Verifying concurrent storage systems with Armada

Anonymous Author(s)

Abstract

There are verified storage systems and verified concurrent systems but no verified concurrent storage systems. Crash safety and concurrency interact in challenging ways: crash-safety requires that recovery finishes operations that were started by an application thread before a crash, and recovery is a special thread: it runs only after all other threads halt and memory is cleared.

Armada is a new framework for verifying concurrent storage systems. Armada extends the Iris [22] concurrency framework with four techniques: recovery ownership, recovery leases, recovery helping, and versioned memory. To ease development and deployment of applications, Armada provides Goose, a translator for importing Go programs into Armada and reasoning about them with a model of Go threads, data structures, and file-system primitives. We implemented and verified a crash-safe, concurrent mail server using Armada and Goose that achieves speedup on multiple cores. Both Armada and Iris use the Coq [32] proof assistant, and the mail server and the framework's proofs are machine checked.

1 Introduction

Concurrent storage systems are difficult to make correct because the programmer must handle 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 [7, 8, 11, 30], but they support only sequential execution. There has also been great progress in verifying concurrent systems [6, 13, 14, 18, 21], but none support crash safety reasoning. This paper takes ideas for reasoning about crash safety and applies them to a concurrent verification system called Iris [22].

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 [17].

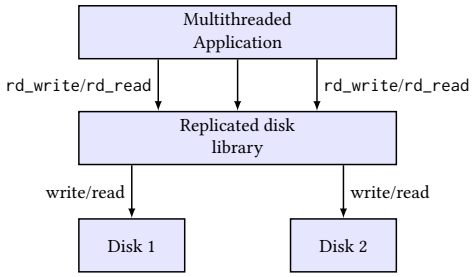

Figure 1. A concurrent, replicated disk library that tolerates a single disk failure using two physical disks. The library provides linearizable reads and writes, and transparently recovers from crashes.

Several implementations are possible, but a straightforward one uses a lock per block. With a lock per block, the implementation can guarantee that concurrent writes and reads of the same disk block are linearizable. To handle crashes in the middle of a write after one disk has been updated but before the second one is updated, the implementation must run a recovery function on reboot. The recovery function can fix up the replicated disk by copying values from the first disk to the second.

Despite a simple implementation, reasoning about the library's correctness in the presence of concurrency and crashes is diffcult. First, while the locks prevent concurrent readers and writers, they cannot prevent crashes. Instead, it is recovery's job to take control of the resources protected by locks at the time of a crash to repair the state of the replicated disk. Second, repairing must modify persistent state, which is justified by the fact that it is completing operations that were in-progress at the time of a crash. Finally, although each block is protected by a lock, recovery does not obtain these locks; this is safe because recovery runs sequentially after reboot.

A developer justifies the implementation intuitively using the reasoning outlined above. But, to prove the implementation correct against its specification using a machine-checked proof, we need to capture this intuition using precise rules that lend themselves to concise proofs. Concurrency frameworks are unequipped to handle these aspects of crashes and recovery reasoning. The core of the issue is that in the same way that concurrent programs require coordination among threads, crash safety requires coordination with crashes and recovery, which might run at any time. However, unlike threads in a concurrent system, recovery is special: it runs only after memory is cleared and other threads are halted.

, ,
2019.

This paper introduces Armada, which extends Iris with four techniques to incorporate crash safety reasoning while preserving Iris's support for concurrency reasoning. *Recovery ownership* gives recovery ultimate ownership of the resources needed to run recovery at any time. The ownership is implemented in terms of a crash invariant over durable resources that is true at every crash point. This mirrors the notion of a lock invariant, except that lock invariants can be violated by a crash that causes recovery to observe an intermediate state before a lock could be released. *Recovery leases* reconcile resource ownership with abrupt crashes and recovery by treating recovery as the ultimate owner of all system resources, and providing a *lease* on those resources to application code when recovery is not running. *Versioned memory* helps developers precisely reason about contents of memory before and after crashes, since a crash causes the computer to lose the contents of main memory. *Recovery helping* is the final technique, which reconciles what each thread was doing before a crash with what recovery code will do on its behalf to clean up, which helps justify all steps taken by recovery in terms of abstract steps allowed by the specification. For example, if `rd_write` in the replicated disk fails after writing to `Disk1`, recovery will finish up the write to `Disk2`, and recovery helping lets the application developer prove that this finishes up the interrupted execution of `rd_write`.

To build real systems, Armada provides Goose, which translates a subset of Go into an Armada program. Developers can then run the Go code using the standard Go toolchain, while writing proofs against an Armada model of Goose operations, which includes threads, pointers, slices, and a subset of the file-system API.

We used Armada and Goose to implement and verify Mailboat, which extends CMAIL [6] with crash handling. The proof shows that after recovery all delivered messages are durably stored and that concurrent readers only observe complete messages. To further evaluate Armada's reasoning principles, we verified other examples on top of a simpler set of primitives that illustrate more patterns of crash safety combined with concurrency.

Our contributions are the following:

- Armada, a system that supports machine-checked proofs of concurrent storage systems that that uses *recovery ownership*, *recovery leases*, *recovery helping*, and *versioned memory* to support crash-safety proofs on top of Iris's support for concurrency reasoning.
- Goose, infrastructure for importing a subset of Go into Armada, together with a model of Go's heap operations as an Armada library.
- Mailboat, a mail server with a proof of atomicity and durability. The mail server is written in Go and uses Goose to integrate with Armada for the proof.

Our prototypes of Armada and Goose have some limitations. Armada does not currently support composing layers of abstraction; we believe that extending multilayered frameworks like CertiKOS [13] and Argosy [7] to the concurrent crash setting is feasible. Goose does not support the entire Go language and it would be difficult to support all language features. Goose's file-system model does not support deferred durability, but we believe that this is not a fundamental limitation.

