Verifying concurrent, crash-safe systems with Perennial

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Abstract

This paper introduces Perennial, a framework for verifying concurrent, crash-safe systems. Perennial extends the Iris concurrency framework with three techniques to enable crash-safety reasoning: recovery leases, recovery helping, and versioned memory. To ease development and deployment of applications, Perennial provides Goose, a subset of Go and a translator from that subset to a model in Perennial with support for reasoning about Go threads, data structures, and file-system primitives. We implemented and verified a crash-safe, concurrent mail server using Perennial and Goose that achieves speedup on multiple cores. Both Perennial and Iris use the Coq proof assistant, and the mail server and the framework's proofs are machine checked.

CCS Concepts • **Software and its engineering → Software verification**; *Concurrency control*; *Software fault tolerance*.

Keywords Concurrency, Separation Logic, Crash Safety

1 Introduction

Making concurrent systems crash-safe is challenging because programmers must consider many interleavings of threads in addition to the possibility of a crash at any time. Testing interleavings and crash points is difficult, but formal verification can prove that the system always follows its specification, regardless of how threads interleave and even if the system crashes.

Several existing verified storage systems address many aspects of crash safety [5, 7, 10, 34], but they support only sequential execution. There has also been great progress in verifying concurrent systems [4, 13, 14, 20, 23, 41], but none support crash safety reasoning. This paper develops techniques for reasoning about crash safety in the presence of concurrency and applies them to a verification system called Iris [24].

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359632

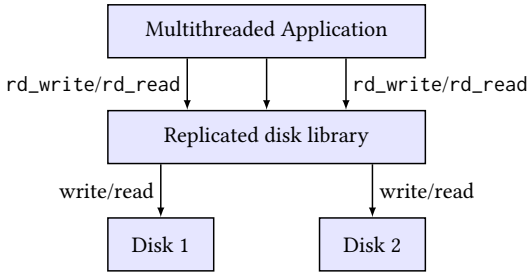

Figure 1. A concurrent, replicated disk library that tolerates a single disk failure using two physical disks. The library provides linearizable reads and writes, and transparently recovers from crashes.

To understand why reasoning about the combination of crash safety and concurrency is challenging, consider the following example: a concurrent disk replication library (Figure 1) that sends writes to two physical disks and handles read failures on the first disk by falling back to the second. The informal specification for the library is simple: the two disks should behave as a single disk. That is, reading a block should return the last value written to that block, and concurrent reads/writes should be linearizable [19].

One way to implement this specification is with a lock per block, which is held during writes and reads. This guarantees that concurrent writes and reads of the same disk block are linearizable. Intuitively, such an implementation is correct because a write is durably stored on both disks before the lock is released.

The lock provides linearizability, but a crash that happens in the middle of a write leaves the disks out of sync. Therefore, the implementation must run a recovery procedure on reboot. Because we want writes to be durable when they finish, recovery must not revert or corrupt completed writes. For example, it would be wrong for recovery to make the disks in sync by zeroing them both. A correct recovery procedure copies values from the first disk to the second. This is safe because it *logically completes* write operations that crashed during execution and only overwrites old data.

To *prove* that this justification is correct and that the developer has considered all interleavings and crash points correctly, we need to capture this reasoning using precise rules that lend themselves to concise, machine-checked proofs. Formalizing this argument is challenging and beyond the scope of previous concurrency verification tools.

This paper introduces Perennial, which extends the Iris concurrency verification framework [24] to incorporate crash safety reasoning. In order to explain Perennial's contributions, we give some brief background on Iris itself. Proofs in Iris are based on tracking *capabilities* owned by individual threads. One proves that each thread manipulates local capabilities correctly and then composes these proofs together to reason about interacting threads. The notion of a capability and how capabilities compose is highly flexible in Iris, which allows encoding complex protocols and fine-grained sharing.

In this work we extend Iris with new capabilities for reasoning precisely about the interaction between concurrent threads, crashes, and executing recovery, with three techniques:

- *Versioning*: In a crash-safe system some data is persistent while the rest is only in memory. We express this property by attaching an execution version number to any capabilities for volatile resources, like in-memory pointers. Only capabilities for the current version are considered valid.
- *Leases*: Threads often need exclusive access to durable resources while running, but this exclusive access should be logically transferred to recovery in case of a crash. We implement this behavior by splitting capabilities for durable resources into two pieces: a "master" copy, which persists across crashes, and a temporary "lease" which is restricted to a particular version number.
- *Helping*: In some cases a thread crashes in the middle of an operation, which recovery subsequently completes, a pattern we call "helping" based on a similar concept in lock-free programming [18]. We formalize that this pattern is still linearizable by introducing a special capability representing the on-going action of the crashed thread, which gives recovery the right to complete the operation on the thread's behalf.

Although we developed Perennial and the above techniques as an extension to Iris, we believe these ideas can be adapted to other concurrency verification frameworks that use capability reasoning.

To verify running systems, we write them in Goose, a subset of the Go language. We implemented a translator to convert this subset of Go to Coq, as well as a formal semantics in Perennial. Developers can then run the Go code using the standard Go toolchain, while writing proofs in Perennial. Goose includes support for threads, pointers, slices, and a subset of the POSIX file-system API.

We used Perennial and Goose to implement and verify Mailboat, a mailserver that extends CMAIL [4] with crash recovery. The proof shows that after recovery all delivered messages are durably stored and that concurrent readers only observe complete messages. To further evaluate whether Perennial's reasoning principles suffice for a variety of crash safety patterns, we verified microbenchmark examples involving write-ahead logging and shadow copies.

Our contributions are the following:

- Perennial, a system for machine-checked proofs of concurrent crash-safe systems that uses *versioning*, *leases*, and *helping* to support crash-safety proofs on top of Iris's support for concurrency reasoning.
- Goose, a subset of Go with a translator to Coq and a semantics in Perennial for reasoning about Goose code.
- Mailboat, a mail server written in Goose with a proof of atomicity and durability that demonstrates using Perennial end-to-end, as well as smaller verified examples covering more reasoning patterns.

Our prototypes of Perennial and Goose have some limitations. Perennial does not currently support composing layers of abstraction; we believe that extending multilayered frameworks like CertiKOS [13] and Argosy [5] to the concurrent crash setting is feasible. Perennial proofs only cover safety properties and not liveness, as is typical in concurrent verification. Goose does not include the entire Go language and lacks support for interfaces or first-class functions. Goose's file-system model does not support deferred durability, but we believe that this is not a fundamental limitation. While Perennial can in principle reason about lower-level storage systems like in-kernel file systems or flash translation layers, this would require a replacement for Goose in a language without a runtime.

2 Related Work

Verified crash safety. Recently several verification frameworks have tackled the problem of crash safety of sequential systems, including verified file systems [5, 7, 10, 34]. These systems address many issues, including handling crashes during recovery and giving an abstract specification that covers non-crashing and crashing execution separately. None of these systems support concurrency, and as the replicated disk example of §1 illustrates, interactions between concurrency and crashes require new reasoning techniques.

Fault-Tolerant Concurrent Separation Logic (FTCSL) [31] does support concurrency and crash safety. FTCSL is built on top of a concurrency verification framework called Views [9], similar to how Perennial is built on Iris. However, in contrast to Perennial, FTCSL does not support fine-grained locking or lock-free reasoning. Additionally, FTCSL does not provide a way to prove linearizability. We developed the leasing and helping techniques mentioned in §1 specifically to address these challenges in Perennial. Finally, FTCSL and Views have only been used for pen and paper proofs, while Perennial supports machine-checked proofs about runnable code.

SMT-based verification. An approach to verification that has been successfully applied to several systems is SMT-based verification [30, 34, 35] and hybrids of SMT solving and another verification tool [16, 17, 23, 32]. In these approaches, the verification tool translates a program and its specification into verification conditions — if an SMT solver

(for example, Z3) can prove these verification conditions, then the program meets its specification. In interactive theorem proving, including this work, the verification conditions are proven by a developer, but the framework proves that the specification is correctly encoded into verification conditions. It would be an interesting direction for future work to combine aspects from SMT-based verification and interactive theorem proving, similar to the approach of hybrid systems like Nickel [35] and F* [37], but extended to support crash safety and concurrency.

