Appears in the proceedings of the 13th ACM International Systems and Storage Conference (ACM SYSTOR '20)

Supporting Transactions for Bulk NFSv4 Compounds

Wei Su, Akshay Aurora
Stony Brook University
{suwei,aaurora}@cs.stonybrook.edu

Ming Chen
Google
v.mingchen@gmail.com

Erez Zadok
Stony Brook University
ezk@cs.stonybrook.edu

ABSTRACT

More applications nowadays use network and cloud storage; and modern network file system protocols support *compounding* operations—packing more operations in one request (e.g., NFSv4, SMB). This is known to improve overall throughput and latency by reducing the number of network round trips. It has been reported that by utilizing compounds, NFSv4 performance, especially in high-latency networks, can be improved by orders of magnitude. Alas, with more operations packed into a single message, partial failures become more likely—some server-side operations succeed while others fail to execute. This places a greater challenge on client-side applications to recover from such failures. To solve this and simplify application development, we designed and built TC-NFS, an NFSv4-based network file system with transactional compound execution. We evaluated TC-NFS with different workloads, compounding degrees, and network latencies. Compared to an already existing NFSv4 system that fully utilizes compounds, our end-to-end transactional support adds as little as ~1.1% overhead but as much as ~25× overhead for some intense micro- and macro-workloads.

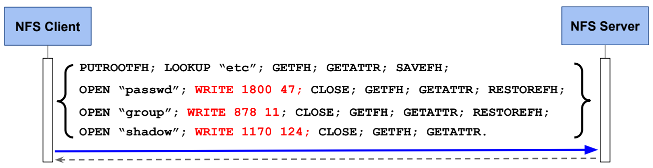
CCS CONCEPTS

• **Networks** → **Network File System (NFS) protocol**; • **Information systems** → **Database transaction processing**.

KEYWORDS

network file system, transaction, NFSv4

ACM Reference Format:
Wei Su, Akshay Aurora, Ming Chen, and Erez Zadok. 2020. Supporting Transactions for Bulk NFSv4 Compounds. In *The 13th ACM International Systems and Storage Conference (SYSTOR '20), June 2–4, 2020, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. *https://doi.org/10.1145/3383669.3398276*



```
PUTROOTFH; LOOKUP "etc"; GETFH; GETATTR; SAVEFH;
OPEN "passwd"; WRITE 1800 47; CLOSE; GETFH; GETATTR; RESTOREFH;
OPEN "group"; WRITE 878 11; CLOSE; GETFH; GETATTR; RESTOREFH;
OPEN "shadow"; WRITE 1170 124; CLOSE; GETFH; GETATTR.
```

Figure 1: An NFSv4 compound with 22 operations that writes three files when creating an UNIX user: `/etc/passwd`**,** `/etc/group`**, and** `/etc/shadow`**. The stateful NFSv4 server maintains two file handles—current (CFH) and saved (SFH)—for use with the various operations. To properly create the user, all three writes (highlighted in red) should succeed.**

1 INTRODUCTION AND MOTIVATION

Over the past several decades, the bandwidth of CPU, memory, storage, and networks have all increased more than a thousand-fold [27]. Taking full advantage of existing high-bandwidth hardware is particularly important as Moore's Law is slowing [38]. For networked storage systems, *compounding*, which packs many I/O operations into one message, is an effective mechanism to reduce network latency. Both NFS [29] and SMB [3] have supported compounds for more than a decade. A recent study [1] showed that I/O-heavy applications can run orders of magnitude faster with a compound-friendly vectorized file-system API.

In that study, Chen *et al.* [1] proposed vNFS, which is an NFS client that offers vectorized file-system APIs to batch multiple file-system operations into one NFSv4 *compound procedure* [30, 31]. It improved performance considerably by amortizing RPC latency among the larger number of operations in a single compound. vNFS improves performance by an order of magnitude, but it indirectly increases the burden on application developers. When applications issue large compounds consisting of multiple file operations, they are faced with a complex error handling problem—if the compound fails mid-way, the applications have to deal with the fact that *some* operations have already changed the persistent file system state. The complexity of handling failures inhibit full utilization of NFSv4's compounds.

Figure 1 illustrates this issue. A vNFS client batches 3 write operations into a single *VWrite* compound, to create a new user on Unix. If the write to `/etc/shadow` fails, NFSv4 servers return to the application with partially completed operations. The application is now responsible for undoing the writes to `/etc/passwd` and `/etc/group` before it can retry. However, another intervening write to the same file offset might have succeeded and thus the application may end up removing a valid user, overwriting an existing entry, or corrupting a Unix file critical for authentication. Figure 1 demonstrates a trivial example, but other data intensive real-world applications also face a problem when they are ported to utilize compounding, such as Hadoop [17, 36, 39], Spark [40] and Powergraph [9].

To address this problem, we added transactional semantics for NFSv4 compounds, thereby reducing the work required to maintain consistent state when leveraging NFS Compounds. We built a transaction layer that layers on top of file-system abstraction in NFS servers. This transaction layer takes care of file-handle management, coordinating concurrent file accesses from multiple clients and restoring consistent state on errors or a crash. We used an embedded database to manage file-handle metadata and store recovery records. We used read-write semaphores to coordinate concurrent requests. We also leveraged Copy-on-Write (CoW) file system semantics to improve performance. We evaluated a prototype implementation using NFS-Ganesha. We show that our system, called *TC-NFS*, is comparable to vNFS in terms of performance for single-client workloads: adding transactional support adds as little as 1.1% overhead to as much as 2.1× overhead—depending on workload characteristics, compounding degree and network latency. In terms of multi-client workloads, TC-NFS's transaction support adds as little as 4.1% to up to 25× of overhead, again depending on the workload's intensity.

2 DESIGN

This section presents TC-NFS's design goals, system overview, components, and implementation details. Although our primary focus is transactional execution of NFS compounds, the key idea of adding transaction semantics to compound execution is also applicable to other network file systems supporting compounds, such as SMB [3].

2.1 Design Goals

Four key design goals guided TC-NFS's design.

(1) Transactional compounds. This is TC-NFS's core feature, designed to make NFS easier to use especially in terms of error handling. All operations in a transactional compound should be executed atomically with proper isolation; the file system state should be always consistent and committed changes should be durable. To achieve this, we need to store recovery data into persistent storage before executing mutating operations. We also need to coordinate concurrent requests and serialize conflicting ones.

(2) Standards compatibility. TC-NFS should be compatible with the NFSv4 standard while providing an extension to execute compounds transactionally. It should also allow non-transactional compounds so clients who do not need transaction semantics do not pay for the associated costs.

(3) Simplicity. Executing many file system operations as one transaction can be difficult. To simplify development we take advantage of existing work on transactions, by delegating transactional execution to a key-value database as much as possible. KVFS [32] already demonstrated that it can be efficient to build a transaction file system on top of a transactional key-value store. Moreover, file system operations may be conflicting (e.g., renaming a directory while writing a file inside). For simplicity, TC-NFS supports only homogeneous mutating NFS compounds: vectorized compounds that contain one or more of the same type of operations that mutate on-disk state. The example compound shown in Figure 1 is homogeneous: it contains the same kind of operation, WRITE. Non-mutating operations such as OPEN, CLOSE, and GETFH are not counted for homogeneity purposes as they do not change data or metadata on the underlying file system.

