Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning

Tej Chajed, *MIT CSAIL;* Joseph Tassarotti, *Boston College;*
Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich, *MIT CSAIL*

https://www.usenix.org/conference/osdi22/presentation/chajed

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

Mark Theng
MIT CSAIL

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Abstract

This paper develops a new approach to verifying a performant file system that *isolates crash safety and concurrency reasoning* to a transaction system that gives atomic access to the disk, so that the rest of the file system can be verified with *sequential reasoning*.

We demonstrate this approach in DaisyNFS, a Network File System (NFS) server written in Go that runs on top of a disk. DaisyNFS uses GoTxn, a new verified, concurrent transaction system that extends GoJournal [9] with two-phase locking and an allocator. The transaction system's specification formalizes under what conditions transactions can be verified with only sequential reasoning, and comes with a mechanized proof in Coq [37] that connects the specification to the implementation.

As evidence that proofs enjoy sequential reasoning, DaisyNFS uses Dafny [26], a sequential verification language, to implement and verify all the NFS operations on top of GoTxn. The sequential proofs helped achieve a number of good properties in DaisyNFS: easy incremental development (for example, adding support for large files), a relatively short proof (only 2× as many lines of proof as code), and a performant implementation (at least 60% the throughput of the Linux NFS server exporting ext4 across a variety of benchmarks).

1 Introduction

File systems are important to implement correctly because applications rely on them to safely store user data. Formal verification offers a promise of showing that the implementation of a file system always meets its specification, including a crash safety property that says the file system recovers correctly from a sudden crash and reboot. However, efficient implementations are internally complicated, especially because they support concurrency and aim to minimize disk writes. Complexity makes the code more error-prone and motivates the desire for formal verification, but also poses a challenge: how can a proof cover concurrency, crash safety, and functional behavior while remaining tractable for a program the size of a file system?

The main contribution of this paper is a new approach to verifying a file system that *isolates crash safety and concurrency reasoning* to a transaction-system implementation. This use of a transaction system wraps the file-system data structures and logic inside a transaction, and permits *sequential reasoning* for the body of each transaction. Sequential reasoning keeps the proof burden manageable even with an efficient implementation that supports many features, such as large files and in-place updates of serialized metadata.

There are three challenges in realizing this approach. The most important lies at the interface between the transaction system and the file system: intuitively, transactions make things simpler, but how do we exploit this for a proof engineer verifying the code running in a transaction? This paper proves a *simulation transfer theorem* that formalizes how the proof engineer can verify the body of each transaction using sequential reasoning, and yet still obtain a proof about concurrent and crash behavior, due to the use of a verified transaction system. This specification and its proof are not specific to the file system written on top and could be applied to another storage system implemented using transactions. We use the transaction system with file-system code verified using Dafny [26], a verification-oriented programming language that is limited to sequential reasoning but in exchange has good automation.

The second challenge is how to implement and verify the transaction system itself. The performance and concurrency of the overall system can only be as good as the transaction system, so efficiency and fine-grained locking are important. To that end we implement a new transaction system called GoTxn by extending GoJournal [9] (a verified journaling system) with two-phase locking. GoJournal's specification guarantees crash safety but requires the caller to implement concurrency control (enforced with separation logic) to achieve atomicity. In proving GoTxn we give a new separation-logic proof of two-phase locking's correctness based on local reasoning rather than the typical textbook approach that reasons about the global conflict graph for a set of transactions. GoTxn cannot make arbitrary transactions appear atomic (for example, if they access global variables), and so the specification for GoTxn applies only to a carefully formalized subset of "safe" transactions that access shared state only through the transaction system.

The third and final challenge is how to implement the file system using only transactions. GoTxn's safety restriction would appear to preclude an in-memory allocator since it re-

quires other shared state, which we address by incorporating allocation with a non-deterministic specification into GoTxn, which is then used in the file system by validating the allocator's output. For sequential reasoning each operation must be implemented as a single transaction, but operations like removing a file can require a large number of disk writes that might not fit in a transaction. We implement freeing using multiple transactions; a first transaction logically deletes a file, and then asynchronously the implementation can run transactions that recover space from the file but have no other visible effect.

The verified artifact from this work is DaisyNFS, which implements a Network File System (NFS) server in Go on top of a bare disk and comes with a proof that clients observe that each operation follows the NFS specification as laid out in RFC 1813 [4]. Operations appear atomic despite concurrency and crashes. Clients can use the Linux or macOS NFS clients to mount DaisyNFS like any other file system and interact with it using the usual POSIX API. As an end-to-end check that our formalization of NFS is accurate and the implementation is reasonably complete, we tested with both Linux and macOS clients running a variety of programs.

A benefit of this file-system design is that it permits using the sharpest tool for each part of the proof: while we use Perennial [9], a program logic for crash safety and concurrency embedded in Coq, for the transaction system's proof, we use Dafny [26], a verification-aware programming language with powerful automation, for the file-system operations. Dafny is a purely sequential language, but we are able to use it despite this limitation due to the transaction system's proof. The value of sequential proofs can be seen in the proof-to-code ratio for the transaction system, which is 20×, versus the Dafny proofs which required about 2× as many lines of proof as code. Further evidence can be seen in the incremental development of DaisyNFS, which we elaborate on in §9.4.

To evaluate DaisyNFS's performance, we compare it to that of the Linux NFS server exporting an ext4 file system. DaisyNFS achieves within 90% of the throughput of Linux with the ext4 `data=journal` option (which gives the same crash-safety guarantees as DaisyNFS) across a variety of benchmarks both on an NVMe and in-memory disk, and at least 60% on the most challenging ones. The comparable performance is due to the efficiency of GoJournal and adding little overhead in the file-system code (e.g., updating data structures in place to avoid copying). We do note that ext4's default `data=ordered` mode can get about 60% better throughput for data-heavy workloads, at the cost of weaker guarantees on crash.

The contributions of this paper are:

- Formalization of a *simulation-transfer theorem* that captures how the transaction system provides sequential reasoning (§5.1) for any system implemented using a transaction per operation.
- A proof that the simulation-transfer theorem holds for the GoTxn implementation (§6). This proof verifies two-phase locking using a new strategy based on *local reasoning* to connect to the GoJournal specification. For the theorem to be true, it needs a precisely formulated definition of *safe* transactions that access shared state through GoTxn in order to behave atomically.
- Techniques to implement a file system using GoTxn, including a validation approach to integrating in-memory allocation into GoTxn and an approach for splitting file removal into multiple transactions of bounded size.
- DaisyNFS, a concurrent, crash-safe file system that is verified in Dafny with sequential reasoning thanks to the above techniques. The Dafny proofs for the file-system code enjoy low overhead compared to the concurrent proofs for GoTxn (2× vs. 20×). A performance evaluation shows that DaisyNFS gets throughput at least 60% that of Linux ext4 exported over NFS for the most challenging benchmarks, and within 90% for many workloads.

Our approach and DaisyNFS have some limitations. The proof approach relies on transactions appearing to run sequentially, which prevents modifying state outside the transaction system. There are cases where that would get better performance in exchange for a more difficult proof. The transaction system does not have a proof of liveness, and we do not prove that transactions avoid deadlock. DaisyNFS does not support NFS unstable writes, which improve performance by not committing writes to stable storage until explicitly requested. Our NFS implementation does not cover some features, such as symbolic links, hard links, and paginated `READDIR`; we believe these features could be implemented and specified with the same approach but have not done so in our prototype.

This paper describes work that is part of the first author's Ph.D. thesis [5], which provides more detail. The thesis also describes the Perennial logic for verifying concurrent and crash-safe systems, the specification and proof of GoTxn (including GoJournal), and Goose, the tool we use to verify GoTxn's implementation written in Go. It goes into more detail about the DaisyNFS proof and evaluation as well.

2 Related work

Our main contribution is a way to use transactions to enable sequential reasoning for a concurrent file system. Our approach allows using Dafny and produces a file system that gets good performance. Prior work has also explored how to compose proofs across layers for modularity, to contain concurrency, or to cross between proof systems in complementary and distinct ways; none use transactions or any similar mechanism to isolate concurrency or crash safety reasoning.

2.1 Verifying storage systems

Directly related systems DaisyNFS directly builds upon GoJournal [9] to implement the transaction system, together with

its new version of the Perennial framework [8] that is used to verify the transaction system's proof. This infrastructure is a program logic designed for storage systems that need a combination of concurrency and reasoning about crashes at any time, built on top of the Iris framework [23] in Coq.

