Verifying a concurrent, crash-safe file system with sequential reasoning

by

Tej Chajed

B.S., University of Illinois Urbana-Champaign (2014)
M.S., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Co-Certified by .....
Joseph Tassarotti
Assistant Professor of Computer Science, Boston College
Thesis Co-Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Verifying a concurrent, crash-safe file system with sequential reasoning

by

Tej Chajed

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Critical systems software such as the file system is challenging to make correct due to the combination of concurrency in the implementation for good performance and the requirement to preserve data even on crash, where the whole computer stops and reboots unexpectedly. To build reliable systems, this thesis extends formal verification — proving that a system always meets a mathematical specification of its behavior — to reason about the concurrency and crash safety.

The thesis applies the new verification techniques to the verification of DaisyNFS, a new concurrent file system. The file system is an important service of an operating system, because nearly all persistent data is ultimately stored in a file system and bugs can lead to permanent data loss in any application running on top. Another contribution of the thesis is a system design and verification techniques to scale verification to a system of the size and complexity of a file system. In particular, the file system is designed around a transaction system that addresses the core challenges of crashes and concurrency, so that the rest of the code can be verified with comparatively simpler sequential reasoning.

An evaluation of proof overhead finds that verification required 2× as many lines of proof as code for the sequential reasoning, compared to 20× for the crash safety and concurrency proofs. A performance evaluation finds that DaisyNFS achieves good performance compared to Linux NFS exporting ext4 over a range of benchmarks: at least 60% the throughput even on the most challenging concurrent benchmarks, and 90% on other workloads.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

Thesis Co-Supervisor: Joseph Tassarotti
Title: Assistant Professor of Computer Science, Boston College

Acknowledgments

How do I thank all the people that helped me throughout my PhD? I've literally left this section for last since it's the most challenging.

Let me start by thanking Frans and Nickolai, who I've been extremely fortunate to have as my advisors from the start of my PhD. Frans is a fantastic advisor in every sense, from technical advice to life advice. He has a knack for keeping track of the bigger picture, honing in on the contributions, finding the *technical nugget* in every research project, and telling just the right story. It amazes me every time how under Frans's guidance a pile of code and proofs turns into a research paper. Frans was also extremely supportive of all my extra-curricular commitments, whether it was being in the CSC, the Comm Lab, or going to PAX East. Nickolai greatly strengthened the work in this thesis. He has an awesome ability to understand the high-level and low-level details of just about anything. Nickolai equally contributed to high-level direction about what to verify, as well as to technical details like figuring out how NFS works. Frans and Nickolai are exceptionally hands-on: they contributed directly to the implementation and proofs described in this thesis.

Next I owe a huge thanks to Joe Tassarotti, my collaborator and also co-advisor for this thesis; the work in this thesis would not have been possible without him. Joe is responsible for introducing me to Iris, and for much of the development of Perennial. Our collaboration has always been productive, totally natural, and fun.

Butler Lampson has been an amazing source of clear thinking around the value of verification, both while teaching 6.826 and also as a reader on this thesis. I also worked with Adam Chlipala and Robert Morris on projects not part of this thesis but important to my journey as a researcher.

Many people helped me get started with research before grad school. My undergraduate advisor, Indy, taught me a lot about the process of conducting research through his patient and hands-on guidance, and was responsible in no small part for getting me to MIT. I also worked with John D'Angelo in the math department, who greatly increased my confidence in my mathematical ability. Before that I'm also grateful to P. R. Kumar and Praveen Kumar for giving me a chance to experience research in high school before I had any idea what I was doing.

I've had a great group of friends keeping me sane throughout my PhD, so thanks to all of you. Special thanks go to Jon Gjengset, Davis Blalock, Clément Pit-Claudel, Sara Achour, Max Wolf, Anish Athalye, Ben Sherman, Deborah Pohlmann, Ajay Brahmakshatriya, Ted Helm, and Alexandra Henzinger. Also thanks to all the folks I played board games with through these many years, and Leilani Battle and Nathan Beckmann for getting me into board games in the first place. And Ankit, Matt Tischer, Li, Liana, and David: it was great to keep in touch with you

this whole time.

I spent a lot of time hanging out with my two roommates through most of grad school, Jon and Davis. We had a lot of fun conversations, in which I could count on Jon and Davis to give both a serious and non-serious take on anything. I really value your friendship.

My time in the Comm Lab has been hugely valuable to me in my growth as a communicator and coach. A big factor in making my experience a good one has been the constant guidance and mentorship of Diana and Deanna, who have been better managers than I could have asked for.

Thank you to all of my mentees — Daniel Ziegler, Alex Konradi, Lef Ioannidis, Sydney Gibson, Sharon Lin, and Mark Theng — for putting up with me. I learned a lot from all of you. Alex, Sydney, and Mark in particular did work that directly contributed to improving this thesis.

On a technical note, the artifacts of this thesis depend crucially on Coq and Iris, so thanks to both of those communities, especially the tireless work of Théo Zimmermann, Ralf Jung, and Robbert Krebbers.

I am truly indebted to my parents in a way that I cannot really express. Thanks for supporting me from the very beginning. I knew I wanted to get a PhD even before undergrad, and they always encouraged me to follow that dream. My sister has also always been there to provide advice and perspective.

Contents

List of Figures

Chapter 1

Introduction

A promising approach for improving reliability of software is *formal verification*, in which a system is developed along with a proof that it always follows a high-level specification of its intended behavior. The first contribution of this thesis is to develop ideas and techniques for formal verification of a class of systems that store persistent data. The key challenges in such systems are that they make guarantees about *crashes* where the whole computer stops and reboots, and that their implementations take advantage of *concurrency* for good performance.

The thesis applies these verification techniques to a *file system*, the service in an operating system that implements the abstraction of files and directories. File systems are important because they are used by all applications to store data, and bugs in file systems are especially costly because they can lead to data loss for any of these applications. This thesis also contributes a design and approach that isolate crash and concurrency reasoning to a transaction system, so that the majority of the file system is verified with much simpler sequential reasoning.

1.1 Motivation

Systems that store persistent data are challenging to make correct; concurrency on its own is hard to reason about, and the possibility of a crash at any time makes it more difficult. File systems are a good instance of such a system where handling concurrency and crashes is important for correctness.

A file system is an especially critical system for three reasons. First, the file system is at a "narrow waist" in the overall software stack — essentially all applications have data that is ultimately stored in a file system. Second, implementations are concurrent and optimized to improve performance, which increases the potential for bugs. Finally, bugs are particularly costly since they can lead to permanent data loss for any application running on top.

Crashes and concurrency are the key challenges in writing a correct file system. A file system

is generally expected to keep application data safe even if the system stops running at any time, say due to a power failure or kernel panic. This thesis uses "crash" to refer to any of these circumstances where the computer stops and is rebooted. After a reboot, the file system is expected to preserve data from before the crash. The second big challenge is concurrency in the implementation. Both challenges complicate testing, fuzzing, and verification. The combination of concurrency and crashes can also lead to more intricate and hard-to-find bugs.

Even in widely used file systems like ext4 and btrfs, new bugs are still discovered, despite a long history of developing, testing, and using these file systems. For example, a recent approach for fuzzing file systems [52] found new crash-safety issues in both ext4 and btrfs, despite not testing for concurrency issues. A study conducted in 2013 looked at all patches for Linux file systems from 2005–2013 [65], finding hundreds of these patches were to fix bugs (for example, 450 for ext4 and 358 for btrfs). About 60% of these bugs lead to data loss or crash the kernel (and as the study points out, these are much more serious consequences than most bugs in application software). File systems have a lot of internal concurrency for performance reasons, which both leads to bugs and makes testing and fuzzing more challenging.

This thesis develops techniques to apply formal verification to a file system, which has the potential to completely eliminate whole classes of bugs. In this approach, we write the code, then a specification of the intended behavior of the code, and finally a mathematical proof that shows the code always meets the specification. For confidence in the proof itself, the proof is carried out with a computer and a piece of software called a proof assistant checks that the proof is valid. The nature of formal verification forces the proof engineer to systematically cover every corner case in the code. As a result, verification can rule out whole classes of bugs, including low-level bugs like memory safety but also logic errors like returning the wrong data. Verification does not guarantee that a system is bug-free, because the specification must be correct and the assumptions in the proof must hold in practice, but it does help since the specification is intended to be easier to understand than the implementation, and verification isolates debugging to identifying where the specification is wrong or an assumption was violated.

1.2 State of the art

While the idea of formal verification is not new, there was essentially no support for reasoning about the combination of crashes and concurrency when this thesis work started (in 2015). Thus this thesis develops new techniques to reason about the combination in the first place. We apply these techniques to DaisyNFS, a verified implementation of the Network File System (NFS) protocol, a standard file-system interface.

Production file systems are generally validated by testing. While testing is indispensable for development, the nature of a file system makes it difficult to catch all bugs with only test-

ing. The fundamental difficulty is a high degree of non-determinism from two sources: crashes in the middle of execution, and concurrency in the implementation that is needed for good performance. This leads to many possible execution paths, which are difficult to exhaustively test.

The importance of file-system correctness has been recognized by the academic community, thus there are many approaches for increasing confidence with improved testing. One line of work has explored systematically testing crashes at intermediate points [69, 75, 97]. Another line of work has focused on fuzz testing as a way to induce crash-safety bugs [52, 96]. These approaches have been successful for finding bugs, including crash-safety bugs, but they only test sequential executions, missing bugs due to concurrently issued operations or from crashes that interrupt multiple operations. Furthermore, unlike formal verification, testing cannot cover all executions of a program, even without crashes and concurrency, potentially missing bugs.

The research community has also recognized the value of formal verification for reasoning about a file-system implementation. The closest related work (carried out concurrently with the work in this thesis) is Flashix [4], a verified, concurrent file system that runs on flash devices. The methodology used in the proof is different from the Perennial logic we developed, as is the Flashix artifact from the DaisyNFS file system described in this thesis. See section 2.6 for a more detailed explanation.

There are other verified file systems, especially the sequential file systems FSCQ [17] and Yggdrasil [86] and an concurrent but in-memory file system AtomFS [98]. These systems use verification techniques that do not support both crashes and concurrency, and they cannot be extended them in a straightforward way to support the other form of reasoning.

1.3 Approach

What does it mean to give a machine checked, formal proof of a system? At a high level, program proofs always have three components: an implementation, a specification, and a proof. When doing machine-checked proofs, all three are physically represented as code in a verification system. The verification system checks the proof against the implementation and specification, ensuring that the proof is complete. This thesis integrates interactive, foundational proofs using custom infrastructure (in Coq [92]) as well as automated verification using a verification-aware programming language (Dafny [61]). These are both machine-checked, formal proofs, but the interaction models of the two systems are different enough that this section describes them separately. Figure 1-1 gives a high-level overview of how the pieces fit together, which this section introduces bit by bit. The system and proof are divided into two parts: the lower half of the figure represents GoTxn, a verified transaction system implemented in the Go programming language [34] (in the lower left of the figure) and verified in Coq (in the lower right), while the

upper half represents the remaining file-system code implemented and verified in Dafny.

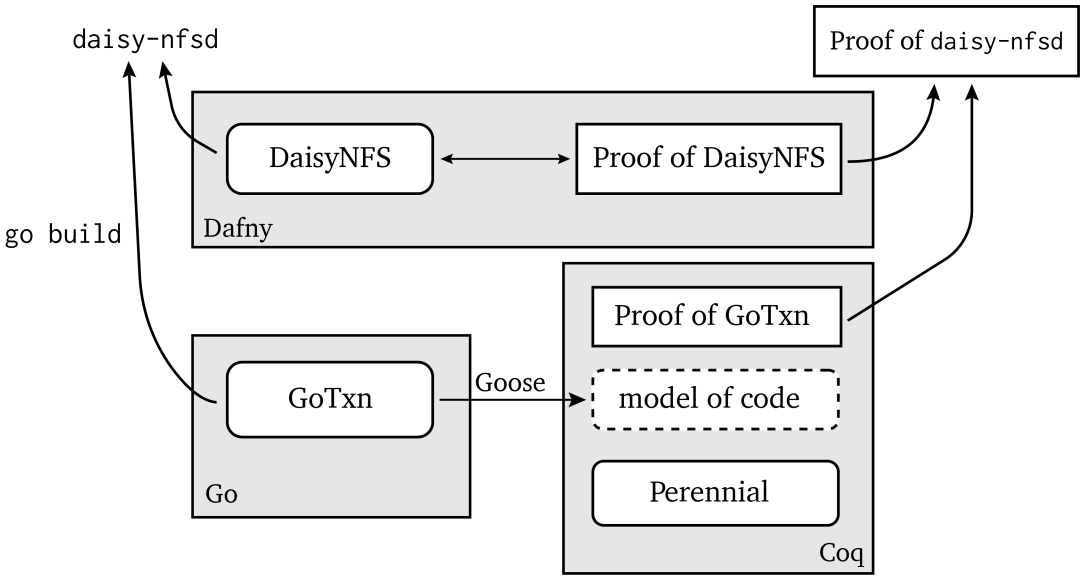


Figure 1-1: Overview of how GoTxn, DaisyNFS, and the proofs fit together.

Goose to model GoTxn in Coq. In Coq, the core feature is proofs based on dependent type theory, which is expressive enough to represent essentially any math. The first step when using Coq is to connect the Go code implementing the GoTxn library to the reasoning in the proof assistant. The particular approach in this thesis translates executable code to a model in Coq, implemented by a new tool called Goose. The model encodes the assumptions the proof makes about how the program behaves in the form of a semantics, structured as a transition system. For GoTxn, this transition system describes its execution as a sequence of states the program might go through along with return values from the library's methods.

The Perennial program logic. Once we have GoTxn modeled in Coq, we can reason about it. The goal of verification is to prove that all possible behaviors of the program are allowed by its specification. The specifications in this work forbid universally incorrect behavior, like reading an out-of-bounds address in an array, but also precisely specifies what the program is supposed to do. GoTxn has a particularly interesting specification that describes how it makes transactions appear to execute atomically, described in greater detail in chapter 5.

GoTxn has a large implementation, so it would be challenging to a direct proof reasoning about all of its behaviors. A common structure to tame the complexity of reasoning about a program is to use a *program logic*. In principle it might be possible to do this reasoning directly, but in practice such a proof would too complex to be feasible. The program logic organizes the proof with a structured way of expressing and proving statements about the program, such

as breaking the proof down into theorems about individual functions. The proof in a program logic will often mirror the structure of the code, since each function has its own specification and groups of related functions have related specs.

Program logics for concurrency are still an active area of research; only recently have they reached the maturity to give completely mechanized proofs of moderate-sized programs. There are few logics that also can reason about crash safety. Our approach in this thesis is to build a new program logic called Perennial with all the concurrency-reasoning features of a modern program logic, plus new features for reasoning about crashes. In fig. 1-1, Perennial appears in the bottom of the Coq shaded box, since it is the basis for the GoTxn proofs. Perennial builds upon Iris, a modular framework for concurrency, preserving its concurrency features while extending it with crash-safety reasoning.

Verifying GoTxn. Using the new Perennial program logic, we verified GoTxn, a concurrent transaction system. The transaction system's correctness theorem says that any program that uses transactions really has transactional behavior: its execution is equivalent to a version of the program where the transactions run atomically. The complete specification includes some important details in order to formalize the intuition behind atomicity. GoTxn is a general-purpose library for transactions, and could be used independently for another storage system.

Because every transaction appears to run sequentially and persist its writes to disk all at once, it is no longer necessary to use a sophisticated program logic like Perennial to reason about the body of each transaction. Instead, we switch to using Dafny, a verification-oriented programming language that only supports sequential code but as a result is highly productive for this use case. The file system is written and verified in Dafny, then compiled to Go and linked with GoTxn.

Implementing and verifying DaisyNFS in Dafny. The top half of fig. 1-1 depicts the implementation and proof for the DaisyNFS file-system code, implemented on top of GoTxn in the Dafny programming language. Dafny verification works quite differently from Coq. Dafny is a programming language with verification features; contrast this with Coq, which supports general math that can *model* programs. A Dafny method can be annotated with a specification. The Dafny checker converts a method and its specification to a logical formula (called a verification condition), which is true if and only if the specification holds for the method. It then queries a *solver* (Z3, in the case of Dafny) to determine if the formula is true. Contrast all of this with Coq, where the user manually develops the program logic and connects the rules of this logic. Checking a program against its specification in Dafny cannot be perfectly automated because it is impossible to answer whether a general logical formula is true or not, but the user can insert annotations to help out the solver, and generally fewer annotations are needed than lines of

proof for Coq. The main downside to this approach is that it fixes a sequential programming language, so unlike in Coq, it isn't possible to reason about concurrency and crashes.

Linking the code and proofs together. Outside of the Go, Coq, and Dafny boxes in fig. 1-1 are two outputs, one for the implementation and another for the proof. The left-hand side depicts the implementation, split between GoTxn and the file-system implemented in Dafny. The Dafny code is compiled to Go and then the two parts are linked together into one `daisy-nfsd` binary in the usual Go build process.

The right-hand side of the figure depicts all aspects related to the proof. The GoTxn proof is carried out in the Coq proof assistant, which checks that the proofs are valid. Meanwhile the DaisyNFS file-system code is verified in Dafny, which integrates implementing, specifying, and verifying code. This proof is checked by the Dafny verifier. Finally, the proof of GoTxn is written in such a way that it can be composed with the proof of DaisyNFS for a theorem about the whole `daisy-nfsd` binary. This is a conceptual composition, not one in either Dafny or Coq, which is described in greater detail in chapter 6.

One of the contributions of this thesis is that the file-system design isolates concurrency and crash safety into the transaction system. The rest of the system implements the file-system logic and data structures. Because each operation is wrapped in a transaction that appears to run sequentially and without crashes, we use Dafny for this implementation and proof. Intuitively what the file-system proof shows is that it correctly implements the NFS protocol: the `daisy-nfsd` server binary appears to atomically respond to requests according to (a mathematical description of) the NFS protocol, despite concurrent requests and even if the system crashes.

In order to connect the file system's sequential proofs to its concurrent execution, we prove a general theorem about the transaction system's implementation. The starting point is the idea of a *simulation* proof, which shows that a system like the file-system transactions correctly implement an abstract specification like the NFS protocol. The GoTxn *simulation-transfer theorem* shows that for any system implemented using transactions with a sequential simulation proof, the concurrent system running with GoTxn concurrently simulates its abstract specification where each operation is atomic. Intuitively this theorem holds because every concurrent execution of the GoTxn transactions appears to be atomic, and then the sequential simulation proof shows those atomic transactions implement the abstract specification. However, the GoTxn theorem formalizes precisely what conditions are required for the simulation transfer, including restrictions on the transactions and exactly what crash safety guarantees are achieved. Note that the simulation-transfer theorem is a general property of GoTxn that this thesis applies to DaisyNFS but which could also be applied to other verified storage systems.

1.4 Contributions

This thesis makes the following contributions:

New verification foundations. **Perennial** is a program logic for crash safety and concurrency using specifications based on crash weakest preconditions. These include crash conditions combined with concurrency and reasoning principles for moving ownership to a recovery procedure following a crash. **Logically atomic crash specifications** are a pattern using Perennial's crash weakest preconditions for specifying libraries that have both concurrent behavior and involve persistent state. These are implemented using Perennial and used in the GoTxn proof.

Reasoning about storage systems. GoTxn has a new **lifting-based specification** for its journaling layer to reason about concurrent transactions separately, which is challenging since the logical disk can change in the middle of a transaction. Its proof uses a novel **history-based abstract state** for the write-ahead log. This model allowed us to carry out a modular proof where the write-ahead logging library hides most of its internal complexity, simplifying reasoning about the rest of the GoTxn code that uses it. Finally, GoTxn exports a **transaction refinement** specification that captures how transactions appear to run atomically.

Verifying efficient code. **Goose** connects efficient code implemented in Go to a model of that code in Perennial. A general contribution here is a design for systems verification that enables efficient code and convenient reasoning. Goose includes reasoning principles for the models that it outputs, to support verifying the translated code.

System design. DaisyNFS is verified using sequential code despite having a concurrent implementation. The formal justification for this approach is a **simulation-transfer theorem**, proven on top of GoTxn's transaction-refinement specification, which shows that sequential reasoning for each transaction in a system implies correctness for the whole system when run on top of GoTxn. Sequential reasoning means DaisyNFS can be verified using Dafny, a sequential verification language, with much lower proof burden than would otherwise be possible. The specification for DaisyNFS uses a mathematical formulation of RFC 1813, the document that (in prose) specifies what a NFSv3 server should do.

The ideas in Perennial and Goose are applied to GoTxn, but they could be used for reasoning about other concurrent storage systems. Similarly, this thesis uses GoTxn to build a verified file system, but it could also be used as the basis for other verified storage systems, like a persistent key-value store.

The techniques and artifacts in this thesis do have some limitations, described in more detail in the appropriate chapters. Some examples include: Perennial can reason about safety but not

liveness (see section 3.7); GoTxn does not support asynchronous durability (see section 5.7); DaisyNFS lacks support for symbolic links (see section 6.1); and Goose does not support Go's channels or interfaces (see section 7.7).

1.5 Reading guide for the thesis

This section briefly outlines the chapters of the thesis, the dependencies between chapters, and the intended audience. Broadly the thesis is intended for a systems audience, except for the verification foundations. The support for concurrency makes all of the formal underpinnings in the thesis more technical than foundations for sequential systems verification.

Chapter 2 covers related work across Perennial, GoTxn, and DaisyNFS, for a broad computer-science audience. To keep the Goose chapter self-contained, its related work is described in the relevant chapter.

Chapter 3 is an overview of Perennial. This chapter is oriented towards a reader interested in the verification underpinnings and not necessarily the systems side. At this level of abstraction Perennial is independent of both the GoTxn proof and even Goose for verifying Go code. Some experience with program logics is helpful for understanding the presentation.

Chapter 4 describes a style of logically atomic specifications that capture both concurrency and crash atomicity using Perennial. It uses an extended example from the GoTxn proof and explains its formal underpinnings at a high level. This is the most technically involved chapter.

Chapter 5 describes the design and proof of GoTxn. An important part of GoTxn is its specification, which captures how transactions are atomic. This chapter does not require the full Perennial or logical atomicity chapters.

Chapter 6 describes the design and proof of DaisyNFS. This chapter explains the proof approach that justifies using Dafny to verify DaisyNFS even though Dafny is a sequential verification tool and DaisyNFS is a concurrent system.

Chapter 7 is about Goose and talks about verifying Go code generally, with nothing specific to GoTxn or even storage systems. This is the first detailed description of Goose, so this chapter is written to be readable without any of the other chapters.

Chapter 8 is a short chapter covering some implementation details in DaisyNFS, covering both GoTxn and the file-system code.

Chapter 9 evaluates the whole file system along a few dimensions, especially performance but also aspects of the proof.

Chapter 10 concludes with some discussion about the broader value of verification, lessons learned, and future work.

Chapter 2

Related Work

While verifying programs is an old idea, going back at least to Floyd and Hoare's work in the late 1960s [31, 44], prior to this thesis (in 2015) there was almost no work on reasoning about a program's execution in the presence of a crash, let alone the combination of concurrency and crashes. The full pipeline of systems verification is also quite new: connecting the formal reasoning to executable programs, writing machine-checked proofs, and scaling up the techniques to systems with sizable implementations.

The Perennial framework for reasoning about crashes and concurrency draws on two lines of research: sequential crash safety verification and concurrency verification, described in sections 2.1 to 2.3. DaisyNFS draws from two other lines of prior work, on verified transactions and file systems, described in sections 2.4 and 2.5.

The most closely related work is Flashix, another verified concurrent and crash-safe file system. Section 2.6 compares the two systems' contributions and results. Finally, section 2.7 discusses some related work on using transactions as part of a file system.

2.1 Foundations for verifying sequential crash safety

There are a variety of foundational tools for reasoning about crash safety, largely for sequential systems.

FSCQ [16–18] is a sequential file system verified with Crash Hoare Logic (CHL). CHL's basic specifications have the form $\{P\}\ e\ \{Q\}\{Q_c\}$, extending the Hoare triple $\{P\}\ e\ \{Q\}$ with a *crash condition* Q_c that holds at all intermediate points in the function's execution. Crash conditions handle a core difficulty of crashes, namely reasoning about the state of the system at all intermediate points. There are two remaining challenges: a crash wipes in-memory state, and the system might crash again while recovering after a crash. CHL connects the system's crash conditions to a specification for recovery to handle these two issues. A crash predicate

transformation captures what can be assumed when recovery starts, and for crashes during recovery CHL requires that recovery be *idempotent* in the sense that its crash condition implies its own precondition. CHL was used to specify and verify FSCQ [17] and DFSCQ [18], a more performant successor.

Yggdrasil [86] takes a different approach to crash safety. The basic definition is *crash refinement*, which says that a system implements an interface correctly, including a specification for what a crash followed by recovery is allowed to do. Note that unlike CHL this specification is about a collection of methods implementing an abstract, specification transition system, not about individual methods. Yggdrasil uses crash refinement to specify and verify a file system comparable to the file system in xv6, a teaching operating system. The implementation uses Z3 to check crash refinement, which the authors show is able to handle a system of this complexity by breaking down the implementation into small enough layers.

Argosy [11], which I led the development of but is not part of this thesis, combines aspects of FSCQ and Yggdrasil. The key new idea is to develop the metatheory for *recovery refinement* that shows how systems compose when both have recovery procedures — what is non-trivial to handle is that a crash in the composed recovery procedure requires starting over from the beginning. Recovery refinement can be viewed as an extension of crash refinement with verified metatheory for recovery, largely left implicit in the Yggdrasil paper. Argosy also shows how to encode the conditions of recovery refinement using CHL so that a single layer is verified using the CHL program logic.

VeriBetrKV [39] takes yet another approach to reasoning about crashes, this time friendly to encoding in Dafny, a sequential verification system with integrated support for programming and verification. The main idea related to crash safety is to adopt the style from IronFleet [41] and think of a storage system as a distributed system made up of the CPU and the storage device. The extension needed for crash reasoning is to add a crash transition to the storage device that non-deterministically wipes any buffered but unacknowledged writes, and then to show that when this happens it corresponds to an appropriate application-level transition modeling crashes (similar to the crash refinement definition from Yggdrasil and recovery refinement in Argosy, although this is associated with a code transition rather than a dedicated recovery procedure). VeriBetrKV is used to verify a persistent key-value store based on B^ϵ trees, a data structure that also underlies BetrFS [46].

Perennial has crash conditions that look similar to CHL's crash conditions, albeit as part of a concurrent program logic rather than a sequential one. We do carry out a refinement proof in the style of Argosy, which is similar to the specification style in VeriBetrKV and Yggdrasil, but because it is connected to concurrent reasoning the proof techniques are more sophisticated.

2.2 Foundations for verifying concurrent programs

There are a number of approaches proposed to verifying concurrent programs, and Iris in particular is the basis for Perennial. It would be hard to do justice to the historical development of concurrency reasoning. Looking at approaches that are actively used in research and connected to executable implementation, two broad strategies are commonly used: developing concurrent *program logics*, and using *refinement*-based techniques.

Many refinement-based techniques are based on the idea of *reduction*, which appeared originally in Lipton's theory of "movers" [63]. The idea behind reduction is to reason about a program through program transformations that show the program is equivalent to a simpler program. These techniques can reason about concurrency by showing that a concurrent program is equivalent to a program with sequential or atomic regions. These ideas have been used as part of the CIVL verifier [42, 56] and Armada [64]; my own prior work on CSPEC [10] was also based on movers, before starting Perennial. Reduction-based techniques generally reason about a program through a series of transformations, each making the program slightly simpler, keeping the proof of each transformation's correctness more manageable.

One large verified system, CertiKOS, is based on a custom verification infrastructure called Certified Concurrent Abstraction Layers [36] which based on refinement but not reduction. This work is notable for verifying a system (a simple operating system) not just at an abstract protocol level but all the way down to concurrent code. The proof composes with the CompCert compiler correctness theorem to carry the guarantees down to the assembly code of the operating system.

Program logics are an alternative approach based on giving specifications to each function in the program, within a logic that has useful rules for proving and composing specifications. Hoare logic is a classic program logic for sequential programs that based on pre- and post-conditions. Concurrency makes it harder to construct a logic that can usefully reason about many concurrency patterns (completeness) while also giving specifications that hold in the presence of concurrent threads (soundness). One productive line of work has been based on concurrent separation logic (CSL) [6].

The Verified Software Toolchain is notable for connecting a CSL-based logic (VST-Floyd) down to proofs of C code, including a connection to CompCert to carry these guarantees down to assembly [8]. It has been used for a number of sequential verified C programs and recently to some shared memory C libraries.

Microsoft VCC [20, 21] is another verification system that is based on annotating C code with pre- and post-conditions and so-called two-state invariants that are similar to rely-guarantee reasoning [30, 47]. VCC is typically used to encode a form of concurrent refinement reasoning while also using specifications similar to that of a program logic. The system was used to verify a portion of Hyper-V. The system is implemented as a verification-condition generator; unlike

the program logics described in this section, VCC has no soundness argument (even on paper) to specify what its specifications mean and argue that it enforces the right conditions for that meaning to hold.

Perennial builds directly on Iris [50]. Iris is a general framework for concurrency, featuring a base logic with key features for concurrency (step indexing and separation-logic resources, including *higher-order* resources used to define invariants for example) and a program logic built on the base logic. This decomposition was valuable for Perennial because we made two core changes to Iris: first, the program logic itself is extended with crash safety reasoning, and second, the framework is applied to our GooseLang models of Go code. In both cases the generality of Iris was valuable, as the framework is not tied to reasoning about a particular programming language or even to its usual program logic. Prior to this work, Iris had not typically been used for reasoning about executable code (though projects like RefinedC [83] and our own work on Goose are changing that).

2.3 Reasoning about crashes and concurrency

Program logics other than Perennial have been developed for formal reasoning about concurrent, crash-safe systems. Fault-Tolerant Concurrent Separation Logic (FTCSL) [70] extends the Views [27] concurrency logic to incorporate crash-safety. POG [80] is a program logic for reasoning about the interaction of x86-TSO weak-memory consistency and non-volatile memory. Both of these logics are only defined on paper, and do not support mechanized proof. Perennial goes beyond their reasoning principles: it includes ownership reasoning and its interaction with crashes, and a style of logically atomic crash specifications for modularly specifying libraries and composing their proofs. FTCSL and POG have no mechanism of modular proofs of layers, which we found essential to scale verification to a file system.

In the case of x86-TSO with persistence, POG required a line of research to define the semantics at the ISA level, validating this semantics against the hardware and in conversations with Intel engineers [79, 81]. A similar semantics effort defines the semantics of ext4 under crashes, but without an accompanying program logic [53]. It would be an interesting direction for future work to combine the persistency semantics of x86 with Perennial, to verify libraries that use non-volatile memory.

2.4 Verified transaction systems

As far as we know, GoTxn is the first transaction system that makes data durable and has a verified implementation. There is much related work on verifying aspects of transaction systems with and without durability, and on unverified transaction systems.

Chkliaev et al. [19] verify serializability of two-phase locking and other transaction concurrency control mechanisms in the PVS theorem prover. Their proof formalizes two-phase locking as an abstract protocol consisting of sequences of read, write, and locking operations, as opposed to a concrete implementation as in GoTxn. Pollak [77] uses a variant of the CAP separation logic [26] to give a pen-and-paper proof of serializability for an abstract, protocol-level description of two-phase locking, connecting the protocol to atomic reasoning about transactions.