(4) Low overhead. The overhead of transactional execution should be minimized. For this goal, our key design decision for TC-NFS is to separate metadata management from data management, as this has helped others before. For example, WiscKey [21] showed that separating smaller keys from large values greatly improves the performance of key-value stores. Similarly, a key enabler of KVFS's I/O efficiency was the use of *stitching* to reduce write amplification of data blocks in compactions [32]. Therefore, TC-NFS uses a key-value database for only the metadata and takes advantage of CoW file systems to store data blocks and create backups.

2.2 System Overview

Figure 2 shows how TC-NFS executes a compound as a transaction. The transaction layer is TC-NFS's server core module: it enforces transaction semantics while other modules store metadata or execute vectorized file operations. We use an embedded database for the transaction layer to maintain one-to-one, bi-directional mappings of NFS file handles (NFH, for clients) and local file handles (LFH) provided by the server's backend file system. This layer also coordinates concurrency, data backup, and failure recovery.

For each transactional compound, TC-NFS generates a unique ID that identifies the transaction, as well as its recovery record and backup files. Figure 2 shows an example compound whose transaction ID is `#23`, recovery record is `RR#23` and backup files are in `directory#23`.

The recovery record is for crash recovery and contains the compound's arguments. TC-NFS inserts the record into the

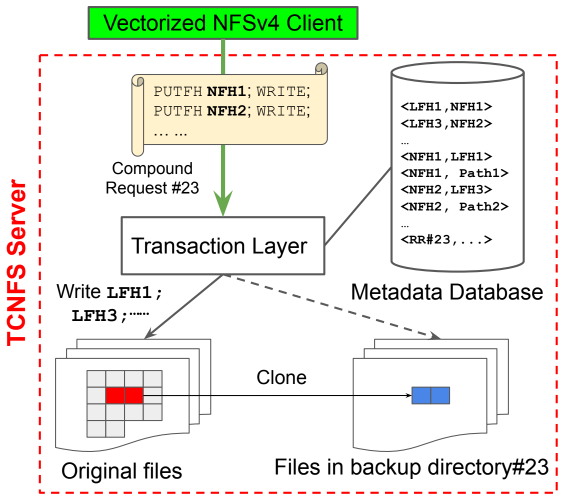


Figure 2: TC-NFS design overview. `LFH` **and** `NFH` **stand for local and NFS file handle, respectively;** `RR#23` **is the recovery record of transaction #23;** `Path` **is the absolute path of the file that the file handle points to. File blocks being written are highlighted.**

metadata database before TC-NFS executes any operation in the compound. If a compound's execution is successful, TC-NFS atomically commits to the database metadata changes together with the removal of the recovery record. When the TC-NFS server recovers from a crash, the transaction layer looks up existing recovery records in the database and rolls back all operations performed by incomplete compound executions before the crash.

Multi-client coordination and locking. To coordinate concurrent accesses from multiple clients or threads, the transaction layer implements a locking mechanism that atomically locks all files involved in each compound request at the beginning of compound execution and unlocks them after the execution ends. TC-NFS cannot directly use LFH or NFH as the key for locks because the files needed to be locked in an operation may not necessarily be the files being operated on directly. For example, in a RENAME operation, we should lock the parent directories of the source and destination files instead of the files themselves. Moreover, in a compound whose operations have dependencies, some relevant files may not exist before the compound is executed. For example, in a transactional compound that creates a directory tree via the sequence of `mkdir /a` and `mkdir /a/b`, the parent directory `/a` does not exist before execution. Therefore, TC-NFS's transaction layer also maintains a mapping between the NFH and the absolute path of the corresponding file. Thus, when locking, the transaction layer calculates the absolute paths of all relevant files from the provided NFH in PUTFH operations as well as path components in other operations in the compound arguments; we then use these paths as the locking keys.

File creation. When creating a file, the transaction layer allocates a globally unique NFH for the new file. Then TC-NFS's backend file system server provides a persistent LFH to retrieve the underlying file. For each new file, the transaction layer creates a one-to-one mapping between the file's NFH and LFH. After finishing all operations in the compound, TC-NFS commits changes of these mappings into the metadata database as a single database transaction, which provides atomicity for the whole compound execution.

NFSv4 creates files using OPENs with a flag equivalent to `O_CREAT`. An existing file may be successfully opened by such an OPEN operation as long as `O_EXCL` is not set. Therefore, during crash recovery, we need to know whether a file was created by an incomplete compound or existed before the compound. The LFH mapping in the metadata database solves this problem: the file pre-existed iff its LFH is in the database. This explains why the transaction layer maintains a mapping between NFH and LFH instead of directly using the LFH given by the backend file system as the unique identifier of a file—we need to query the mapping in rollback or recovery cases.

File handle translation. The transaction layer also performs necessary translation between NFH and LFH for PUTFH, LOOKUP and GETFH operations. For PUTFH, it receives the NFH from the client and queries the LFH in the metadata database, and finds the file in the backend file system. For LOOKUP, TC-NFS first looks up in the backend file system to get the LFH, then finds the corresponding NFH in the database, and finally returns it to the client in subsequent GETFH operations.

Backing up. Before executing mutating operations including WRITE, REMOVE, and OPEN with truncation, TC-NFS's transaction layer creates backups for the files to be operated on. Backup files for a compound are stored in a common backup directory named after the compound's transaction ID. To minimize overhead, TC-NFS backs up files using a Copy-on-Write (CoW) range-cloning mechanism: only the data blocks that will be overwritten are *cloned* into the backup files, saving space and overhead. Figure 2 shows this, where the red color blocks denote the data to be overwritten and the transaction layer clones these blocks into the backup file as the blue blocks. For REMOVE operations, the transaction layer renames the file to be deleted into the backup directory; for OPEN operations that truncate existing files, TC-NFS clones the entire file into the backup.

Rollback on failure. When the compound execution ends, TC-NFS's transaction layer checks the execution status. Upon any error, TC-NFS reverses all the operations that succeeded prior to the error. For example, to reverse CREATE it removes

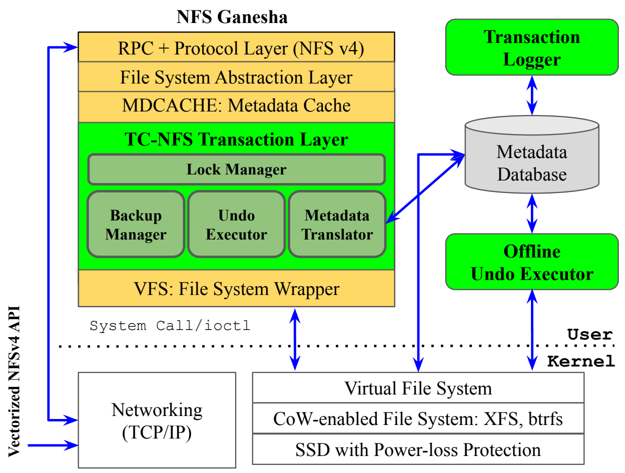


Figure 3: Components of TC-NFS server prototype. The blue arrows show the data path. We added three server-side components, shown in green: TXNFS, transaction logger, and the offline undo executor. The rest are existing components.

the created file, and to reverse WRITE it clones the data in backup files back into the source file.