The transaction system differs from GoJournal in that the GoJournal specification requires the caller to prove that concurrent transactions do not attempt to read or write the same objects, whereas the transaction system guarantees this automatically with per-object locks. The specification styles are also different: whereas the GoJournal proof is a set of specifications within the Perennial logic, the transaction system's proof uses a more general refinement-based definition that we can apply to the Dafny code. This is necessary to combine the tools, since Dafny cannot express the GoJournal specification's concurrency restrictions directly.

Directly related applications In prior work with the Perennial framework, we verified a crash-safe, concurrent mail server under the assumption that the file system is crash-safe [8]. DaisyNFS is a crash-safe file system and its complexity is significantly larger than a mail server: the mail server is about 150 lines with a monolithic proof while DaisyNFS combines a transaction system (itself 1,600 lines) with a 4,000-line file system, each of which involve many intermediate abstractions.

The authors of GoJournal verify a simple NFS file server on top of GoJournal, but that server is not complete enough to run real applications (it supports only one directory and 4KB files). Furthermore, the simple NFS server does its own locking and so the proof must reason about concurrency, increasing the proof overhead compared to DaisyNFS.

Other verified file systems Flashix [33] is a verified file system for flash storage, recently extended to support concurrency by Bodenmüller et al. [2]. File-system operations are proven to be atomic using a variant of Lipton's movers [28] technique with additional conditions to ensure crash-atomicity [31]. In contrast, DaisyNFS proves once and for all that operations encapsulated in a transaction are atomic. Flashix uses per-file locks to enable concurrent file accesses, but the directory tree is protected by a single reader-writer lock, so operations creating or moving files cannot proceed concurrently. DaisyNFS's two-phase locking system allows operations to proceed in parallel if they access disjoint parts of the file system.

VeriBetrKV [18] is a verified key-value store similar to the one that underpins the BetrFS [22] file system. It uses Dafny for crash-safety reasoning but does not layer any file-system proof on top. This file-system design does not involve general transactions, so the code on top of the key-value store must still carry out crash reasoning. The system has I/O concurrency but no CPU concurrency.

AtomFS [39] is a verified concurrent file system that does not persist data. It uses a custom concurrent relational logic implemented in Coq. Because the system does not persist data, AtomFS does not have any transaction system and implements the file-system operations together with appropriate locking for concurrency control.

2.2 Concurrency verification

A number of verification frameworks address concurrency, including CIVL [20], CSPEC [6], Armada [29], Iris [24], CCAL [16, 17], and FCSL [34], among many others. These frameworks use a range of methods, such as movers [28] and concurrent separation logic [3]. Although there has been much recent progress in using these frameworks to verify shared-memory concurrent systems, handling concurrency still brings additional proof burden compared to verification of sequential systems. DaisyNFS's design isolates this verification overhead to the transaction system's proof, and then uses Dafny to reason about file-system operations. Furthermore, it would be challenging to extend a concurrency framework with crash safety compared to starting with Perennial, which required non-trivial extensions to add crash-safety support to Iris.

IronFleet [19] applies Dafny's sequential reasoning to a non-sequential setting, namely to verify event handlers for distributed systems. Each event handler is structured in phases: first messages are received, some local computation is done, and then messages are sent. This structuring enables a reduction argument [28] which makes it sound to treat each event handler as if it ran in an atomic step, with no interleaving of steps by other machines. Instead of a reduction argument, DaisyNFS uses the transaction system to make operations atomic. Although DaisyNFS operations may only access shared state through the transaction-system API, there are no phases or constraints on the ordering of reads and writes within a transaction.

2.3 Verified two-phase locking

Chkliaev et al. [11] verify serializability of two-phase locking and other transaction concurrency control mechanisms in the PVS theorem prover. Their proof formalizes two-phase locking as an abstract protocol consisting of sequences of read, write, and locking operations, as opposed to a concrete implementation as in DaisyNFS. Pollak [32] uses a variant of the CAP separation logic [15] to give a pencil-and-paper proof of serializability for a two-phase locking implementation.

Lesani et al. [27] developed a framework for verifying software transactional memory algorithms, modeled as I/O automata. They applied their framework to sophisticated STM algorithms, such as the NOrec algorithm [14]. The STM algorithms considered do not handle persistence and the framework does not address crash-safety reasoning.

2.4 Unverified file systems

We chose to verify an NFS server because it is widely used in practice and the expected behavior of NFS operations is well

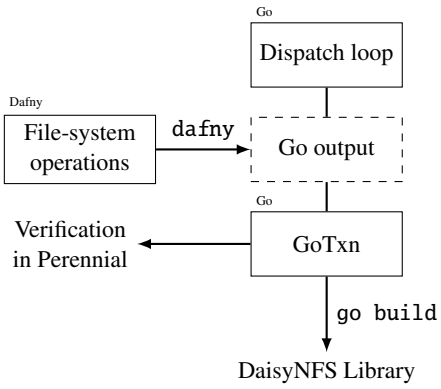


Figure 1: The structure of the code.

documented in RFCs. FUSE is an alternative for implementing file systems in user space, but its operations have a less clear specification.

Isotope [35] is a block-level transaction system similar to GoTxn in its API which was used to implement a file system called IsoFS. Its logging design is based on multi-version concurrency control (MVCC) [1] rather than our use of pessimistic locking. IsoFS has a similar design to DaisyNFS: it factors out isolation and atomicity to the transaction system, making it easy to handle crashes and concurrency. Unlike GoTxn and DaisyNFS, Isotope is unverified and thus prone to subtle concurrency bugs in the transaction system and bugs in the IsoFS code, whereas DaisyNFS uses the split design to verify both the transaction system and the transactions themselves.

To be conducive to verification, DaisyNFS is implemented differently than many NFS servers; the main differences are that using two-phase locking is not common practice, and most NFS servers are implemented on top of an existing file system. For example, the Linux NFS server can export any underlying file system supported by the kernel. An exported file system such as ext4 may use a journaling system, but the file system and VFS layers perform locking and are still prone to concurrency bugs. WAFL [21] is an NFS appliance that provides snapshots and logs NFS requests to NVRAM. It has evolved its locking plan to obtain good parallelism [13]. Both the Linux NFS server and WAFL are more complicated and have more features than DaisyNFS.

3 System design

As shown in Figure 1, DaisyNFS is implemented in three layers: 1) a dispatch loop that speaks the NFS wire protocol and calls the appropriate method for each operation; 2) a Dafny class that implements each method; and 3) a transaction system that applies the updates of each method to the disk atomically. The dispatch loop is unverified; we assume that the server correctly decodes messages, calls the right method for an operation, and encodes the response. The middle layer implementing the file-system operations is written and verified in Dafny, which has a backend for Go. The third layer is directly written in Go and verified using Coq and Perennial. By implementing the file system on top of the transaction system, we can implement each NFS method in Dafny as sequential code calling into a concurrent transaction system library. The NFS operations supported by DaisyNFS are listed in Figure 6.

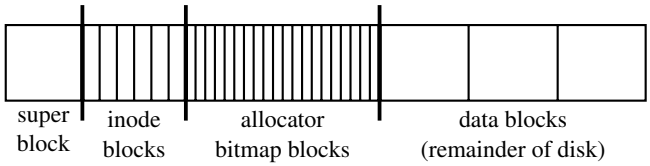


Figure 2: The layout of the file system on top of the transaction system's disk. The number of inode blocks and data bitmap blocks are compile-time constants, but easy to change without affecting the proofs.

3.1 Dafny file system

The file system is responsible for implementing files and directories onto an array of disk blocks that is exported by the transaction system. The disk layout used by the file system is shown in Figure 2, with regions for inode blocks, bitmap blocks, and data blocks for files and directories. This figure is in terms of the disk exported by the transaction system; the transaction system itself has a 513-block write-ahead log to support multi-block atomic writes to the disk.

The high-level organization of the file system separates three concerns, each building upon the previous: (1) implementing indirect blocks so files can be up to 512GB, (2) implementing byte-granularity reads and writes on top of blocks, and (3) implementing directories by encoding them as files with a special type together with operations to manipulate those files. §7 explains the internals of the file-system design in more detail, alongside the structure of the Dafny proof.

3.2 Transaction system

The transaction system handles concurrency and crash safety, and its API is listed in full in Figure 3. The file system creates an empty transaction by calling `Begin()`. The entire transaction appears to execute atomically when the caller finishes with `Commit`, or the transaction is discarded with no effect on `Abort`. Reads and writes operate on addresses which specify a position by giving a block number and an offset in bits (always less than $4096 \cdot 8$, the number of bits in a block). The `Read` method requires an explicit size argument while the size of a `Write` is implicit in the size of the `data` slice. We separate out the bit-sized operations to `ReadBit` and `WriteBit` (rather than using a single-element byte slice) to simplify the specification.