Concurrent verification. There are many approaches to verifying concurrent software [4, 9, 14, 23, 24, 33, 41]. None of these approaches directly supports crash-safety reasoning. Incorporating crash safety into an existing verification framework is not obvious because crash safety requires reasoning about a different mode of execution, where crashes can occur at any time and recovery should run after any crash. Additionally, crash safety requires a different specification that distinguishes what is allowed if the system crashes versus if it does not. FTCSL's design highlights this difficulty: FTCSL [31] re-uses the Views framework [9], but still require a new logic, an encoding into Views, and a proof that the resulting theorems have the right meaning in the context of recovery execution; any mistake in this on-paper reasoning could render any proofs built on top of the framework invalid. In contrast, Perennial introduces techniques to encode crash safety into Iris and then has a machine-checked proof that this encoding is correct.

Distributed Systems. Distributed systems face some of the challenges of concurrency and crash safety. However, of the existing verified distributed systems [17, 26, 40], only Verdi [40] covers reasoning about replication to maintain the consistency of nodes that crash and rejoin a system. Perennial can be used to verify the kind of crash-safe, concurrent node-storage system that Verdi assumes.

Connecting verification to runnable code. There are two broad approaches to connecting a running system and its proof. Extraction-based approaches take a model of the system in a form the verification system understands, and transform it into runnable code. Import-based approaches take runnable code and then convert them into proof obligations in the verification system. Both extracting and importing have been explored extensively in prior work [1, 3, 6, 8, 15, 21, 22, 27, 29, 36]. The Goose translator takes the import approach, starting from a subset of Go.

3 Overview

The components of Perennial are organized as shown in Figure 2. The developer fills in the green boxes: the code, the spec, and the proof. The code is implemented Goose (§6), a subset of Go equipped with a translator to a Perennial model. Goose includes enough of Go to write useful code, yet is simple enough to be represented in a concise model. For example, the Goose model supports pointers and slices, but not interfaces, which require function pointers to accurately model. To implement systems that store data, and to reason about crash safety, Goose includes a file-system library. Finally, the code can be compiled using the standard Go compiler, and linked with unverified code written in ordinary Go.

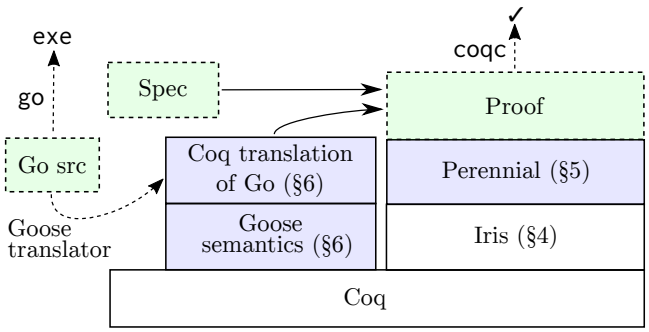

Figure 2. Overview of Perennial. Blue boxes are provided by Perennial, while green ones with dashed borders are written by the developer. White boxes are inherited by Perennial.

The proof is written inside Coq, reasoning about possible executions of the Go code using Perennial's model of Go. The proof is machine-checked and, if correct, implies that all possible code executions are allowed by the specification.

3.1 Defining correctness

Perennial defines correctness of a system using refinement between a system's code and its specification. Both the code and the spec are *transition systems*: that is, a state that can evolve over time through a sequence of well-defined atomic steps. The spec transitions are calls to the system's top-level operations, whereas the code transitions at a finer granularity for every primitive operation in the implementation. Both transition systems include crash transitions and allow interleaving operations from multiple threads. We introduce the idea of *concurrent recovery refinement* for Perennial's particular form of refinement.

Concurrent recovery refinement requires that every sequence of code transitions must correspond to some interleaving of spec transitions with the same external I/O (i.e., invocations and return values of top-level procedures). This allows a user of the system to abstract away from the code and reason purely about the spec, since the spec covers all possible code executions. Concurrent recovery refinement also requires that whenever the implementation crashes (followed by recovery and perhaps some number of crashes during recovery), the whole sequence should simulate a single atomic crash step in the specification. A particular challenge in reasoning about crashes in Perennial is that the implementation can crash in the middle of many concurrent operations,

rather than just a single operation that was invoked before the crash.

To write concise specifications, Perennial provides a domain-specific language embedded in Coq for writing transition systems. For example, consider the disk replication system described in the introduction. We show a specification for the replicated disk's operations, `rd_read` and `rd_write`, in Figure 3. The spec says that the replicated disk's state is a single logical disk, represented as a mapping from addresses to disk blocks. The `rd_read(a)` specification looks up the value of address `a` by reading from the state with the `gets` primitive. If the address is out of bounds, the behavior is undefined. The `rd_write(a, v)` specification first checks that the address is in-bounds, then updates that address in the state with the `modify` primitive.

```
Definition State := Map uint64 block.

Definition rd_read (a:uint64)
  : transition State block :=
  mv <- gets (fun σ => Map.lookup a σ);
  match mv with
  | Some v => ret v
  | None => undefined
  end.

Definition rd_write (a:uint64) (v:block)
  : transition State unit :=
  mv <- gets (fun σ => Map.lookup a σ);
  match mv with
  | Some _ => modify (fun σ => Map.insert a v σ)
  | None => undefined
  end.

Definition crash : transition State unit :=
  ret tt.
```

Figure 3. Specification for the replicated-disk operations. Callers observe the transitions in these definitions atomically even if the system crashes, and the crash transition specifies that no data is lost after recovery.

Perennial's concurrent recovery refinement for the replicated disk says these two operations are linearizable; that is, they behave as if the transitions described in the spec occur atomically when called from multiple threads.

Figure 4 shows a simple pseudo-code implementation of `rd_read` and `rd_write`, which uses per-address locks to ensure linearizability. To handle crashes, which can interrupt even critical sections, the implementation runs a recovery procedure after a crash that repairs the state of the replicated disks before accepting new `rd_read` and `rd_write` requests. Suppose the system were to crash between the two calls to `disk_write(a, v)` in `rd_write`, which would leave `Disk1` with the value `v` but `Disk2` with the old value. Without a recovery procedure, calling `rd_read(a)` after this crash would return `v` from `Disk1`, but if `Disk1` were to fail, subsequent calls to `rd_read(a)` would fail over to `Disk2` and then return the old value from before `v` was written, which is disallowed by the specification. A recovery procedure eliminates this inconsistency; one possible implementation is shown in Figure 5, which copies the blocks from `Disk1` to `Disk2` to bring the disks back into a consistent state.

```
1  func rd_read(a) []byte {
2    acquire_lock(a)
3    v, ok := disk_read(Disk1, a)
4    if !ok {
5      v, _ = disk_read(Disk2, a)
6    }
7    unlock(a)
8    return v
9  }
10
11 func rd_write(a uint64, v []byte) {
12   acquire_lock(a)
13   disk_write(Disk1, a, v)
14   disk_write(Disk2, a, v)
15   unlock(a)
16 }
```

Figure 4. Go pseudo-code for replicated-disk read and write. These implement the specifications in Figure 3 atomically, using a lock to protect each disk address.

```
1  func rd_recover() {
2    for a := 0; a < DiskSize; a++ {
3      v, ok := disk_read(Disk1, a)
4      if ok {
5        disk_write(Disk2, a, v)
6      }
7    }
8  }
```

Figure 5. Go pseudo-code for replicated-disk recovery. Recovery guarantees that writes are atomic and persistent even after crashes, as specified by `crash` in Figure 3.