2.3 System Components

We designed TC-NFS on top of NFS-Ganesha [4], an open-source user-space NFS server; Figure 3 presents its basic components and data paths.

Transaction layer. We developed TC-NFS's transaction layer as a stackable file system abstraction layer (FSAL). This layer has four major components that we describe next: a lock manager, a backup manager, an undo executor, and a metadata translator.

The *lock manager (LM)* coordinates concurrent compound operations from multiple clients or threads by locking relevant files inside a compound. At the beginning of compound execution, LM scans through the compound arguments to calculate the absolute paths of all relevant files that involve in the compound operations and should be locked. For PUTFH operations, LM queries the metadata database for the absolute path associated with the provided NFH and updates LM's "current path." For other operations, such as LOOKUP and OPEN, LM joins the "current path" with the path components in the operation arguments. See Figure 1: when evaluating LOOKUP, LM joins the current path "/" with the LOOKUP's argument "etc", updating the current path to "/etc"; LM then adds the "current path" along with the read/write property to a list. Here, CREATE is a writing operation while READDIR is a reading one. LM also sorts all paths in a compound in lexicographical order to avoid deadlock from competing compounds. Finally, the lock manager attempts to lock the sorted paths atomically using read-write semaphores. If there is more than one compound that contains operations accessing the same file, the conflicting compounds will be serialized by the lock manager to ensure isolation of compound transactions. LM unlocks these paths atomically when the compound execution ends.

The *backup manager* manages backup files and directories. Before executing a mutating file operation, it creates a backup so that the file can be recovered if the enclosing transaction aborts. To minimize overhead, it leverages the underlying file system's CoW mechanism and uses the `ficlonerange` ioctl to back up only the file range being changed. When a compound is successfully committed as a transaction, the backup manager schedules backup files to be cleaned up later by a background process; this also reduces latency overhead of transactional execution.

The *undo executor* rolls back all preceding operations in a compound upon a failure. If one of the operations in the compound fails with an error, execution stops and the undo executor starts to roll back. The undo executor iterates over the compound's arguments in reverse order, starting from the last successful operation, all the way to the first operation, and undoes these operations sequentially. For example, to undo a CREATE operation, it removes the created file; to undo a WRITE operation, it clones the data from the backup file into the original file at the exact position specified by the WRITE argument; and to undo a REMOVE operation, it renames the backup file back to its original name.

The *metadata translator* manages the one-to-one bidirectional mapping between the NFS (NFH) and the local file handle (LFH), as well as the mapping between NFH and the absolute file's path. To keep track of the absolute path of the file that each NFH points to, the metadata translator joins the relative path read from FSAL's metadata and the root path of the NFS export, which is a known string from the config file. When the server starts, the root path is assigned to the root NFH. When performing lookups from the root directory, the metadata translator joins the root path with the path component names provided in the arguments to build absolute paths for these looked-up file handles. The same process runs for other operations such as `open`, `mkdir`, and `readdir`, thus gradually building absolute paths for all file handles as the clients create and access the files in the TC-NFS server.

When creating a file, the metadata translator generates a unique NFH: it is used as the NFS file handle for clients to identify this file. The file's NFH is associated with its local file handle (LFH) provided by the backend file system. The mappings between LFH and NFH and between LFH and the file's absolute path are inserted into the TXN buffer for eventual database commit. When a file is renamed, the metadata translator inserts the file's NFH along with the new absolute path into the TXN buffer (described below), requesting an update of the absolute path in the metadata database. When deleting a file, the metadata translator inserts into the TXN buffer a negative entry that has the file's NFH as the key and NULL as the value, notifying the transaction layer to remove the file from the metadata database when the compound transaction commits. The metadata translator uses the bidirectional mapping to translate between the two types of file handles: for a LOOKUP, it converts the LFH from the server's backend file system to the corresponding NFH for NFS clients; for a PUTFH, it translates the NFH provided by clients into the corresponding LFH to retrieve the actual file.

TXN buffer. TXN buffer is a contiguously allocated vector initialized at the beginning of a compound's execution; each vector element contains the corresponding NFH, LFH and a pointer to the absolute path of the file associated with the NFH. This data structure buffers insertions and deletions of file handle mapping that resulted from file creations and deletions, and path changes made by renaming operations. At the end of the compound's execution, TC-NFS combines these insertions and deletions into a single database transaction. This not only reduces overheads by consolidating database changes, but it also ensures atomic execution of compounds (see Section 2.2). The TXN buffer, as a vector, runs faster than hash tables or linked lists because it is a small contiguous memory block, which helps data locality in CPU caches.

Transaction logger. This is an auxiliary component outside of the FSAL layer (see Figure 3). It gives each compound a unique, monotonically increasing 64-bit integer as the *transaction ID.* It then serializes the compound's request arguments using Protobuf [10], to serve as the recovery record, and writes the record into the metadata database. The transaction logger creates and commits recovery records only for compounds that contain mutating operations such as WRITE and REMOVE. This eliminates overheads that could be introduced by creating recovery records for non-mutating operations because there is no need to roll back such operations. Note that transaction IDs need to be unique during only one incarnation of a TC-NFS server (i.e., one continuous run without restart). Upon restart, a TC-NFS server needs to undo all partial compounds, if any, and then it starts using transaction IDs from zero.

Offline undo executor. This is another auxiliary component outside of the FSAL layer (see Figure 3). It is called every time the server starts. It checks to see if there is any existing recovery record in the database. Since a recovery record is removed when the compound's execution finishes, any remaining recovery record indicates a compound's execution was interrupted due to an unexpected server crash. In that case, the offline undo executor examines the protobuf-encoded recovery record and reverses those operations that did succeed, by calling the right system calls.

3 IMPLEMENTATION

We implemented a prototype of the TC-NFS server in C and C++ on Linux, based on NFS-Ganesha [5]. We used libuuid [20] to generate UUIDs as NFH in the metadata translator; we assume that a UUID uniquely identifies each file considering the low probability of collisions [2]. As our NFS file handles (NFH), we chose to use the UUID instead of the server's local file system's handle (LFH) for the following two reasons: (1) As is explained in Section 2.2, we need to check a file's pre-existence using the mapping between NFH and LFH. (2) The UUID is a simple and fixed-length data type; using it as the NFH simplifies the implementation of the TXN buffer and is more efficient. Conversely, the local file handles provided by the backend file system are variable length and depend on the type of the file system. We used LevelDB [19] as the metadata database and XFS [34] as the server's backend file system. LevelDB uses less space [6] than RocksDB [8] and LMDB [13], and is faster than SQLite3 [11, 14]. We also tested and found that XFS provides better I/O performance than other popular CoW-based file systems (i.e., BtrFS [26]). To offer clients an option whether to execute compounds with transaction semantics, we used the most significant bit of the NFSv4 procedure number; when the bit is not set, TC-NFS acts as a standard NFSv4 server.