Figure 3 also shows the allocator API alongside the transaction API because its implementation is part of the interface that the Dafny code has access to. Allocation does not behave atomically along with the rest of the transaction, which the proof handles by allowing allocation to return any value. In

```
type Addr struct {
  Blkno  uint64
  Offset uint64
}

// starting and stopping a transaction
func Begin() *Txn
func Abort(tx *Txn)
func Commit(tx *Txn)

// operations within a transaction
func Read(tx *Txn, a Addr, sz uint64) []byte
func ReadBit(tx *Txn, a Addr) bool
func Write(tx *Txn, a Addr, d []byte)
func WriteBit(tx *Txn, a Addr, d bool)

// allocator API
func NewAllocator(max uint64) *Allocator
func Alloc(a *Allocator) uint64
func Free(a *Allocator, n uint64)
```

Figure 3: The API for the transaction system and allocator, both of which are available within the Dafny file-system implementation. Reads and writes between `Begin` and `Commit` appear to execute atomically on disk and for other threads, while `Abort` guarantees the transaction has no effect. The allocator's `Alloc` and `Free` operations are safe to call concurrently.

practice the way the file system uses such a non-deterministic specification is to store the ground-truth allocation state in the transaction system, and then to use the allocator as a hint to find free bits. As a result the return value of `Alloc()` must be checked against the durable bitmap with `ReadBit()`. Similarly, to free an address it must be both freed in memory and on disk with `WriteBit()`.

The transaction system builds upon GoJournal, verified in prior work [9], adding two-phase locking on top to implement transactions. While a transaction is running, it acquires locks for any addresses it reads or writes, and on abort or commit, it releases all locks held. Transactions that don't conflict can prepare in parallel, and GoJournal will batch concurrently committed transactions for efficiency.

Acquiring multiple locks during a transaction creates the possibility for deadlocks, for example if two threads acquire a pair of locks in the opposite order. The two-phase locking implementation does not implement a specific lock acquisition order, leaving it to the file system to avoid deadlock — the most interesting case is `RENAME`, which is discussed in more detail in §7.1.1.

4 Specifying DaisyNFS

The specification for DaisyNFS is a state machine describing an ideal NFS server in the form of an abstract state and a transition for each operation. The implementation of DaisyNFS is a binary `daisy-nfsd` that implements the NFS protocol, running on top of a disk. Then the DaisyNFS correctness theorem is a *refinement* property, which intuitively says that for any interaction with the implementation, the ideal, atomic NFS state machine could produce the same responses; §4.2 gives a more formal definition. As a result a client interacting with the server can pretend that it is the NFS state machine and ignore the complexities of its implementation.

4.1 Formalizing NFS

RFC 1813 specifies the NFS protocol, which we make mathematically precise with a state-machine representation defined in Dafny. The formalization requires first defining what state operations modify, and then a transition for each NFS operation that specifies how it changes the state and what return values are allowed. While most of the specification is deterministic, some operations have to be specified with non-determinism; for example, we allow returning an out-of-space error in many operations, and the specification allows any timestamp to be picked for the current time. The RFC is precise about arguments and allowed return values, and the text is good about explaining the intended behavior, but it does not describe the state an NFS server maintains. We define the NFS server state as shown in Figure 4.

```
// the abstract state of the file system
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Dir(dir: map<FileName, Ino>, attrs: Attrs)

type Ino = uint64
type FileName = seq<byte>
datatype Attrs = Attrs(mode: uint32, ...)
```

Figure 4: Dafny definition of the NFS server state (simplified).

This definition says that an NFS server conceptually maintains a mapping from inode numbers to files, where a file can either be a regular file with bytes, or a directory. Both types of files have a number of attributes, storing metadata like the file's mode (permission bits) and modification time. A directory is a partial map from file names (which are just bytes) to inode numbers. Note that DaisyNFS doesn't represent the file system as a tree but as a collection of links, which is sufficient to model all NFS operations, because NFS clients resolve pathnames.

The NFS state machine models each operation as a non-deterministic transition, written as a predicate that holds when it is allowed for an operation to change the state from `fs` to `fs'` and return `r`. The return value is always wrapped in a `Result` type, which can be either `Ok(v)` for a normal return or an error code for one of the errors defined in the standard. We systematically guarantee that the state is unchanged when an operation returns an error (though this is stronger than what the RFC requires); the transaction system makes this easy to achieve by aborting the whole transaction. For example, Figure 5 shows the specification for a (hypothetical) `GETSZ` operation that returns the size of the inode `ino`.

There are four clauses in the specification. The first just says that this operation is read-only. The second is one possi-

```
predicate GETSZ_spec(ino: Ino, fs: FilesysData,
  fs': FilesysData, r: Result<uint64>)
{
  fs' == fs &&
  (r.ErrBadHandle? ==> ino !in fs) &&
  (r.ErrIsDir? ==> ino in fs && fs[ino].Dir?) &&
  (r.Ok? ==> ino in fs && fs[ino].ByteFile? &&
            r.v == |fs[ino].data|)
}
```

Figure 5: Specification of a hypothetical `GETSZ` operation, a simplification of the real `GETATTR` operation.

Category	Operations	Verified
File and directory ops	`GETATTR`, `SETATTR`, `READ`, `WRITE`	✓
	`CREATE`, `REMOVE`, `MKDIR`, `RENAME`	✓
	`LOOKUP`, `READDIR`	✓
Unsupported features	`READLINK`, `SYMLINK`, `LINK`, `MKNOD`	✗
	`READDIRPLUS`, `ACCESS`	✗
Configuration	`FSINFO`, `PATHCONF`, `FSSTAT`	✗
Trivial operations	`NULL`, `COMMIT`	✓

Figure 6: NFS API and which operations DaisyNFS supports and verifies.

ble error: if the server returns `ErrBadHandle`, then `ino` is not allocated. The third is a different error, which says this operation returns `ErrIsDir` for directories. Finally the fourth case says that if the operation is successful, it returns the length of the data in `fs[ino]`. Dafny checks several consistency properties of this specification itself; for example, a use of `fs[ino]` will not even compile if the specification does not earlier imply `ino in fs`.

We developed a state-machine model of the regular file and directory operations in NFS in this style, including specifying what certain errors signify. Figure 6 lists the entire NFS API and what parts we verified.

DaisyNFS implements `FSINFO` and `PATHCONF`, which give the client static configuration information about the file system (for example, the maximum supported write size). These return constants and thus have no specification. DaisyNFS also implements `FSSTAT` to report total and free space, but it does not have a meaningful specification.

DaisyNFS could support some of the remaining operations with some more effort. Support for symlinks and `MKNOD` would require mostly mechanical changes to accommodate new file types. `LINK` is more complicated because in addition to tracking the link count of every file in the state, the specification for `REMOVE` needs to say that the link count is decremented and that the file is deleted if its link count drops to zero.

4.2 Specifying correctness for DaisyNFS

The transition system in §4 describes the abstraction of an NFS server, but what does it mean for the `daisy-nfsd` binary to implement this specification? To formalize DaisyNFS's correctness we use a definition of *concurrent, crash-safe refinement*, which informally says that every execution of that server binary — including with concurrent operations and crashes — has user-visible behavior that the specification could also produce (that is, the behavior is allowed by the specification). In DaisyNFS's specification the visible behavior is defined to be network requests and responses.

To define the specification, we need to be more precise about what a program is and how it executes, since these programs are used to model the DaisyNFS code and specification. We write $p : \texttt{Go}\langle \text{X} \rangle$ to say p is a Go program written using operations from layer X, where X is one of NFS, Txn, or Disk. Layer operations are always atomic transitions in a state machine. In the NFS layer, the operations behave according to the NFS state machine described previously in §4.1 and defined formally in Dafny. The Txn layer is specified both in Coq where it is part of the transaction system's correctness theorem and in Dafny where it appears as an assumption. The Disk transition system is formalized in Coq as part of the GoJournal proof, and assumes reads and writes of 4KB blocks are atomic. Each layer includes concurrent threads that interleave layer operations, basic heap operations on pointers, slices, and maps, and computation on primitives like integers and structs.

The correctness of DaisyNFS is stated in terms of a program that repeatedly receives a request, processes it in a background thread, and sends a response, which is intended to model the core behavior of the `daisy-nfsd` server. A schematic depiction of this server loop is given in Figure 7. This code starts by recovering the state of the system on line 3. Then it repeatedly accepts new requests from the network, abstracted with `GetRequest()` (including parsing the NFS wire protocol). These requests are each processed in a background thread due to the goroutine spawned on line 6. The processing for each request dispatches to the appropriate file-system operation (e.g., lines 9 and 12). The implementations of these operations are compiled from Dafny to Go and then linked with the transaction system.