Lesani et al. [62] developed a framework for verifying software transactional memory algorithms, modeled as I/O automata. They applied their framework to sophisticated STM algorithms, such as NOrec algorithm [25]. The STM algorithms considered do not handle persistence, so the framework does not address crash-safety reasoning.

A specification called the Push/Pull model of transactions [55] is similar to the *lifting* technique in the journal system's specification (section 5.5.1) — the core problem addressed is that a journal operation atomically modifies a small number of objects, but other objects can change between the start of the operation and when it commits. The Push/Pull model also discusses reasoning on top of the specification, using Lipton's reduction [63] rather than separation-logic ownership to handle concurrency. However that work is about on-paper specifications and proofs, while we also prove an implementation meets our specification and proved DaisyNFS on top.

DFSCQ [18] verifies a high-performance file system built on top of a logging system with asynchronous disks and log-bypass writes, which are challenging optimizations that GoTxn does not support. ARIES is a database write-ahead logging protocol that was verified (with a pen-and-paper proof) in FTCSL [70]; it is more sophisticated than GoTxn's write-ahead log in that it can both undo and redo operations.

An important but unverified journaling system is jbd2, the "journaling block device" that underpins ext3 and ext4 in Linux. The design of jbd2 is broadly similar to that of GoTxn's journaling layer; GoTxn goes one step further and also has automatic locking for transactions. One difference in the interfaces of GoTxn and jbd2 is that in jbd2, transactions can reserve space ahead of time. This has the benefit that a transaction can start writing to the journal on disk while it is being prepared, potentially improving performance, but it also means that a concurrent transaction must wait for all previously started transactions to finish before becoming persistent, since a journaled operation can only be logically completed when all prior operations are finished. This is a reasonable tradeoff for a file system since transactions are generally short-lived, but it sacrifices a bit of tail latency when an operation must be persisted, for example when an application issues an `fsync()`.

2.5 Verified file systems

The Dafny side of DaisyNFS is a new implementation but its design and aspects of the proof strategy were inspired by other verified file systems like DFSCQ [18] (especially its indirect block implementation described in Alex Konradi's master's thesis [54]) and Yggdrasil [86].

AtomFS [98] is a verified, concurrent file system, but its implementation does not store data durably. The proof structure is quite different from DaisyNFS in that all of the concurrency reasoning is part of the file-system operations. AtomFS is verified using a relational logic with rely-guarantee reasoning; unlike separation logic, the basic specification in this logic is a refinement statement that relates an implementation to a (simpler) specification program.

We chose to verify an NFS server because it is widely used in practice and the expected behavior of NFS operations is well documented in RFCs. FUSE is an alternative for implementing file systems in user space that was used for the verified file systems mentioned above, but its operations have a less clear specification. FUSE also increases the trusted code to include both the FUSE kernel component and the user-space library that connects the implementation to the kernel.

2.6 The Flashix file system

The Flashix project deserves special attention since it also develops a verified concurrent and crash-safe file system (carried out concurrently with the work in this thesis) [4]. Flashix targets flash storage, a lower-level storage technology than the drives that DaisyNFS targets. Crash and concurrency reasoning in Flashix is based based on refinement, in particular using Lipton's theory of movers [63], extended with conditions for crash safety [73, §13.3]. Flashix is implemented using abstract data types in a high-level language; a code generator transforms this code into executable C, but this process is both not verified and has difficulty producing the most efficient code using in-place updates.

Refinement in Flashix is defined in terms of "components" which are like the classes in an object-oriented language. The proof is split into two parts, corresponding to concurrency and then crash reasoning. First, the proof shows that the entire file-system implementation component is refines an abstract file-system component where the operations have `atomic` blocks in their implementation. This proof is carried out using *movers*, mostly automated due to the use of locks. Next, the proof reasons about the effect of a crash and abstracts those to a simple transition in the top-level specification. The former is an atomicity refinement that shows operations appear to behave atomically, while the latter is a crash refinement that shows crash safety. We also have some experience with mover-style reasoning from CSPEC [10], but ultimately decided to build on top of concurrent separation logic instead and developed Perennial to verify GoTxn.

The Perennial program logic gives both specifications and proofs in a different way than Flashix. The smallest unit of specification is a judgment in the logic about an individual procedure rather than a whole component. This specification covers crash and concurrency behavior together, rather than separating them into two refinement steps. The GoTxn implementation naturally has libraries whose methods are specified together, much like a component, for which on top of Perennial we developed a pattern for logically atomic crash specifications. The advantage of reasoning about crashes and concurrency together is that Perennial can also reason about lock-free code, including code that persists data without holding locks. In Flashix, the proofs always first reduce the concurrency atomicity and then reason about crashes with the reduced atomicity. GoTxn has internal concurrency that doesn't fit this pattern, where its background threads write to disk without locks.

Perennial comes with a soundness theorem that allows to extract a final theorem about executions, proven in the Coq proof assistant. It is used to show a sophisticated transaction refinement theorem for GoTxn: unlike the component refinements in Flashix which are about fixed operations, this specification involves showing arbitrary code inside transactions has serializable behavior. In contrast the Flashix refinement approach is implemented in the KIV theorem prover as axioms — that is, the system generates proof obligations based on the theory for a program being verified. However, the connection between the obligations and the desired theorem about executions is given by a set of on-paper proofs showing the methodology is sound.

The concurrency in Flashix is different from in DaisyNFS. At the file-system layer, the system has a reader-writer lock over the entire directory structure, and additionally supports concurrent writes to different files. Internally, the system can also perform garbage collection and erase-block management concurrently with file-system operations. DaisyNFS in contrast can concurrently write to distinct files or directories, and concurrently installs from the write-ahead log to the data region, but does not support read-read concurrency to the same file. Flashix has only lock-based concurrency, whereas DaisyNFS also has lock-free access to the disk in the write-ahead log. Flashix achieves comparable performance to UBIFS (an existing flash file system in Linux) [4], and DaisyNFS achieves comparable performance to the Linux kernel NFS server. The Flashix benchmarks do not evaluate scalability to many cores, focusing on the impact of concurrent garbage collection in experiments that ran with three cores.

2.7 Transactions to simplify system design

To be conducive to verification, DaisyNFS is implemented differently than many NFS servers; in particular, using two-phase locking is not common practice. Other user-level NFS servers are typically implemented on top of an existing file system, relying on the underlying file system for logging and locking. Ext3 and ext4 use a journaling system underneath, but the file system

and VFS layers perform locking. This locking is inherited when these file systems are exported with the Linux NFS server. WAFL [43] is an NFS appliance that provides snapshots and logs NFS requests to NVRAM. It has evolved its locking plan to obtain good parallelism [23]. Both the Linux NFS server and WAFL are more complicated and have more features than DaisyNFS.

Isotope [85] is a block-level transaction system similar to GoTxn in its API, but without formal verification, which was used to implement a file system called IsoFS. Its logging design is based on multi-version concurrency control (MVCC) [3] rather than our use of pessimistic locking. The Isotope paper uses Isotope not only for the IsoFS file system but also two persistent key-value stores. While GoTxn is in principle also suitable to implement a key-value store, we have so far only used it in combination with DaisyNFS.

IsoFS has a similar design to DaisyNFS: it factors out isolation and atomicity to the transaction system, making it easy to handle crashes and concurrency. Unlike GoTxn and DaisyNFS, Isotope is still prone to subtle concurrency bugs in the transaction system and bugs in the IsoFS code, whereas DaisyNFS uses the split design to verify both the transaction system and the transactions themselves. The transaction API provided by Isotope has an interesting performance optimization that GoTxn could support: the API includes a `please_cache` call that keeps an address in the transaction system's cache, used for small but frequently-accessed metadata like the allocator state in the file system.

Chapter 3

Perennial

Perennial is a framework for reasoning about crash safety and concurrency that we developed in order to verify GoTxn. The main component of Perennial is a program logic based on concurrent separation logic, with extensions for reasoning about crash and recovery behaviors.

Who is this chapter for? This chapter describes Perennial for an audience with some programming languages or verification background, but not necessarily experience with concurrency specifically. The high-level ideas are meant to be broadly accessible even if some background on Iris is needed to appreciate the details. To that end the presentation is somewhat simplified, with details and side conditions omitted from the logical rules. A more systems-oriented reader can safely skip this chapter and the next (chapter 4) and still understand the later chapters.

3.1 Primer on Iris and separation logic

A program logic is a formal system for specifying and reasoning about programs. One of the simplest program logics is Hoare logic, still the basis for much sequential reasoning today. The judgments of Hoare logic consist of specifications of the form $\{P\}\ e\ \{Q\}$ (which might be pronounced "e requires P and ensures Q"), interpreted as meaning "if e is run in a state where P holds and it terminates, then the final state will satisfy Q". The predicate P is called the precondition and Q the postcondition. Hoare logic has various rules for proving and combining these specifications.

3.1.1 Separation logic

Separation logic is an extension of Hoare logic that has proven profitable for reasoning about heap-manipulating programs with pointers and concurrency (surprisingly, the same techniques help solve both problems). A good introduction to the basic ideas of separation logic is found in

O'Hearn's "Separation Logic" article [72]. This section gives a more terse overview, especially to introduce the relevant notation.

Separation logic introduces some notation for the logical assertions that describe the heap. The core assertion to talk about pointers is $p \mapsto v$, pronounced "p points to v", which says that the pointer p when dereferenced has value v. The new logical connective of separation logic is $P * Q$, pronounced "P and separately Q", which says that the heap can be divided into two disjoint pieces, one satisfying P and the other satisfying Q. Entailment between propositions is written $P \vdash Q$, read as "P entails Q" or "P proves Q", which says that in any heap where P holds, Q must also hold.

When working in separation logic, specifications like $\{P\}\ e\ \{Q\}$ are generally stated in a "small footprint" style where P mentions only the state e relies on for its execution. This intuition is backed by the celebrated frame rule, which says that if $\{P\}\ e\ \{Q\}$ holds, any disjoint state is unaffected, namely $\{P * F\}\ e\ \{Q * F\}$.

Instead of working with Hoare triples, it is convenient to instead define specifications in a different style of *weakest preconditions* (WPs). We will use $\mathsf{wp}\ e\ \{Q\}$ to denote the weakest precondition of e with postcondition Q; if e is run in a state satisfying $\mathsf{wp}\ e\ \{Q\}$ and terminates, the final state will satisfy Q. Note that the wp is a *predicate over states*, not a judgment of the logic like a Hoare triple. To build intuition, the statement $P \vdash \mathsf{wp}\ e\ \{Q\}$ is equivalent to $\{P\}\ e\ \{Q\}$. An excellent comparison between weakest preconditions and Hoare triples can be found in "Separation logic for sequential programs" [15].

The term "weakest precondition" is because $\mathsf{wp}\ e\ \{Q\}$ is supposed to be the *weakest* predicate that implies Q holds after e's execution, in the sense that any other precondition would imply $\mathsf{wp}\ e\ \{Q\}$, but our work does not emphasize this aspect of weakest preconditions. Furthermore the literature will sometimes distinguish between weakest *liberal* preconditions that only guarantee Q if e terminates and reserve the term weakest preconditions for a predicate that also guarantees termination. This thesis uses the term weakest precondition for the "liberal" version (also called *partial correctness* as opposed to *total correctness*), because proving termination in the presence of concurrency is quite challenging.

Figure 3-1 shows some basic rules of separation logic, phrased in terms of weakest preconditions. As an example, the frame rule becomes WP-FRAME in terms of the wp assertion. Reading this rule forwards, if in a proof the assumptions include F and separately $\mathsf{wp}\ e\ \{Q\}$, then the proof can move F to the postcondition because separation logic guarantees the proof of the WP does not affect or invalidate the part of the heap covered by the frame F.

3.1.2 Ghost state and concurrency in Iris

Concurrent separation logic [6] generalizes separation logic to also reason about concurrency. Iris is a type of concurrent separation logic, with several advances beyond the original formu-

WP-FRAME
$$F * \mathsf{wp}\ e\ \{Q\} \vdash \mathsf{wp}\ e\ \{F * Q\}$$

WP-MONO
$$\frac{P \vdash P' \qquad \forall v.\,([v/x]Q' \vdash [v/x]Q) \qquad \{P'\}\ e\ \{Q'\}}{\{P\}\ e\ \{Q\}}$$

WP-SEQ
$$\frac{\{P\}\ e_1\ \{Q\} \qquad \{Q\}\ e_2\ \{R\}}{\{P\}\ e_1; e_2\ \{R\}}$$

WP-LOAD
$$\{p \mapsto v\}\ \mathrm{Load}(p)\ \{\mathbf{ret}\ v, p \mapsto v\}$$

WP-STORE
$$\{p \mapsto v\}\ \mathrm{Store}(p, v')\ \{p \mapsto v'\}$$

Figure 3-1: Selection of proof rules for sequential separation logic.

lation. A full explanation of the Iris logic is out-of-scope for the thesis; "Iris from the ground up" [50] is a comprehensive introduction while the original "Iris 1.0" paper [48] is a shorter introduction for a reader already familiar with separation logic. Two features of Iris are most relevant since they are used in the GoTxn proof: ghost state and invariants.

A key technique in Iris is to verify a program by augmenting its physical state (local variables and the heap) with some additional *ghost state* which is maintained only for the sake of the proof and has no effect on the program's execution (hence the term "ghost"). It is easier to understand ghost state via its API in Dafny as a programming-language feature, so let us first see how they help there and then return to Iris.

In Dafny, a variable can be marked `ghost`. Ghost variables can be written and read in the proof, but Dafny enforces that the ghost variables' values never influences execution; they can only be used to inform uses of lemmas, assertions, and other proof annotations. Then at run time ghost variables and all uses of them are *erased* before running the program. Why would adding a ghost variable to a program help with its proof? The simplest examples are code where a ghost variable holds the old value of some variable, say prior to a loop; this lets the proof refer to the old value while clarifying that the regular execution does not need it.[1]

Ghost variables can also be used to give abstract specifications to a piece of code. For example, consider a "statistics database" that maintains the running mean of a sequence of numbers

[1] For a concrete example, see the bubble sort example in https://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf.

(written in Go for readability):

```
type StatDB struct {
  count int
  sum   float64
}

func (db *StatDB) Add(n float64) {
  db.count++
  db.sum += n
}

func (db *StatDB) Mean() float64 {
  if db.count == 0 {
    panic("empty db")
  }
  return db.sum/db.count
}
```

The code only tracks the count of elements and their sum, but the behavior of the library is easiest to state in terms of the list of all numbers added. Thus in Dafny such a library can use a ghost variable to track the full database, relate this ghost variable to the physical variables of the code, and then prove that the code returns the correct running mean in terms of the ghost state.

One intuition for the technique of ghost variables is that it augments the execution of the program with additional information, which is used only for the proof and thus not tracked at run time. For every actual execution, there is a corresponding execution where the ghost variables are maintained and updated. The proof is carried out on this augmented execution, but the proofs apply to the normal execution because by design they have the same behavior. Verifying the program with ghost variables is easier because the ghost variables can track important information about the history of the program, such as in the example above of the pre-loop values of the local variables.

In Iris, the proof is a separate entity from the code. The program logic still has a way to use ghost variables, with proof rules that construct and update a ghost variable, applied at the appropriate points in the proof rather than added to the code. The high-level idea for why this works — that there is an augmented execution with the ghost variables — remains the same. In fact in Iris it is more obvious that the ghost variables do not affect program execution, since their creation and updates only appear in the proof and are not added to the code.

So far, we've explained ghost state in terms of ghost variables, with the familiar API where they can be read and written. Iris ghost state is a bit more sophisticated in order to support concurrency reasoning. Iris has separation logic assertions for *ownership* of ghost state, which can be split and divided among threads. In conjunction with this analogy to ownership, in concurrent separation logic a piece of ghost state is also referred to as a *resource* that a thread can

own by having an assertion over the ghost state in its precondition. A key principle in concurrent separation logic is that in a proof about a thread of interest, any resources or ownership in that proof's precondition can never be invalidated by the actions of other threads. Ghost state can also have restrictions on how it may be updated, so that it defines a shared protocol that all threads respect.

A simple example of an interesting type of ghost state is Iris's fractional ghost variables. The assertion $\boxed{\gamma \hookrightarrow_q v}$ says that the ghost variable γ has value v (of any fixed type) and asserts ownership over a (positive) fraction q of it — any fraction $q < 1$ represents read-only access to the ghost variable, while full ownership $q = 1$ allows writing as well. Full ownership $\boxed{\gamma \hookrightarrow_1 v}$ is common enough that it is often abbreviated to $\boxed{\gamma \hookrightarrow v}$, with no fraction. The dashed box around this assertion emphasizes that this assertion is about ghost state and not about the heap, as in the points-to assertion $p \mapsto v$. There are several rules for manipulating and using this fractional ghost state:

FRAC-ALLOC
$$\vdash \Rightarrow \exists \gamma. \boxed{\gamma \hookrightarrow_1 v}$$

FRAC-UPDATE
$$\boxed{\gamma \hookrightarrow_1 v} \vdash \Rightarrow \boxed{\gamma \hookrightarrow_1 v'}$$

FRAC-SPLIT
$$\boxed{\gamma \hookrightarrow_{q_1+q_2} v} \dashv\vdash \boxed{\gamma \hookrightarrow_{q_1} v} * \boxed{\gamma \hookrightarrow_{q_2} v}$$

FRAC-AGREE
$$\boxed{\gamma \hookrightarrow_{q_1} v_1} * \boxed{\gamma \hookrightarrow_{q_2} v_2} \vdash v_1 = v_2$$

UPD-FIRE
$$\frac{P \vdash \mathsf{wp}\ e\ \{Q\}}{\Rightarrow P \vdash \mathsf{wp}\ e\ \{Q\}}$$

The new notation $\Rightarrow$ is an Iris *update modality*. The assertion $\Rightarrow P$ expresses ownership of resources which could be used to become P with some update to ghost state. As an example of proving an update modality, FRAC-ALLOC shows that starting with no assertions it is possible to allocate a new ghost variable γ with value v and complete ownership over it; this is analogous to how the Hoare triple for allocation has no precondition. The formal rule that allows the user to get access to P is UPD-FIRE. It corresponds to advancing the proof of $\mathsf{wp}\ e\ \{Q\}$ by changing whatever ghost state is needed to turn $\Rightarrow P$ into P. As long as the user of the logic is proving a weakest precondition as the goal, they can apply this rule to "eliminate" an update modality.

Fractional ghost state can be updated after creation with FRAC-UPDATE. The update requires full ownership. Fractional ghost variables can instead be split into smaller pieces with FRAC-SPLIT; two assertions for the same ghost variable must be for equal values (FRAC-AGREE) and the pieces cannot be updated, since the variable has only one underlying value. Fractional ownership expresses a simple protocol among threads where when a thread owns a fraction less than one, it can read but not write, and full ownership is sufficient to write to a ghost variable.

This thesis describes a few constructions for ghost state to carry out parts of the GoTxn proof, such as the example of fractional ghost state described above. In reality all ghost state in Iris is defined using a single, general mechanism. Ghost state can come from any instance of an

algebraic structure M called a "resource algebra", where ownership really means ownership of an element $a \in M$. This thesis does not explain the details of how ghost state is constructed using resource algebras — see "Iris from the ground up" [50]. For the ghost state in this thesis, we will only give the API, in terms of resources and rules that allow updating those resources. The Iris logic ensures that the updates are "sound", enforcing a global property that updates to a resource in one part of the proof never invalidate resources owned by concurrent threads at the same time.

$$\text{WP-INV-ALLOC}\quad \frac{P * \boxed{R} \vdash \mathsf{wp}\ e\ \{Q\}}{P * R \vdash \mathsf{wp}\ e\ \{Q\}} \qquad \text{INV-ATOMIC}\quad \frac{\mathsf{atomic}(e) \qquad R * P \vdash \mathsf{wp}\ e\ \{R * Q\}}{\boxed{R} * P \vdash \mathsf{wp}\ e\ \{Q\}}$$

$$\text{WP-FORK}\quad \frac{P \vdash \mathsf{wp}\ e\ \{\mathsf{True}\} \qquad Q \vdash \mathsf{wp}\ e'\ \{R\}}{P * Q \vdash \mathsf{wp}\ (\mathsf{fork}\{e\};\ e')\ \{R\}}$$

Figure 3-2: Key concurrency rules in Iris for invariants and forking

A fundamental reasoning principle for concurrency is the notion of an *invariant*. Threads eventually do share state, and invariants are the main way to reason about how threads coordinate on that shared state. The assertion $\boxed{I}$ expresses the knowledge that I is an invariant. Once this invariant is established, the proof rules in Iris guarantee that I holds at all steps of the program. A thread that has $\boxed{I}$ in its precondition can make use of the invariant by "opening" it to obtain ownership over I, but only for a single program step; it must be returned afterward to guarantee the invariant holds for other threads. Finally, invariants are freely *duplicable* — that is, $\boxed{I} \vdash \boxed{I} * \boxed{I}$ — reflecting that knowledge of an invariant, once it is established, is an assertion that all threads agree on which cannot be invalidated.

The formal rules for using invariants in Iris (which are all valid rules in Perennial) are given in fig. 3-2. To create an invariant $\boxed{R}$, a thread gives up R, as given in WP-INV-ALLOC. The rule for using an invariant is INV-ATOMIC, which can only be used over an "atomic" instruction e. Perennial is defined for a general language, as in Iris, and the semantics of the language defines what is modeled to be atomic. Formally $\mathsf{atomic}(e)$ says that the expression e reduces to a value in a single reduction step, so that other threads cannot run in between. See section 7.4.2 for the details on the specifics of what GooseLang defines to be atomic.

Ownership transfer is reflected in the rule for forking new threads, WP-FORK. If a thread has $P * Q$ in its precondition, it can pass some of those resources P to a newly-forked thread and retain the remainder Q for the subsequent code. Note that due to the separating conjunction these resources must be separate, so that the rules of the separation logic guarantee e cannot invalidate Q using its resources P. However, invariants give a way for the two threads to safely

share resources: both can have access to $\boxed{I}$ because this is merely *knowledge of the invariant* and can be duplicated, which is sound because each thread only uses I for an atomic step and then guarantees I holds afterward.

3.2 Crash weakest preconditions

Iris gives tools for proving specifications that capture the concurrency behavior of a program, but storage systems need stronger specifications that also cover crash safety. The formal definition of crash safety is ultimately stated as a property of a storage system combined with a recovery procedure. Crash safety is *defined* in terms of the results possible after a program crashes mid-operation, the system reboots, and subsequently a recovery procedure re-initializes the program.

In Perennial, most of the reasoning required for this recovery-based definition goes into specifying what happens if the system halts at some intermediate point and all threads stop running. The core specification idea for reasoning about this situation is a *crash weakest precondition* $\mathsf{wpc}\ e\ \{Q\}\ \{Q_c\}$. Similar to the weakest precondition, if e is run from a state satisfying this predicate and terminates, the resulting state will satisfy Q. However, in addition if the system halts at any time Q_c is guaranteed to hold. We also sometimes write a *crash Hoare quadruple* $\{P\}\ e\ \{Q\}\{Q_c\}$ that is defined to be $P \vdash \mathsf{wpc}\ e\ \{Q\}\ \{Q_c\}$. There is one subtlety in the terminology: though we use the term "crash," these postconditions hold *just prior* to the system actually shutting down, so that the contents of memory is unchanged.[2] Section 3.4 connects these crash specifications to the memory wipe and reboot that happens immediately afterward.

Some basic structural rules for these crash weakest preconditions are given in fig. 3-3. These largely mirror structural rules from Iris and do not say anything crash-specific. The rule WPC-MONO allows weakening a WPC by replacing the postcondition Q with a weaker assertion P and replacing the crash condition Q_c with a weaker assertion P_c. The rule WPC-LET is a bit long, but it expresses formally sequentially reasoning about e_1 followed by e_2. What it expresses is that to verify $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, a proof can verify e_1 first, then in its post-condition reason about e_2 with the return value v from e_1 substituted for the bound variable x (denoted $e_2[v/x]$). The crash condition P_c is carried throughout, since both e_1 and e_2 must maintain it. Perennial has a similar but more general rule for arbitrary *evaluation contexts* (of which let is just an example), to reason about sequencing between a sub-expression and its context.

Some more interesting structural rules for wpc with sequential code are listed in fig. 3-4. One such rule that is often used in Perennial is the crash frame rule, WPC-FRAME. Like the traditional frame rule, this is a reasoning principle for ignoring some resources while proving part of a program. When reasoning about crashes, framing is a useful way to dismiss the crash

[2] They might more properly be called *halt conditions*, but the original FSCQ paper [17] used "crash conditions" and the term has stuck since then.

WPC-VALUE
$$P_c \wedge [v/x]P \vdash \mathsf{wpc}\ v\ \{\mathbf{ret}\ x, P\}\ \{P_c\}$$

WPC-MONO
$$\frac{\forall v.\ ([v/x]P \vdash [v/x]Q) \qquad P_c \vdash Q_c}{\mathsf{wpc}\ e\ \{\mathbf{ret}\ x, P\}\ \{P_c\} \vdash \mathsf{wpc}\ e\ \{\mathbf{ret}\ x, Q\}\ \{Q_c\}}$$

WPC-LET
$$\mathsf{wpc}\ e_1\ \{\mathbf{ret}\ v, \mathsf{wpc}\ e_2[v/x]\ \{P\}\ \{P_c\}\}\ \{P_c\} \vdash \mathsf{wpc}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \{P\}\ \{P_c\}$$

Figure 3-3: Basic structural rules for crash weakest preconditions.

WPC-FRAME
$$Q * \mathsf{wpc}\ e\ \{P\}\ \{P_c\} \vdash \mathsf{wpc}\ e\ \{Q * P\}\ \{Q * P_c\}$$

WP-WPC
$$\mathsf{wp}\ e\ \{\mathbf{ret}\ x, P\} \dashv\vdash \mathsf{wpc}\ e\ \{\mathbf{ret}\ x, P\}\ \{\mathrm{True}\}$$

Figure 3-4: Interesting crash-related structural rules for crash weakest preconditions.

condition when it refers to durable resources that aren't needed for reasoning about some part of the code.

The way the rule works is that in analogy to WP-FRAME, the premise is a proof of Q (the frame) and separately $\mathsf{wpc}\ e\ \{P\}\ \{P_c\}$. However, in addition to framing from the postcondition, Perennial also frames from the crash condition. A common case is where $P_c = \mathrm{True}$, which is useful when e is a purely in-memory piece of code. In that case a proof can combine framing and WP-WPC to reason about part of a crash Hoare quadruple using crash-free reasoning, by temporarily ignoring the durable resources P_d in the precondition, using a derived rule:

$$\frac{\{P\}\ e_1\ \{Q\} \qquad \{Q * P_d\}\ e_2\ \{R\}\{Q_c\}}{\{P * P_d\}\ e_1; e_2\ \{R\}\{Q_c\}}$$

Another example of combining crash and crash-free reasoning with both wpc and wp is in the WPC-ATOMIC rule. The precondition $\mathsf{atomic}(e)$ says this rule only applies to atomic expressions, which take a single step, and the conclusion is a wpc for this expression. The premise involves a connective $P \wedge Q$. This is a *non-separating conjunction* or "logical and". $P \wedge Q$ holds in some state when P and Q both hold, but unlike $P * Q$ they do not have to be over disjoint parts of the state, so for example $p \mapsto v \vdash p \mapsto v \wedge p \mapsto v$ is trivially true.

The logical "and" is important in this rule. If at some point at the proof we have resources R and want to prove $\mathsf{wpc}\ e\ \{P\}\ \{P_c\}$, the rule says it is sufficient to prove two things: $R \vdash P_c$ and $R \vdash \mathsf{wp}\ e\ \{P_c \wedge P\}$. Observe that this rule reduces crash reasoning to non-crash reasoning. Also notice that unlike most separation logic reasoning, we don't need to split up R to prove these two entailments; the full contents of R are available in both, which would not be the case if the rule had a separating conjunction.

There are two reasons why this reasoning is sound: first, e takes only a single step, and the

WPC-INV-ALLOC
$$\frac{P * \boxed{R} \vdash \mathsf{wpc}\ e\ \{Q\}\ \{Q_c\}}{P * R \vdash \mathsf{wpc}\ e\ \{Q\}\ \{Q_c * R\}}$$

WPC-ATOMIC
$$\frac{\mathsf{atomic}(e)}{P_c \wedge \mathsf{wp}\ e\ \{\mathbf{ret}\ x,\ P_c \wedge P\} \vdash \mathsf{wpc}\ e\ \{\mathbf{ret}\ x,\ P\}\ \{P_c\}}$$

Figure 3-5: Rules for reasoning about concurrency and crashes.

system either crashes or not, but not both, so the proof only needs to show either the crash or post condition. If the system crashes before e then $R \vdash P_c$ shows the crash condition holds now, and otherwise the resources R are still available to prove $\mathsf{wp}\ e\ \{P_c \wedge P\}$. That proof shows that the post-crash resources satisfy both P_c (if the system crashes right after e) and P (to show the postcondition).

To reason about concurrency, Perennial needs some more principles for sharing ownership between threads. The core mechanism in concurrent separation logic for reasoning about concurrency is the invariant, as explained earlier. The rules for working with invariants in a non-crash setting are given in fig. 3-2: WP-INV-ALLOC gives up R in exchange for $\boxed{R}$, and INV-ATOMIC "opens" an invariant for a single atomic step. $\boxed{R}$ is duplicable — that is, $\boxed{R} \vdash \boxed{R} * \boxed{R}$.

Invariants have a special role in Perennial. Figure 3-5 lists Perennial's rules for concurrency reasoning. Perennial extends invariant allocation with a rule WPC-INV-ALLOC. The non-crash parts of this rule are identical to WP-INV-ALLOC, but applying the rule has the additional benefit of *removing R from the crash condition*. The intuitive reason this is sound is that since threads maintain R at all intermediate steps, it must also hold in case the system crashes, and the thread that created the invariant can get "credit" for this on crash. After allocating an invariant it has no special role for crashes, and proofs can use INV-ATOMIC as usual for opening an invariant across an atomic step.

Invariants are especially useful for lock-free reasoning, but concurrent code commonly coordinates between threads using locks. A lock guarantees that no two threads own the lock at the same time. This is expressed in separation logic by associating a *lock invariant* P with each lock when it is created, which we think of as something the lock "protects". When a thread acquires the lock, it obtains ownership of P, and when it releases the lock, it gives up the same ownership. This is sound precisely because either a single thread holds the lock, or it is free (in which case intuitively we can think of the lock as holding ownership).

What happens if the system crashes while a lock is held? The standard lock specification gives a thread exclusive access to P during a critical section (between $\mathsf{acquire}(\ell)$ and $\mathsf{release}(\ell)$), and says nothing about the locked state if the system crashes in the middle of a critical section — this is fine if the lock protects in-memory state that is anyway lost on crash, but insufficient for reasoning about locks that protect durable state. One intuition for what goes wrong is that a crash "steals" ownership of the locked state and forcibly transfers it to recovery, which is a

WP-LOCK-ALLOC
$$P \vdash \mathsf{wp}\ \mathsf{newlock}()\ \{\mathbf{ret}\ \ell,\ \mathsf{isLock}(\ell, P)\}$$

WP-LOCK-USE
$$\frac{P * R \vdash \mathsf{wp}\ e\ \{P * Q\}}{\mathsf{isLock}(\ell, P) * R \vdash \mathsf{wp}\ (\mathsf{acquire}(\ell);\ e;\ \mathsf{release}(\ell))\ \{Q\}}$$

WPC-LOCK-ALLOC
$$\frac{P \vdash P_c}{P \vdash \mathsf{wpc}\ \mathsf{newlock}()\ \{\mathbf{ret}\ \ell,\ \mathsf{isCrashLock}(\ell, P, P_c)\}\ \{P_c\}}$$

WPC-LOCK-USE
$$\frac{P * R \vdash \mathsf{wpc}\ e\ \{P * Q\}\ \{P_c\}}{\mathsf{isCrashLock}(\ell, P, P_c) * R \vdash \mathsf{wp}\ (\mathsf{acquire}(\ell);\ e;\ \mathsf{release}(\ell))\ \{Q\}}$$

Figure 3-6: Rules for reasoning about concurrency and crashes with locks.

possibility the proof of a concurrent, crash-safe system needs to reason about.

