Overhead

Reading a scattered directory in SwitchFS requires metadata aggregation. To evaluate its overhead, we measure the latency of the *statdir* operation after a sequence of file *create* in the directory. During the *statdir* operation, change-log entries generated by previous *create* operations are aggregated and applied to the directory. We vary both the number of servers and the number of preceding *create* operations.

Fig. 18(a) shows how the latency of *statdir* increases with a growing number of preceding *create*s, eventually converging at approximately 500 μs. The reason is that as the number of preceding *create*s increases, more change-log entries must be processed during aggregation. However, the number of entries per server remains bounded (to 29 in our implementation) since the servers proactively push local change-log entries to the inode once they can fill an MTU.

Fig. 18(b) demonstrates that the average *statdir* latency increases with the number of servers. As it increases, more entries can be kept in the change-logs before proactive aggregation is triggered, resulting in more entries being aggregated during *statdir*.

Workload	Operation Ratio
Data Center Services [45]	52.6% open/close, 12.4% stat, 9.58% create, 11.9% delete, 9.3% file rename, 0.1% chmod, 3.9% readdir, 0.2% statdir
CNN Training	42.8% open/close, 21.4% stat, 14.2% read, 7.1% write, 7.1% create, 7.1% delete, 0.1% mkdir, 0.1% rmdir, 0.1% statdir, 0.1% readdir
Thumbnail	43.9% open/close, 21.9% stat, 12.2% read, 10.9% write, 10.9% create, 0.1% mkdir, 0.1% statdir, 0.1% readdir

Tab. 5: Metadata operation ratios in real-world traces.

7.6 End-to-End Performance

We evaluate the end-to-end performance under real-world workloads, with data access included, using a synthetic workload based on realistic operation ratios and two real-world traces. This experiment uses eight metadata servers and eight data nodes, and the client issues 256 in-flight requests. No dirty-set overflow occurs in this evaluation. Tab. 5 summarizes the operation ratios for each workload.

The synthetic workload is derived from the operation ratios of traces in Alibaba's deployed PanguFS, which serves various data center services including data processing, object storage, and block storage [45]. Based on these ratios, we perform 10 million operations in 1024 directories, with 80% of the operations in 20% of the directories to simulate workload skew. Data access is omitted from the synthetic workload because the corresponding operation ratio is unavailable.

The CV Training and Thumbnail workloads are representative many-small-file workloads. The CV Training workload is traced from training ALEXNET model [27, 28] on the ImageNet dataset [20], which contains 1.28 million files (mostly under 256KB) grouped into 1000 directories. This trace captures the entire lifecycle of the dataset, including download, access, and removal. The Thumbnail workload records the process of accessing 1 million images (mostly under 256KB) and creating thumbnails. We replay the trace with data access enabled and disabled.

As Fig. 19 shows, SwitchFS's improvement in metadata throughput translates into considerable gains in end-to-end throughput. Compared to CephFS, E-InfiniFS, and E-CFS, SwitchFS delivers speedups of up to 76.3×, 2.1×, and 0.7× for metadata throughput, and up to 21.1×, 1.1×, and 0.3× for end-to-end performance.

7.7 Crash Recovery Time

We evaluate SwitchFS's recovery time after a server failure and a switch failure. We simulate each type of failure after creating 10 million files in 100 thousand directories. SwitchFS uses eight servers. After a server failure, it takes the crashed

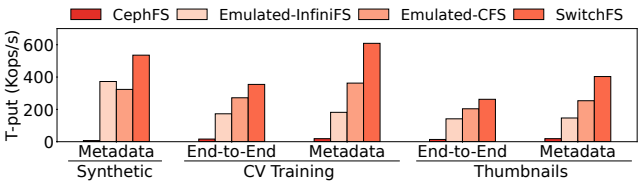


Fig. 19: Throughput of end-to-end workloads.

server 5.77 seconds to recover ~1.25 million inodes to its key-value store and ~1.25 million not-yet-applied change-log entries. After a switch failure, it takes 3.82 seconds for all the servers to flush not-yet-applied change-log entries to restore SwitchFS into a consistent state after the switch reboots. Note that the recovery time is proportional to the number of operations to be recovered, and could be substantially reduced through the use of checkpointing. During the recovery process, the end-to-end throughput is zero.

