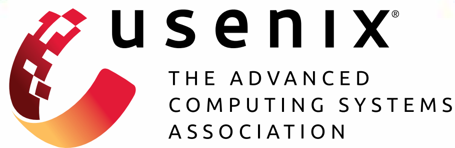


GoJournal: a verified, concurrent, crash-safe journaling system

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

https://www.usenix.org/conference/osdi21/presentation/chajed

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

GoJournal: a verified, concurrent, crash-safe journaling system

Tej Chajed *MIT CSAIL*
Joseph Tassarotti *Boston College*
Mark Theng *MIT CSAIL*
Ralf Jung *MPI-SWS*
M. Frans Kaashoek *MIT CSAIL*
Nickolai Zeldovich *MIT CSAIL*

Abstract

The main contribution of this paper is GoJournal, a verified, concurrent journaling system that provides atomicity for storage applications, together with Perennial 2.0, a framework for formally specifying and verifying concurrent crash-safe systems. GoJournal's goal is to bring the advantages of journaling for code to specs and proofs. Perennial 2.0 makes this possible by introducing several techniques to formalize GoJournal's specification and to manage the complexity in the proof of GoJournal's implementation. *Lifting predicates* and *crash framing* make the specification easy to use for developers, and *logically atomic crash specifications* allow for modular reasoning in GoJournal, making the proof tractable despite complex concurrency and crash interleavings.

GoJournal is implemented in Go, and Perennial is implemented in the Coq proof assistant. While verifying GoJournal, we found one serious concurrency bug, even though GoJournal has many unit tests. We built a functional NFSv3 server, called GoNFS, to use GoJournal. Performance experiments show that GoNFS provides similar performance (e.g., at least 90% throughput across several benchmarks on an NVMe disk) to Linux's NFS server exporting an ext4 file system, suggesting that GoJournal is a competitive journaling system. We also verified a simple NFS server using GoJournal's specs, which confirms that they are helpful for application verification: a significant part of the proof doesn't have to consider concurrency and crashes.

1 Introduction

Storage systems, such as file systems, need to be carefully structured to not lose persistent user data, even in the face of application and whole-system crashes. They often achieve this *crash safety* property by delegating writing to storage to a *journaling system*, which exposes an API for executing an operation such that its writes appear on disk atomically. The journaling system simplifies implementing the storage system's logic: to atomically modify a set of objects, the file system simply writes to them one at a time within a single journal operation. The result is that each storage operation is atomic with respect to crashes.

While a journaling system exposes a simple API, its implementation must address crash safety and also be concurrent for good performance. Maintaining correctness in the presence of both concurrency and crashes is challenging. For example, in pursuit of performance, journaling systems often avoid holding locks while performing I/O, but reasoning about the correctness of such optimizations requires considering what happens if one thread's disk reads interleave with another thread's disk writes, and what happens when the system crashes anywhere during that interleaving.

This paper presents GoJournal, a Go package that provides the first formally verified concurrent journaling system. To verify GoJournal, we developed Perennial 2.0, an extension to the Perennial [5] framework with several features designed to enable modular reasoning about concurrent, crash-safe systems. In this work we set a goal of giving GoJournal a specification that *reflects the simplicity of using a journal for crash atomicity*. GoJournal can be used by an application like a file system or a key-value store. As long as the application follows a locking discipline for its on-disk state, such as per-file locks for a file system, proving the correctness and crash-safety of that implementation on top of GoJournal should involve largely *sequential* reasoning, despite the fact that the application has multiple concurrent threads and can crash at any time.

Realizing this goal raises two challenges: specifying GoJournal in a way that makes application reasoning sequential, and proving GoJournal's implementation correct. The specification makes reasoning about an operation sequential with a *lifting* interface where the proof has an abstraction of a "checked out" private fragment of the disk that the operation appears to synchronously modify. At commit time the private fragment is "checked in", at which point it is durable and can be exposed to other threads. The journal guarantees the operation is atomic by delaying all writes to commit time, so the developer should not need to explicitly reason about crash safety until commit time. Perennial 2.0 supports a new technique called *crash framing* to formalize the intuition that during an operation the developer need not explicitly consider crash safety.

The second challenge lies in proving GoJournal itself. This is difficult because we desire modularity to make the system's proof tractable, which requires giving suitable specifications to the internal interfaces of the system. While the user-visible interface of GoJournal is simple, the internal interfaces of a high-performance journaling system are hard to specify and fit together. To address this challenge, Perennial 2.0 contributes *logically atomic crash specifications* which enable natural specifications of system layers in terms of a transition system with atomic transitions for the public methods. These specifi-

cations include a *crash transition* to describe what happens to the state of a layer during a crash. Such specifications make it possible to build upper layers of the system on top without worrying about implementation details of how atomic transitions are achieved. This separation of concerns lines up with the modularity in the implementation; the proof layers divide up the reasoning along the same lines that the code divides up functionality among Go packages.

To test the performance and completeness of GoJournal, we built GoNFS, a functional (but unverified) NFSv3 server that can be mounted through the Linux NFS client. GoNFS imports GoJournal and uses it to achieve crash consistency for NFS operations. We focus on NFSv3 because it is widely used in practice, its performance matters for applications, and it has a crash-safety and correctness specification in the form of RFC 1813 [2]. The crash-safety properties are advanced; for example, the protocol supports unstable writes which let the implementation delay flushing them to disk.

On a combination of microbenchmarks and a software-development workload, GoNFS achieves at least 90% of the throughput of Linux's in-kernel NFS server exporting ext4 running on either a RAM disk or fast NVMe storage. On slower SSD storage without using unstable writes GoNFS gets 20% of Linux's throughput due to inefficient I/O. GoJournal's concurrency is crucial to performance: the throughput of GoNFS scales with the number of clients, but if GoJournal is modified to execute sequentially (as in previous verified storage systems), even with 20 clients GoNFS achieves only double the throughput of a single client.

To demonstrate that GoJournal's specifications enable effective verification of client applications, we implemented and verified a simplified NFS server, which we call SimpleNFS, covering the core operations, such as READ, WRITE, GETATTR, and SETATTR (which can shrink and grow a file). By using GoJournal's specifications, the proof for SimpleNFS largely involves crash-free reasoning (only 44 lines of code, out of a total of 462, require explicit reasoning about crashes). This translates into a lower proof overhead: SimpleNFS requires 3,749 lines of proof for 462 lines of Go code. GoJournal itself requires 25,797 lines of proof for 1,345 lines of Go code.

The contributions of this paper are (1) GoJournal, a concurrent journaling system with a machine-checked proof of correctness and crash-safety; (2) the Perennial 2.0 framework, with extensions to the original Perennial framework that enable modularity and crash-free reasoning on top of GoJournal; and (3) SimpleNFS, a verified core of an NFSv3 file server built on top of GoJournal.

Although GoJournal is advanced enough to support a high-performance NFS server, it has some limitations. GoJournal's internals (code and proof) support deferred durability, but for simplicity, GoJournal's top-level specification requires applications to immediately flush committed journal operations, which is sufficient to prove SimpleNFS. GoJournal is also less general than JBD2 (e.g., GoJournal does not support floating commit blocks), and less general than database transaction systems (e.g., GoJournal does not support undoing journaled operations). While GoJournal provides atomic updates for crash consistency, it does not implement automatic concurrency control. Objects accessed by a journal operation cannot be concurrently accessed by another thread. GoJournal provides a verified library for locking objects tracked by the journal, which clients can use to implement concurrency control.

2 Related work

To the best of our knowledge, GoJournal is the first verified concurrent, crash-safe journaling system. The verification of GoJournal builds on a large body of previous work, as described in the rest of this section.

2.1 Perennial 2.0 vs Perennial 1.0

The verification approach we take is based on a new version of our earlier Perennial [5] framework, so we draw a contrast between the two here. The new implementation is conceptually similar in that it supports reasoning about concurrency and crash-safety, it is implemented on top of the Iris [17, 18] concurrency verification system, and it uses Goose [6] to enable verification of Go programs by translating them into a model in Perennial 2.0. However, to make verification of GoJournal feasible, we had to re-write many core parts of the framework. To clarify which framework is being referenced we will write Perennial 1.0 for the original framework and Perennial 2.0 for the new one in this section, in order to highlight the new features Perennial 2.0 supports. The rest of the paper generally refers only to Perennial 2.0.

Some of Perennial 2.0's features are needed to support the GoJournal top-level specification and enable verification on top of this interface. The reason this problem is complicated is because the journal does not make operations automatically atomic but requires the caller to correctly manage ownership, and Perennial 1.0's refinement specifications do not give a good way to talk about ownership. The top-level specification of GoJournal relies on *crash framing* (§5.5) and *crash-aware locks* (§5.4) to enable application proofs that reason about ownership of durable data.

Perennial 2.0 also scales to a larger system than the mail server verified in Perennial 1.0. One of the challenges with the larger system is that it has many internal layers that need their own specifications, so that the proof can be carried out modularly. Normally a separation logic or refinement-based specification would be sufficient, but we need internal specifications that capture the crash and concurrent behavior of each internal library. To that end Perennial 2.0 incorporates a new specification style which adds *crash atomicity* to the logically atomic specification styles developed in earlier work [10, 15, 27]. Modularity in the proof was necessary to scale verification to all of GoJournal's performance optimizations and concurrency.

