CFS: Scaling Metadata Service for Distributed File System via Pruned Scope of Critical Sections

Yiduo Wang[†§] Yufei Wu[†] Cheng Li[†ζ] Pengfei Zheng[§] Biao Cao[§]
Yan Sun[§] Fei Zhou[§] Yinlong Xu[†ζ] Yao Wang[§] Guangjun Xie[§]

[†]University of Science and Technology of China [§]Baidu (China) Co., Ltd
[ζ]Anhui Province Key Laboratory of High Performance Computing

{duo,yufeiwu}@mail.ustc.edu.cn,chengli7@ustc.edu.cn,{zhengpengfei,caobiao,sunyan09,zhoufei05}@baidu.com
ylxu@ustc.edu.cn,{wangyao02,xieguangjun}@baidu.com

Abstract

There is a fundamental tension between metadata scalability and POSIX semantics within distributed file systems. The bottleneck lies in the coordination, mainly locking, used for ensuring strong metadata consistency, namely, atomicity and isolation. CFS is a scalable, fully POSIX-compliant distributed file system that eliminates the metadata management bottleneck via pruning the scope of critical sections for reduced locking overhead. First, CFS adopts *a tiered metadata organization* to scale file attributes and the remaining namespace hierarchies independently with appropriate partitioning and indexing methods, eliminating cross-shard distributed coordination. Second, it further scales up the single metadata shard performance by *single-shard atomic primitives*, shortening the metadata requests' lifespan and removing spurious conflicts. Third, CFS drops the metadata proxy layer but employs the light-weight, scalable *client-side metadata resolving*. CFS has been running in the production environment of Baidu AI Cloud for three years. Our evaluation with a 50-node cluster and microbenchmarks shows that CFS simultaneously improves the throughput of baselines like HopsFS and InfiniFS by 1.76-75.82× and 1.22-4.10×, and reduces their average latency by up to 91.71% and 54.54%, respectively. Under cases with higher contention and larger directories, CFS' throughput benefits expand by one order of magnitude. For three real-world workloads with data accesses, CFS introduces 1.62-2.55× end-to-end throughput speedups and 35.06-62.47% tail latency reductions over InfiniFS.

CCS Concepts: • **Software and its engineering → Distributed systems organizing principles**; • **Information systems → Distributed storage**; • **Social and professional topics → File systems management**.

Keywords: Distributed file system, Metadata management

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3587443

1 Introduction

For two decades since the origin of GFS [22] and HDFS [24], Distributed file systems (DFSs) adopted the conventional wisdom to decouple metadata management from file data storage for independently scaling the two parts, and to operate on a cluster of machines, each of which assigned to roles such as metadata server (MDS) or storage server [28, 39, 43, 55, 63, 70, 76]. With this setting, MDS should be first contacted prior to the data access.

Scaling metadata performance is crucial. First, its operations dominate as the number of files in modern workloads grows at an unprecedented scale with the majority of files being small, e.g., a few or tens of KBs [7, 11, 27, 57, 74, 75]. Second, data access have been made extremely fast, due to parallel processing enabled by evenly balancing file data across storage servers [24, 35, 42], and the adoption of fast-evolving storage media [10, 31].

Unfortunately, the metadata scalability is limited by offering strong consistency in terms of atomicity and isolation, which comprises POSIX semantics [12, 25, 30], desired by the vast majority of applications. Conventional approaches heavily rely on transactions to group multiple metadata mutations belonging to a POSIX system call, and the lock-based coordination, to survive failures and avoid data races [44, 48]. However, our study reveals that the locking overhead is not affordable, e.g., accounting for 52.91% of the metadata request processing in the best case with even no contention, while reaching 93.86% with highly conflicting workloads.

This coordination problem can be further exacerbated by the fine-grained metadata partitioning for load balance [28, 42, 48, 52, 63, 76], due to cross-shard accesses. To reduce the overhead of distributed locking and coordination, recent proposals [40, 44, 54, 55, 70] group related metadata while sharding. But, this comes at the price of sacrificing the load balance of serving file attributes, whose accesses are more

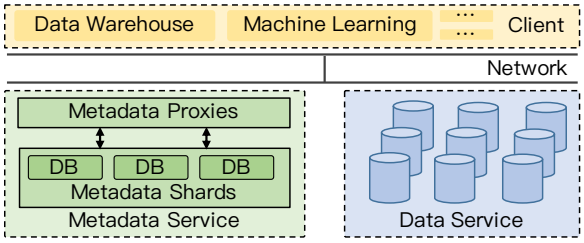


Figure 1. The general architecture of modern distributed file systems with a disaggregated metadata service

prevalent in production environments, and also giving up the scalability of a single metadata shard.

In this paper, we present a highly scalable, fully POSIX-compliant distributed file system CFS, which scales out its metadata service with the following innovations.

First, we adopt *a tiered metadata organization* to prune the scope of critical sections of cross-shard metadata requests, while perfectly balancing massive visits to file attributes. We argue that the file attributes and the rest of namespace hierarchies favor different partitioning methods for being scaled out. Therefore, within CFS, the two kinds of metadata are not treated equally, instead, managed in a tiered storage, where we store file attributes close to their data on data storage, while using a distributed database layer to host the remaining namespace hierarchies. This design relieves the tension on the distributed database layer, and further enables to use the locality-preserving partitioning for namespace hierarchies to eliminate cross-shard transactions and their associated coordination overhead. On the other hand, it balances file attributes as their file data via hash-based partitioning, thus precluding hotspots.

Second, we create a few *single-shard atomic primitives* in the database layer, each grouping multiple commands that read/write a few metadata objects and belong to one metadata request into a single command for fast execution within a single metadata shard. To support various metadata requests, we design these primitives as parameterized functions, which can be instantiated with different inputs. In addition, conflict reconciliation procedures remove spurious conflicts (which are easily mergeable) between concurrent metadata requests so that they can be executed in parallel without acquiring locks. All together, these primitives scale up the single shard processing capacity.

Finally, we incorporate the above two design principles into our system CFS. Due to the simplified metadata processing, CFS further improves the modern metadata service architecture by eliminating the metadata proxy layer, instead, merging the metadata resolving function into the client-side library. Note that CFS has been deployed and running in the production environment of Baidu AI Cloud for three years, and supported various applications ranging from social networks, online shopping, to machine learning.

We evaluate CFS with metadata benchmarks and production workloads using a 50-node cluster. Experimental results show that for the no contention cases, where all systems perform the best, CFS significantly improves HopsFS' and InfiniFS' peak throughput by up to 75.82× and 4.10×, and reduces their latency by up to 91.71% and 54.54% , respectively, across all microbenchmark tests. With workloads exhibiting high contention, CFS scales significantly better than baselines, e.g., it achieves a throughput record of 3.44 Mops and 17.96 Kops per second for `getattr` and `mkdir`, respectively, 34.19× and 54.48× of that of InfiniFS. For three production workloads with data access enabled, CFS achieves end-to-end throughput speedups of 1.62-2.55×, compared with InfiniFS.

2 Background and Motivation

It has been proven that metadata scalability is equally or more important than that of file data, since metadata operations dominate the whole file system operation space in the modern workloads [1, 3, 37]. To understand this, we measure the ratios of metadata and data operations in nine representative workloads of various applications that use DFSs, ranging from social networks to online shopping, in Baidu AI Cloud. Surprisingly, the metadata operations account for 67%-96% of the total number of DFS requests. In the most skewed workload, the percentage of metadata operations is 28.2× higher than the data counterpart.

2.1 Metadata Management in DFSs

Two decades ago, GFS [22] and HDFS [24] decoupled metadata from file content with the former being managed via a single separate metadata server (MDS), with a special focus on multi-GB files. However, the new trends that file sizes are decreasing to a few tens of KBs but the file quantity continues to expand (e.g., up to billions) have already undermined some earlier and driven new designs [39, 48, 71]. To scale out the metadata service, modern DFSs such as CephFS [70] and Lustre [43] partition file system namespace across a cluster of MDSs with the hope to serve metadata requests in parallel.

HopsFS [48] and InfiniFS [44] adopt a disaggregated metadata architecture with a few independently-scalable metadata layers. Figure 1 illustrates such a DFS architecture with a two-layer metadata service. A group of database instances organize metadata as database records and tables, which are further sharded and distributed for load balance. In addition, a proxy layer constructs the file system metadata logic over the database layer, e.g., encapsulating metadata operations into database transactions. This disaggregated design has been already incorporated into industrial systems such as Azure ADLS [54] and Facebook Tectonic [52].