8 Related Work

Metadata partitioning. Early DFSs such as GFS [13], HDFS [66], Farsite [11], and QFS [50] manage all metadata on a single metadata server, resulting in poor scalability. More recent DFSs partition the namespace across multiple metadata servers to enable scalable metadata management. Some DFSs partition the directory tree at subtree granularity, such as AFS [19] and CephFS [76]. While subtree partitioning preserves metadata locality, it is prone to load imbalance [64, 74]. Some other DFSs use per-directory partitioning, such as BeeGFS [15], IndexFS [52, 57, 59], HopsFS [47], Tectonic [51], and InfiniFS [45]. Since they still group files' inodes with parent directories, they also suffer load imbalance when operations are intensively performed in a few directories. CFS [75] adopts per-file partitioning, which achieves good load balance but incurs cross-server coordination overhead for double-inode operations.

Previous DFSs have to trade off between load balance and operation overhead when choosing partitioning granularity. In contrast, SwitchFS achieves good load balance by partitioning at per-file granularity while hiding cross-server coordination overhead through asynchronous metadata updates. In this way, SwitchFS achieves the best of both worlds.

Directory contention mitigation. Prior DFSs explored how to mitigate directory contention. LocoFS [38] relaxes POSIX semantics to avoid updating directory metadata for double-inode operations. DeltaFS [83] does not provide a globally shared namespace but asks clients to explicitly manage synchronization scope, reducing the contention domain. CFS [75] requires cross-server coordination for double-inode operations while using ordered updates and database primitives to prune the critical section of directory updates. SingularFS [16] stores metadata on non-volatile main memory (NVMM) on a single server and updates directories with atomic instructions to reduce contention. SwitchFS provides a POSIX-compliant global namespace and does not rely on NVMM. SwitchFS's change-log compaction merges concurrent directory updates, eliminating the single-point serialization for directory updates, and can be combined with previous techniques.

In-network acceleration. There are previous studies that explored in-network coordination in distributed systems, including key-value cache [12, 23, 24, 31, 37, 39, 42, 44, 65, 67, 85], in-network computation [18, 29, 40, 43, 62, 63], distributed memory [30, 73], distributed locks [79, 80], consensus and concurrency control [8, 9, 26, 35, 36, 68, 84], and scheduling [56, 70]. SwitchFS's in-network structure is similar to that of Harmonia [84]. However, Harmonia targets in-network conflict detection for fast key-value store replication, while SwitchFS adopts in-network state tracking for zero-cost asynchronous DFS operations. To our knowledge, SwitchFS is the first to introduce in-network optimization to distributed filesystems.

Asynchronous update. There are previous studies [41, 48, 83] that adopted asynchronous updates for data and metadata, and SwitchFS is different from them. XSYNCFS [48] and POSIX-data-write [41] adopt asynchronous data updates, but their approaches are not applicable for metadata, as they allow update loss after crashes, whereas POSIX semantics forbid any loss of metadata updates. In contrast, SwitchFS's asynchronous protocol is persistent and consistent. DeltaFS [83] explored asynchronous metadata synchronization between clients. However, it uses explicit APIs to "get" and "publish" metadata updates, leaving the synchronization burden to users. In contrast, by introducing careful protocol design and in-network coordination, SwitchFS's asynchronous metadata operations are transparent to users and POSIX-compatible.

9 Conclusion

This paper proposes SwitchFS, a distributed filesystem with asynchronous metadata update to resolve the trade-off between operation efficiency and load balancing. SwitchFS leverages a programmable switch to track directory states in the network, and adopts change-log compaction to alleviate contention over directory metadata. Evaluation shows that SwitchFS outperforms state-of-the-art systems.

Acknowledgements

We sincerely thank our shepherd, Marc Shapiro, and the anonymous reviewers for their constructive comments and insightful suggestions. This work is supported in part by the National Key R&D Program of China (2024YFB4506200), the National Natural Science Foundation of China (No. 62132014), the Fundamental Research Funds for the Central Universities, Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM 113), and Huawei Technologies. The corresponding author is Mingkai Dong (mingkaidong@sjtu.edu.cn).

References

[1] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. 2012. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*. IEEE Computer Society, Chicago, Illinois, USA, 125–132. doi:10.1109/UCC.2012.27

[2] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal Urso. 2012. File system on CRDT. arXiv:1207.5990 http://arxiv.org/abs/1207.5990

[3] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Vancouver, BC, Canada, 47–60.