2 Related Work

Verified crash safety Recently several verification frameworks have tackled the problem of crash safety of sequential systems, including several verified file systems [7, 8, 11, 30]. These systems address several 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 over existing ones. The Fault-Tolerant Concurrent Separation Logic (FTCSL) framework of Ntzik et al. [28] does support concurrency, but that work does not prove linearizability for an entire implementation, and it has only been used for pencil and paper proofs of pseudocode, rather than machine-checked proofs about runnable code.

Concurrent verification There are many approaches to verifying concurrent software [6, 10, 14, 21, 22, 29]. 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: Ntzik et al. [28] re-use the Views framework [10], but they still requires a new formalism, an encoding into Views, and a proof that the resulting theorems have the right meaning in the context of recovery execution. This reasoning is all carried out on paper rather than in a machine-checked way; any mistake in this reasoning could render any proofs built on top of the framework invalid.

In contrast, Armada introduces techniques to encode crash safety into Iris and then has a machine-checked proof that takes any application-level proof of a system's correctness and turns it into a simple refinement theorem that makes no mention of Iris. While the techniques in Armada leverage Iris because of its flexibility, the resulting theorem statements can be understood by a developer without knowledge of Iris.

Distributed Systems There are also verified distributed systems [16, 24, 33]. Of these, only Verdi [33] attempts to

prove a correctness property in the presence of node failures. Verdi assumes that it has access to a correct high-level storage API, and verifies replication systems that hide failures at the abstract level. Armada addresses storage systems that have more complex interactions between concurrent operations and crashes, especially when crashes cannot be hidden from clients of the storage system. Armada 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 extraction and importing have been explored in prior work; there are many projects based on Coq's built-in extraction functionality [25], other languages modeled in proof assistants that can be exported to source code [3, 9, 27], as well as tools to import code in other languages into a proof assistant [5, 15, 20, 31]. Of the prior approaches, few support concurrency: the major exceptions are VST and CompCert **Tej:** they support concurrency but they focus on verified compilation, not a program logic for concurrency. Goose is the first system we know of to support Go. While our approach does not include a formally-verified compiler, verification of Go programs in Armada is lightweight enough that we were able to verify a realistic system with much less effort than either developing VST and CompCert or using them to verify programs. **Tej:** need to be somewhat careful but I don't know what big systems have been verified with VST; the DeepSpec web server seems to be the first big example of using all the tools together and it isn't done

Tej: the Oeuf paper from CPP '18 has a bunch of systems in their intro that do extraction

Verified mail servers There are some existing proofs of mail servers in other concurrency frameworks [2, 6]. We verify Mailboat, a mail server with similar functionality to CMAIL [6], but with two important distinctions. First, our mail server includes a crash-safety proof in addition to a comparable specification of linearizability. Second, Mailboat is written in Go, while CMAIL is extracted to Haskell. Mailboat's proof is therefore carried out at a lower level of abstraction to reason about mutable memory in Go.

3 Overview

This section provides an overview of Armada. Armada uses the standard approach of proving using refinement, which we will explain using the replicated disk as an example (§3.1). To enable refinement proofs that involve concurrency and crashes, Armada provides a set of libraries (§3.2).

3.1 Refinement

Proving the correctness of a system in Armada requires showing a refinement between the system's code and its spec. 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 transitions of the spec transition system are calls to the system's top-level operations, whereas the code transitions at a finer granularity for every primitive operation in the implementation. Refinement requires that every sequence of code transitions must correspond to a sequence of spec transitions, with the same external I/O (i.e., invocations and return values of top-level functions). 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.

The replicated disk exports two operations, `rd_read` and `rd_write`. We show a specification for the behavior of these two operations using Armada's spec transition domain-specific language in Figure 2. The spec says that the replicated disk's state is a logical disk, represented as a mapping from addresses to disk blocks. The `rd_read(a)` specification returns the value of the disk at `a`, while `rd_write(a, v)` updates the disk (both operations give undefined behavior if the address argument is out-of-bounds). Armada's refinement specification for the replicated disk says these two operations are linearizable. It promises caller will observe return values consistent with each operation running atomically, even when called from multiple threads.

```
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 :=
  (* crashes have no visible effect *)
  return tt.
```

Figure 2. Specification for the replicated disk operations. Caller observe the transitions in these definitions atomically even if the system crashes, and the crash transition specifies no data is lost after recovery.

Armada incorporates recovery execution into its definition of the crash-safety aspect of refinement; the approach is conceptually similar to recovery refinement from Argosy [7], but Armada also includes concurrency. Armada's definition of linearizability not only covers concurrent calls to the library but also crashes: when the system crashes and after restart the caller runs recovery, the specification promises that the caller sees behavior consistent with atomic spec operations followed by a spec crash transition. This lets the caller abstract away from the recovery code and reason about the spec's crash transition and atomic operations, which covers all executions of the implementation followed by recovery. The code may crash and restart recovery multiple times, which is abstracted away by refinement to a single spec crash.

Figure 3 shows a simple implementation of `rd_read` and `rd_write`, which uses locks to ensure linearizability. To handle crashes, the implementation runs a recovery function after rebooting after a crash that repairs the state of the replicated disks before accepting new `rd_read` and `rd_write` requests. In the absence of recovery, a system running on top of the replicated disk might observe an inconsistency. Initially, calling `rd_read(a)` would return `v` from `Disk1`, but if `Disk1` were to fail, `rd_read(a)` would fail over to `Disk2` and suddenly return the old value that was there before `v` was written, which is disallowed by the specification. One possible implementation of the recovery function is shown in Figure 4, which copies the blocks from `Disk1` to `Disk2` to bring the disks back into a consistent state.

```
func rd_read(a) {
  lock_address(a)
  v, ok := disk_read(Disk1, a)
  if !ok {
    v, _ = disk_read(Disk2, a)
    // We assume the two disks cannot both fail
  }
  unlock_address(a)
  return v2
}

func rd_write(a uint64, v []byte) {
  lock_address(a)
  disk_write(Disk1, a, v)
  disk_write(Disk2, a, v)
  unlock_address(a)
}
```

Figure 3. Go code for replicated disk read and write. These implement the specifications in Figure 2 atomically.