2.2 High Overhead for Strong Metadata Consistency

To support rich applications, the vast majority of DFSs [12, 25, 32, 40, 43, 46] offer POSIX(-like) semantics [6, 23, 30, 68],

```
1 START TRANSACTION;
2 SET @parent := (SELECT * FROM inodes WHERE id=
      @parent_id); -- Check if the parent dir exists
3 IF @parent.type=DIR THEN
4   INSERT (parent_id=@parent.id, id=@id, name=@name,
      ...) INTO inodes;-- Insert the record of new file
5   UPDATE inodes SET (time=@now, ...) WHERE (id=
      @parent_id);-- Update parent dir's attributes
6 COMMIT
```

Figure 2. Code snippet for the file creation transaction within HopsFS and other similar systems

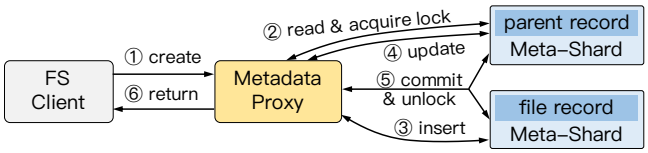


Figure 3. The workflow of executing the file creation transaction (Figure 2) in HopsFS

which require maintaining strong metadata consistency in terms of the *atomicity* and *isolation* properties. The former survives partial updates in the presence of failures, and the latter coordinates concurrent metadata requests for avoiding inconsistent states left by harmful data races.

The metadata POSIX semantics are often protected by the form of transactions, each of which groups a few metadata reads and writes and executes the grouped commands in an atomic and isolated fashion. Figure 2 showcases the transaction composition of a file `create` metadata request within HopsFS. It contains a `read` statement (line 2) to fetch the metadata of the target parent directory, followed by a conditional check that verifies its existence (line 3). If so, this transaction further executes two `write` statements, which first create the metadata record for the new file (line 4), and then update the namespace hierarchy to link the newly created file metadata to its parent directory (line 5). In the end, this transaction commits to persist the changes (line 6).

Modern DFS metadata services [32, 44, 48, 55, 64] heavily rely on the expensive locking mechanism to offer atomicity and isolation so that on-going metadata transactions will not see partial results from others and conflicting transactions are serialized. Figure 3 illustrates the execution of the above transaction. The file system client initiates the transaction by issuing the `create` request to a metadata proxy (①), responsible for coordinating the transaction execution internally within HopsFS' metadata service. The proxy reads the parent directory metadata from a shard and simultaneously acquires the write lock on it (②). Upon success, the proxy contacts the corresponding metadata shards (can be the same one, depending on different partitioning methods) to perform the file metadata creation and the parent metadata mutation (③ - ④). Finally, the client will be notified (⑥) upon commit and lock release (⑤). Note that, the ⑤ step may go through the two phase-commit protocol to coordinate the atomic execution of cross-shard changes.

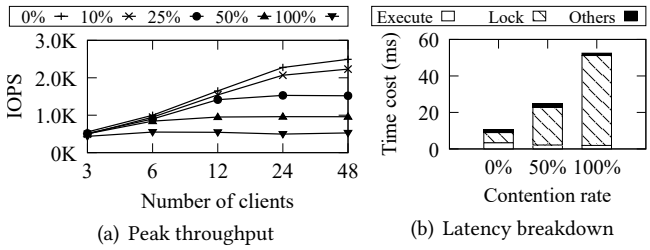


Figure 4. Performance of `create` within HopsFS under different workload intensity levels and contention rates

To explore the locking overhead and its negative impacts on metadata scalability, we deploy a small-scale HopsFS with its metadata service consisting of three database instances (see details in Section 5.1), and run 3-48 concurrent clients to issue `create` operations, while emulating different contention rates by the probability for clients to touch the same directory. In Figure 4(a), with no contention, HopsFS' metadata service almost scales linearly w.r.t the number of clients. However, with increasing contention rates, the metadata performance significantly degrades. The throughput curve becomes flat when reaching 100% contention rate.

We further break down the file creation latency in Figure 4(b). First, even without contention, HopsFS still pays substantial locking overhead, accounting for 52.91% of the whole metadata request processing time. Second, the time cost of the "Execute" step (including ③-⑤) and the remaining internal process except ② ("Others") remain almost constant across different contention rates. In contrast, the locking overhead increases dramatically, reaching 83.18% and 93.86% of the average metadata request time cost at the contention rate of 50% and 100%. Note that high contentions in metadata access are not rare, and many studies and our experience show that applications like big data analysis often concurrently read from or write to a shared directory [48, 52].

Unfortunately, the locking complexity can be further exacerbated by the metadata partitioning methods, which are the key to improve metadata load balance and overall performance. On the one hand, many systems use fine-grained partitioning methods like hashing for perfectly balancing metadata accesses across shards [28, 43, 63, 76], at the price of introducing cross-shard metadata transactions. The direct consequence is expanding the scope of their critical sections and increasing the lock overhead. On the other hand, systems adopt locality-preserving metadata partitioning, such as subtree partitioning [70], range partitioning [8, 20, 54], and parent-children grouping [44], to eliminate the cross-shard transactions. Nevertheless, this comes at the price of potential hotspots [47, 59, 69]. In the worst case, the locking mechanism would serialize all transactions on a single shard, also limiting the overall metadata performance.

Table 1. Aggregated percentage of metadata operations triggered by POSIX calls from nine workloads in Section 2

Op	Ratio	Op	Ratio	Op	Ratio
create	1.44%	lookup	17.80%	unlink	1.14%
getattr	75.25%	mkdir	0.08%	setattr	3.21%
rmdir	0.04%	readdir	0.92%	rename	0.12%

3 CFS Overview

3.1 Design Rationale

In this paper, we aim to build CFS, a highly scalable, fully POSIX-compliant distributed file system with its metadata service scaling out to offer high throughput and low latency. To do so, we inherit the aforementioned metadata decoupling and disaggregation architectures but with a set of novel approaches to safely prune the scope of critical sections of metadata requests for reducing coordination overhead, while offering strong metadata consistency. Our design is based on the following key observations and innovations, reflecting a marked departure from their earlier assumptions.

First, we aim to reduce the scope of metadata request execution from cross-shard to single-shard for eliminating distributed coordination. This requires to re-think the organization of metadata objects to simultaneously cope with their different access patterns and scaling natures. In Table 1, the majority of metadata operations such as `getattr`[1] are concentrating on file attributes, which favor a flat organization, and are accessed in parallel alongside their data operations. Contrary, the directory-related metadata operations, such as `create` and `unlink`, insert or remove file inodes, followed by modifying their parent directories' inode attributes. Thus, it prefers to preserve the namespace hierarchical structure and co-locate the parents' attributes and the children's inodes.

These trends motivate us to divide the metadata management into two separated layers, where namespace hierarchies are still handled by a distributed database layer with locality-preserving partitioning and rich semantics, while file attributes are managed by another storage tier via hash-based partitioning and simple key-value interfaces.

Second, we further trim the critical sections of metadata requests by safely relaxing the transactional atomicity and isolation (e.g., serializable snapshot isolation [9]), which are too strong for ensuring POSIX metadata semantics. We argue that metadata transactions are not arbitrary ones, instead, have quite limited forms. Throughout an analysis of POSIX APIs, we conclude that without coordination, there are only two possible inconsistencies, namely, (1) corrupted mappings, e.g., the file creation succeeded, but linking itself to its parent failed; and (2) lost updates, e.g., two simultaneous directory creations under a common parent directory both overwrite the `links` attribute of their parent, leaving the `links` value to be 1 less than the actual value.

[1] `getattr` is not a POSIX interface but a common metadata operation in DFSs such as NFS to implement POSIX interfaces like open and `stat`.

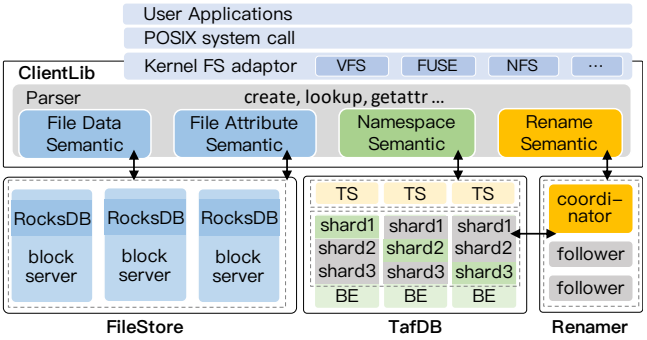


Figure 5. The architecture of CFS. Sub-components within dotted-line boxes are managed by a raft group for high availability. Arrows indicate network communication going through our highly optimized RPC library.

With the tiered metadata organization, we can tolerate corrupted mappings by executing metadata changes across two tiers in a deterministic order so that the file system indices may be internally corrupted but are always externally consistent, i.e., corrupted states are not visible to outside. We can further shrink the scope of critical section of single-shard metadata transactions by grouping their logics that may span across multiple objects and execute all of them at once so that the lifespan and locking overhead of those transactions can be greatly reduced. Furthermore, we avoid lost updates by leveraging the fact that these concurrent updates are always mergeable. Thus, we treat them as spurious conflicts, which can be safely removed from the critical sections.

Third, compared to the state-of-the-art solutions, the metadata proxy layer in our case has been much simplified. It is not required to be in charge of coordinating transactions and keeping track of locks. Instead, the only function left is to resolve clients' metadata requests and route them to the appropriate system components in the metadata service. Following that, we argue that there is no need to maintain an independent metadata proxy layer anymore, instead, we can collapse it into the client library for light-weight *client-side metadata resolving*. Doing so will simplify the metadata service architecture, avoid extra communication steps, and can scale to many clients for free.