Hooks. To ease the development of the transaction layer in NFS-Ganesha's workflow, we added three hooks into its FSAL export operations: **(1)** `start_compound` is called before a compound executes, it initializes the necessary data structures including the TXN buffer and invokes the transaction logger to create and commit a recovery record. Locking is also performed in this function. **(2)** `end_compound` is called after the compound's execution has ended. Here we check the status of the compound's execution to decide whether to rollback. We then delete the recovery record from the database and clean up data structures created in `start_compound`. Finally, we unlock relevant files in this compound to let other conflicting compounds in. **(3)** `backup_nfs4_op` is invoked before each compound operation execution; it backs up the file to be operated on using the backup manager.

Asynchronous backup cleanup. The backup manager removes the backup files after a compound's execution ends. To reduce overhead, we clean up asynchronously: in `end_compound` we submit the compound's transaction ID to the cleanup thread's message queue. The cleanup thread removes backup files asynchronously; this reduces the time to process a transactional compound because we defer the cleanup that otherwise would have taken place in `end_compound`.

Improving durability and fsync performance. Commitment of recovery records and file handles into the transactional database is important to maintain transaction semantics; such operations, however, can be expensive due to `fsync` [12] calls to persist the data. We chose to use an enterprise-grade SSD with power-loss protection (i.e., a capacitor-backed cache) as the backend storage device for two reasons: (1) such an SSD can write much faster when writes just have to go to its internal RAM, and (2) the capacitor ensures that writes are not lost in the event of abrupt power loss.

Code size. Our TC-NFS prototype server adds 5,700 Lines of C code (LoC) into NFS-Ganesha for implementing TXNFS, transactional hooks, and the lock manager. The transaction logger, offline undo executor, and the lock manager add 5,321 C++ LoCs, including 1,831 LOCs of tests. We also added 2,099 C++ LoCs into for testing the FSAL layer in NFS-Ganesha.

Limitations. Our prototype currently fully supports transaction semantics for the following operations: OPEN, WRITE, REMOVE, CREATE, and LINK. In terms of RENAME operations, we provide transaction support for simple cases in which only files are renamed. Renaming directories can be more complex and is subject to future work.

4 EVALUATION

We validate TC-NFS's transaction semantics and evaluate its performance using the prototype we implemented. Our testbed consists of three identical machines running Ubuntu 18.04 with Linux kernel v4.15. Each machine is equipped with a six-core Intel Xeon X5650 CPU, 64GB of RAM, and an Intel 10GbE NIC. One machine acts as the NFS server and the other two as clients. The NFS server exports to the client an XFS file system, stored on an Intel DC S3700 200GB SSD. To test multi-client workloads, we installed four KVM virtual machines running Ubuntu 18.04 on each client machine, so there are 8 clients in total. These machines are connected with a 10GbE switch, and we measured an average RTT of 0.2ms between them. To emulate different network latencies, we injected delays of 1–30ms into the outbound link of the server using `netem`. The test NFS export uses the default options: the attribute cache (`ac` option) is enabled and the maximum read/write size (`rsize`/`wsize` options) is 1MB. For each experiment, we report the average measure of 16 runs excluding a preceding warm-up run.

We designed a series of test cases to verify the transaction semantics when TC-NFS executes compounds with transaction support enabled. The test cases, as well as their expected outcomes are described next. We used multithreading to simulate multiple clients.

Atomicity tests. We issue a compound that contains a sequence of NFS file operations, with a forced invalid request mid-way to cause an error. We expect that the partially executed compound is rolled back entirely (i.e., file system state is rolled back to exactly where it was before the compound began executing). Our tests cover all NFS operations for which TC-NFS provides transaction support.

Serializability tests. These tests check if TC-NFS can serialize and isolate concurrent compound requests from multiple clients. There are two test cases: (1) Files and directories creation and removal. One writer thread constantly creates a set of files or directories using one transactional compound, and several reader threads check for the existence of the files or directories concurrently. We expect that the reader threads will see either all files or directories exist or none of them exist, but they will never detect partial existence. (2) File reading and writing. Several writer threads write some data to a file in parallel. Each writer thread writes data of some fixed content, and the data every thread writes is different. A number of reader threads concurrently read the file to check its content. We expect that the content of the file matches the data of one of the writers writes, and it should not be a mixture.

We performed the aforementioned test cases using the vNFS client and TC-NFS server. All tests passed as expected. When using the vanilla NFS-Ganesha as the server, the test cases failed as expected due to a lack of transaction support.

Performance benchmarks. We benchmarked TC-NFS with micro- and macro-workloads. We experimented with both the in-kernel NFS client and open source vNFS client; we reproduced those results—that the vectorized NFS client significantly outperforms the in-kernel NFS client regardless whether the NFS server is transactional or not. Since TC-NFS focuses on transactional execution of compounds on the server-side, all figures shown here consistently use the same vNFS client and compare TC-NFS's transactional server with a non-transactional *baseline*—the vanilla NFS-Ganesha server. To the best of our knowledge, TC-NFS is the first system that adds transactional compounds support to the network file system, and thus we did not find any counterpart system for TC-NFS to compare with. Results of single-client benchmarks are shown in terms of *Transaction Slowdown*, which is defined as the ratio of the workload's runtime on TC-NFS to that on the vanilla NFS-Ganesha server. Those of multi-client benchmarks are presented as *Relative Throughput*, which is the ratio of the workload's total throughput on TC-NFS to that of the baseline. We used relative throughput as results for multi-client benchmarks because we fixed the runtime of each experiment run in those benchmarks to 30 seconds, in order to have all clients saturated with the test workloads. It is more reasonable to count total bytes operated on and calculate the overall throughput. In any 3D figures, the Z axis is vertical.

4.1 Micro-workloads

4.1.1 Write files. To evaluate TC-NFS's transaction overhead, we compared its performance against the vanilla NFS-Ganesha server (baseline) on a workload of fully writing 1,000 fixed-sized files; we varied the file size from 1KB to 16MB in powers of 2, the network latency from 0.2ms to 5.2ms and the number of clients from 1 to 8.

Single client. Figure 4 shows the results of the benchmark when there is only one client. TC-NFS had to synchronously write a recovery record into LevelDB, lock the files to be written, and create backup files before writing; therefore it performed slower than the baseline, especially when the file size was small. The worst case occurred when the file size was 4KB and the latency was 0.2ms: the relative runtime was 1.55× (an overhead of 55% compared with the original NFS-Ganesha server). TC-NFS's overhead dropped when the file size grew larger and the network latency increased because the time spent on data writing and network transmission became dominant. When the file size was 1MB or larger, the overhead was less than 12%; This suggests that we can reduce the overhead of transactional execution by packing more data to write in one compound.

