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 [21] 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 [31] 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 [6, 7, 10, 29], but they support only sequential execution. There has also been great progress in verifying concurrent systems [5, 12, 13, 17, 20], but none support crash safety reasoning. This paper takes ideas for for reasoning about crash safety and applies them to a concurrent verification system, specifically Iris [21].

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 [16].

, ,

2019.

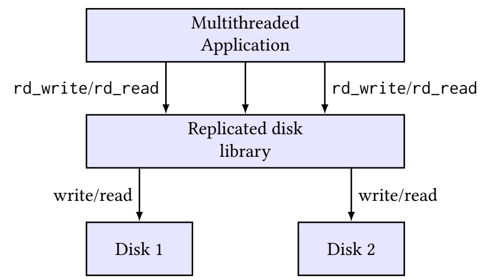

Figure 1. A concurrent, replicated disk library that tolerates a single disk failure using two physical disks. The library provides linearizable reads and writes, and transparently recovers from crashes.

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 be able 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 reasoning as 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. Unlike threads in a concurrent system, recovery is special: it runs only after memory is cleared and other threads are halted.

This paper introduces Armada, which we implemented using Iris, and which provides four techniques to incorporate crash safety reasoning into Iris while preserving its 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 helps 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, a concurrent mail server with a proof that includes 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 multiple levels of abstraction; we believe the approach taken in Argosy [6] would work, but we have not proven the appropriate theorem in the concurrent setting. Goose does not support the entire Go language, and software must generally be written with the translator's limitations in mind. The file-system model used for the mail server does not capture deferred durability and hence crashes model process termination and not power failures or kernel panics.

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 [6, 7, 10, 29]. 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. [27] does support concurrency, but that work does not prove linearizability for an entire implementation, nor does it prove running code.

Concurrent verification There are many approaches to verifying concurrent software [5, 9, 13, 20, 21, 28]. 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, and also because crash safety requires a new specification that distinguishes what is allowed if the system crashes versus if it does not. FTCSL's design highlights this difficulty: Ntzik et al. [27] re-use the Views framework [9], 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 do not reference Iris.

Distributed Systems There are also verified distributed systems [15, 23, 32]. Of these, only Verdi [32] attempts to verify a correctness property in the presence of node failures.

Verdi assumes that it has access to a correct high-level storage API, and proves 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 is applicable to verify a crash-safe, concurrent storage system that Verdi assumes.

Connecting verification to runnable code There are two broad approaches to connecting a running system and its proof. There are extraction-based approaches that take a model of the system in a form the verification system understands (for example, a program in a proof assistant) and transform it into runnable code, and there are approaches where the code is written first and then imported into the verification system where the proofs are carried out. Both extraction and importing have been explored in prior work; there are many projects based on Coq's built-in extraction functionality [24], other languages modeled in proof assistants that can be exported to source code [2, 8, 26], as well as tools to import code in other languages into a proof assistant [4, 14, 19, 30]. 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 [1, 5]. We verify Mailboat, a mail server with similar functionality to CMAIL [5], 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 as opposed to CMAIL, which is extracted to Haskell, and therefore Mailboat's proof is carried out at a lower level of abstraction to reason about mutable memory in Go.

3 Overview

Figure 2 illustrates the components of Armada. To use Armada, an application developer first writes their application code in Go, using the subset of Go supported by Armada's Goose translator. As an example, Figure 3 and Figure 4 show

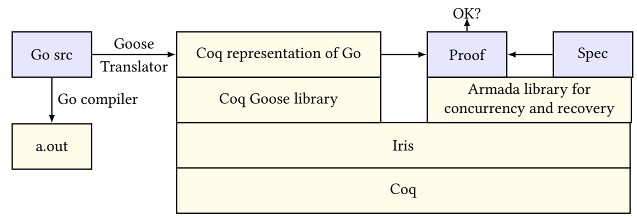

Figure 2. Overview of Armada. Yellow-shaded boxes are provided by Armada, while blue-shaded once are written by the developer.

the implementation for a simple implementation of the replicated disk. To prevent inconsistent updates and reads, each logical address is protected by a single lock that the `read` and `write` operations acquire before reading or writing from the underlying disks. The recovery function repairs the replicated disk by copying the disk blocks from the first disk to the second disk. The recovery loop is written in a particular style supported by Goose that is flexible and easy to reliably translate.