3.2 Overall Architecture

In Figure 5, CFS is made of four key system components, namely, *TafDB*, *FileStore*, *Renamer*, and *ClientLib*.

Namespace store layer (*TafDB*). We deploy *TafDB* as the namespace store layer to manage the namespace hierarchies except file attributes in a distributed database via a set of customized primitives and safely relaxed atomicity and isolation (details in Section 4.1 and 4.2). *TafDB* is further divided into two sub-modules as follows. First, there is a group of time

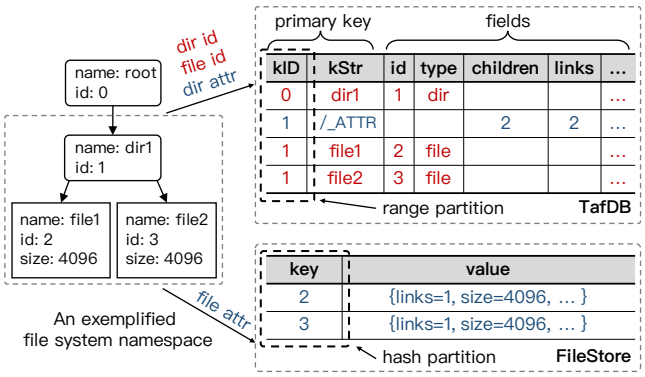


Figure 6. Organizing metadata between *TafDB* and *FileStore*

servers (TS) assigning monotonically increasing timestamps to order metadata transactions. Second, we maintain a set of backend servers (BEs) which store metadata information as database tables. To offer high availability, we replicate BEs' states in groups, managed and coordinated via the Raft consensus protocol [49]. All metadata mutations going through *TafDB* must be first persisted in the write-ahead log (WAL) replicated within the target Raft-BE group.

File storage layer (*FileStore*). *FileStore* is a flat, distributed object store for file data blocks, the unit of data storage in CFS. Each store node additionally runs a local RocksDB [61] to keep the attribute metadata of the corresponding files for fast metadata retrieval. To offer high availability, again, each data block and metadata WAL is replicated across multiple store instances. Typically, a three-way replication is deployed.

Rename service (*Renamer*). We appoint a dedicated service *Renamer* for processing the complex `rename` metadata requests, which may introduce anomalies and cannot be handled by the stand-alone *TafDB*, similar to InfiniFS [44]. Enhancing *TafDB* would resort to the strongest isolation level with unnecessary performance penalties imposed on the remaining types of metadata requests. *Renamer* consists of a group of raft-protected servers, with one designated as the coordinator. The coordinator will inspect complex `rename` requests and execute valid ones by reading from and writing to *TafDB* and *FileStore*. This enables to deploy the *Renamer* cluster at a small scale to reduce the deployment cost. The details will come in the Section 4.3.

Client library (*ClientLib*). The entrance to CFS is *ClientLib*, exposing to applications with CFS' metadata operations such as `getattr`, `create`, and `mkdir`, and file operations such as `open`, `read` and `write`. As *ClientLib* caches the partition information of *TafDB* and *FileStore*, it implements a *client-side metadata resolving*, and directly interacts with the different components of CFS for shortening the request handling time interval. Overall, there are three paths from *ClientLib* to the rest of CFS, namely, file data and attribute requests sent to *FileStore*, complex `rename` requests forwarded to *Renamer*, and the remaining ones posted to *TafDB*.

POSIX File System Interface Compliance. CFS complies with POSIX semantics by leveraging the VFS adapter, which sits in the Linux kernel and acts as a switch to map POSIX file system interfaces to low-level functions offered by specific implementations of various file systems. To do so, CFS attaches *ClientLib* to VFS and implements VFS' functions to orchestrate user-level POSIX function calls into CFS' internal metadata or data operations. For instance, the POSIX `open` interface with `O_CREAT` flag, which may be composed of the `lookup` and `create` internal metadata operations in CFS if the target file does not exist. Similarly, the POISX `read` interface, fetching file content, will trigger the following workflow in CFS, i.e., *ClientLib* issues `getattr` to *FileStore* for retrieving attributes such as file modification time, followed by executing `read` to *FileStore* to fetch requested data when the modification time differs from the cached one. To make CFS easy to use, we additionally offer FUSE [21], Samba [66] and NFS-Ganesha [19] compatible interfaces.

To assess the full POSIX file system interface compliance, we intensively test CFS with the commonly used test suite, pjdfstest [53], and compare with other popular file system implementations such as ext4, XFS, and CephFS. The test log files can be found [13] and the comparison shows that both ext4 and CFS passed all 8832 test cases, while XFS and CephFS failed in 2 and 237 test cases, respectively.

4 Design and Implementation

4.1 Metadata Organization and Partition

Storing namespace hierarchies in *TafDB* for better locality. In Figure 6 (top right), we organize the file system metadata except file attributes into a giant `inode_table` in *TafDB*. Unlike the conventional designs [48, 54] that put the metadata information in multiple tables, our unified design is simple yet efficient as we avoid the complexity to maintain cross-table consistency for many metadata operations such as `create` that additionally update parents' metadata. Similar to InfiniFS [44], we also divide each directory inode into two database records, namely, the inode id record (row 1), and the attribute record (row 2). This decoupling removes contentions on accessing the two types of records. In addition, the table keeps records of file inode ids (row 3-4).

All table records share the same composite primary index in the form of <`kID`, `kStr`>, where the former stands for the inode id, and the latter is a *string* field. As for each directory or file id record, the `kID` and `kStr` values are its parent's inode id and its directory or file name (row 1, 3, 4 in Figure 6), respectively. Unlike this, each directory attribute record sets its inode id to `kID`, and a reserved keyword `"/_ATTR"` to `kStr` (row 2). In addition, each record has a list of optional fields, such as `id`, `type`, `children`, `links`, `size`, `time`, etc, with the unused fields set to `NULL`.

Here, we break `inode_table` into a set of shards, the unit of metadata storage in *TafDB*, by a range partitioning scheme

Table 2. The *single-shard atomic primitives* provided by TafDB, where *cond* stands for condition

Primitives	Use cases	Syntax	Description
insert_ with_update	`create` `mkdir` `symlink` `link`	INSERT (*value_list*) WITH UPDATE *table_name* SET (*assignment_list*) WHERE (*update_cond*)	(1) check if the record that matches *update_cond* exists; (2) insert a new inode id record; (3) update the existing record according to *assignment_list*.
delete_ with_update	`unlink` `rmdir`	DELETE (*delete_cond*) WITH UPDATE *table_name* SET (*assignment_list*) WHERE (*update_cond*)	(1) check if records matching *delete/update_cond* exist; (2) delete the record that matches *delete_cond*; (3) update the record that matches *update_cond* according to *assignment_list*.
insert_ and_delete _ with_update	`rename`	INSERT (*value_list*) WITH DELETE (*delete_cond_list*) WITH UPDATE *table_name* SET (*assignment_list*) WHERE (*update_cond*)	(1) check if records that match their specified conditions exist; (2) delete records that match any condition in *delete_cond_list*; (3) insert a new inode id record with changes of primary key and fields specified by *value_list*; (4) update the record based on *update_cond*, *assignment_list*.

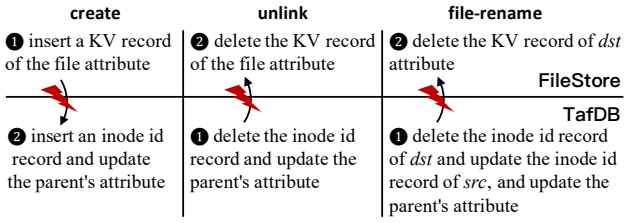


Figure 7. Fixed execution order for metadata requests that contact both *TafDB* and *FileStore* with no 2PC coordination

on the `kID` values, and distribute shards across backend servers. The combination of the above metadata organization and the `kID`-based range partitioning preserves locality by placing a directory's attribute and all its children's inode ids on a single metadata shard. This is made possible even for large directories with millions of files, which are considered not to be uncommon [48], since we offload file attributes to *FileStore* (details as follows), each consuming 0.2KB of storage space. Thus, the metadata of extremely large directories will not exceed the storage capacity of a single *TafDB* shard.

Finally, we introduce a few customized primitives (details in Section 4.2) to simplify the directory-related metadata reads or updates, which are simpler than SQL queries but offer stronger semantics than the key-value interfaces.

Storing file attributes in *FileStore* for better load balance and hotspots elimination. We put the file attributes close to their data on the same *FileStore* node. Each such node runs a RocksDB instance, at the bottom right of Figure 6, which organizes the file attributes as key-value pairs, where keys are inode ids while values are byte streams encoded by file attributes, similar to the above directory attributes. Due to the simple key-value interface, manipulating file attributes through *FileStore* is cheaper than doing so in *TafDB*.