Method	Description	Spec
`func Begin() *Op`	Start operation	§5.2
`func (*Op) ReadBuf(addr Addr, sz uint64) *Buf`	Read a buffer	§5.3
`func (*Buf) SetDirty()`	Mark a buffer as modified	§5.3
`func (*Op) OverWrite(a Addr, sz uint64, data []byte)`	Write without reading	§5.3
`func (*Op) Commit(wait bool) bool`	Commit by appending to in-memory log. If `wait=true`, also wait until changes are on disk.	§5.6
`func Flush() bool`	Flush in-memory log	
`func (*Lockmap) Acquire(i uint64)`	Acquire ith lock	§5.4
`func (*Lockmap) Release(i uint64)`	Release ith lock	§5.4

Figure 1: GoJournal interface and API for lockmap. Not shown are auxiliary interfaces for initialization; checking operation size; etc.

At the same time, GoJournal's specification allows the proof of SimpleNFS to mostly avoid reasoning about crashes.

2.2 Related verification frameworks

Crash-safe systems. Any crash-safe system must reason about the possible states after a crash, and several prior works have formalized this in different ways for *sequential* crash-safe systems. FSCQ [7, 8] uses Crash Hoare Logic (CHL) to specify crash behavior through a crash condition, which describes the state of a system if a crash happens during execution of a function. Alternatively, a number of systems verify crash safety using refinement reasoning [4, 12, 14, 26], but none support the combination of concurrency and crash-safety.

Although they are not concurrent, some of these systems address other aspects of performant storage systems that are not found in GoJournal. DFSCQ [7] verifies a high-performance file system built on top of a logging system with asynchronous disks and log-bypass writes, which are challenging optimizations that GoJournal does not support. VeriBetrKV [14] verifies a key-value store based on B^{ε} trees, a data structure that also underlies BetrFS [16]. GoJournal and SimpleNFS use simple data structures; the challenge lies in accounting for concurrent accesses.

Concurrent systems. In addition to specifying behavior at intermediate crash points, Perennial 2.0's specifications describe the atomic commit points of concurrent operations. A range of verification techniques have been used to address this kind of challenge in concurrent systems. AtomFS [29] uses a framework called CRL-H (concurrent relational logic with helpers) to verify a concurrent in-memory file system implemented in C. Refinement-based systems such as CSPEC [3], Armada [23], and Concurrent CertiKOS [13] typically prove that a function implements an atomic operation at a more abstract layer. However, in GoJournal, many internal APIs provide operations that are only atomic if the caller owns some data. This kind of conditional atomicity is easy to express in Perennial 2.0 using separation logic, but hard to express as a precondition in a transition system.

Concurrent, crash-safe reasoning. Program logics other than Perennial have been developed for formal reasoning about concurrent, crash-safe systems. Fault-Tolerant Concurrent Separation Logic (FTCSL) [24] extends the Views [11] concurrency logic to incorporate crash-safety. POG [25] is a program logic for reasoning about the interaction of x86-TSO weak-memory consistency and non-volatile memory. Neither logic has a mechanism for modular proofs of layers, which we found essential to scale verification to a system of GoJournal's complexity. Both are restricted to pen-and-paper proofs, whereas both Perennial 1.0 and 2.0 have machine-checked proofs.

A specification called the Push/Pull model of transactions [19] is similar to the *lifting* technique in the journal system's specification (§5.2) — the core problem addressed is that a journal operation atomically modifies a small number of objects, but other objects can change between the start of the operation and when it commits. The Push/Pull model also discusses reasoning on top of the specification, using Lipton's reduction [22] rather than separation-logic ownership to handle concurrency. However that work is about on-paper specifications and proofs, while we also prove an implementation meets our specification and proved SimpleNFS on top.

3 System design

The verified artifact of this paper is GoJournal, a Go package that gives clients an abstraction of a disk with crash-safe writes. This section aims to convey what the journal is, why its implementation deserves verification, and how systems can be built using it. First, §3.1 explains how a developer uses GoJournal to write a concurrent storage system, informally laying out what the package's requirements and guarantees are. Then, §3.2 explains how the journal is implemented.

3.1 Programming with GoJournal

Developers use the journal to turn several storage operations into an atomic journal operation that commits to disk using the GoJournal interface listed in Figure 1. `Begin` starts a journal operation, returning a `*Op` object, which keeps track of the objects read or written in the operation. An object is addressed by the `Addr` struct, which names a block address and bit offset within the block. SimpleNFS has objects for on-disk blocks

```go
func NFS3_WRITE(args WRITE3args) WRITE3res {
  inum := fh2ino(args.File)
  if !validInum(inum) {
    return WRITE3res{Status: NFS3ERR_INVAL}
  }
  inode_locks.Acquire(inum)
  reply := NFS3_WRITE_locked(args, inum)
  inode_locks.Release(inum)
  return reply
}

func NFS3_WRITE_locked(args WRITE3args,
    inum Inum) (reply WRITE3res) {
  op := Begin()
  if !NFS3_WRITE_op(op, args, inum, &reply) {
    return
  }
  if txn.Commit(true) {
    reply.Status = NFS3_OK
  } else {
    reply.Status = NFS3ERR_SERVERFAULT
  }
  return
}
```

Figure 2: RPC handler for NFS WRITE showing locking and committing a journal operation.

and on-disk inodes, while the complete NFS server also uses objects for individual allocator bits.

`ReadBuf` reads an object into an in-memory `*Buf` struct, returning the latest value of the object within this journal operation. If the operation hasn't read the object yet, it reads the latest value from disk (or from a recently committed operation). A journal operation can modify the returned buffer in place and then mark the buffer as dirty with `SetDirty`. To overwrite an object without reading it the application can call `OverWrite`. When the operation is fully prepared, the application commits it atomically using `Commit`; setting `wait=true` additionally forces the journal to flush the results to disk. In either case the writes in the operation appear together on disk or not at all even if the system crashes. The application can also call `Flush` to make the journal persist several committed but unstable operations to disk.

While GoJournal provides crash-safe atomic updates to disk with this interface, it is the developer's job to provide concurrency control to prevent concurrent operations from manipulating the same on-disk objects. In a file system a common strategy for concurrency control is to use a per-file lock that protects both the file metadata and any data blocks associated with the file, and this strategy is the one used by GoNFS and SimpleNFS. To make it easier for a file system to maintain these locks, GoJournal includes a lockmap library that behaves as if it were a large array of locks but with a more memory-efficient implementation; the Guava Striped documentation describes the idea well [1].

Figure 2 and Figure 3 show how SimpleNFS uses the Go-

```go
func NFS3_WRITE_op(op *Op, args WRITE3args,
    inum Inum, reply *WRITE3res) bool {
  ip := ReadInode(op, inum)
  count, ok := ip.Write(op, args.Offset,
        args.Count, args.Data)
  ... // set count and status
}

func (ip *Inode) Write(op *Op, off uint64,
    count uint64, data []byte) (uint64, bool) {
  if count != uint64(len(data)) ||
     util.SumOverflows(off, count) ||
     off+count > disk.BlockSize ||
     off > ip.Size {
    return 0, false
  }

  buf := op.ReadBuf(block2addr(ip.Data),
        NBITBLOCK)
  copy(buf.Data[off:], data)
  buf.SetDirty()
  if off+count > ip.Size {
    ip.Size = off + count
    ip.WriteInode(op)
  }
  return count, true
}

func (ip *Inode) WriteInode(op *Op) {
  op.OverWrite(inum2Addr(ip.Inum),
    INODESZ*8, ip.Encode())
}
```

Figure 3: `NFS3_WRITE_op` prepares a journal operation `op` for the WRITE RPC.

Journal API and the lockmap. The server runs each NFS request in a separate Go thread running a single journal operation. Figure 2 shows the RPC handler for an NFS WRITE RPC, in particular acquiring a per-inode lock (lines 6 and 8) and preparing an operation starting at line 14.

The handler is split into several nested functions for ease of verification. Figure 3 shows how the WRITE RPC's journal operation of type `*Op` is prepared. For example, lines 18–21 read and modify the block data, while line 30 modifies the inode. The combination of per-file locking and using the journal for disk access frees the developer from thinking about either concurrency or crashes during the entire `NFS3_WRITE_op` code, which we will show is also the case in the proof using Perennial's specification techniques in §5.

For ease of explanation, SimpleNFS has the limitation that each file consists of only one block, but note that WRITE modifies two on-disk objects: the inode and the block owned by the file; the two together must be written atomically, which the proof shows using the GoJournal specification. Also note that there is no explicit locking of blocks; ownership of the data block is implicit because a block can belong to only one file.

Layer	Description
JRNL	In-memory object operations
OBJ	Journaling sub-block writes
WAL	Whole-block write-ahead logging
CIRCULAR	Circular log structure

Figure 4: GoJournal layers.

3.2 GoJournal implementation

The journal is structured into several layers, as shown in Figure 4. At a high level, the system is split into two halves. The low-level half is a write-ahead log that behaves like a disk with an atomic multiwrite operation, which appears to update multiple disk blocks simultaneously even if the system crashes. The upper half, called the object system, allows callers to perform read and write operations on objects smaller than a block ("sub-block" objects). Writes are buffered in memory until the caller chooses to commit, at which point a multiwrite to the write-ahead log commits the writes to disk.

The write-ahead log is implemented by organizing the disk into a small, fixed-size circular buffer and a remaining data region. Data is first atomically *logged* to the circular buffer and then eventually *installed* to the data region, to free space in the circular buffer. Reads first go through the circular buffer (which is cached for efficiency) and then access the data region.