The correctness theorem references three versions of this loop, at different levels of abstraction. At the top, the specification is a loop $\texttt{s}_{\text{NFS}} : \texttt{Go}\langle \text{NFS} \rangle$ which atomically processes each NFS operation according to the NFS state machine.

Below the NFS layer, $\texttt{s}_{\text{dfy}}$ models the server where each operation is replaced with its Dafny implementation, wrapped in a transaction. In this layer we write $\texttt{atomically}\{\texttt{f}\}$ to represent a transaction running `f`, which by definition in the Txn layer runs atomically for specification purposes. An `atomically` block corresponds to executable code that follows a pattern like `tx := Begin(); f(tx); tx.Commit()` to run `f` in the context of a GoTxn transaction (some additional code handling aborts is omitted in this snippet).

The final layer that models the executable code is given using a function $\text{link}(p, i)$, which takes a program p using operations from layer S and substitutes each operation with an implementation according to $i : S \to \texttt{Go}\langle \text{T} \rangle$. The notation "link" is intended as an analogy to the linking phase of compilation, taking a program p with some undefined symbols

```
// this is the core of daisy-nfsd
func main() {
  fs := filesys.Recover()
  for {
    req := GetRequest()
    go func() {
      switch req.Op {
      case CREATE:
        ret := fs.CREATE(req.Args)
        SendReply(req, ret)
      case LOOKUP:
        ret := fs.LOOKUP(req.Args)
        SendReply(req, ret)
        // ... other cases ...
      }
    }()
  }
}
```

Figure 7: A schematic depiction of the server loop, written in Go. $\mathtt{s}_{\mathrm{NFS}}$ looks like this code, but by definition all operations (for example the calls to `fs.CREATE` and `fs.LOOKUP`) are processed atomically and according to the NFS transition system. As far as the proof goes `GetRequest()` and `SendReply()` just produce a trace of I/O behavior and are unverified.

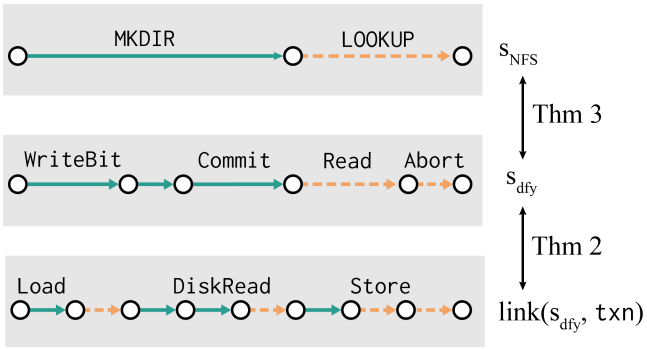


Figure 8: Illustration of the DaisyNFS proof strategy in terms of one possible execution of DaisyNFS, receiving parallel `MKDIR` and `LOOKUP` operations, at its three abstraction levels. Operations in each row are coded green and solid or orange and dashed according to which operation they correspond to (the top-level `MKDIR` or `LOOKUP` respectively). The refinement proof first shows that for every code execution (bottom row), there exists an atomic execution at the Txn layer (middle row), as proven in Theorem 2. This justifies sequential reasoning to show the transactions follow the NFS specification (top row), as proven in Theorem 3. Finally Theorem 1 puts the two together.

and substituting each symbol s with a call to an implementation of that method given by the library code $i(s)$. We write $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn})$ to represent linking the Dafny code with the transaction system's implementation `txn`.

The proof is about the server loop at the core of `daisy-nfsd` at three layers of abstraction. Figure 8 illustrates one execution of the DaisyNFS server where two clients issue `LOOKUP` and `CREATE` in parallel, at the three levels of abstraction: the bottom shows an execution of $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn})$ at the Disk layer, the middle a corresponding atomic execution of $\mathtt{s}_{\mathrm{dfy}}$ at the Txn layer, and finally the top-level has a single transition for each operation at the NFS layer.

Refinement relates two programs in terms of their visible behavior, which we will use to connect the server loop at the disk layer to the transaction layer and finally to the NFS layer. For the purposes of this paper, all of the programs involved are servers that issue network I/O, either receiving an NFS request or responding to one. Regardless of the level of abstraction, each model of the server defines a trace of network I/O consisting of requests and responses, and this is the behavior refinement talks about:

Definition (Concurrent, crash-safe refinement)**.** An implementation program p_c is a *concurrent, crash-safe refinement* of a specification program p_s, written $p_c \sqsubseteq p_s$, if whenever there are initial states σ_s and σ_c satisfying $\mathrm{init}(\sigma_s, \sigma_c)$ and p_c can execute from σ_c and produce a trace of network I/O tr, then p_s can execute from σ_s and produce the same trace tr. Execution might involve crashing and restarting a program (potentially multiple times), wiping out any in-memory state after each crash. When we state $p_c \sqsubseteq p_s$ we leave implicit the definition of initial states $\mathrm{init}(\sigma_s, \sigma_c)$, which will generally say both states are all zeros and of the same size.

The intuition behind the notation $p_c \sqsubseteq p_s$ is that the set of behaviors of p_c (the set of traces of network I/O tr) is a subset of the behaviors of p_s.

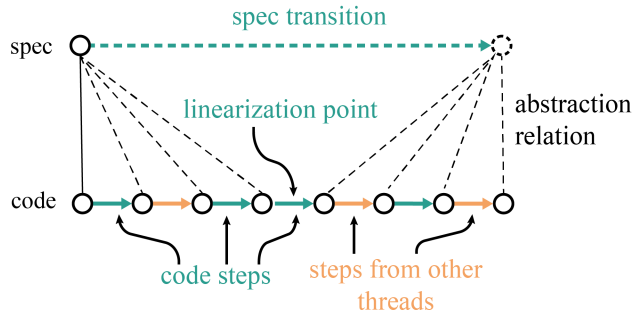
Now we have enough to state the final DaisyNFS correctness theorem:

Theorem 1 (DaisyNFS correctness)**.** $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn}) \sqsubseteq \mathtt{s}_{\mathrm{NFS}}$.

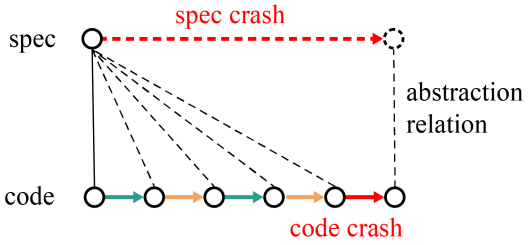
In this correctness theorem, initialization requires running a Dafny method on an empty disk. Subsequently the system boots by first recovering the transaction system, then restoring the file system. Theorem 1 will follow from the correctness of the transaction system combined with the results from Dafny.

5 Verification approach

DaisyNFS's concurrent, crash-safe refinement is a much more sophisticated property to verify than sequential refinement. Figure 9 illustrates the complexity of proving a concurrent and crash-safe refinement, whereas Figure 10a shows the relatively simple per-operation obligation for sequential reasoning. For both forms of refinement, the basic proof technique is to construct a *forward simulation* from the code execution to the specification transition system, which requires an abstraction relation connecting their states and a proof that shows the abstraction relation is preserved by operations. In a sequential, non-crash simulation, it is sufficient to show that each operation restores the abstraction relation when it returns since its intermediate states are invisible. The complication in a concurrent simulation is that the code can have many concurrent threads, each running a different operation at the specification level. The proof of any given operation must also show that the intermediate states satisfy the abstraction relation, since at any time other threads might run. Similarly, the proof of each operation's implementation must consider interference with its execution from other threads at any time.



(a) Linearizability obligation (for each operation)



(b) Crash-safety obligation (for each operation)

Figure 9: Obligations for verifying a concurrent, crash-safe refinement. The proof of refinement must show that every operation simulates the abstract specification for the operation at some *linearization point* (as illustrated in 9a), and that a crash simulates a specification crash transition (as illustrated in 9b). The abstraction relation must be preserved at all intermediate points, including after a crash.

5.1 Simulation transfer

The design of DaisyNFS uses transactions, and in particular GoTxn, to simplify the proof of concurrent refinement. Transactions appear to run sequentially, and thus should permit reasoning about the body of each transaction sequentially even though the actual execution interleaves multiple transactions. A key contribution of this paper is the formalization of a *simulation-transfer theorem* which proves that a system implemented with transactions that is verified with a *sequential* forward simulation against some specification refines the same specification in the sense of a *concurrent, crash-safe refinement* when run through GoTxn.