Perennial addresses this issue by introducing a new crash-aware lock specification that also gives guarantees if the system crashes during a critical section. Compared to the regular lock spec (WP-LOCK-ALLOC and WP-LOCK-USE), the crash-aware specification (WPC-LOCK-ALLOC and WPC-LOCK-USE) associates both a regular lock invariant P with the lock and a crash condition P_c. The rule for allocating a lock dismisses P_c from the caller's crash condition, similar to allocating an invariant, and intuitively the reason why this rule is sound is that when the lock is used with WPC-LOCK-USE the caller is obliged to prove P_c holds throughout the entire critical section (whereas the stronger lock invariant P only needs to be restored at the end).

3.3 Crash model

Perennial has to model a system crash and reboot. A crash is represented as another transition for a program to take, one which wipes in-memory state. The reboot is modeled by running a dedicated procedure that restores the storage system on boot, which we call a *recovery procedure*. So far, we have used Perennial for reasoning about systems that interact with a disk, a device with an interface for reading and writing blocks (assumed to be 4KB in our development). These disks might be a block device like `/dev/sda1` in Linux, or can be a file hosted in another file system (such as tmpfs for an in-memory disk).

A crash transition resets the heap to empty while preserving the disk state. Encoding the effect on the heap in separation logic is a bit tricky, since it would appear that assertions about in-memory pointers like $\ell \mapsto v$ need to be invalidated. The logic handles this by parameterizing all heap assertions and weakest preconditions with a *generation number*. Across a crash, the generation number is incremented, so that old assertions are true but no longer work for

loads and stores since they do not apply to the current heap. A class of assertions $\mathsf{durable}(P)$ only concern durable state like the disk, which in the logic means that P is independent of the current generation number. These assertions are special in that they can be proven as the crash condition of the system and then used in the precondition of recovery, as described subsequently in section 3.4.

The Perennial logic, like the underlying Iris framework, is parameterized over a language and its semantics, which defines the crash transition. The language we generally work with is GooseLang, described in detail in chapter 7. Our proofs so far with GooseLang use a simple crash model, with the disk unchanged on crash. This corresponds to a synchronous semantics, since it means that a disk write is immediately considered durable when it returns. Perennial also supports defining an asynchronous disk by modeling buffered writes and then non-deterministically choosing which buffered write is persistent at crash time, similar to the model in FSCQ [17]. Because the crash transition is defined by the user, it is not necessary in Perennial to strictly partition the state into ephemeral and durable state.

3.4 Recovery reasoning

To allow the user to reason about the recovery process, Perennial has a *recovery* weakest precondition $\mathsf{wpr}\ e\ \circlearrowleft\ e_r\ \{P\}\ \{P_r\}$, where e represents the storage system, e_r is a recovery procedure that will run on restart, P is the postcondition for normal execution, and P_r is the *recovery postcondition*. When e is run in a state satisfying this wpr, if it terminates normally then P holds, and if the system crashes and e_r terminates then P_r holds. The latter is true even if the system repeatedly crashes and restarts while running e_r.

Perennial has only one rule to prove a wpr, which reduces it to proving a wpc about e and an *idempotent* specification for e_r, namely one where the crash condition implies the precondition:

$$\frac{\mathsf{durable}(P_c) \qquad P_c \vdash \mathsf{wpc}\ e_r\ \{P_r\}\ \{P_c\}}{\mathsf{wpc}\ e\ \{P\}\ \{P_c\} \vdash \mathsf{wpr}\ e\ \circlearrowleft\ e_r\ \{P\}\ \{P_r\}}\ \text{WPR-IDEMPOTENCE}$$

At a high level, the user proves $\mathsf{wpr}\ e\ \circlearrowleft\ e_r\ \{P\}\ \{P_r\}$ in two steps:

1. First, prove $\mathsf{wpc}\ e\ \{P\}\ \{P_c\}$ to cover normal termination and establish a global crash invariant P_c for the program.

2. Second, prove $P_c \vdash \mathsf{wpc}\ e_r\ \{P_r\}\ \{P_c\}$ to establish the recovery postcondition from the crash condition and show that recovery maintains the crash invariant so that crashes during recovery are also handled.

3. Finally, prove durable(P_c), which asserts that the crash invariant is stated using only *durable* resources that survive a crash.

The final step, the durability side condition, is where the proof takes into account the effect of a crash. For example wpc e $\{P\}$ $\{P_c\}$ proves that P_c holds just prior to a crash while $P_c \vdash$ wpc e_r $\{P_r\}$ $\{P_c\}$ starts reasoning just after the crash. Durability is defined so that if P_c is durable, it holds across a crash.

In practice a user of the logic proves a wpr assertion about an expression e that is a server loop that accepts operations, executes them, and replies. The same loop e is also e_r since it restores its state from disk. Finally, the user will separately prove that e is safe to run from an initial state with an all-zero disk.

The recovery code sets up the global invariants and appropriate crash locks for the whole system and thus "cancels" all of these assertions from its crash condition (formally, these assertions are guaranteed at crash time by the invariants or crash locks, and the rest of the recovery proof no longer needs to prove them as part of the crash condition). The separating conjunction of all of these invariants will be P_c. Then the proof rules in Perennial guarantee that recovery can assume P_c holds if the system crashes and reboots at any time, as given by the idempotence rule. Chapter 4 gives more detail on how systems are specified in Perennial.

3.5 Soundness

Above we presented an intuitive meaning for $P \vdash$ wpr e $\circlearrowleft$ e_r $\{Q\}$ $\{Q_c\}$, but it is not obvious that the definitions of wpr and wpc imply that this intuitive meaning holds up. Interpreting the mechanisms of the logic is complicated since it requires relating logical features like ghost state, invariants, and ownership to the execution of the program. To address these concerns, Perennial comes with a *soundness theorem* that formally connects the definitions to a guarantee about e and e_r independent of the details of the logic. The soundness theorem establishes what an end-to-end theorem about the system *means* without trusting the interpretation and implementation of these features.

Most of Perennial makes no assumptions about the programming language or state, but the soundness theorem has one aspect specific to verifying code running on a disk. It uses the assertion $a \mapsto_{\mathsf{d}} v$, a "disk points-to" assertion that says the disk has value v at address a, analogous to $p \mapsto v$. The basic version of the soundness theorem says:

Theorem 3.1 (Perennial soundness)**.** Let $d \in \mathit{Disk}$ be a disk. Let ϕ and ϕ_r be predicates over values. Suppose that

$$\ast_{a \mapsto v \in d} \; a \mapsto_{\mathsf{d}} v \vdash \mathsf{wpr}\ e \circlearrowleft e_r\ \{\mathbf{ret}\ x,\ \phi(x)\}\ \{\mathbf{ret}\ y,\ \phi_r(y)\}$$

is derivable in Perennial. Running e (recovering with e_r) in a state σ with an empty heap and disk d will not get stuck, and if the execution terminates with an expression e' in state σ' (possibly with additional forked threads), then

1. If this execution is a normal termination with the value e', then $\phi(e')$ holds.
2. If this execution is a crash and recovery execution and e' is a value produced by e_r, then $\phi_r(e')$ holds.
3. Any forked threads are either a value or are reducible in σ'.

To satisfy the premise of the soundness theorem, the user gets to assume points-to facts for all of the addresses in the initial disk (the assertion $\ast_{a \mapsto v \in d} a \mapsto_{\mathsf{d}} v$). In return the theorem has several postconditions. First, the theorem promises that e runs "safely" in the sense of never encountering an irreducible expression; the semantics uses irreducibility ("stuckness") to indicate an expression that has hit an error. This immediately rules out a number of errors, including any calls to Go's built-in `panic()` function and lower-level issues like reading slices out-of-bounds and data races. Second, the properties (1) and (2) state that the normal postcondition and recovery postcondition have the intended meaning for the return value of the main thread e or the recovery procedure e_r, respectively. Finally, property (3) is a safety statement that says forked threads also do not encounter errors.

This soundness theorem is machine-checked in the Perennial implementation. Its proof hides complexity in the definitions of wpc and wpr, and in the definitions of all the Perennial mechanisms using lower-level Iris primitives.

The reader might observe that this theorem is about the return value of an expression, but it doesn't say much about intermediate results of the program. For a system like a server, which has no interesting return value, the main outcome of this theorem is the safety statement. It is possible to go beyond this soundness theorem to a *refinement* soundness theorem that specifies a program by relating its observable I/O behavior to a simpler, more abstract program. We used this more general support to prove a theorem for GoTxn that relates an arbitrary program using the transaction system to a simpler specification program. Both the simple soundness theorem above and the refinement soundness theorem are proven using lower-level mechanisms from Iris and Perennial.

3.6 Implementation

Perennial is implemented on top of Iris in about 25,000 lines of code (a breakdown is given in fig. 3-7). The Iris logic is divided into two parts: a "base logic" which defines a notion of resources, ownership, and separation logic, and on top of this base logic a *program logic* that

Component	**Lines of Coq**
Helper libraries (maps, lifting, tactics)	6,292
Ghost state and resources	6,585
Program logic for crashes	12,527
Perennial total	25,404

Figure 3-7: Lines of code (including proofs) for Perennial.

defines the weakest precondition wp e $\{Q\}$ as an assertion in the base logic about the execution of e. All of the proof rules in the program logic are defined as theorems in the base logic, using the definition of wp.

Perennial re-uses the Iris base logic with no changes, and builds a custom program logic on top with wpc e $\{Q\}$ $\{Q_c\}$ and wp e $\{Q\}$. The non-crash part of these definitions are similar to Iris, but the crash aspects required significant implementation work. This thesis aims only to explain how the logic is used, at a high level of abstraction, while the code documents exactly how the logic is implemented in the Iris base logic (for example, the definition of Perennial's wpc).

One aspect of the implementation that should be noted is that the rules as presented in this thesis have been simplified for exposition and are not sound as written. The real theorem statements require Iris features like the later modality, invariant namespaces, and masks to be sound. The later modality addresses issues with circularity that arise from Iris's flexible ghost state. Namespaces and masks are used to prevent re-entrancy where an invariant is opened twice by the same thread, which would be unsound. The on-paper presentation still aims to capture the essence, but the code is the source of truth for the Perennial logic. The soundness proof ensures that the code's rules are sound, and Coq checks that these details and their side conditions are all written correctly.

Iris comes with a proof mode called MoSeL [58, 59] for interactive proofs. The proof mode supports proving separation logic theorems and proving programs correct interactively. The style of proof highly resembles the Coq proof mode, but extended for reasoning in separation logic. The proof mode is highly extensible, and we used these features to integrate support for MoSeL with Perennial.

There are two interesting aspects in the Perennial MoSeL support that we developed. The first is *named propositions*, a general feature for interactive separation logic proofs that arose out of our experience writing and changing large invariants. The second is tactics specific to the crash reasoning in Perennial: *proof caching* reduces the burden of crash-safety reasoning where the crash condition must often be re-proven at intermediate steps of the proof, and *crash framing* simplifies the mechanics of framing away resources that are needed for the crash condition but

not being used otherwise.

3.6.1 Named propositions in separation logic

To prove a large system requires writing down many invariants (one for each layer of the system) and then proving these invariants are preserved by the code. Coming up with these invariants is a challenging part of the proof, and requires frequent cycles of finding an invariant doesn't work, editing it, and then revising the proofs. The proof will typically need to *destruct* the invariant, a proof step which breaks an assumption into separate hypotheses for each conjunct. For example if an invariant `inv := P * Q * R` appears as hypothesis "H" in the proof, the next proof step would be `iDestruct "H" as "(HA & HB & HC)"` which would produce hypotheses "HA", "HB", and "HC" for the propositions `P`, `Q`, and `R` respectively. The problem is that updating this proof step when the invariant changes is tedious and error-prone. For example, adding `* S` to the end of the invariant will mean "HC" is now `R * S` rather than `R`, and the situation is worse if a conjunct is added to the middle of the invariant.

Named propositions help solve this problem. With this feature, the invariant is written combining the logical statement with names for each conjunct:

```
Definition inv := "HA" :: P *
                  "HB" :: Q *
                  "HC" :: R.
```

The notation `"HA" :: P` means the exact same thing as `P`, logically, but named propositions extend the proof mode with the ability to destruct a hypothesis and automatically name its conjuncts. This has two advantages: first, we don't repeat the names for the conjuncts throughout the proof, and second, extensions to the invariant are minimally invasive to the proof (even if conjuncts are reordered) since each name always refers to the same thing it did before.

Named conjuncts made a huge difference to proof burden — many individual definitions have 5–6 conjuncts, and across the entire GoTxn proof we name some 900 total conjuncts. Implementing them was actually quite simple, taking only about 400 lines of code in Coq on top of the Iris Proof Mode. The implementation is separate from Perennial, depending only on Iris.[3]

3.6.2 Perennial-specific interactive proofs

Caching: Crash-safety reasoning involves repeatedly showing that the crash condition holds at each intermediate point. Writing local specifications (by "framing away" anything not needed) helps reduce this burden, but does not fully eliminate it. Perennial includes a tactic for caching

[3]The code is available at https://github.com/tchajed/iris-named-props.

and reusing proofs of the crash condition. A proof engineer proves the crash condition as it stands at some point, using as few assumptions as possible for locality, and the caching infrastructure saves the proof.[4] Whenever the crash condition appears later in the proof with the same assumptions available, the caching infrastructure proves the goal automatically.

Framing: Recall the WPC-FRAME rule for proving a wpc using a wp when no durable resources are required for a sub-part of the proof. This is commonly used, so Perennial has a `wpc_frame` tactic to apply the theorem. The tactic takes a list of hypotheses and applies WPC-FRAME, leaving the user to prove that the listed assumptions together imply the crash condition, with the remaining hypotheses available to prove a wp for the original postcondition. In addition, the hypotheses used for the crash condition are restored after proving the wp. This support integrates with the caching support, automatically using a cached proof to prove the crash condition if possible.

3.7 Limitations

Perennial has some limitations. The logic can prove *safety* properties ("nothing bad happens") but not *liveness* ("something good eventually happens"), a limitation inherited from Iris and in fact due to deeper theoretical reasons related to step indexing — Transfinite Iris [88] is an approach to solve this issue, and the paper explains the difficulties well. Due to different technical reasons related to step indexing, it is not currently possible to put weakest precondition assertions inside crash conditions; such a feature is useful for reasoning about procedures stored on disk, something that shows up in practice with logical logging [68].

The current version of Perennial does not support *recovery helping* from the original version [12] (although it has many new features), a reasoning principle where in a simulation proof recovery logically completes an operation from before the crash. As described here, operations must be simulated during the crash. It also does not implement Iris's support for prophecy variables [51], though we believe this is doable — an interesting extension to work out would be prophecies that predict a crash, or prophecies over the behavior of a future recovery execution.

3.8 Takeaways beyond formal verification

Crash-aware locks. One of the benefits of concurrent separation logic that is useful even without verification is that it crystallizes the idea of a "lock invariant" as a way to think about the *purpose* of locks. Rust's `sync::Mutex` API implements a basic version of the lock invariant idea by allowing the programmer to at least express what data is protected by the lock, but the idea of

[4] For readers familiar with Iris, the proof is a persistent implication that can be reused as many times as needed.

a separation logic invariant is more general (for example, locks can protect just a part of a data structure). Perennial's crash-aware lock specification formalizes an idea that is not well-known, of how locking lock invariants interact with durable state: such a locks should have both the usual lock invariant and a weaker crash invariant.

Invariants for crash reasoning. Perennial has a rule where allocating an invariant gives "credit" for it on crash. This idea formalizes an intuition about how ownership flows between threads and on crash. A system called RECIPE has an insight along these lines, observing that a lock-free data structure should work on persistent memory [60]. Intuitively this is true because the code already maintains an invariant at all intermediate steps due to other threads, and thus the same invariant should hold of the persistent memory on crash, but some more conditions are required for the intuition to hold up. RECIPE has no proof of correctness, and a mechanized proof would be challenging, but the ideas in this thesis could be used to think about its correctness and the required conditions with greater precision.

Chapter 4

Logically atomic crash specifications in Perennial

On top of the Perennial logic, we developed a specification pattern to organize proofs of libraries and compose them. Of particular interest is a pattern for *logically-atomic crash specifications*, which capture that a set of methods in a library appear atomic with respect to both other threads and on crash. In order to illustrate how all of these techniques work together, this section focuses on the specification and proof intuition for the circular buffer, a library used in GoTxn that has interesting crash safety and concurrency behavior. One feature of this specification style, the idea of an *exchanger*, only appears in the journaling code which is much higher in the software stack for GoTxn, so section 4.5 does discuss this higher-level specification. The circular buffer is relatively independent of other details in GoTxn, since it operates directly on top of the disk, but journaling is covered in more detail within the GoTxn chapter, in section 5.5.1.

This chapter builds upon the Perennial background from chapter 3, and the note about audience from that chapter's introduction is relevant here as well — a reader interested in the ideas in GoTxn, DaisyNFS, or Goose can safely skip to the relevant chapters.

4.1 Circular buffer API

The circular buffer is a fixed-size, *on-disk* queue of updates, which are pairs of an address and a disk block. It is used to implement write-ahead logging: updates are first stored in the circular buffer, and then eventually moved to a separate data region. It supports two operations during normal usage: `Append(end, upds)` appends a list of updates to the buffer and `TrimTill(pos)` deletes updates up to the position `pos`. These logical queue positions grow indefinitely, but the buffer can hold a finite and fixed number of update at a time and it is the caller's responsibility to avoid overflowing with `Append`. The only read operation is recovery, which restores the

durable updates and current queue position; the write-ahead log caches updates during normal execution, but this is not the circular buffer's concern. The complete Go API is shown in fig. 4-1.

```
// Circular buffer Go API

func InitCircular() *Appender
func (c *Appender) Append(end uint64, upds []Update)
func TrimTill(start uint64)
func RecoverCircular() (c *Appender, start uint64, upds []Update)

type Update {
  Addr  uint64
  Block []byte
}
```

Figure 4-1: The Go API for the circular buffer.

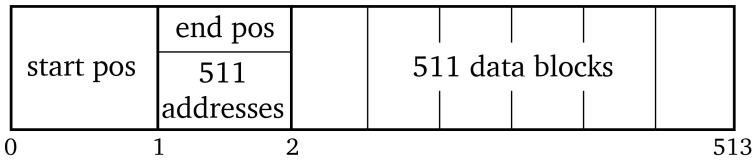


Figure 4-2: The circular buffer's on-disk layout, consisting of a fixed 513 blocks located at the beginning of the disk.

The basic idea is that the circular buffer is specified in terms of an abstract state that transitions for each operation. The recovery operation restores the pre-crash state of the queue from disk. The state for this library consists of two components: a queue of updates (which are pairs of an address to write to and a 4KB block to write there) and a 64-bit start position that indicates how many updates occurred prior to the current list of updates. The `Append` operation, as mentioned, simply appends to the list of updates. The argument to `TrimTill` is an absolute position; the effect of `TrimTill(pos)` is to delete `pos - start` updates from the beginning of the list of updates and set the new start position to `pos`. A more formal specification is given in fig. 4-3.

The circular buffer supports atomic and concurrent `Append` and `TrimTill` with a careful on-disk layout. It consists of two header blocks and 511 data blocks, depicted graphically in fig. 4-2. The first header block (the "trim" block, since it is used for trimming) holds the logical start position of the queue, while the second header (the "append" block, since it is used for appending) holds the logical end position and 511 addresses; the data blocks for these addresses are in the remaining data blocks. This append header has 4096 bytes, or enough space for 512 64-bit numbers; one is used for the end position, and the remainder for update addresses.

Recovery uses the header blocks to determine what subset of the circular buffer holds complete multiwrites; partially-appended updates are ignored for atomicity.

To append atomically, `c.Append(end, upds)` first writes the data (using `end` to know where to start writing), and then writes the append header block to atomically incorporate the writes, and simultaneously to write the addresses. The in-memory `c *Appender` struct keeps track of the existing addresses so they can be re-written to disk. Meanwhile `TrimTill` logically deletes by merely writing a new, higher start position to the trim header.

Since these operations touch disjoint parts of disk, they can be performed concurrently. However, one subtlety is that it is important to preserve the queue abstraction that append operations do not overflow the circular buffer (since this would overwrite old values) and that trim operations do not delete past the current end (this would invalidate future appends immediately). The insight that allows the caller to guarantee these properties is that the start and end positions of the circular buffer are monotonically increasing, meaning any snapshot of their positions is a *lower bound* that is valid from then on. Thus without holding any locks a call to `c.TrimTill(newStart)` can guarantee that the trim fits by showing that `newStart` $\leq$ `end_lb`, where `end_lb` is merely a lower bound and not necessarily the exact end position. However, it is necessary for the caller to know the exact start position and that it will not change concurrently, in order to guarantee that `newStart` is greater than the current `start`. The proof of `c.Append(end, upds)` is the dual of trim, requiring a lower bound on the start position and exact control over `end`.

```
(* Circular buffer Coq specification *)

Definition update := u64 * Block;
Record circ_state := {
  upds: list update;
  start: u64;
}.

(* requires length s.(upds) + length new_upds <= 511 *)
Definition append (s: circ_state) (new_upds: list update) : circ_state :=
  s <[ upds := s.(upds) ++ new_upds ]>.

(* requires s.(start) <= new_start *)
Definition trim_till (s: circ_state) (new_start: u64) : circ_state :=
  let num_removed := new_start - s.(start) in
  s <[ start := new_start ]>
    <[ upds := drop num_removed s.(upds) ]>.
```

Figure 4-3: The specification for the circular buffer, described as transitions on the abstract state.

There are three challenges in specifying how the circular buffer's implementation connects to the abstract transitions given above. First, we want to show that `Append` appears to be atomic to the caller, even though its implementation writes many disk blocks and the system could crash after persisting only some of them. Second, the library has some thread-safety requirements to enforce on the caller: in particular `Append` and `TrimTill` can be called concurrently from separate threads, but it is not safe to concurrently issue multiple appends or multiple trim requests. Finally, the library relies on the caller to guarantee that `Append` and `TrimTill` are called with enough available space. These last two challenges are due to leaving concurrency control to the caller, which is more efficient than the circular buffer having its own redundant locking.

The write-ahead log uses the circular buffer from two threads, one dedicated to *logging* which appends to the circular buffer to make writes durable and another dedicated to *installation* which trims writes from the circular buffer after installing them. Appending and trimming can be performed concurrently, but appending requires the `*Appender` struct which is not thread-safe. The caller only reads from the circular buffer if the system crashes, at which point recovery restores the previous state.

4.2 Circular buffer custom resources

Perennial, like Iris, has support for user-defined separation-logic resources. These are defined using lower-level mechanisms, but this section simply presents the resources and the intuition for how they are used without going into details on how they are implemented. The circular buffer defines and issues the following custom separation-logic resources:

- $\texttt{circ\_state}(\gamma, \sigma)$, which says that the current state of the circular buffer is σ. The γ argument is a "ghost name" for the ghost state associated with this instance of the circular buffer; at any time there is only one circular buffer, but this name will change every time the system crashes and reboots and thus distinguishes different "generations" of the circular buffer.

- $\texttt{is\_circular}(\gamma)$, defined to be the invariant assertion $\boxed{\exists \sigma, \texttt{circ\_state}(\gamma, \sigma) * P(\sigma)}$. The $P(\sigma)$ part of this invariant is part of the HOCAP specification style, explained below.

- $\texttt{circ\_appender}(\gamma, \ell)$. This relates the circular buffer's ghost state with ℓ, a `*circularAppender` pointer that has some data needed to append to the circular buffer. This resource is not persistent and is owned by the logger thread.

- $\texttt{start\_is}(\gamma, \texttt{start})$ and $\texttt{end\_is}(\gamma, \texttt{end})$ give the exact current positions of the start and end (that is, $\sigma.\texttt{start} + \texttt{length}(\sigma.\texttt{updates})$) of the circular buffer. These resources are not persistent and are owned by the installer and logger threads respectively.

- `start_at_least`$(\gamma, \texttt{start})$ and `end_at_least`$(\gamma, \texttt{end})$ give *lower bounds* on the start and end positions. It is possible to issue such resources because the start and end monotonically increase. As a result of monotonicity, these resources are persistent; once `start` or `end` reaches some value, that value is always a valid lower bound.

4.3 Specifications for `Append` and `TrimTill`

Formally the way each operation is specified is using a style called higher-order concurrent abstract predicates (HOCAP) [45, 90]. The basic idea is that the library maintains an invariant $P(\sigma)$, where σ is the current abstract state; a key idea in the HOCAP specification is that this predicate P is chosen by *the client* (that is, the code calling the library).

The predicates in HOCAP specifications express ownership over ghost state. This aspect of Perennial is inherited from Iris and described in more detail in section 3.1.2. In this chapter we use the *view-shift assertion* $R \Rrightarrow Q$ to express ghost updates, which are used to update the predicate $P(\sigma)$ that the library maintains.[1] The intuition for $R \Rrightarrow Q$ is that it represents some resources which allow the holder of the assertion to take $R * (R \Rrightarrow Q)$ and "fire" the view shift, updating the underlying ghost variables, and finally obtain Q.

For each operation, the client passes in a view-shift $P(\sigma) \Rrightarrow P(\sigma') * Q$, for any transition from σ to σ' that is a valid transition for the operation. This view shift represents a kind of once-usable "callback" which transforms $P(\sigma)$ into $P(\sigma')$, by updating the caller's ghost state in P. At the end of the operation, the client receives Q as a kind of certificate that the callback ran. The intuition here is that the library's proof must "fire" or "call" the view shift in order to preserve P and produce Q, but the proof gets to pick the exact moment when it should fire the view shift, which is the linearization point of the operation. Note that the view shift might be proven using exclusive resources that cannot be duplicated, so it is important that it is only fired once.

We can see both the HOCAP pattern and the circular-buffer resources in action in the speci-

[1] Note that this is a slight change in notation, compared to the update modality used in the Perennial chapter; the view shift is equivalent to $R \mathrel{-\!\!*} \Rrightarrow Q$.

fication for Append:

$$
\begin{array}{l}
\{\texttt{is\_circular}(\gamma) * \\
\quad (\forall \sigma, P(\sigma) \Rrightarrow P(\sigma +\!\!+ \texttt{upds}) * Q) * \\
\quad \texttt{circ\_appender}(\gamma, \texttt{c}) * \\
\quad \texttt{end\_is}(\texttt{end}) * \texttt{start\_at\_least}(\texttt{start}) * (\texttt{end} - \texttt{start} + \texttt{length}(\texttt{upds}) \leq 511) \\
\} \\
\qquad \texttt{c.Append(end, upds)} \\
\{\texttt{circ\_appender}(\gamma, \texttt{c}) * \\
\quad \texttt{end\_is}(\texttt{end} + \texttt{length}(\texttt{upds})) * Q\}
\end{array}
$$

There are several aspects to the specification, mostly in the precondition. The first line is the most straightforward: we need the circular buffer's invariant to hold, for a particular ghost name γ. The second line is part of the general HOCAP pattern. The view shift updates $P(\sigma)$ to the appropriate new state, denoted with $\sigma +\!\!+ \texttt{upds}$ as shorthand for σ with `upds` appended to the updates field (and the same `start`). The proof of this specification executes the view shift at the linearization point of the `Append` operation, changing the state inside the `is_circular` invariant. The view shift produces a client-supplied assertion Q, which is returned in the postcondition.

The assertion $\texttt{circ\_appender}(\gamma, \texttt{c})$ asserts that the in-memory state cached in `c` is correct (these are the addresses for the updates but not their contents). This resource is unique, since the appending thread needs ownership this in-memory buffer, and is returned in the postcondition for the next `Append` operation. Exclusive ownership over this buffer is one reason why appending is only safe from one thread.

The rest of the specification is specific to how the circular buffer manages concurrency. Appending requires precise knowledge of the end position of the circular buffer, which the logger thread maintains. A lower bound on the start position is sufficient to prove that there is currently—and will continue to be—enough space for the updates to fit on disk; the left-hand side of the inequality computes an *upper bound* on how much space will be used after the append because the `start` variable is a lower bound. The postcondition gives the caller back the `end_is` assertion with the new precise end point. Note that the precondition guarantees that the append will not fail due to running out of space. Because the start and end are manipulated without locks the circular buffer implementation does not even have a way to safely dynamically check if there is enough space.

The specification for `TrimTill` is similar to Append:

$$
\begin{array}{l}
\{\texttt{is\_circular}(\gamma) * \\
\quad (\forall \sigma, P(\sigma) \Rrightarrow P(\sigma[:\texttt{newStart}]) * Q) * \\
\quad \texttt{start\_is}(\texttt{start}) * \texttt{end\_at\_least}(\texttt{end}) * (\texttt{start} \le \texttt{newStart} \le \texttt{end}) \\
\} \\
\qquad \texttt{c.TrimTill}(\texttt{newStart}) \\
\{\texttt{start\_is}(\texttt{newStart}) * Q\}
\end{array}
$$

The effect of a `TrimTill`, abbreviated $\sigma[:\texttt{newStart}]$, is to set the start position and delete the first $\texttt{newStart} - \texttt{start}$ updates. For this operation to be safe, it requires that the start not go backwards (recall the Append proof relies on a lower bound for `start`) and not go past the current end position (concretely this would logically delete updates that haven't yet been written!). The precondition encodes these by taking a precise current `start` position and a lower bound on the end, and then requiring the new start variable be between the two. Just like with Append, the installer thread that calls this operation always maintains precise knowledge (ownership) of the start of the circular buffer.

Notice that neither of these specifications have a crash condition. The reason this is not required is because both methods already maintain an Iris invariant in the `is_circular` predicate. The next section, 4.4, details how crashes and recovery are specified in Perennial.

4.4 Crash and recovery reasoning

The strategy behind proving crash safety is to prove a theorem about running a whole system, restarting after every crash. For example, for DaisyNFS this theorem applies to the server's main loop that receives a message from the network, processes it in a background thread, and replies over the network. Immediately following boot, before processing any requests, the server runs a recovery procedure to restore in-memory state. This whole procedure — recovery followed by running the system — is given an idempotent specification that is proven with WPR-IDEMPOTENCE from section 3.4. The crash condition for this theorem is a description of the state of the whole system at any intermediate step; describing this directly would be daunting, but it is doable since each layer describes its part of the crash condition.