The main restriction in Goose is that the developer cannot use interfaces or first-class functions. However, Goose supports the core of the Go language, including data structures, maps, goroutines (lightweight threads), slices, etc. The developer can directly compile and run this source code using the standard Go compiler toolchain.

```go
func rd_read(a) {
	lock_address(a)
	v, ok := disk_read(Disk1, a)
	if ok {
		unlock_address(a)
		return v
	}
	v2, _ := 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.

Once the developer writes the Go source 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

```go
func rd_recover() {
	for a := uint64(0); ; {
		if a > DiskSize {
			break
		}
		v, ok := disk_read(Disk1, a)
		if ok {
			disk_write(Disk2, a, v)
		}
		a = a + 1
		continue
	}
}
```

Figure 4. Go code for replicated disk recovery. The loop is written in the format supported by the Goose translator.

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).

The Armada library helps application developers specify the expected behavior of their application in the presence of concurrency and recovery, as well as prove the correctness of their application code. For instance, in the replicated disk example, a possible specification might say that the replicated disk behaves as if there was a single logical disk, and each read and write on the replicated disk executes atomically. This specification is simple for callers to reason about, but requires the implementation to adhere to a strict guarantee.

Armada defines the correctness of an implementation against a specification as *refinement*. Specifically, this requires that every possible execution of the application code must be allowed by the specification, including all possible interleavings due to concurrency, and all possible sequences of crashes and recovery.

Importantly, recovery is not part of the specification, but rather is part of the implementation that is proven to meet the spec. Recovery's job is to fix up the state of a system after a crash, so that after recovery, it appears to meet the specification. For instance, in the replicated disk example, a system might crash during a call to `rd_write(a, v)`, after the write to `Disk1` completed, but before the write to `Disk2` started, thus leaving the replicated disks out of sync. 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.

Finally, to prove correctness (i.e., refinement), the application developer uses reasoning principles provided by the Armada library. Armada borrows reasoning principles for concurrency from the Iris framework, which reasons about

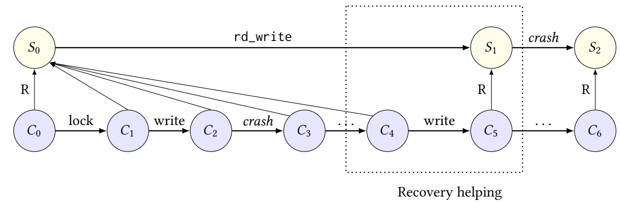

Figure 5. Refinement diagram for a crash in the middle of `rd_write`. Yellow states are spec states and blue ones are code states.

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.

4 Reasoning about systems with Armada

Proving the correctness of a system in Armada requires showing a refinement between the system's code and its specification. Both the code and the specification are *transition systems*: that is, a state that can evolve over time through a sequence of well-defined atomic steps. 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 implementation's code, and reason purely about the spec, since the spec covers all possible code executions.

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. In this example, the replicated disk is running `rd_write(a, v)` but crashes after writing to `Disk1`. 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 the replicated disk's recovery).

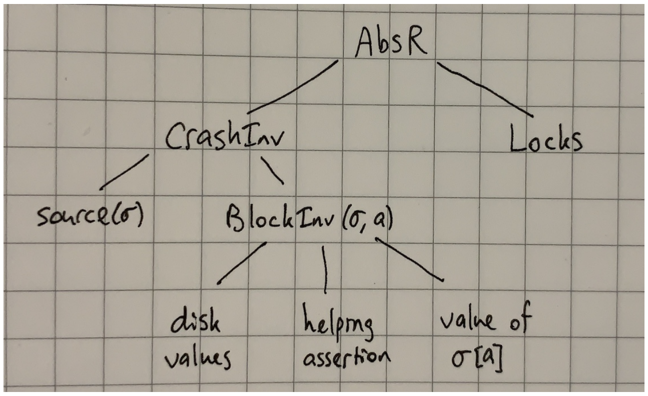

Figure 6. The structure of the replicated disk abstraction relation.

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

Figure 7. Abstraction relation for the replicated disk.

To prove refinement for all possible executions, the developer uses a standard technique called forward simulation [25]. 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. If the developer shows this to be true, it is sufficient to show a refinement relation like the one in Figure 5 for every possible code-level execution.

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. First, we introduce the intuition behind the replicated disk's abstraction relation specifically, then introduce specific concepts to express the abstraction relation using Iris. Figure 6 shows the hierarchy of 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: the CrashInv and the 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 subcomponent of CrashInv is source(σ), an Iris assertion that states the current abstract state is σ. This is one of the yellow, spec states in Figure 5, in particular the one corresponding to the current blue, code state.

Next in the CrashInv is a BlockInv(σ, a), and actually one per 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, a single logical disk, must match the block on the second disk. When the first disk has been updated but not the second, the logical abstract disk does not change in case the first disk fails during recovery, in which case the write is lost. Second, 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.

4.1 Separation logic

Armada uses the Iris variant of separation logic to express the abstraction relation and carry out proofs about the implementation. The original use of separation logic was 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. Where we wrote multiple components in

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 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. Separation logic statements can also include facts; the pure fact P is written $\lceil P \rceil$ when used as a resource.

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 7 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.

4.2 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 need for recovery to access all disk locations after a crash, 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 $m_{\gamma+1}[a] \mapsto 0$.

4.3 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, an Iris invariant that 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.