The object system maintains a list of buffers of data read or written by each journal operation. Reads first check the write-ahead log's cache since they must observe committed operations. To commit, the object layer gathers all the dirty buffers and submits them as a multiwrite to the write-ahead log. To allow reading and writing objects that are smaller than a block, the object layer assembles these into block writes by doing a read-modify-write sequence.

Because disk writes are slow, for good performance the journal executes many tasks in parallel. Committing new journal operations in memory, logging operations from memory to disk, waiting for operations to be made durable, and installing logged writes all happen concurrently. Concurrency ensures that in-memory operations need not wait for any in-flight disk reads or writes, and that many disk reads and writes can happen at the same time. Finally, to reduce the number of disk writes, the write-ahead log implements two optimizations. Multiwrites are combined and written together ("group commit"), and if they update the same disk block multiple times, only the most recent update of that disk block is written to the log ("absorption"). Concurrency makes these optimizations useful even for synchronous operations, which can be committed together and absorbed if they are issued concurrently.

Concurrency in the write-ahead log complicates not just its internals but also reasoning about the multiwrite abstraction built on top. One difficulty is that reading requires checking the log's in-memory cache and then falling back to the disk, but the disk read happens without a lock. If a multiwrite commits after the read misses in the cache, then the disk read will not observe the latest value. The write-ahead log specification specifies that reading the installed value might return an old view of the disk, and the object layer can handle this weak specification with an invariant that guarantees the object being read has not been modified since that old view.

The object layer implements sub-block access on top of the write-ahead log's block-level multiwrites. Objects accessed by an operation must be locked, so supporting fine-grained access is necessary to allow operations to run concurrently even if they happen to access the same disk block. For example, a file system might pack inodes into a block, and locking an inode should not prevent concurrent operations for other inodes in the same block. The object-layer implementation is able to execute reads and writes during an operation without any additional locks, but something more is needed to commit. Imagine a situation where between reading some disk block and writing it an unrelated object was modified in the same block; committing the modified block would overwrite the concurrent modification, losing data. The code addresses this with a global commit lock that prevents concurrent modifications while reading the blocks to be written.

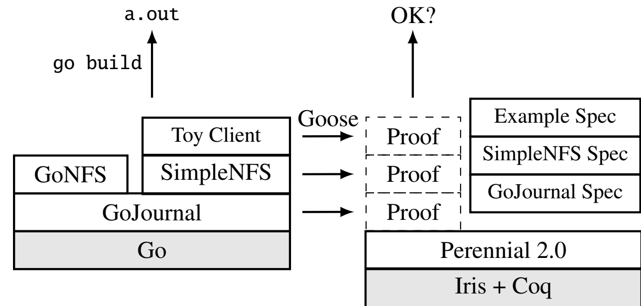


Figure 5: Overview of Perennial, GoJournal, SimpleNFS, and GoNFS.

4 Verification overview

Figure 5 gives an overview of how GoJournal and systems building on it are verified using Perennial. On the left of the figure is the executable code, which is written in Go. On top of GoJournal, we have implemented two NFS servers to evaluate GoJournal along different dimensions. GoNFS is a functional NFS server that is sufficient to run real applications, which we use to assess GoJournal's scalability and performance. Meanwhile, SimpleNFS is a verified, core subset of GoNFS's functionality, which evaluates the usability of GoJournal's specs for building verified systems on top of it.

On the right side of the figure is the verification stack. The verification builds on the Perennial 2.0 framework, which is itself implemented in the Iris framework in the Coq proof assistant. To reason about executable code, a tool called Goose translates a Go implementation into a model that we can prove specifications about in Perennial. Perennial provides a model of execution for Go code that incorporates crash-safety and

concurrency, which includes a model of the disk (with atomic, synchronous reads and writes of 4KB sectors) as well as a model of crashes and recovery (crashes at arbitrary points during execution, and jumping to specific boot code for recovery after a crash).

GoJournal's top-level specification describes its API in terms of an extension of concurrent separation logic, with pre-, post-, and crash conditions. These specifications capture the behavior of individual Go functions: if the function is run in a state satisfying its precondition, then the final state will satisfy the postcondition, and if the system crashes the state will satisfy the crash condition. The specification for the journaling API is described in detail in §5. We demonstrate the usefulness of this specification by proving correctness of the SimpleNFS server using logically atomic crash specifications (§6.2). The top-level theorem for SimpleNFS states that its RPCs atomically follow transitions of a state machine formalizing the NFSv3 protocol (based on RFC 1813 [2]).

As described in FSCQ and Argosy [4, 8], crash conditions can be used to reason about recovery procedures, even crashes during recovery. A recovery procedure can safely be re-run after a crash if its specification is *idempotent*: its crash condition should imply its precondition. As an end-to-end check of the crash specs in SimpleNFS and GoJournal, showing they support recovery correctly, we prove an idempotent specification for a toy example client on top of SimpleNFS, establishing that it can successfully execute even if SimpleNFS crashes and recovers an arbitrary number of times.

The proof of GoJournal's specification depends on a number of assumptions. We assume that the disk writes 4KB blocks atomically, even on crash, and assume that the code executes according to the Perennial model generated by Goose. The specification relies on the caller to provide concurrency control; the proof of SimpleNFS checks that locking is performed correctly, but GoNFS is unverified and we trust that its concurrency control is correct in order to make operations atomic (though this does not say they correctly implement the NFS specification).

5 Specifying GoJournal

The goal of GoJournal's specification is to support convenient reasoning about atomic operations, like the NFS WRITE implementation in Figure 2 and Figure 3. In this section we walk through how the specification guarantees atomicity for the caller without forcing the caller to do much application-specific reasoning about concurrency or crashes.

The key to this specification is tracking resources, like the disk blocks making up a file, as they flow through the steps of the proof. We start by reviewing how separation logics like Perennial represent these resources, and how specifications in the logic track logical *ownership* of resources (§5.1). The specification for GoJournal introduces resources that distinguish between a journal operation's local view of an object and the durable, on-disk representation; obtaining either resource requires the caller to use correct synchronization, as required by the journal's implementation. *Lifting* provides a way to translate a locked object from its on-disk view to a local view within the operation (§5.2). While preparing a journal operation, reads and writes modify the local view (§5.3). Finally, committing an operation writes its updates to disk, so the specification asserts that the local view becomes a view over durable state.

To take full advantage of the durable and operation-local views of journal objects, the proof of WRITE uses two new techniques introduced by Perennial 2.0: crash-aware locking (§5.4) and crash framing (§5.5). With these techniques, the proof of `NFS3_WRITE_op` uses entirely sequential reasoning for preparing the journal operation, even though concurrent operations might write to disk and its disk writes are buffered rather than synchronous. Finally, §5.7 summarizes how the proof techniques combine to prove correctness and crash-safety for the NFS WRITE example.

5.1 File representation

First, in both designing the code and writing the proof, the NFS server must establish a disk layout to arrange its data in terms of disk objects. The disk layout is expressed using a separation-logic *representation invariant*, a predicate which connects the logical (specification-level) contents of files to the objects (inodes and blocks) that encode those files.

Representation invariants over the state of the journal use a "points-to predicate" $a \mapsto o$, which serves two purposes: it asserts that the address a (of type `Addr`) contains an object o (which is represented by the `*Buf` type in the API), and it represents exclusive *ownership* over the address a. When a thread has $a \mapsto o$ in its precondition, ownership allows the proof to assume that the value at address a does not change until the thread gives up ownership, and that it will not be read by other threads. Locks help threads transfer ownership so a thread only retains exclusive ownership during a critical section.

The SimpleNFS proof connects each file to its representation with the following representation invariant:

$$\begin{aligned}&\texttt{file_rep}(i, \mathit{data}) \triangleq \exists \mathit{meta}, \exists \mathit{blk},\\ &\quad i \mapsto \mathit{meta} * \mathit{meta}.\texttt{blkno} \mapsto \mathit{blk} \land\\ &\quad \mathit{meta}.\texttt{size} = \texttt{length}(\mathit{data}) \land \texttt{prefix}(\mathit{data}, \mathit{blk})\end{aligned}$$

Informally the representation invariant says the file i with logical contents *data* is represented by some metadata *meta* stored at the inode number i and a data block at *meta*.`blkno`. It then says the file's bytes are a *meta*.`size`-length prefix of the data block.

This definition uses the separating conjunction $P * Q$ (pronounced "P and separately Q"), which says that two predicates hold over disjoint state. For example, this asserts the inode and its data block are stored separately. To initialize the system the caller must prove that the `file_rep` predicates hold separately

for each file, that is, $\texttt{file_rep}(1, data_1) * \texttt{file_rep}(2, data_2) * \cdots$. Here the separating conjunction asserts files are represented disjointly, so that when a thread modifies one file it is guaranteed not to affect data in other files.

5.2 Lifting

The key idea of GoJournal's specification is to consider two view of the disk: a conceptual in-memory view that a buffered journal operation observes, as well as an on-disk view that reflects what would be on disk after a crash. Parts of both views are constantly changing as other threads commit operations concurrently, so we use separation logic to define a local view that contains only objects locked by and involved in a journal operation. Because the journal operation logically owns these objects, the caller can use sequential reasoning—disk objects have the same value throughout—and can commit all of the objects written in the operation at the end without fear of interfering with concurrent journal operations. The specification makes this informal reasoning concrete using *lifting*, which we use to refer to this strategy of transferring ownership to and from the on-going operation.