The circular buffer is the lowest layer of the implementation, so the way it fits into the larger plan is that the whole system's recovery starts by recovering the circular buffer's state, then uses that state to recover the next layer, and so on until the whole system is ready to run. Schematically this looks like the following:

```
func RunSystem() {
  c, start, upds := RecoverCircular()
  // recover rest of system from c, start, upds
  fs := recoverFilesystem(...)
  for {
    req := GetRequest()
    go func() {
      ret := fs.Handle(req)
      SendReply(req, ret)
    }()
  }
}
```

The circular buffer supplies three things to fit into the whole-system recovery proof: (1) an abstract crash predicate for the state the circular buffer requires for recovery, (2) a crash specification for the circular buffer's recovery procedure itself, and (3) a post-recovery init theorem that helps the caller maintain the circular buffer's crash predicate and also initializes the circular buffer's invariant.

The specification for the library's recovery procedure `RecoverCircular()` is:

$$
\begin{aligned}
&\{\texttt{circ\_state}(\gamma,\sigma) * \texttt{circ\_resources}(\gamma)\}\\
&\quad \texttt{RecoverCircular()}\\
&\{\textbf{ret}\ (\ell, \texttt{diskStart}, \texttt{upds}),\ \sigma = (\texttt{diskStart}, \texttt{upds}) *\\
&\quad \texttt{circ\_state}(\gamma,\sigma) *\\
&\quad \texttt{start\_is}(\texttt{diskStart}) * \texttt{end\_is}(\texttt{diskStart} + \texttt{len}(\texttt{upds})) *\\
&\quad \texttt{circ\_appender}(\gamma,\ell)\}\\
&\{\texttt{circ\_state}(\gamma,\sigma) * \texttt{circ\_resources}(\gamma)\}
\end{aligned}
$$

The first thing to notice about the recovery specification is that it preserves $\texttt{circ\_state}(\gamma,\sigma)$, which gives the current state of the circular buffer, both on crash and if recovery completes. It is also the case that $\texttt{circ\_appender}(\gamma,\ell) \vdash \texttt{circ\_resources}(\gamma)$, so that $\texttt{circ\_resources}(\gamma)$ is also preserved by recovery. These two together are the crash predicate for the circular buffer. We say the crash predicate is abstract because the user of this theorem does not need to know how it is defined.

The circular buffer's proof also supplies the following *post-recovery init* theorem, which sets

up the circular buffer for use after recovery:

$$
\begin{aligned}
&\forall \sigma, P(\sigma) \Rrightarrow P_{\text{rec}}(\sigma) * P_{\text{crash}}(\sigma) \vdash \\
&\texttt{circ\_state}(\gamma, \sigma) * P(\sigma) \Rrightarrow \\
&\exists \gamma'. \texttt{is\_circular}(\gamma) * \\
&\quad \text{cfupd}\left(\exists \sigma. \texttt{circ\_state}(\gamma', \sigma) * \texttt{circ\_resources}(\gamma', \sigma) * P_{\text{rec}}(\sigma)\right) * \\
&\quad \text{cinv}\left(\exists \sigma. \texttt{circ\_state}(\gamma, \sigma) * \texttt{circ\_exchanger}(\gamma, \gamma') * P_{\text{crash}}(\sigma)\right)
\end{aligned}
$$

There are several components to this theorem. First, the circular buffer has the caller provide a way to split the HOCAP predicate $P(\sigma)$ into two parts, $P_{\text{rec}}(\sigma)$ and $P_{\text{crash}}(\sigma)$ — intuitively the former is transferred to recovery as part of its precondition, whereas the latter is "immutable" and held inside an invariant. Second, the theorem consumes $\texttt{circ\_state}(\gamma, \sigma) * P(\sigma)$ and allocates three things: (1) $\texttt{is\_circular}(\gamma)$ (notice that the `is_circular` is defined to be an invariant, and the premise of this theorem is the contents of that invariant), (2) a "cfupd", and a (3) "cinv". The latter two are low-level features of Perennial that we'll now introduce.

The assertion $\text{cfupd}(R)$ ("crash fancy update") is similar to a view-shift $\text{True} \Rrightarrow R$ in that it is a single-use update that produces the resources R. However, the "crash" part indicates that this update can only be fired after a crash, so it can be used to prove a crash condition but is otherwise unusable. Practically speaking, having $\text{cfupd}(R)$ in a precondition allows the user to *cancel* R from the proof's crash condition, as reflected in the following rule:

CFUPD-USE

$$\frac{P \vdash \text{wpc}\ e\ \{Q\}\ \{Q_c\}}{P * \text{cfupd}(R) \vdash \text{wpc}\ e\ \{Q\}\ \{Q_c * R\}}$$

In the context of the circular buffer, what the cfupd sets up for the caller is that if the system crashes, the recovery proof can dismiss the circular buffer's crash predicate, along with $P_{\text{rec}}(\sigma)$, from the overall crash condition. Thus using CFUPD-USE before running the system both sets up the circular buffer for use (by establishing $\texttt{is\_circular}(\gamma)$) and guarantees the circular buffer's part of the crash condition from now on. A simpler but abstract version of this reasoning appears in the WPC-INV-ALLOC rule in section 3.2; this theorem is a concrete use case (split into creating the cfupd and later using it, rather than in one step as in WPC-INV-ALLOC). What Perennial makes possible is to take advantage of the invariant for crash reasoning — what must be carefully handled is that while the invariant is shared by all threads, only one thread can get "credit" for this invariant holding at crash time, in the sense of permission to cancel it from its crash condition.

Notice in the circular buffer's post-recovery init theorem that the ghost name for the circular

buffer changes on crash to a new γ'; the reason for this is that other threads (the logger and installer) have access to some of the circular buffer resources associated with γ. Rather than requiring those threads to "return" those resources on crash and impose a crash condition on that code, this theorem simply creates a new instance of the circular buffer and stops using the old one. As a result on crash the theorem can produce `circ_resources`(γ', σ) and hand out these resources to the caller.

The third component of the theorem is a cinv ("crash invariant"). cinv(R) behaves very similarly to an invariant assertion $\boxed{R}$, except that like a crash fancy update it can only be used after a crash. This crash invariant "freezes" the old state of the circular buffer (notice that `circ_state`(γ, σ) is now unchanging, since future operations will interact with the instance named γ'). This process also produces `circ_exchanger`(γ, γ'), which includes additional ghost state relating the old and new instances in the form of the predicate. The circular buffer in particular does not have an interesting exchanger predicate, but this feature of the specification pattern shows up in the GoTxn proof.

In general, each layer supplies a post-recovery init theorem. The view shift for each of these theorems is invoked in the proof of a program like `RunSystem()` after running all the recovery code and just before normal processing (after line 4 in the code above). Firing these view shifts allocates all the layers' invariants while simultaneously getting credit for this allocation in the form of the cfupd assertions, similar to how WPC-INV-ALLOC works. Prior to calling this initialization, the recovery procedure has access to the inner contents of all the invariants, reflecting that it has exclusive access, but in turn this proof is required to maintain a complicated crash condition. (One interesting side effect is that recovery code can safely call library functions without using locks, since this code is still guaranteed to be single-threaded.) Once the system starts running the invariant allows multiple threads to share access to this state, and the invariant implicitly guarantees the crash condition recovery depends on.

4.5 Exchanging resources

The final aspect of the logically-atomic specification pattern is the role of the exchanger, one of the conclusions of the post-recovery init theorem. The purpose of this predicate is to give the caller a way to, on crash, relate resources issued by the library before a crash to those after a crash. In the circular buffer, the `start_is(start)` and `end_is(end)` resources are thrown away on crash and re-created during recovery; there are only two of them, so it is easy enough to reconstruct them globally. In contrast, some resources are used locally by a thread and we would like a way to retain ownership across a crash, into that thread's crash condition. This is exactly what happens with GoTxn's lifting-based specification for the journaling layer, explained in greater detail in section 5.5.1. Exchanging gives threads a way to "trade" ownership during

a crash, trading ownership associated with the old γ instance for associated with γ'; both forms of ownership can be exclusive because the process destroys the old ownership and thus can only be performed once.

To make this a bit more concrete, let us look at a part of the GoTxn lifting specification. This specification issues three related resources: the two more important ones are $a \mapsto_d o$ and $a \mapsto_{op} o$. Both represent exclusive access to address a. $a \mapsto_d o$ is a durable statement about the logical disk, while $a \mapsto_{op} o$ gives the value at address a that an in-progress operation op would observe. The lifting-based specification gets its name from a logical "lift" operation that trades $a \mapsto_d o$ for $a \mapsto_{op} o * a \mapsto_d^{\text{lftd}} o$. This brings us to the third type of journaling resource, $a \mapsto_d^{\text{lftd}} o$, which is almost identical to $a \mapsto_d o$ except that it has been lifted and thus cannot be lifted again.[2] We can give the journal's `Commit` operation a specification like the following (simplified, in particular to operate on a single address):

$$\begin{aligned}
&\left\{a \mapsto_d^{\text{lftd}} o * a \mapsto_{op} o'\right\} \\
&\quad op.\texttt{Commit}() \\
&\left\{\textbf{ret } ok, \text{ if } ok \text{ then } a \mapsto_d o' \text{ else } a \mapsto_d o\right\} \\
&\left\{a \mapsto_d o \vee a \mapsto_d o'\right\}
\end{aligned}$$

In the non-error, non-crash return, the postcondition turns $a \mapsto_{op} o'$ into a durable fact $a \mapsto_d o'$. If the system doesn't crash but the transaction fails (which happens if it is too large to fit in the circular buffer), then the caller gets back an unlifted disk fact $a \mapsto_d o$, reflecting that the transaction did not affect the disk. In the case of a crash the specification promises to give one of these assertions, and which one depends on exactly when the crash occurs.

This specification is simplified to only give the case of committing an operation that modifies a single address, when in reality `Commit`'s main purpose is to atomically commit multiple addresses; a single address is enough to understand the proof issues involved. If the system doesn't crash, it is relatively straightforward to reverse the lifting process and restore full ownership of the disk values. $a \mapsto_d o'$ reflects that the commit actually succeeded in changing the disk value, while $a \mapsto_d o$ results from aborting and throwing away buffered writes in op.

On crash it is more difficult to show $a \mapsto_d o \vee a \mapsto_d o'$. The challenge is how to propagate whether or not the writes were made durable from the lower-level abstractions up to this proof; indeed the durability of this high-level operation depends on whether or not the lowest level of the system, the circular buffer, successfully wrote a single header block!

The write-ahead logging layer conveys durability by issuing two resources: one gives the history of multiwrites, and another gives a lower bound on how many multiwrites are durable.

[2] In the implementation, $a \mapsto_d^{\text{lftd}} o$ is the primitive resource. There is a resource $\mathsf{token}(a)$ that gives the right to lift a. Then $a \mapsto_d o$ is simply defined to be $a \mapsto_d^{\text{lftd}} o * \mathsf{token}(a)$.

Its exchanging resource allows the caller to trade an assertion about the history before a crash for a similar assertion after a crash, albeit with some recent writes lost. Crucially the effect of a crash is constrained by the durable lower bound, which can also be exchanged across a crash. On top of this exchanging, the journaling proof implements a fine-grained exchange of ownership over individual addresses.

The actual implementation of exchanging uses a neat separation logic trick. To distinguish between $a \mapsto_d o$ in two different generations, we'll annotate the resource with a γ. The exchanger for the journaling proof is defined in terms of facts like:

$$\texttt{exchange\_addr}(a) \triangleq (a \mapsto_d^{\gamma,\text{lftd}} o) \lor (a \mapsto_d^{\gamma'} o)$$

When proving the post-recovery init theorem, we initially prove $\texttt{exchange\_addr}(a)$ using the right-hand side; this is easy since γ' is fresh, so we've just allocated all of its associated ghost state and it hasn't been used by any threads so far. We can then prove exchange lemmas like $a \mapsto_d^{\gamma,\text{lftd}} o * \texttt{exchange\_addr}(a) \vdash a \mapsto_d^{\gamma'} o * \texttt{exchange\_addr}(a)$. The reason this proof works is that $a \mapsto_d^{\text{lftd}} o$ is *exclusive*, thus from the premise the proof can rule out that $\texttt{exchange\_addr}(a)$ is proven with the left disjunct and learn that the right-hand side holds. The proof takes out this right disjunct $(a \mapsto_d^{\gamma'} o)$ and gives up $a \mapsto_d^{\gamma,\text{lftd}} o$ to re-prove $\texttt{exchange\_addr}(a)$, this time using the left disjunct. This all ensures that exchanging is only performed once, which is necessary since all the resources involved are exclusive.

There is an important asymmetry here where the old ghost state only requires $a \mapsto_d^{\text{lftd}} o$ while exchanging returns $a \mapsto_d o$. This asymmetry is possible because $a \mapsto_d o$ is just like $a \mapsto_d^{\text{lftd}} o$ but with an additional exclusive token that gives the right to lift the address, and we can "drop" the old tokens in the proof and switch to using fresh tokens generated for γ'. In practice this is what permits proving the `Commit` operation's crash condition that has $a \mapsto_d o$ when the specification's precondition only has $a \mapsto_d^{\text{lftd}} o$.

4.6 Summary

The overall logically-atomic crash specification pattern has four components for every layer of the system:

- An opaque predicate for the abstract state of the system (e.g., $\texttt{circ\_state}(\gamma, \sigma)$).
- Proofs for each operation that use a client-specified predicate, modified by requiring a view shift that updates the predicate in accordance with the operation's effect on abstract state.
- An opaque crash predicate for the system.

- A recovery theorem, with a crash condition, that preserves the crash predicate for the system and re-builds any in-memory state needed.
- A post-recovery init theorem, to be called after the whole system has recovered, which sets up the layer's invariant for normal execution, creates ghost state for the next generation, and produces an exchanger resource to relate the old state to the new one.

When layers are composed, the upper layer proof generally hide the lower level in its own proofs — for example, the write-ahead log's crash predicate includes the circular buffer's crash predicate as a sub-term, and subsumes the circular buffer's recovery and post-recovery init theorems.

This pattern can be combined with user-defined ghost resources in order to give more powerful specifications (rather than requiring every operation to be unconditionally atomic). User-defined ghost resources can interact with crashes and recovery; for example, they can be returned from the recovery theorem and used across a crash using exchanging lemmas specific to the library.

4.7 Takeaways beyond formal verification

Physical versus logical logging. The GoTxn log uses physical logging, where log entries are 4KB disk writes. An alternative is to use logical logging, where entries instead correspond to higher-level operations like appending to a file. From a correctness (and especially crash safety) perspective, what makes logical logging challenging is that it appears to break modularity: the concept of a file now shows up in the write-ahead log.

The techniques in Iris and this chapter give a way to think about logical logging modularly where the log stores *procedures*. The insight is that the calling code passes a function to the write-ahead log that interprets the logical log entry into block operations. This makes logical logging a higher-order interface. Note that in practice the code may not be organized this way; we can use *defunctionalization* to specialize the logging code to only the procedures used by the caller, especially for the common use case of a tightly integrated file system and journal. The write-ahead log will rely on some specification (including a crash condition) for the function that interprets its entries. In this simplest case it might require idempotent operations; a more sophisticated logging design like ARIES [68] can handle non-idempotent operations using a combination of redo logging (like GoTxn) and undo logging.

Chapter 5

GoTxn, the transaction system

GoTxn is a transaction system we implemented and verified with the goal of making crash safety and concurrency simple for a storage system implemented on top. Transactions appear to run atomically both on crash and to other threads. The GoTxn specification formalizes this property, which we use in DaisyNFS to enable sequential reasoning for a concurrent file system (described in chapter 6).

This chapter describes GoTxn's implementation, top-level specification, and several aspects of the proof. The implementation is composed of a software stack with several layers; each layer implements a new interface on top of one below. The proof follows the same structure, with several new specifications for intermediate layers. Of particular note are the transaction-refinement specification for the transaction system, the lifting specification for the journaling layer, and the abstraction for the write-ahead log layer.

While as part of this thesis we only used GoTxn to implement and verify a file system, the system and its specification are generic for any storage system implemented on top, as long as its operations are implemented using transactions.

GoTxn faces three key challenges in its specification, design, and proof. First, GoTxn's specification is stated in terms of programs using the transaction system, and not all programs will observe atomic transactions — for example, accessing global variables or the network in the middle of a transaction is not atomic. To state a provable specification for GoTxn we needed a formalization of *safe transactions*, described in section 5.2.3, which obey some restrictions that the specification assumes the caller follows.

The second challenge is supporting an in-memory allocator, which is necessary for the file system to get good performance but appears to violate GoTxn's rule that transactions do not access shared memory since allocating and freeing affect other transactions immediately, not at commit time. To address this challenge we include an allocator in the GoTxn API and proof. A non-deterministic, *under-specification* of the allocator makes its behavior serializable, and chap-

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

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

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

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

Figure 5-1: The API for the transaction system and allocator. Reads and writes between `Begin` and `Commit` appears to execute atomically on disk and for other threads, while `Abort` guarantees the transaction has no effect. The allocator's `Alloc` and `Free` operations are safe to call concurrently.

ter 6 demonstrates that this allocator is usable by using it in the DaisyNFS verified file system.

Finally, this specification is related to a concrete implementation, and thus requires a proof using a program logic. The proof manages the complexity of GoTxn's implementation by combining a *lifting-based specification* that captures the journaling layer's crash atomicity (section 5.5.1) with a simulation proof that captures how two-phase locking's concurrency control works (section 5.5.2).

5.1 Programming with GoTxn

The transaction system exports a transactional API for durable, on-disk objects. The caller can wrap a whole sequence of reads and writes in a transaction between a call to `Begin()` and `Commit()`, and the transaction system guarantees that all of the operations appear to execute atomically (that is, all at once). The programming interface is listed in fig. 5-1. Reads and writes can be for objects smaller than a full 4KB block, which improves concurrency. Transactions make

it much easier to implement a correct storage system by handling the challenges of crash safety and concurrency, so that the system on top only needs to implement its own data structures and operations on top of disk objects.

To use GoTxn, a storage system wraps all of the code for every operation in a single transaction in order to make it atomic. The `Begin()` call creates an empty transaction. The body of the transaction appears to execute atomically when the operation finishes it with `Commit`, or the transaction is discarded with no effect on `Abort`. Reads and writes operate on addresses that specify a position by giving a block number and an offset in bits (always less than $4096 \cdot 8$, the number of bits in a block). The `Read` method requires an explicit size argument while the size of a `Write` is implicit in the size of the `data` slice. We separate out the bit-sized operations to `ReadBit` and `WriteBit` (rather than using a single-element byte slice) to simplify the specification.

Figure 5-1 also includes an API for allocation alongside the transaction API. The allocator should be considered part of the transaction system API insofar as its operations are allowed within a transaction. In contrast, using other shared memory within a transaction is not permitted since it would compromise the atomicity guarantees — the transaction system acquires locks to make reads and writes seem atomic, but it doesn't have any locking discipline for other state. Section 5.2.3 explains these restrictions and an *under-specification* technique for fitting the in-memory allocator API into GoTxn's atomicity guarantee.

As described in section 5.3, GoTxn is implemented using two-phase locking. As a result, a transaction acquires a per-address lock on every address it reads or writes along the way. Acquiring multiple locks during a transaction creates the possibility for deadlocks, if two threads acquire locks with different orderings, and the specification does not forbid deadlocks.[1] The two-phase locking implementation does not implement a specific lock acquisition order, leaving it to the calling code to avoid deadlock — for example, in DaisyNFS the implementation of `RENAME` makes sure to lock the smaller inode number first (by reading from it) if the rename is between different directories.

5.2 Specifying GoTxn using refinement

At a high level, GoTxn makes transactions atomic. This chapter formalizes this intuitive definition in the form of *transaction refinement*. Later in chapter 6 we build upon this specification to show how it enables sequential reasoning. Transaction refinement is defined in terms of *refinement*, which relates code to a specification.

[1] Liveness reasoning is quite challenging when combined with concurrency. However, it would be interesting to specify and verify deadlock freedom, a safety property, without verifying the liveness of the whole system.

5.2.1 Crash-safe, concurrent refinement

Abstractly, refinement from a code program to a specification says that the behaviors of the code are a subset of the behaviors of the specification. Refinement relates two programs in terms of their visible behavior, which is used in the specification to connect a specification where transactions are defined to be atomic to an implementation where they physically take many steps. In this thesis we use a *concurrent, crash-safe* notion of refinement that allows the code to have concurrency (that is, it can spawn new threads) and that captures that crashes in the code behave according to a specification of crash behavior. For the purposes of this work, the visible behavior is always network I/O, corresponding to receiving requests for the system or responding to them (for example, processing NFS requests).

Definition (Crash-safe, concurrent refinement)**.** A concurrent implementation p_c *crash refines* (or simply *refines*) a specification program p_s, written $p_c \sqsubseteq p_s$, if whenever there are initial states σ_s and σ_c satisfying $\text{init}(\sigma_s, \sigma_c)$ and p_c can execute from σ_c and produce a trace of network I/O tr, then p_s can execute from σ_s and produce the same trace tr. Execution might involve crashing and restarting a program (potentially multiple times), wiping out any in-memory state after each crash.

The intuition behind the notation $p_c \sqsubseteq p_s$ is that the set of behaviors of p_c (the set of traces of network I/O tr) is a subset of the behaviors of p_s. The statement $p_c \sqsubseteq p_s$ leaves implicit a definition of initial states $\text{init}(\sigma_s, \sigma_c)$, which will generally say both states are all zeros and of the same size. This notion of refinement extends a standard definition of concurrent refinement with crash-safety by allowing crashing in the execution of the implementation, which must correspond to a specification-level crash transition.

5.2.2 Modeling programs for refinement

To define the transaction-refinement specification, we need to be more precise about what a program is and how it executes. We write $p : \texttt{Go}\langle \text{X} \rangle$ to say p is a Go program written using operations from layer X. Layer operations are always atomic transitions in a state machine. Layers will be one of Sys, Txn, or Disk, where Sys is a stand-in for an arbitrary system implemented on top of GoTxn. We write "Sys" generically to emphasize that the GoTxn specification does not fix how the caller uses transactions; in practice it will be instantiated with the NFS state machine for DaisyNFS, as described in section 6.3.2. The Txn layer allows the operations of the GoTxn API, with reads and writes over a logical disk and the ability to wrap these into an atomic transaction. The Disk transition system is formalized in Coq as part of the GoTxn proof, and assumes reads and writes of 4KB blocks are atomic.

The concept of a layer is part of how GooseLang formalizes Go. The definition of GooseLang is parameterized by some "external" operations and state, as described in section 7.4, and these

two together comprise a layer. In all layers, the type Go⟨X⟩ includes a Fork expression to spawn a thread, heap operations on pointers, slices, and maps, and computation on primitives like integers and structs. A program p : Go⟨X⟩ represents a Go program which uses any of these Go primitives and only imports additional operations from the layer X. This factoring is used to represent a class of Go programs that uses external state in a controlled way (limited to a layer), for the purpose of defining the specification. Each GooseLang program is limited to one external layer to simplify the definitions, but of course Go permits unlimited imports.

The transaction-refinement specification for GoTxn below in section 5.2.3 relates a *spec program* p : Go⟨Txn⟩ that uses transactions, to its executable version which invokes the GoTxn implementation. These specification programs can invoke transactions, which we write `atomically{f}` to represent a transaction running `f`, which in turn might have calls to Read and Write from the GoTxn API. The Read and Write operations in this model are primitive, external calls. Each specification program p has an associated implementation program link(p, `txn`) : Go⟨Disk⟩. The notation is intended to suggest the linking process where each call to GoTxn is replaced with an implementation that uses the Disk operations. Linking has the additional effect, specific to the transaction system, of replacing a specification transaction `atomically{f}` with a sequence `tx := Begin(); f(tx); tx.Commit()` (with some additional code handling aborts).

5.2.3 Transaction refinement

The specification we give to GoTxn uses *transaction refinement*, which formalizes serializability (sometimes known as atomicity and isolation) for transactions running on top of GoTxn. To set up this specification, consider a program p : Go⟨Txn⟩ that uses transactions. To run p, it is combined with the transaction-system implementation, producing a program link(p, `txn`) : Go⟨Disk⟩ that can be run on top of a disk. Transactions in the linked program continue to have the expected atomic behavior, so long as transaction code in p follows certain restrictions, such as not accessing shared state outside the journal system. We write safe(p) to mean p is "safe" in the sense that it follows these restrictions.

The correctness of the transaction system is expressed by the following theorem:

Theorem 5.1 (Transaction refinement)**.** The transaction system's implementation `txn` is a *transaction refinement*, meaning for all p : Go⟨Txn⟩, if safe(p), then link(p, `txn`) $\sqsubseteq p$. The definition of init(σ_s, σ_c) in this refinement relates an all-zero physical disk to an all-zero transactional disk of the same size.

What the theorem says is that if a program is safe, the program linked with the transaction system always behaves as if its transactions were atomically accessing a transactional disk logically maintained by the transaction system. Unlike the definition of concurrent, crash-safe

refinement, which is between two complete programs, transaction refinement is a *contextual refinement* property about a library that is stated in terms of all callers of that library. The theorem is stated in Coq and has a fully mechanized proof in Perennial.

The definition of safe(p) formalized in Coq requires that any code within a transaction not access any shared memory outside of the transaction layer; other than that, transactions can use the GoTxn operations to interact with the logical disk and the allocator, and do any computation in between these operations. For example it is safe for transaction to issue data-dependent operations, where the addresses in a transaction depend on earlier reads. The restriction to not use other shared state is a natural one for the system's correctness; for example, reads and writes to global variables would clearly be non-atomic since the transaction system does not have any concurrency control or protection over such variables.

Safety also requires that transactions follow the preconditions of the `Read` and `Write` operations, which require a discipline of accessing each object with a fixed size — this is a limitation of the current GoTxn implementation and proof, which does not handle concurrency between overlapping addresses. Finally, safe programs can only `Abort` or `Commit` a given transaction once. The notion of safe program will be important when linking this proof with the Dafny proofs, since the transaction system's proof only applies to a safe caller.

The allocator is part of the transaction API so that safe transactions have access to this important in-memory data structure. However, the allocator operations `Alloc` and `Free` do not hold a lock throughout the transaction, and thus they would appear to have an affect on concurrent transactions before `Commit`. The reason why transactions are serializable despite this behavior is that we under-specify the allocator's behavior to cover possible concurrent interference. In particular the specification for `Alloc` merely promises to return an in-bounds address, and GoTxn's specification does not even track the set of allocated/free addresses for each in-memory allocator.

In practice the way the caller uses this specification for a given allocator is to store the ground-truth allocated/free state in the on-disk state of the transaction system, then use an in-memory allocator as an efficient way to find a free bit. Without some in-memory allocator, searching for a free bit over the on-disk state would be difficult to do in an efficient way. The return value of `Alloc` must be checked against the durable bitmap, which is easily done since GoTxn supports accessing an individual bit with `ReadBit`. There is a chance that `Alloc` returns a used address, if it was freed in memory by a concurrent transaction that hasn't committed to disk, but the allocator is designed not to return recently freed addresses to avoid this issue. Similarly, in addition to calling `Free` the caller also issues `tx.WriteBit(`a`,`false`)` to mark the correct bit free on disk. During recovery, for the in-memory allocator to be useful it must know what numbers have already been allocated. The caller initializes the in-memory allocator to the same state as what is stored on disk with `MarkUsed(n)`.

5.3 Implementing GoTxn

The transaction system is structured into several layers, as shown in fig. 5-2. At a high level, three of the abstractions are useful to understand the overall structure: the WAL, the JRNL, and finally the top-level TXN. The first useful abstraction is the write-ahead log, which behaves like a disk with an atomic multiwrite operation. Reads and writes still operate on 4KB blocks, but a multiwrite appears to update multiple disk blocks simultaneously even if the system crashes. Next, the JRNL layer implements *journaling*, persisting a whole operation with reads and writes to disk atomically. Concurrent operations must access disjoint sets of addresses for safety; concurrency control is left to the caller in this layer. Operations can manipulate objects smaller than a block ("sub-block" objects), which improves concurrency by making more operations disjoint. Support for sub-block objects is implemented in the OBJ layer between the WAL and JRNL. Finally, the TXN layer exports the complete transactional interface in fig. 5-1. This layer implements automatic concurrency control so that the caller can freely read and write any addresses.

Layer	Description
TXN	Transactions implements concurrency control using two-phase locking
JRNL	Journaling implements in-memory buffering
OBJ	Sub-block objects implements reads and atomic writes within a block
WAL	Atomic whole-block multiwrites implements whole-block write-ahead logging
CIRC	On-disk queue with atomic append implements a durable circular log

Figure 5-2: The layers in the GoTxn implementation.

The write-ahead log is implemented by organizing the disk into a small, fixed-size circular buffer and a remaining data region. Data is first atomically *logged* to the circular buffer and then eventually *installed* to the data region, to free space in the circular buffer. Reads first go through the circular buffer (which is cached for efficiency) and then access the data region. The circular buffer's implementation is described in greater detail in section 4.1.

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

Because disk writes are slow, for good performance the journal executes many tasks in par-

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

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

The JRNL layer implements a journaling system which gives the caller a useful abstraction over the disk that makes it easy to update the disk in a crash atomic way but which requires that the caller implement appropriate concurrency control. The TXN layer automatically implements the required concurrency control; this isn't much code beyond the journal, but it does dramatically change the specification since the caller sees any sequence of reads and writes as atomic both with respect to threads and crashes (also known as *serializability*). The TXN layer ensures transactions don't conflict using two-phase locking. This algorithm acquires a lock on each address the first time it is used in a transaction. Writes are buffered locally until commit time, at which point they are written atomically using the journaling system. Finally all the locks are released, exposing the transaction's effects to subsequent transactions.

5.4 Verification overview

Figure 5-3 gives an overview of the libraries (or layers) in the GoTxn proof. On the left side of the figure is the code, organized into a stack of libraries, each of which uses the library below. The code is written in Go and then translated to a model in Coq using Goose (described in chapter 7); each proof is about the model of that library. Each intermediate layer has a specification which

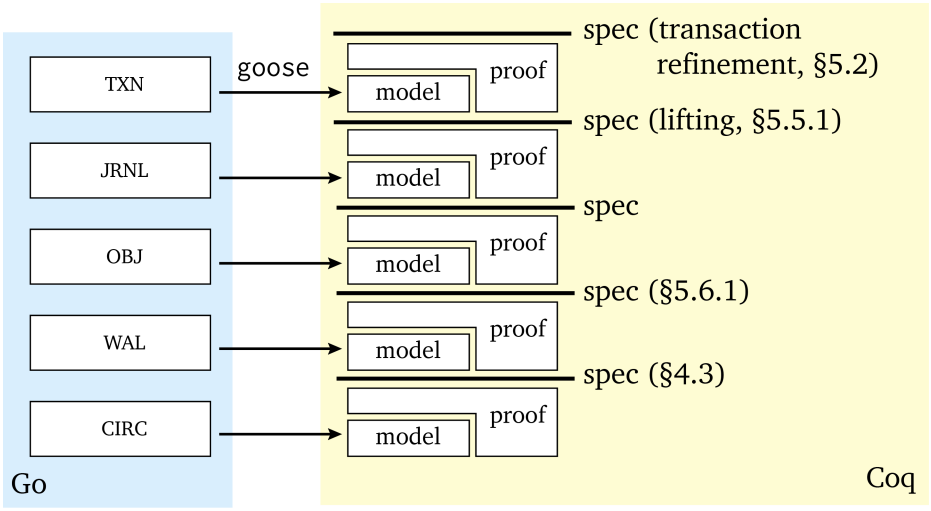


Figure 5-3: Overview of the GoTxn proof, which is divided into proofs for each layer. The proofs are all built on top of Perennial.

is formally described using Perennial's logically atomic crash specifications (TXN has a different specification). The proofs are all implemented using Perennial for crash safety and concurrency reasoning, combined with Goose's reasoning principles for Go.