4.4 Recovery helping

After a crash, the recovery code synchronizes the contents of disk 1 onto disk 2. If the two disks differ in any location, this action needs to be justified with a spec-level transition. Informally, in the replicated disk example, this is always justified: if the two disks differed before a crash, there must have been a thread writing to that address, and thus when recovery updates the contents of disk 2 for that address, the spec will appear to finish executing that spec-level write.

To formally capture this intuition, Armada introduces the notion of *recovery helping*. To use recovery helping, the proof shows that when the disks disagree there must be some thread writing to the disk. The formal statement of the BlockInv(σ, a) denotes the existence of a spec-level thread writing with the Iris assertion $j \Rrightarrow K[\texttt{Write(a, v1)}]$, but only promises this assertion if the disk blocks differ. The recovery proof can 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.

4.5 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 (§4.5.1) and walk through the proof of the replicated disk recovery triple (§4.5.2).

4.5.1 Operation crash-refinement triples

The operation crash-refinement triples for the replicated disk are the following, one for each operation in the library:

$$\begin{aligned}&\{j \Mapsto K[\texttt{Read(a)}] * \mathrm{inv}(\mathrm{AbsR})\}\\ &\quad \texttt{rd\_read(a)}\\ &\{v.j \Mapsto K[\mathrm{ret}\, v]\}\end{aligned}$$

$$\begin{aligned}&\{j \Mapsto K[\texttt{Write(a, v)}] * \mathrm{inv}(\mathrm{AbsR})\}\\ &\quad \texttt{rd\_write(a, v)}\\ &\{r.j \Mapsto K[\mathrm{ret}\, r]\}\end{aligned}$$

These state that, if thread j is invoking `Read(a)` at the spec level, then running the implementation `rd_read(a)` will return the correct value v that matches the spec, and similarly that `rd_write(a, v)` correctly implements the specification `Write(a, v)`. 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's important every thread uphold the abstraction relation since `rd_read` or `rd_write` could be interrupted at any time, and other threads are relying on it. Similarly, the crash invariant must hold at every crash point since the system can crash at any time and recovery relies on getting access to the 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 justifies recovery copying from the first disk to the second, an instance of recovery helping. Once both disks are updated the proof simulates a spec transition for `Write(a, v)`; this still follows 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 trivially preserves the crash invariant since it never writes to disk, but it must justify that it reads the correct value. This makes use of both the abstraction relation, which connects the values on disk to the abstract state σ of the spec transition system, as well as the lock invariant. The lock invariant guarantees that after acquiring the lock, both disks agree (or one has failed), which is why `rd_read` can return the value from the first disk. The lock invariant helps simplify the reasoning for reads and writes since it makes both writes appear to execute atomically, but crashes can occur in the middle of a critical section regardless of locking so recovery can only rely on the CrashInv and writes must be careful to preserve it even during locked critical sections.

4.5.2 Recovery triple

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

$$\begin{aligned}&\{\mathrm{mem\_zeros}_{\gamma+1} * \mathrm{inv}(\mathrm{CrashInv}) * \Mapsto \mathrm{Crashing}\}\\ &\quad \texttt{recover}\\ &\{\Mapsto \mathrm{Done} * \mathrm{AbsR}\}\end{aligned}$$

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 $\Mapsto$ Crashing replaces the $j \Mapsto K[j]op$ 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 $\Mapsto$ 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.

The replicated disk proof must restore AbsR. The crash invariant already largely holds; what is left is to initialize the locks. Initializing a lock requires two things: a memory address to represent the lock itself, and the invariant that the lock protects for it to initially hold. The memory for all the locks themselves comes from $\mathrm{mem\_zeros}_{\gamma+1}$. To restore the lock invariants, namely 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 $j \Rrightarrow K[$`Write(a, v1)`$]$ fact from the crash invariant; 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.

As a final detail, 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.

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 [6, 7, 27, 29], implemented using Iris invariants in Armada.

5 Goose: verifying Go programs with Armada

So far we've illustrated the reasoning principles using the replicated disk as an example. To use Armada for real, runnable systems we implemented Goose, an approach for reasoning about Go code using Armada. Goose consists of three parts:

1. A model of Go in Coq that includes pointers, slice, maps, locks, and a subset of the filesystem API.
2. `goose`, a program that converts Go source code into an Armada representation that has the same behavior under the Go model.
3. An encoding of Go resources in Iris, including pointers, files, and OS file handles. These allow us to model ownership over the components of the file system, including reading directories, opening files, and creating hard links.

5.1 A semantics for Go

Tej: I'm using Goose to refer to this semantics — we may want to solidify using goose for the translator, Goose overloaded for the semantics/approach

In this section we highlight some interesting aspects involved in modeling Go.

Goose needs to model pointers and slices since these are fundamental components of writing Go programs. Modeling Go's shared memory support requires care: the Go memory model [cite https://golang.org/ref/mem] specifies that accessing data simultaneously from multiple goroutines (lightweight threads) requires serialization, for example using locks.