To do anything with the journal, a thread must first `Begin` an atomic operation:

$$\begin{array}{l}\{\text{True}\}\\ \quad \texttt{Begin}()\\ \{\textbf{ret } op,\ \texttt{is_op}(op) * \texttt{durable_pred}(op, \text{True})\}\end{array}$$

The specification above is a Hoare triple for the `Begin()` function. It says that executing `Begin()` starting with its precondition (in this case True) will run without errors and if it terminates it will return *op* along with the postcondition, namely $\texttt{is_op}(op) * \texttt{durable_pred}(op, \text{True})$. The `is_op` part of the post-condition simply says that *op* is a valid `*Op` object. The $\texttt{durable_pred}(op, \text{True})$ clause is what tracks the on-disk data "underneath" a journal operation, which would be left behind if the operation aborted; since the operation starts out with an empty local view, it starts out with no on-disk footprint, written as True.

The different views of a journal operation are tracked using *ghost state* in Iris. Ghost state is separate from the physical state of the program—the contents of memory and disk—and is only manipulated by the proof. The journaling system's proof introduces ghost state for durable state of the system, including an $a \mapsto_d o$ predicate for ownership over individual objects. Note that an object is expressed through ghost state because the block holding the object might be located in the on-disk log or in the data region, and ownership of an object says nothing about other objects in the same disk block.

The proof also introduces a similar $a \mapsto_{op} o$ predicate for the local view of operation `op`, and it is this ownership that is needed for reads and writes. A caller obtains these predicates with a logical operation we call *lifting* that converts ownership of $a \mapsto_d o$ into $a \mapsto_{op} o$, granting the ability to read and write.

To make it easier to work with lifting, the specification allows lifting an entire *predicate* P and transforms all of its points-to facts simultaneously, which we denote this paper denotes by switching subscripts. For example, we re-use the definition `file_rep` from §5.1 for both a file laid out on disk and a file as owned by a journal operation, which we denote with $\texttt{file_rep}_d$ and $\texttt{file_rep}_{op}$ respectively. The specification for lifting a generic predicate P is:

$$\begin{array}{l}\{P_d * \texttt{durable_pred}(op, Q_d)\}\\ \quad \textit{noop}\\ \{P_{op} * \texttt{durable_pred}(op, P_d * Q_d)\}\end{array}$$

Since lifting is purely logical (it only modifies ghost state), we write it as a Hoare triple for a no-op, much like how Dafny and F* lemmas are simply methods with pre- and post-conditions but no code [21: §12.2.3].[1] The outcome of lifting is to expand the memory covered by the journal operation to incorporate P_d. Observe that `durable_pred` is expanded to "snapshot" P_d, which tracks that if the operation were to abort or crash, the durable P_d that we started with would still hold. The on-disk values do not change over the course of a buffered journal operation (as expected, since these are in-memory writes). The key part of the postcondition, however, is P_{op}: the $a \mapsto_{op} o$ predicates within P_{op} (e.g., the $i \mapsto_{op} meta$ within $\texttt{file_rep}_{op}(i, data)$) give the caller the right to read and write objects from within the operation, as we will see in §5.3.

5.3 Reads and writes

The specification for `OverWrite` describes the effect of writing to the local memory of a buffered journal operation:

$$\begin{array}{l}\{\texttt{is_op}(op) * a \mapsto_{op} o * \texttt{buf_obj}(buf, o')\}\\ \quad op.\texttt{OverWrite}(a, buf)\\ \{\texttt{is_op}(op) * a \mapsto_{op} o'\}\end{array}$$

The precondition includes $\texttt{buf_obj}(buf, o')$ to say that the in-memory buffer *buf* encodes the object to be written o'. The `is_op` predicate is both required and returned by the specification, which reflects the fact that `OverWrite` operates on the in-memory state covered by this predicate.

The specification for `ReadBuf` is more subtle. `ReadBuf` returns a buffer that the caller is allowed to modify in-place, which has the side-effect of updating the in-memory state of the ongoing journal operation, which will in turn be committed by `Commit`. Figure 3 shows an example, where lines 18–20 modify a read buffer in-place. The specification captures this

[1] In case the reader is already familiar with Iris, these Hoare triples represent what is usually called a "view shift" in Iris.

behavior as follows:

$$\{\texttt{is_op}(op) * a \mapsto_{op} o\}$$
$$op.\texttt{ReadBuf}(a)$$
$$\left\{ \mathbf{ret}\, buf, \begin{array}{c} \texttt{buf_obj}(buf, o) * \\ (\forall o', \texttt{buf_obj}(buf, o') \twoheadrightarrow \\ \texttt{is_op}(op) * a \mapsto_{op} o') \end{array} \right\}$$

This states that, when ReadBuf finishes, it returns a buffer *buf* and two resources: $\texttt{buf_obj}(buf, o)$ says the buffer has the old object o, while the second is a separating implication or *wand* $\twoheadrightarrow$. The wand says that if the caller modifies the buffer to produce $\texttt{buf_obj}(buf, o')$ for some other data o' (or leaves it unchanged, picking $o' = o$), it can get back the $\texttt{is_op}(op)$ predicate, along with a $a \mapsto_{op} o'$ fact indicating that a has been modified in-place to the new data o'.[2] The wand is just another resource that the caller can invoke at the right time in the proof (e.g., after the call to SetDirty in Figure 3 on line 21).

5.4 Crash-aware locking

As seen in Figure 2, the NFS server acquires a per-file lock (within the lockmap) to prevent concurrent access to the same disk object. Each lock logically protects both the file metadata stored in its inode and the data block pointed to by the inode. The usual specification for a lock in concurrent separation logic says that it protects some lock invariant, guaranteeing that this invariant holds upon acquiring the lock and conversely obliging the caller to prove the lock invariant to release. This invariant may claim ownership of resources which are then owned by clients during their critical section. The file i's lock in SimpleNFS protects roughly $\texttt{file_rep}_d(i, data)$, where we write d to indicate the file is laid out on disk; we make the invariant more precise later when we connect it to crash safety.

This lock specification, however, is insufficient to prove that the SimpleNFS server maintains all relevant invariants when the system crashes. The specification makes no guarantees about the protected data during a critical section—however, a crash while the lock is held exposes any durable data that was protected by the lock. The lock specification fails to express that the lock holder should keep the durable data in a state that can be recovered from after a crash.

To solve this problem, Perennial 2.0 contributes a new specification for locks called *crash-aware lock specifications* that is useful for protecting durable data like $\texttt{file_rep}_d$. We proved this specification both for ordinary locks (`*sync.Mutex` in Go) and for the stripes in the lockmap, but here we present just the lockmap version. With this specification, the proof associates not just a lock invariant but also a *crash obligation* $I_c(\texttt{i})$ to each file. Like the ordinary lock specification, acquiring the lock gives the caller access to the lock invariant $I(\texttt{i})$, but unlike that spec, this specification also obliges the caller to prove the crash obligation $I_c(\texttt{i})$ at every intermediate step. The proof enforces this using crash specifications: $\{P\}\ e\ \{Q\}\{Q_c\}$ is like a Hoare triple but it has an extra predicate Q_c, the crash condition, describing what holds if the system crashes during e's execution. When the caller wants to prove something about code that acquires a lock using the crash-aware specification, it must do so with $I_c(\texttt{i})$ in its crash condition for the critical section:

$$\{P * I(\texttt{i})\}\ \texttt{f()}\ \{Q * I(\texttt{i})\}\{I_c(\texttt{i})\}$$
$$\vdash \{P\}\ \texttt{Acquire(i); f(); Release(i)}\ \{Q\}$$

In exchange for the extra work of having to prove a crash specification, the crash-aware lock spec guarantees that the lock's crash obligation holds at crash time, ready to be used by new threads spawned following the crash.

One final subtlety in the specification is that Perennial distinguishes between the disk while running d_k and the new disk following a crash d_{k+1}, where k is a so-called *generation number*. This creates a distinction between the invariant protected by the lock (in generation k) and the crash obligation (in the next generation):

$$I(i) \triangleq \exists data, \texttt{file_rep}_{d_k}(i, data)$$
$$I_c(i) \triangleq \exists data, \texttt{file_rep}_{d_{k+1}}(i, data)$$

It is important that on crash the developer show `file_rep` holds in the post-crash generation d_{k+1}, because any ephemeral resources in the current generation do not survive to the next. Any in-memory state the system requires has to be reconstructed from only the durable state.

5.5 Crash framing

As we have seen, acquiring a crash-aware lock imposes that the crash obligation holds at every step until the crash lock is released. For example, the developer must show that the crash obligation $I_c(i)$ holds at every step of `NFS3_WRITE_locked`. However, much of the code for `NFS3_WRITE_locked` resides in `NFS3_WRITE_op`, which modifies only in-memory state. This presents an opportunity to simplify the proof: because no durable state is modified, the developer should not need to think about crashes at each individual step.

Perennial 2.0 formalizes this using the *crash framing* technique, expressed in the following rule:

$$\{P\}\ \texttt{f()}\ \{Q\}$$
$$\vdash \{I_c * P\}\ \texttt{f()}\ \{I_c * Q\}\{I_c\}$$

Informally, this rule says that if we currently own the crash condition I_c, we can temporarily "give up" access to that ownership when proving `f()`. In exchange, the crash condition is removed from our proof obligation: it is sufficient to prove a regular crash-free Hoare triple for `f()`. I_c is not available for the proof of `f()` (this is the "giving up" aspect of crash framing), but the proof can continue to use I_c after the call to `f()` returns.