3.2 Proving correctness

To prove refinement for all possible executions, the developer uses a standard technique called forward simulation [28]. The first step in using forward simulation is an *abstraction relation* that connects the code and spec states and defines which states are reachable to begin with. Forward simulation requires the developer to show that, starting from any pair of code and spec states connected by the abstraction relation, any valid code-level transition results in a new code-level state that is connected to the same spec-level state, or another spec-level state that is the result of one or more spec-level transitions. Any output from the code (including

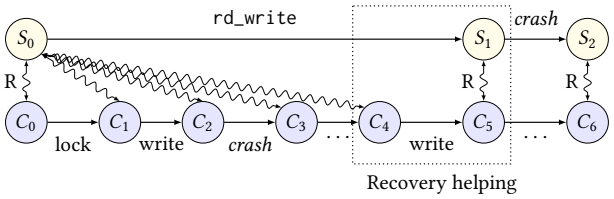

Figure 6. Refinement diagram for one execution with a crash in the middle of `rd_write`. Yellow states are spec states and blue ones are code states. Right arrows in the top row are spec transitions while those in the bottom row are code transitions.

return values) should be allowed by the spec transition as well. As mentioned above, forward simulation for concurrent recovery refinement also requires relating a crash followed by recovery to a crash transition at the abstract level.

As an example, consider Figure 6, which illustrates refinement using R as the abstraction relation for one possible execution of `rd_write`, with no concurrency but one crash followed by running the recovery procedure `rd_recover`. In this example, the system crashes after writing to `Disk1` but before writing to `Disk2`. The write has not yet logically completed, so in this example, the state in which `rd_write(a, v)` crashed still corresponds to the original spec state; no spec transitions have happened yet. Once the recovery code copies the new value from `Disk1` to `Disk2`, however, the spec takes a transition and appears to have executed `rd_write(a, v)`. Finally, after recovery finishes, the spec itself appears to execute a *crash* transition, to reflect the fact that even at the spec level, the executing code crashed and restarted.

Refinement, especially between a concurrent program and its atomic specification, is difficult to prove by direct forward simulation for two reasons. First, even correctly stating the abstraction relation is difficult, since it must capture all intermediate states from partially executed operations in different threads. Second, threads interact in complicated ways, and the forward simulation proof must consider any transition, including all interleavings of concurrent threads.

To make refinement proofs manageable we add some structure. The structure we use is to prove a per-operation Hoare triple that *summarizes* each operation, separates the state that each operation touches from other threads as much as possible, and otherwise prescribes a protocol for concurrent threads to follow. The summaries ensure that each thread maintains a forward simulation. The Perennial framework then provides a (difficult) proof showing that for any operations and summaries, as long as the summaries agree with each other on a common protocol and abstraction relation, the whole system satisfies refinement.

4 Background on Iris

Perennial is an extension to Iris, which is a variant of concurrent separation logic to reason about program correctness and an implementation of this logic in Coq. There are two aspects to separation logic: the assertion language, which represents *capabilities* that threads logically own, and *Hoare triples*, which describe what a piece of code does in terms of capabilities. An example of a capability is the basic "a points to v", written $a \mapsto v$. This points-to capability gives permission to read and write the memory address a and also asserts that the value in memory at a is v. These capabilities are all logical and expressed within the proof, with no runtime enforcement; if the code does not follow the permissions, the proof would not go through.

Hoare triples have the form $\{P\}\ e\ \{Q\}$, which we can interpret as "the procedure e when run with capabilities P (the precondition) returns capabilities Q (the postcondition)". For example, the $a \mapsto v$ capability is used in Hoare triples such as:

$$\{a \mapsto v\}\ \texttt{write(a, v')}\ \{a \mapsto v'\}$$

This triple consumes the capability $a \mapsto v$ in the pre-condition, and gives back the updated $a \mapsto v'$, reflecting the completed write to a.

Separating conjunction. Separation logic introduces the *separating conjunction*, $P * Q$, which represents ownership of capabilities in P and Q, which are required to be *disjoint*, meaning that the capabilities in P are compatible with the capabilities in Q. For example, the assertion $a \mapsto v * a \mapsto v$ never holds, because the left copy of the capability could be used to carry out a write, thereby invalidating the value in the right copy.

The separating conjunction is useful because it lets us verify each thread of a concurrent system in isolation, and then compose these proofs together to derive a theorem about the whole system. The way this works is that a thread with access to the capability $P * Q$ can fork a child thread and pass it the capability Q while retaining P. The proof for the child can simply assume access to Q without any reference to what the parent thread concurrently does with P, which is why these proofs can be carried out separately. This process of splitting resources and giving them to forked threads is illustrated in Figure 7.

Preconditions in separation logic only need to mention the capabilities the code uses to establish its postcondition, not the global state of the system. More formally, $\{P\}\ e\ \{Q\}$ implies $\{P * R\}\ e\ \{Q * R\}$ for any "extras" R. Intuitively this so-called *frame rule* is true because the capabilities in R are disjoint from the resources e uses and modifies.

Invariants. As we have described, it would seem threads must operate on disjoint data for capabilities to partition according to $*$. However, in many cases threads need to share or transfer capabilities. For example, consider a scenario

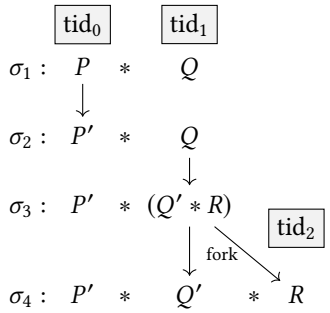

Figure 7. Illustration of the relationship between capabilities owned by all concurrent threads. At each step, the state (the σ_i) satisfies the separating conjunction of all threads' assertions. Because the threads' capabilities are all compatible, it suffices to prove separate Hoare triples about them.

where two threads should both be able to read from address a. As mentioned above $a \mapsto v * a \mapsto v$ is not provable, so it is not possible for both threads to simultaneously own this capability. Instead, in order to share this capability, Iris has a mechanism called *invariants*. Invariants are assertions that are required to hold at every step of execution once established. Iris defines a capability $\boxed{P}$ indicating that the capability P is an invariant. Proofs can use $\boxed{P}$ by temporarily *opening* the invariant, using the capabilities inside, and then closing the invariant by showing that P still holds. To ensure that the invariant is really preserved at all times, opened invariants must be closed after an atomic step. The upshot of preserving the invariant is that the capability $\boxed{P}$ can be shared among multiple threads — all threads in a system agree on the invariant. Finally, when invariants are *allocated*, the creating thread must provide the underlying capability. We show an example of an invariant being created, shared, and used in Figure 8.

Extensible capabilities. Classic separation logic has just the basic capability $a \mapsto v$ for describing memory contents. However, modern separation logics, like Iris, provide ways for expert users to define new capabilities for different kinds of state. For the replicated disk system, this mechanism can be used to define capabilities $d_i[a] \mapsto v$ to mean that disk i has value v at address a *if it has not failed*. When defined correctly, this capability would have rules similar to the `write` rule above for writing to the disks. We will not describe how this extensibility works, because it is not necessary to understand or use Perennial.

Reasoning about locks uses another user-defined capability. Each lock is associated with a *lock invariant*, an Iris capability logically protected by the lock. The lock guarantees that at most one thread (the lock owner) has access to the capabilities in the invariant at any given time. Lock invariants are expressed with a capability is_lock(ℓ, I), which

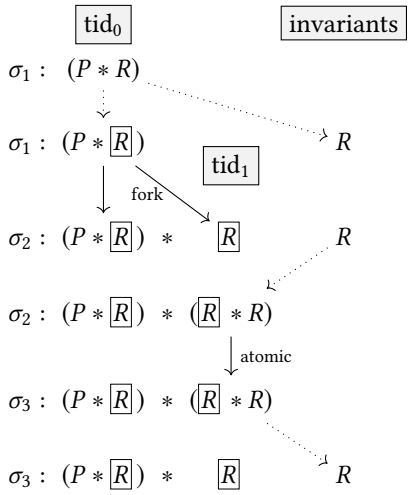

Figure 8. Diagram showing how resources are transferred between threads through invariants. Dotted lines indicate steps in the proof, while solid lines are steps of execution. Thread 0 initializes an invariant containing R. It then forks a child and passes it knowledge of the invariant. The child thread opens the invariant to obtain R, and then returns it after taking a single atomic step.

asserts that ℓ is the in-memory location of a lock with invariant I. Acquiring the lock gives the owner access to all the capabilities in I. To release the lock the lock owner must return the resources I for the next owner. The capability is_lock(ℓ, I) can be freely shared by multiple threads since it merely gives access to the lock—the protected resources require first acquiring the lock.