Goose enforces serialized access to shared data — pointers, slices, and maps — by making unsychronized, racy access to the same data from multiple threads undefined behavior. A *race* can be formalized as any instance of two unordered accesses to the same object where at least one is a write. Every operation in Armada is atomic, so how can two operations race? The Goose Go semantics solves this problem by representing pointer writes as a combination of separate start and end operations. Now races are detectable — a read-write race occurs whenever a read occurs between the start and end of a write, and a write-write race whenever a write starts in the middle of another write. The Go model logically tracks in-progress operations in order to implement this race detection.

Note that the operation splitting and race detection occur entirely within the *model* of Go. The code we run simply uses Go's standard pointer dereferencing and write operations, `x := *p` and `*p = v`, but the model of the `*p = v` statement has two operations, and both operations trigger undefined behavior if programs use them without proper synchronization. This imposes an obligation on all programs verified with Goose. However, it gives programs freedom to implement the synchronization in multiple ways: for example, using a lock to obtain exclusive ownership of a pointer before writing to it, or by proving exclusive ownership of a local pointer by never sharing it with other threads.

We use a variant of the same idea to model hashmap iteration, which has a similar problem with races. As is common in imperative languages, it is not safe in Go to write to hashmaps while they are being iterated, an issue known as the *iterator invalidation* problem. The semantics tracks the start and end of an iteration and defines any concurrent writes to be undefined behavior. "race detection" idea to capture iterator invalidation and make it undefined behavior; it's no longer quite race detection since iterator invalidation is possible without concurrent interleavings, since a thread can invalid its own iterator in the middle of a loop. The race detection technique captures this behavior as well; the semantics keeps track of the fact that the map is being iterated and signals undefined behavior for all inserts to and deletes from the map.

Goose does not currently support atomic operations (such as atomic integer increment or compare-and-swap) that can be used to build synchronization primitives or do lock-free programming. There's nothing fundamental to the approach that prevents this (in fact, representing non-atomic operations requires more work since atomic operations are the default), but our examples did not require these operations.

Goose also includes a library to access the file-system that supports a subset of the POSIX file-system API. The API is low-level and hews close to the system calls; the main limitation is that it supports a fixed directory structure. The Goose model of the API includes file descriptors and inodes, which are needed to model concepts like opening a file read-only, hard links, and accessing an open but deleted file. While our examples do not exploit all of the sophisticated features of the POSIX interface, modeling them precisely helps increase confidence that we didn't forget a corner case. In some places the model leaves undefined behavior, for example when linking a file that doesn't exist. This is safe since it imposes an obligation on programs to prove they avoid the undefined behavior, even though we could include and model error conditions.

The Go semantics includes a crash model. As expected, on crash all data structures on the heap are lost. Goose's current file-system crash model says that all file and metadata writes are immediately persisted, so on crash only open file descriptors are lost. This is a realistic model of process failures (which can occur in a program calling even a verified library due to a bug in the calling code, unhandled exception, or forcible termination, for example), since when a system call returns the data has been transferred at least to kernel memory. However, if the entire host fails due to a sudden power outage or kernel panic then the file system can lose unflushed data.

While we currently do not model the more severe crash model of host failures, in principle we could support using it instead. One important note is that no system can tolerate arbitrary faults; while we could prove crash safety for host failures, it would be challenging to formally verify the robustness of the system against problems like disk corruptions, and impossible to preserve any data in the case of outright disk failure; a truly robust system would use higher-level mechanisms like replication to address these issues.

5.2 Converting Go source to the Armada model

To connect a Go program to Armada, we wrote `goose`, a program that parses Go source code and outputs an equivalent Armada program that uses the Go model. For the verification guarantees to apply to the running source code, `goose` must produce an Armada representation that accurately models how the Go code will execute. For that reason we choose a subset of the syntax of Go to make the translation simple. These restrictions occasionally lead to verbose code (especially for loops) but are generally workable when writing new code.

The translator is written in Go and uses Go's built-in `go/ast` package for parsing and to represent Go syntax. Not only do these make writing the translator considerably simpler, but they help avoid mistakes where `goose` interprets the code differently from the compiler. There are a few places where `goose` relies on types to disambiguate methods that aren't apparent syntactically; these types are readily available in Go by using the `go/types` package for type checking.