At the bottom level, the CIRC library is implemented using Go primitives supported by Goose, including an API to access the disk. The disk supports synchronous reads and writes of 4KB sectors, a standard feature for disk hardware. On crash, the model of the disk in Perennial assumes writes are atomic at this 4KB granularity.

At the top level, TXN's specification is the transaction refinement specification described earlier in section 5.2. This specification is about an arbitrary program that uses GoTxn, which is formalized using GooseLang, the language that we use to model Go code. Note that the statement of transaction refinement only talks about the GoTxn implementation and a (GooseLang) program using it; it no longer references the Perennial logic. We are able to prove such a theorem because Perennial comes with a soundness theorem that relates the theorems proven using Perennial's crash specifications to the code's execution. Making the final theorem independent of the logic is important in the next chapter, chapter 6, which connects the GoTxn specification to proofs in Dafny that do not use the Perennial logic.

5.5 Verifying atomicity in GoTxn

This section outlines the proof of GoTxn's transaction-refinement specification from section 5.2. The implementation consists of multiple layers stacked together, as described in section 5.3. This section focuses on proofs of the upper two layers: the journaling layer implements *crash*

atomicity while the TXN layer uses two-phase locking for *concurrency atomicity*. The proof is carried out modularly, so the specification for journaling and the proof of two-phase locking are both described here. The following section, section 5.6, describes aspects of verifying the lower layers.

Recall that transaction refinement is a statement about an arbitrary program that uses transactions. It would be infeasible to directly reason about the execution of an arbitrary program using GoTxn, including handling concurrently issued transactions and crashes at any time. Instead, the first step is to give a specification to the individual methods of the GoTxn API, such as `Read`, `Write`, and `Commit`. Next, the proof considers an arbitrary transaction, but this part of the proof can now invoke the specifications for each method.

To give a specification at the method level, we first develop the right representation of the intermediate state of a transaction. This representation needs to capture buffered writes (which will eventually be durable at commit time) and the old state of the logical disk (which it will revert to if the system crashes or the transaction aborts). It also needs to handle concurrent transactions — the tricky thing here is that the logical disk can change in the middle of a transaction due to a concurrently committed transaction, so for example the model of aborts cannot be as simple as reverting to the disk at the start of the transaction.

A new lifting-based specification for journaling solves these problems. We introduce a purely logical operation called *lifting* that moves ownership of a subset of the logical, durable disk into a transaction. The transaction then operates freely on that part of the disk, regardless of concurrent transactions. This works because the proof can only lift disjoint parts of the disk, which is physically guaranteed by using a per-address lock in the TXN layer. Eventually the transaction finishes reading and writing, and when it commits any buffered changes can be merged into the global disk. Aborts are modeled as returning ownership and merging back the old values of just the lifted part of the disk.

5.5.1 Lifting specification for journaling (JRNL)

Lifting is defined at an intermediate layer, JRNL (for journaling), rather than for TXN which is the top-level API of GoTxn. Journaling uses what we call "operations" to distinguish them from "transactions" — reads and writes are performed from within the context of an operation, and like a transaction the writes are committed atomically to disk, but the difference is that the calling library is responsible for guaranteeing concurrent operations manipulate disjoint sets of addresses. The complete API is shown in fig. 5-4. In the case of GoTxn, that caller is the two-phase locking code, whose proof is described in section 5.5.2.

In the lifting specification, there is a global, logical disk representing the state of the journaling system, and within each ongoing operation there is an operation-local view of the disk. The operation-local view is like the disk view but also incorporates buffered writes. The durable

```
func Begin() *Op
func (op *Op) Commit()

func (op *Op) Read(a Addr, szBits uint64) []byte
func (op *Op) Write(a Addr, szBits uint64, d []byte)
```

Figure 5-4: The API for the journaling layer, JRNL. There is no `Abort()` method (the caller simply abandons the `op` struct). Reads and writes support bit-sized objects within the same API. The `Addr` struct is the same as the one given in the GoTxn API (fig. 5-1).

views are disjoint, so that an address is either in the global disk or "lifted" into a particular operation. Disjointness is what guarantees that every operation-local view is really local and doesn't change due to concurrent threads.

The lifting-based specification uses a logical concept of ownership to keep all of these views disjoint and consistent. The state of each view is defined using ghost state in Iris, and the proof uses separation-logic resources to express ownership over that ghost state; for a general introduction to the idea of ownership in separation logic, see section 3.1.2. The journaling proof issues three types of resources, all per-address: $a \mapsto_d o$, $a \mapsto_{op} o$, and $a \mapsto_d^{\text{lftd}} o$. The first is the most straightforward: $a \mapsto_d o$ asserts that the global durable disk has value o at address a (we use the metavariable o since these addresses have objects that can be smaller than a block, such as a 128-byte inode value or even a single bit). In addition, $a \mapsto_d o$ expresses ownership over the address a. The second resource, $a \mapsto_{op} o$, asserts ownership of the operation-local view associated with an in-memory operation op; it mean that either the on-disk value is o or that op contains a buffered write with the value o to address a. The last resource, $a \mapsto_d^{\text{lftd}} o$ is similar to $a \mapsto_d o$, but the difference is related to lifting and explained below.

The life cycle of these resources, as they are modified by lifting, writes, commit, and crashes, is shown in fig. 5-5. The complete specifications are detailed in the remainder of this section.

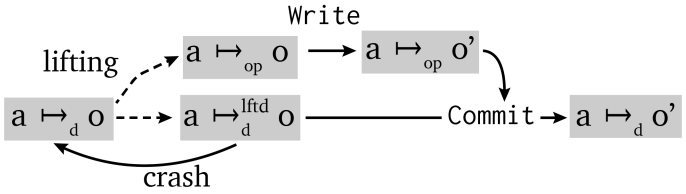


Figure 5-5: Life cycle for the resources in the lifting specification.

Initialization

The first step in using the journaling specification is to initialize the whole system. Initialization is somewhat complicated by restrictions GoTxn places on how sub-block objects are used. The

proof requires the caller to fix ahead of time (when the whole journaling system is initialized) a size for the objects within each block. Currently our proofs assume that objects are either a single bit, 128 bytes (to hold inodes), or a full block, but supporting an arbitrary power-of-two-sized block should be a straightforward extension. To initialize the system, the caller supplies a "schema" which gives a size for the objects in each disk offset (that is, every 4KB aligned offset, not the journal's logical sub-block addresses); this enforces that all the objects within a block are of the same size.

The journal's initialization is specified by giving the caller a lemma that transform ownership over the entire physical disk into the precondition for the journal's recovery procedure; from this point forward the usual crash and recovery reasoning in Perennial applies. For each offset i and block size k, the initial ownership for that disk block consists of the address $(i, k \times o)$ within that block (this is mathematical notation for the Go struct `Addr{Blkno: i, Offset: k*o}`). More formally the initial ownership for a given i and k is given by the following definition:

$$\text{zeroObjects}(i,k) \triangleq \mathop{\Large *}_{0 \le k \times o < 4096} (i, k \times o) \mapsto_d 0$$

The initialization lemma JRNL-INIT creates resources for all of these initial objects, starting from an all-zero disk. The lemma takes a schema *kinds* that maps disk addresses to block sizes and a disk size s; this schema is arbitrary but fixed at initialization time. We use $i \mapsto_{\text{disk}} 0$ to denote ownership over a physical disk offset i (not to be confused by the $a \mapsto_d o$ issued by the journaling proof).

JRNL-INIT

$$513 \le s \wedge \text{dom}(\mathit{kinds}) = \{i \mid 513 \le i < s\} \wedge$$
$$\mathop{\Large *}_{0 \le i < s} i \mapsto_{\text{disk}} 0 \mathrel{-\!\!*}$$
$$\Rrightarrow \texttt{is\_txn\_crash\_condition}(\mathit{kinds}) *$$
$$\mathop{\Large *}_{(i,k) \in \mathit{kinds}} \text{zeroObjects}(i,k)$$

Notice that this initialization is just a logical operation; the schema is not even required at runtime for the code. It creates `is_txn_crash_condition`, which is the precondition and crash condition for the transaction system's recovery procedure. The journal objects in this lemma's conclusion are initially fully owned by recovery, which then shares them with threads. The journaling system's proof does not fix a particular mechanism for *how* access to these resources is mediated; it simply requires exclusive access to an address in the form of a $a \mapsto_d o$ resource. In practice, GoTxn mediates access to $a \mapsto_d o$ by protecting it with a per-address lock.

Preparing operations

The specification for `Begin()` is straightforward, since the operation has not yet interacted with the disk:

$$\begin{array}{l}\{\text{True}\}\\ \quad \texttt{Begin}()\\ \{\textbf{ret}\ op, \texttt{is\_op}(op)\}\end{array}$$

The specification has a trivial precondition of True and its postcondition returns an assertion `is_op`, which says that *op* is a valid `*Op` object that represents a journaling operation. (This is different from the `*Txn` struct that represents an ongoing transaction, which has additional state to track the locks acquired so far.) The operation starts out with an empty operation-local view and ownership over no addresses; the ghost state for this operation's view is initialized by the proof of this theorem and represented as part of $\texttt{is\_op}(op)$.

In order to interact with an address within an operation, the specification requires the caller to start with $a \mapsto_d o$ and then *lift* it to obtain an operation-local resource, where lifting is the following purely logical operation JRNL-LIFT:

JRNL-LIFT

$$\texttt{is\_op}(op) * a \mapsto_d o \Rrightarrow \texttt{is\_op}(op) * a \mapsto_{op} o * a \overset{\text{lftd}}{\mapsto}_d o$$

Notice that the result of lifting is both an operation-local assertion and $a \overset{\text{lftd}}{\mapsto}_d o$; this latter assertion is much like $a \mapsto_d o$ in that it asserts the on-disk value of address a, but it cannot be lifted again. This is required for soundness; only one of $a \mapsto_d o$ and $a \mapsto_{op} o$ is allowed for ownership to actually be exclusive. Lifting takes and preserves $\texttt{is\_op}(op)$, which reflects that the ghost state for *op* has been set up.

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

$$\begin{array}{l}\left\{\texttt{is\_op}(op) * a \mapsto_{op} o * \texttt{buf\_obj}(buf, o')\right\}\\ \quad op.\texttt{OverWrite}(a, buf)\\ \left\{\texttt{is\_op}(op) * a \mapsto_{op} o'\right\}\end{array}$$

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

The specification for `ReadBuf` is similar. `ReadBuf` returns a buffer that points into the `op`

struct, so it has a more sophisticated spec:

$$\begin{array}{l}\left\{\texttt{is\_op}(op) * a \mapsto_{op} o\right\} \\ \quad op.\texttt{ReadBuf}(a) \\ \left\{\begin{array}{rl} & \texttt{buf\_obj}(buf, o) * \\ \textbf{ret}\ buf, & (\texttt{buf\_obj}(buf, o) \mathrel{-\!\!*} \\ & \quad \texttt{is\_op}(op) * a \mapsto_{op} o)\end{array}\right\}\end{array}$$

This states that, when `ReadBuf` finishes, it returns a buffer *buf* and two resources: $\texttt{buf\_obj}(buf, o)$ says the buffer has the old object o, while the second is a separating implication or *wand* $-\!\!*$. The wand says that if the caller returns ownership of $\texttt{buf\_obj}(buf, o)$, it can get back the $\texttt{is\_op}(op)$ predicate. This allows the caller to use the buffer to read the data before continuing to read and write within the operation. The complete specification in the JRNL layer also supports modifying the object using the buffer returned by `ReadBuf` [13], but GoTxn does not expose that feature.

Commit

The final method in the journaling API is `Commit()`, which atomically persists all the writes buffered in the operation. The specification is based on two maps m and m' whose domains are the set of all addresses lifted into the operation. The first map m gives the on-disk, pre-operation values while m' has the new values buffered in the operation.

$$\begin{array}{l}\left\{\begin{array}{l}\mathsf{dom}(m) = \mathsf{dom}(m') * \\ \left(\mathop{\Large *}_{(a,o)\in m} a \mapsto_d^{\text{lftd}} o\right) * \left(\mathop{\Large *}_{(a,o')\in m'} a \mapsto_{op} o'\right)\end{array}\right\} \\ \quad op.\texttt{Commit}() \\ \left\{\textbf{ret}\ ok, \text{if } ok \text{ then } \mathop{\Large *}_{(a,o')\in m'} a \mapsto_d o' \text{ else } \mathop{\Large *}_{(a,o)\in m} a \mapsto_d o\right\} \\ \left\{\left(\mathop{\Large *}_{(a,o)\in m} a \mapsto_d o\right) \vee \left(\mathop{\Large *}_{(a,o')\in m'} a \mapsto_d o'\right)\right\}\end{array}$$

The precondition in this specification requires ownership over a subset of the logical disk, the addresses in $\mathsf{dom}(m)$, both on disk and within the operation-local view. If the system does not crash and returns $ok = \texttt{true}$, `Commit` returns ownership over the new values on disk. It returns full disk points-to assertions $a \mapsto_d o'$ because these addresses are "lowered" so that subsequent operation can lift them again. If the commit aborts (which only happens if the transaction does not fit in memory), then the specification still returns full disk ownership albeit with the old

values. The crash condition of the `Commit` specification captures that even if the system crashes, it still returns ownership over the disk resources, with either the old or new values. Whether this disjunction goes to the left- or right-hand side depends on exactly when the system crashes; see section 4.4 for a discussion of how the proof handles this.

It's a little easier to see the overall structure of `Commit`'s specification when specialized to a single address (although the real specification is powerful precisely because it captures atomicity across multiple addresses):

$$
\begin{array}{l}
\left\{a \mapsto_d^{\text{lftd}} o * a \mapsto_{op} o'\right\} \\
\quad op.\texttt{Commit}() \\
\left\{\textbf{ret}\ ok, \text{ if } ok \text{ then } a \mapsto_d o' \text{ else } a \mapsto_d o\right\} \\
\left\{a \mapsto_d o \vee a \mapsto_d o'\right\}
\end{array}
$$

5.5.2 Proving transaction refinement

Recall that theorem 5.1 is the overall correctness theorem for GoTxn. It says that the GoTxn implementation is a *transaction refinement* between the Txn layer (with programs that use transactions) and the Disk layer (with an implementation that interacts with a disk). That is, given a program $p : \texttt{Go}\langle\text{Txn}\rangle$, if the transaction operations of p are linked with GoTxn (implemented in $\texttt{Go}\langle\text{Disk}\rangle$), the implementation program's observable behaviors (from running on a disk) are a subset of the specification's behaviors (with high-level transaction-system operations and atomic execution for transactions). The proof of this theorem leverages the journaling specification from section 5.5.1, extending it to also reason about the concurrency control implemented with two-phase locking.

In general, the Perennial framework supports proving crash-safe, concurrent refinements between a specification layer $\texttt{Go}\langle\text{S}\rangle$ and an implementation layer $\texttt{Go}\langle\text{I}\rangle$ by constructing a *forward simulation* between the specification and its implementation. The simulation is expressed in terms of refinement conditions that are written using the usual Perennial Hoare triples specifications (with pre- and postconditions), so that they can be proven using the full spectrum of techniques in Perennial.

Following an approach developed by Turon et al. [94], in a refinement proof in Perennial the execution of the specification program is represented by *ghost state*. The logic then has assertions for describing this ghost state. For example, for the Txn layer, the assertion $\texttt{a} \mapsto_{\texttt{txn}} b$ says that address a contains the value b in the ghost transaction system's state. In addition, there is the *thread points-to* assertion, written $j \Rightarrow \texttt{e}$, which says that thread j in the specification program is executing program `e`. Perennial has rules for updating the ghost state by "executing" these

ghost threads, such as the following view shift:

$$(\texttt{a} \mapsto_{\texttt{txn}} b) * (j \Rightarrow \texttt{Write(a, b')}) \Rrightarrow \texttt{a} \mapsto_{\texttt{txn}} b'$$

which updates the value stored at `a` as a result of a write.

To establish the refinement, the proof engineer first defines a *representation invariant* I, an assertion in the logic that describes a relation between the specification state and the implementation state. Perennial's proof rules ensure that this designated invariant must hold before and after each step of a program throughout the proof. Next, the proof engineer proves a Hoare triple for each operation o of $\texttt{Go}\langle\texttt{S}\rangle$ and its corresponding implementation p_o in $\texttt{Go}\langle\texttt{I}\rangle$:

$$\left\{(j \Rightarrow o) * \boxed{I}\right\}\ p_o\ \{\textbf{ret}\ v, (j \Rightarrow v)\}$$

Such a *refinement triple* says that if a specification thread is executing o and an implementation thread is running p_o, then the representation invariant I is maintained and the value v returned by running p_o is a valid return value of operation o. In the proof of this triple, the ghost execution rule above is used at the linearization point of p_o, to mark when the operation logically takes effect by executing o in the ghost code.

These refinement triples imply a concurrent, crash-safe refinement between programs in $\texttt{Go}\langle\texttt{S}\rangle$ and $\texttt{Go}\langle\texttt{I}\rangle$, which is proven in the Coq development using Perennial's soundness theorem. The soundness theorem combines the classic technique of *logical relations* with a low-level soundness theorem from Perennial to derive the desired refinement between the two programs, rather than the usual Hoare triple soundness theorem about a single program's execution. The resulting theorem statement expresses refinement directly (without referencing the Perennial logic), while its proof has access to the full range of Perennial's features.

In the particular case of the transaction system, the key refinement triple to prove is for a block of code f enclosed in transaction `Begin` and `Commit` operations; for example the triple might look the following (for a particular transaction f that copies from address 0 to 1):

$$\left\{(j \Rightarrow \texttt{atomically}\{\, v \leftarrow \mathit{Read}(0);\ \mathit{Write}(1, v)\,\}) * \boxed{I}\right\}$$

```
tx := Begin();
v := tx.Read(0); tx.Write(1, v);
tx.Commit()
```

$$\{j \Rightarrow ()\}$$

The difficulty in proving this triple is that the linearization point is at the very end when the code calls `Commit`, at which point the actual earlier execution of f becomes visible to other threads. The proof must show that ghost-executing the specification's `atomically` block at this point is

valid by tracking the behavior of f.

To show this, our proof maintains a stronger invariant during a transaction's execution. As the transaction executes, we track the initial, on-disk value of any objects accessed in a map J. The domain of this map $\Sigma = \mathrm{dom}(J)$ is the *footprint* of the transaction, which two-phase locking keeps locked during the transaction. The intuition behind the invariant is that if the transaction only depends on J, the transaction's execution can be delayed to take place atomically at the call to `Commit` and its behavior will be the same since the subset of the journal J is the same.

The proof needs to reason about the two-phase locking concurrency control in order to use the journaling layer's lifting specification as part of this proof. Perennial has a crash-aware specification for locks, described in section 3.2, that allows us to reason about the per-address locks while also reasoning about crashes in the middle of a transaction. To use this specification, the proof defines a per-address lock invariant and crash invariant — the lock invariant is a property that holds when the lock is acquired (and must be shown when it is released), while the crash invariant is a guarantee that also holds on crash. In the case of the two-phase locking code, both the lock and crash invariants for the lock associated with address a contain ownership of $\exists o, a \mapsto_d o$, which gives the transaction the ability to lift address a and read and write it with the journaling system's interface. The invariant has additional constraints to assert that the value o is the same as the one in the transactional disk.

More formally, the proof constructs a second simulation relation during the execution of a transaction `f`. Let J be a map giving the values of each object in the transaction's footprint Σ at the first time they are accessed by `f`, and let J' be a mapping giving the transaction's current buffered in-memory view of the same addresses. Then, the invariant requires that after n steps of execution:

1. The transaction holds the lock for every address $a \in \Sigma$. The transaction has ownership over $\ast_{(a,o)\in J} a \overset{\mathrm{lftd}}{\mapsto_d} o$ and $\ast_{(a,o')\in J'} a \mapsto_{op} o'$. These come from lifting the $a \mapsto_d o$ assertion from each lock invariant, and then writing to update the operation-local points-to assertion.

2. Executing n steps of `f` in *any* starting state that has the same values as J for the addresses in Σ can lead to a state with values given by J', and the same value for the other addresses.

When the transaction is about to commit, the locking described by the first part of the invariant ensures that the durable value of each address still match the values in J. The second part of the invariant means that even though other parts of the state outside of Σ may have changed since the transaction started, those changes do not affect execution of `f`. Thus, the ghost execution of `f` at this point will have the same behavior as the implementation. The transaction's invariant maintains ownership of the resources to use the journal's `Commit` specification. The postcondition and crash condition of that specification return new disk points-to assertions consistent

with either J or J' as appropriate. If the commit is successful, then the second part of the transaction system's invariant allows issuing a ghost-execution update to change the transactional disk from J to J' (for the subset Σ), and it guarantees that this correctly simulates `f`.

Showing that the second part of the invariant holds requires that code within a transaction must not access global state outside of the transaction system, as mentioned in section 5.2.3. Accesses to such global state would violate the invariant because their behavior would then depend upon state outside of the footprint Σ. Because those global values could change by the time the transaction commits, the above argument would no longer work if they were allowed.

The allocator creates another subtlety related to the second part of this invariant. Allocations do not hold the allocator lock throughout the remainder of a transaction. This seems to violate the two-phase locking pattern, since allocations could be implicitly observed by other concurrent transactions from the fact that an allocated address is no longer free. As described in section 5.2.3, GoTxn under-specifies the allocator's behavior so that it atomically follows its (non-deterministic) specification even though it is not protected by the 2PL locking discipline.

5.6 Verifying GoTxn's implementation layers

This section walks through the proof of the lower layers of GoTxn's implementation. Rather than give complete specifications and proof sections, it focuses on some of the most interesting aspects.

5.6.1 Write-ahead logging (WAL)

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

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

A contribution from this part of the proof is a new *abstract state* for the write-ahead log, which represents its state in terms of a history of all the multiwrites that have ever been issued. The abstract state is useful for giving a specification for the WAL's operations that is consumed by the proof of the OBJ layer, which uses the write-ahead log. It is not materialized at run time; there is no overhead to logically tracking the history in the proof. This section presents two variants of this state, an internal one that is closer to the implementation but less convenient for the caller, and a more abstract external one that is easier to use in the rest of the proof. Both variants are based on the key idea of using an append-only history as the state of the WAL.

Internal abstract state: logical history of multiwrites. To prove the write-ahead log layer correct, the proof represents the state of the write-ahead log as a logical history of multiwrites together with some important internal pointers, as shown in fig. 5-6. Multiwrites before `memStart` have already been installed, and their log entries do not physically exist in memory or on disk. Multiwrites from `memStart` to `diskEnd` are already logged on disk. Multiwrites from `diskEnd` to `nextDiskEnd` are currently being logged from memory to disk. Finally, multiwrites between `nextDiskEnd` and `memEnd` are purely in-memory, and are eligible for absorption.

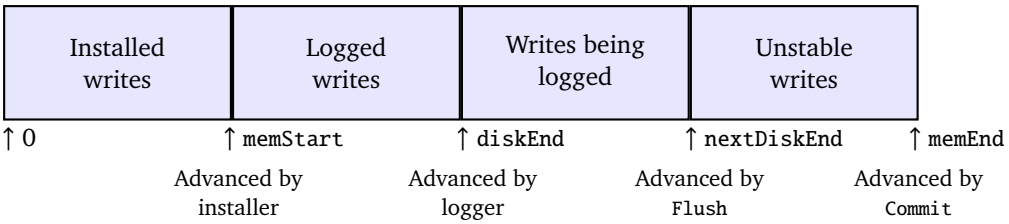


Figure 5-6: The internal abstract state of the write-ahead log. Vertical arrows indicate designated positions in the logical log. Labels below the arrows indicate what thread or function is responsible for advancing that logical position to the right.

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

External abstract state: durable lower bound. Although the details of the logical log are important for proving the WAL layer, the caller (i.e., the OBJ layer) does not need to know about

installation, group commit, and absorption. To abstract away these details, the WAL provides a simplified state as its interface, as shown in fig. 5-7. The simplified state consists of the same history of multiwrites, together with `durable_lb`, which is a lower bound on what set of multiwrites will be preserved on crash. Using a lower bound instead of precisely exporting `diskEnd` means that this abstract view does not need to change if the logger thread adds more multiwrites to disk in the background, and thus hides this concurrency.

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

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

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

Figure 5-7: Parts of the specification for the WAL interface.

Lock-free logging and installation. For performance, GoTxn has dedicated threads that perform logging and installation. However, these threads do not hold any locks while reading or writing to disk. To allow these threads to run concurrently, the write-ahead log layer uses two separate header blocks, as shown in fig. 5-8. One header block (owned by the installer thread) stores the start of the on-disk log, and another header block (owned by the logger thread) stores the end of the on-disk log. This lets the installer and logger concurrently advance their pointers (`memStart` and `diskEnd` respectively) without locks.

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

The logger thread needs to check that `memStart` is far enough along that the log will have

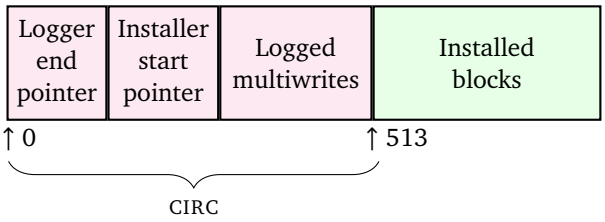


Figure 5-8: The physical write-ahead log.

space for the new multiwrite. The proof reasons that the `memStart` variable read while holding a lock is a *lower bound*, which remains valid even after releasing the lock. Even though `memStart` might grow after it is initially read, the log will only have more space and thus the multiwrite will still fit.

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

The proof of this aspect of the write-ahead log is one of the trickiest parts of GoTxn, because of the lock-free concurrency in the write-ahead log. For more details on the proof strategy, see Mark Theng's thesis [93]. Mark helped complete the proof and described the proof and its invariants in more detail in his thesis.

Specifying lock-free reads. Concurrency in the write-ahead log complicates not just its proof but its specification due a challenge with reading installed data. The difficulty is that reading requires checking the log's in-memory cache and then falling back to the disk, but the disk read happens without a lock. If a multiwrite commits after the read misses in the cache, then the disk read will not observe the latest value. The write-ahead log specification specifies that reading the installed value might return an old view of the disk, and the OBJ layer above handles this weaker specification with an invariant that guarantees the object being read has not been modified since that old view.

More concretely, fig. 5-9 shows the implementation of `Read` in the WAL layer. This implementation is split into two parts. The first part checks the in-memory state by consulting `l.memLog` under a lock (line 9), checking for any updates to the address being read between `memStart` and `memEnd`. If no in-memory updates match the address being read, the second part of `Read` falls back to reading from the installed area on disk (line 15). No single lock is held across the whole operation, so other threads can run between the call to `ReadMem()` and `ReadInstalled()` across lines 2–4. In particular, a thread could run `Commit()` to the same block that another thread is reading, if the two threads are accessing different parts of the same block, as shown in fig. 5-12.

```
func (l *Wal) Read(a uint64) Block {
  b, ok := l.ReadMem(a)
  if ok { return b }
  return l.ReadInstalled(a)
}

func (l *Wal) ReadMem(a) (Block, bool) {
  l.Lock()
  b, ok := l.memLog.get(a)
  l.Unlock()
  return b, ok
}

func (l *Wal) ReadInstalled(a uint64) Block {
  return disk.Read(a)
}
```

Figure 5-9: The implementation of `Read` in the WAL layer.

There is a challenge in specifying the behavior of `Read` because its commit point is not obvious. Consider a case where address a is not in the in-memory log, so that the `Read` operation falls back to `ReadInstalled(a)` — the situation is depicted graphically in fig. 5-10. If there is a concurrent write to address a, then there are two possibilities for the linearization point of the `Read` call: it can either appear to occur before the concurrent write, if `ReadInstalled(a)` returns the old value, or its linearization point might be after the write if the concurrent write is installed before `ReadInstalled(a)` runs. Unfortunately the decision to linearize before `ReadInstalled(a)` or at the point it runs *depends on the future behavior* of the system (the linearization point is before if there is no concurrent write that gets installed, but if there is such an installed write it needs to be after that write).

Instead of proving that `Read` is atomic, we instead give atomic specifications to `ReadMem()` and `ReadInstalled()` individually. The write-ahead log's abstract state includes a lower bound on its installed point in order to describe `ReadInstalled()` in isolation. This installed lower-bound is a lower-bound on the `memStart` pointer from the internal abstract state, in much the same way that the durable lower bound gives a bound on `diskEnd`. The specification for `ReadMem(a)` says that if the address a is not in the log, the operation advances the installed lower bound enough to guarantee that there are no writes to that address after the new installed lower-bound. The specification for `ReadInstalled()` can return any value the address had after the installed lower-bound. To reason about the combination of these operations, the OBJ layer maintains a strong invariant about all the possible blocks that `Read` could return from

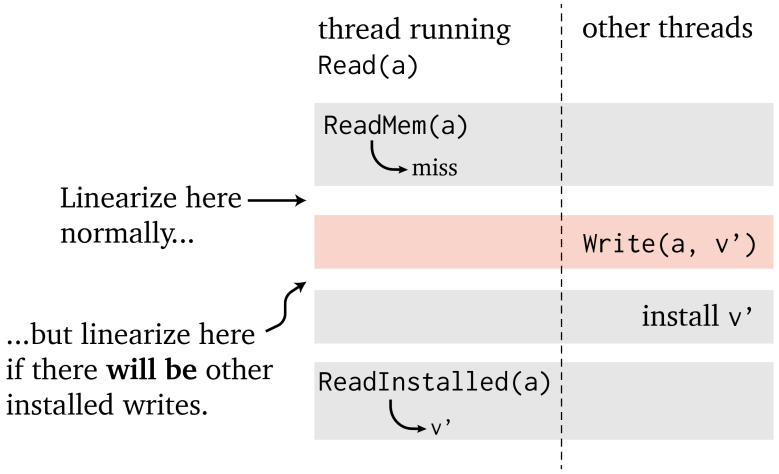


Figure 5-10: The linearization point for `Read(a)` depends on future operations. In this example, there is a concurrent write to the address being read (highlighted in red). Because this write is installed before the `ReadInstalled` operation, the correct linearization point is after the write, but if the installation had not happened yet the `Read(a)` operation would return the old value and the linearization point needs to be before the concurrent write.

the installed point onward, and this is enough to reason about a block returned from either the in-memory cache or the on-disk installed data.

Formally, we believe that it is possible to prove that the WAL `Read()` operation is linearizable, but because of the future-dependence this will require using *prophecy variables*. Perennial does not support them, which forced us to adopt the non-atomic specification described above. Recent results on prophecy variables in Iris [51] could be used to avoid specifying `ReadInstalled()` separately.

5.6.2 Logically atomic crash specifications

Throughout the GoTxn stack we specify internal layers using a transition-system specification, such as the examples illustrated in fig. 5-7 for the WAL layer. Perennial formalizes what it means for the code in a layer to implement a transition system in terms of Perennial's crash triples in a style we call *logically atomic crash specifications*. Chapter 4 gives a more complete description of this encoding. This section gives the high-level intuition for how these specifications are used in the context of the GoTxn layers.