[4] Nikolaj Bjørner. 2007. *Models and Software Model Checking of a Distributed File Replication System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23. doi:10.1007/978-3-540-75221-9_1

[5] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. 2009. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*. IEEE Computer Society, Rome, Italy, 1–11. doi:10.1109/IPDPS.2009.5161029

[6] Cisco. 2020. *Cisco Nexus 34180YC and 3464C Programmable Switches Data Sheet*.

[7] Austin T. Clements, M. Frans Kaashoek, Eddie Kohler, Robert T. Morris, and Nickolai Zeldovich. 2017. The scalable commutativity rule: designing scalable software for multicore processors. *Commun. ACM* 60, 8 (July 2017), 83–90. doi:10.1145/3068914

[8] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1726–1738. doi:10.1109/TNET.2020.2992106

[9] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. Association for Computing Machinery, Santa Clara, California, Article 5, 7 pages. doi:10.1145/2774993.2774999

[10] Shobhit Dayal. 2008. Characterizing HEC storage systems at rest. *Parallel Data Lab, CMU* (2008).

[11] John R. Douceur and Jon Howell. 2006. Distributed directory service in the Farsite file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Seattle, Washington, 321–334.

[12] Roy Friedman, Or Goaz, and Dor Hovav. 2023. PKache: A Generic Framework for Data Plane Caching. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*. Association for Computing Machinery, Tallinn, Estonia, 1268–1276. doi:10.1145/3555776.3590826

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

[14] Gluster. 2019. Storage for Your Cloud. https://www.gluster.org/. Accessed: April 18, 2023.

[15] ThinkParQ GmbH. 2023. BeeGFS Documentation 7.4.2. https://doc.beegfs.io/latest/index.html. Accessed: November 17, 2023.

[16] 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

[17] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Santa Clara, CA, 199–212.

[18] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, Vancouver, BC, Canada, 33–47. doi:10.1145/3575693.3575708

[19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. 1987. Scale and Performance in a Distributed File System. *SIGOPS Oper. Syst. Rev.* 21, 5 (nov 1987), 1–2. doi:10.1145/37499.37500

[20] ImageNet. 2020. ImageNet Object Localization Challenge. https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data. Accessed: November 19, 2023.

[21] Intel. 2023. Intel Tofino Series. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html. Accessed: September 5, 2023.

[22] Intel. 2023. Intel® Tofino 6.4 Tbps, 4 pipelines. https://www.intel.com/content/www/us/en/products/sku/218643/intel-tofino-6-4-tbps-4-pipelines/specifications.html. Accessed: November 19, 2023.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 121–136. doi:10.1145/3132747.3132764

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. Association for Computing Machinery, El Paso, Texas, USA, 654–663. doi:10.1145/258533.258660

[26] Gyuyeong Kim and Wonjun Lee. 2022. In-Network Leaderless Replication for Distributed Data Stores. *Proc. VLDB Endow.* 15, 7 (mar 2022), 1337–1349. doi:10.14778/3523210.3523213

[27] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014). arXiv:1404.5997 http://arxiv.org/abs/1404.5997

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6, 84–90. doi:10.1145/3065386

[29] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Virtual Event, USA, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao

[30] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, Virtual Event, Germany, 488–504. doi:10.1145/3477132.3483561

[31] Jason Lei and Vishal Shrivastav. 2024. Seer: Enabling Future-Aware Online Caching in Networked Systems. In *21st USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 24). USENIX Association, Santa Clara, CA, 635–649. https://www.usenix.org/conference/nsdi24/presentation/lei

[32] Paul Lensing, Dirk Meister, and André Brinkmann. 2010. hashFS: Applying Hashing to Optimize File Systems for Small File Reads. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*. IEEE Computer Society, Los Alamitos, CA, USA, 33–42. doi:10.1109/SNAPI.2010.12

[33] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

[34] Jiahao Li, Biao Cao, Jielong Jian, Cheng Li, Sen Han, Yiduo Wang, Yufei Wu, Kang Chen, Zhihui Yin, Qiushi Chen, Jiwei Xiong, Jie Zhao, Fengyuan Liu, Yan Xing, Liguo Duan, Miao Yu, Ran Zheng, Feng Wu, and Xianjun Meng. 2025. Mantle: Efficient Hierarchical Metadata Management for Cloud Object Storage Services. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, Seoul, South Korea.

[35] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 104–120. doi:10.1145/3132747.3132751

[36] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[37] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, Virtual Event, USA, Article 22, 20 pages.