As a sanity check on the translation, the resulting Coq code is itself strongly typed. For example, in Armada all side effects must be explicitly sequenced, so an operation like `x := *a + *b` is forbidden. This restriction is imposed by the Coq type system rather than by goose or through undefined or non-deterministic behavior in the semantics, so running such code through `goose` and importing it into Coq immediately triggers a type error. By doing so we completely avoid writing down semantics for Go's lexical left-to-right sequencing, which are easy to get wrong (not to mention more complicated indeterminate sequencing rules in a language like C).

Finally, as a practical matter, `goose` produces human-readable output that is easy to audit since it is written in the natural style we would have written if using extraction (except that it has better indentation).

5.3 Reasoning about Go operations

The Go semantics is written as a transition system; to actually reason about the a program emitted by `goose` within Armada, we need to encode the resources manipulated by Go programs using Iris. There are two classes of such resources: Go's built-in data (pointers, slices, and maps), and file-system resources.

We define a resource $p \mapsto_\gamma v$ that grants exclusive access to a pointer p. Ownership of this resource hides much of the complexity of the race detection in the semantics, since it is sufficient to show that no other thread is concurrently accessing the same pointer and avoid undefined behavior from races. Allocating a pointer grants exclusive ownership, at least until the thread shares it; threads can also coordinate access using locks and reason about the safety of this sharing using standard Iris mechanisms. We define a similar assertion $s \mapsto_\gamma (l_1, l_2)$ that states a slice points to an underlying array with the elements l_1 and represents a view of just the elements l_2. Both of these assertions include a memory version number since they represent in-memory resources.

Reasoning about file system resources is more involved than memory, but the same ownership principles apply. As mentioned above, we only model a fixed directory structure with top-level directories that are created before running the mail server, so we do not reason about directory creation, deletion, or a recursive directory structure. We represent ownership of these resources with the following four assertions:

- $dir \mapsto N$: 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.
- $(dir, name) \mapsto i$: 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. Note that this corresponds to a single directory entry within the directory dir.

Component	Lines of code
Transition system library	1,530
Core framework	6,270
Armada total	7,810
Goose translator (Go)	1,780
Goose library (Go)	200
Go semantics	2,090
Mailboat code (Go)	160
Mailboat proof	3,190

Table 1. Lines of code for Armada, Goose, and Mailboat.

Example	Lines of code
Two-disk semantics	1,440
Replicated disk	820
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.

- $fd \mapsto_\gamma (i, md)$: the file descriptor *fd* points to the inode *i*, with a mode *md* (corresponding to flags passed to open, though we only support read and append). This resource represents an open file descriptor. 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.
- $i \mapsto bs$: the inode *i* contains the bytes *bs*. This is used through a file descriptor to modify and read from a file.

The above resources are durable across crashes, except for file descriptors. File descriptors are therefore versioned, as in §4.2. As with all durable resources, recovery can create leases (as described in §4.3) for directories, directory entries, and inodes.

JDT: technically there is also the dirlock resource... do we want to describe this? it's a degree of realism that is superior to cspec for later, but we can only talk about it if we described the file system semantics in more detail above

6 Implementation

We implemented Armada using Coq. A breakdown of lines of code is given in Table 1. The framework consists of around 7,800 lines of code. Goose is implemented as a binary to convert Coq to Go, which is around 1,770 lines of Go, as well as an around 2,100-line semantics in Armada giving a model of Go primitives and reasoning principles for proofs. The mail server proof is 3,190 lines of code.

Our code is open source (URL not included for anonymity).

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

We verified a few examples as microbenchmarks of Armada's applicability to patterns in concurrent storage systems.

Storage systems broadly speaking use one of three classes of techniques for crash safety: replication, shadow copies, and write-ahead logging [11]. We wrote small examples illustrating the reasoning that goes into each of these techniques; Table 2 shows a breakdown of the lines of proof for each verified example. For each set of primitive operations, Armada requires a semantics and infrastructure to reason about those operations in Iris. This infrastructure is a significant amount of code, but it involves a good deal of boilerplate and only needs to be written once for a given interface. Armada currently includes three such semantics: one for Goose that supports Go and the file system which we used for Mailboat, one for the replicated disk that models failure of one of two disks, and one that has a single disk for the remaining examples.

The replicated disk example 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 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 class 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. This kind of deferred durability API is common in real systems; for example, file systems expose the `fsync` system call to explicitly request persistence, and can lose data in the case of a crash before an `fsync`.

We focused our examples on crash safety since that is the novel aspect of Armada. There are many examples of verification of concurrent systems using Iris, demonstrating its applicability to fine-grained concurrency [22], weak memory [18], and unsafe Rust [17]. 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 Assumptions

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 [5], 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