Here, instead of applying the range partitioning, we evenly distribute file attributes along with their data across data storage nodes in *FileStore* by computing the hash values of their ids. This design allows to serve file attributes in parallel from many data storage nodes, and achieve load balance even under the large directory cases. In such cases, the conventional approaches would likely suffer hotspots, as their locality-preserving partitioning methods place attributes of files under a shared directory on a single metadata shard. Contrary, CFS breaks this limit by assigning proper partitioning methods to namespace hierarchies and file attributes, by taking into account their access patterns.

Distributed transaction elimination. Spanning metadata across *TafDB* and *FileStore* would require coordinating distributed transactions involving the two parties for atomicity. To eliminate such coordination overhead, we carefully design the flow of metadata requests, which executes mutations in a deterministic order so that failures occurring between them would not lead to harmful inconsistencies. The idea is to push the file system namespace hierarchy update to be the final step of any creation metadata request, but to promote it to be the first step for any deletion request. This avoids corrupting the more important file system indices, at the price of introducing orphaned, non-externally visible attribute records in *FileStore*. However, this is tolerable by employing a periodical garbage collection procedure running in the background to clean up these benign inconsistencies (see details in Section 4.4). We've listed the execution orders of three representative metadata requests, namely, `create`, `unlink` and `file rename`, in Figure 7. Take the `create` for an example. If a crash occurs after the insertion in *FileStore* completes, upon state recovery, the indices in *TafDB* remain unchanged so that this newly inserted attribute will not be returned in any subsequent accesses.

In addition, CFS co-locates the directory's attribute and the children's id records on a single shard within *TafDB*, due to the combination of the file attribute offloading, the unified index design of `inode_table` and the range-partitioning method. Thus, the vast majority of distributed transactions that used to be present in conventional designs can be transformed into single-shard transactions.

4.2 Fast Single-shard Primitives within *TafDB*

Next, we scale up the single *TafDB* shard's performance by pruning the scope of critical sections of metadata requests with the following techniques.

Single-shard atomic primitives. In Figure 2, most of the metadata transactions touch multiple records, and follow a *read-and-modify* pattern. A naive implementation against *TafDB*'s `inode_table` would span the logics over a few read and write statements, and also include certain consistency checks. Executing such transactions is not efficient as it requires *ClientLib* to perform the individual statement execution in an interactive fashion. To this end, we propose to execute the mutations belonging to a single metadata request all at once via three *single-shard atomic primitives*, which are parameterized functions to perform reads, writes, and conditional checks in a single command, shown in Table 2.

All mutations will be written to *TafDB* if conditional checks are evaluated to `True`; Otherwise, nothing will happen. *TafDB* admits checks written by following SQL-like syntax in a flexible way. First, we allow to specify predicates in the `where` clause of primitives. For instance, for the two important checks, namely, existence check and directory emptiness check, we just need to explicitly specify the primary keys or "children = 0" in the where clause. We also enable to associate predicates with DELETE/UPDATE clause, and conclude if certain conditions are not satisfied when the corresponding state mutation fails. Finally, there is an implicit existence check supported by INSERT, i.e., the unsuccessful insertions with duplicated records indicate that the existence check succeeds. More examples are shown in Figure 8.

The advantages of these primitives are as follows. First, their usage greatly reduces the communication between *ClientLib* and *TafDB*. Second, the lifespan of metadata transactions and lock contentions are further reduced by the fast, single-command *TafDB* execution. Third, their usage also simplifies the design of metadata requests as these primitives can be reused across different requests just by being instantiated with different parameters, rather than being implemented case by case. Note that the idea looks similar to the database stored procedures [33] but with major difference, where stored procedures just simply group statements, which are still executed one by one, while our primitives are single statements and can be made even faster.

Conflict reconciling procedures. However, the above primitives still suffer the locking overhead and are likely serialized by *TafDB* for correct semantics. To further reduce locking overhead, we make another important observation that many metadata requests contain spurious contentions when they concurrently read or modify the attributes of a shared parent directory. Take `create` for an example. Two concurrent file creations under the same directory would both update the `links`, `children`, `size`, and `mtime` fields of the directory record within `inode_table`. However, we

```
-- Insert file attribute into FileStore
FileStore.Put(key=@id, value={links=1, children=0, ...})
-- Insert inode record into TafDB while update parent
INSERT (kID=@parent_id, kStr=@name, id=@id)
WITH UPDATE inode_table
SET (children+=1, mtime=@now, ...)
WHERE (kID=@parent_id, kStr="/_ATTR", type=dir)
```

(a) `create`

```
-- Remove inode from TafDB while update parent
DELETE (kID=@parent_id, kStr=@name)
WITH UPDATE inode_table
SET (children−=1, mtime=@now, ...)
WHERE (kID=@parent_id, kStr="/_ATTR", type=dir)
FileStore.Delete (key=@id) -- Remove file attribute
```

(b) `unlink`

```
-- Remove inodes of A and B, and insert a new inode into
     TafDB while update parent attribute
INSERT (kID=@parent_id, kStr="B", id=@A_id)
WITH DELETE (kID=@parent_id, kStr="A", type=file),
     (kID=@parent_id, kStr="B", type=file, ifexist)
WITH UPDATE inode_table
SET (children−=Δ, mtime=@now, ...)
WHERE (kID=@parent_id, kStr="/_ATTR")
FileStore.Delete (key=@B_id) -- Remove B's attr
```

(c) Intra-directory file `rename`

Figure 8. Code snippets of the file `create`, `unlink`, and intra-directory file `rename` metadata requests within CFS

claim that these conflicting updates on these attributes are always mergeable. In more detail, there are only two types of changes induced on metadata attributes. First, attributes like `links`, `children`, and `size` are all numbers and always incremented or decremented with commutative side-effects, and thus applying these delta mutations in different sequential orders without lock protection will lead to the same final state. Second, for the remaining attributes such as `time` and `permission`, their final values are always determined by the last modifier, specified in a globally agreed total order.

Here, we exclude those spurious contentions from critical sections by enhancing *single-shard atomic primitives* with two conflict reconciling procedures. First, we introduce a *delta apply* merge procedure for merging numerical increments or decrements, where we encode into the primitives with positive or negative delta values and let DB apply these values to database records in any order. Second, for the overwritten attributes, this *delta apply* procedure will no longer work. Instead, we adopt a *last-writer-win* strategy [36], where we rely on the monotonically increasing timestamps assigned by *TafDB* to determine the final values that should be associated with the largest admitted timestamp.

Case studies. Here, we demonstrate the implementation of metadata operations for file creation and deletion by leveraging the above primitives. In Figure 8(a), `create` first executes a *FileStore* metadata operation (line 2) to insert the file attribute key-value pair into RocksDB, followed by the `insert_with_update` primitive towards *TafDB* (line 4-7). This primitive ensures that if the corresponding parent directory still exists, then an inode id record of the new file will be inserted into `inode_table` and the attributes of its parent directory will be accordingly updated. Here, "`children += 1`" and "mtime = now" (line 6) showcase the use of the *delta apply* and *last-writer-win* merge procedure, respectively. Again, the execution of the two insertions will not be protected by any locks. Similarly, the code snippet for file deletion is shown in Figure 8(b), where the `delete_with_update` primitive (line 2-5) is handled by *TafDB* to ensure the atomicity of removing the file inode record from `inode_table` and decreasing the number of children of the corresponding parent directory by 1, followed by a removal statement going through *FileStore* (line 6).

4.3 Strongly Consistent `Rename`

`Rename` is the most complex metadata operation, requiring the strongest consistency, as in the worst case, one such request causes the change of six metadata objects, namely, 4 attribute records and 2 inode id records of either the target files or the corresponding parent directories, possibly spanning across different metadata shards. Although the percentage of `rename` operations is relatively small (Table 1), it still makes sense to optimize their performance as they negatively impact the request tail latency and metadata service scalability. Our solution is motivated by another key observation that in the production workloads, the vast majority (almost 99%) of `rename` requests are touching two files within the same directory. The intra-directory file `rename` can be simple since changes are within a single shard, but the other cases require coordination for correct semantics. Therefore, within CFS, we handle them differently.

Fast path. Here, we optimize the execution path for file `renames` within the same directory by not contacting *Renamer*, but instead directly interacting with *TafDB* and *FileStore* and using the aforementioned `insert_and_delete_with_update` primitive within *TafDB*. This is made possible, as a POSIX `rename` request will trigger a sequence of `lookup` and `getattr` internal metadata operations within CFS to fetch the target inodes, which are further cached at *ClientLib* and sufficient for the client to identify if a `rename` metadata operation follows the fast path.

Figure 8(c) shows the code snippet of such a `rename` request, assuming the source and destination files are *A* and *B*, respectively. Within *TafDB*, the following conditions must be met: (1) *A* must be an existing file, described by the conjunction of predicates in the first pair of parentheses (line 3) in the DELETE clause; and (2) *B* does not exists, or *B* exists and is also a file, both specified by the conjunction of predicates in the second pair of parentheses, plus a keyword "ifexist", which ignores failed deletion that correspond to B's non-existence. When these checks succeed, metadata mutations will be applied as follows. First, it inserts a new file id record, whose composite primary key is <parent_id, "B"> and the id field has the value of *A*'s inode id (line 2). Second, it removes the inode ids of old *A* and *B* (line 3). Third, it updates the attributes of the parent directory (line 4-6). Note that the *delta* applied to `children` is determined by *TafDB* internal, and can be either 0 if one of the file does not exist, or -1 if both existed. Upon the completion of changes within *TafDB*, it continues to delete the file attribute key-value pair of *B* from *FileStore* if existed (line 7). Again, these two phases can be done without coordination as mentioned in Section 4.1.