The core of every refinement proof is an *abstraction relation* that both connects the code and spec states as well as defines which states are reachable to begin with. As an example, consider Figure 5, which shows an abstraction relation R between states of the replicated disk for one possible execution of `rd_write`, with no concurrency but one crash. In this example, the replicated disk crashes after writing to `Disk1` but before writing to `Disk2`. The abstraction relation uses the contents of `Disk2` to determine the spec state, 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 program crashed and restarted (albeit without having to understand the details of recovery).

To prove refinement for all possible executions, the developer uses a standard technique called forward simulation [26]. Specifically, 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 return values) should be allowed by the spec transition as well. Figure 5 is an example of proving this property for one execution; the complete refinement theorem requires a proof for all possible executions of the code.

```
func rd_recover() {
  for a := 0; a < DiskSize; a++ {
    v, ok := disk_read(Disk1, a)
    if ok {
      disk_write(Disk2, a, v)
    }
  }
}
```

Figure 4. Go code for replicated disk recovery. Recovery guarantees that writes are atomic and persistent even due to crashes, as specified by `crash` in Figure 2.

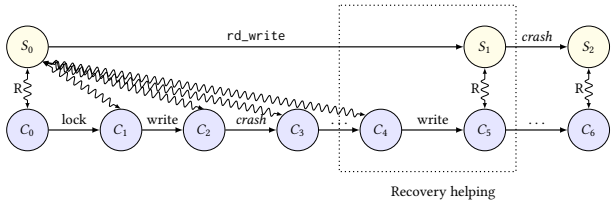

Figure 5. Refinement diagram for one execution with a crash in the middle of `rd_write`. Yellow states are spec states and blue ones are code states.

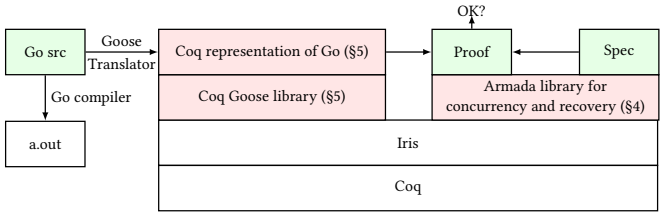

Figure 6. Overview of Armada. Red boxes are provided by Armada, while green ones are written by the developer. White-boxes are inherited by Armada.

3.2 Armada overview

To support these kind of proofs that involves concurrency and crashes, Armada is organized as shown in Figure 6. The developer writes the green boxes: code in Goose, a spec, and a proof. The proof is machine-checked and, if correct, then all possible code executions are allowed by the specification. The code can be compiled using the standard Go compiler into a binary, and then run.

To be able to reason about code, the Goose translator imports it into the Coq proof assistant, linking it with a Goose library in Coq that defines the semantics of the Go language. The semantics define how Go code executes, modeling sequential code execution, shared memory and slices, Go's built-in maps, as well as concurrent execution (including specifying certain operations as undefined behavior, to prohibit racing accesses to slice variables, for example). We describe Goose and its semantics for Go in §5.

With the code and specification defined inside of the Coq proof assistant, the remaining job of the developer is a refinement proof. To help develop such a proof, Armada provides a library with reasoning principles. Armada borrows reasoning principles for concurrency from the Iris framework, which reasons about concurrency through *ownership* of resources. For example, shared memory protected by a lock is said to be owned by the thread that acquired the lock; acquiring a lock grants ownership, and releasing a lock gives up ownership.

This notion of ownership can also be used for lock-free reasoning. For instance, in a Maildir-based mail server, renaming a message file into the `new` directory transfers ownership of the message from the delivery code to the mailbox itself. If there were a bug in the delivery code that modified the message after `rename`, it could not be proven correct, because it would be modifying a resource that it no longer owns.

Armada extends Iris with four new techniques to reason about concurrency in the presence of crashes, which we describe in the next section §4.

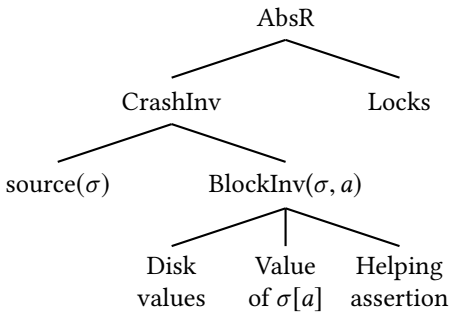

Figure 7. The structure of the abstraction relation for the replicated disk.

4 Reasoning about systems with Armada

The first challenge in proving refinement through forward simulation lies in establishing a correct and sufficient abstraction relation. To this end, Armada provides a number of techniques to help application developers write precise and powerful abstraction relations that handle both concurrency and crashes. The second challenge lies in considering all possible combinations of threads, and interleavings of their execution, in the context of forward simulation. Here, Armada introduces several proof principles that allow the developer to reason about application code and recovery code using Hoare logic, instead of explicitly considering every possible interleaving of threads and crashes. This section presents Armada's techniques using the replicated disk as a driving example.

4.1 Components of abstraction relatioship

Figure 7 shows how the abstraction relation is built from smaller components, which we explain one at a time.

At the top level, the abstraction relation AbsR has two components: CrashInv, the crash invariant, and Locks, which governs in-memory locks. The crash invariant represents durable resources — the blocks on both disks — that are logically *owned by recovery* in Armada as part of crash safety. Though we separate the crash invariant from the rest of the abstraction relation, it contains invariants that are true at all times and are thus used during normal execution as well as for crashes. The first sub-component of CrashInv is source(σ), an Iris assertion that states the current abstract state is σ, a single logical disk. This is one of the yellow, spec states in Figure 5 corresponding to the current blue, code state.