[38] 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 (SC '17)*. Association for Computing Machinery, Denver, Colorado, Article 4, 12 pages. doi:10.1145/3126908.3126928

[39] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 31–44. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-xiaozhou

[40] Zhaoyi Li, Jiawei Huang, Yijun Li, Aikun Xu, Shengwen Zhou, Jingling Liu, and Jianxin Wang. 2023. A2TP: Aggregator-aware In-network Aggregation for Multi-tenant Learning. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, Rome, Italy, 639–653. doi:10.1145/3552326.3587436

[41] Linux. 2025. *openat(2) - Linux manual page 6.15*. Linux man-pages project. https://man7.org/linux/man-pages/man2/openat.2.html Accessed: 2025-9-12.

[42] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, Xi'an, China, 795–809. doi:10.1145/3037697.3037731

[43] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray C. C. Cheung, and Jianfei He. 2023. In-Network Aggregation with Transport Transparency for Distributed Training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, Vancouver, BC, Canada, 376–391. doi:10.1145/3582016.3582037

[44] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 143–157. https://www.usenix.org/conference/fast19/presentation/liu

[45] 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

[46] Peter Macko and Jason Hennessey. 2022. Survey of Distributed File System Design Choices. *ACM Trans. Storage* 18, 1, Article 4 (mar 2022), 34 pages. doi:10.1145/3465405

[47] 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

[48] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2008. Rethink the sync. *ACM Trans. Comput. Syst.* 26, 3, Article 6 (Sept. 2008), 26 pages. doi:10.1145/1394441.1394442

[49] OpenSwitch. 2019. Cavium XPliant. https://www.openswitch.net/cavium/. Accessed: September 19, 2023.

[50] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The quantcast file system. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1092–1101. doi:10.14778/2536222.2536234

[51] 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, Virtual Event, USA, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[52] 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 (FAST'11)*. USENIX Association, San Jose, California, 13.

[53] DPDK Project. 2023. The Open Source Data Plane Development Kit Accelerating Network Performance. https://www.dpdk.org/. Accessed: November 19, 2023.

[54] Yingjin Qian, Wen Cheng, Lingfang Zeng, Xi Li, Marc-André Vef, Andreas Dilger, Siyao Lai, Shuichi Ihara, Yong Fan, and André Brinkmann. 2023. Xfast: Extreme File Attribute Stat Acceleration for Lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, Denver, CO, USA, Article 96, 12 pages. doi:10.1145/3581784.3607080

[55] Yingjin Qian, Wen Cheng, Lingfang Zeng, Marc-André Vef, Oleg Drokin, Andreas Dilger, Shuichi Ihara, Wusheng Zhang, Yang Wang, and André Brinkmann. 2022. MetaWBC: POSIX-compliant metadata write-back caching for distributed file systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE Press, Dallas, Texas, Article 56, 20 pages.

[56] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. 2023. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, Koblenz, Germany, 182–199. doi:10.1145/3600006.3613170

[57] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 145–156. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren

[58] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. 2013. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 853–864. doi:10.14778/2536206.2536213

[59] 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 (SC '14)*. IEEE Press, New Orleans, Louisana, 237–248. doi:10.1109/SC.2014.25

[60] RocksDB. 2021. A persistent key-value store for fast storage environments. http://rocksdb.org. Accessed: April 18, 2023.

[61] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, San Diego, California, 4.

[62] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets '17)*. Association for Computing Machinery, Palo Alto, CA, USA, 150–156. doi:10.1145/3152434.3152461

[63] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[64] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: a programmable metadata load balancer for the ceph file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, Austin, Texas, Article 21, 12 pages. doi:10.1145/2807591.2807607

[65] Siyuan Sheng, Huancheng Puyang, Qun Huang, Lu Tang, and Patrick P. C. Lee. 2023. FarReach: Write-back Caching in Programmable Switches. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 571–584. https://www.usenix.org/conference/atc23/presentation/sheng

[66] 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)*. IEEE, Incline Village, NV, USA, 1–10. doi:10.1109/MSST.2010.5496972

[67] Y. Su, D. Feng, Y. Hua, Z. Shi, and T. Zhu. 2018. NetRS: Cutting Response Latency in Distributed Key-Value Stores with In-Network Replica Selection. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 143–153. doi:10.1109/ICDCS.2018.00024

[68] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. 2020. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 723–737. https://www.usenix.org/conference/nsdi20/presentation/takruri