Lock invariants are quite similar to Iris invariants with the important difference that a thread can violate the lock invariant for as many steps as they have the lock, unlike invariants which must be returned after an atomic step.

Refinement. Iris can also support refinement reasoning using Hoare triples, following the approach in CaReSL [39] (which has previously been implemented in Iris [25]). The idea is to create new capabilities called *specification resources* that represent a forward simulation proof up to the current moment of execution, and then prove that the implementation advances this simulation proof by one step. A bit more concretely, we introduce a capability source(σ) that says the abstract state is σ and $j \Rightarrow\!\!\!\!\models op$ that says thread j in the specification transition system is about to run op. These resources only have meaning in the proof, but their encoding in Iris means they can only be manipulated by simulating a step and producing the correct return value. Specifically, when $step(op, \sigma, \sigma', v)$ is allowed by the specification (where v is some return value of the appropriate type for op) then source(σ) $* j \Rightarrow\!\!\!\!\models op$ can be replaced with source(σ') $* j \Rightarrow\!\!\!\!\models \mathrm{ret}\, v$. Finally, the abstraction relation is

encoded in Iris as an invariant that talks about source(σ) and relates it to the code state.

As a result of these rules, we can prove the following theorem:

Theorem 1 (concurrent forward simulation)**.** If all of the operations in an implementation satisfy

$$\{j \Rightarrow op * \boxed{AbsR}\}$$
$$\texttt{op\_impl()}$$
$$\{v.\ j \Rightarrow \mathrm{ret}\, v\}$$

then the implementation is a refinement of its specification with abstraction relation *AbsR*; that is, any sequence of calls to the operation implementation, starting from states that are initially related by *AbsR*, will returns values that match some interleaving of abstract operations.

In summary, Iris has a flexible notion of capabilities and disjointness that lets us reason about each thread independently. A core feature of Iris is invariants, which allow reasoning about threads sharing exclusive capabilities. Finally, capabilities in Iris are extensible, which we take advantage of in Perennial with extensions for reasoning about crashes and refinement with recovery (§5).

5 Proving concurrent recovery refinement

Perennial extends Iris with techniques for reasoning about crashes and provides a theorem that connects Hoare triples in Iris to concurrent recovery refinement. We summarize all of these techniques in Table 1. We first describe how to reason about crashes and recovery by connecting a Hoare triple for a procedure with a Hoare triple for recovery. This connection is based on a crash invariant (§5.1) and uses two extensions to Iris's capabilities, *versioned state* (§5.2) and *recovery leases* (§5.3). Next, we describe how Perennial supports concurrent recovery refinement with *recovery helping* (§5.4) and Theorem 2 (§5.5). All of the techniques we describe here are formalized and proven correct in Coq.

5.1 Crash Invariants

The first idea is to adapt the notion of *crash invariants* from Crash Hoare Logic (CHL) [7] to this concurrent setting. As the name suggests, a crash invariant in CHL is an invariant that must be shown to hold at every step of execution. Whenever a crash occurs, a special recovery procedure will run after the computer reboots. To ensure that the recovery procedure is safe to run, the crash invariant must imply the precondition of the recovery procedure.

Perennial uses Iris invariants to implement support for crash invariants. When a crash occurs, the recovery procedure starts with the capabilities from the distinguished crash invariant $\boxed{C}$. Any other capabilities owned by the executing threads are lost. This process is illustrated in Figure 9. Since the recovery procedure could immediately crash again, it re-allocates the crash invariant and maintains it throughout recovery.

Technique	Rules for using it
crash invariant (§5.1)	distinguished invariant $\boxed{C}$ which recovery starts with access to
versioned memory (§5.2)	$\{p \mapsto_n v\}$ `read`, `write` $\{\ldots\}_n$ (Hoare triples are at a version number and only allow capabilities at the current version)
recovery leases (§5.3)	$\{d[a] \mapsto_n v * \mathrm{lease}_n(d[a], v)\}$ `read`, `write` $\{\ldots\}_n$ (both capabilities are required to use master copy) $d[a] \mapsto_n v \implies d[a] \mapsto_{n+1} v * \mathrm{lease}_{n+1}(d[a], v)$ (can synthesize new lease after a crash using master copy)
refinement (§4)	when step(op, σ, σ', v), source$(\sigma) * j \Rightarrow op \implies$ source$(\sigma') * j \Rightarrow \mathrm{ret}\, v$ (simulate an abstract operation step)
crash refinement (§5.5)	when crash(σ, σ'), source$(\sigma) * \Rightarrow$ Crashing $\implies$ source$(\sigma') * \Rightarrow$ Done (simulate an abstract crash step)
recovery helping (§5.4)	operation stores $j \Rightarrow op$ in crash invariant, recovery simulates it

Table 1. Summary of techniques in Perennial.

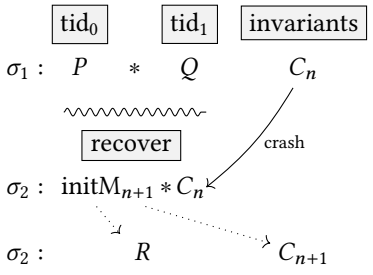

Figure 9. Illustration of how resources are passed from a crash invariant to the recovery thread. Recovery obtains the crash invariant C_n from the crashed execution, along with capabilities for the fresh memory at the new version number. It uses these capabilities to establish fresh leases for durable state, and then re-establishes the crash invariant for the new version number.

5.2 Versioned state

Crash invariants allow Perennial to handle transferring capabilities across a crash. However, Perennial also needs to account for the fact that crashes invalidate capabilities for volatile memory; for example, $a \mapsto v$ no longer holds after a crash since all memory contents are lost.

Iris has no built-in notion of invalidating capabilities. We use version numbers to encode the same idea without modifying the underlying framework. Perennial versions all in-memory capabilities with a generation number, for which we use the variable n. For example, we write $p \mapsto_n v$ for the basic points-to capability that says p points to a value v and gives ownership over that address, now with version number n. Every Hoare triple is for a particular version number, and any capabilities for old versions are invalid while proving a triple. The following Hoare triple describes the basic pointer store operation:

$$\{p \mapsto_n v_0\}\ \mathsf{p}\ \mathsf{:=}\ \mathsf{v}\ \{p \mapsto_n v\}_n$$

Note that the entire Hoare triple now has a version subscript, and if the capability's version does not match this triple does not apply.

When we connect Hoare triples across a crash, recovery begins with all capabilities in the crash invariant. However, we invalidate any in-memory capabilities by connecting a Hoare triple for a procedure at version n to a recovery Hoare triple *at version* $n+1$. Thus recovery must transform the resources from the crash invariant C_n into a new crash invariant $\boxed{C_{n+1}}$ at the new version number.

5.3 Recovery leases

In contrast to volatile state like memory, capabilities for durable state are still meaningful after a crash. The crash invariant gives us a way to prove that threads preserve some invariant about durable state throughout execution, so that recovery can rely on that invariant. On the other hand, locks protect some capabilities in an invariant from concurrent modification by other threads. However, these two types of invariants are incompatible: the same capability cannot appear in both.

For a concrete example of this issue, consider the replicated disk example. Recovery needs access to disk blocks to copy between the disks in case of a crash, so the crash invariant must store address capabilities. In addition, a lock protects each address so that a writer can modify the blocks in both disks without interference from another thread. The natural way to encode how the lock provides isolation would be to store the block capabilities in an Iris lock invariant, as described in §4, to enforce that a thread can only modify the blocks if it holds the lock. But we cannot store the $d_1[a] \mapsto v$ capability in both the lock invariant *and* the crash invariant, because capabilities cannot be duplicated.