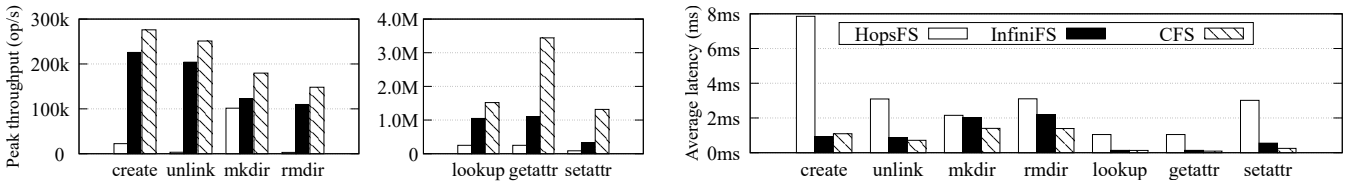
Normal path. The remaining `rename` requests other than the fast-path ones are more complex and demand the following special treatments. The reasons are two-fold. First, they may touch files/directories having different parents that span across different *TafDB* shards. In this case, we cannot use single-shard primitives, instead, we employ a dedicated *Renamer* coordinator to follow the conventional locking and two-phase commit protocol to execute the corresponding sequence of metadata mutations between *FileStore* and *TafDB*. This will guarantee the consistency in the presence of concurrent `rename` operations that go through normal paths.

Second, normal-path `rename` may introduce orphaned loops, not allowed by POSIX semantics, where two directories become each other's parent and their corresponding metadata subtrees are disconnected from the rest of the metadata hierarchy. This can happen when renaming ancestor directories to their descendant directories, or due to uncoordinated concurrent normal-path requests. To preclude this, *Renamer* will first acquire locks of the source/destination directories/files and their most recent common ancestor directory, and then verify if the corresponding request is orphaned loop-free [30].

Finally, the fast-path `rename` operations will not interfere with normal-path ones, due to the following reasons. First, `rename` following fast path only touches files rather than directories, and thus is always orphaned loop-free. Second, CFS offers the strong isolation between single-shard atomic primitives used by fast-path `rename` and the conventional distributed transactions, precluding any inconsistencies.

4.4 Garbage Collection

As both *FileStore* and *TafDB* are protected by the raft fault tolerance protocol, data inconsistencies would only occur when *ClientLib* crashed or was network-partitioned from the rest of CFS' metadata service in the middle of executing metadata requests spanning *TafDB* and *FileStore*. Here, we rely on a light-weight garbage collector to find orphaned records in the two system components for cleaning up. In more detail, the collector watches the write ahead logs of *TafDB* and



(a) Peak throughput (500 clients, higher is better)

(b) Average latency (single client, lower is better)

Figure 9. Peak throughput under high load and average latency under light load of seven metadata requests

FileStore to learn recent metadata mutations, similar to the widely used change data capture service [17], and perform a pairing analysis of the relevant metadata mutations between *TafDB* and *FileStore* to find unmatched/orphaned records.

The collector runs in background, isolated from foreground requests. It has two modes. First, it begins the analysis at a fixed time interval, during which the collector identifies crashed `create` operations if any new attribute records in *FileStore*'s incremental WAL does not have matched inode ids in *TafDB*'s counterpart. For `unlink` and file `rename`, where the *TafDB*'s metadata removal takes place prior to that of *FileStore*, the collector discovers still-existing attributes in *FileStore* of deleted inode ids from *TafDB*. Second, garbage collection can be on-demand. This is applied to handle corrupted `rmdir`, where we recycle orphaned inode ids when `getattr` and `readdir` fail to fetch attribute records.

5 Evaluation

5.1 Experimental Setup

Hardware configurations. We conduct experiments in a local cluster of 50 physical servers, where 40 servers with high-end disks are used to deploy DFSs, while the other 10 with more CPU cores for launching up to 500 concurrent clients. Each DFS server has 32 cores and two 7TB NVMe SSD disks, while each client server has 96 cores. These servers all have 128GB DRAM, and run CentOS 6.3.

Baselines systems. We compare CFS to HopsFS 3.2.0.4 [26] and InfiniFS [44]. HopsFS' backend database is MySQL NDB 8.0. Note that we have to disable the *quota* feature of HopsFS namenode, since it significantly slows down HopsFS due to a bug [15]. As InfiniFS is not made publicly available, we have implemented it by carefully following [44], with verified results showing that the performance trends of InfiniFS and the gap between InfiniFS and HopsFS are consistent with the ones reported in [44].

Deployment configurations. First, to follow the industrial practice, we have a mixed deployment of metadata service and data store on every physical server. Second, for metadata service, the deployment plans for the three systems slightly vary. With regard to HopsFS, each server runs a `Hops-Namenode` process, and a `NDB-Data` process, where the two types of processes comprise the metadata proxy layer and metadata store layer, respectively. Similarly, every server within InfiniFS hosts an Infinifs-MDS process and database-backend process (here is a local *TafDB*). Unlike them, for CFS, as we remove the metadata proxy layer, each metadata server only runs a *TafDB* instance, which collaboratively serves as a distributed database. Additionally, we deploy a 3-node cluster for renaming for both InfiniFS and CFS. To offer high availability, all WAL data of different systems are persisted to NVMe-SSD disks and the replication degree is 3.

Workload configurations. We run the mdtest-like benchmarks [56] to evaluate individual metadata requests with different parameters including contention rates, the number of clients, the directory size, etc. To focus on metadata performance, file operations are tested on empty files like previous work [40, 44, 55]. We also exercise three real-world traces from production environments with data access enabled.

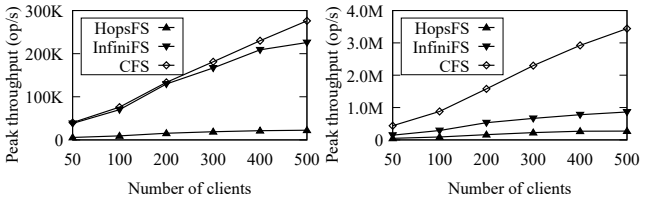
5.2 Overall Performance

We begin with the overall performance assessment of individual metadata requests in the best case, where client workloads do not contain conflicts, and each client reads from or writes to its private directory. Here, we sample seven metadata requests, namely, `create`, `unlink`, `mkdir`, `rmdir`, `lookup`, `getattr`, and `setattr`.

Peak throughput. We deploy 500 concurrent clients to stature three metadata services. In Figure 9(a), CFS performs the best, and introduces speedups of 1.76-75.82× and 1.22-4.10× over HopsFS and InfiniFS, respectively.

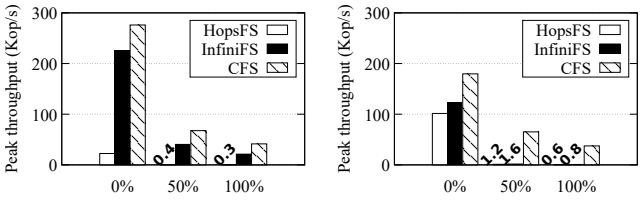
Concerning `create` and `unlink`, HopsFS does not scale due to the heavy distributed coordination. In contrast, InfiniFS and CFS perform almost equally well, but with CFS being 22.32-23.12% better. This is because with no contention, InfiniFS behaves similarly to CFS. For instance, its partitioning method could also reduce the number of distributed transactions, and its locking overhead is constant. However, CFS still wins over InfiniFS due to the resource saved by the critical section pruning and *client-side metadata resolving*, which is then used to enhance the performance of *TafDB*.

For `mkdir` and `rmdir`, the pure directory requests, both HopsFS and InfiniFS do not scale well, mainly because of the heavy distributed coordination. In contrast, thanks to the metadata organization and partitioning method used in CFS, the distributed metadata transactions have been completely eliminated, leading to speedups of 1.76-49.45× and 1.34-1.47×, compared to HopsFS and InfiniFS, respectively.



(a) create (b) getattr

Figure 10. Scalability w.r.t the number of concurrent clients for `create` and `getattr` workloads with no contentions



(a) create (b) mkdir

Figure 11. Throughput of `create` and `mkdir` workloads with 500 clients and 0-100% contention rates

CFS achieves the highest peak throughput, reaching 1.51, 3.44, and 1.31 million requests per second for `lookup`, `getattr`, and `setattr`, respectively. HopsFS achieves the least performance, due to the extra cross-server communication between its metadata proxy and metadata store and the limited scalability of each `NDB-data` node. The best-performed baseline InfiniFS still loses to CFS, due to different reasons. For `lookup`, both InfiniFS and CFS read from *TafDB*, but CFS leverages the *client-side metadata resolving* so that CFS' metadata service obtains more power than that of InfiniFS. Unlike this, the major reason for the gap between the two systems for `getattr` and `setattr` is that CFS further offloads file attributes to a more light-weight *FileStore*.