Next in the CrashInv is a BlockInv(σ, a) for each disk address a. This invariant captures everything true of address a at all times, including on crash. We will return to this invariant after describing some other components, but at a high level it captures two properties. First, $\sigma[a]$, the block in the abstract state must match the block on the second disk. During `rd_write`, when the first disk has been updated but not

$$
\begin{aligned}
\text{AbsR} &\triangleq \text{CrashInv} * \text{Locks} \\
\text{CrashInv} &\triangleq \exists \sigma.\, \text{source}(\sigma) * \left(\mathop{\Large *}_{a} \text{BlockInv}(\sigma, a)\right) \\
\text{BlockInv}(\sigma, a) &\triangleq d_1[a] \mapsto v_1 * d_2[a] \mapsto v_2 * \\
&\quad\ \sigma[a] = v_2 * \\
&\quad\ \left(v_1 \neq v_2 \rightarrow j \Rrightarrow K[\texttt{Write(a, v1)}]\right) \\
\text{Locks} &\triangleq \mathop{\Large *}_{a} \text{is\_lock}(m_\gamma[a], \text{LockInv}(a)) \\
\text{LockInv}(a) &\triangleq \text{lease}(d_1[a], v) * \text{lease}(d_2[a], v)
\end{aligned}
$$

Figure 8. Abstraction relation for the replicated disk.

the second, the logical abstract disk does not yet change in case the first disk fails during recovery, losing the write. Second, it contains what we call a *helping assertion*, which says that if the two disks have different values at address a, then some thread must be in the middle of writing to that address. The proof of recovery needs this fact to justify updating the second disk when the first disk remains operational.

Finally, the abstraction relation also has a Locks component representing the per-address locks in the replicated disk. Iris reasons about locks in terms of *lock invariants*, a formalization of the intuition that each lock protects access to particular resources. The replicated disk has a lock per address to prevent concurrent reads and writes; formally these locks give exclusive access to the address to avoid races, and guarantee that when the lock is acquired both disks have the same value.

Figure 8 shows how this abstraction relation is stated formally in Armada. It follows the structure of Figure 7, but uses Armada's new techniques, which we describe in the rest of this section.

4.2 Separation logic

Armada uses the Iris variant of separation logic to express the abstraction relation and carry out proofs about the implementation. Originally, separation logic was developed for reasoning about pointer-based data structures; Iris extends this idea from reasoning only about pointers to general support for reasoning about *ownership*. The central idea of separation logic is the *separating conjunction* $P * Q$, which represents ownership of disjoint logical resources P and Q. Resources in separation logic can be intepreted as knowledge of some fact or a capability to modify some value. For example, $a \mapsto v$ represents both the capability for disk address a, as well as knowledge that its current value is v. The replicated disk's abstraction relation uses this "points-to" notation to describe the contents of the disk in part of BlockInv.

Importantly, resources in separation logic cannot in general be duplicated, making it possible to express exclusive ownership and unforgeable capabilities. This allows the lock invariant LockInv from Figure 8 to establish the fact that no other threads can be modifying a disk address if some thread holds the lock. Note that these permissions 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.

Putting these together, we use $d_1[a] \mapsto v_1 * d_2[a] \mapsto v_2$ in BlockInv to represent ownereship of address a in both disks as well as the blocks on disk. In the replicated disk scenario Armada defines $d_1[a] \mapsto v_1$ to mean Disk 1 has value v_1 at address a *if it has not failed*. This definitions is convenient for the replicated disk since it concisely expresses the abstraction relation without many special cases. The block invariant also includes the equality $\sigma[a] = v_2$; as mentioned above, the logical single disk agrees with the second disk.

4.3 Versioned state

To extend separation logic to work across crashes, Armada introduces versioned state. Traditional separation logic has a strong notion of ownership that does not allow memory or disk locations to change their state while a thread has ownership of them. To reconcile separation logic with the fact that crashes can modify volatile resources, Armada versions all "points-to" notation with a generation number, for which we use the variable γ. For example, we write $m_\gamma[a]$ for the in-memory addresses, which the replicated disk uses to store the lock variables.

The generation number γ corresponds to a crash count; after a crash, the new version number becomes $\gamma + 1$. This allows old "points-to" facts to always be valid, but to no longer apply to the current memory. Returning to the lock example, if a lock were held before a crash, then $m_\gamma[a] \mapsto 1$ would be true. After a crash, recovery always gains exclusive ownership of the new memory, which starts out zeroed, including a resource $m_{\gamma+1}[a] \mapsto 0$.

4.4 Recovery leases

Separation logic requires ownership of a resource in order to access it. To ensure recovery can run, we logically give recovery ownership of durable resources ($d_1[a]$ and $d_2[a]$ in the replicated disk example). Recovery ownership is implemented as a crash invariant, which threads can use but must uphold after every atomic step. The same ownership concept and invariant implementation applies to locks: each lock has an associated lock invariant, which threads obtain on acquiring the lock and must restore to release it. However, these two conflict when a lock should protect a durable resource: both the lock and recovery cannot simultaneously own the same resource in an invariant or ownership would not truly be exclusive.

Armada addresses this problem by introducing a *recovery lease* to a resource $a \mapsto v$, which we write $\text{lease}_\gamma(a, v)$: the most important feature of the lease is that both the original resource and lease are needed to modify the resource. A lock can protect a durable resource by protecting a lease, while still giving recovery ultimate ownership by leaving the original resource in the crash invariant. Notice that the lease is tied to the current version of memory. The recovery-owned portion more formally also includes a version number, written $a \mapsto_\gamma v$, but we leave it implicit when it is the current generation.

Armada ties leases to the memory version number so that on crash the old leases are invalidated. Just after a crash, the recovery proof can freely take $a \mapsto_\gamma v$ and create a new lease for the next version number, synthesizing $a \mapsto_{\gamma+1} v * \text{lease}_{\gamma+1}(a, v)$. This means that the recovery procedure gains full access to the entire disk right after a crash, and before it returns it can logically give a lease for the new memory version $\gamma + 1$ to the application code (in this case, via each LockInv(a)).

In the replicated-disk abstraction relation, recovery owns $d_n[a] \mapsto v$ facts as part of BlockInv(σ, a) while leases to these resources are protected by the locks. In addition to preventing concurrent writes, LockInv(a) requires that the two disks have the same value.