We solve this problem using *recovery leases*. We split every durable capability $d[a] \mapsto v$ into a new lease capability $\mathrm{lease}_n(d[a], v)$ and a master copy $d[a] \mapsto_n v$. The lease represents the permission to modify the state for the current version number only, while the master copy records the value of the state so that recovery can use it after a crash. The lease has three important features. First, both the master copy and lease are required to update the value, via the following rule:

$$\{d[a] \mapsto_n v_0 * \mathrm{lease}_n(d[a], v_0)\}$$
$$\quad \texttt{write(a, v)}$$
$$\{d[a] \mapsto_n v * \mathrm{lease}_n(d[a], v)\}_n$$

Second, only one thread can hold the lease at any time; and finally, both capabilities are tied to the current version number and are invalidated on crash. These three features encode all of the properties needed to accurately model exclusive access to durable resources.

To give recovery access to the resource, the master copy $d[a] \mapsto_n v$ is stored in the crash invariant. To give running threads exclusive access, we use locks to protect the logical capability $\mathrm{lease}_n(d[a], v)$ and borrow $d[a] \mapsto_n v$ from the crash invariant to write to disk whenever necessary. Finally, recovery can take an old master copy $d[a] \mapsto_n v$ and synthesize a new lease/master pair for the new version number, $d[a] \mapsto_{n+1} v * \mathrm{lease}_{n+1}(d[a], v)$, keeping the master copy in a new crash invariant and handing out the lease to application code after recovery finishes. As a notational convenience we write $d[a] \mapsto v$ for the master copy at the current version, leaving implicit that this capability has been leased out. The code must track the difference to prevent duplicate leases, but the difference is not important to understand Perennial.

In the replicated disk example, we use a lock for each address a to protect recovery leases for $d_1[a] \mapsto_n$ and $d_2[a] \mapsto_n$, as well as ensure that their values agree when the lock is free. While the lock is held, the system might crash, in which case recovery can use the master capabilities to copy from the first to the second disk.

5.4 Recovery helping

In the replicated disk example, after a crash, the recovery code synchronizes the contents of disk 1 onto disk 2. If the disks differ in any location, this needs to be justified with a spec-level transition. Informally, the justification is that if the disks differed at a before a crash, there must have been a thread writing to a, and recovery simulates its operation when it updates disk 2.

To formalize this intuition, Perennial introduces the notion of *recovery helping*, where recovery completes the operation of a thread running prior to the crash. We borrow the term "helping" from a similar concept in lock-free programming [18] where one thread completes another's operation. We found that Iris reasoning techniques for verifying concurrency helping could be adapted to handle recovery helping.

The implementation of recovery helping uses techniques we have already described. Recall that Iris refinement proofs

use a $j \Mapsto op$ assertion to represent a pending source-level operation in a forward simulation proof. Because this is just another capability, we can use it in any invariant — in recovery helping, we store a $j \Mapsto op$ assertion in the crash invariant temporarily, and if the system crashes, the recovery proof uses the stored assertion to justify completing the high-level operation from before the crash.

The replicated disk recovery procedure uses helping when it copies from the first disk to the second. The crash invariant says that for every disk address a where disk 1 has value v_1 and disk 2 has value v_2, if $v_1 \neq v_2$, then $j \Mapsto$ `Write(a, v1)`. Ordinarily the two values agree and the implication is vacuously true, but in the middle of an update the values on disk differ. Concurrent threads do not observe this difference due to locks, which protect leases on the two $d_i[a]$ capabilities, but recovery might due to a crash. If the system does crash in the middle of a write, recovery uses the helping assertion to simulate a write from just before the crash. Note that this helping assertion is per address, so recovery might simulate many operations, in any order; this is still a forward simulation, since those writes were concurrent and thus can be simulated in any order.

5.5 Verifying Refinement

With these extensions for reasoning about crashes in the middle of concurrent code, we are now ready to explain how Perennial can be used to prove a concurrent refinement by applying the following general theorem:

Theorem 2 (recovery forward simulation)**.** If a system's implementation satisfies the following properties:

- **operation triples**: for all operations op and their implementations `op_impl`,

$$\{j \Mapsto op * \boxed{AbsR_n}\}$$
$$\texttt{op\_impl()}$$
$$\{v.\ j \Mapsto \text{ret } v\}_n$$

- **recovery triple**: the recovery procedure satisfies

$$\{\Mapsto \text{Crashing} * \boxed{CrashInv_{n+1}}\}$$
$$\texttt{recover()}$$
$$\{\_. \Mapsto \text{Done} * \boxed{AbsR_{n+1}}\}_{n+1}$$

- **crash invariance**: for all n, $AbsR_n \implies CrashInv_{n+1}$
- **idempotence**: for all n, $CrashInv_{n+1} \implies CrashInv_{n+2}$

then the implementation is a concurrent recovery refinement.

Compared to Theorem 1, the main difference is the addition of a recovery triple and side conditions that relate the abstraction relation and the crash invariant; the only other difference is that Hoare triples in Perennial are parameterized by a current version n. Note that in the Coq formalism, initial conditions are part of the specification of the system. In this informal presentation of the theorem we omit preconditions related to initial conditions, which establish the abstraction relation in the first place.

This theorem introduces two new capabilities we have not yet explained: $\Mapsto$ Crashing and $\Mapsto$ Done. These are capabilities to express progress in a recovery refinement proof; they are analogous to $j \Mapsto op$ and $j \Mapsto \text{ret } v$, except they represent a spec-level crash transition rather than an operation transition. The rules for these capabilities permit replacing source$(\sigma) * \Mapsto$ Crashing with source$(\sigma') * \Mapsto$ Done when the specification allows a crash transition from σ to σ'.

The recovery triple proof demonstrates that recovery restores the abstraction relation AbsR following a crash. If the system halts at any time, the operation crash-refinement triples guarantee that CrashInv holds. If this invariant uses memory version n, then after a crash the new memory version is $n+1$. The developer must design the crash invariant to hold even after a crash by only referring to durable resources, which is enforced by the crash-invariance property. Due to the possibility of crashes during recovery, the recovery triple must also preserve the crash invariant in the same way all regular operations do. This requirement is enforced in two ways: first, the crash invariant is also a crash invariant for recovery, as specified by $\boxed{\text{CrashInv}}$; and second, the idempotence condition requires that the crash invariant be itself crash invariant by only referring to durable resources. The requirement that recovery maintain a crash invariant corresponds to the *idempotence* principle identified in previous sequential verification systems [5, 7, 31, 34].

In the case of the replicated disk, recovery copies data from the first disk to the second. The abstraction relation includes lock invariants that require that the values on both disks agree. This property can be violated by a crash in the middle of a critical section — the lock invariant is not guaranteed to hold at all times, but the crash invariant is. Recovery uses a recovery helping assertion in the crash invariant to justify completing any in-progress writes, restoring the synchrony of the disks and establishing all the lock invariants and thus the abstraction relation. The proof also synthesizes the recovery leases for the abstraction relation corresponding to the new memory version number $n + 1$.

6 Verifying Go code with Goose

To verify real, runnable systems we developed Goose, a subset of Go amenable to reasoning in Perennial. Goose includes the core of the Go language, including slices, maps, structs, and goroutines (lightweight threads). The developer can directly compile and run this source code using the standard Go compiler toolchain. The most relevant part of Goose for this paper are the capabilities we implemented to model Go data in Iris, including pointers, files, and OS file descriptors.

6.1 Modeling shared memory

Goose needs a semantics for operations on pointers and slices since these are fundamental components of writing Go code. Modeling Go's shared memory support requires care: the Go memory model [11] specifies that accessing data simultaneously from multiple goroutines (lightweight threads) requires serialization, for example using locks. This requirement is important to ensure that on real hardware with weak memory (for example, x86-TSO for the Intel x86 architecture) Go can use efficient loads and stores yet ensure threads observe a sequentially consistent view of memory.