Average latency. We deploy latency experiments with light loads, i.e., a single client issues requests to the corresponding metadata service. In Figure 9(a), HopsFS delivers highest latency across all tests. InfiniFS reduces HopsFS' average latency by 7.24%-88.16%. CFS achieves comparable or better average latency than InfiniFS for all cases except `create`. Within CFS, `create` requires to additionally persist file attributes to *FileStore*. Therefore, CFS introduces an extra RPC, leading to 17.41% higher average latency. Though the workflow of `unlink` looks similar to `create`, CFS performs almost the same as InfiniFS, since we asynchronously write back to *FileStore*, whose latency can be hidden. As for `mkdir` and `rmdir`, CFS shortens the latency of InfiniFS by 29.87% and 36.55%, respectively, due to the distributed transaction elimination and locking overhead reduction. Furthermore, both CFS and InfiniFS spend the same amount of time executing `lookup`, since they both directly read one metadata object from *TafDB*. Unlike this, CFS outperforms InfiniFS for `getattr` and `setattr` by up to 54.54%. This is because of the faster processing enabled by *FileStore*, compared to *TafDB*.

5.3 Scalability w.r.t Concurrent Clients

Next, we explore in Figure 10 how CFS performs with the increasing number of concurrent clients, from 50 to 500, with no contentions using the same setup as Figure 9(a). Due to space limit, we only show the results of `create` and `getattr`. As expected, CFS' performance scales well with the number of concurrent clients for both workloads. For instance, the throughput of 500 clients is 6.88× of that of 50 clients. In contrast, HopsFS does not scale well, with the increase in the throughput numbers slowing down when adding more clients. For example, compared to the 400 client setup, adding 100 more clients only increase the throughput by 4.87% and 1.36% for `create` and `getattr`, respectively. Again, InfiniFS' scalability is between HopsFS and CFS. It scales well for `create`, but its performance gap with CFS gradually expands as the number of clients increases. For instance, with 500 clients, CFS obtains a speedup of 1.20×, compared to InfiniFS, 11.09% larger than the speedup (1.08×) achieved with 400 clients. Similar to HopsFS, InfiniFS does not scale well to the `getattr` workload, observing no significant improvements when increasing the number of clients from 400 to 500.

5.4 Impacts of High Contention

Here, we investigate the impacts of high contention rates on the system performance of CFS' metadata service. Here, we emulate contention rates via forcing a certain number of concurrent clients to visit the shared files' or directories' metadata. We use three contention rates, namely, 0%, 50%, and 100%, where 0% represents a contention-free workload, while the other two are conflict-intensive workloads.

In Figure 11, all three systems perform the best with no conflicts, while observing worsen throughput with increasing contention rates, as higher contention rates lead the workloads to concentrate on fewer metadata servers. HopsFS' throughput drops remarkably to $0.3K$, while InfiniFS' processing speed reduces to $21.1K$. For the `create` workload, InfiniFS performs better than HopsFS, mainly because InfiniFS reduces the number of distributed transactions.

However, CFS still significantly outperforms HopsFS and InfiniFS. For instance, with contention rates $\geq$ 50%, CFS' throughput is 115.96-177.40× and 1.67-1.96× of that of HopsFS and InfiniFS, respectively, for `create`. This is because CFS allows concurrent `create` requests to run in parallel with trimmed critical sections, and the *client-side metadata resolving* further improves the throughput of metadata service. For the `mkdir` workload, with high contention, CFS' throughput is 55.18-62.42× and 41.52-48.36× of that of HopsFS and InfiniFS, respectively. This is because both HopsFS and InfiniFS adopt 2PC transaction for `mkdir`, while CFS prunes the scope of the critical section and allows `mkdir` requests to run in an almost lock-free manner. In summary, our lock-free design

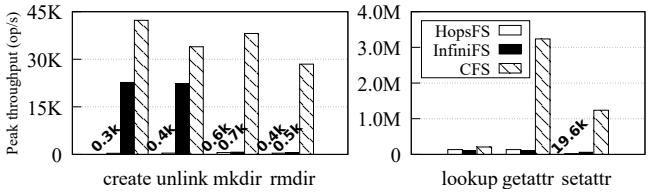
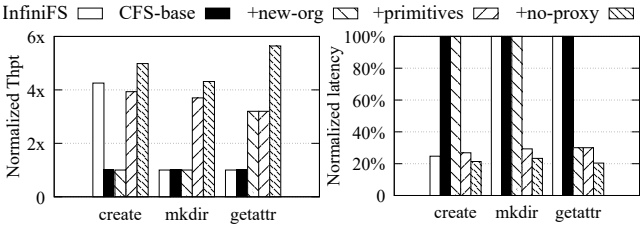


Figure 12. Throughput comparison when 500 clients manipulate a shared, large directory with 1 million files



(a) Throughput (higher is better) (b) Latency (lower is better)

Figure 13. Impacts of CFS' individual optimizations.

enabled by the single-shard primitives and safe isolation relaxation eliminates the heavy lock overhead, and thus scales up the processing capacity of every single metadata shard.

5.5 Large Directory Test

We stress the large directory support of CFS by creating a shared flat directory with one million files, and letting 500 clients to issue metadata requests to the shared directory.

Figure 12 reports the throughput numbers. As for `create`, `unlink`, `mkdir`, `rmdir`, and `lookup`, all three systems' absolute throughput numbers are much lower than those reported in Figure 9(a). This is because the metadata partitioning methods adopted by all systems would group the children's metadata of the same directory, leading to skewed workloads on a single metadata server. This case looks similar to the 100% contention rate test in Section 5.4. However, CFS significantly outperforms baselines mostly due to the coordination/lock elimination. CFS scales well for `getattr` and `setattr`, and introduces speedups of 24.08-63.23× and 20.84-34.19×, compared to HopsFS and InfiniFS. This huge improvement is credited to our novel metadata organization with file attributes offloaded to *FileStore*. This design enables to using hash-based partitioning to evenly balance file attributes among all *FileStore* nodes with high scalability. For comparison, HopsFS and InfiniFS cannot scale out the file attribute workloads as they still use locality-preserving partition methods and suffer the hotspot problem [48].

5.6 Rename Test

We run a workload made up of 90% of intra-directory file `rename` requests and 10% covering all other types of `rename`. With 500 clients, CFS can process 151.3*k* `rename` requests per second as its peak throughput, 252.68% and 63.92% higher than HopsFS' 42.9*k* and InfiniFS' 92.3*k*, respectively. HopsFS leverages a heavy subtree-based locking mechanism, which needs to lock two subtrees for performing `rename` requests, and thus leads to limited scalability. Compared to InfiniFS, the improvements of CFS are contributed by the fact that CFS pays the same overhead as InfiniFS for normal paths, but significantly optimizes the intra-directory ones via the usage of single-shard primitives. The P99 tail latency achieved by CFS is 20.75 ms, 89.89%, and 72.78% shorter than the counterpart of HopsFS and InfiniFS, respectively. Furthermore, CFS delivers a P999 tail latency of 33.29 ms and cuts the baselines' counterparts by 79.00-91.56%.

5.7 Breakdown Analysis

Next, we explore the performance impact of gradually enabling CFS' optimizations, beginning with CFS-base, where all metadata are stored and range-partitioned in *TafDB*, and metadata proxies are deployed to accept requests. "+new-org" refers to a system configuration where CFS-base adds the new metadata organization, presented in Section 4.1, while "+primitives" extends "+new-org" to make use of single-shard primitives (Section 4.2). Finally, "+no-proxy" employs *ClientLib* and evolves to be the full CFS deployment.

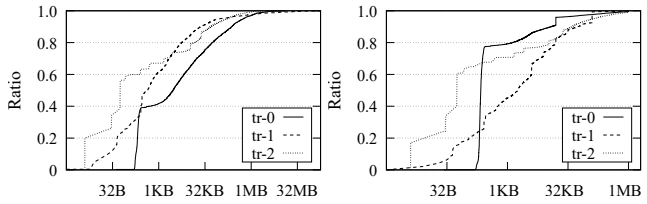
Here, we run experiments with a smaller cluster consisting of 6 servers, 100 concurrent clients, and 10% contention rate. Figure 13 summarizes the peak throughput and average latency numbers normalized to CFS-base. As the above analysis implies that HopsFS always loses to InfiniFS, we only compare performance of all CFS variants with InfiniFS.

For `create`, CFS-base performs much worse than InfiniFS, as CFS-base heavily uses distributed transactions but InfiniFS eliminates. Unlike this, CFS-base performs similarly to InfiniFS for `mkdir` and `getattr`, since both systems have almost identical workflows.

The remaining results indicate that different metadata requests see different optimization sweet points.