Multiple clients. To evaluate the multi-client performance and scalability of TC-NFS, we ran the same workload on multiple clients. In this experiment, we distributed the 1,000 files evenly among the clients and repeated the workload until it ran for at least 30 seconds. Figure 5 shows the results of the multi-client benchmark.

As the number of clients increased, the relative performance of TC-NFS dropped dramatically especially for small files (≤ 256KB). The worst-case performance was when the file size was 1KB with 8 clients: TC-NFS's performance was only 3.8% of the vanilla NFS-Ganesha server (25× overhead). For small files (≤ 128KB), the average relative throughput

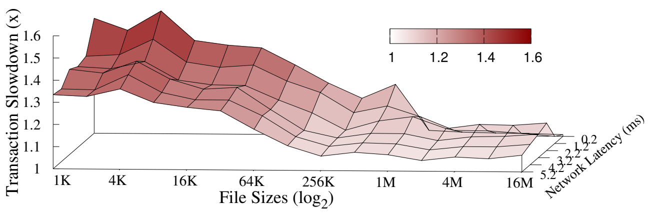


Figure 4: TC-NFS's relative performance when writing 1,000 fixed-size files of 1K–16M. Transaction slowdown is defined as the ratio of TC-NFS's completion time to the vanilla NFS-Ganesha server's completion time (higher Z values are worse).

was 0.19 (i.e., 4.3× overhead). This is because the synchronous I/O and backup creation required for transaction writes significantly limits TC-NFS's ability to scale with the number of concurrent clients, whereas the vanilla NFS-Ganesha server scales well because it does not enforce transactional semantics. Figure 6 compares the scalability of TC-NFS and the vanilla NFS-Ganesha server. TC-NFS failed to scale its write throughput with the number of clients. Worse, we witnessed performance declines on TC-NFS when there were more than 4 clients. We analyze and discuss this issue below. Figure 5 indicates that when the file size exceeded 256KB, TC-NFS's throughput approached that of the vanilla NFS-Ganesha. However, that was because the throughput of large-size writing was capped by the backend storage hardware (the SSD) and unable to scale on both TC-NFS and vanilla NFS-Ganesha (see "VFS, 1M" and "TXNFS, 1M" lines in Figure 6).

Local workload simulation. To understand this performance bottleneck, we wrote a C program to simulate the Writefiles workload locally. We ran two workloads: *interleaving-backup* and *no-backup*. Both workloads write 1,000 equally sized files repeatedly for at least 30 seconds. `fsync` is called after each write to match the behavior of the simulated workloads with that of the NFS server. Data is written to the SSD that TC-NFS's server used as the backend storage in our tests. The workloads may write the data in parallel using multiple

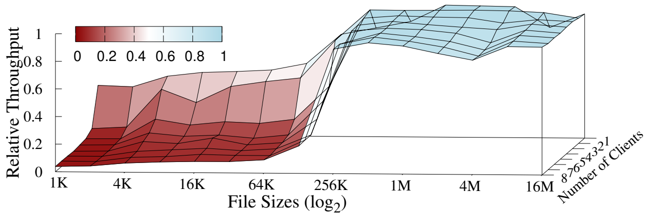


Figure 5: TC-NFS's relative performance when multiple clients write 1,000 fixed-size files in parallel. Relative throughput is defined as the ratio of TC-NFS's throughput to the vanilla NFS-Ganesha server's throughput (higher Z values are better).

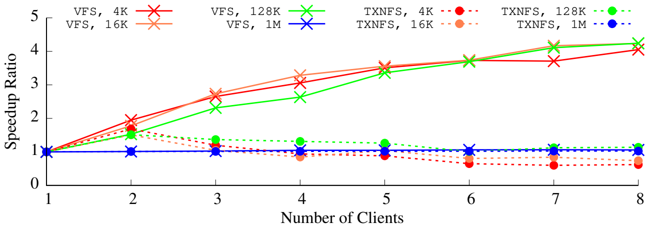


Figure 6: Speedup ratio of TC-NFS and vanilla NFS-Ganesha server as the number of clients increases. VFS denotes the vanilla NFS-Ganesha server. For brevity, we only show a few representative file size (higher Y values are better).

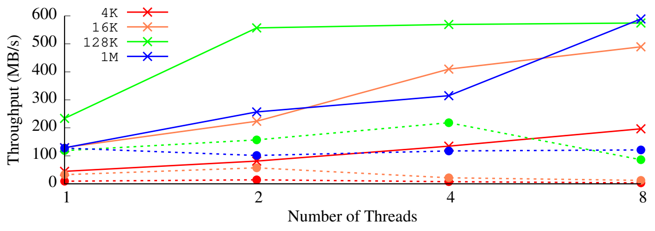


Figure 7: Throughput of the locally simulated Writefiles workload. Solid lines show the results of the "no-backup" workload; dotted lines show the "interleaving-backup" workload (higher Y values are better).

threads, simulating the multi-client case. The only difference is that the interleaving-backup workload utilizes XFS's CoW cloning to create a backup for the target file before each data write, whereas no-backup does not create any backup. The two workloads mimic the internal workflow of TC-NFS and the vanilla NFS-Ganesha server, respectively.

We tested the two workloads using 1, 2, 4, and 8 threads; and we varied file sizes using 4K, 16K, 128K, and 1M. Figure 7 shows the results of the simulated Writefiles workloads. By comparing the throughput of the two workloads for the same file size (i.e., the solid and dotted lines of the same color in Figure 7), we show that, on average, the interleaving-backup workload was 4.6× slower than the no-backup workload. We also repeated the same experiment on BtrFS; the results were worse: Btrfs's throughput was 5× *worse* than XFS.

This experiment shows that the backup creation in TC-NFS's transaction layer is the main reason for the performance bottleneck, and hence restricts TC-NFS's scalability, especially with more than 4 clients (as per Figure 6). In sum, performing CoW cloning and synchronized file write is slow on XFS and BtrFS. If the CoW feature gets optimized in these local file systems (outside the scope of this work), TC-NFS's performance and overall transactional write throughput will improve too.

Non-transactional workload. To improve performance, TC-NFS lets clients disable transaction support for compounds that do not need transactional semantics. Here, the transaction layer does not create backups or recovery records, but it still locks files and commits their metadata changes into the database to ensure the transactional semantics of *other* transactional compounds.

We repeated the experiment *without* enforcing transaction semantics when writing the files. Figure 8 shows TC-NFS's relative performance when writing files without transactions, compared to the vanilla NFS-Ganesha server. The average relative throughput of writing small files was 0.7, indicating an average of 43% overhead. In the worst case, with 16KB-large files and 3 clients, the relative throughput was 0.56. Still, performance in other cases was better: with more than 6 clients, relative throughput numbers in most cases exceeded 0.7.