4.5 Recovery helping

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, if the two disks differed at a before a crash, there must have been a thread writing to a, and thus when recovery updates the contents of Disk 2, this simulates finishing that spec-level write.

To formalize this intuition, Armada introduces the notion of *recovery helping*. The invariant BlockInv(σ, a) states that when the disks disagree at a there must be some thread writing to a, which is represented by an Armada assertion of the form $j \Mapsto K[\texttt{Write(a, v1)}]$. The recovery proof can then use this fact to formally justify the code writing v_1 to address a on Disk 2, by appearing to finish j's write operation.

4.6 Hoare triples

To prove the correctness of individual operations in the absence of crashes, the developer proves particular Hoare-logic triples about each operation. A triple $\{P\}\texttt{impl()}\{v.Q(v)\}$ intuitively means that when `impl` is run with resources P and returns v, it terminates with resources $Q(v)$. More specifically for refinement, the developer must prove a *crash-refinement triple* for each operation. To prove crash safety, the developer additionally must prove a *recovery triple* for the recovery procedure. At a high level, the crash-refinement triples demonstrate that the abstraction relation AbsR is preserved at all times, CrashInv holds at crash points, and that the behavior of each operation is as expected. The recovery triple shows that, assuming the CrashInv invariant guaranteed by each operation, recovery can correctly restore the abstraction relation.

In this section, we discuss the crash-refinement triples for the replicated disk (§4.6.1) and walk through the proof of the replicated disk recovery triple (§4.6.2).

4.6.1 Operation crash-refinement triples

The crash-refinement triples for `rd_write` is as follows:

$$\{j \Mapsto K[\texttt{Write(a, v)}] * \text{inv(AbsR)}\}$$
$$\texttt{rd\_write(a, v)}$$
$$\{r.j \Mapsto K[\text{ret } r]\}$$

JDT: it's a little unfortunate to use write when explaining the return value thing, since the return value is unit This states that, if thread j is invoking `Write(a, v)` at the spec level, then running the implementation `rd_write(a, v)` will return the correct value v that matches the spec. The inv(AbsR) in the precondition requires that the implementation maintains the abstraction relation as an *invariant*, including the crash invariant, at all intermediate points. It is important for every thread to uphold the abstraction relation, since `rd_read` or `rd_write` could be interrupted at any time, and other threads rely on it. Similarly, the crash invariant must hold at every crash point since if the system crashed, recovery would need access to durable resources in the crash invariant.

The write triple proof shows that `rd_write` preserves the crash invariant. The first interesting case is a crash after the first disk has been updated but not the second. If the disks differ, then the proof transfers ownership of the $j \Mapsto K[\texttt{Write(a, v)}]$ assertion to recovery by putting it in the crash invariant (as part of BlockInv(σ, a)). This later justifies recovery copying from the first disk to the second, if necessary, as described in §4.5. Once both disks are updated, the proof simulates a spec transition for `Write(a, v)`. This preserves, BlockInv(σ, a) now that the value `v` is on the second disk. Note that the linearization point for `rd_write` is either after it updates the second disk or as a part of recovery in the case of a crash, a complexity that Armada is able to reason about precisely.

The read triple (not shown) trivially preserves the crash invariant since it never writes to disk, but it must justify that it reads the correct value.

4.6.2 Recovery triple

In addition to proving crash refinement triples, the proof engineer must prove a single recovery triple:

$$\{\text{mem\_zeros}_{\gamma+1} * \text{inv(CrashInv)} * \Mapsto \text{Crashing}\}$$
$$\texttt{recover}$$
$$\{\Mapsto \text{Done} * \text{AbsR}\}$$

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 γ, then after a crash the new memory version is $\gamma+1$. The developer must design the crash invariant to hold even after a crash by only referring to durable resources.

The special token $\Longmapsto$ Crashing replaces the $j \Longmapsto K[op]$ tokens for spec-level operations. Instead of representing running code, it represents the spec transition system in a state just before a crash, and can be turned into $\Longmapsto$ Done within the recovery proof to simulate completing that crash. In the case of the replicated disk this step is trivial, since the logical disk in the specification state machine is unaffected by a crash.

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, as specified by inv(CrashInv). The requirement that recovery preserve the same crash invariant as it assumes is the *idempotence* principle identified in previous sequential verification systems [7, 8, 28, 30], implemented using Iris invariants in Armada.

In the case of the replicated disk, recall that AbsR is divided into two main components: CrashInv and Locks. As just described, the former is an invariant of recovery itself, so at the end of recovery, it must hold. What is left is to initialize the lock assertions. Initializing a lock requires two things: a memory address to represent the lock itself, and a proof that the lock invariant initially holds. The memory for all the locks themselves comes from $\text{mem\_zeros}_{\gamma+1}$. To restore the lock invariants, namely $\text{LockInv}(\gamma+1, a)$, the replicated disk must make the value of both disks equal. The implementation copies from Disk 1 to Disk 2, which the proof justifies using the $j \Longmapsto K[\texttt{Write(a, v1)}]$ fact from the crash invariant if the disks disagree; this is an example of reasoning about recovery helping.

Note that recovery would also be correct if it copied from Disk 2 to Disk 1. Copying from Disk 1 to Disk 2 is better in that it causes a partial write to succeed if the system crashes, but it requires more complex reasoning: recovery must complete an operation started outside its own code. Armada supports this precisely reasoning about this "helping" reasoning to prove the correctness of copying from Disk 1 to Disk 2. **JDT:** perhaps we should cut this? it's interesting but it feels like a digression.

Finally, the lock invariant holds recovery leases since ownership of the actual disk addresses is given to recovery. Recovery has exclusive ownership of $d_n[a] \mapsto v$, so it can freely generate fresh leases $\text{lease}_{\gamma+1}(d_n[a], v)$ to put in the new lock invariants that replace the now-invalidated leases $\text{lease}_{\gamma}(d_n[a], v)$ from just before the crash.