Goose enforces serialized access to shared data (pointers, slice, and maps) by making racy access to the same data undefined behavior. A *race* is formally defined as any instance of unordered accesses to the same object where at least one is a write. Perennial disallows systems that encounter this racy behavior by modeling writes, such as a store `*p = v`, as two atomic operations, a start and an end, and making it undefined behavior in Perennial for a procedure to ever overlap a write with another operation on the same pointer. All refinement proofs must show the code never triggers undefined behavior. This is easy to do in Iris since the capability for accessing pointers, written $p \mapsto_n v$, represents *exclusive* access to the pointer p; threads obtain this exclusive access either by allocating a new pointer and not sharing it, or by mediating access with locks. Because this resource is in memory, it refers to the current memory version number n (as described in §5.2). We use a variant of the same idea to model hashmap iteration, which has a similar problem with iterator invalidation.

Goose does not currently support Go's `sync/atomic` package that can be used to build synchronization primitives or do lock-free programming. Our examples did not require these operations, but Goose could be extended to include them. Goose also doesn't support Go's interfaces and first-class functions, because these features are difficult to model and weren't necessary in the examples we wrote.

6.2 Modeling the file system

Goose also includes a subset of the POSIX file-system API. The API is mostly a thin wrapper around a selection of system calls, which for simplicity only provides access to a subdirectory of the operating system's file system with a fixed layout since directories cannot be renamed or created within that subdirectory. To reason about the file system, Goose logically represents the file system with a four different capabilities, described below. These capabilities are deliberately low-level to accurately model features like hard links and the difference between paths and file descriptors.

- Directories: $dir \mapsto N$ states that the directory *dir* contains the set of file names N. This capability is needed to list the contents of *dir* and to add/delete files.
- Directory entries: $(dir, name) \mapsto i$ states that the contents of file *name* in directory *dir* are in the inode i. We use this to open *name* or when creating a new hard link to it.
- File descriptors: $fd \mapsto_n (i, md)$ states that the file descriptor *fd* points to the inode i, with a mode *md* (corresponding to flags passed to open; we support read and append). It references the current memory version number n since file descriptors are lost on crash.
- Inode contents: $i \mapsto bs$ states that the inode i contains the bytes *bs*. This is used with a file descriptor to access a file.

The Goose semantics includes a crash model, which describes the effects of a process crashing. As expected, on crash all data structures on the heap are lost. All file data is persisted (this is true for process crashes since the data is stored safely in the kernel), but open file descriptors are lost. Goose's semantics model every file-system operation as atomic with respect to other threads. Because file descriptors are lost on crash, they are tied to the current memory version, as in §5.2. As with all durable resources, recovery can create leases (as described in §5.3) for directories, directory entries, and inodes. It would be possible to reason about buffered data in the file system to model whole machine crashes, but our prototype does not do so.

7 Implementation

We implemented Perennial using the Coq proof assistant [38], and implemented Goose using a combination of Go for the translator and Coq for the semantics and Iris resources.[1] A breakdown of lines of code is given in Table 2. The framework consists of around 8,900 lines of code, including the transition-system domain-specific language used to write specs. The Go semantics Goose uses is around 2,000 lines of code in Perennial, which includes both a model of Go operations as well as the Iris resources to prove Go code correct.

Component	Lines of code
Transition system language	1,710
Core framework	7,220
Perennial total	8,930
Goose translator (Go)	1,790
Goose library (Go)	220
Go semantics	2,020

Table 2. Lines of code for Perennial and Goose.

The Goose translator is an executable called `goose` that translates Go to Coq and links with the Goose semantics. The translator is written in around 1,800 lines of Go. Running `goose` on supported Go code produces a Coq representation

[1] The framework and examples are at https://github.com/mit-pdos/perennial while the Goose translator is at https://github.com/tchajed/goose.

ready to import into Perennial that represents the Go code. Goose uses Go's built-in `go/ast` and `go/types` packages to parse and analyze source code: relying on these official tools helps reduce the chance of a mismatch between the translator and the Go compiler, which is important since the translator is a trusted tool. Furthermore, the Coq code must type check, which rejects unhandled code that would be difficult to detect with the translator alone. Finally, as a practical matter, `goose` produces human-readable output that is easy to audit.

8 Using Perennial to verify Mailboat

To demonstrate Perennial's usefulness, we developed Mailboat, a mail server that supports users reading and deleting their mail concurrently with mail delivery and uses a Maildir-like format to store messages using the file system. Mailboat is functionally similar to the CMAIL mail server verified using CSPEC [4], but Mailboat's proof includes a crash-safety guarantee and the implementation is lower level, using Go instead of extracted Haskell.

8.1 Specification

The verified Mailboat library implements the core operations to store, read, and delete user mail. The Go signatures of these functions are shown in Figure 10. In this section we informally describe the behavior of these operations; the Mailboat proof shows the implementation meets a more rigorously defined specification. Before executing any operations, the library requires that the caller run `Init` to initialize internal state in the library, or run `Recover` to restore the system following a shutdown or crash.

```
type Message struct {
  ID       string
  Contents string
}

func Init()                             { /* ... */ }
func Pickup(user uint64) []Message      { /* ... */ }
func Deliver(user uint64, msg []byte) { /* ... */ }
func Delete(user uint64, id string)     { /* ... */ }
func Unlock(user uint64)                { /* ... */ }

func Recover() { /* ... */ }
```

Figure 10. Go signatures for the Mailboat API.

The abstract state maintained by the Mailboat library is that of a set of users' mailboxes (one per user ID), where a mailbox is a mapping from message IDs to contents.

To read and delete mail, Mailboat requires holding a per-user lock to prevent messages from being deleted while the user is reading their mail. This lock is implicitly acquired as part of initially listing mail with `Pickup` and released with the `Unlock` operation. In practice the SMTP server calls `Pickup` when a user connects and `Unlock` when they disconnect. For simplicity the library assumes that users only attempt to delete message IDs that were returned by `Pickup`. Mailboat supports mail delivery concurrently at any time, without acquiring locks.

The signatures include mutable slices. To prove the implementation correct, the specification states precisely how the caller can use these slices. For example, for delivery to be atomic, the caller must not concurrently modify the slice passed to `Deliver`. On the other hand, the slice returned from `Pickup` is not retained by the mail library, so the caller can freely mutate it. The formal specification makes the restriction on `Deliver` precise by making concurrent modification to the slice undefined behavior, while allowing the caller to use the returned slice from `Pickup` arbitrarily.

8.2 Implementation

Mailboat stores each user's mailbox as a directory with a file per message. For crash safety, messages are spooled in a separate directory before being atomically stored in the user's mailbox. The library supports concurrent operations and guarantees that delivered mail is not lost on crash, which is achieved with the following mechanisms:

Pickup/Delete: `Pickup` reads a list of files in the user's mailbox directory, and then reads each file. To avoid a file being deleted between listing the files and reading them, pickup and delete acquire a common lock per user.

Pickup/Deliver: Concurrent deliveries are permitted during a pickup, even for the same user. To prevent reading partially written messages, `Deliver` first writes the message to a separate `spool` directory. It then atomically links the file into the user's mailbox and deletes the temporary file.

Deliver/Deliver: Multiple threads can concurrently deliver. To avoid file-name conflicts within the spool and mailbox directories, threads randomly generate IDs, retrying if the name is already taken.

Recovery: If the mail server crashes, the spool directory may contain temporary files that are no longer needed. Thus, `Recover` deletes all of the files in `spool/`. While the specification does not mandate this cleanup, doing so frees space.

Using Mailboat: We used the library to implement an SMTP and POP3-compatible mail server by implementing these protocols and interfacing with the network using Go's standard library. The protocol implementation is unverified, but works with the Postal mail server benchmarking library's `postal` tool, which delivers messages rapidly, and `rabid`, which tests retrieval by checking each message against a hash in an email header.

8.3 Proof

We highlight interesting aspects of the Mailboat proof here.

Leasing strategy. There are two types of durable state in Mailboat: the users' message files and the spool files. We use Perennial's leasing technique in order to handle the capabilities for these pieces of state.