Due to simulation transfer, we can use the simpler verification methodology of sequential simulation for the DaisyNFS file-system code, compared to the Perennial program logic used to verify the transaction system underneath. To fully take advantage of this difference, DaisyNFS is verified using Dafny [26], an entirely different tool. Dafny is a verification-oriented programming language that is restricted to sequential proofs. The use of Dafny greatly reduces the proof burden for verifying DaisyNFS, because sequential proofs are well-suited to automation and Dafny's automation is well-developed (in contrast automation for concurrent proofs is still nascent, and would need to be integrated into Perennial to be used for these proofs).

The value of sequential proofs can be seen in the proof-to-code ratio for the transaction system, which is around 20×, versus the Dafny proofs which required about 2× as many lines of proof as code. Further evidence can be seen in the incremental development of DaisyNFS, which §9.4 further elaborates on.

To make simulation transfer this precise, let us first define "sequential reasoning" more formally. Suppose we have an implementation of layer S using operations from T. Note that all the proofs about the transaction system are for an arbitrary system with operations in S; though we use the system with an implementation of NFS, the GoTxn proof is more general. The implementation i consists of a function $i(op) : \texttt{Go}\langle \text{T} \rangle$ for each operation $op \in S$. The statement seq_refinement$\langle T, S \rangle(i)$ says that i is a correct sequential implementation of S using T. To specify the normal behavior of each operation, the definition refers to $s \overset{op}{\rightsquigarrow} s'$, which says op can transition from s to s' according to the definition of layer S. To specify correctness under crashes, this definition refers to crash(t,t') and crash(s,s'), which are the crash transitions for layers T and S respectively and model, for example, clearing the contents of memory.

Definition (Sequential refinement)**.** The implementation $i : S \to \texttt{Go}\langle \text{T} \rangle$ is a *sequential refinement*, written seq_refinement$\langle T, S \rangle(i)$, if there exists an abstraction relation $R \subseteq \Sigma_S \times \Sigma_T$ such that:
(1) for every operation $op \in S$, the following sequential Hoare triple holds:

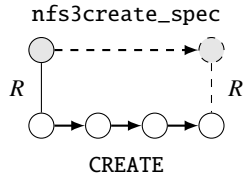
$$\{\lambda t.\, R(s,t)\}\ i(op)\ \left\{\lambda t'.\, \exists s'.\, R(s',t') \wedge s \overset{op}{\rightsquigarrow} s'\right\},$$

(2) init(s,t) must imply $R(s,t)$, and
(3) if $R(s,t)$ and crash(t, t') hold, then there exists an s' such that $R(s',t')$ and crash(s,s').

Conditions (1) and (2) in this definition are standard for sequential verification of refinement, while condition (3) is a standard condition for sequential crash-safety [7]. Though condition (3) requires the abstraction relation to be preserved by crashes, the proof engineer does *not* have to reason about crashes in the middle of operations. The diagram in Figure 10 depicts the main refinement condition (1) diagrammatically.

Simulation transfer takes a proof of *sequential* refinement conditions for a system implemented using transactions and derives a *concurrent and crash-safe* refinement. A transaction must satisfy some conditions to ensure atomicity. We write safe(p) to say that p is a valid transaction. The main restriction is that p cannot access global state such as the heap, since the transaction system does not make such accesses atomic. The implementation i in this theorem gives only the body of each transaction; the theorem instead references atomically $\circ\, i$ where $(\text{atomically} \circ i)(op) = \texttt{atomically}\{\, i(op)\,\}$ uses the macro from the Txn layer to specify that the operation is wrapped in a transaction and is thus by definition atomic.

Theorem 2 (Simulation transfer)**.** Let S be a spec layer implemented using transactions with $i : S \to \texttt{Go}\langle \text{Txn} \rangle$, such that



(a) Obligation for sequential refinement.

```
method CREATE(d_ino: uint64,
              name: Bytes)
 returns (r: Result<Ino>)
 requires R(txn_disk, fs)
 ensures R(txn_disk, fs)
 ensures r.Ok? ==>
 nfs3create_spec(d_ino, name,
   old(fs), fs, r.v)
```

(b) Dafny encoding

Figure 10: Illustration of seq_refinement(i_{NFS}) (left) and its encoding in Dafny seq_refinement$_{\text{dfy}}(i_{NFS})$ (right), for one particular operation. In the diagram, the solid parts are assumed, and the dashed parts must be shown to exist. The complete Dafny spec is more precise about errors.

seq_refinement(i) and $\forall op.\,\text{safe}(i(op))$ hold. Then

$$\forall p : \texttt{Go}\langle \text{S}\rangle, \text{link}(\text{link}(p, \text{atomically} \circ i), \texttt{txn}) \sqsubseteq p.$$

Simulation transfer says that if an implementation of S using transactions is correct in a sequential sense, then this is sufficient for any spec program p to have atomic and correct behavior for its primitives when run with GoTxn. The executable code for p derived in two steps: $\text{link}(p, \text{atomically} \circ i)$ replaces the operations in S with their atomic implementations at the GoTxn API level, while $\text{link}(\text{link}(p, \text{atomically} \circ i), \texttt{txn})$ takes the result of this process and substitutes the actual GoTxn implementations of `Begin`, `Read`, `Commit`, and so on. In §6 we discuss how this theorem is proven using Perennial and Coq.

5.2 Putting simulation transfer together with Dafny proofs

In order to use simulation transfer to obtain Theorem 1, we need to prove that DaisyNFS's implementation, i_{NFS}, satisfies the sequential refinement conditions. To do so, we define seq_refinement$_{\text{dfy}}(i)$, an encoding of sequential refinement using Dafny pre- and post-conditions (as illustrated in Figure 10), and prove that DaisyNFS satisfies these conditions in Dafny. The crash refinement condition (3) is straightforward; crashes have no effect in both the Txn layer and the NFS layers because they do not have ephemeral state. Details on how the Dafny obligations handle initialization and recovery are found in the first author's thesis [5: §6.4].

Lemma 3. seq_refinement$_{\text{dfy}}(i_{NFS})$ holds.

From here we can apply Theorem 2 to Lemma 3 and obtain Theorem 1, which says $\text{link}(\texttt{s}_{\text{dfy}}, \texttt{txn}) \sqsubseteq \texttt{s}_{\text{NFS}}$ (note that $\texttt{s}_{\text{dfy}} = \text{link}(\texttt{s}_{\text{NFS}}, \text{atomically} \circ i)$). Figure 8 illustrates just one execution that the theorem covers: the transaction system proof guarantees an atomic execution while the sequential refinement guarantees the transactions themselves are correct. There are two trusted assumptions needed for the theorems to compose. First, seq_refinement$_{\text{dfy}}(i_{NFS})$ should imply seq_refinement(i_{NFS}). That is, the encoding of the refinement conditions in Dafny must be correct, but also the semantics of the transaction system operations modeled in Dafny must match the Coq proof. Second, every Dafny transaction must be valid, meaning safe($i_{NFS}(op)$). The Dafny code satisfies safety due to a simple syntactic check: the only mutable state in the file-system Dafny class is the transaction system, so file-system operations cannot make mutations other than through GoTxn.

6 Verifying the transaction system

This section describes the implementation and proof of the transaction system, GoTxn. A contribution of this paper detailed in this section is to verify the powerful specification of Theorem 2 on top of a real implementation, which required verifying two-phase locking using local reasoning in Perennial unlike the more typical textbook proofs that reason about the global execution of many concurrent transactions. Note that this section is only about the transaction system and has nothing specific to the file system implemented on top.

6.1 GoTxn's implementation

GoTxn is implemented as an extension to GoJournal [9], a journaling system verified in Perennial. The journaling system provides the ability to write multiple objects atomically, with an implementation that provides good concurrency. For correctness GoJournal relies on the caller to guarantee that concurrent operations do not access the same disk objects. GoTxn automatically provides the concurrency control to guarantee this precondition using two-phase locking (2PL). The result is an interface that behaves atomically without any concurrency reasoning from the caller.

The two-phase locking system logically maintains a lock per object. The algorithm gets its name from an *expanding* phase in which reads and writes acquire locks as needed, followed by committing the transaction's writes to the journal and a *contracting* phase where all the acquired locks are released. Instead of committing, a transaction can abort early to abandon buffered writes and release the locks acquired so far, in which case the disk is unaffected. The whole operation appears to execute atomically at commit time; reads return their results early, but the locks ensure these values remain consistent up until the commit point. The GoTxn proof makes the informal correctness argument precise by giving a proof of a refinement-based specification.

6.2 Verifying two-phase locking with local reasoning

In §5.1, we gave Theorem 2 as the specification for the transaction system. Recall that this theorem converts sequential refinement proofs for transactions into concurrent refinement. To prove this, we first use Perennial to show that code encapsulated in a transaction truly behaves atomically, formalized with the following theorem:

Theorem 4. The GoTxn implementation `txn` is a *transaction refinement*, meaning for all $p : \texttt{Go}\langle\text{Txn}\rangle$ where $\text{safe}(p)$ holds, $\text{link}(p, \texttt{txn}) \sqsubseteq p$. The definition of $\text{init}(s,t)$ in this refinement relates an all-zero physical disk to an all-zero transactional disk of the same size.