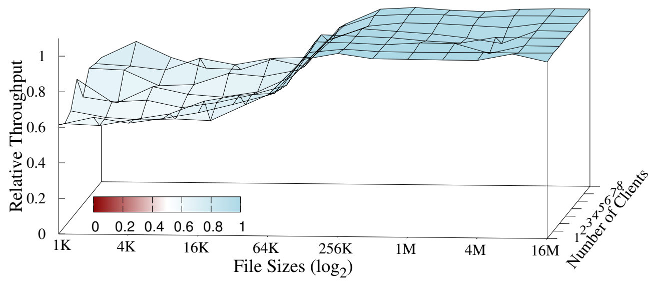


Figure 8: TC-NFS's relative performance when clients running the Writefiles workload chose *not* to use transaction support (higher Z values are better).

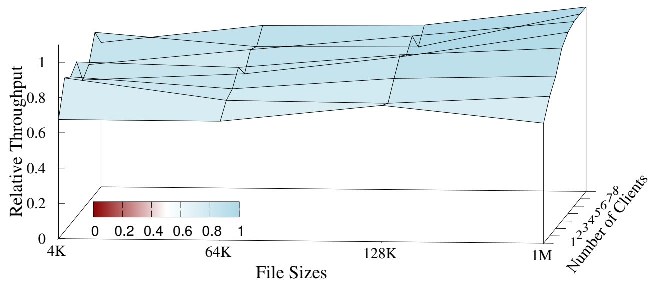
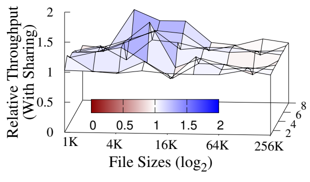


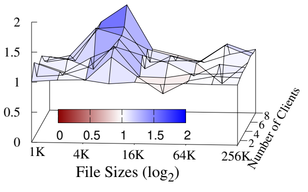
Figure 9: TC-NFS's relative performance when Ramdisk is used as the backend storage system (higher Z values are better).

In this non-transactional experiment, TC-NFS's relative performance did not drop as the number of clients increased, suggesting that TC-NFS's write throughput scales well with the number of clients in this setting. This further confirms that the reason for the performance bottleneck seen in Figure 5 is that XFS's slow CoW cloning.

Ramdisk. We also experimented with the Writefiles workload on Ramdisk. We created a disk image file on `tmpfs`, formatted the image file with XFS and mounted it as the backend storage of TC-NFS and the vanilla NFS-Ganesha server. We ran the Writefiles workload on both TC-NFS and the vanilla NFS-Ganesha server with the same sets of number of clients and file sizes settings; on TC-NFS we enabled transaction support. Figure 9 shows that using Ramdisk, TC-NFS's performance relative to the vanilla server increased significantly: the average relative throughput of writing small files was 0.86 (versus 0.19 on SSD). This is because the time needed for `fsync` and CoW cloning is greatly reduced. Although Ramdisk is an unlikely practical backend storage, this experiment demonstrates the throughput *possible*; it confirms the source of overheads seen in Figures 5 and 6: slow synchronous I/O and CoW operations on SSD.



(a) 5%



(b) 20%

Figure 10: TC-NFS's relative performance when there are shared files among clients that execute the Writefiles workload. Here relative throughput is defined as TC-NFS's write throughput when there are shared files to that when each client write independent set of files (higher Z values are better).

Multi-client with shared files. TC-NFS uses the lock manager to coordinate concurrent compound execution and serializes conflicting compounds. To test TC-NFS's performance when multiple clients share some files, we ran the Writefiles workload on TC-NFS with the number of clients and file sizes, but clients share a fraction of files to write. We define the sharing degree as the percentage of files in a client's task list that are shared by all clients. For example, if there are 5 clients, each writing 200 files, a 20% sharing degree means that 160 files are unique to each client and 40 files are common and written by all clients. Files are accessed in random order.

Figure 10 shows TC-NFS's relative performance when the sharing degree is 5% and 20%. We believe that 5% and 20% of sharing degrees are representative settings for real-world workloads. According to Leung *et al.* [18], in a real-world corporate data center, up to 16.6% of files get shared by 2 clients concurrently; 7.3% of files are shared by more than 2 clients in the corporate data center whereas only 0.3% of files are shared by over 2 clients in the engineering data center. Thus, we can regard a 20% sharing degree as an extreme case and 5% as a more common case.

Here, the baseline is TC-NFS's write throughput when running the Writefiles workload without any file sharing among clients. A surprising discovery seen in Figure 10 is that the performance of Writefiles with shared files was *better* than without sharing. When the sharing degree was 5%, the throughput of Writefiles with file sharing was 6% higher than the baseline on average for small files (i.e., ≤ 128K); with a sharing degree of 20%, Writefiles with shared files was on average 9% faster for small files. When the sharing degree was 20%, there were 8 threads, and the file size was 8K, the relative throughput was the highest and reached 1.77, meaning 77% faster than the non-sharing workload.

While counter-intuitive, this is explained in Figures 6 and 7: TC-NFS's total throughput increased when fewer clients wrote files in parallel with transaction support. When clients write to the same files, some compounds are serialized.

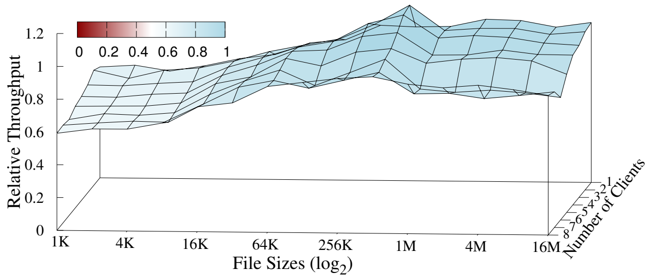


Figure 11: TC-NFS's relative performance when multiple clients read 1,000 fixed-size files in parallel. Relative throughput is defined the same as that in Figure 5 (higher Z values are better).

This serialization reduces parallelism, which reduces contention in XFS's CoW operation, and hence overall throughput *increases*. When the files were greater than 128K, the relative throughput was close to 1.0. This is because NFS-Ganesha's RPC layer limits the size of a compound to 1MB. Therefore, when the write size is large, fewer WRITE operations fit in one compound. Because TC-NFS executes each compound as a transaction, access to a list of files with conflicting ones may not be serialized as these WRITE operations are distributed across a larger number of compounds, each of which only contains a few operations.

4.1.2 *Read files.*

We also compared the throughput of TC-NFS with the vanilla NFS-Ganesha server to read 1,000 files entirely, using the same set of file sizes and number of clients; we did not inject additional network latencies; see Figure 11. The overhead of Readfiles was smaller. For small files (≤ 128KB), the average relative throughput was 0.74 (35% overhead). The worst case was when the file size was 1KB and there were 6 clients: relative throughput was 0.58 (72% overhead), much better than the Writefiles workload. This was because TC-NFS did not create backups or recovery records for READ operations; therefore the overhead was smaller and only came from looking up file handles in LevelDB and locking the target files. Without backup creation, TC-NFS's read throughput was able to scale normally with multiple clients, and its relative throughput to the vanilla NFS-Ganesha server was stable regardless of the number of clients.