[2] To simplify the presentation, we have omitted the obligation that forces the caller to call *buf*.SetDirty() before getting back is_op.

The proof of `NFS3_WRITE` gets access to $I(\texttt{i})$ by acquiring the `i`th lock, lifts the $\texttt{file_rep}_{d_k}$ predicate into its buffered operation, and then immediately uses the crash framing rule to give up access to $\texttt{durable_pred}(op, \texttt{file_rep}_{d_k})$ and prove the crash condition for the duration of `NFS3_WRITE_op` (which only manipulates the in-memory $\texttt{file_rep}_{op}$). The crash framing rule gives back the `durable_pred` predicate at the end of the operation, which is required to reason about commit.

5.6 Commit

The remainder of the proof after preparing $\texttt{file_rep}_{op}$ with the new data is to reason about committing the operation with the new file. The code commits this operation using the following specification for `Commit`:

$$\begin{array}{l} \{Q_{op} * \texttt{is_op}(op) * \texttt{durable_pred}(op, P_d)\} \\ \quad op.\texttt{Commit}(\text{true}) \\ \{\textbf{ret } ok, \text{ if } ok \text{ then } Q_d \text{ else } P_d\} \\ \{P_d \lor Q_d\} \end{array}$$

This specification nicely captures how `Commit` works: if we started with data P_d on disk, then modified it to Q_{op} in memory, then if `Commit` succeeds the new data Q_d is on disk. If `Commit` fails (which happens if the journal operation is too large to fit on disk) then the data reverts back to P_d. On crash either of these could happen, depending on when the crash occurs.[3]

The caller will sometimes start an operation and then abort it, say due to encountering an error. The API has no method for this because aborting is a purely logical operation that restores ownership of the on-disk objects:

$$\{\texttt{durable_pred}(op, Q_d)\}\ noop\ \{Q_d\}$$

The Commit proof internally executes the same logical operation when the commit fails in order to return the original durable data.

5.7 Summary

The combination of above features mean the developer is mostly left with sequential crash-free reasoning about how each operation (for example, each NFS3 RPC implementation) transitions from the representation invariant in one state to another, following the transition system of the specification. We illustrate that proof flow using the `NFS3_WRITE` call in Figure 2 as an example.

First, the function starts a journal operation and acquires a lock on `i`. Then the proof requires some purely mechanical work to lift the lock invariant (§5.2) and frame the crash obligation (§5.5). Next, the developer proves the correctness of the sequential code. This proof does involve the bulk of the application code, but it requires neither worrying about concurrency (since reads and writes operate on the exclusive ownership of $a \mapsto_{op} o$) nor about crash safety (since crash framing has dismissed any crash obligations while reasoning about the in-memory operations on the `*Op`).

The sequential code must prove that the reads and writes with `ReadBuf`, `SetDirty`, and `OverWrite` transform $\texttt{file_rep}_{op}(i, data)$ to produce $\texttt{file_rep}_{op}(i, data')$, where $data'$ is the correct state of the file as described by the transition of the formalized NFS state machine for a write. The new file representation with contents $data'$ is the Q_{op} in the precondition to `Commit`'s specification, while P_d is the old file with contents *data* on disk (snapshotted while lifting).

If the system doesn't crash and `Commit` returns true, then the operation succeeds, producing a new file representation $\texttt{file_rep}_{d_k}(i, data')$. If the operation fails (say due to not fitting in the log), then `Commit` returns the old representation invariant with contents *data*. On crash, either of these two is possible, but not some inconsistent combination of the two, guaranteeing crash atomicity.

The proof for `NFS3_WRITE` wraps up by releasing the lock. Whether or not `Commit` succeeds, we have a file with some contents: $\exists data, \texttt{file_rep}_{d_k}(i, data)$; this is exactly the lock invariant $I(i)$ required to release the lock.

6 Verifying GoJournal

GoJournal consists of multiple layers, as described in §3.2. This section provides some highlights of the complexity involved in GoJournal's implementation, along with the proof techniques required to formally reason about that complexity.

6.1 Write-ahead logging (WAL)

The write-ahead log layer is responsible for updating multiple disk blocks (a multiwrite) atomically. Each multiwrite is a list of updates, where an update consists of a disk block number and the new data to write in that block. A background logger thread moves multiwrites from an in-memory buffer to an on-disk log. To make this atomic, the logger first writes the contents of a multiwrite in a log entry, and then updates a designated header block to indicate the entry is complete. If a crash happens before the header is updated, none of the multwrite's updates are applied; if a crash happens after the header update, the multiwrite will be applied during recovery. Meanwhile, an installer thread applies entries in the log to the disk, clearing space for new multiwrites. If a crash happens before the updates in an entry are fully installed, recovery installs the updates again from the on-disk log.

The write-ahead log implements two optimizations related to combining multiwrites. Two or more multiwrites can be *group committed* by logging them together, which still guarantees their atomicity. If multiwrites being committed together update the same block, the first update can be *absorbed* and replaced with the second. These optimizations trigger both for multiwrites that are committed without waiting for durability and also for concurrent, synchronous multiwrites.

[3] For $op.\texttt{Commit}(\text{false})$, which does not flush to disk right away, GoJournal provides a lower-level spec that allows expressing the more complex resulting crash condition.

Internal abstract state: logical log. To prove the write-ahead log layer correct, GoJournal represents the state of the write-ahead log as a logical list of multiwrites, as shown in Figure 6. Multiwrites before `memStart` have already been installed, and their log entries do not physically exist in memory or on disk. Multiwrites from `memStart` to `diskEnd` are already logged on disk. Multiwrites from `diskEnd` to `nextDiskEnd` are currently being logged from memory to disk. Finally, multiwrites between `nextDiskEnd` and `memEnd` are purely in-memory, and are eligible for absorption.

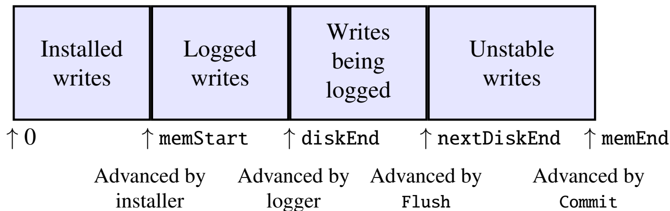


Figure 6: The logical write-ahead log. Vertical arrows indicate designated positions in the logical log. Labels below the arrows indicate what thread or function is responsible for advancing that logical position to the right.

This representation allows GoJournal to precisely specify how concurrent operations modify this abstract state, and how the state changes on crash. For example, although the installer thread performs many disk writes to install multiwrites, its only effect on the abstract state is that it advances `memStart`. Similarly, the logger thread's only change to the abstract state is to advance `diskEnd`. Calling `Flush()` advances `nextDiskEnd`, freezing the data to be logged, then waits for the logger to advance `diskEnd` up to that point. Committing a new multiwrite simply appends it at `memEnd`. Finally, on crash, an arbitrary suffix of the log from `diskEnd` onwards is discarded.

External abstract state: durable lower bound. Although the details of the logical log are important for proving the WAL layer, the caller (i.e., the OBJ layer) does not need to know about installation, group commit, etc. To abstract away these details, the WAL provides a simplified state as its interface, as shown in Figure 7. The simplified state consists of the same list of multiwrites, together with `durable_lb`, which is a lower bound on what set of multiwrites will be preserved on crash. Using a lower bound instead of precisely exporting `diskEnd` means that this abstract view does not need to change if the logger thread adds more multiwrites to disk in the background, and thus hides this concurrency.

Lock-free logging and installation. For performance, GoJournal has dedicated threads that perform logging and installation. However, these threads do not hold any locks while reading or writing to disk. To allow these threads to run concurrently, GoJournal uses two separate header blocks, as shown in Figure 8. One header block (owned by the installer thread) stores the start of the on-disk log, and another header block (owned by the logger thread) stores the end of the on-disk log.

```
Record update := { addr: u64; data: Block; }.
Record State :=
  { multiwrites: list (list update);
    (* at least durable_lb elements are durable *)
    durable_lb: nat; }.

Definition mem_append (ws: list update) :
    transition State unit :=
  modify (set multwrites (fun l => l ++ [ws]));
  ret tt.

(* non-deterministically pick how many
   multiwrites survive the crash. *)
Definition crash : transition State unit :=
  durable <- suchThat (fun s i => durable_lb s ≤ i);
  modify (set multiwrites (fun l => l[:durable]));
  modify (set durable_lb (fun _ => durable));
  ret tt.
```

Figure 7: Parts of the specification for the WAL interface.

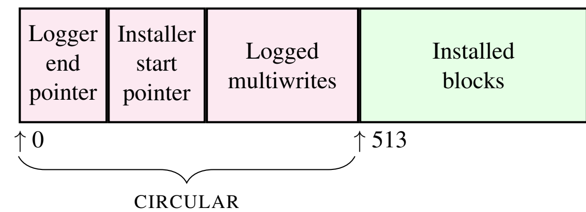


Figure 8: The physical write-ahead log.

This lets the installer and logger concurrently advance their pointers (`memStart` and `diskEnd` respectively) without locks.