[69] Vinh Tao, Marc Shapiro, and Vianney Rancurel. 2015. Merging semantics for conflict updates in geo-distributed file systems. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR '15)*. Association for Computing Machinery, Haifa, Israel, Article 10, 12 pages. doi:10.1145/2757667.2757683

[70] Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh, and Samer Al-Kiswany. 2024. Draconis: Network-Accelerated Scheduling for Microsecond-Scale Workloads. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, Athens, Greece, 333–348. doi:10.1145/3627703.3650060

[71] Romain Vaillant, Dimitrios Vasilas, Marc Shapiro, and Thuy Linh Nguyen. 2021. CRDTs for truly concurrent file systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*. Association for Computing Machinery, Virtual, USA, 35–41. doi:10.1145/3465332.3470872

[72] Marc-André Vef, Rebecca Steiner, Reza Salkhordeh, Jörg Steinkamp, Florent Vennetier, Jean-François Smigielski, and André Brinkmann. 2020. DelveFS - An Event-Driven Semantic File System for Object Stores. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Kobe, Japan, 35–46. doi:10.1109/CLUSTER49012.2020.00014

[73] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 277–292. https://www.usenix.org/conference/fast21/presentation/wang

[74] Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. 2021. Lunule: An Agile and Judicious Metadata Load Balancer for CephFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, St. Louis, Missouri, Article 47, 16 pages. doi:10.1145/3458817.3476196

[75] 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 (EuroSys '23)*. Association for Computing Machinery, Rome, Italy, 331–346. doi:10.1145/3552326.3587443

[76] 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 (OSDI '06)*. USENIX Association, Seattle, Washington, 307–320.

[77] Jingwei Xu, Junbin Kang, Mingkai Dong, Mingyu Liu, Lu Zhang, Shaohong Guo, Ziyan Qiu, Mingzhen You, Ziyi Tian, Anqi Yu, Tianhong Ding, Xinwei Hu, and Haibo Chen. 2025. FalconFS: Distributed File System for Large-Scale Deep Learning Pipeline. arXiv:2507.10367 [cs.DC] https://arxiv.org/abs/2507.10367

[78] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. 2021. Access Control Conflict Resolution in Distributed File Systems using CRDTs. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '21)*. Association for Computing Machinery, Virtual Event, United Kingdom, Article 1, 3 pages. doi:10.1145/3447865.3457970

[79] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, Virtual Event, USA, 126–138. doi:10.1145/3387514.3405857

[80] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. 2024. Fast and Scalable In-network Lock Management Using Lock Fission. In *18th USENIX Symposium on Operating Systems Design and Implementation*

(OSDI 24). USENIX Association, Santa Clara, CA, 251–268. https://www.usenix.org/conference/osdi24/presentation/zhang-hanze

[81] Shuanglong Zhang, Robert Roy, Leah Rumancik, and An-I Andy Wang. 2020. The Composite-File File System: Decoupling One-to-One Mapping of Files and Metadata for Better Performance. *ACM Trans. Storage* 16, 1, Article 5 (March 2020), 18 pages. doi:10.1145/3366684

[82] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K. Paul, Keren Chen, and Ali R. Butt. 2021. Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems. *IEEE Trans. Parallel Distrib. Syst.* 32, 4 (April 2021), 918–930. doi:10.1109/TPDS.2020.3034517

[83] 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 (SC '21)*. Association for Computing Machinery, St. Louis, Missouri, Article 48, 15 pages. doi:10.1145/3458817.3476148

[84] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.* 13, 3 (nov 2019), 376–389. doi:10.14778/3368289.3368301

[85] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. 2024. In-Memory Key-Value Store Live Migration with NetMigrate. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 209–224. https://www.usenix.org/conference/fast24/presentation/zhu

A Appendix

In the appendix, we provide a detailed discussion of the correctness and consistency of SwitchFS.

A.1 Correctness After a Crash

In this subsection, we discuss how SwitchFS maintains consistency after a server crash, a switch crash, or both, and provide a concrete example of handling server failure during aggregation.

Switch failure. When a switch recovers from failure, it initializes an empty dirty set, and servers aggregate all directories. Thereby, SwitchFS recovers to a consistent state.

Server failure. When a server recovers from failure, it recovers its volatile states as follows: it recovers the key-value store and change-log from its local WAL, and recovers the invalidation by cloning that on the other servers. Note that each not-yet-aggregated directory update is stored in the WAL of either the change-log's or the directory's owner server, so no updates are lost. The server also proactively aggregates all directories it owns to ensure any interrupted aggregations it issued before the crash (whose corresponding fingerprint may have been removed on the switch) are executed to completion, so that the on-switch dirty set correctly reflects the states of directories.