5 Verifying Go programs with Goose

To verify real, runnable systems we developed Goose, an approach for reasoning about Go code using Armada. Goose supports 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 the paper is the encoding of Go resources in Iris, including pointers, files, and OS file handles.

5.1 Modeling shared memory

Goose needs a semantics for operations on pointers and slices since these are fundamental components of writing Go programs. Modeling Go's shared memory support requires care: the Go memory model [1] 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. We represent this in the semantics by representing writes, such as a store `*p = v`, as two atomic operations, a start and an end. It is undefined behavior in Armada for a program to ever overlap a write with another operation on the same pointer. Our proofs in turn must show that the program never triggers undefined behavior. This is easy to do in Iris since the resource for pointers, written $p \mapsto_{\gamma} v$, 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 γ (as described in §4.3).

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 higher-order features are hard to model.

5.2 Modeling the file system

The Go semantics includes a subset of the POSIX file-system API that Goose exposes to programs. The API is mostly a thin wrapper around a selection of system calls, except it requires a fixed directory structure for simplicity. To reason about code that uses the file-system, the Go semantics also

includes Iris resources to represent the file system. The resources are fairly low-level to accurately model file-system features like hard links and the difference between paths and file descriptors. Goose represents the file system with four different resources:

- Directories: $dir \mapsto N$ states that the directory dir contains the set of file names N. This permission 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: $fd \mapsto_\gamma (i, md)$: 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 γ since file descriptors are part of the in-kernel state for the process and are lost on crash.
- Inode contents: $i \mapsto bs$: the inode i contains the bytes bs. This is used through a file descriptor to modify and read from a file.

The Go 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, but open file descriptors are lost. Goose's semantics for file-system operations state that they are atomic and immediately persisted. Because file descriptors are lost on crash, they are tied to the current memory version, as in §4.3. As with all durable resources, recovery can create leases (as described in §4.4) for directories, directory entries, and inodes.

6 Implementation

We implemented Armada using Coq and the Goose translator in Go. A breakdown of lines of code is given in Table 1. The framework consists of around 7,800 lines of code. The Go semantics Goose uses is around 2,100 lines of code in Armada, 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 library	1,530
Core framework	6,310
Armada total	7,840
Goose translator (Go)	1,780
Goose library (Go)	200
Go semantics	2,090

Table 1. Lines of code for Armada and Goose

Goose includes a binary `goose` that translates Go to Coq and links with the Go semantics. The translator is written in around 1,800 lines of Go. Running `goose` on a supported Go program produces a Coq file that represents the Go code and is ready to import into Armada to carry out proofs. 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 `goose` 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.

Our code is open source.[1]

Example	Lines of code
Two-disk semantics	1,440
Replicated disk	1,170
Single-disk semantics	1,390
Write-ahead logging	840
Shadow copy	340
Group commit	1,380

Table 2. Breakdown of lines of code for each storage pattern we verified.

7 Evaluation

To evaluate Armada, we consider four questions:

1. Can Armada be used to verify a variety of crash-safety patterns in concurrent storage systems?
2. What assumptions do the proofs in Armada rely on?
3. Can Armada together with Goose be used for realistic systems?
4. How much effort is using Armada?

7.1 Crash safety patterns

Storage systems broadly speaking use one of three classes of patterns for crash safety: replication, shadow copies, and write-ahead logging [12]. We wrote small examples illustrating the reasoning that goes into each of these patterns; Table 2 shows a breakdown of the lines of proof for each verified example.

The replicated disk patterns illustrates proving aspects of replication correct (namely that failover works correctly). The shadow copy technique involves making writes to storage atomic by first performing the write on a new copy of the object, then atomically installing the new object (possibly replacing the old version). If the system crashes, the shadow copy is invisible and its storage is reclaimed. The mail server uses a shadow copy to deliver mail atomically: first mail is

[1] URLs not included for double-blind submission

created in a temporary directory, then it is installed atomically with a call to `link`. Recovery reclaims the space used by shadow copies by deleting all temporary files.

The final pattern is write-ahead logging, in which transactions are written to a log before being applied to some other storage. In case of 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 "Shadow copy" example in Table 2 implements the same atomic update using a shadow copy. The logging system 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 at the cost of potentially losing buffered transactions on crash. We separately wrote and verified a simple group commit system that does this buffering and specifies precisely when transactions can be lost.

We focused our patterns on crash safety with simple concurrency (i.e., mostly using locks). There are many examples of verification of concurrent systems using Iris, demonstrating its applicability to fine-grained concurrency [23], weak memory [19], and unsafe Rust [18]. One advantage of using Iris is that the ideas in Armada can co-exist with the sophisticated features that are needed to support concurrency proofs.

7.2 Trusted computing base

The proofs in Armada 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 (although any undefined behavior is provably not triggered by the implementation). The `goose` translator should faithfully represent the source Go program within Armada. Armada's refinement theorems apply to programs that do not trigger undefined behavior in the specification; for example, the mail server proof assumes that `Delete` is called on messages that were previously listed. Finally, as usual in verification, the user must confirm that the theorem corresponds to their expected guarantees from the system.

7.3 Mailboat: a mail server verified with Armada

We used Goose to write Mailboat, a mail server that uses a Maildir-like format to store messages using the file system. The mail server supports users reading and deleting their mail concurrently with mail delivery. The mail server is structured as a library implementing the core mail management operations that interact with the file system combined with a server that implements SMTP and POP3 and is compatible with existing mail servers. The proof of Mailboat's correctness shows that pickup reads a consistent snapshot of the user's mailbox and that delivery is all-or-nothing even if the system crashes; more generally, all operations in the mail library are linearizable with respect to both concurrency and crashes.

Mailboat is functionally similar to the CMAIL mail server verified using CSPEC [6], although Mailboat's proof includes a crash-safety guarantee and the implementation is lower level. The concurrency aspect of Mailboat's specification is analogous to the guarantees from CMAIL's specification in CSPEC, though it too is lower level since it returns mutable Go data structures rather than immutable data.