"+new-org". Storing file attributes in a separate *FileStore* introduces a 3.19× throughput speedup and a 69.96% latency reduction for `getattr`, compared to CFS-base (InfiniFS), thanks to the parallel request serving capabilities by the scalable *FileStore*. However, this optimization has no effect on the other two requests for different reasons. `mkdir` does not touch *FileStore*. Though `create` needs to write file attributes to *FileStore*, its extra cost is avoided by piggybacking this write on the data block creation.

"+primitives". Using single-shard primitives brings remarkable speedups for `create` and `mkdir` instead of `getattr`, since `getattr` does not involve distributed transactions and locking. CFS +primitives outperforms InfiniFS for `mkdir` by 2.70×, since InfiniFS implements ad-hoc distributed transaction elimination, which is not applied to this type of requests, while CFS' solution is general. In contrast, considering `create`, CFS +primitives almost catches up with InfiniFS



(a) File size distribution (b) I/O size distribution

Figure 14. File/IO size distribution of sampled traces

but with 7.52% lower throughput and 8.63% higher latency. This is because CFS follows common practice and pays extra cost for additionally creating the data blocks, while InfiniFS, as a demo system, does not.

"+no-proxy". Employing the client-side metadata resolving further shortens the latency by 20.42%, 20.38%, and 32.23% for `create`, `mkdir` and `getattr`, respectively, with their throughput accordingly higher, compared to "+primitives". InfiniFS can also leverage *ClientLib* but with minior performance benefits. For single-server requests like `create` and `getattr`, using *ClientLib* shortens the metadata access paths. However, the performance benefits are still constrained by not having CFS's "new-org" and "primitives" optimizations. Furthermore, the client-side metadata resolving brings little improvements for multi-server operations like *mkdir* in InfiniFS, as their major bottlenecks are cross-sever distributed transaction execution. These results prove that within CFS, *ClientLib* is an important optimization whose potentials can be only exploited when the metadata is re-organized and the distributed transactions are eliminated.

Finally, when all the three optimizations are stacked up, CFS achieves 4.31-5.64× and 1.17-5.64× throughput speedups over CFS-base and InfiniFS, while reducing their latency by 13.55-79.64% and 76.66-79.64%, respectively.

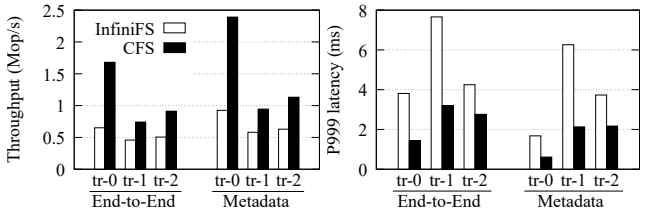
5.8 Production Workload Analysis

CFS has been deployed and running in the Baidu AI Cloud for three years, supporting various applications ranging from social networking, online shopping, enterprise office automation, autonomous driving, to machine learning. Typically, a single CFS deployment often runs thousands of filesystem instances, including metadata and data servers, via a multi-tenant fashion, and scales to billions of files or beyond.

Experience report. We summarize the statistics of three traces, *tr-0*, *tr-1*, and *tr-2*, from the nine production workloads mentioned in Section 2. First, as shown in Figure 14, consistent with other studies [7, 27, 75], the vast majority of files and I/O sizes of the sampled modern cloud applications are extremely small. For instance, 75.27%, 91.34% and 87.51% files are 32KB and below in size for the three traces, respectively. Furthermore, up to 96.37% of I/Os are no more than 32KB with 45.20-70.70% even not exceeding 1KB. These results further imply that the metadata performance is crucial.

Table 3. Operation compositions of three real-world traces

tr	file system operations	metadata operations
0	17.8% `read`, 6.0% `opendir` 51.8% `stat`, 24.4% `open`	95.1% `getattr` 4.9% `lookup`
1	11.6% `read`, 8.2% `write`, 3.1% `open` 8.4% `open(O_CREAT)`, 47.2% `stat` 13.1% `opendir`, 8.0% `unlink` 0.3% `rename`	23.9% `lookup`, 0.2% `rename` 6.5% `create`, 6.1% `unlink` 63.2% `getattr`
2	6.3% `write`, 1.0% `read`, 5.6% `open` 6.2% `open(O_CREAT)`, 49.3% `stat` 6.2% `chmod/chown`, 5.1% `unlink` 19.0% `opendir`, 1.3% `mkdir`	5.0% `create`, 4.1% `unlink` 18.1% `lookup`, 1.0% `mkdir` 66.8% `getattr` 5.0% `setattr`



(a) Throughput (higher is better) (b) Tail latency (lower is better)

Figure 15. Performance comparison between InfiniFS and CFS with three real-world traces and data access enabled

We further report the detailed compositions of the three traces with a division into file system operations (measured as the POSIX function calls) and metadata operations in Table 3. *tr-0* is read-only and its metadata workload has a `getattr`/`lookup` ratio of 95.1% and 4.9%. *tr-1* and *tr-2* are both read-intensive but with substantial writes. For instance, the file system calls in *tr-1* contain 8.2% `write` operations that modify file content, and 12.8% metadata update operations. These calls result in 6.5%, 6.1%, and 0.2% of metadata operations are `create`, `unlink`, and `rename`.

Performance evaluation. We replay these traces with data access enabled, using the same experimental setup in Section 5.2, e.g., with 50 servers and 500 clients, and report the performance comparison between CFS and InfiniFS in Figure 15. We did not deploy HopsFS as it implements HDFS semantics rather than the full POSIX semantics, and only supports file append-only operations, which are not compatible with the random accesses in traces.

First, CFS introduces 2.58×, 1.63×, and 1.80× speedups of metadata throughput over InfiniFS for *tr-0*, *tr-1*, and *tr-2*, respectively. The enhanced metadata service further improves the end-to-end file system performance with 1.62-2.55× speedups. The absolute metadata throughput numbers are slightly higher than those of file systems. This is expected as one file system operation may trigger multiple metadata operations, e.g., `stat` requires `lookup` and `getattr`. Second, we also observe that CFS remarkably optimizes the tail (P999) latency, with 41.82-63.69% and 35.06-62.47% reductions for metadata operations and file system operations, respectively.

tr-1 observes the largest tail latency improvement than the other two traces, as it has more `rename` operations, whose costs were largely trimmed down by CFS.

6 Related work

Metadata decoupling, disaggregation. To scale namespace and data independently, modern distributed file systems have already decoupled metadata from data [22, 42, 46]. To adapt to the fast-evolving workloads with many small files and massive concurrent metadata accesses, the disaggregated metadata architecture has been proposed to decouple metadata serving from its storage. Recent DFSs including CalvinFS [64], HopsFS [48], ADLS [54], Tectonic [52] and InfiniFS [44] have adopted high performance databases like CalvinDB [65], NDB [77], and ZippyDB [5] to scale out/up the metadata storage. Additionally, they employ a separate metadata proxy layer, acting as gateways, to interact with clients. We follow but innovate this disaggregated architecture by removing the metadata proxy layer and offering client-side metadata resolving.

Metadata organization and partitioning. HopsFS [48] organizes metadata as MySQL database tables, while Tectonic [52] and InfiniFS [44] map metadata into key-value pairs. Additionally, Tectonic separates file system metadata into naming, file and block layers. InfiniFS [44] divides directory metadata into access and content parts. These logically separated namespace layers are govern by different indices and mapping methods within the same storage. We further explore the differences between file attributes and the remaining namespace hierarchies, and manage them via a tiered storage consisting of TafDB and FileStore.

To parallelize metadata accesses, BeeGFS [28], Lustre-DNE2 [43], SoMeta [63], SkyFS [72], PVFSv2 [29], HBA [76], CalvinFS [64], and Tectonic [52] hash-partition metadata, at the price of introducing cross-shard coordination. Unlike them, to preserve locality, the subtree-based partitioning scheme has been used by Sprite [50], Coda [58], Ursa Minor [2], and Lustre-DNE1 [43], and CephFS [70], where namespace is divided into subtrees being assigned to different MDSs. Similarly, ADLS [54] adopts range partitioning, and InfiniFS [44] uses the locality-aware grouping to place related data on the same shard. However, they are prone to hotspots. Unlike the above approaches, our CFS considers to simultaneously preserve namespace locality and improve metadata load balance with a tiered metadata storage.

Transaction and consistency. A recent study reveals the POSIX implementation can be error-prone [45]. To this end, HopsFS [48] encapsulates metadata operations into (distributed) transactions using the row-level database locks for isolation protection and the two-phase commit protocol for ensuring atomicity of cross-shard transactions. GlobalFS [51] orders metadata updates by atomic multicast. Tectonic [52] and InfiniFS [44] heavily rely on single-shard strongly-consistent transactions in the local key-value stores, while either dropping out the distributed transactions at the price of introducing inconsistencies, or resorting to the standard two-phase commit protocol for cross-shard operations. Databases like Spanner [16] and CockroachDB [62] offer strict serializable isolation via optimistic concurrency control, and deliver good performance for workloads with low contention. Unlike them, CFS simplifies metadata operations into single-shard atomic primitives without the notion of (distributed) transactions, and safely relaxes the atomicity and isolation requirements but without sacrificing the intended semantics. Thus, CFS could achieve better metadata scalability than these relevant works even for workloads with high contention.