Example of server failure during aggregation. Let's consider an example of server failures during aggregation, involving Server-A (which hosts directory D's inode) and Server-B (which hosts one of D's change-logs).

Consider that at the time one or both servers crash, some change-log entries may have been written to Server-A's WAL, and a subset of those are marked as "applied" in Server-B's WAL.

If Server-A crashes, it applies the change-log entries that have been written into Server-A's WAL to D's inode. Then, Server A proactively aggregates all directories it owns, including D, retrieving the remaining change-log entries of D from Server-B.

If Server-B crashes, Server-A's aggregation times out (due to not receiving a signal from Server-B that all change-log entries have been retrieved) and begins retrying. Server-B then recovers all change-log entries not marked as "applied" from its WAL, and Server-A aggregates them upon retrying. Notably, Server-A may receive duplicate entries and leverages entry IDs to ensure each entry is processed only once.

If both servers crash, Server-A applies the change-log entries that have been written into Server-A's WAL to D's inode, and Server-B recovers the remaining change-log entries from Server-B's WAL. Then, Server A proactively aggregates all directories it owns, including D, retrieving all the change-log entries of D.

Idempotence of recovery. The recovery procedures described above are idempotent. First, the recovery of volatile states (i.e., rebuilding the key-value store, the change-log, and the invalidation list) is idempotent, as it always starts from a clean state after a crash. Second, the aggregation is idempotent, because: (a) A directory owner server persists a change-log entry in its WAL before the sender server marks it as "applied" in its WAL, ensuring that no change-log entry is lost. (b) Repeated aggregation of the same change-log entry does not compromise correctness, as servers check entry IDs and ensure each entry is processed only once. Therefore, nested crashes do not compromise correctness.

A.2 Operation Consistency and Properties

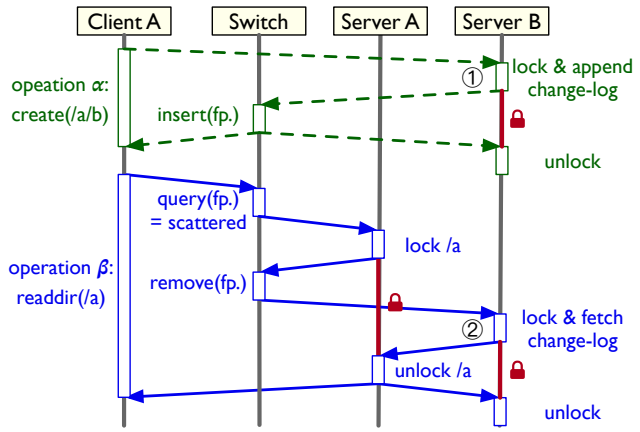
In this subsection, we first demonstrate that SwitchFS is serializable and preserves the real-time order in a crash-free environment. Then, we show that SwitchFS maintains serializability in the presence of crashes.

A.2.1 Consistency in a Crash-Free Environment. We demonstrate that SwitchFS ensures the following two properties, when no crash occurs:

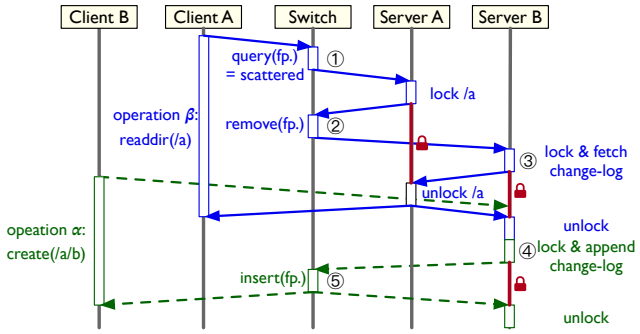
- Metadata operations are serializable.
- Any metadata operation β that is issued after operation α returns can see the changes made by α.

Property 1: *Metadata operations are serializable.*

In SwitchFS, synchronous operations (e.g., open, read, stat) are serialized through locking as in a traditional DFS, and two asynchronous updates (e.g., *mkdir*, *create*) are serialized according to the order in which they obtain the write lock



(a) *readdir(/a)* is serialized after *create(/a/b)*.



(b) *readdir(/a)* is serialized before *create(/a/b)*.