Although the logger and installer threads can perform lock-free disk writes, they must still coordinate with one another. For example, the installer cannot run ahead of the logger thread, and the logger thread must coordinate with threads that are appending new multiwrites in memory. GoJournal's proof uses the notion of *monotonic counters* to reason about the safety of the logger and installer's lock-free operations.

The logger thread needs to check that `memStart` is far enough along that the log will have space for the new multiwrite. The proof gets a *lower bound* on the `memStart` variable while holding a lock, which remains true even after releasing the lock. Even though `memStart` might grow after the initial check, the log will only have more space and thus the multiwrite will still fit.

The installer has a similar lock-free region that also reasons using a lower bound. The installer retrieves the updates from the current `memStart` to `diskEnd` in order to start installing them to disk. When the installer eventually trims the log, it needs to be sure not to advance beyond the current logger position, which the proof demonstrates using a lower bound on `diskEnd` from when the logger initially started.

6.2 Logically atomic crash specifications

Throughout the GoJournal stack we specify internal layers using a transition-system specification, such as the examples

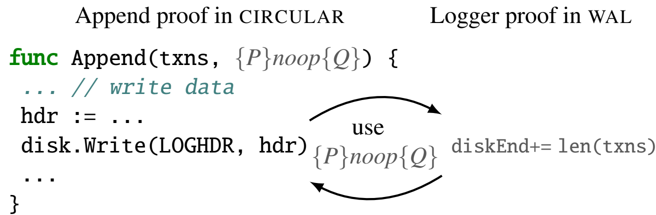


Figure 9: Illustration of how the proof of `Append` executes a logical callback $\{P\}noop\{Q\}$. The logger passes a callback that adds `len(txns)` to `diskEnd`.

illustrated in Figure 7 for the WAL layer. Perennial formalizes what it means for the code in a layer to implement a transition using Hoare triples in a style we call *logically atomic crash specifications*. While the precise encoding involves some technical details of Iris, we explain here the intuition behind these specifications as well as why they are useful.

As a motivating example, consider the moment when the logger thread commits a new batch of multiwrites to the physical log in order to advance the durable point `diskEnd` in the logical log of the WAL layer. It does this by calling into the `Append` method of the CIRCULAR layer, which appends to the small buffer of logged multiwrites. The code for `Append` commits at some internal step when it writes the header block and makes the data valid, and it is at this instant that the logical log's `diskEnd` should be incremented. How can we verify `Append` in the CIRCULAR layer separately from the WAL layer, while still executing the right update in the logger proof?

Logically atomic specifications achieve this separation by having the precondition to `Append` take a logical *callback* [15], which the proof promises to "execute" at the commit point. This callback is a Hoare triple of the form $\{P\}$ *noop* $\{Q\}$, where P and Q are later selected by the logger proof to update the `diskEnd` ghost state of the logical log, as shown in Figure 9. This specification for `Append` provides modularity in that the `Append` proof does not need to know about the logical log and its `diskEnd`, and the logger proof does not need to worry about why `Append` is atomic. A key advance of Perennial's logically atomic crash specs lies in additionally capturing the crash behavior in this callback style, so as to enable a complete proof of crash safety across layers.

6.3 Concurrency within a block (OBJ)

GoJournal's OBJ layer allows the caller to issue reads and writes that are smaller than a full block. This finer granularity helps increase concurrency: for example, the NFS file server packs multiple inodes into a single disk block, and OBJ allows threads to concurrently read and write multiple inodes even if they share a disk block.

At commit time, OBJ's `Commit` may need to perform an "installation read" and read a full block, update the range that was modified by the caller as part of a journal operation, and write back the full block using the WAL layer. To ensure correctness of this read-modify-write operation, `Commit` uses a lock to serialize all commit operations. However, `Read` operations are lock-free: they can execute concurrently with one another and concurrently with `Commit`.

Lock-free reads pose a verification challenge because the disk block can be modified during the read. Consider the example shown in Figure 10, where a single disk block stores many inodes. Inode 1 initially contains the value A, while inode 4 contains B. Thread 1 is committing a write of B' to inode 4 in that block, while thread 2 concurrently reads inode 1 from the same block. To read inode 1, thread 2 will read the entire block, and then copy out the part of the block corresponding to inode 1. The block seen by thread 2 will differ depending on whether thread 1's write happens before or after the read, but inode 1 will contain A in either case.

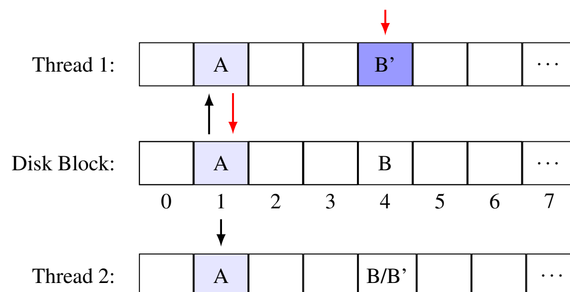


Figure 10: An example of a concurrent `Read` of inode 1 and `Commit` modifying inode 4 in the OBJ layer.

Formally reasoning about the `Read` operation requires the OBJ layer to connect the $a \mapsto_{op} o$ predicate about a disk object (such as an inode) to the disk block containing that object at the WAL layer. However, due to the race condition described above, the `Read` implementation might observe many possible values of the containing disk block. As a result, it is important for the OBJ invariant to relate the $a \mapsto_{op} o$ predicate not just to the latest value of the containing block, but to all recent contents of that block. Specifically, the invariant for $a \mapsto_{op} o$ requires that all recent writes to a's block (since `Read(a)` started) must agree on the part of the block storing o. As a result, regardless of what block happened to be read, the caller is guaranteed to see the correct object o.

7 Implementation

Perennial 2.0 is a re-write of the Perennial 1.0 framework [5], implemented on top of Goose [5, 6], Iris [17, 18], and Coq [28]. Figure 11 shows the lines of specifications and proof for Perennial. Perennial extends the Iris Proof Mode [20] to support convenient interactive proofs in Coq for crashes and Perennial's atomic crash specifications.

Perennial's program logic for crashes provides the formal foundations for framing away crash conditions and for atomic crash specifications. Lifting is implemented as part of the helper libraries. Ghost resources implement lock-free concurrent reasoning, including monotonic counters (to track log positions in Figure 6) and multi-versioned disks (to track logical disk contents at crash time for disk-object ownership).

Component	Lines of Coq
Helper libraries (maps, lifting, tactics)	5,760
Ghost state and resources	5,125
Program logic for crashes	9,375
Total	20,260

Figure 11: Lines of specs and proofs for Perennial.

	Lines of code (Go)	Lines of proof (Coq)	Ratio
CIRCULAR	109	1,905	17×
WAL-STS	555	10,125	23×
WAL	—	2,854	
OBJ	133	2,971	22×
JRNL-STS	121	1,261	24×
JRNL	—	1,640	
LOCKMAP	118	864	7×
Misc.	311	4,177	13×
GoJournal total	1,345	25,797	19×
GoNFS	3,911	*Not verified*	—
SimpleNFS	462	3,749	8×

Figure 12: Lines of code and proof for the components of GoJournal and for SimpleNFS. Ratio is the proof:code ratio, a rough measure of verification overhead.

Using GoJournal, we implemented GoNFS and its core verified subset, SimpleNFS. Both implementations can be mounted by the Linux NFS client, which translates file-system calls into NFS RPCs. GoNFS is sufficiently complete that it can run fsstress and fsx-linux tests through the Linux NFS client.

The breakdown of lines of code and proof by layer, as seen in Figure 12, shows a proof-to-code ratio of about 20× for the layers that involve tricky crash safety and concurrency reasoning. Notably the SimpleNFS proof is relatively short due to the GoJournal implementation and specification largely hiding crash reasoning. The WAL and JRNL layers are split into two parts for proof purposes; the layers labeled "STS" are specified with an atomic state-transition system while the next layer presents an easier-to-use ownership-based interface using separation logic. The write-ahead log's proof is largely in establishing its atomic transitions, while half of the top-level GoJournal proof is proving its separation logic specification as described in §5.

All of the proofs for Perennial, GoJournal, and SimpleNFS are checked by Coq, and we used `Print Assumptions` to verify that the proofs are complete. The code is publicly available.[4]

8 Evaluation

This section empirically answers several questions:

[4]GoJournal is available at `https://github.com/mit-pdos/go-journal` while GoNFS and SimpleNFS are at `https://github.com/mit-pdos/go-nfsd`.

- Is GoJournal sophisticated enough to support real storage systems and to achieve good performance? (§8.1)
- Is GoJournal's concurrency important for storage systems to achieve high performance? (§8.2)
- Are Perennial's verification techniques important for proving the correctness of GoJournal (§8.4) and for enabling application developers to prove their code on top of GoJournal (§8.3)?
- How much effort is required to prove the correctness of GoJournal and applications on top of GoJournal? (§8.5)
- Does verification help developers avoid bugs? (§8.6)

8.1 GoJournal is functional and performant

To evaluate whether GoJournal is sophisticated enough to support real storage systems and to achieve good performance, we measure the performance of GoNFS using three benchmarks: the LFS smallfile and largefile benchmarks, as well as a development workload, consisting of running `git clone` on the xv6 source-code repository [9] and compiling it with `make`. These benchmarks were also used by DFSCQ [7], a previous state-of-the-art verified file system. As a comparison point for GoNFS, we run the Linux kernel NFS server exporting an ext4 file system. The ext4 file system writes data through the journal (using the `data=journal` mount option), so that both systems provide the same crash-safety guarantees. The GoJournal implementation supports atomic but unstable writes, which match the semantics of unstable NFS `WRITE` operations. While all the internal layers of the proof support unstable writes, the separation logic specification presented in §5 does not, so we conducted the evaluation without using unstable writes in GoNFS.