As a motivating example, consider the moment when the logger thread commits a new batch of multiwrites to the physical log in order to advance the durable point `diskEnd` in the logical log of the WAL layer. It does this by calling into the `Append` method of the CIRC layer, which appends to the small buffer of logged multiwrites. The code for `Append` commits at some internal step

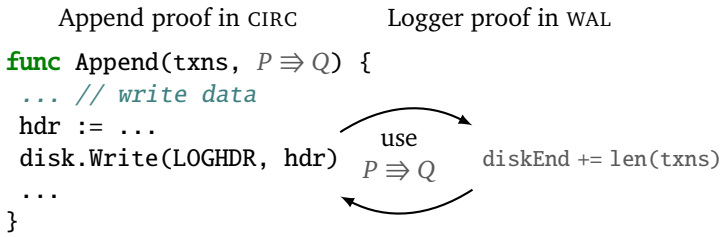


Figure 5-11: Illustration of how the proof of `Append` executes a logical callback $P \Rrightarrow Q$, an assertion in Perennial which updates ghost state. The logger passes a callback that adds `len(txns)` to the `diskEnd` ghost variable.

when it writes the header block and makes the data valid, and it is at this instant that the logical log's `diskEnd` should be incremented. How can we verify `Append` in the CIRC layer separately from the WAL layer, while still executing the right update in the logger proof?

Logically atomic specifications achieve this separation by having the precondition to `Append` take a logical *callback* [45], which the proof promises to "execute" at the commit point. This callback is a view-shift assertion of the form $P \Rrightarrow Q$, a feature in Iris which expresses an update to ghost state that takes the assertion P as input and produces Q as output. The exact update is selected by the logger proof to update the `diskEnd` ghost state of the logical log, as shown in fig. 5-11. This specification for `Append` provides modularity in that the `Append` proof does not need to know about the logical log and its `diskEnd`, and the logger proof does not need to worry about why `Append` is atomic. A key feature of Perennial's logically atomic crash specs lies in that they capture the crash behavior in this callback style, so as to enable a complete proof of crash safety across layers.

5.6.3 Concurrency within a block (OBJ)

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

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

Lock-free reads pose a verification challenge because the disk block can be modified during the read. Consider the example shown in fig. 5-12, where a single disk block stores many inodes. Inode 1 initially contains the value A, while inode 4 contains B. Thread 1 is committing a write

of B' to inode 4 in that block, while thread 2 concurrently reads inode 1 from the same block. To read inode 1, thread 2 will read the entire block, and then copy out the part of the block corresponding to inode 1. The block seen by thread 2 will differ depending on whether thread 1's write happens before or after the read, but inode 1 will contain A in either case.

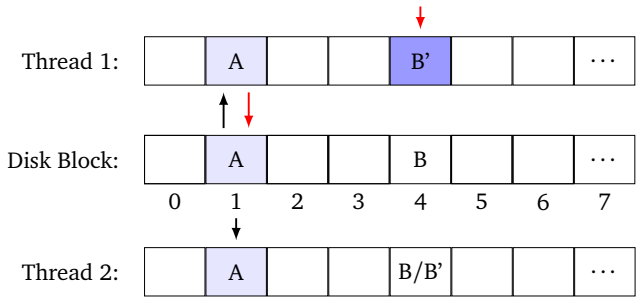


Figure 5-12: An example of concurrent operations on sub-block objects in the OBJ layer. The example has a concurrent `Read` to inode 1 and a `Commit` modifying inode 4.

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

The object layer supports *bit-sized* objects, which create a verification challenge since the operations involved are implemented with low-level bit manipulation. One example is a function `installBit(src byte, dst byte, off uint64) uint64` that returns `src` with the `off`th bit replaced with the corresponding bit from `dst`. The specification for this code uses a pair of conversion functions `byte_to_bits` and `bit_to_bytes` that go between an 8-bit integer (a `u8`) and `list bool` (of length 8). To reason about the implementation of `installBit` we wrote tactics to prove theorems about bytes and bits by brute force, simply considering all possible cases; this works reasonably well in Coq even when there are thousands of cases (but not too much more).

5.7 Limitations

GoTxn has some limitations. The transaction-refinement proof currently only support *synchronous* `Commit`, which makes writes durable when it returns. The code also implements an

asynchronous version of `Commit` that makes the writes atomically visible to other threads, but durable only at a later point. It would be interesting to extend the transaction-refinement specification and proof to cover asynchrony. The two-phase locking implementation only supports write locks and not reader-writer locking. At the bottom layer, GoTxn's disk interface assumes a synchronous disk interface — it would be interesting to relax this to reason about asynchronous writes, which model how disks can lose recent writes on power failure due to internal buffering.

5.8 Conclusion

GoTxn is a concurrent, crash-safe transaction system with a machine-checked proof. The caller wraps a sequence of reads and writes in a transaction, which the transaction system promises to make atomic. GoTxn has a relatively sophisticated implementation. The write-ahead log supports buffering new writes, logging for durability, and installation concurrently since the logger and installer threads access the disk without holding locks. The object layer implements safe concurrent access to objects within the same block. Two-phase locking ensures transactions do not conflict, guaranteeing isolation.

We formalized the system's atomicity guarantee in the form of a *transaction-refinement specification*, which says that when the caller issues an arbitrary sequence of transactions, they appear to execute atomically, even if transactions are issued concurrently and if the system crashes. A key challenge in this specification is formalizing "safe" transactions which GoTxn can make atomic, forbidding arbitrary access to shared memory that would fall outside the two-phase locking discipline. To make GoTxn practical with this restriction it includes an in-memory allocator with a specification that uses non-determinism to fit the allocator into GoTxn's atomicity specification.

GoTxn is verified against its transaction-refinement specification using Perennial. The proof is modular, using Perennial's logically atomic crash specifications to verify each layer separately and compose the proofs together. Of particular note in the proof is the *lifting-based specification* of the journaling layer that captures what concurrent transactions do, and the *history of multiwrites* abstract state to specify the write-ahead log's behavior.

Chapter 6

DaisyNFS, the file system

DaisyNFS is a verified, concurrent, and crash-safe file system, built on top of GoTxn. This chapter describes its specification, implementation, and proof. A key aspect of DaisyNFS is that the proofs of the file-system code use *sequential reasoning*, even though DaisyNFS is a concurrent file system. This is possible due to a *simulation-transfer theorem* that uses the strong guarantee of GoTxn to enable verifying a system written with transactions using a different proof strategy, so that the file-system proofs are carried out in Dafny, a sequential verification-oriented programming language.

6.1 Verification approach

DaisyNFS is an implementation of the Network File System (NFS) API. This is a standard file-system interface, specified in RFC 1813 [7] (specifically, this is the standard for NFS version 3, which is what DaisyNFS implements). NFS is widely used, generally to export a file system across a network to multiple clients, and widely supported by operating systems — Windows Server, macOS, and Linux each include an implementation of both an NFS server and client.[1]

RFC 1813 is a good starting point for DaisyNFS's specification. However, the RFC is a prose document with about 130 pages of English text, which is unsuitable for a mathematical proof. Thus the first step is to turn the RFC into something more precise. DaisyNFS's specification (described in section 6.3.1) uses a transition system defined in Dafny for this purpose, defining the behavior of the protocol with an abstract state and transitions for each NFS operation that define how it evolves the state and what value it returns. The transition system allows non-determinism in the specification to give the implementation some flexibility.

The transition system describes the abstraction of an NFS server, but what does it mean for the `daisy-nfsd` binary to implement this specification? To formalize DaisyNFS's correctness

[1] Windows 10 and 11 can act as NFS clients but do not include an NFS server.

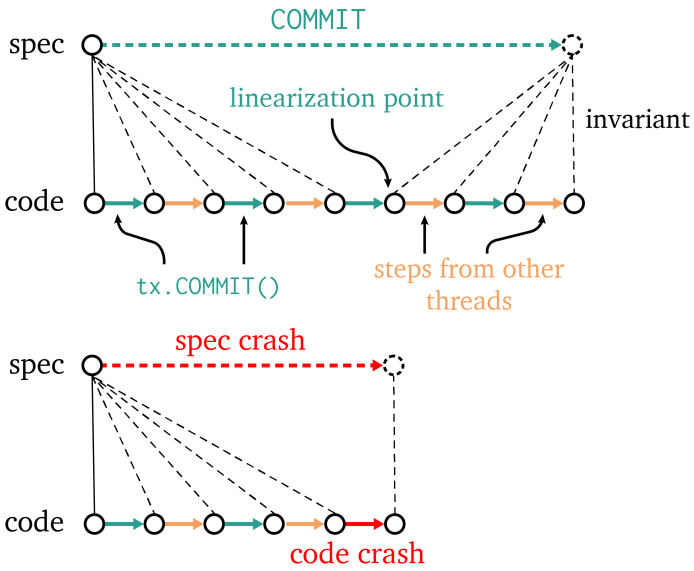
we use the same *concurrent, crash-safe refinement* that was used in the GoTxn specification. The server binary is a refinement of the NFS transition system if every execution of the code — including with concurrent operations crashes — has user-visible behavior that the specification could also produce (that is, the behavior is allowed by the specification). In DaisyNFS's specification the visible behavior is defined to be network requests and responses.

DaisyNFS's concurrent, crash-safe refinement is a much more sophisticated property to verify than sequential refinement. Figure 6-1 illustrates the difference between the concurrent and sequential simulation obligations. For both forms of refinement, the basic proof technique is to construct a *forward simulation* from the code execution to the specification transition system, which requires an invariant connecting their states and a proof that shows the invariant is preserved by operations. In a sequential, non-crash simulation, it is sufficient to show that each operation restores the invariant when it returns since its intermediate states are invisible. The complication in a concurrent simulation is that the code can have many concurrent threads, each running a different operation at the specification level. The proof of any given operation must also show that the intermediate states satisfy the invariant, since at any time other threads might run. Similarly, the proof of each operation's implementation must consider interference with its execution from other threads at any time.

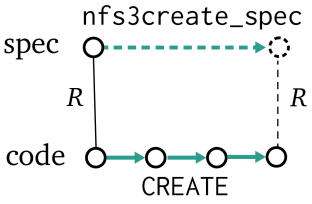
The design of DaisyNFS uses transactions, and in particular GoTxn, to simplify the proof of concurrent refinement. Transactions appear to run sequentially, and thus should permit reasoning about the body of each transaction sequentially even though the actual execution interleaves multiple transactions. This chapter describes a formalization of this intuition in the form of a *simulation-transfer theorem* (described in section 6.4.1) which proves that a system implemented with transactions that is verified with a sequential forward simulation against some specification refines the same specification in the sense of a concurrent, crash-safe refinement when run through GoTxn. The proof of this theorem is an extension of the transaction-refinement specification from section 5.2.

Due to simulation transfer, we can use the simpler verification methodology of sequential simulation for the DaisyNFS file-system code, compared to the Perennial program logic used to verify the transaction system underneath. To fully take advantage of this difference, DaisyNFS is verified using Dafny [61], an entirely different tool. Dafny is a verification-oriented programming language that is restricted to sequential proofs. The use of Dafny greatly reduces the proof burden for verifying DaisyNFS, because sequential proofs are well-suited to automation and Dafny's automation is well-developed (in contrast automation for concurrent proofs is still nascent, and would need to be integrated into Perennial to be used for these proofs).

The value of sequential proofs can be seen in the proof-to-code ratio for the transaction system, which is 18×, versus the Dafny proofs which required about 2× as many lines of proof as code. Further evidence can be seen in the incremental development of DaisyNFS, which



(a) Concurrent, crash-safe simulation



(b) Sequential simulation

Figure 6-1: Contrast between verifying a concurrent, crash-safe refinement (left) and sequential refinement (right), for a single procedure running the green-colored transitions. Proving concurrent refinement requires a proof that shows every operation (1) preserves the invariant at all intermediate points, and (2) simulates the abstract specification for the operation at some *linearization point* (the above diagram), as well a similar obligation for crashes (the below diagram). For crash safety, a crash at any time should behave like a specification crash transition. Unlike sequential refinement, the proof must show the invariant holds at intermediate points in order to reason about potential interference with other threads.

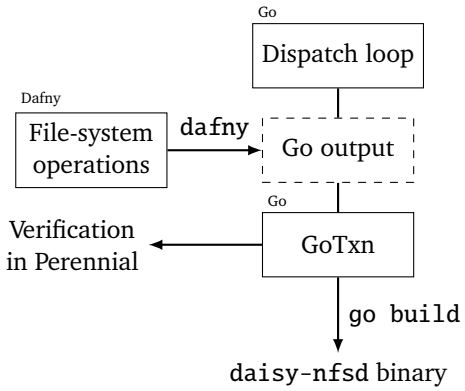


Figure 6-2: The structure of DaisyNFS.

section 9.3 further elaborates on.

Simulation transfer greatly reduces the proof burden for verifying DaisyNFS. There remains the task of actually implementing the file system using transactions. Section 6.5 discusses some key challenges to address, in particular how to ensure transactions are bounded in size (since GoTxn transactions cannot exceed the size of the on-disk circular buffer) and avoiding deadlocks in transactions.

Limitations. The design of DaisyNFS does impose limitations. The proof approach relies on transactions appearing to run sequentially, which prevents modifying state outside the transaction system. An example that would benefit from combining the transaction system with other state is a persistent key-value store, which needs atomic metadata updates but manages data separately to minimize I/O — the proof of such a system would require explicit crash and concurrency reasoning for the data reads and writes, but GoTxn would still be useful for managing the metadata. GoTxn does not have a proof of liveness, and the file-system proof does not show that transactions avoid deadlock. Our NFS implementation lacks some features, such as symbolic links, hard links, and paginated `READDIR`; we believe all of these could be implemented and specified with the same approach but have not done so in our prototype.

6.2 System design

As shown in fig. 6-2, DaisyNFS is implemented in three layers: 1) a dispatch loop that speaks the NFS wire protocol and calls the appropriate method for each operation; 2) a Dafny class that implements each method; and 3) a transaction system that applies the updates of each method to the disk atomically. The dispatch loop is unverified; we assume that the server correctly decodes messages, invokes the verified code with the right arguments, and correctly encodes the response to the client. The middle layer implementing the file-system operations is written and

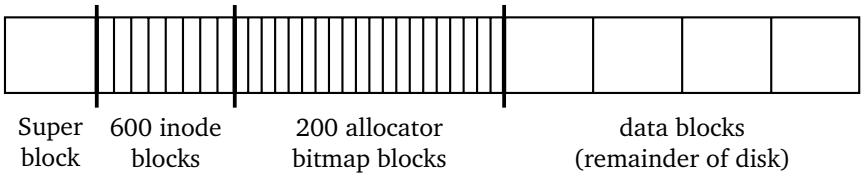


Figure 6-3: The layout of the file system on top of the transaction system's disk. The number of inode blocks and data bitmap blocks are compile-time constants, but easy to change without affecting the proofs.

verified in Dafny, which has a backend for Go. The third layer is directly written in Go and verified using Coq and Perennial. By implementing the file system on top of the transaction system, we can implement each NFS method in Dafny as sequential code calling into a concurrent transaction system library. The NFS operations supported by DaisyNFS are listed in fig. 6-6.

The file system is responsible for implementing files and directories onto an array of disk blocks that is exported by the transaction system. The disk layout used by the file system is shown in fig. 6-3, with regions for inode blocks, bitmap blocks, and data blocks for files and directories. This figure is in terms of the disk exported by the transaction system; the transaction system itself reserves a prefix of 513 blocks for the write-ahead log.

The high-level organization of the file system separates three concerns, each building upon the previous: (1) implementing indirect blocks to support large files; (2) implementing byte-granularity reads and writes on top the block-granularity interface below; and (3) implementing directories by encoding them as files with a special type together with operations to manipulate those files. Section 6.5 explains the internals of the file-system design in more detail, alongside the structure of the Dafny proof.

Recall that GoTxn supports accessing objects smaller than a block. DaisyNFS uses three sizes of objects: bit objects comprise the inode and block allocators, 128-byte objects are used to represent inodes, and full 4KB blocks are used for file and directory data (including indirect blocks). The file system statically allocates regions for these three kinds of blocks, much like ext4.

Acquiring multiple locks during a transaction creates the possibility for deadlocks, for example if two threads acquire two locks in different orders. The two-phase locking implementation does not implement a specific lock acquisition order, leaving it to the file system to avoid deadlock — the most interesting case is `RENAME`, which is discussed in more detail in section 6.5.1.

6.3 Specifying DaisyNFS

The specification for DaisyNFS is a state machine describing an ideal NFS server in the form of an abstract state and a transition for each operation. The implementation of DaisyNFS is a binary

`daisy-nfsd` that implements the NFS protocol, running on top of a disk. Then the DaisyNFS correctness theorem is a *refinement* property, which intuitively says that for any interaction with the implementation, the ideal, atomic NFS state machine could produce the responses; this section shortly gives a more formal definition. As a result a client interacting with the server can pretend that it is the NFS state machine and ignore the complexities of its implementation.

6.3.1 Formalizing NFS

RFC 1813 specifies the NFS protocol, which we make mathematically precise with a state-machine representation defined in Dafny. The formalization requires first defining an abstract state, and then a transition for each NFS operation that specifies how it changes that state and what return values are allowed. While most of the specification is deterministic, some operations have to be specified with non-determinism; for example, we allow returning an out-of-space error in many operations, and the specification allows any timestamp to be picked for the current time. The RFC is precise about arguments and allowed return values, and the text is good about explaining the intended behavior, but it does not separately describe an abstract state to make that behavior mathematically precise. We define the NFS server state as shown in fig. 6-4.

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

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

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

Figure 6-4: Dafny definition of the NFS server state (simplified).

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

The NFS state machine models each operation as a non-deterministic transition that answers when it is allowed for an operation to change the state from `fs` to `fs'` and return `r`. The return value is always wrapped in a `Result` type, which can be either `Ok(v)` for a normal return or an error code for one of the errors defined in the standard. The file system systematically guar-

antees that the state is unchanged when an operation returns an error (though this is stronger than what the RFC mandates); the transaction system makes this easy to achieve by aborting the whole transaction. For example, fig. 6-5 shows the specification for a (hypothetical) `GETSZ` operation that returns the size of the inode `ino`.

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

Figure 6-5: Specification of a hypothetical `GETSZ` operation, a simplification of the real `GETATTR` operation. When `GETSZ_spec(ino, fs, fs', r)` returns true, it is valid when the server receives `GETSZ(ino)` to transition from the state `fs` to `fs'` and return `r`. The return value can be an error (e.g., `r.ErrBadHandle?`), or if `r.Ok?` holds then `r.v` is the `uint64` return value from the operation.

There are four clauses in the specification. The first just says that this operation is read-only. The second is one possible error: if the server returns `ErrBadHandle`, then `ino` is not allocated. The third is a different error, which says this operation returns `ErrIsDir` if passed a directory inode. Finally the fourth clause says that if the operation is successful, it returns the length of the data in `fs[ino]`. Dafny checks several consistency properties of this specification itself; for example, a use of `fs[ino]` only compiles if the specification earlier implies `ino in fs`.

We developed a state-machine model of the regular file and directory operations in NFS in this style, including specifying what certain errors signify. Figure 6-6 lists the entire NFS API and what parts are verified in DaisyNFS. The file system has unverified implementations `FSINFO` and `PATHCONF`, which give the client static configuration information about the file system (for example, the maximum supported write size or the maximum path length). These return constants and thus have no associated proof. DaisyNFS also implements `FSSTAT` to report total and free space, but it does not have a meaningful specification.

DaisyNFS could support some of the remaining operations with some more effort. A symbolic link is essentially a file that holds a path, which is created with `SYMLINK` and can be read with `READLINK`. `MKNOD` similarly creates a new type of special file that consists of a major and minor device number. Specifying these operations would require mostly mechanical changes to the specification to accommodate the new file types. `LINK` is more complicated because in addition to tracking the link count of every file in the state, the specification for `REMOVE` needs to say that the link count is decremented and that the file is deleted if its link count drops to zero.

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

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

The current implementation of `READDIR` always returns the entire directory, which is impractical for large directories; NFS supports a paginated API, where `READDIR` returns some directory entries and then an offset to resume iteration. The complication with this API is that the directory could change between reads, so the interface has provisions to handle this kind of *iterator invalidation* and the specification would need similar adaptations. We believe it would be possible to adapt the work in SibylFS [82], a specification for the POSIX file-system API, which gives a specification for the analogous `readdir` system call. DaisyNFS does not support `READDIRPLUS`, which differs from `READDIR` only in that it also returns attributes along with names and inode numbers.

6.3.2 Specifying correctness for DaisyNFS

The previous section (section 6.3.1) defines the transition system for NFS, which this section relates to the actual DaisyNFS implementation in order to define correctness. The definition uses the ideas from sections 5.2.1 and 5.2.2, namely the definition of refinement and the definition of $\texttt{Go}\langle X \rangle$, where X is instantiated separately with both the Txn and NFS layers. $\texttt{Go}\langle \text{NFS} \rangle$ has a primitive for each operation supported by DaisyNFS, and the semantics of these primitives is defined to be the NFS transition system written in Dafny. Implicit in such a semantics is that each operation is atomic both with respect to other threads and on crash.

With the NFS layer we can model a *server loop* that recovers the file-system state, then repeatedly accepts a new request, processes it with the appropriate NFS transition, and sends the reply back. Let us denote this server loop with $s_{\text{NFS}} : \texttt{Go}\langle \text{NFS} \rangle$. Note that when this (abstract) server processes an NFS operation, it does so by definition atomically, and its state is again by definition preserved on crash. s_{NFS} can be viewed as an abstraction of the following pseudo-code that represents the essence of the real `daisy-nfsd` binary:

```
1  // this is the core of daisy-nfsd
2  func main() {
3    tx := txn.Recover()
4    fs := filesys.Recover(tx)
5    for {
6      req := GetRequest()
7      go func() {
8        switch req.Op {
9        case CREATE:
10         ret := fs.CREATE(req.Args)
11         SendReply(req, ret)
12       case LOOKUP:
13         ret := fs.LOOKUP(req.Args)
14         SendReply(req, ret)
15       // ... other cases ...
16       }
17     }()
18   }
19 }
```

This code starts by recovering the state of the system, starting with the transaction system on line 3 and then continuing on to the file-system state on line 4. Then it repeatedly accepts new requests from the network, abstracted with `GetRequest()` (including parsing the NFS wire protocol). These requests are each processed in a background thread due to the goroutine spawned on line 7. The processing for each request dispatches to the appropriate file-system operation (e.g., lines 10 and 13). The implementations of these operations are compiled from Dafny to Go and then linked with the transaction system.

The term s_{NFS} represents the abstract version of this loop where, for example, the entire call to `fs.CREATE` is an atomic primitive. In the Dafny code, this corresponds to an entire method that calls the GoTxn API. We can think of each method as having a corresponding GooseLang term that represents its implementation at the Txn layer of abstraction, for example *CREATE* : $\text{Go}\langle\text{Txn}\rangle$ might model the `fs.CREATE` method. These constructions are all for the sake of the proof argument; it is part of the assumptions of the proof that every method can be modeled using GooseLang. Next, let s_{dfy} denote the same overall server dispatch loop with methods at the GoTxn abstraction level. Finally, the lowest-level model of the server is derived by combining the Dafny implementation with the actual GoTxn implementation, which we write as $\text{link}(s_{\text{dfy}}, \texttt{txn})$.

$\text{link}(s_{\text{dfy}}, \texttt{txn})$ and s_{NFS} are both models of the `daisy-nfsd` server binary's behavior. The former is a model of its implementation running on top of a disk, while the latter is its specifica-

tion at the level of NFS methods. These two together are enough to state the overall DaisyNFS correctness theorem:

Theorem 6.1 (DaisyNFS correctness)**.** link($\mathtt{s}_{\text{dfy}}$, `txn`) $\sqsubseteq$ $\mathtt{s}_{\text{NFS}}$; that is, the DaisyNFS code follows the NFS specification. Initialization requires initializing the transaction system on an empty disk and then running the `Init` constructor for the file-system. Subsequently the system boots by recovering the transaction system and then the file system with its `Recover` constructor.

This theorem connects the server's implementation at the disk layer, link($\mathtt{s}_{\text{dfy}}$, `txn`), which combines a model of the Dafny code with the GoTxn implementations, to the most abstract version of the server $\mathtt{s}_{\text{NFS}}$ which by definition atomically processes operations, preserves data on crash, and follows the NFS state machine written in Dafny. Section 6.4 explains how this specification is proven. The basic idea is that the Dafny reasoning relates $\mathtt{s}_{\text{dfy}}$ to $\mathtt{s}_{\text{NFS}}$ with a sequential proof, and GoTxn guarantees that sequential reasoning for transactions is sufficient to guarantee the whole system satisfies concurrent, crash-safe refinement due to the atomicity of every transactions.

6.4 Verifying DaisyNFS using Dafny

A key contribution of DaisyNFS is a proof structure that isolates concurrency and crash reasoning to the transaction system. The proof works in two steps. First, a *simulation transfer* theorem extends GoTxn's transaction refinement to show how transactions enable sequential reasoning for an arbitrary system implemented on top (section 6.4.1). Second, simulation transfer is applied to DaisyNFS and its sequential reasoning carried out in Dafny (section 6.4.2).

6.4.1 Simulation transfer

The GoTxn specification from section 5.2 uses transaction refinement to capture that transactions are atomic in the sense that every interleaved execution has a corresponding execution where each transaction runs all at once. This section describes a *simulation transfer* theorem, proven in Coq, that uses transaction refinement to show how GoTxn enables *sequential reasoning* for a system implemented with transactions on top.

The idea behind the simulation transfer specification is to express that a system verified using sequential reasoning for each transaction is also correct when run concurrently through GoTxn — intuitively, this follows from the atomicity provided by transaction refinement. To make this precise, we define formally what we mean by "sequential reasoning". Suppose we have an implementation of layer S using operations from T. The implementation i consists of a function $i(op) : \mathtt{Go}\langle\mathrm{T}\rangle$ for each operation $op \in S$. The statement seq_refinement$\langle T, S\rangle(i)$ says that i is a correct sequential implementation of S using T. The effect of each operation is

given by a transition relation $\sigma \overset{op}{\rightsquigarrow} \sigma'$ that is part of the layer S. To specify correctness under crashes, the definition of sequential refinement refers to $\text{crash}(\sigma, \sigma')$, which is a layer-specific crash transition that models, for example, clearing the contents of memory.

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

$$\{R(\sigma)\}\ i(op)\ \left\{\exists \sigma'.\, R(\sigma') \wedge \sigma \overset{op}{\rightsquigarrow} \sigma'\right\},$$

(2) $\text{init}(\sigma_S, \sigma_T)$ implies $R(\sigma_S, \sigma_T)$, and
(3) if $R(\sigma_S, \sigma_T)$ holds and $\text{crash}(\sigma_T,\ \sigma'_T)$, then there exists a σ'_S such that $R(\sigma'_S, \sigma'_T)$ and $\text{crash}(\sigma_S, \sigma'_S)$.

Conditions (1) and (2) in this definition are standard for sequential verification of refinement, while condition (3) is a condition for sequential crash-safety [11, 86]. Though condition (3) requires the abstraction relation to be preserved by crashes, the proof engineer does *not* have to reason about crashes in the middle of operations. The diagram in fig. 6-1b depicts the main refinement condition (1) diagrammatically.

The simulation transfer theorem takes a proof of *sequential* refinement conditions for a system implemented using transactions and derives a *concurrent and crash-safe* refinement. Like transaction refinement, it is stated as a theorem about an arbitrary program $p : \texttt{Go}\langle \mathit{Sys}\rangle$ using the specification-level API. To model that program using the transactions given by implementation i, we write $\text{link}(p, \texttt{atomically} \circ i)$. The function $(\texttt{atomically} \circ i)(op) = \texttt{atomically}\,\{\, i(op)\,\}$ wraps the implementation $i(op)$ in an `atomically` block. Physically this `atomically` block corresponds to running code in a pattern like the following, written using Go's support for generics:

```
type txnBody[T any] func(tx *Txn) (T, bool)
func runTxn[T any](f txnBody[T]) (v T, ok bool) {
  tx := Begin()
  v, ok := f(tx)
  if ok { tx.Commit() } else { tx.Abort() }
  return v, ok
}
```

In order for simulation transfer to work, every transaction must satisfy some conditions to ensure atomicity. We write $\text{safe}(i(op))$ to say that $i(op)$ is a valid transaction. The main restriction is that $i(op)$ cannot access global state such as the heap, since the transaction system does not make such accesses atomic; further details are discussed in section 5.2.3.

Theorem 6.2 (Simulation transfer)**.** Let *Sys* be a layer implemented using transactions with $i : Sys \to \texttt{Go}\langle\text{Txn}\rangle$, such that seq_refinement$\langle\texttt{Go}\langle\text{Txn}\rangle, Sys\rangle(i)$ and $\forall op.\,\text{safe}(i(op))$ hold. Then

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

The theorem is about a program p using some API *Sys*, such as the server loop for DaisyNFS seen in section 6.3.2, and an implementation i of this API. The conclusion is a refinement that uses p as the *specification*; recall that the semantics of p defines all of the *Sys* operations to be atomic at this layer. The left-hand side is the executable code for this program, derived by composing two functions: first $\text{link}(p, \texttt{atomically} \circ i)$ takes each abstract operation op in p and replaces it with $\texttt{atomically}\,\{\,i(op)\,\}$, and second $\text{link}(\text{link}(p, \texttt{atomically} \circ i), \texttt{txn})$ takes the result of this process and further replaces $\texttt{atomically}\,\{\,i(op)\,\}$ with executable code that uses the GoTxn implementation for each call to the Txn API and the `runTxn` pattern above to make the snippet atomic. The overall refinement says that this executable code has a subset of behaviors of p, so that each operation is not only atomic but follows the abstract specification of the *Sys* layer.

Simulation transfer is stated and proven in Coq. The proof builds on top of transaction refinement. For every execution of the code, transaction refinement promises a corresponding atomic execution of $\text{link}(p, \texttt{atomically} \circ i)$, and the sequential refinement assumption is enough to show that the transactions correctly simulate p at the *Sys* abstraction level.

6.4.2 Using simulation transfer with Dafny

Simulation transfer reduces verifying DaisyNFS's correctness to reasoning about the transaction that implements each NFS operation. Because this reasoning is sequential, we carry it out in Dafny [61], a verification-oriented programming language. Let $i_{\text{NFS}} : \text{NFS} \to \texttt{Go}\langle\text{Txn}\rangle$ denote a model of the Dafny implementation of each NFS operation, as a program using the GoTxn interface. Note that this is merely a hypothetical model of the Dafny code, since Goose does not support enough of Go to explicitly construct these terms in GooseLang. The Dafny annotations on these methods prove sequential refinement conditions that we will denote $\text{seq\_refinement}_{\text{dfy}}(i)$, as illustrated in fig. 6-9. This sequential refinement is indexed by "dfy" to indicate that it refers to what is proven in Dafny. It is intended that this obligation captures seq_refinement(i) from the simulation-transfer theorem, but using a formal distinction allows to say that the connection between these definitions is trusted. Dafny checks the following lemma that i_{NFS} is correct, which will eventually be used to show the overall DaisyNFS correctness theorem:

Lemma 6.3. $\text{seq\_refinement}_{\text{dfy}}(i_{\text{NFS}})$ holds, as checked by Dafny.

```
class Filesys {
  // the external abstract state
  ghost data: FilesysData;
  var txs: TxnSystem;

  // abstraction relation that relates txs.disk to data
  predicate Valid() { ... }

  // see text for these signatures
  constructor Init() { ... }
  constructor Recover() { ... }

  method GETATTR(ino: uint64) returns (r: Result<Attrs>)
    requires Valid() ensures Valid()
    ensures GETATTR_spec(old(fs), fs, ino, r) { ... }

  // ... methods for other NFS operations ...
}
```

Figure 6-7: Outline of how the Dafny proof is expressed.