For the messages, after a crash we need to know that all of the users' delivered message files have the right contents and are in the correct directories. Therefore, the master capabilities for mailbox directories and message files must reside in the crash invariant, just as the master permission for blocks did in the replicated disk. We store a lease on directory contents in each mailbox's lock invariant. However, the mailbox lock only prevents concurrent deletes, not concurrent delivery, so the set of files in the directory can be modified while the lock is held. Rather than locking the capability lease(dir, N), the Mailboat proof accounts for concurrent delivery using a *lower-bound* lease, written lease(dir, $\supseteq N$), that guarantees dir contains at least the files in N. The holder of the lock can use this lease to delete files, while other threads may only create new ones.

For the temporary files, recovery needs the capability to delete these files, but their contents are irrelevant after a crash. Therefore, we store the master permissions for the temporary directory in the abstraction relation, but not the inode content permission, $i \mapsto bs$, since the contents of the inode are unnecessary to unlink the corresponding file. Recall that there are no locks protecting the temporary files. Instead, deliveries allocate a name for a temporary file in the spool directory by trying random numbers until one succeeds. The `create(fname)` system call can either fail and do nothing (if the destination exists), or succeed and return a master permission and lease for the newly created file's directory entry and corresponding inode. Upon success, the master permission for the directory entry is transferred to the crash invariant, and the delivery thread retains the rest.

Abstraction relation. Following the leasing strategy just described, the high-level structure of the abstraction relation has the form:

$$\text{CrashInv}(\sigma) \triangleq \text{source}(\sigma) * \text{MsgsInv}(\sigma) * \text{TmpInv}$$
$$\text{AbsR} \triangleq \exists \sigma.\, \text{CrashInv}(\sigma) * \text{HeapInv}(\sigma) * \text{MailboxLocks}$$

These assertions correspond to the different parts of the state maintained by the mail server:

- MsgsInv(σ): This assertion connects the files representing user mailboxes to the abstract state σ of the specification, which does not mention inodes or file names. It includes capabilities for accessing the files that hold each user's mail.
- TmpInv: This tracks the temporary files in the `spool/` directory, so that recovery can clean them up after a crash.
- HeapInv(σ): This assertion tracks when a message slice is being used by `Deliver` to exploit the specification's assumption that the caller will not concurrently modify it.
- MailboxLocks: Recall that each mailbox has a pickup/delete lock to prevent a race between reading a user's message and deleting it. MailboxLocks includes capabilities for these locks with their respective lock invariants.

Example	Lines of code
Two-disk semantics	1,350
Replicated disk	1,180
Single-disk semantics	1,310
Shadow copy	390
Write-ahead logging	930
Group commit	1,410

Table 3. Lines of code for each crash-safety pattern we verified.

Exploiting undefined behavior. The proof exploits the fact that the refinement specification only applies to clients that do not trigger undefined behavior. For example, as mentioned above, clients may not concurrently mutate a message slice during a call to `Deliver`. Because the code writes out the file 4KB at a time, delivery only appears atomic in the absence of such races. Concretely, HeapInv tracks whether a message slice is being read from. Then, during the proof for `Deliver(user, msg)` we argue that `msg` remains unchanged while writing the temporary file, since any modification would trigger undefined behavior in the specification.

Recovery. Mailboat's `Recover` does not involve helping, because it just cleans up the temporary files in the `spool/` directory. With the use of leases, the proof is therefore comparatively straightforward. `Recover` takes ownership of these files via the TmpInv part of AbsR and deletes them.

9 Evaluation

To evaluate Perennial, we consider four questions:

1. Can Perennial be used to verify a variety of crash-safety patterns in concurrent systems? (§9.1)
2. What assumptions do Perennial's proofs rely on? (§9.2)
3. Does Mailboat's scale with more cores? (§9.3)
4. How does Perennial compare in terms of effort with CSPEC? (§9.4)
5. What bugs did we encounter while implementing and verifying Mailboat? (§9.5)

9.1 Crash-safety patterns

Storage systems broadly speaking use one of three patterns for crash safety: replication, shadow copies, and write-ahead logging [12]. We used Perennial to implement and verify small examples illustrating the reasoning that goes into each

of these patterns; Table 3 shows a breakdown of the lines of proof for each verified example. These examples are built using an alternate set of simpler primitives and are not compatible with Goose — the intention is to demonstrate reasoning principles rather than more reasoning about running code.

We implemented and verified a version of the replicated-disk example described throughout this paper. It demonstrates proving that failover works correctly in a simple replication system.

The shadow-copy technique implements atomic writes to storage by first performing the write on a new copy of the object, then atomically installing the new object (replacing the old version). If the system crashes, the shadow copy is invisible and its storage is reclaimed. The "Shadow copy" example in Table 3 implements atomic update of a pair of disk blocks this way. Mailboat also uses this technique with the temporary messages files that are linked atomically.

The final pattern is write-ahead logging, in which transactions are written to a log before being applied to some other storage. After a crash, the recovery procedure uses the log to delete incomplete transactions and finish applying committed transactions. We implemented a simple form of write-ahead logging to atomically update a pair of disk blocks. The proof uses recovery helping to justify completing a committed but unapplied transaction.

For better performance, logging systems buffer writes in memory before committing them; this enables an optimization called group commit in which multiple transactions are combined, amortizing the cost of committing but potentially losing buffered transactions on crash. We separately wrote and verified a simple group-commit system that does this buffering and specifies when transactions can be lost.

9.2 Trusted computing base

The proofs in Perennial rely on a number of assumptions to hold of the implementation running in the real world. The Coq proof assistant must correctly check the proofs. The Goose model should accurately reflect Go primitives and the running file system, and we trust the Go compiler to produce correct code. The Goose translator should faithfully represent the source Go code within Perennial. We assume the code does not trigger integer overflow, which Goose does not model. Finally, as usual in verification, the user must confirm that the theorem corresponds to their expected guarantees from the system. In particular, Perennial's refinement theorems apply to code that does not trigger undefined behavior in the specification; for example, the Mailboat proof assumes that `Delete` is called on messages that were previously listed.

9.3 Mailboat's performance

To show that Mailboat's throughput increases with more cores, we replicate the experiment for CMAIL described by Chajed et al. [4]. We use the same mixed workload of SMTP deliveries (i.e., `Deliver` in Mailboat) and POP3 pickups (i.e., `Pickup` followed by `Delete` in Mailboat), with an equal ratio of each. Each request chooses one of 100 users uniformly at random. The experiments were run on a server with two 6-core Intel Xeon CPUs running at 3.47 GHz, with the total requests fixed as we varied the number of cores used. Each core operates in a closed loop, issuing a new request as soon as the previous finishes, and we measure how long all the requests take to complete across all cores. Like CMAIL, Mailboat supports SMTP and POP3 over the network, but we simulated requests on the same machine to measure scalability without network overhead. Similarly, we ran the experiments on tmpfs, Linux's in-memory file system, to keep disk performance from being the limiting factor.

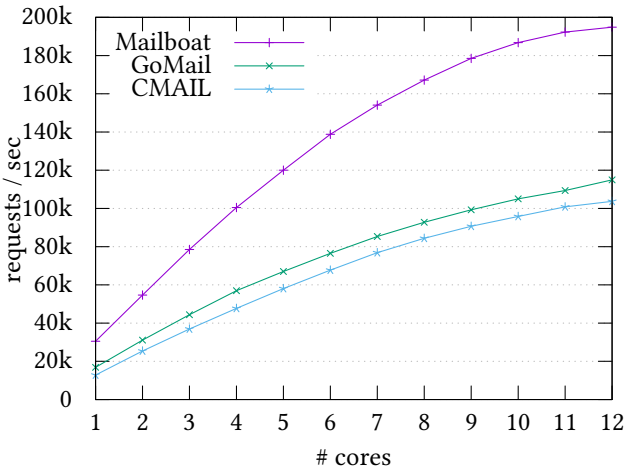

Figure 11. Throughput of Mailboat with a varying number of cores.