Fig. 20: **Two possible interleavings between *create(/a/b)* and *readdir(/a)* when the dirty set query returns *scattered*.** Numbers in cycles indicate the causal order of events.

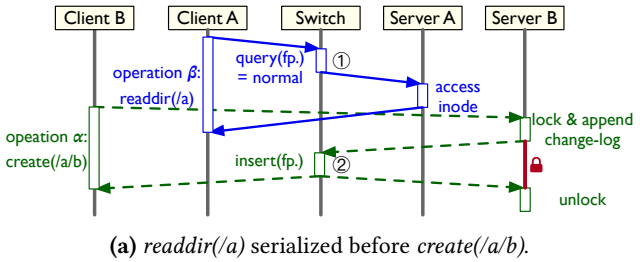
on the parent directory's change-log. We discuss how asynchronous updates and directory reads are serialized below.

Consider two operations, α and β: α performs an asynchronous update to a directory, and β reads the same directory. Without loss of generality, let α be *create(/a/b)* and β be *readdir(/a)*. Assume that Server-A hosts the inode of directory */a*, and Server-B hosts the inode of file */a/b*.

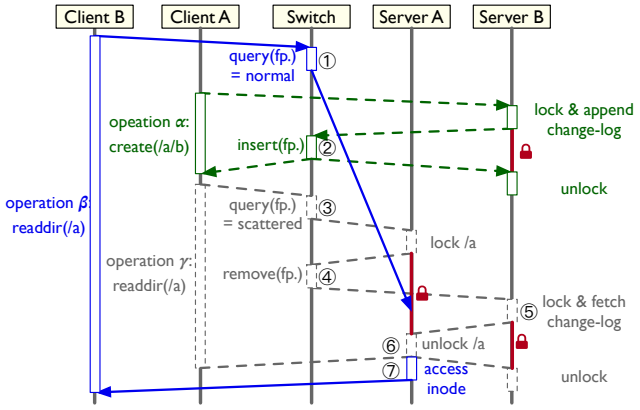
The executions of α and β may interleave in several ways. We classify these interleavings by whether β observes directory */a* as *scattered* or *normal* in the on-switch dirty set.

Case 1: β finds /a marked as* scattered *during its query of the dirty set. Depending on whether α updates the change-log on Server-B before or after β acquires the lock, two sub-cases arise. Fig. 20 illustrates their respective workflows.

- **Case 1.a**: α updates the change-log (①) before β acquires the lock (③), as shown in Fig. 20(a). In this case, β fetches α's update during aggregation and thus observes its effect, serializing α before β.
- **Case 1.b**: α updates the change-log (④) after β acquires the lock (③), as shown in Fig. 20(b). In this case, β does not fetch α's update during aggregation and therefore does not observe its effect, serializing α after β. Notably, a chain of happen-before relations exists in this interleaving: (a) α inserts the fingerprint of */a* into the dirty set (⑤) after appending the change-log (④), and (b) β removes the fingerprint from the dirty set (②) before acquiring the change-log lock (③). Therefore, α's insertion of the fingerprint (⑤) occurs strictly after β's removal of it (②). As a result, a subsequent directory read will detect the fingerprint inserted by α and initiate an aggregation that retrieves α's update, thereby making α's effect visible.



(a) *readdir(/a)* serialized before *create(/a/b)*.



(b) *readdir(/a)* serialized after *create(/a/b)*.

Fig. 21: **Two possible interleavings between *create(/a/b)* and *readdir(/a)* when the dirty set query returns *normal*.** Numbers in cycles indicate the causal order of events.

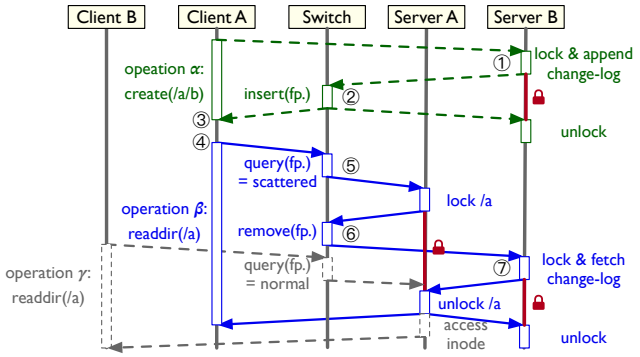


Fig. 22: **Directory reads issued after a directory update can see the effect of the latter.** In this figure, operation α is a directory update, and operations β, γ are directory reads. Numbers in cycles indicate the causal order of events.