We now state more formally what seq_refinement$_{\text{dfy}}$ means. The Dafny code is implemented as a class with ghost variables for its abstract state and an invariant that expresses the proof's refinement relation R between the abstract state and its concrete state (which is limited to constants and the transaction system's logical disk). The top-level interface is outlined in fig. 6-7. Condition (1) in the definition of sequential refinement is encoded exactly in Dafny, using `requires` and `ensures` clauses. Condition (2) for initialization and condition (3) for crashes relate to the Dafny code in a slightly more subtle way.

To describe how Dafny encodes this form of refinement, we start by describing its specification transition system in more formal detail. The specification transition system is a triple $S = (\Sigma, \mathbb{M}, \delta)$ where $\sigma \in \Sigma$ is a set of states, $m \in \mathbb{M}$ is a set of operations,[2] and δ is a relation, where $\delta(\sigma, m, \sigma', v)$ is true when in state σ operation m can transition to σ' and return v. To simplify the notation, the transition system has a special operation $\mathsf{crash} \in \mathbb{M}$ to represent a crash transition.

To define S, we build on a subset of its transitions $\tilde{S}$ that only defines the methods of the Dafny class. A state $\sigma \in \tilde{\Sigma}$ of this sub-transition system consists of the values for all the ghost variables that define the API; in the case of the file system this is just the `data` field of type `FilesysData`, which was defined in fig. 6-4. Each NFS method and its arguments is an operation $m \in \tilde{\mathbb{M}}$, and predicates like `GETSZ_spec` define the transition relation $\tilde{\delta}$. Building on $\tilde{S}$, we can

[2] we use the metavariable m to suggest that these are methods

now define S by incorporating initialization and recovery, which are implemented in Dafny as *constructors* `Init` and `Recover`. Constructors are needed to set up the system's global constants, and to initially establish the abstraction relation. Below we define S in terms of $\tilde{S}$. To write down the definition of δ, we use $\sigma \xrightarrow[v]{m} \sigma'$ to mean that $\delta(\sigma, m, \sigma', v)$ is true, and let δ implicitly be false elsewhere if not specified.

$$
\begin{aligned}
\Sigma &\triangleq \mathsf{UnInit} \mid \mathsf{Running}(\sigma : \tilde{\Sigma}) \mid \mathsf{Crashed}(\sigma : \tilde{\Sigma}) \\
\mathbb{M} &\triangleq \mathsf{Init} \mid \mathsf{Recover} \mid \mathsf{Method}(m : \tilde{\mathbb{M}}) \\
\delta &\triangleq \mathsf{Running}(\sigma) \xrightarrow[v]{\mathsf{Method}(m)} \mathsf{Running}(\sigma') \text{ when } \sigma \xrightarrow[v]{o} \sigma' \text{ (in } \tilde{\delta}) \\
&\quad \mathsf{UnInit} \xrightarrow{\mathsf{Init}} \mathsf{Running}(\sigma_0) \\
&\quad \mathsf{Crashed}(\sigma) \xrightarrow{\mathsf{Recover}} \mathsf{Running}(\sigma) \\
&\quad \mathsf{Running}(\sigma) \xrightarrow{\mathsf{crash}} \mathsf{Crashed}(\sigma') \text{ when } \sigma \xrightarrow{\mathsf{crash}} \sigma' \text{ (in } \tilde{\delta}) \\
&\quad \mathsf{UnInit} \xrightarrow{\mathsf{crash}} \mathsf{UnInit} \\
&\quad \mathsf{Crashed}(\sigma) \xrightarrow{\mathsf{crash}} \mathsf{Crashed}(\sigma)
\end{aligned}
$$

Figure 6-8: Defining the specification transition system S for the Dafny code, in terms of $\tilde{S}$ which only defines the transitions for the methods.

The transition system S embeds $\tilde{S}$ in the transitions over $\mathsf{Running}(\sigma)$, and these are the usual transitions during normal operation. However, the Dafny code has to transition into the $\mathsf{Running}(\sigma)$ state in two places: first the Init operation starts the system in state σ_0, and following a crash, the system generally transitions from $\mathsf{Running}(\sigma)$ to $\mathsf{Crashed}(\sigma)$. The system has to be restored to its usual running state in order to use any methods.

In the Dafny code, $\mathsf{Running}(\sigma)$ corresponds to when there *is an instance* of the `Filesys` class. The formal Init and Recover operations are special insofar as they are implemented as *constructors* `Init` and `Recover`, which set up the system's global constants and create the instance in the first place.

The Dafny code implements the initialization using a `NewTxnSystem` method that creates a fresh instance of GoTxn and sets up its initial object schema, as described in section 5.5.1. The `Init` constructor establishes the class invariant `Valid()` starting from nothing, implicitly assuming that the disk is all-zero to be ready for initialization. It promises an initial file-system abstract state σ_0 with just a single root inode (the attributes include a time stamp and are thus not fully specified):

```
constructor Init()
  ensures Valid()
  ensures exists attrs0 ::
```

```
        && init_attrs(attrs0)
        && data == map[1 := Dir([], attrs0)]
```

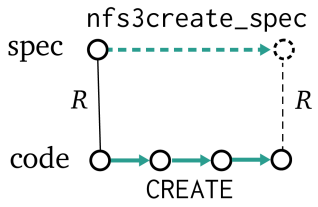
The `Recover` constructor has a more interesting verification story. According to S above, it is permitted to assume that when recovery runs, the system starts out in the state Crashed(σ). The only difference between Crashed(σ) and Running(σ) as far as the simulation is concerned is that in Crashed(σ), the `Filesys` instance is no longer accessible in memory — the system always maintains an abstraction relation `fs.Valid()` that relates the physical state (`txs.disk` in the Dafny code) to the abstract state (the `data` ghost field). The proof of recovery shows it transitions from Crashed(σ) to Running(σ), expressed in Dafny with the following specification:

```
constructor Recover(txs: TxnSystem,
    ghost fs: Filesys)
  requires fs.Valid()
  requires txs.disk == fs.txs.disk
  ensures Valid()
  ensures data == fs.data
```

The specification takes a *ghost* `fs` that satisfies the invariant, as well as an assumption that the transactional disk from GoTxn is the same as the one for the old file system. The former is true since the system is in the state Crashed(σ), and the latter is due to the transaction system's specification defining a crash as a no-op on the logical state. In order to transition to the state Running(σ), recovery returns a fully initialized file system (implicit in the fact that this is a constructor), which satisfies the abstraction relation `Valid()` and has the same underlying state σ as before (written `data == fs.data` in Dafny). The proof of recovery is not that involved, since all that must be verified is that recovery restores the global constants of the file system to their old values, which is implemented by storing them on disk during initialization in a fixed-location file-system super block and reading them back in the recovery procedure.

The methods in Dafny are verified with the usual pre- and post-conditions. By virtue of being methods, they implicitly assume that the system in the state Running(σ); the `Filesys` object must be initialized to call any method on it. Initialization and recovery implicitly require that the user follows the protocol above; specifically initialization must be called only once, and subsequently recovery should be used to restore the previous state. The code tries to check these assumptions: for example, the super block contains a "magic number" (a fixed, initially randomly chosen 64-bit integer) and recovery fails if the super block has the wrong magic number, indicating an attempt to recover from a disk that has not been initialized. Ext4 implements a similar feature to prevent mounting an invalid image.

We can now take simulation transfer and the Dafny proof and combine them to get the overall DaisyNFS correctness theorem:




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

Figure 6-9: Illustration of seq_refinement(i_{NFS}) (left) and its encoding in Dafny seq_refinement$_{\text{dfy}}$(i_{NFS}) (right), for one particular operation. In the diagram, the solid parts are assumed, and the dashed parts must be shown to exist. The complete Dafny spec is more precise about errors.

Theorem 6.1 (DaisyNFS correctness, restated)**.** DaisyNFS atomically implements the NFS protocol, formally stated as the refinement link($\mathtt{s}_{\text{dfy}}$, `txn`) $\sqsubseteq$ $\mathtt{s}_{\text{NFS}}$. Note that $\mathtt{s}_{\text{dfy}} = \text{link}(\mathtt{s}_{\text{NFS}}, i_{\text{NFS}})$. The system must be initialized with the `Init` constructor on an empty disk, and then after every reboot re-initialized with the `Recover` constructor.

This theorem re-states the specification developed in section 6.3.2. Its proof is a simple on-paper argument that connects theorem 6.2 (which *assumes* a sequential refinement) to theorem 6.3 (which *proves* this sequential refinement). Figure 6-10 illustrates the intuition for this theorem in terms of an example execution of $\mathtt{s}_{\text{NFS}}$: the transaction system proof guarantees an atomic execution while the sequential refinement guarantees the transactions themselves are correct.

There are two assumptions needed for the theorems to compose. First, seq_refinement$_{\text{dfy}}$(i_{NFS}) should imply seq_refinement(i_{NFS}), to bridge the premise in simulation transfer and the theorem being proven in Dafny. That is, the encoding of the refinement conditions in Dafny must be correct, but also the semantics of the transaction system operations modeled in Dafny must match the Coq proof. Second, every Dafny transaction must be safe, that is safe($i_{NFS}(op)$) must hold. Safety has a static restriction that transactions should not modify global state, which the Dafny code satisfies because the only mutable state in the file-system Dafny class is the transaction system, so file-system operations cannot make mutations other than through GoTxn. The dynamic restrictions for safety are expressed with preconditions on the GoTxn interface so that Dafny automatically enforces them.

6.5 Verifying the Dafny implementation

Section 6.4.2 explains how DaisyNFS connects sequential verification in Dafny to concurrency and crash safety in GoTxn. This section focuses on the sequential verification and file-system

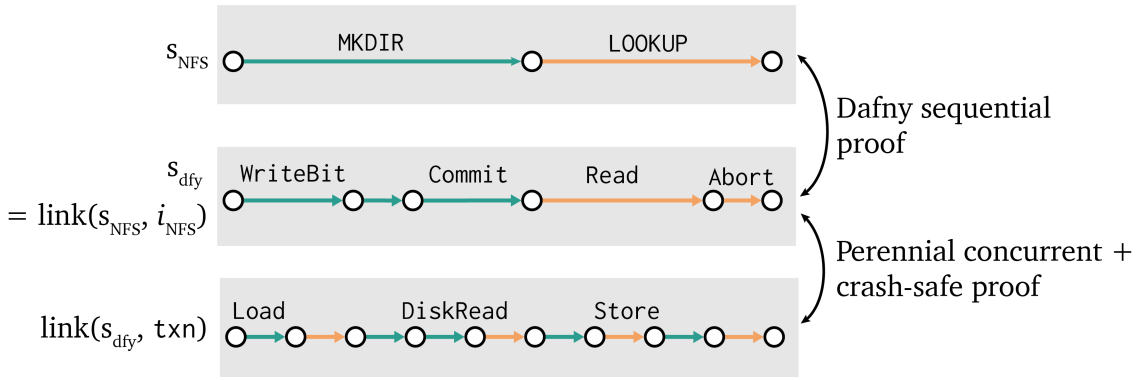


Figure 6-10: Illustration of the DaisyNFS proof strategy in terms of one possible execution of DaisyNFS, receiving parallel `MKDIR` and `LOOKUP` operations, at its three abstraction levels. Operations in each row are color-coded green or orange according to which operation they correspond to (the top-level `MKDIR` or `LOOKUP` respectively). The refinement proof first shows that for every code execution (bottom row), there exists an atomic execution at the Txn layer (middle row), as proven in theorem 5.1. This justifies sequential reasoning to show the transactions on top follow the NFS specification (top row), as proven in theorem 6.3. Putting the two together, theorem 6.1 shows the entire DaisyNFS server atomically implements the NFS specification.

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

Figure 6-11: Layers in the Dafny implementation and proof of the file-system operations.

design themselves.

DaisyNFS is implemented and verified in several layers of abstraction, depicted in fig. 6-11. Each layer is implemented as a class that wraps the lower layer as a field, until finally the transaction system is an assumed interface in Dafny. The `daisy-nfsd` binary implements the NFS wire protocol in unverified Go code and calls the top-level Dafny class and its verified methods to handle each operation.

The layers of the file system can be organized into groups that implement three difficult pieces of functionality: organizing data blocks into metadata and data (the indirect and block layers), translating byte-level operations into block operations (the byte and typed layers), and implementing directories as special files that the file system itself reads and writes (the dir layer). The modularity was essential to complete the proof in manageable chunks (to avoid

overwhelming both the developer and prover), and these intermediate interfaces would have been natural to implement even without verification.

6.5.1 Implementing the file system using transactions

Aspects of the design of DaisyNFS are similar to the file system in xv6 [22], as well as Yggdrasil [86], a verified sequential file system. We also adopt the recursive strategy for implementing and verifying indirect blocks from DFSCQ [54]; recursion simplifies the implementation of triply-indirect blocks, which are needed to reach a reasonable maximum file size of 512GB. Unlike most file systems, DaisyNFS is designed to fit every operation into a transaction in order to support our goal of sequential reasoning. This is a non-standard design and we encountered some unique challenges in doing so. In this section we highlight difficulties in fitting two features into transactions: rename and freeing space from deleted files.

Rename

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

In a rename operation, the source and destination are each specified by a combination of the parent directory inode and name within that directory. Rename has an additional functionality of overwriting the destination if the source and destination are files, or if both are directories and the destination is empty. It is this overwrite-compatibility check that makes deadlock avoidance difficult: it is necessary to lock the source and destination directories first to lookup the source and destination names, but those might be files that are earlier in the inode lock order. We address this in the code by returning an error from the Dafny transaction before the lock order would be violated. The error comes with the set of inodes that should have been acquired. The rename is then re-run with this set of inodes as a lock hint, and they are all acquired in the correct order at the beginning of the operation. The inodes to lock are only a hint and must be compared against the current source and destination, in case those inodes have changed since the last transaction. The overall rename operation runs in a loop until the locking succeeds; the loop is potentially unbounded, but each iteration can only fail due to concurrent renames that involve the same inodes.

At this point it is worth discussing the performance considerations that lead to handling lock ordering in the file system, rather than generically in GoTxn. The transaction system could avoid deadlocks by either enforcing a global order over addresses or by timing-out operations. Enforc-

ing a global order is inefficient for the file system; data blocks will never cause deadlock because the file system only accesses a block after locking the (unique) inode that contains it. Timing-out operations would lead to slow and spurious transaction failures that could more rapidly be avoided in the higher-level code, hence we do not attempt to detect deadlock dynamically.

Freeing space

Freeing space becomes surprisingly tricky with large files. The problem is that a large-enough file may reference too many blocks to be freed in a single transaction. Transactions are bounded by the size of the on-disk log (which can hold 511 blocks), whereas freeing a file requires writing zeros to the block allocator for all of its formerly used blocks to mark them as free. DaisyNFS handles this issue by splitting file removal and space reclamation into separate transactions. The latter is implemented with an operation `ZeroFreeSpace(ino)` which frees and zeros the unused space in an inode that we prove has no effect on the logical file-system state. Because this operation is a logical no-op, it is safe to call it at any time.

The implementation is careful to call `ZeroFreeSpace` after any operation that leaves unused blocks, in particular `REMOVE`, which deletes a file, and `SETATTR`, which can shrink a file by reducing its size. Since `ZeroFreeSpace` doesn't affect the user-visible data, it can return early to avoid overflowing a transaction. The unverified code that manages freeing space runs the operation in a loop until it covers all of the unused space in an inode.

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

The freeing implementation is an interesting example of using validation in verification. The specification for much of the freeing code is loose, allowing any data to be written to the free space. Only the code that checks if the zeroing is done needs to be verified against a strong specification. The rest of the code does still needs to be correct for this check to succeed, but we aren't required to prove it.

6.5.2 Verifying the indirect block implementation

DaisyNFS supports large files using indirect blocks. A file's inode has a fixed number of addresses for block addresses, some of which are used as indirect blocks that hold another layer of addresses rather than direct blocks that have file data. A single level of indirection is insufficient for a practical file system, so DaisyNFS implements support for arbitrarily indirect blocks, and in practice the file system uses up to triply-indirect blocks to support files up to 512GB.

The indirect and block layers together implement an abstraction of a file as a sequence of blocks, hiding the fact that some of the blocks are used as metadata. These sequences are always of the maximum size, and only the next layer reasons about file sizes. To efficiently represent files of the maximum size, the code uses a convention that a zero block address is treated implicitly as encoding a zero block, including for indirect blocks, an idea borrowed Alex Konradi's work on DFSCQ, a sequential verified file system [54]. Indirect blocks are implemented recursively, where a k-indirect block is always treated as containing 512 pointers to $(k-1)$-indirect blocks, and a 0-indirect block contains file data. Zeros are also treated recursively so that a single zero in an inode for the root of a triply-indirect block efficiently stores a whole tree corresponding to many gigabytes of zeros.

An inode has space for twelve 64-bit block addresses, after accounting for space used by its attributes and type information. In principle all of these addresses could be used uniformly as triply-indirect blocks. However, this would create a lot of indirection and lower performance for the common case of small files. Thus instead of organizing them in this way, the code uses a range of indirection, with mostly direct blocks and a handful of indirect blocks. To keep the code general, the indirection level of each of the twelve blocks is given in a global indirect-block configuration, and most of the code is generic over the configuration. We currently configure DaisyNFS with 8 direct blocks, two indirect blocks, a doubly-indirect block, and a triply-indirect block. Just the direct and indirect blocks can address 4 MB fairly efficiently, but the triply-indirect block allows files to be large (up to 512 GB).

Indirect blocks pose a challenge for verification due to the classic problem of *aliasing*. The proof must show that modifying a data block or indirect block has no effect on other files. In the DFSCQ proof, the invariant captures the non-aliasing between files using separation logic, which makes disjointness easy to express. In Dafny we have no such logical technique, so we instead use a standard SMT-friendly trick for the invariant: in addition to the physical mapping that tracks how to dereference a block address, the indirect layer proof tracks a ghost *reverse* mapping that tracks where each in-use block number is stored. The invariant states that the forward and reverse mappings are inverses of each other, which implies that modifying an address only affects its owner and nothing else.

To encode the reverse mapping, the code uses a "position" datatype `Pos` to represent the location of a block within an inode. With indirect blocks, the metadata blocks themselves also

need to be considered locations, since the invariant must also rule out aliasing between metadata and data. A `Pos` encodes an inode, an indirection level, and an offset within that indirection level to uniquely identify where a block is used. If we imagine that an inode's block pointers are organized in a tree, the roots are stored directly in the inode while the leaves are direct blocks. An indirection level which is higher than the leaf level describes a metadata block.

The indirect block proof is split into the indirect and blocks layers. In the indirect layer, the abstract state maps a `Pos` to the 4KB data block stored there, and separately tracks the size and attributes of each inode. The invariant also expresses that the block allocator's used blocks have an associated `Pos` and that the free ones do not. The interface for the indirect layer exposes reads and writes for positions, regardless of whether they are metadata or data blocks. The block layer above instead exposes a map from inode number to a flat sequence of blocks by mapping each leaf position to its linear index within the inode. Separating these two made it easier to work on the indirect layer while exposing a much more natural abstraction of a file as a sequence of blocks for the rest of the file-system implementation.

Chapter 7

Goose, a tool for verifying Go code

This chapter is about Goose, which solves a practical problem of connecting formal reasoning to efficient code. The developer writes code in Go, uses the `goose` translator to convert the code to a model in the Coq proof assistant, then carries out the proof on top of the model in the Perennial logic. Goose encompasses the entire process: it includes the translation tool itself, the way it models Go code, and the reasoning principles for proving properties of translated Go. (We will also use "Goose" in some places to refer to the subset of Go supported by the translation tool.)

Previously chapter 5 discussed the GoTxn proof. GoTxn is implemented in Go; Goose is the system that allows us to verify its implementation. The proof is carried out in the Perennial logic, described in chapter 3. The description of Perennial is language-independent because the program logic can be used to reason about any programming language modeled in Coq. To verify GoTxn, we instantiate Perennial with GooseLang, the language that Goose provides to model Go code.

This description of Goose is written to be accessible to someone relatively new to verification or systems research. The hope is that others can learn from this approach to modeling and verifying code, and apply it to other languages and domains. Some familiarity reading programming language semantics will be helpful to appreciate section 7.4, but fully understanding that section is not needed to understand the overall idea and reasoning principles in the other sections. Section 7.5 gives specifications in separation logic, starting with enough introduction to understand their intuitive meaning.

The chapter is intentionally fairly independent of the rest of the thesis, in case the reader is not interested in the specifics of the Perennial logic or the file system, and since there is no published paper describing the details of how Goose works or its reasoning principles. Concurrency is an important part of the Goose model, though pleasantly enough it doesn't complicate the sequential parts of Goose. Crash-safety concerns do not show up here since crashes are modeled simply by stopping execution and wiping out all of the state except for the disk, which does not

relate to the specifics of Go. Goose is not specific to GoTxn and can handle a much broader range of Go than that single codebase.

7.1 Goals and motivation

There are three main goals for Goose: supporting **efficient** code, **soundness** for the translation process, and **convenient reasoning**. The subset of Go that Goose supports is intended to enable writing and verifying high-performance code. Goose is sound if the model captures the behaviors of the code, which is required for the proofs to say something meaningful about the executable Go code. If the model misses some buggy behavior in the code, then a correctness proof wouldn't mean anything about the code. Finally, Goose aims to make reasoning convenient by developing reasoning principles for the model of Go primitives like structs, slices, and maps.

There are alternate setups for verification where the connection between the proofs and the code is not given by a model and translation. Goose supports efficient code both by connecting to ordinary Go, which benefits from the Go compiler and runtime, and by supporting a variety of features. Sometimes efficient code is more complex to prove, but this is a tradeoff the user can make.

The design of Goose tries to achieve soundness through simplicity and careful choice of what to model. There is sometimes a tradeoff between simplicity and supporting efficient code. If a language feature is needed for good performance (for example, taking pointers to individual struct fields), then Goose models it, even if the feature is complex. If a feature would only result in more idiomatic code and modeling it seems subtle, then it might not be implemented (for example, simple uses of `defer` could be modeled but aren't because the feature is complicated in general). The result is that Goose is generally pleasant and productive enough to write in, but requires some practice for a Go programmer.

Convenient reasoning remains a goal for Goose, but not one that was always achieved. All of the verified libraries are usable but pain points remain; some of these are simply a matter of engineering effort or fixing bugs, but we have also found code patterns that weren't well captured in the reasoning.

7.1.1 Why Go?

Go is a convenient language for building verified systems. It is productive enough to build systems that get good performance. The language is simple, facilitating a sound translation.

For our first goal, efficiency, Go has enough features to build good systems in. It has efficient and useful built-in slice and map data structures. The runtime handles concurrency efficiently and has good support for synchronization using locks and condition variables, allowing a low-level implementation.

There is also an advantage to Go as a programming environment rather than programming language. The tooling for testing, debugging, and profiling is extremely good, making it easy to fix bugs (before verification or in unverified code) and find performance problems while optimizing. We were able to use low-level interfaces to the operating system to access the disk — these are easier to understand in isolation, compared to understanding the combination of a file-system library and the operating system's behavior. Garbage collection simplifies writing code, particularly with concurrency, and carries a relatively low performance impact due to the efficient runtime.

For the second goal, soundness, it helps that Go is a simple language. The Goose translator effectively gives a semantics to the source code; in a complex language this can be a daunting task (such as attempts to formalize JavaScript and Python [37, 76]). It isn't too difficult to give Go a semantics, especially the Goose subset. Go's tooling helped, including libraries for parsing and type-checking Go source code. Not only do these libraries save time in implementing Goose, they greatly improve reliability since they are written by experts (the Go compiler team, extracting code from the compiler itself).

7.1.2 Why not C or Rust?

C is not too different from Go as a basis for Goose, as long as the comparison is to a useful subset of C and not the entire language (as it is with Goose). The main differences are the need to implement and verify manual memory management for proofs on top, and that it would be more challenging to parse and type-check C code.

Using Rust as a source language seems attractive but would likely not be much better than Go from a verification perspective. One subtlety is that while the source code is type checked, the model is an untyped program. It would be difficult to take advantage of the fact that the code is type-checked, and thus the proof engineer would anyway re-prove memory safety in the same way as Goose requires with Go. Recent approaches for verifying Rust like RustHorn [66] and RustHornBelt [67] show promise for giving a model that benefits from Rust's unique type system guarantees. There would still remain the practical benefit of Rust helping prevent memory safety and concurrency bugs, which is better for fixing the code than carrying out the proof and discovering issues during verification.

Another difficulty with Rust would be the size and complexity of the language — the subset supported might be restrictive enough that the experience no longer feels like Rust. For example, the `Vec<T>` type has 152 methods without even including trait implementations. Using any of these methods would require assuming a semantics for it, which is trusted to be sound (that is, getting the semantics wrong could compromise the whole verification). Thus in practice the expansive standard library would mostly not be available; for soundness only a core subset would be modeled and the rest manually implemented and verified.

7.2 Related work

There are several areas of related work. First and foremost Goose is an approach to verifying executable programs, so this section discusses alternate approaches. Second, Goose implicitly gives a semantics to Go, and there are related projects giving semantics to other programming languages.

7.2.1 Verification approaches

At a high level, all verification systems need to solve the problem of connecting the world of the proof assistant to the "real world" that runs the code. The proofs are always over a model of the code, and the theorems are always contingent on the assumption that the running code has been modeled correctly. The approach imposes some requirements on the code that can be written and how proofs are written. Thus each approach makes a tradeoff between how efficient verified code can be, soundness, and convenient reasoning, as outlined above in section 7.1.

There are basically three ways to verify programs (with several interesting caveats). We can translate code to a model (as in Goose), pretty-print a model to code, or compile from a language intended for both verification and implementation. In addition, verified compilation and translation validation can implement parts of the translation process from model to code in a verified way; this can usefully make reasoning more convenient without sacrificing soundness.

It is easiest to understand this characterization in the context of concrete examples. Goose translates from Go code to a model in Coq. Both CFML [14] and hs-to-coq [87] work in a similar way, for OCaml and Haskell respectively. Fiat-Crypto [29] prints a model of C code to a string with source code that is then compiled with a C compiler. Programs in Dafny [61] are typically run using Dafny's built-in compiler (which has backends for C#, Java, JavaScript, and Go at the time of this writing). Similarly, languages like Coq and F* have support for *extraction* that translate functional programs in those languages to something executable like OCaml.

There are approaches that are somewhere in between using built-in compilation and writing code in a model. For example, FSCQ [17] uses Coq extraction, but it works from a model of I/O interaction that is implemented by combining the extracted code with a Haskell library. This combines Coq's extraction (compilation) with translating from a model to code. For F*, there is a specialized toolchain called KaReMeL [78] that targets a subset of F* called Low* and extracts imperative C code.

Verified compilation can reduce trust when using either approach. VST [8] in its most basic form looks like a code-to-model translation, using a tool called `clightgen` to translate C code to an abstract syntax tree (AST) in Coq, which the user can then specify and verify in VST. However, it is then possible to run this AST through the CompCert verified compiler and produce assembly code (or more specifically, a model of assembly represented in Coq). The proof is then over this

assembly model, which is pretty-printed and compiled with an assembler to run it. While the user writes C code, this system only requires trusting the model of assembly. On the other hand, the performance of the code is determined by the quality of the CompCert compiler.

Verified compilation is used in a way similar to VST in Vale [5, 32]. Code is written in a specialized assembly-like language called Vale, with proof annotations. This is then compiled to Dafny or F* (Vale has backends for both) where the proofs are carried out before finally being printed back down to assembly. Vale supports verifying highly efficient assembly code, but this code pays a price in being harder to verify.

Cogent [1, 71] also uses a form of verified compilation known as translation validation to make proofs easier without compromising on soundness. Code written in the Cogent language (an imperative language with linear types) is translated to a model in Isabelle/HOL along with a functional specification for the code and a proof of correspondence. Thus the user can write proofs on top of the functional specification, but the proofs are about an imperative model of the code.

7.2.2 Modeling programming languages

Goose gives a semantics to a subset of Go. Each function in Go is translated to a term in GooseLang, which has a semantics in Coq in the form of a transition system. Giving a semantics to a real-world language is interesting in its own right, not just for verifying code in that language. Many similar semantics efforts are focused on the goal of completeness, in order to understand how all of the language features interact.

Featherweight Go [35] gives a semantics to a core subset of Go, for the purpose of reasoning precisely about adding *generics* to Go (this paper influenced the final design for Go's generics, which were added to Go 1.18 and released in March 2022). The purpose of this paper is quite different from Goose, since it aims mainly to reason about Go's type system rather than model specific programs.

CH20 [57] is a fairly complete formalization of the C11 standard. CH20 defines C in operational and axiomatic styles, along with an executable semantics to test the semantics on (small) concrete programs. All of these semantics are formally related to each other. Having multiple, related semantics helps give confidence in each of them, since they independently express the same thing in different ways.

VST [8] uses CompCert C to model C code. One similarity to Goose is support for structs in terms of their individual fields. Both systems need to model the semantics of features like taking a pointer to an individual struct field (as CH20 also handles). What sets VST and Goose apart from CH20 is to also have reasoning principles for verifying code that uses structs in interesting ways, such as structs where a subset of fields are protected by a lock.

RustBelt [49] is intended as a model of Rust code, at the Rust MIR (mid-level IR) level of

abstraction. The modeling language in RustBelt, λ_{Rust}, has many similarities to GooseLang, and both are used together with Iris [48] and the Iris Proof Mode [58]. RustBelt has different goals, being intended for reasoning about Rust as a whole rather than program verification. This is a somewhat subtle difference: the goal in RustBelt is to prove properties about *all* λ_{Rust} programs, whereas we only prove correctness of *particular* GooseLang programs. As a result it is important for λ_{Rust}, together with its type system, to rule out programs that Rust does not allow, whereas GooseLang is more expressive than Go with unsafe features like pointer arithmetic. Because it has different goals, RustBelt also does not have an automatic translation from Rust to λ_{Rust}; instead, some important libraries have been translated by hand and verified using Iris.

Finally, WebAssembly is a rare example of a production language with a formal semantics [38], and moreover it was designed with the formal semantics in mind.

7.3 High-level overview

The goal of the translation is to model a Go program using GooseLang, which is a programming language defined in Coq for this purpose. That is, GooseLang is defined using expressions defined in Coq, equipped with a semantics also given in Coq. Figure 7-1 gives a graphical depiction of how Goose and GooseLang fit together. Section 7.4 gives a detailed and formal description of GooseLang. Since GooseLang programs support references to model the Go heap, the semantics is written in terms of transitions of (expression, heap) pairs where the heap maps pointers to values. The intention of the translation is that the semantics of the translated function should cover all the behaviors of the Go code, in terms of return values and effect on the heap. As long as this is true, a proof that the translated code always satisfies some specification means that the real running code will, too.

GooseLang is a low-level language, so many constructs in Go translate to (small) implementations in GooseLang. This implementation choice proved to be much more convenient than adding primitives to the language for every Go construct. For example, a slice is represented as a tuple of pointer, length, and capacity, and appending to a slice requires checking for available capacity and copying if none is available. Appending to a slice is a complicated operation, and it was easier to write it correctly as a program rather than directly as a transition in the semantics. The one cost to this design strategy is that an arbitrary GooseLang program is much more general than translated Go programs — for example, GooseLang has support for pointer arithmetic. This has no impact on verifying any *specific* Go program.