Weaker consistency models have been proposed to safely relax the consistency (isolation levels) of invariant-preserving and commutative operations [14, 38]. Additionally, conflict resolving is another key idea to offer high throughput under workloads with high contention [18], and ElmerFS [67] discusses the usage of CRDT [60] to organize the metadata of geo-replicated file systems. We draw inspirations from this line of work and apply them into the metadata management in DFSs combined with our single-shard atomic primitives.

Other related work optimizes metadata performance with methods ranging from using one-sided RDMA [41], storing metadata in persistent memory [4, 73], to offloading functions to SmartNICs [34]. CFS' focus is orthogonal to that of these hardware-driven solutions.

7 Conclusion

CFS is a scalable, fully POSIX-compliant distributed file system with the primary focus on improving metadata performance. It adopts *a tiered metadata organization* with locality-preserving and load balance-friendly partitioning methods for distributed transaction elimination. Then, it leverages a few *single-shard atomic primitives* with safely relaxed atomicity and isolation. Experimental results show that CFS achieves the best metadata performance, compared to baselines, especially with high contention rates.

Acknowledgments

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Haibo Chen for his thorough guidance in our camera-ready preparation. We also thank Yongqiang Yang for his technical contribution for developing CFS. This work is supported in part by the National Natural Science Foundation of China under Grant No. 62141216, 62172382, and 61832011, the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045, and the USTC Research Funds of the Double First-Class Initiative under Grant No. YD2150002006. Cheng Li is the corresponding author.

References

[1] Cristina L Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H Campbell. Metadata traces and workload models for evaluating big storage systems. In *IEEE Fifth International Conference on Utility and Cloud Computing*, 2012.

[2] Michael Abd-El-Malek, William V Courtright II, Chuck Cranor, Gregory R Ganger, James Hendricks, Andrew J Klosterman, Michael P Mesnier, Manish Prasad, Brandon Salmon, Raja R Sambasivan, et al. Ursa Minor: Versatile cluster-based storage. In *4th USENIX Conference on File and Storage Technologies (FAST 05)*, 2005.

[3] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage (PDSW 11)*, 2011.

[4] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027, 2020.

[5] Muthu Annamalai. ZippyDB: a modern, distributed key-value data store. Data @Scale Seattle.

[6] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys 2016), London, United Kingdom, April 18-21, 2016*, pages 19:1–19:17. ACM, 2016.

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[8] Welch Brent, Unangst Marc, Abbasi Zainul, G Garth, M Brian, S Jason, Z Jim, and Z Bin. Scalable performance of the Panasas parallel file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

[9] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[10] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[11] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *2009 IEEE International Symposium on Parallel Distributed Processing (ISPDC 09)*, pages 1–11, 2009.

[12] Ceph. Differences from POSIX. https://docs.ceph.com/en/latest/cephfs/posix/. Accessed May 16, 2022.

[13] CFS. Posix compatibility of cfs. https://github.com/cfs-for-review/CFS-for-review. Accessed Oct 19, 2022.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4), jan 2015.

[15] Hopsworks Community. The hdfs_quota_update table grows unexpectedly. https://community.hopsworks.ai/t/the-hdfs-quota-update-table-grows-unexpectedly/511/8. Accessed May 16, 2022.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[17] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. All aboard the databus! linkedin's scalable consistent change data capture platform. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC 12)*, 2012.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP 07)*, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[19] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. Ganesha, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, volume 113, 2007.

[20] John R Douceur and Jon Howell. Distributed directory service in the Farsite file system. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI 06)*, pages 321–334, 2006.

[21] FUSE: Filesystem in userspace. https://github.com/libfuse/libfuse. Accessed May 16, 2022.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 03)*, 2003.

[23] Andreas Grünbacher. POSIX access control lists on linux. In *2003 USENIX Annual Technical Conference (ATC 03)*, pages 259–272, 2003.

[24] Apache Hadoop. Hadoop distributed file system. http://hadoop.apache.org. Accessed May 16, 2022.

[25] Apache Hadoop. The Hadoop FileSystem API definition. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/filesystem/index.html. Accessed May 16, 2022.

[26] Hops Hadoop. Hopsfs 3.2.0.4. https://github.com/hopshadoop/hops/tree/3.2.0.4. Accessed Oct 20, 2022.

[27] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, page 199–212, USA, 2014. USENIX Association.

[28] Jan Heichler. An introduction to BeeGFS, 2014.

[29] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pNFS. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27. IEEE, 2005.

[30] IEEE Computer Society and The Open Group. IEEE standard for information technology–portable operating system interface (POSIX) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, 2018.

[31] Intel. Optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dcpersistent-memory.html. Accessed May 16, 2022.

[32] Juicedata. JuiceFS: A better way to use s3. https://juicefs.com/. Accessed May 16, 2022.

[33] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[34] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, pages 756–771, 2021.

[35] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson,

Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 18)*, pages 649–660. IEEE, 2018.

[36] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[37] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *2008 USENIX Annual Technical Conference (ATC 08)*, 2008.

[38] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Conference on Operating Systems Design and Implementation (OSDI 12)*, page 265–278, 2012.

[39] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. LocoFS: a loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 17)*, 2017.

[40] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. CFS: A distributed file system for large scale container platforms. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD 19)*, pages 1729–1742, 2019.

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

[42] Lustre. Lustre file system. http://www.lustre.org. Accessed May 16, 2022.

[43] Lustre. Lustre metadata service. https://wiki.lustre.org/Lustre_Metadata_Service_(MDS). Accessed May 16, 2022.

[44] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An efficient metadata service for large-scale distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, 2022.

[45] Stathis Maneas and Bianca Schroeder. The evolution of the hadoop distributed file system. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 67–74, 2018.

[46] MooseFS. MooseFS – a petabyte distributed file system. https://moosefs.com. Accessed May 16, 2022.

[47] Theofilos Mouratidis. Ceph MDS balancing issues. https://www.spinics.net/lists/ceph-devel/msg44650.html. Accessed May 16, 2022.

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

[49] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (ATC 12), Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[50] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[51] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Rivière, and Pascal Felber. GlobalFS: A strongly consistent multi-site file system. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS 16)*, pages 147–156, 2016.

[52] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's Tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[53] Pjdfstest: File system test suite. https://wiki.freebsd.org/Pjdfstest. Accessed Oct 19, 2022.

[54] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure Data Lake Store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*, pages 51–63, 2017.

[55] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 14)*, 2014.

[56] HPC IO Benchmark Repository. IOR and mdtest. https://github.com/hpc/ior. Accessed May 16, 2022.

[57] Drew S. Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *2000 USENIX Annual Technical Conference (ATC 00)*, pages 41–54, 2000.

[58] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers (TC)*, 39(4):447–459, 1990.

[59] Michael A Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A Brandt, Sage A Weil, Greg Farnum, and Sam Fineberg. Mantle: a programmable metadata load balancer for the Ceph file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 15)*, pages 1–12. IEEE, 2015.

[60] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium (SSS 11)*, volume 6976, pages 386–400. Springer, 2011.

[61] Facebook Open Source. RocksDB. https://rocksdb.org. Accessed May 16, 2022.

[62] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD 20)*, pages 1493–1509, 2020.

[63] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. SoMeta: Scalable object-centric metadata management for high performance computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER 17)*, pages 359–369. IEEE, 2017.

[64] Alexander Thomson and Daniel J Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[65] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)*, pages 1–12, 2012.

[66] Jay Ts, Robert Eckstein, and David Collier-Brown. *Using Samba.* " O'Reilly Media, Inc.", 2003.

[67] Romain Vaillant, Dimitrios Vasilas, Marc Shapiro, and Thuy Linh Nguyen. CRDTs for truly concurrent file systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 21)*, page 35–41, New York, NY, USA, 2021. Association for Computing Machinery.

[68] Stephen R. Walli. The POSIX family of standards. *ACM Stand.*, 3(1):11–17, 1995.

[69] Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. Lunule: an agile and judicious metadata load balancer for CephFS. In *Proceedings of the International Conference for High*

Performance Computing, Networking, Storage and Analysis (SC 21), pages 1–16. IEEE, 2021.

[70] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[71] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC 04)*, 2004.

[72] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC 09)*, pages 1–11. IEEE, 2009.

[73] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

[74] Shuanglong Zhang, Helen Catanese, and Andy An-I Wang. The composite-file file system: Decoupling the One-to-One mapping of files and metadata for better performance. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 15–22, Santa Clara, CA, February 2016. USENIX Association.

[75] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K. Paul, Keren Chen, and Ali R. Butt. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 32(4):918–930, 2021.

[76] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA: Distributed metadata management for large cluster-based storage systems. *IEEE transactions on parallel and distributed systems (TPDS)*, 19(6):750–763, 2008.

[77] MySQL NDB Cluster Reference Guide. https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html. Accessed May 16, 2022.