We ran the benchmarks on Linux 5.12.3, using its NFS client to mount both GoNFS and the Linux NFS server. The experiments are run on two machines, a desktop with a relatively slow SSD and an EC2 machine with a fast NVMe disk. The desktop has an Intel Xeon E5-2640 20-core CPU at 2.4 GHz, 64 GB of RAM, and a 256 GB Samsung 850 PRO SSD, which we use to measure in-memory performance with no disk bottleneck as well as the impact of relatively slow storage. The EC2 instance is an i3.metal, which has 72 vCPUs, 512 GB of RAM, and a local 1.9 TB NVMe SSD, which we use to measure performance on fast storage with good random-access performance. To reduce variability we limit the experiment to a single socket, disable turbo boost, disable processor sleep states, and disable Spectre mitigations in the kernel.

We first evaluate GoNFS's performance with a single client issuing requests. Figure 13 shows the results on the Intel Xeon desktop with both file systems backed by RAM, to avoid any I/O overhead — GoNFS takes a simple Go interface for the disk, which we implemented with a large array, while ext4

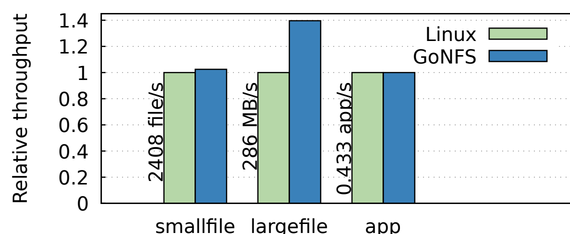


Figure 13: Performance of Linux NFS and GoJournal + GoNFS for `smallfile`, `largefile`, and `app` workload, on a RAMdisk. On an NVMe disk GoNFS achieves at least 90% of Linux's throughput.

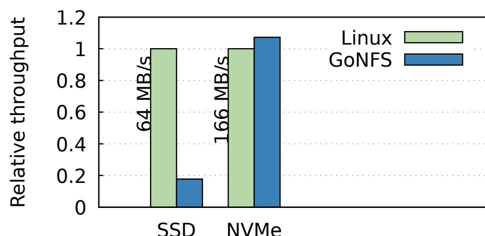


Figure 14: Performance of `largefile` depends on the storage medium. Linux takes advantage of unstable writes to write a large amount of data between barriers but GoNFS flushes to disk frequently.

uses a file in tmpfs.[5] GoNFS achieves at least the throughput of ext4 across the different workloads.

On both the NVMe and slower SSD, GoJournal's performance relative to ext4 is similar on the `smallfile` and `app` workloads (not plotted), again achieving at least 90% of the throughout of ext4. However, GoNFS performance on the `largefile` benchmark is sensitive to disk I/O characteristics, as shown in Figure 14. On the faster NVMe device, GoNFS's large file performance is comparable to ext4's, but on the slower SSD, it drops to under 20% of ext4's throughput. The reason is that the `largefile` benchmark produces a large number of parallel, unstable writes to the same file. GoNFS runs them sequentially due to a per-inode lock, and then journals sequentially because it ignores the unstable write flag. A disk barrier on the SSD takes about 2 milliseconds, so getting good disk throughput requires writing a large amount of data before issuing a barrier, and the 64 KB batch size is insufficient to get the maximum SSD write throughput. Re-running the experiment with unstable writes enabled in GoNFS raises its throughput to 90% of ext4's.

8.2 GoJournal concurrency improves performance

To test whether the concurrency of GoJournal is important for performance we measure the aggregate throughput of GoNFS with an increasing number of clients that run the `smallfile` benchmark. We run the experiment on a physical disk instead of an in-memory file system so that while a thread is waiting for the disk another thread can run. We compare the performance of GoNFS to that of Linux ext4, and to a single-threaded version of GoNFS that has no concurrency.

Figure 15 shows the results on an EC2 i3.metal instance with an NVMe SSD. Both GoNFS and Linux ext4 take advantage of concurrent requests to increase throughput. The single-threaded GoNFS does just barely improve performance, from parallelization among the clients and NFS server, but this amounts to less than 2× throughput with 20 clients than with one. Even with one client, GoNFS achieves 35% higher throughput than single-threaded GoNFS due to concurrency between the RPC thread, the logger thread, and the installer thread. GoNFS achieves higher throughput than Linux ext4, but it is hard to pin down the reason why, because there are many differences in the designs. One possibility is that Linux ext4 does not have concurrent logging and installation (but GoJournal does); another possibility is that ext4 waits for outstanding transactions to finish before flushing to disk (but GoJournal does not).

Figure 16 shows the scaling of GoNFS and Linux, this time on the Xeon desktop with a slower SSD. While GoNFS obtains comparable performance for 7 or fewer cores, Linux scales linearly beyond while GoNFS does not. The scaling in this case primarily comes from batching writes from concurrent clients, resulting in better disk write throughput. GoJournal is not as careful about this, sometimes committing a small amount of data rather than gathering many multi-writes and issuing them together. The NVMe experiment in Figure 15 uses storage with fast enough random-write access that CPU efficiency is more important than issuing large sequential writes; while a disk barrier takes 2 milliseconds on the SSD it takes only 30 *microseconds* on the NVMe disk.

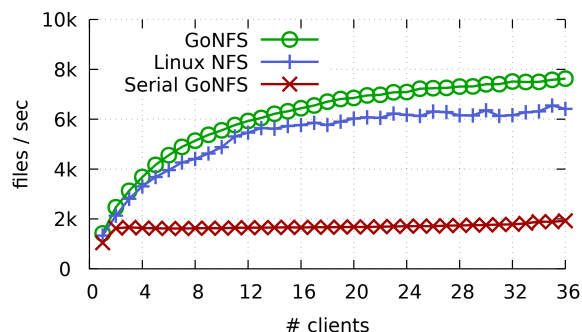


Figure 15: Combined throughput of multiple parallel `smallfile` microbenchmarks, each creating files in different directories, on an NVMe SSD.

8.3 Journaling atomicity simplifies proofs

Many storage systems use journaling because they simplify the implementation in terms of crash safety: the only point at which durable state is modified is when an operation commits. A goal of GoJournal is to carry this insight into proofs, so that a storage system using the journal can prove an operation is atomic using reasoning about durable storage only at the commit point.

[5]Running GoNFS on tmpfs performs slightly worse due to the around 1 microsecond syscall overhead of each disk operation, which ext4 does not incur since everything happens within the kernel.

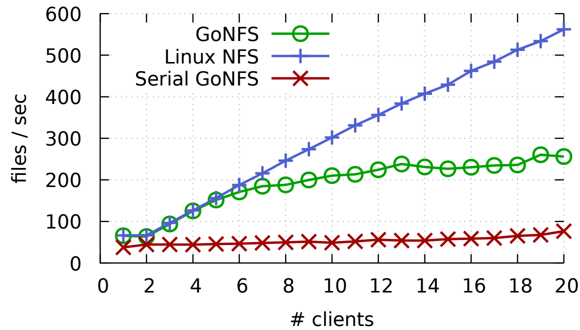


Figure 16: Combined throughput of multiple parallel `smallfile` microbenchmarks, each creating files in different directories, on a (slow) SSD.

One measure of how well GoJournal achieves this goal is the lines of code in SimpleNFS that require reasoning about durable state. SimpleNFS consists of 462 lines of verified code. Only 44 lines of code require proofs to explicitly consider durable state, using crash conditions. In Figure 3, for example, crash reasoning is only needed for lines 6–8 when acquiring and releasing with the crash-aware lock specification. All of the other code does not require reasoning about durable state; it suffices to prove simple crash-free specifications that have a pre- and post-condition, but no crash condition. This formal reasoning is enabled by two techniques from Perennial: lifting disk-object ownership and crash framing.

8.4 Perennial enables modular crash reasoning

Atomic crash specifications are crucial for enabling modular reasoning about crash safety. In GoJournal, atomic crash specs are used at many layer boundaries. Out of the layers shown in Figure 4, CIRCULAR, WAL, OBJ, and JRNL all provide atomic crash specifications, which are used by the layer above. One benefit of atomic crash specs is that they allowed us to develop these layers independently, using the specifications of lower layers before their implements were fully proven, as one would expect of any good API.

The modularity in Perennial largely follows the same structure as the code. Figure 12 shows that the WAL and JRNL proofs were split into an atomic transition specification about the code and a proof-only abstraction on top, but the bulk of the division was due to boundaries in the code that made the implementation manageable. Using separation logic it was easy to prove data structures (like the striped lockmap) and individual utility functions and use their abstract specifications elsewhere in the proof.

8.5 Proof effort

Figure 12 shows the lines of code and lines of proof for GoJournal and SimpleNFS. The hardest part of GoJournal lies in the WAL layer, which has significant lock-free concurrency, and requires careful reasoning about crashes and recovery. This is reflected in WAL's relatively high lines of code, lines of proof, and proof:code ratio. In contrast, SimpleNFS leverages GoJournal's atomicity, and ends up with a much smaller proof relative to its code size.