4.1.3 *Compounding degree.*

To amortize network and I/O latency, it is desirable to write a large number of NFS files at once, but in practice, this is not always possible. To study the potential benefit, we varied the number of files in each compound as well as the file size and compared the time taken by TC-NFS and the vanilla NFS-Ganesha server to serve 1,000 compounds. We did not inject additional network latency in this experiment. Figure 12 shows that when writing one 1KB file in each compound, TC-NFS performed the worst and was 3.1× slower than the vanilla NFS-Ganesha server; the

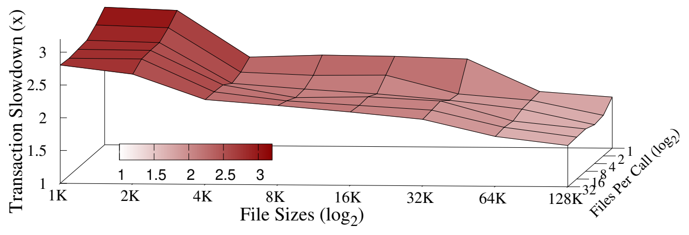


Figure 12: TC-NFS's relative performance when writing a different number of equally sized files, ranging from 1K to 128K. No extra latency added. The Y axis shows the number of files written per compound (higher Z values are worse).

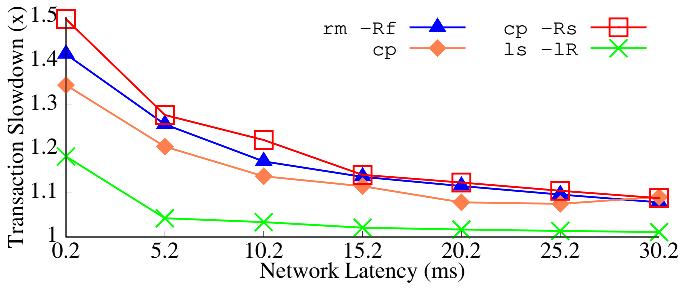


Figure 13: TC-NFS's runtime relative to the non-transactional baseline when symbolically copying (`cp -Rs`), listing (`ls -Rl`), and removing (`rm -Rf`) a Linux source tree. The Y-axis is logarithmic (higher Y values are worse).

overhead dropped to 63% when writing 32 × 128KB files in a single compound request. Therefore, compounds with fewer operations can add significant overhead due to TC-NFS's transactional nature; we can lower this overhead by packing more operations and more data to write in a compound.

4.2 Macro-workloads

To evaluate TC-NFS using realistic applications, we ran workloads from GNU Coreutils and BSD Tar. We compare TC-NFS's with the non-transactional NFS-Ganesha server while using the vNFS client for both.

GNU Coreutils. We used the ported Coreutils programs to list, copy, symlink-copy and then remove the Linux-4.20.7 source tree: it contains 62,447 files with an average size of 14.9KB, 4,148 directories with average 15 children per directory, and 35 symbolic links. Figure 13 shows the relative runtime of recursive-listing (`ls -lR`), copy (`cp -R`), symlink-copy (`cp -Rs`), and recursive-removal (`rm -rf`) on TC-NFS compared with the vanilla NFS-Ganesha server.

When the network latency is 0.2ms, symlink-copy and recursive-removal added 49.5% and 41.5% overhead, respectively. Regular copy had a lower overhead of 34.5%. Metadata-intensive workloads had higher overhead because the time used for these operations themselves (such as CREATE and REMOVE) was short and extra operations added by transaction support account for a larger part of the total runtime. Recursive-listing had the smallest overhead of 18.3% because it was not a mutating workload and TC-NFS did not have to commit anything to the metadata database. Its overhead came only from locking the directories `ls -lR` read. As expected, when network latency increased, overheads decreased because the time used for network transmission started to dominate. When the latency was 30.2ms, the overhead of all three workloads dropped below 10%. The overhead of `ls -lR` dropped below 5% when the latency was greater than 5ms, and got as low as 1.1% when latency was 30.2ms.

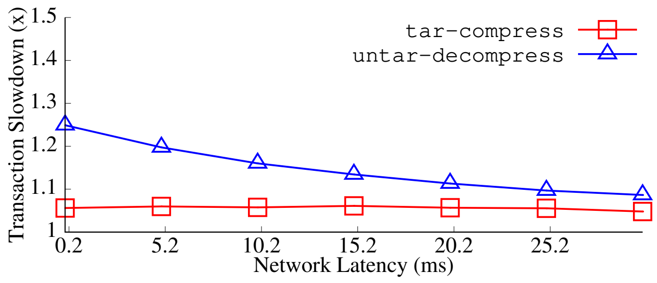


Figure 14: TC-NFS's runtime relative to the non-transactional baseline when archiving and extracting the Linux-4.6.3 source tree, with and without `xz` compression (higher Y values are worse).

BSD Tar. We used `tar` to archive and compress a Linux-4.20.7 source tree, and `untar` to decompress and extract the created archive. `tar` created a 104MB archive through reading 62,447 small files with the `xz` option enabled (default compression used by `kernel.org`). `untar` extracted the archive by reversing the process. There were also metadata operations on 35 symbolic links and 4,148 directories.

Figure 14 shows the relative runtime of `tar`/`untar` on TC-NFS compared to the vanilla NFS-Ganesha server. When there was no added latency, `untar`'s worst-case overhead was 25%. When the latency was greater than 25ms, the overhead dropped below 10%. `untar`'s overhead on TC-NFS mainly came from database interactions as well as files locking: `untar` created a large number of new files and thus TC-NFS needed to create and commit recovery records regarding these OPEN/CREATE operations, allocate UUIDs for new files, and commit these UUIDs with their corresponding file system handles and absolute paths into the database. When creating new files, TC-NFS needed to lock their parent directories; moreover, when writing data to files, TC-NFS also had to lock the target files.

`tar`'s runtime on TC-NFS was close to that on vanilla NFS-Ganesha server, and the worst case overhead was 6.1%. The overhead was small because `tar` read a lot of files but created only one file and constantly appended data to it. Reading files does not trigger backup or transactional database writes. Workloads packed with mutating metadata operations impose a higher overhead on TC-NFS, but `tar` performed only one metadata operation, to create the archive file. Writing the archive caused TC-NFS to create backups, but in practice, TC-NFS would not take time cloning data for append-only writes, thanks to our range-backup mechanism (see Section 2.2).

Although transaction support introduces an overhead, applications still run much faster when using a vectorized vNFS client with the transactional TC-NFS server, compared to using an in-kernel NFSv4 client with the non-transactional vanilla NFS-Ganesha server. For example, when using vNFS as the client and TC-NFS as the server, and depending on network latency, `untar` ran 2.73–116.7×, faster compared to the in-kernel NFSv4 client with the vanilla NFS-Ganesha server. Therefore, TC-NFS's performance is fairly reasonable when used together with a vectorized NFSv4 client. In fact, transactional semantics makes *more* sense when one tries to perform *multiple* operations at once, atomically.