Case 2: β finds /a marked as* normal *during its query of the dirty set. Depending on whether β observes the effect

of α, two sub-cases arise. Fig. 21 illustrates their respective workflows.

- **Case 2.a:** β reads the inode of directory */a* on Server-A and returns without observing α's update, as shown in Fig. 21(a). In this case, α is serialized after β. Notably, since β finds */a* marked as "normal" in the dirty set, β must have queried the dirty set (①) before α inserts the *fingerprint* of */a* (②).
- **Case 2.b:** Another operation γ triggers an aggregation after α updates the change-log but before β reads the inode, as shown in Fig. 21(b). Consequently, β observes the effect of α, and α is serialized before β. Specifically, β queries the dirty set (①) before α inserts the *fingerprint* of */a* (②). Subsequently, γ queries the dirty set (③) and initiates an aggregation on directory */a* (④), during which the inode of */a* is locked. β's request arrives at Server-A after γ acquires the lock, waits for the aggregation to complete (⑤, ⑥, ⑦), then reads the inode of */a* and returns (⑧). This case is possible due to the potential arbitrary reordering of packets in the network.

In all the scenarios discussed above, operations are correctly serialized, eliminating any Time-Of-Check-To-Time-Of-Use (TOCTTOU) issues.

Property 2: *Any metadata operation β that is issued after operation α returns can see the changes made by α.*

In SwitchFS, since synchronous operations apply in-place updates to objects before returning, their effects are visible to subsequent operations, as in traditional DFSs. Below, we explain how asynchronous updates and directory reads preserve this property.

In Fig. 22, operation α performs an asynchronous directory update on directory */a*, while operations β and γ read directory */a*. We classify directory reads issued after α returns into two categories based on whether they observe the directory as *scattered* or *normal* in the dirty set, and operations β and γ are examples. We illustrate why both categories must observe the changes made by α below.

- If the directory read (i.e., operation β) finds the directory marked as *scattered* during its query of the dirty set (⑤), it triggers an aggregation that fetches updates for */a*. According to the causal order of events marked in Fig. 22, β must acquire the lock on the change-log (⑦) after α updates it (①). Therefore, as discussed in Case 1.a of Property 1, β is guaranteed to see the effect of α.
- If the directory read (i.e., operation γ) finds the directory marked as *normal* during its query of the dirty set, it indicates that the *fingerprint* of the directory has been removed from the dirty set by an earlier aggregation (⑥). Since the aggregation blocks directory reads until it completes, γ will see the directory updates fetched by the aggregation, including α's update.

A.2.2 Consistency in the Presence of Crashes.

We demonstrate that operations are serializable in the presence of crashes, as follows.

When a crash occurs, interrupted operations do not return, and clients retry them after recovery. Interrupted metadata reads and uncommitted updates are invisible to clients and leave no side effects, thus not affecting operation serializability. We only need to consider the order of committed metadata update operations after recovery.

The recovery procedures in §A.1 recover operations in the same order as they were committed (written into the WAL) before the crash. Since operations acquire locks before committing (§5.2), the commit order is serialized by locks. Given that locks also serialize the order of metadata-write operations in a crash-free environment (see discussions under property 1 in §A.2), the operation order with and without a crash is consistent. Therefore, operations are serializable in the presence of crashes.

A.3 Fault Tolerance of Cluster Reconfiguration

The procedure of cluster reconfiguration in §5.5 is fault-tolerant. We first present the detailed reconfiguration procedure, then discuss its fault tolerance.

Reconfiguration procedure. SwitchFS employs a centralized coordinator to manage cluster configuration changes. When servers are added or removed, the coordinator carries out the reconfiguration in three steps:

1. The coordinator notifies all servers to stop serving normal requests and to aggregate the directories they currently own.
2. Once all servers acknowledge the completion of the aggregation, the coordinator notifies them of the new configuration. Each server then determines the set of inodes that it no longer owns according to the new configuration and consistent hashing, and migrate these inodes to the new owners via two-phase commit.
3. After all servers acknowledge completing the migration, the coordinator notifies them to resume serving requests.

Fault handling. Each step of the reconfiguration is designed to be idempotent. The coordinator persistently records the start and completion of each step in its local WAL. If the coordinator crashes during reconfiguration, it asks all the servers to re-execute the reconfiguration from the last step that was not complete. If a server crashes, it first restores its volatile state by replaying its WAL, then obtains the current reconfiguration step from the coordinator, and re-executes that step.