Theorem 4 captures the intuition that transactions provide atomicity, while Theorem 2 formalizes why atomicity provides sequential reasoning. The proof of Theorem 2 from Theorem 4 is conceptually straightforward. Since the `atomically` blocks in p ensure transaction operations run without interruption, the sequential refinement diagram can be applied to code inside these blocks.

The proof of Theorem 4 itself in Perennial is more involved. The high-level approach is to encode refinement as Perennial Hoare triples, one for each operation [8, 38]. To make this sound for concurrent refinement, (1) the proof must identify and verify the *linearization point* of an operation, the time at which the operation appears to have executed; and (2) the proof tracks logical *ownership* of state, and threads may only modify state that they have "acquired" ownership of through synchronization. The resulting proof style is called "local" because we reason about each thread in isolation, considering just the parts of state it accesses. Using Perennial enables us to re-use the existing GoJournal proof, but this local proof-style is quite different from standard proofs of serializability for two-phase locking, which reason globally about the set of transactions and ordering constraints imposed by locks.

In more detail, the refinement proof must show that the code `tx := Begin(); f(tx); tx.Commit()` has a subset of executions of the `atomically{f}` construct. The difficulty in proving this is that the linearization point is at the very end when the code calls `Commit`, at which point the actual earlier execution of `f` becomes visible to other threads. We must argue that at this point the entire `atomically{f}` block's effect has occurred by tracking the behavior of `f`.

As the transaction executes, the proof tracks the initial value of any objects accessed in a map J. The domain of this map $\Sigma = \text{dom}(J)$ is the *footprint* of the transaction, which two-phase locking keeps locked during the transaction. The intuition behind the invariant is that if the transaction only depends on J, the transaction's execution can be delayed to take place atomically at the call to `Commit`, because locking prevents the subset J of the journal from being accessed by other threads. In particular the proof sets up a set of *lock invariants* that say the lock for address a is needed to access the GoJournal resource $a \mapsto_d o$, which gives permission to read and write to a. See the thesis for a more formal connection to the GoJournal specification [5: §5.5].)

The proof maintains a refinement relation during the execution of a transaction `f`, which is formally expressed using the GoJournal resources but explained more intuitively here. Let J be a map with the values of each object in the transaction's footprint Σ at the first time they are accessed by `f`, and let J' be a map with the transaction's current buffered in-memory view of the same addresses. Then, the invariant requires that after n steps of execution:

1. The transaction holds the lock for every address $a \in \Sigma$.
2. Executing n steps of `f` in *any* starting state that has the same values as J for the addresses in Σ can lead to a state with values given by J'.

At the start of a commit, the locking described by the first part of the invariant ensures that the durable value of each address still match the value in J, and is required to call the GoJournal `Commit` operation. The second part of the invariant means that even though other parts of the state outside of Σ may have changed, those changes do not affect the execution of `f`. Thus, executing `f` at this point in a single step would have the same behavior as the implementation has observed. The GoJournal `Commit` specification ensures that the durable values of objects in the footprint are atomically updated to match J'.

Showing that the second part of the invariant holds requires that code within a transaction must not access global state outside of the transaction system, as mentioned at the end of §5.1. Accesses to such global state would violate the invariant because their behavior would then depend upon things outside of the footprint Σ. Because those global values could change by the time the transaction commits, the above argument would no longer work if they were allowed.

The allocator creates another subtlety related to the second part of this invariant. Allocations do not hold the allocator lock throughout the remainder of a transaction. This seems to violate the two-phase locking pattern, since allocations could be implicitly observed by other concurrent transactions from the fact that an allocated address is no longer free. Correspondingly, in the proof, the footprint J of a transaction does not describe the allocator state. Thus, at the linearization point, the addresses returned by the allocator may no longer be free. However, because the specification for the allocator does not guarantee that returned addresses are actually free, the second part of the invariant above still holds.

7 Verifying the Dafny implementation

We follow the standard approach for verifying software in Dafny: each file-system operation is implemented as a method on a class and its specification is given using pre- and post-conditions. §5.1, explains how the Dafny proof shows the code is a correct implementation of NFS in terms of sequential refinement. This section provides details about the file-system design and proof.

DaisyNFS is implemented and verified in several layers of abstraction, depicted in Figure 11. Each layer is implemented as a class that wraps the lower layer as a field. The transaction system is an assumed interface in Dafny, while the complete server implements the NFS wire protocol and calls into the top-level Dafny class for each operation.

Layer	Functionality
dir	Directories and top-level NFS API.
typed	Inode allocation.
byte	Implement byte-level operations using blocks.
block	Gather blocks for each file into a single sequence.
indirect	Triple-indirect blocks organized in a tree.
inode	In-memory, high-level inodes; block allocation.
txn	Assumed interface to GoTxn.

Figure 11: Layers in the Dafny implementation and proof of the file-system operations.

Between the layers of the file system there are three difficult pieces of functionality: organizing data blocks into metadata and data (the indirect and block layers), translating byte-level operations into block operations (the byte and typed layers), and implementing directories as special files that the file system itself reads and writes (the dir layer). The modularity was essential to complete the proof in manageable chunks (to avoid overwhelming the developer and prover), and it would have been natural even without verification.

7.1 Implementing the file system using transactions

The design of DaisyNFS is broadly similar to the file system in xv6 [12], as well as Yggdrasil [36], a verified sequential file system. We also adopt the recursive strategy for implementing and verifying indirect blocks from DFSCQ [25]; recursion simplifies the implementation of triple-indirect blocks, which are needed to reach a reasonable maximum file size of 512GB. Unlike most file systems, DaisyNFS is designed to fit every operation into a transaction in order to support our goal of sequential reasoning. This is a non-standard design and we encountered some unique challenges in doing so. In this section we highlight difficulties in fitting two features into transactions: renaming and freeing space from deleted files.

7.1.1 Avoiding deadlock in renames

The NFS RENAME operation is similar to the `rename` system call: it moves a source file or directory to a destination location. What makes it tricky is that it involves more than one inode and hence introduces the possibility for deadlock. We use the standard strategy of enforcing a global ordering where inodes are always locked in numerical order (smaller inode numbers first); this avoids a deadlock where a cycle of threads is waiting on each other.

In a rename operation, the source and destination are each specified by a combination of the parent directory inode and name within that directory. Rename has an additional functionality of overwriting the destination if the source and destination are files, or if both are directories and the destination is empty. It is this latter check that makes deadlock avoidance difficult: it is necessary to lock the source and destination directories first to lookup the source and destination names, but those might be files that are earlier in the inode lock order. We address this in the code by returning an error from the Dafny transaction before the lock order would be violated. The error comes with the set of inodes that should have been acquired. The rename is then re-run with this set of inodes as a lock hint; these are first acquired in the correct order, then compared against the current source and destination in case they have been renamed concurrently.

At this point it is worth discussing the performance considerations that lead to handling lock ordering in the file system, rather than generically in GoTxn. The transaction system could avoid deadlocks by either enforcing a global order over addresses or by timing-out operations. Enforcing a global order is inefficient for the file system; data blocks will never cause deadlock because the file system only accesses a block after locking the (unique) inode that owns it. Timing-out operations would lead to slow and spurious transaction failures that could more rapidly be avoided in the higher-level code, hence we do not attempt to detect deadlock dynamically.

7.1.2 Freeing space

Freeing space becomes surprisingly tricky with large files. The problem is that a large-enough file may reference too many blocks to be freed in a single transaction. DaisyNFS handles freeing by removing a file from its directory and marking it free in one transaction, and in separate transactions reclaiming the space it took by deallocating its blocks.

Removal is implemented as a combination of two transactions, one which performs the logical operation but leaks space, and an operation `ZeroFreeSpace(ino)` which frees and zeroes the unused space in an inode that we prove has no effect on the file-system state. Because this operation is a logical no-op, it is safe to call it at any time. In practice the implementation is careful to call it after any operation that leaves unused blocks, in particular `SETATTR`, which can shrink a file by reducing its size, and `REMOVE`, which deletes a file. Furthermore since `ZeroFreeSpace` doesn't affect the user-visible data, it may return early to avoid overflowing a transaction, which GoJournal limits to 511 blocks.