Figure 11 shows the performance in total requests per second for different numbers of cores. Mailboat achieves higher performance than CMAIL on a single core for three reasons. First, Mailboat is multithreaded and uses Go locks to protect mailboxes, while CMAIL runs as several processes and uses file locks. Acquiring and releasing a file lock uses several file-system calls (including opening and closing the file), which is more expensive than using in-memory locks. Second, Mailboat uses the Goose file-system library, which caches a directory file descriptors and performs all lookups relative to that root, which speeds up lookups. Third, Mailboat is written in Go while CMAIL extracts to Haskell.

To analyze the impact of each reason, we also measure the performance of GoMail, the unverified comparison from the CMAIL paper. GoMail is a mailserver written in Go in a similar style to CMAIL using file locks. Mailboat is 81% faster than GoMail on a single core because it uses in-memory Go locks instead of file locks and uses relative lookups, while GoMail is in turn 34% faster than CMAIL on a single core, which we attribute to using Go instead of Haskell. Thus, Perennial's Goose translator enables significant performance benefits.

Component	Mailboat LOC	CMAIL LOC
Implementation	159 (Go)	215 (Coq)
Proof	3,360	4,050
Framework	8,900 (Perennial)	9,600 (CSPEC)

Table 4. Lines of code for Mailboat and CMAIL.

All three mail servers scale in a similar way: throughput increases with cores, but not perfectly. All three achieve speedup because tmpfs can execute the file-system calls in parallel. Mailboat's scalability is limited by lock contention in the runtime during garbage collection.

9.4 Effort

Perennial and the Goose translator took two people 5 months to develop, and Mailboat took one person 2 weeks to verify. We compare lines of code for Mailboat and CMAIL in Table 4. Mailboat has a more concise implementation and proof, despite verifying crash safety and reasoning about mutable memory in Go.

Perennial is relatively concise compared to CSPEC for a few reasons. The main difference is that Mailboat is verified in a flattened style rather than using multiple layers of refinement. CMAIL's proof requires specifying 11 intermediate interfaces that are only used for the proof and five abstraction relations, while Mailboat's proof uses a single abstraction relation that directly connects the code to a high-level specification. The many layers in the CMAIL proof served two purposes. First, each layer applies one of CSPEC's patterns, and the CMAIL proof uses the abstraction, movers (for reasoning about concurrency), and loop patterns, each multiple times. Second, separate abstraction relations factor out the proof into modular pieces.

Perennial does not need layers to solve these problems because separation logic in Iris gives a powerful way to combine multiple reasoning patterns in a modular way. Hoare logic allows a natural decomposition of a subproof for each helper procedure. Loops are proven using a standard loop invariant approach. The single abstraction relation can be factored into different components that are connected by the separating conjunction $*$, as depicted in §8.3. Importantly, Perennial supports these three patterns using Iris rather than implementing them from scratch, so the framework itself (omitting Iris) is also fewer lines of code than CSPEC (which implements these patterns with no external support).

9.5 Bug discussion

This section highlights a few interesting bugs we encountered while developing Mailboat. One bug was that if a message was larger than 512 bytes, `Pickup` would loop infinitely. While we do not prove loops terminate, we nevertheless caught this bug while doing the proof.

A bug we did not catch during the proofs was a resource leak where a file was opened but not closed. Perennial's proofs do not cover these kind of guarantees. However, there is research on precise reasoning about resources in Iris [2].

One subtlety that the proof highlighted was that for delivery to be correct, the caller must not concurrently modify the message passed to it. While our mail server did not exhibit this bug, the proof helped us notice this requirement. We were only able to observe this because we verified and modeled Mailboat at a low level, including modeling that `Deliver` might run concurrently with arbitrary Go code.

10 Conclusion

We introduce Perennial, the first framework for verifying concurrent, crash-safe systems with machine-checked proofs. The framework is implemented using Iris, inheriting its support for reasoning about concurrency using capabilities. Perennial extends Iris with three techniques that introduce new capabilities for crash and recovery reasoning: *recovery leases* allow threads to coordinate on recovery-owned, durable resources; *versioned memory* allows the developer to precisely reason about volatile memory clearing on crash; and finally *recovery helping* allows forward-simulation proofs to reason about recovery completing operations that started prior to a crash.

To reason about systems using Perennial we introduce Goose, a subset of Go, for which we implemented a translator to Coq and a semantics in Perennial. Using Perennial we were able to verify Mailboat, a mail server written in Goose that achieves feature parity with a similar prior verified mail server, includes a proof of crash safety, yet takes fewer lines of code by leveraging features of Iris to handle the concurrency aspects of the proof. Mailboat also achieves better performance due to its lower-level implementation, thanks to the Goose approach.

Acknowledgments

We'd like to thank Butler Lampson, Jay Lorch, the anonymous reviewers, and our shepherd, Gernot Heiser, who provided comments that helped improve this paper. This research was supported by NSF awards CNS-1563763 and CCF-1836712, Google, and Oracle Labs. Tej Chajed is supported by an SOSP 2019 student scholarship from the National Science Foundation.

References

[1] S. Amani, J. Andronick, M. Bortin, C. Lewis, C. Rizkallah, and J. Tuong. Complx: A verification framework for concurrent imperative programs. In *Proceedings of the 6th International Conference on Certified Programs and Proofs*, pages 138–150, Paris, France, Jan. 2017.

[2] A. Bizjak, D. Gratzer, R. Krebbers, and L. Birkedal. Iron: Managing obligations in higher-order concurrent separation logic. *Proceedings of the ACM on Programming Languages*, 3(POPL):65:1–65:30, Jan. 2019.

[3] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel. VST-Floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning*, 61(1-4):367–422, June 2018.

[4] T. Chajed, M. F. Kaashoek, B. Lampson, and N. 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, Oct. 2018.

[5] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. 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.

[6] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, Sept. 2011.

[7] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. 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, Oct. 2015.

[8] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[9] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. 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, Jan. 2013.

[10] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming*, 131:3–21, 2016.

[11] Google. The Go memory model, May 2014. URL https://golang.org/ref/mem.

[12] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. 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, Nov. 2016.

[14] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. 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.

[15] A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming Languages and Systems*, pages 584–610, Uppsala, Sweden, Apr. 2017.

[16] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[17] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, Oct. 2015.

[18] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[19] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[20] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–34, Dec. 2017.

[21] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–70, Feb. 2014.

[22] N. Koh, Y. Li, Y. Li, L.-y. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic. From C to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th International Conference on Certified Programs and Proofs*, pages 234–248, Cascais, Portugal, Jan. 2019.

[23] B. Kragl and S. Qadeer. Layered concurrent programs. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, pages 79–102, Oxford, United Kingdom, July 2018.

[24] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems*, pages 696–723, Uppsala, Sweden, Apr. 2017.

[25] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 2017.

[26] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 357–370, St. Petersburg, FL, Jan. 2016.

[27] P. Letouzey. Extraction in Coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe*, pages 359–369, Athens, Greece, June 2008.

[28] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[29] M. O. Myreen and S. Owens. Proof-producing synthesis of ML from higher-order logic. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 115–126, Copenhagen, Denmark, Sept. 2012.

[30] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[31] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 169–188, Pohang, South Korea, Nov.–Dec. 2015.

[32] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17:1–29, Sept. 2017.

[33] I. Sergey, A. Nanevski, and A. 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.

[34] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. 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, Nov. 2016.

[35] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, Oct. 2018.

[36] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich. Total Haskell is reasonable Coq. In *Proceedings of the 7th International Conference on Certified Programs and Proofs*, pages 14–27, Los Angeles, CA, Jan. 2018.

[37] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270, St. Petersburg, FL, Jan. 2016.

[38] The Coq Development Team. *The Coq Proof Assistant, version 8.9.0*, Jan. 2019. URL https://doi.org/10.5281/zenodo.2554024.

[39] A. Turon, D. Dreyer, and L. 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, Sept. 2013.

[40] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

[41] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. 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, Oct. 2019.