7.3.1 Specification

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

func Init() { /* ... */ }

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

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

Figure 9. Go signatures for Mailboat API.

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 9. 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 `Recover` to repair the system following a crash and `Init` to initialize internal state in the library.

The abstract state maintained by the Mailboat library is that of a set of user's mailboxes (one per user ID), where a mailbox is mapping from message IDs to its 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 initially connects and `Unlock` when the user disconnects. 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 must make precise how

these slices can be used, though Go cannot express these restrictions in its type system. The slice returned from `Pickup` is not retained by the mail library, so the mail server can freely mutate it. On the other hand, for delivery to be atomic, the caller must not modify the slice passed to `Deliver`. The formal specification makes this restriction precise by making concurrent modification to the slice undefined behavior. Our implementation does not retain the slice passed to deliver, and the specification encodes this by allowing the client to use the slice freely when `Deliver` completes.

7.3.2 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 several concurrent operations while guaranteeing that on crash delivered mail is not lost. In this section we briefly describe how the implementation handles various interactions:

Pickup/Delete: `Pickup` reads a list of file names in the user's mailbox directory, and then reads each of these files. To avoid a delete in between the listing and the read, pickup and delete acquire a common lock per user. Callers of the library acquire this lock by issuing a pickup, which the mail server issues when a user connects and then releases when the user disconnects.

Pickup/Deliver: Concurrent deliveries are permitted during a pickup, even for the same user. To ensure that pickup does not observe partially written messages, `Deliver` first writes the entire message to a separate `spool` directory. Once the file is stored, the code atomically links the message into the user's mailbox and deletes the temporary file. The linking is the linearization point for delivery, because at that point the message becomes visible to subsequent calls to `Pickup`. Conversely, the linearization point for `Pickup` occurs when it lists the contents of the user's directory: although additional messages can be delivered concurrently after that point, they need not be returned.

Deliver/Deliver: Multiple threads can concurrently deliver, but they all share the same `spool` directory. To avoid filename conflicts, threads randomly generate a name for the temporary and then attempt to crate a spool file. If this process fails due to a pre-existing spool file, delivery retries with a new name. Similarly, messages need unique IDs to avoid conflicting names within the user's mailbox, which delivery similarly generates randomly. If the attempt to `link` the spool fails, then delivery tries again with a new name.

Crashes: If the mail server crashes, the `spool` directory may contain temporary files for partially-written messages that are no longer needed. Thus, `Recover` deletes all of the files in `spool`. While the specification does not mandate this cleanup, the implementation does so to free space in the file system.

7.3.3 Proof

We highlight interesting aspects of the Mailboat correctness proof here; the full proof is more complex than that of the replicated disk, so we do not present the complete abstraction relation.

Abstraction relation. The abstraction relation has the following structure:

$$\text{CrashInv}(\sigma) \triangleq \text{source}(\sigma) * \text{MsgsInv}(\sigma) * \text{TmpInv}$$

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

These assertions correspond to the different parts of program 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 resources for accessing the files that hold each user's mail.
- TmpInv: For each temporary file in the `spool` directory, TmpInv tracks ownership of the underlying storage so recovery can clean it up in the event of a crash. Note that threads coordinate access to temporary files in a lock-free manner using random names and a retry loop, so the abstraction relation does not mention any locks or leases protecting these temporary files; the appropriate leases only show up in the `Delivery` proof since they are thread-local.
- HeapInv(σ): The Mailboat library requires that the caller not concurrently modify the slice with a message while it is being delivered. The HeapInv(σ) invariant tracks when a delivery in progress so the proof can exploit this requirement to deduce that the message is immutable.
- MailboxLocks: Recall that each mailbox has a pickup/delete lock to prevent a race between reading a user's message and deleting it. MailboxLocks represents these locks and an appropriate lock invariants.

Concurrent interactions. Concurrent deliveries allocate a name for a temporary file in the spool directory by trying random numbers until one succeeds. This a lock-free coordination strategy that Iris makes simple to reason about: the `create(fname)` system call can either fail and do nothing (which happens if the destination exists), or succeed and return exclusive access to the newly-created file $(dir, fname) \mapsto i$. Recovery needs ownership to delete the temporary file in case of a crash, so the delivery proof gives recovery ultimate ownership as part of TmpInv and uses a recovery lease $\text{lease}((dir, fname), i)$ to reason about the rest of the operation.

Each mailbox uses a lock to prevent races between delete and pickup. Intuitively the lock protects the file names in the user's mailbox, the resource $dir \mapsto N$. However, it only

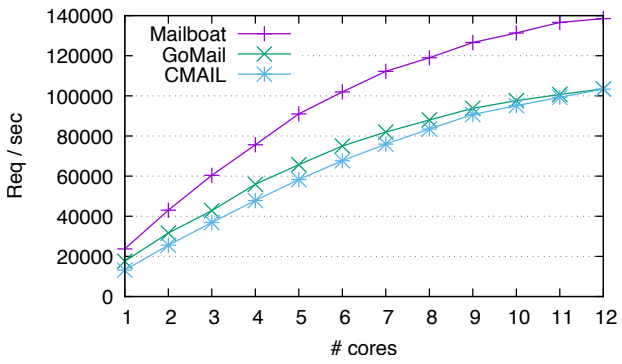

Figure 10. Throughput of Mailboat with a varying number of cores.

prevents concurrent deletes, not concurrent delivery, which does modify the set of files N. We reason about this by using not the standard lease lease(dir, N) in the mailbox lock invariant but instead a *lower-bound* lease lease$_\gamma(dir, \supseteq N)$ that guarantees *dir* contains at least the files N. This owner of this lease can delete files while knowing that other threads will only add and not delete files.