There is one case where freeing blocks is important for correctness and not just to reclaim space. Growing a file is supposed to logically fill the new space with zeros. If the file had old data in that space, it would not be zero but some previously written and deleted data, which both violates the specification and is a potential security risk. The way we handle this with background freeing is with a run-time check: when the `SETATTR` operation grows a file checks, it checks if the free space is already zero first, and if not fails with a special error code. The unverified code interprets this as a signal to immediately call `ZeroFreeSpace` and try the operation again. The same support also handles holes created by writing past the end of a file, which are similarly supposed to be zero.

The freeing implementation is an interesting example of using validation in verification. The specification for much of the freeing code is loose, allowing any data to be written

	proof	code	spec
GoJournal	29,000	1,419	
Transaction system	10,000	250	932 (Thm 2)
File system	6,787	4,051	630 (Thm 3)
Trusted interfaces	—	—	558
`daisy-nfsd`	*unverified*	1,144	—

Figure 12: Lines of proof, code, and trusted specification. GoJournal is included only for comparison; its specification is subsumed by the transaction system's.

to the free space. We only needed a strong specification for the code that checks if the zeroing is done; the rest of the code needs to be correct for this check to ever succeed, but we aren't required to prove it.

7.2 Achieving good performance

An important aspect of the Dafny proof was to write code in a way that produces high-performance Go code. Compared to Dafny's C# backend, the generated Go code for Dafny's built-in immutable collections has much additional pointer indirection and defensive copying. Using these data structures for byte sequences would simplify proofs, but has unacceptably poor performance in Go.

To avoid this performance problem we use an axiomatized interface to Go byte slices (`[]byte` in Go) whenever raw data is required, including file data and paths, and then modify these slices in-place. It was possible to axiomatize this API without any changes to Dafny; we use a standard Dafny feature of `extern` classes to specify a Dafny class `Bytes` in terms of ghost state of type `seq<byte>` but then implement it as in Go as a thin wrapper around the native `[]byte` type. This API is trusted, so we test it: for example to catch off-by-one errors in the specification, we wrote tests like `[]byte{1,2,3}[2]` and ran them in Go and (equivalent) Dafny.

The on-disk data structures—inodes, indirect blocks, and directories—are represented in memory in their serialized form and modified by updating this representation directly, avoiding copies to move between representations. These were first written with slower purely functional code, which was then migrated to imperative code that used the functional code as a specification.

Dafny's default integer type `int` is unbounded and compiled to big-integer operations. We used Dafny's `nativeType` support to instead define a type of 64-bit integers (that is, natural numbers less than 2^{64}) and compile this to Go's `uint64`. This requires overflow reasoning, but automation makes this palatable in the proof and the performance gain is significant.

8 Development effort

We implemented DaisyNFS in a combination of Go and Dafny, with proofs in the Perennial framework (which is a library in the Coq proof assistant, heavily based on Iris [23]) and inline in Dafny. The Go side uses GoJournal, which we extend with a transaction system and concurrent allocator. The implementation is publicly available.[1]

The lines of proof, code, and specification for the layers of the system are summarized in Figure 12. GoJournal is prior work but included for comparison purposes. The GoTxn correctness proof, Theorem 2, is relatively large because code executed in `atomically` blocks can include many Go operations modeled by Perennial, and the proof has cases to handle each operation. However the result of the proof is a relatively concise specification as a plain Coq statement that doesn't refer to the Perennial logic.

The file-system operations are implemented in Dafny, which helped us verify a relatively complete system without too much tedium. The proof-to-code ratio (where code is the number of lines extracted by Dafny's `/printMode:NoGhost` flag) is about $2\times$ for the file system code. The proof summarizes the implementation well, with about $1/7$th as many lines of specification as code (about half that specification is quite verbose and concerns error codes and attributes). For efficiency, the Dafny code has trusted interfaces to primitives like byte slices and integer-to-byte encoding. Together these are written in 558 lines of trusted Dafny code. Finally, to complete the NFS server required around 1,000 lines of Go code, about half of which bridge between the Dafny method signatures and the actual NFS structs.

Similar to VeriBetrKV [18], we followed a discipline of identifying and addressing timeouts in the proof. As a result, the overall build is fast: compiling the proofs takes only 12 minutes on a slow machine in continuous integration and 4 minutes on a laptop using eight CPU cores.

9 Evaluation

In this section we evaluate DaisyNFS along the dimensions of performance (§9.1 and §9.2), correctness (§9.3), and ease of change (§9.4).

9.1 Performance

To evaluate the performance of DaisyNFS, we ran three benchmarks: the LFS smallfile and largefile benchmarks, and a development workload that consists of `git clone` from a local repository followed by running `make`. These are the same benchmarks used by DFSCQ [10] (a state-of-the art sequential verified file system) and for an unverified NFS server implemented on top of GoJournal [9]. To evaluate the benefit of concurrency, we also evaluate against a "seq txn" variant of DaisyNFS that replaces its per-address locking with a single global transaction lock. In non-concurrent workloads, this variant performs slightly better, demonstrating the overhead of fine-grained locking.

As a baseline, this evaluation uses a Linux NFS server exporting an ext4 file system mounted with `data=journal`

[1]The Dafny implementation of DaisyNFS is at `github.com/mit-pdos/daisy-nfsd`. It imports the transaction system from `github.com/mit-pdos/go-journal`.

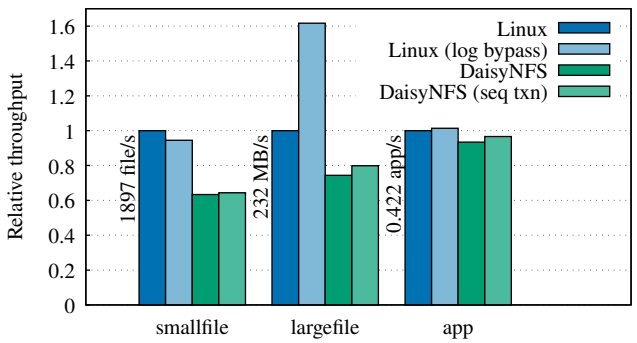


Figure 13: Performance of Linux NFS and DaisyNFS for `smallfile`, `largefile`, and `app` workloads, on an NVMe disk. DaisyNFS achieves comparable performance to ext4 in `data=journal` mode.

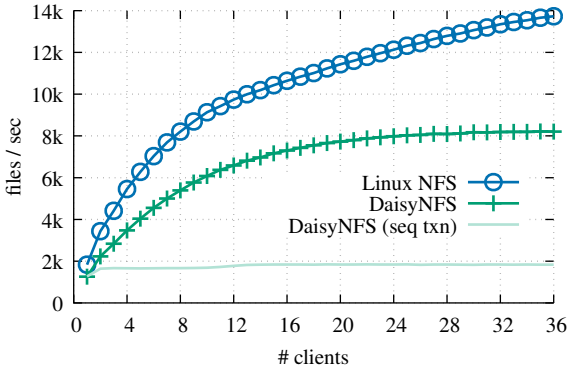


Figure 14: Combined throughput of the smallfile microbenchmark running on an NVMe disk while varying the number of concurrent clients. DaisyNFS's performance scales with the number of cores, though not as well as Linux; both eventually saturate the disk and scale sub-linearly.

mode. The NFS server lets us compare fairly since both go through the Linux NFS client and use the same underlying protocol. Using `data=journal` forces all data to go through the journal and disables log-bypass writes, which ensures that ext4 and DaisyNFS both guarantee NFS RPCs are committed durably when they return. The evaluation also presents results with ext4's log-bypass optimization (in `data=ordered` mode), which gets better performance for some benchmarks but can lose recently written data if the system crashes.

All of these benchmarks were run using Linux 5.15 and Go 1.18.1 on an Amazon EC2 i3.metal instance, which has 72 cores, 512 GB of RAM, and a local 1.9 TB NVMe SSD. To reduce variability we limit the experiment to a single 36-core socket, disable turbo boost, and disable processor sleep states; the coefficient of variation for all experiments is under 5% so we omit error bars for visual clarity.

The results are shown in Figure 13. DaisyNFS gets about 60% the throughput of Linux on the smallfile benchmark, which is intended to be metadata-heavy. The smallfile benchmark repeatedly creates a file, writes 100 bytes to it and syncs the file, then deletes it. Performance is lower than with Linux due to less efficient use of the drive; we used `blktrace` to confirm that Linux issues fewer I/O requests per iteration and that those writes are entirely sequential, unlike with DaisyNFS. Performance is comparable when run on an in-memory disk (not shown in the graph).

DaisyNFS gets comparable throughput to Linux on the largefile benchmark, which is intended to measure bulk data writes. The benchmark creates a 300 MB file by appending repeatedly, then syncs it. Note that in this benchmark ext4 is 60% faster with its log-bypass optimization due to no longer writing all data through the journal. For this workload, the Linux NFS client buffers the entire append process until the final sync, at which point it issues the writes in many chunks in parallel. These RPCs are challenging to support efficiently because they do not arrive at the server in order, so some are past the end of the file. The semantics of such a write are to fill the gap with zeros, but both DaisyNFS and Linux get good performance despite this because they implicitly encode those zeros without even allocating a block.