```go
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 8. 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 8. 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:

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

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

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 $\mathrm{lease}((dir, fname), i)$ 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 prevents concurrent deletes, not concurrent delivery, which does modify the set of files N. We reason about this by using not the standard lease $\mathrm{lease}(dir, N)$ in the mailbox lock invariant but instead a *lower-bound* lease $\mathrm{lease}_\gamma(dir, \supseteq N)$ that guarantees *dir* contains at least the files N. This owner of

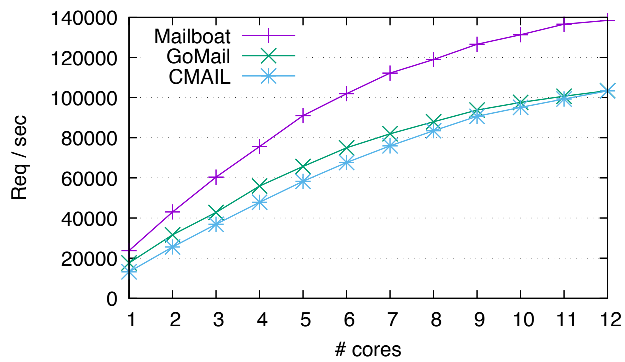

Figure 9. Throughput of Mailboat with a varying number of cores.

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 [5]. 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, 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.

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.

Figure 9 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

Using Armada requires the developer to write the system in "gooseable" Go (the subset of Go that goose supports) and prove the system using Armada's crash-safety techniques and Iris. The mail server provides a case study of this process carried out end-to-end. 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

While developing Mailboat, we naturally found and fixed some bugs. At one point there was a bug where if a message was larger than 512 bytes, `Pickup` would infinite loop; we caught this bug while doing the proof, where the loop invariant was not obvious since the loop made no progresss. Technically the proof does not show that loops always terminate, but since are proof are manual, it's difficult to take advantage of an unintentional infinite loops. We did not encounter any deadlock bugs, but deadlocking is an easy mistake to make and the proof is unlikely to catch them — orthogonal techniques to statically rule out deadlocks would be helpful even for systems verified in Armada.

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 (eg, Go's `defer` statement) would help prevent this kind of bug. Alternately, there is research on precise reasoning about resources in Iris [3].

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] 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).

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.