8.6 Verification prevents bugs

When developing GoJournal, we wrote unit tests to quickly find problems before starting verification, but they did not catch all bugs. While proving GoJournal, we found a subtle bug in absorption. When appending a new transaction in memory, GoJournal has an optimization called absorption where earlier writes to the same address are replaced with the new values. However, we discovered a race condition, where the logger thread could have been already flushing those earlier writes to disk, leading to unpredictable disk contents depending on the order of absorption vs logging. We fixed this issue by introducing the `nextDiskEnd` boundary, as shown in Figure 6: the logger thread only logs up to `nextDiskEnd`, and absorption is only allowed to modify values after `nextDiskEnd`.

9 Conclusion

GoJournal is the first concurrent crash-safe journaling system with a machine-checked proof, built on top of the Perennial 2.0 framework. GoJournal uses Perennial's techniques, including lifting and crash framing, to carry over the atomic benefits of journaling to its formal specification. This enables storage applications to use mostly crash-free reasoning in their proofs. For example, in the verified SimpleNFS server, only 44 lines of code, out of 462, required crash reasoning. GoJournal is sophisticated enough to implement a functional (but unverified) NFSv3 server, GoNFS, that achieves 90% of the performance of a Linux ext4 NFSv3 server on a development workload, far higher than any previous verified file systems, and GoJournal's concurrency enables GoNFS to scale with concurrent client requests. To simplify GoJournal's proofs, Perennial provides logically atomic crash specifications, which capture the crash properties of internal interfaces as single logical transitions, enabling modular proofs for GoJournal's internal layers.

Acknowledgments

We are grateful for feedback from many people that improved this paper, especially Alexandra Henzinger, Jonathan Behrens, Henry Corrigan-Gibbs, Jon Howell, the anonymous reviewers, and our shepherd, James Bornholt. This research was supported by NSF awards CNS-1563763 and CCF-1836712, and by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

References

[1] Dimitris Andreou. Striped (Guava: Google core libraries for Java 19.0). `https://guava.dev/releases/19.0/api/docs/com/google/common/util/concurrent/Striped.html`.

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

[3] 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.

[4] 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.

[5] 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.

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, January 2020.

[7] 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.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[9] 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`.

[10] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, pages 207–231, Uppsala, Sweden, July–August 2014.

[11] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 287–300, Rome, Italy, January 2013.

[12] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming*, 131:3–21, 2016.

[13] 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.

[14] 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.

[15] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, Austin, TX, January 2011.

[16] 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. USENIX Association.

[17] 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.

[18] 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.

[19] Eric Koskinen and Matthew Parkinson. The Push/Pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 186–195, Portland, OR, June 2015.

[20] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, January 2017.

[21] K. Rustan M. Leino, Richard L. Ford, and David R. Cok. Dafny reference manual. `https://github.com/dafny-lang/dafny/raw/master/docs/DafnyRef/out/DafnyRef.pdf`, December 2020.

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

[23] 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.

[24] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 169–188, Pohang, South Korea, November–December 2015.

[25] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. Persistent Owicki-Gries reasoning: A program logic for reasoning about persistent programs on Intel-x86. In *Proceedings of the 2020 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Chicago, IL, November 2020.

[26] 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.

[27] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22nd European Symposium on Programming (ESOP)*, pages 169–188, Rome, Italy, March 2013.

[28] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*, July 2020.

[29] 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.

A Artifact

A.1 Abstract

The artifact has the code for three tasks: calculating lines of code, running performance experiments, and checking the proofs. Since the artifact is packaged as a virtual machine, the generated graphs do not look exactly like the ones in the paper, but they do demonstrate respectable performance and that everything runs correctly.

A.2 Scope

The artifact will reproduce Figure 11 and Figure 12 (the lines of code tables). It has the code to run the performance evaluation, generating Figure 13 and Figure 16. To back up the claim that the proofs verify, we include the Coq source code and compilation instructions. For convenience the source code already includes the Goose-generated Perennial model of the source code, so the artifact also includes instructions on regenerating this output from scratch.

The paper includes a broader array of graphs than the artifact scripts generate, because it combines data from two benchmarking machines. The performance evaluation was expanded after artifact evaluation to include these more detailed results.

Note that the performance is highly sensitive to your machine and SSD's performance characteristics. We ran the paper's experiments with a litany of techniques to control variance, such as disabling turbo boost and using a single socket (as described in §8.1); until we did this, results were variable, and often hurt GoJournal more than Linux. The artifact is packaged as a VM which doesn't have the same careful setup, but we still believe it is useful because the VM setup documents the software requirements to run the benchmarks.

A.3 Contents

The artifact consists of a virtual machine with all the software required pre-installed and a checkout of the GoNFS source code, which has all the evaluation scripts.

A.4 Hosting

You can obtain the artifact's VM image via Zenodo DOI 10.5281/zenodo.4657115. The artifact instructions are at `https://github.com/mit-pdos/go-nfsd/tree/master/artifact`, as well as the Vagrantfile used to generate the VM image.

A.5 Requirements

The virtual machine uses VirtualBox. We configured it with 8GB of RAM (though 4GB is probably fine) and 4 cores; more cores might improve scalability numbers, although more clients help saturate the SSD even if you have fewer cores than clients. If the drive hosting the image is a hard drive, the "SSD" performance numbers will look quite bad.

Running the evaluation natively requires a variety of software that is documented by the VM provisioning scripts, which are in the `mit-pdos/go-nfsd` repo alongside the instructions.

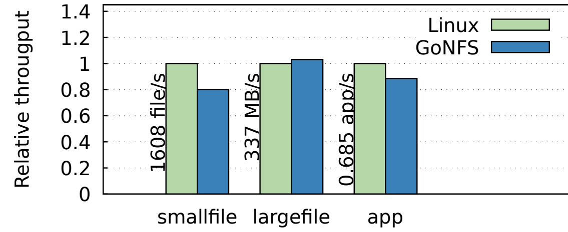
A.6 Results from artifact VM

On a MacBook Pro with a 2.4 GHz Intel i9-9980HK, we obtained the performance results in this section from running the artifact in a virtual machine. These experiments use the default VM configuration, with 8GB of RAM and 4 cores, on a host with 8 cores.

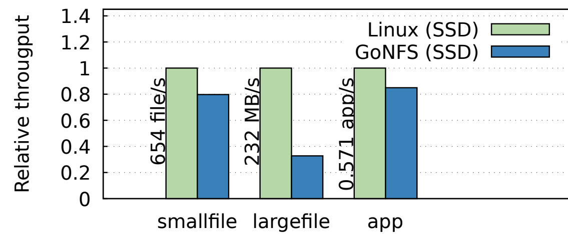
Figure 17 shows the results of running the microbenchmarks on this hardware configuration. Figure 17a is analogous to Figure 13. Figure 17b includes the `largefile` results shown in Figure 14. Between these two figures we see more variability on `smallfile` than when run on physical hardware. The `largefile` results are as expected, since the SSD in this machine has performance somewhere in between the SSD in the desktop machine and the NVMe drive from an i3.metal instance.

Figure 17c shows the results of running the largefile benchmark across a variety of software configurations, all on an SSD; these were not directly shown in the paper. From these results we concluded that GoNFS can get good performance, if using unstable writes. The "Linux (sync)" configuration uses ext4 in `data=journal` mode but additionally mounts the NFS share with the `sync` flag, which changes the benchmark to a completely sequential and synchronous one. In this configuration Linux's optimizations do not kick in and it obtains the same performance as GoNFS using stable writes.

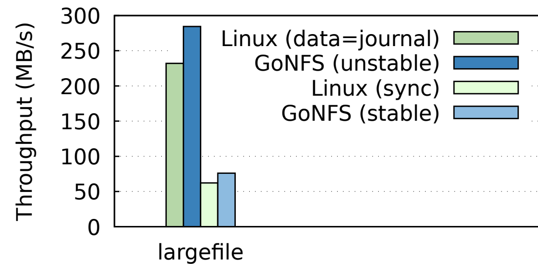
Finally, Figure 18 shows scalability of the `smallfile` benchmark, analogous to Figure 16. Even though this disk gets much better throughput and has a barrier latency of only 0.4 ms (in the virtual machine) rather than 2 ms, the experiment has the same trend as on the slower SSD.



(a) `bench.pdf` (RAM)



(b) `bench-ssd.pdf` (SSD)



(c) `largefile.pdf` (SSD)

Figure 17: Microbenchmarks and app benchmark run inside a VM.

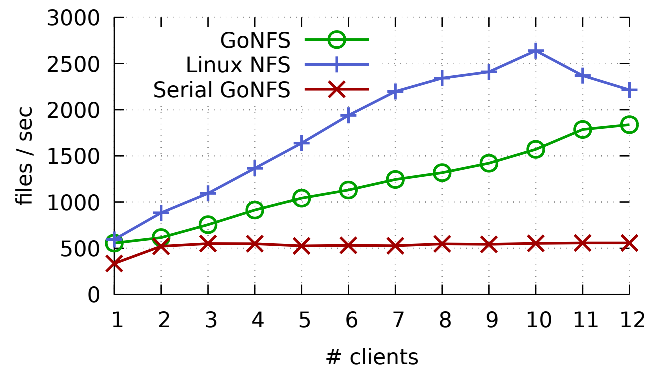


Figure 18: `scale.pdf`, showing scalability of the `smallfile` benchmark. The VM had 4 CPU cores for this experiment.