DaisyNFS achieves good performance on the app workload, which consists of running `git clone` on the xv6 repo followed by `make`. xv6 is an operating system, so building it requires running the usual development tools—`gcc`, `ld`, `ar`—but also running `dd` to generate a kernel image. Builds take about 3s (of which about 1.2s are spent compiling and not in the file system), which are reported as a throughput number so higher is better.

9.2 Scalability

DaisyNFS executes NFS operations concurrently to achieve better performance with multiple cores. The transaction system is built on GoJournal, which already demonstrated scalability. Here we report a similar experiment to demonstrate that DaisyNFS can take advantage of GoJournal's scalability, after accounting for the transaction system's two-phase locking and any overhead added by the transactions themselves. The benchmark used is the smallfile benchmark from §9.1, with a varying number of cores. Because this experiment runs on a physical drive, other threads have a chance to prepare transactions while the journal is committing to disk.

The results are shown in Figure 14. The graph shows that DaisyNFS gets higher throughput with more clients, though its scalability is not as good as the Linux NFS server and its peak throughput is 60% that of Linux. DaisyNFS scales sublinearly due to a lock in GoJournal that serializes installation of writes into disk blocks at commit time. As expected, with a global transaction lock performance does not improve with more clients.

9.3 Testing the trusted code and spec

For the NFS server to satisfy Theorem 1, we trust that (1) the Dafny code is a "safe" use of the transaction system, (2) sequential refinement is correctly encoded into Dafny, (3) the libraries for Go primitives are correctly specified in Dafny, and (4) the unverified Go code calling the Dafny methods

Bug	Why?
XDR decoder for strings can allocate 2^{32} bytes	Unverified
File handle parser panics if wrong length	Unverified
WRITE panics if not enough input bytes	Unverified
Directory REMOVE panics in dynamic type cast	Unverified
Panic on unexpected enum value	Unverified
Concurrent writes can conflict	Unverified
The names `.` and `..` are allowed	Not in RFC 1813
RENAME can create circular directories	Not in RFC 1813
CREATE/MKDIR allow empty name	Specification
Proof assumes caller provides bounded inode	Specification
RENAME allows overwrite where spec does not	Specification

Figure 15: Bugs found by testing at the NFS protocol level.

and implementing the NFS wire protocol is correct. Finally, the user must follow the assumed execution model and run initialization from an empty disk, run recovery after each boot, and the disk should preserve written data and not corrupt it.

Beyond satisfying this formal theorem statement, we want two more things from the implementation and specification: first that the specification as formalized actually reflects the RFC, and second we would like DaisyNFS to be compatible with existing clients, including implementing enough of the RFC's functionality. These fall outside the scope of verification so we cover them with testing.

To evaluate the file system we mounted it using the Linux NFS client and ran the fsstress and fsx-linux tests, two suites used for testing the Linux kernel. In order to look for bugs in crash safety and recovery, we also ran CrashMonkey [30], which found no bugs after running all supported 2-operation tests.

While elsewhere in this paper we interact with DaisyNFS via the Linux client, a collaborator (but not an author) tested it more directly using an NFS-specific testing tool.[2] This testing produces a wider range of requests than are possible via the Linux client. This process helped us find and fix several bugs in the unverified parts of DaisyNFS and in the specification itself. These are reported in Figure 15.

Two of the specification bugs are particularly interesting. The bounded inode bug was due to an `ino` argument of type `Ino`; this type is a Dafny *subset type*, thus adding an implicit precondition that `ino < NUM_INODES`, which is violated by the (unverified) Go code. The fix is to instead use a `uint64` and check the bound in verified code. The RENAME bug was due to having an incomplete specification (and implementation) that did not capture that RENAME should only overwrite when the source and destination are compatible.

9.4 Incremental improvements

DaisyNFS was implemented and verified over the course of three months by one of the authors, until it had support for enough of NFS to run. We added several features incrementally after the initial prototype worked, both to improve performance and to support more functionality. Some of the interesting changes are listed in Figure 16. To improve performance, we switched to operating on the serialized representation of directories directly (decoding fields on demand and encoding in-place) and then added also multi-block directories. We added support for attributes so that the file system stores the mode, uid/gid, and modification timestamp for files and directories. Finally, we implemented the freeing plan described in §7.1.2, which required additional code through the whole stack (but by design no changes to the file-system invariant). We believe additional features such as symbolic links could be added incrementally with modest effort because of sequential reasoning and proof automation.

Feature	Time	Lines
In-place directory updates	2 days	600
Multi-block directories	5 days	800
NFS attributes	4 days	500
Freeing space (§7.1.2)	3 days	1400

Figure 16: Incremental improvements were implemented quickly and without much code (which includes both implementation and proof).

10 Conclusion

This paper presented DaisyNFS, a verified crash-safe, concurrent file system. DaisyNFS was built with verification in mind in two parts: a transaction system called GoTxn, and a file system on top implemented with one transaction per operation. This design allowed us to use the sharpest tool for each part: Perennial for concurrency and crash-safety reasoning and Dafny for sequential reasoning with much proof automation inside a transaction. The specification of the transaction system was designed to support sequential reasoning from Dafny. Overall this approach results in proof overhead of about $2\times$ for the file system part (vs. $20\times$ for the transaction system), allowing us to verify and build a functional file system with good performance.

Acknowledgments

Many people helped improve this paper, including the anonymous reviewers, the PDOS students who gave feedback, Henry Corrigan-Gibbs, and our shepherd, Manos Kapritsos. James Wilcox provided expert debugging assistance. Robert Morris tested DaisyNFS and reported the bugs in Figure 15. This research was supported by NSF awards CNS-1563763 and CCF-1836712.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[2] This framework is part of an unrelated research project so we unfortunately lack space to give details on the methodology itself.

[2] Stefan Bodenmüller, Gerhard Schellhorn, Martin Bitterlich, and Wolfgang Reif. Flashix: Modular verification of a concurrent and crash-safe flash file system. In *Logic, Computation and Rigorous Methods*, pages 239–265. Springer International Publishing, 2021. Festschrift for Egon Börger's 75th Birthday.

[3] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[5] Tej Chajed. *Verifying a concurrent, crash-safe file system with sequential reasoning*. PhD thesis, Massachusetts Institute of Technology, May 2022.

[6] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018.

[7] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019.

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, October 2019.

[9] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[10] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[11] Dmitri Chkliaev, Jozef Hooman, and Peter van der Stok. Serializability preserving extensions of concurrency control protocols. In *Proceedings of the 3rd International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI)*, pages 180–193, Novosibirsk, Russia, July 1999.

[12] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. `http://pdos.csail.mit.edu/6.828/xv6`.

[13] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 419–434, Carlsbad, CA, October 2018.

[14] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 67–78, Bangalore, India, January 2010.

[15] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, pages 504–528, Maribor, Slovenia, June 2010.

[16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, November 2016.

[17] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 646–661, Philadelphia, PA, June 2018.

[18] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, Banff, Alberta, Canada, November 2020.

[19] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, October 2015.

[20] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 449–465, San Francisco, CA, July 2015.

[21] Dave Hitz, Michael Malcolm, and James Lau. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, January 1994.

[22] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, Santa Clara, CA, February 2015.

[23] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[25] Alex Konradi. Performance optimization of the VDFS verified file system. Master's thesis, Massachusetts Institute of Technology, June 2017.

[26] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, April–May 2010.

[27] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR)*, page 516–530, Newcastle upon Tyne, UK, September 2012.

[28] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.

[29] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[30] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.

[31] Jörg Pfähler. *A Modular Verification Methodology for Caching and Lock-Based Concurrency in File Systems*. PhD thesis, Universität Augsburg, 2018.

[32] David Harver Pollak. Reasoning about two-phase locking concurrency control. Master's thesis, Imperial College London, June 2017.

[33] Gerhard Schellhorn, Gidon Ernst, Jorg Pfähler, Dominik Haneberg, and Wolfgang Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, pages 9–24, Toulouse, France, June 2014.

[34] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–87, Portland, OR, June 2015.

[35] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional isolation for block storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 23–37, Santa Clara, CA, February 2016.

[36] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[37] The Coq Development Team. *The Coq Proof Assistant, version 8.15*, January 2022. URL `https://doi.org/10.5281/zenodo.5846982`.

[38] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the*

18th ACM SIGPLAN International Conference on Functional Programming (ICFP), pages 377–390, Boston, MA, September 2013.

[39] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helper for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.