Exploiting undefined behavior. One additional complexity that arises in this example, as opposed those described previously, is exploiting the fact that the refinement specification only applies to clients that do not trigger undefined behavior. For example, consider `Deliver(id, msg)`. As mentioned above, clients are not allowed to concurrently mutate the `msg` slice. Because the implementation writes out the file 4KB at a time, delivery only appears atomic in the absence of such races. Concretely, this means that HeapInv tracks there are writes to a given slice. Then, during the proof for `Deliver(id, 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 recovery procedure does not involve helping. Instead, it just cleans up the temporary files in the `spool/` directory. With the use of leases, the proof is therefore comparatively straightforward. In the proof, `Recover` takes ownership of these files via the TmpInv part of AbsR and deletes them.

7.3.4 Experiments

To demonstrate that Mailboat's throughput increases with more cores we replicate the experiment for CMAIL [6]. We run the same mixed workload of SMTP deliveries (i.e., `Deliver` in Mailboat) and POP3 pickups (i.e., `Pickup` and `Delete` in Mailboat). The mix is an equal ratio of new messages being delivered and existing messages being read and deleted. Each request (delivery or pickup) chooses one of 100 users at random, and we run a fixed number of requests. Like CMAIL,

Component	Mailboat LOC	CMAIL LOC
Implementation	160 (Go)	215 (Coq)
Proof	3,190	4,050
Framework	7,800 (Armada)	9,500 (CSPEC)

Table 3. Comparison of lines of code for Mailboat and CMAIL.

Mailboat supports full-fledged SMTP and POP3 over the network, but we simulated SMTP and POP3 requests on the same machine to stress the scalability of the mail servers. We ran the experiment on a server with two Intel Xeon CPU, each with 6 cores running at 3.47 GHz. To keep the disk from being the bottleneck, we ran the experiments on tmpfs, Linux's in-memory file system.

Figure 10 shows the performance in requests per second for different numbers of cores for both Mailboat and CMAIL. Mailboat achieves higher performance on single core than CMAIL for two 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 requires several file-system calls (including opening and closing the file), which is more expensive than using in-memory locks. Second, 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 multiprocess mailserver written in Go in a similar style to CMAIL. Mailboat is 18s faster than GoMail on a single core because it uses in-memory Go locks, and GoMail is 23s faster than CMAIL on a single core because of Go instead of Haskell. Thus, Armada's Goose translator enables significant performance benefits.

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.

7.4 Effort

We compare lines of code for Mailboat and CMAIL in Table 3; Mailboat has a more concise implementation despite also requiring a recovery procedure, and a more concise proof despite also proving crash safety and reasoning about mutable memory in Go.

There are a few reasons why Armada is relatively concise compared to the CSPEC approach. The most noticeable difference is that Mailboat is written and verified in a flattened style rather than using layers; whereas CMAIL's proof requires specifying 11 intermediate interfaces that are only used for the proof and five abstraction relations, Mailboat's proof only requires a single abstraction relation and 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 factored out the proof into modular pieces.

Armada 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. The proof of a given implementation can be factored into subproofs, for example corresponding to helper functions in the implementation, a natural decomposition in Hoare logic. 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 §7.3.3. Importantly, Armada supports these patterns using Iris rather than implementing them from scratch, so the framework itself (not including Iris) is also fewer lines of code than CSPEC (which has no dependencies beyond Coq).

7.5 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 infinite loop; we caught this bug while doing the proof. Technically the proof does not show that loops always terminate, but it's difficult to accidentally take advantage of an unintentional infinite loops in a proof.

One bug we did not catch during the proofs was a resource leak where a file was opened but not closed. Armada's proofs do not cover these kind of guarantees, although better support for file closing idioms (e.g., Go's `defer` statement) would help prevent this kind of bug. Alternately, there is research on precise reasoning about resources in Iris [4].

An interesting subtlety that the proof highlighted for us 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 elucidated that the mail server has this requirement. It's important to note that this subtlety was only possible because we verified and modeled Mailboat at a low level, including modeling that `Deliver` might run concurrently with arbitrary Go code.

8 Summary

We introduce Armada, the first framework for verifying concurrent, crash-safe storage systems. The framework is implemented using Iris, inheriting its support for reasoning about concurrency using ownership. Armada extends Iris with four techniques that reconcile crash and recovery reasoning with ownership: *recovery ownership* treats the recovery procedure as the owner of durable resources; *recovery leases* allow threads to coordinate on recovery-owned, durable resources; *recovery helping* allows recovery to complete operations that started prior to a crash; and finally *versioned memory* allows the developer to precisely reason about volatile memory clearing on crash.

To reason about systems using Armada, we implemented Goose, a translator that converts Go into a Coq model equipped with a semantics of Go. Using Armada we were able to verify Mailboat, a mail server written in Go 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. Mailboat also achieves better performance due to its lower-level implementation, thanks to the Goose approach.

References

[1] 2014. The Go Memory Model. https://golang.org/ref/mem

[2] Reynald Affeldt and Naoki Kobayashi. 2008. A Coq Library for Verification of Concurrent Programs. *Electronic Notes in Theoretical Computer Science* 199 (2008), 17 – 32. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[3] Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joseph Tuong. 2017. Complx: A Verification Framework for Concurrent Imperative Programs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 138–150.

[4] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 65 (Jan. 2019), 30 pages.

[5] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (June 2018), 367–422.

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

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

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

[9] Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, 234–245.

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

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

[12] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.). Springer-Verlag, 393–481.

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[14] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, PA.

[15] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 584–610.

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

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

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages.

[19] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29.

[20] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. ACM, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

[21] Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. *Computer Aided Verification (CAV)* 10981 (2018), 79–102.

[22] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag New York, Inc., New York, NY, USA, 696–723.

[23] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 205–217.

[24] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, 357–370.

[25] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–369.

[26] Nancy Lynch and Frits Vaandrager. 1995. Forward and Backward Simulations – Part I: Untimed Systems. *Information and Computation* 121, 2 (Sept. 1995), 214–233.

[27] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 395–404.

[28] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-tolerant Resource Reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*. Pohang, South Korea.

[29] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 77–87.

[30] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[31] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 14–27.

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

[33] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, 357–368.