5 RELATED WORK

Transactions in database management systems [8, 16, 19, 23–25] are well studied and have greatly simplified application development. Transactional storage (e.g., object, file system, or distributed) is also not new. However, to the best of our knowledge, no prior work considered executing NFSv4 compounds transactionally. We classify existing work as (1) transactional file systems and (2) transactional distributed storage.

Transactional file systems. Transactional file systems let developers offload the work of maintaining a consistent storage state to the file system. Microsoft's TxF [37] and QuickSilver's [28] database file systems leverage the early incorporation of transaction support into the OS. Transactional NTFS (TxF) allows file operations on an NTFS file system volume to be performed as a transaction. TxF transactions increase application reliability by protecting data integrity across failures and simplify application development by greatly reducing the amount of error handling code.

Spillane *et al.* [33] implemented transactional file access via lightweight kernel extensions in Valor, enabling high-performance transactions on any Linux file system through several new system calls. KVFS [32] implements a transactional file-system on top of a key-value database backed by a VT-Tree—an LSM-Tree with enhancements to workloads with large sequential I/Os.

TxFS [15] utilizes Ext4's journal to support atomicity, consistency, and durability; it offers a simple begin/commit/abort application API, but it supports only data operations and not meta-data ones that we needed. TxFS modifies the kernel VFS directly; our initial experiments on its prototype showed high overhead for all I/Os even when clients do not need transaction semantics. We chose to develop TC-NFS with user-level code in part to avoid the need for custom kernel changes. TxF, Valor, KVFS, and TxFS are all local file systems and only KVFS does not require platform-specific dependencies or kernel changes.

Transactional distributed storage. Distributed file systems expose storage units to clients over a network. The Wave Transactional File System (WTF) [7] is a distributed file system that lets applications operate on multiple files transactionally using a file slicing API, boosting performance by leveraging references to existing data. However, its multi-file operations are limited to yank and concatenate. CalvinFS [35] leverages a distributed database to build a scalable distributed file system with WAN replication and strong consistency guarantees. It implements compound transactions to scale read/write metadata operations but does not expose an interface to perform multi-file operations. Tyr [22] implements transactions using a blob-storage API. It enables applications to operate on multiple blobs atomically without complex application-level coordination while providing sequential consistency under heavy access concurrency. Unlike TC-NFS, Tyr lacks support for transactions on metadata operations as it is built on top of blob APIs.

6 CONCLUSIONS

NFSv4 compounds greatly improve performance but they also impose a burden on application developments due to complex error handling of large compounds. To solve this, we proposed TC-NFS, an NFSv4-based network file system that supports transactional compound execution. TC-NFS uses an embedded transactional database to manage its recovery records and mappings between NFS file handles and local file handles. To minimize overhead, TC-NFS utilizes Copy-on-Write mechanisms in modern file systems, to create partial or full file backups without copying data unnecessarily. Benchmarks of our prototype demonstrated that when compounds are utilized, TC-NFS's transaction support adds approximately 1.1% to 25× of overhead compared to a vanilla non-transactional NFSv4 server. This overhead is acceptable compared to the orders of magnitude of performance improvement from large compounds [1]; therefore we believe that transactional execution of compounds is not only desirable but also practical.

ACKNOWLEDGMENTS

We thank the anonymous Systor reviewers for their valuable comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1729939, and CNS-1730726.

REFERENCES

[1] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–314, Santa Clara, CA, February-March 2017. USENIX Association.

[2] Raymond Chen. When you start talking about numbers as small as 2^{-122}, you have to start looking more closely at the things you thought were zero. *https://devblogs.microsoft.com/oldnewthing/20160114-00/?p=92851*, January 2016.

[3] Microsoft Corporation. Server message block (SMB) protocol. Technical report [MS-SMB] - v20180912, Microsoft Corporation, September 2018.

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

[5] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucriere. GANESHA, a multi-usage with large cache NFSv4 server. Work-in-Progress Report, February 2007. *http://www.usenix.org/events/fast07/wips/deniel.pdf*.

[6] Paul Dix. Benchmarking leveldb vs. rocksdb vs. hyperleveldb vs. lmdb performance for influxdb. *https://www.influxdata.com/benchmarkingleveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-forinfluxdb/(visitedon05/26/2017)*, 2014.

[7] Robert Escriva and Emin Gün Sirer. The design and implementation of the Warp transactional filesystem. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 469–483, 2016.

[8] Facebook. RocksDB. *https://rocksdb.org/*, September 2019.

[9] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[10] Google. Protocol buffers. *https://developers.google.com/protocol-buffers*.

[11] Google. Leveldb benchmarks. *http://www.lmdb.tech/bench/microbench/benchmark.html*, Jul 2011.

[12] M. Haardt and M. Coleman. *fsync(2)*. Linux Programmer's Manual, Section 2, 2001.

[13] Gavin Henry. Howard chu on lightning memory-mapped database. *IEEE Software*, 36(6):83–87, 2019.

[14] D. R. Hipp. SQLite. *www.sqlite.org*, February 2006.

[15] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging file-system crash consistency to provide ACID transactions. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association.

[16] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[17] Chuck Lam. *Hadoop in action*. Manning Publications Co., 2010.

[18] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference*, volume 1, pages 5–2, 2008.

[19] LevelDB, September 2019. *https://github.com/google/leveldb*.

[20] Libuuid API. *UUID - DCE compatible Universally Unique Identifier library*, May 2009. *http://man7.org/linux/man-pages/man3/uuid.3.html*.

[21] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[22] Pierre Matri, Alexandru Costan, Gabriel Antoniu, Jesús Montes, and María S Pérez. Týr: Blob storage meets built-in transactions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 49. IEEE Press, 2016.

[23] MongoDB, Inc. MongoDB: The database for modern applications. *https://www.mongodb.com/*, September 2019.

[24] MySQL AB. MySQL: The world's most popular open source database. *www.mysql.org*, July 2005.

[25] Eric Newcomer and Philip A. Bernstein. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[26] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[27] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[28] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 239–253, Pacific Grove, CA, October 1991. ACM Press.

[29] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. RFC 3530, Network Working Group, April 2003.

[30] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, Network Working Group, April 2003.

[31] S. Shepler, M. Eisler, and D. Noveck. NFS version 4 minor version 1 protocol. RFC 5661, Network Working Group, January 2010.

[32] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.

[33] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[35] Alexander Thomson and Daniel J Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2015. USENIX Association.

[36] J Venner. Pro hadoop: Build scalable. *Distributed Applications in the Cloud, Berkeley: Apress*, 2009.

[37] S. Verma. Transactional NTFS (TxF). *http://msdn2.microsoft.com/en-us/library/aa365456.aspx*, 2006.

[38] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[39] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[40] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.