The extra generality of GooseLang is a downside when a theorem talks about an *arbitrary* GooseLang program, as shows up in the transaction system's transaction refinement and simulation-transfer theorems. In order to make these theorems true, they must rule out some ill-defined GooseLang programs, which are not possible to produce by writing Go and translat-

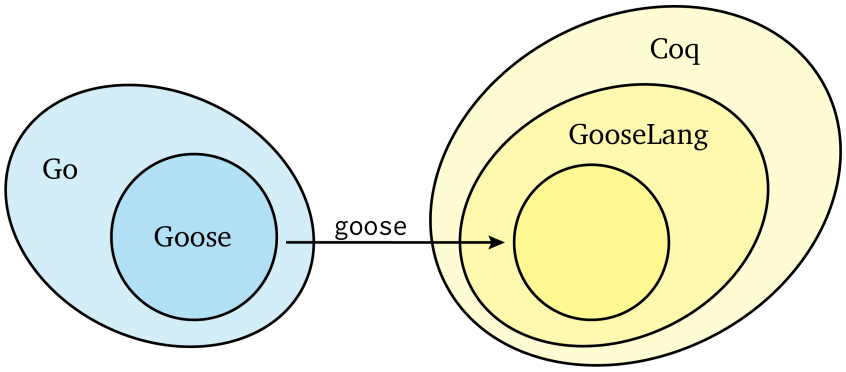


Figure 7-1: Pictorial representation of Goose's translation. Goose supports a subset of Go, which it translates to GooseLang. The target is more general, so not all of GooseLang can be produced from Go code. GooseLang itself is embedded in the Coq proof assistant.

ing it to GooseLang. Both specifications make these assumptions using a standard technique of using a type system for GooseLang developed in Coq, and encoding syntactic restrictions via that type system.

An important aspect of GooseLang is supporting interactive proofs on top of the translated code. The interactive proofs use separation logic, a variant of Hoare logic, so specifications describe the behavior of each individual function. In order to support verification of any translated code, GooseLang comes with a specification for any primitive or function that the translated code might refer to, including libraries like slices used to model more sophisticated Go features. GooseLang has many "pure" operations that have no effect on the heap, due to many primitive data types and operations (for example, there are 8-, 32-, and 64-bit integers, and arithmetic and logical operations for each). The specifications for these operations are handled with a single lemma, which is applied automatically with a tactic `wp_pures`.

Since our goal is to support interactive rather than automated proofs, it is helpful to make the model simple to work with. The translation process maintains a strong correspondence between the model and source code: each Go package translates to a single Coq file, and each top-level declaration in the Go code maps to a Gallina definition (a GooseLang constant or function). Goose has a special case for translating immutable variables to let bindings in GooseLang (rather than allocating a pointer that will only be read). As a result, factoring out a sub-expression to a variable has little impact on proofs.

The translation process sometimes translates a Go operation to a sophisticated model in order to capture some corner-case behavior, for example in the model of slices described in section 7.5.4. These complex models don't have an undue affect on proofs as long as Goose's reasoning principles can abstract away the complexity for common cases. Section 7.5 walks through several features of Go. Each subsection first describes a model that implements the feature in GooseLang, which primarily aims to be faithful to Go. Next, the subsection describes

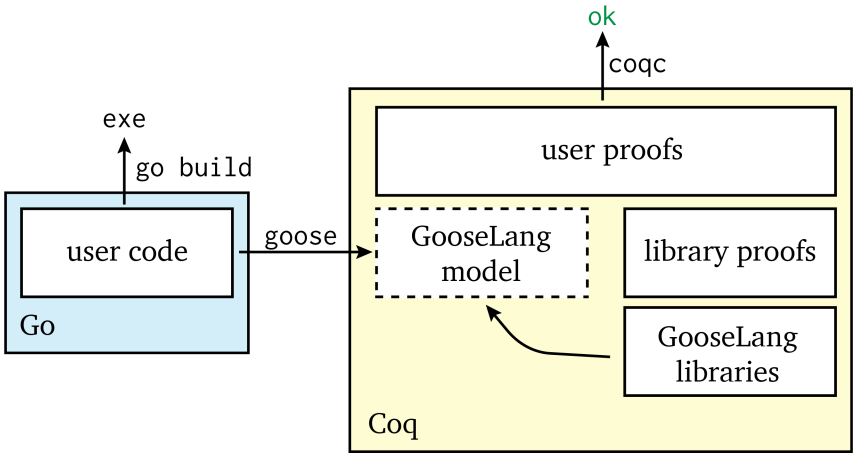


Figure 7-2: Workflow for verifying code using Goose. The user writes the code and proofs on top of the generated model. The model references libraries that model features of Go, which come equipped with proof libraries to make proofs easier. The code is ordinary Go that can be compiled, tested, and debugged with the usual Go tools. The proofs are all checked by the Coq proof assistant.

reasoning principles we developed for that feature, in the form of separation logic assertions (for example, to represent a slice) and Hoare triples (for example, to specify the behavior of the `append()` operation). The goal for the model is to capture Go's behavior, whereas the reasoning principles aim to make proofs using the model practical. See fig. 7-2 for an overview of how the code and proofs fit together when using Goose.

7.4 GooseLang syntax and semantics

GooseLang is an effectful, untyped lambda calculus, with mutable references and concurrency. Defining the language involves defining a *syntax* in the form of expressions (section 7.4.1), and a *semantics* for how those expressions execute in the form of a transition system (section 7.4.2). The transitions also require some state, for example to track the value stored at each allocated pointer. This section focuses on the low-level aspects of the language itself. Section 7.5 discusses how Go features are modeled on top of GooseLang, and the reasoning principles for those models.

The basic unit of GooseLang is the expression e, which supports features such as recursive functions rec $f(x) = e$ and data types like tuples (e_1, e_2). GooseLang has a semantics for these expressions in terms of the reduction relation $(\sigma, e) \rightsquigarrow (\sigma', e')$ which says that e *reduces to* (or "executes to") e' in the state σ, and the resulting state is σ'. This reduction relation is lifted to describe not just a single expression but to a whole *threadpool*, a list of concurrently executing expressions. One can think of each expression in the threadpool as representing the

computation of a single thread, and each reduction step as modeling running the thread for one transition. The semantics determines what primitive operations are considered atomic for the purpose of concurrent reasoning. For a reader unfamiliar with the lambda calculus and formal presentations of its syntax and semantics, one can find a detailed introduction in the textbook *Types and Programming Languages* [74, Chapter 5].

7.4.1 GooseLang Syntax

Syntax

$$
\begin{array}{llll}
 & x, f & \in & \mathit{Var} \\
 & \ell & \in & \mathit{Loc} \\
 & n & \in & \mathit{U64} \cup \mathit{U32} \cup \mathit{U8} \\
 & s & \in & \mathit{String} \\
 & \circledast & ::= & + \mid - \mid * \mid = \mid < \mid \ldots \\
 & \ominus & ::= & - \mid \mathsf{ToString} \mid \mathsf{ToU64} \mid \ldots \\
\mathit{Val} & v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \ell \mid s \\
 & & \mid & \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid \mathsf{rec}\, f(x) = e \\
\mathit{Exp} & e & ::= & x \mid v \mid e\, e \mid \mathsf{rec}\, f(x) = e \\
 & & \mid & \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
 & & \mid & \mathsf{Fork}(e) \\
 & & \mid & (e, e) \mid \pi_1\, e \mid \pi_2\, e \\
 & & \mid & \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid \mathsf{case}\, e\, \mathsf{of}\, \mathsf{inj}_1\, x \Rightarrow e\, \mathsf{or}\, \mathsf{inj}_2\, x \Rightarrow e \\
 & & \mid & e \circledast e \mid \ominus e \mid \mathsf{ArbitraryU64} \\
 & & \mid & \mathsf{AllocN}(e, e) \mid \mathsf{CmpXchg}(e, e, e) \mid \mathsf{Load}(e) \\
 & & \mid & \mathsf{PrepareWrite}(e) \mid \mathsf{FinishStore}(e, e) \mid \mathsf{StartRead}(e) \mid \mathsf{FinishRead}(e) \\
 & & \mid & \mathsf{Panic}(s) \\
 & & \mid & \langle \mathsf{External} \rangle
\end{array}
$$

Derived forms and notation

$$
\begin{array}{rcl}
\lambda x.\, e & \triangleq & \mathsf{rec}\, \_(x) = e \\
\lambda x, y.\, e & \triangleq & \lambda x.\, \lambda y.\, e \\
\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 & \triangleq & (\lambda x.\, e_2)\, e_1 \\
e_1;\, e_2 & \triangleq & \mathsf{let}\, \_ = e_1\, \mathsf{in}\, e_2 \\
\mathsf{ref}\, e & \triangleq & \mathsf{AllocN}(1, e) \\
!x & \triangleq & \mathsf{Load}(x) \\
\mathsf{Store} & \triangleq & \lambda x, e.\, \mathsf{PrepareWrite}(x); \mathsf{FinishStore}(x, e) \\
x \leftarrow e & \triangleq & \mathsf{Store}\, x\, e
\end{array}
$$

Figure 7-3: GooseLang syntax

The syntax for GooseLang programs is given in fig. 7-3. The important top-level definition is $e \in \mathit{Exp}$, giving expressions in GooseLang. Before getting to the expressions, it's worth noting that the values in GooseLang (the definition of *Val*) are designed for Go's primitive data types,

in particular with first-class support for bytes and 32-bit and 64-bit unsigned integers. Our code does not use signed integers and so Goose does not model them, but adding them would simply require extending the type of literals and adding several pure operations to implement various integer conversions.

The basic lambda calculus primitives are given on the first line: variables x, values v, function application $e\,e$, and (recursive) functions rec $f(x) = e$. The first few derived forms encode more basic primitives like a non-recursive lambda, let bindings, and sequencing, all on top of recursive functions. The language has two types of composite data: products (used pervasively to model structs), and sums (used only to model maps). There is also if c then e_1 else e_2 as the main control-flow primitive, and Fork to create concurrent threads. Finally, there are many primitives for the heap, which are discussed in more detail later.

This section builds intuition for both the syntax and semantics of GooseLang by presenting some very simple Go programs and their translations into GooseLang. First, let's look at some *pure* functions that don't use pointers:

```
func Midpoint(x uint64,
              y uint64) uint64 {
  return (x + y) / 2
}
```

$$\text{Midpoint} \triangleq \lambda x, y.\,(x + y)/2$$

```
func Max(x uint32,
         y uint32) uint32 {
  if x > y { return x }
  return y
}
```

$$\text{Max} \triangleq \lambda x, y.\;\; \text{if } x > y \text{ then } x \text{ else } y$$

```
func Arith(a uint64,
           b uint64) uint64 {
  sum := a + b
  if sum == 7 { return a }
  mid := Midpoint(a, b)
  return mid
}
```

$$\begin{aligned}&\text{Arith} \triangleq \lambda a, b.\\ &\quad \text{let } sum = a + b \text{ in}\\ &\quad \text{if } sum = 7 \text{ then } a \text{ else}\\ &\qquad \text{let } mid = \text{Midpoint } a\ b \text{ in}\\ &\qquad mid\end{aligned}$$

Notice that the translation maps each Go function to a GooseLang definition. For readability in this thesis the GooseLang definitions are written mathematically; what the Goose tool emits is a Coq file where each definition is a Gallina term of type `expr` (the Coq type of GooseLang

expressions). This is what allows definitions to refer to each other, such as how Arith calls the previously defined Midpoint.

The next interesting feature of GooseLang is support for pointers. In this aspect the language is a bit unusual in order to model concurrency soundly, described in more detail in section 7.5.1. For now, it is sufficient to think of $x \leftarrow e$ as the usual pointer store operation, and $!x$ as the usual load. Here are a couple examples to illustrate:

```
func Swap(x *uint64,
          y *uint64) {
  tmp := *x
  *x = *y
  *y = tmp
}
```

$$\mathsf{Swap} \triangleq \lambda x, y.\ \mathsf{let}\ tmp = !x\ \mathsf{in}\ x \leftarrow !y\ y \leftarrow tmp$$

```
func NewPtr() *uint64 {
  return new(uint64)
}
```

$$\mathsf{NewPtr} \triangleq \lambda \_.\ \mathsf{ref}\ 0$$

The NewPtr definition deserves some explanation. First, even though the Go code takes no arguments, the GooseLang expression takes an unused argument; this is so that NewPtr is syntactically a function, a requirement of the way GooseLang is encoded in Coq.[1] Second, this definition allocates a pointer with ref and gives it an initial value of 0. Go promises that this initial value is the "zero value" of the appropriate type, `uint64` in this case.

The CmpXchg operation in GooseLang is special in that it is not directly exposed by the translator. It is used to model locks, since it is the only way to safely coordinate threads in GooseLang. Section 7.5.2 explains how locks are modeled, including the implementation of the lock model that uses CmpXchg.

The Goose syntax includes an ⟨External⟩ alternative. The language, both syntax and semantics, are parameterized by an external interface of operations for interacting with the outside world. For example, in GoTxn the code uses an interface of external disk operations, presented separately in fig. 7-5. The parameterization is primarily useful for refinement proofs that relate two instantiations of Goose with different sets of external operations (for example, we use this to specify GoTxn). Other than that use case, the reader can imagine that the disk's external rules are simply part of the definition of expressions and the semantics.

[1] For expert readers, top-level functions are translated to GooseLang *values*. Values in GooseLang are always closed, to avoid difficulties with capture-avoiding substitution.

7.4.2 Semantics of GooseLang

Goose also has a mathematical, small-step, operational semantics; this style of specification and indeed most of the semantics is quite standard for languages of this type. The full details are presented in fig. 7-4.

Structure of the semantics The semantics is defined in three relations: the first, the *pure reduction relation* $e \overset{\text{pure}}{\rightsquigarrow} e'$, is the easiest to understand because it describes "pure" expressions that do not depend on any external state. The semantics of such expression is simply to "reduce" to simpler expressions, eventually reaching values. The semantics of function application is pure. We use $e[v/x]$ as notation for *substituting* the expression v for the variable x in e (it might be pronounced "e with x for v", or "e with v over x" to directly suggest the notation). To implement recursion, the function definition is itself substituted for its name.

The next relation describes the semantics of the heap operations. This relation is over $((h, w), e)$. h is the heap and e is the expression being executed; w is the type of an external "world" that is used to define the semantics of external operations. Even the type of this world state is a parameter at this level, and is defined as part of defining a set of external operations and their semantics. The specific case of the disk is presented in fig. 7-5. This relation is called the *head reduction relation* because it only gives the semantics of expressions where the relevant operation is at the top level or "head" of the expression.

The third and final relation gives the full GooseLang semantics, a relation over $((h, w), \mathcal{E})$. Instead of a single expression, this relation now transitions between *threadpools* $\mathcal{E}$, which are lists of expressions that are running concurrently. The semantics is presented in a standard style using *evaluation contexts* — one can find an explanation in a simpler setting in *Practical Foundations for Programming Languages* [40, §5.3]. The head reduction relation only gives the semantics for an expression where a primitive is at the top level, and furthermore only when it is applied to values (notice for example that the rule for $\mathsf{AllocN}(n, v)$ is defined only when applied to a number and a value, not other expressions). Context reductions describe where the next head reduction rule should apply, for example in $\mathsf{AllocN}(3+2, \mathsf{true}\|\mathsf{false})$ the contexts define what order the arguments should be evaluated in.

Context reductions are defined by a type of evaluation contexts *ECtx*. Rather than going through the details of how these are used (the rule CONTEXT-REDUCE), this section just explains intuitively what evaluation contexts are. Every context E is an expression with a "hole" in it, indicating where the next reduction should take place. As an example, the pair of contexts $E \circledcirc e$ and $v \circledcirc E$ determine how all binary operators work (for example, $\circledcirc$ might be $+$). These say that in an expression $e_1 \circledcirc e_2$, the first argument e_1 can always be reduced (due to the first context), but it must reduce to a value before any reduction steps can take place over the second argument (due to the second context). Once both are values, the pure reduction for binary operators will

Pure reduction $\boxed{e \overset{\text{pure}}{\rightsquigarrow} e'}$

$$
\begin{array}{rcll}
v \circledcirc v' & \overset{\text{pure}}{\rightsquigarrow} & v'' & \text{if } v'' = v \circledcirc v' \\
\ominus v & \overset{\text{pure}}{\rightsquigarrow} & v' & \text{if } v' = \ominus v \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \overset{\text{pure}}{\rightsquigarrow} & e_1 & \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \overset{\text{pure}}{\rightsquigarrow} & e_2 & \\
\pi_i(v_1, v_2) & \overset{\text{pure}}{\rightsquigarrow} & v_i & \\
\mathsf{case\ inj}_i\, v\ \mathsf{of\ inj}_1\, x_1 \Rightarrow e_1\ \mathsf{or\ inj}_2\, x_2 \Rightarrow e_2 & \overset{\text{pure}}{\rightsquigarrow} & e_i[v/x_i] & \\
(\mathsf{rec}\ f(x) = e)\, v & \overset{\text{pure}}{\rightsquigarrow} & e[(\mathsf{rec}\ f(x) = e)/f][v/x] & \\
\mathsf{ArbitraryInt} & \overset{\text{pure}}{\rightsquigarrow} & n & \forall n \in U64
\end{array}
$$

Per-thread head reduction $\boxed{((h, w), e) \rightsquigarrow ((h', w'), e')}$

$$
\begin{array}{llll}
\mathit{NonAtom} & z & ::= & \mathsf{reading}(n, v) \mid \mathsf{writing}(v) \\
\mathit{Heap} & h & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{NonAtom} \\
\mathit{World} & w & & \langle \mathsf{External} \rangle \\
\mathit{TPool} & \mathscr{E} & \in & \mathit{List}\ \mathit{Exp}
\end{array}
$$

$$
\begin{array}{rcll}
((h, w), e) & \rightsquigarrow & ((h, w), e') & \text{if } e \overset{\text{pure}}{\rightsquigarrow} e' \\
((h, w), \mathsf{CmpXchg}(\ell, v_1, v_2)) & \rightsquigarrow & ((h[\ell \mapsto v_2], w), (v, \mathsf{true})) & \text{if } h(\ell) = \mathsf{reading}(0, v) \wedge v = v_1 \\
((h, w), \mathsf{CmpXchg}(\ell, v_1, v_2)) & \rightsquigarrow & ((h, w), (v, \mathsf{false})) & \text{if } h(\ell) = \mathsf{reading}(0, v) \wedge v \neq v_1 \\
((h, w), \mathsf{AllocN}(n, v)) & \rightsquigarrow & ((h[\ell + i \mapsto v \mid 0 \le i < n], w), \ell) & \text{if } \forall 0 \le i < n,\ \ell + i \notin \mathsf{dom}(h) \\
((h, w), \mathsf{PrepareWrite}(\ell)) & \rightsquigarrow & ((h[l \mapsto \mathsf{writing}(v)], w), ()) & \text{if } h[\ell] = \mathsf{reading}(0, v) \\
((h, w), \mathsf{FinishStore}(\ell, v)) & \rightsquigarrow & ((h[l \mapsto \mathsf{reading}(0, v)], w), ()) & \text{if } h[\ell] = \mathsf{writing}(v') \\
((h, w), \mathsf{StartRead}(\ell)) & \rightsquigarrow & ((h[l \mapsto \mathsf{reading}(n+1, v)], w), ()) & \text{if } h[\ell] = \mathsf{reading}(n, v) \\
((h, w), \mathsf{FinishRead}(\ell)) & \rightsquigarrow & ((h[l \mapsto \mathsf{reading}(n, v)], w), v) & \text{if } h[\ell] = \mathsf{reading}(n+1, v) \\
((h, w), \mathsf{Load}(\ell)) & \rightsquigarrow & ((h, w), v) & \text{if } h[\ell] = \mathsf{reading}(n, v)
\end{array}
$$

Context reduction $\boxed{((h, w), \mathscr{E}) \rightsquigarrow ((h', w'), \mathscr{E}')}$

$$
\begin{array}{rcll}
\mathit{ECtx} & E & ::= & \square \mid e\, E \mid E\, v \mid E \circledcirc e \mid v \circledcirc E \mid \ominus E \mid \\
& & \mid & \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& & \mid & (E, e) \mid (v, E) \mid \pi_i\, E \mid \mathsf{inj}_i\, E \\
& & \mid & \mathsf{AllocN}(E, e) \mid \mathsf{AllocN}(v, E) \mid \ldots
\end{array}
$$

CONTEXT-REDUCE

$$\frac{((h, w), e) \rightsquigarrow ((h,', w'), e') \qquad E \text{ is an evaluation context}}{((h, w), \mathscr{E}[i \mapsto E[e]]) \rightsquigarrow ((h', w'), \mathscr{E}[i \mapsto E[e']])}$$

FORK-REDUCE

$$\frac{j \notin \mathsf{dom}(\mathscr{E}) \cup \{j\} \qquad E \text{ is an evaluation context}}{((h, w), \mathscr{E}[i \mapsto E[\mathsf{Fork}(e)]]) \rightsquigarrow ((h, w), \mathscr{E}[i \mapsto E[()]][j \mapsto e])}$$

Figure 7-4: GooseLang semantics

apply. The result is a left-to-right evaluation order for binary operators. The context $\square$ that is just a hole corresponds to the head reductions.

The relation $\rightsquigarrow$, both the head-step reduction and threadpool reduction, define what is assumed to be atomic in GooseLang. The semantics allows any of the concurrent threads in the threadpool $\mathscr{E}$ to execute (due to CONTEXT-REDUCE), but when that thread takes a step, it executes an entire head-step reduction atomically. For example, this is what makes $\mathsf{CmpXchg}(\ell, v_1, v_2)$ an *atomic* compare and exchange; in the same reduction step it can both read *and* set the location $h(\ell)$, which other operations do not do.

Atomicity is also important for the disk. The fact that DiskWrite take a single transition in the semantics is what makes the model of the disk atomic on crash, since a crash also stops the system only between the reduction steps of the semantics.

Disk operations

$$e \quad ::= \quad \ldots \mid \mathsf{DiskRead}(e) \mid \mathsf{DiskWrite}(e, e) \mid \mathsf{DiskSize}$$

Disk external semantics

$$\begin{array}{llll} \mathit{Block} & b & \in & \mathit{Vec}\,4096\,\mathit{U8} \\ \mathit{World} & w & \in & \mathbb{N} \xrightarrow{\text{fin}} \mathit{Block} \end{array}$$

$$\frac{\forall 0 \le i < 4096,\ \ell + i \notin \mathsf{dom}(h) \qquad w[a] = b}{((h, w), \mathsf{DiskRead}(a)) \rightsquigarrow ((h[\ell + i \mapsto \mathsf{reading}(0, b[i]) \mid 0 \le i < 4096], w), \ell)}$$

$$\frac{\forall 0 \le i < 4096,\ \exists k.\, h[\ell + i] = \mathsf{reading}(k, b[i])}{((h, w), \mathsf{DiskWrite}(a, \ell)) \rightsquigarrow ((h, w[a \mapsto b]), ())}$$

$$((h, w), \mathsf{DiskSize}) \rightsquigarrow ((h, w), 1 + \max \mathsf{dom}(w))$$

Figure 7-5: Syntax and semantics for an external disk.

7.5 Modeling and reasoning about Go

A key principle in the design of Goose is to model features of Go as code, written as libraries on top of the base GooseLang language described in section 7.4. This section describes some of the features of Go that Goose models, as well as reasoning principles developed on top to verify code that uses these features.

Specifications in this section are part of the Perennial logic, described in detail in chapter 3. Perennial is based on separation logic, and only features of separation logic are needed to understand this section and not the rest of the Perennial logic. The basic specification in this logic is the *Hoare triple* $\{P\}$ `f()` $\{Q\}$, which says that if `f()` is run in a state satisfying the precondition P and terminates, the final state will satisfy the postcondition Q. Separation logic additionally means the specification implicitly states other state is unmodified, supporting so-called "small-footprint" specifications where P and Q describe only the state actually involved in `f()`. Assertions make use of the *points-to assertion* $\ell \mapsto v$, which gives the value of one address in memory. For example, some basic specifications in separation logic are those for the load and store operations:

$$\{\ell \mapsto v\}\ !\ell\ \{\mathbf{ret}\ v, \ell \mapsto v\} \qquad\qquad \{\ell \mapsto v_0\}\ \ell \leftarrow v\ \{\ell \mapsto v\}$$

The specification for load uses **ret** v, to specify what the return value is. Stores also execute to a value, but since it is the GooseLang unit value we omit it in the specification.

The specifications in this section can be appreciated with only basic familiarity with sequential separation logic. The model of pointers does handle some subtleties related to concurrency, but as explained in section 7.5.1 these are abstracted away for the purpose of proofs. None of the reasoning principles for GooseLang are specific to crash safety. All of these specifications are actually proven in the Perennial logic and thus can be used as part of concurrency and crash safety proofs, as we did for GoTxn.

The Goose translator converts each function in the Go source to a Coq definition of an analogous function in GooseLang. A call to a function `foo` in the Go code is translated to the `foo` Coq declaration in GooseLang. As mentioned in later in section 7.7, this does have some limitations, especially that it does not attempt to model recursion.

Each struct declaration in Go is translated to a Coq definition that declares the struct fields, which is used by the GooseLang struct model (described in detail in section 7.5.3). Top-level constants are translated in a similar way to how Go functions are translated.

7.5.1 Modeling pointers

The model of pointer loads and stores turns out to be subtle because of concurrency. In short, GooseLang disallows concurrent reads and writes to the same location by making such racy access *undefined behavior*. Any program verified with Goose must show that it never has this kind of undefined behavior. This restriction is intended to be more restrictive than the hardware and language guarantees regarding races. The hardware provides some guarantees, but they are typically weak: for example, it is common that different cores can observe the effect of a write at different times (for example, in x86-TSO [84]). Go's own memory model documentation specifies even weaker guarantees. Rather than attempt to formalize Go's rules, the semantics side-steps the issue and makes any races undefined, which works for our intended use cases since the verified code always synchronize concurrent access with locks.[2]

To disallow concurrent reads and writes the semantics must first detect them. The key is to make $x \leftarrow v$, the primitive store operation, *non-atomic* by splitting it into two operations. The GooseLang semantics tracks the behavior of these operations by augmenting the heap with extra information; each address in the semantics has a *NonAtom* which can be $\mathsf{reading}(n, v)$ if there are n readers and the value is v, or $\mathsf{writing}(v)$ if a thread is currently writing. For most pointers, stores are translated to a non-atomic store while loads are translated to GooseLang's atomic Load (concurrent loads to the same address are permitted in both Go and GooseLang). GooseLang has support for non-atomic reads as well in order to model map iteration.

Ordinarily values in the heap are of the form $\mathsf{reading}(0, v)$, to indicate no readers or writers. Writes are split into $\mathsf{PrepareWrite}(\ell)$, which sets the value of ℓ to $\mathsf{writing}(v_0)$, and $\mathsf{FinishStore}(\ell, v)$ which sets it to $\mathsf{reading}(0, v)$. A concurrent write will be undefined since $\mathsf{PrepareWrite}(\ell)$ requires no concurrent writers, and similarly for a concurrent read which is undefined if the address is being written. Non-atomic reads are similar with StartRead and FinishRead; these increment and decrement the number of readers, respectively, so that multiple readers can run concurrently but any concurrent writer has undefined behavior.

Goose provides reasoning principles to abstract away this complexity while verifying programs. Separation logic turns out to provide the right framework to reason about racy access. When a thread owns $\ell \mapsto v$ in its precondition, the logic guarantees no other thread can have access to location ℓ, so the specifications for reads and writes are unaffected by the operations being non-atomic. The only change is that the $x \leftarrow v$ store operation is no longer an atomic primitive but a function that takes two execution steps. In Iris this means that two threads cannot share a pointer with an invariant and must mediate access with a lock, which gives a thread ownership of the $\ell \mapsto v$ assertion for multiple execution steps.

[2] Go doesn't have any formal semantics, but it does have a memory model document that says reads may observe any concurrent write. This document generally encourage well-synchronized programs, which is what the GooseLang semantics enforces.

7.5.2 Locks and condition variables

Locks (Go's `*sync.Mutex` type) are not built-in to GooseLang but modeled using an implementation based on simpler primitives. Since locks are the only synchronization primitive, implementing them requires shared concurrent access, which ordinary pointers do not have in GooseLang. Instead, the language also includes a primitive atomic compare-and-exchange operation that is only used to implement a model of locks. Similar primitives could be used to model Go's low-level atomic operations, like `atomic.CompareAndSwapUint64` and `atomic.LoadUint64`, but Goose does not support them since our code hasn't required such synchronization primitives.

The model of locks is simple enough to give the code in its entirety. The lock is represented as a pointer to a boolean that is true if the lock is held. As a helper we define CAS (compare-and-swap), a variant of compare-and-exchange that only returns a boolean on success and not also the previous value.

$$
\begin{aligned}
\mathsf{CAS} &\triangleq \lambda x, v_1, v_2.\, \pi_2\, \mathsf{CmpXchg}(x, v_1, v_2) \\
\mathsf{NewLock} &\triangleq \lambda \_.\, \mathsf{ref}\ \mathsf{false} \\
\mathsf{Acquire} &\triangleq \lambda l. \\
&\quad \mathsf{let}\ f = (\mathsf{rec}\ \mathsf{tryAcquire}(\_) = \\
&\qquad \mathsf{if}\ (\mathsf{CAS}\ l\ \mathsf{false}\ \mathsf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{tryAcquire}\ ())\ \mathsf{in} \\
&\quad f\ () \\
\mathsf{Release} &\triangleq \lambda l.\, l \leftarrow \mathsf{false}
\end{aligned}
$$

Acquiring a lock is modeled as repeatedly using $\mathsf{CAS}\ l\ \mathsf{false}\ \mathsf{true}$ to atomically test that whether the lock is free and set it to true if so, while release stores false to the lock. This implementation as a spin lock is merely an operational model that captures what the lock does: acquire blocks until the lock is free and sets it to locked, while release frees the lock. This code is used to model Go's builtin `*sync.Mutex`, which is implemented more efficiently than spin locks with cooperation from the runtime and operating system.

The specification for locks is a typical one for concurrent separation logic, based on associating a *lock invariant* with each lock, which is a predicate that holds when the lock is free:

WP-NEWLOCK

$\{P\}\ \mathsf{NewLock}\ \{\mathbf{ret}\ \ell, \mathsf{isLock}(\ell, P)\}$

WP-LOCK-ACQUIRE

$\{\mathsf{isLock}(\ell, P)\}\ \mathsf{Acquire}(\ell)\ \{P\}$

WP-LOCK-RELEASE

$\{P * \mathsf{isLock}(\ell, P)\}\ \mathsf{Release}(\ell)\ \{\mathsf{True}\}$

Because this is a separation logic, we can also interpret the lock invariant as ownership over some data (for example, some region of memory); the lock mediates access to this ownership, handing it out when the lock is acquired and requiring it back when the lock is released. GooseLang has a proof that the specification holds for the spin-lock implementation.

Goose also supports Go's condition variables. As is standard, the idea is that there is some "condition" associated with each condition variable; one thread calls `Signal()` or `Broadcast()` when that condition is true, and another thread calls `Wait()` to wait for the condition to become true. The condition variable is associated with a mutex in Go, which must be held to call `Wait()`. It will be released and then re-acquired when the thread is signaled, but before the waiting thread wakes up other threads can wake up and potentially invalidate the condition (the code must therefore call `Wait()` in a loop, waiting for the condition to be true afterward).

Because condition variables provide few guarantees, the GooseLang model is simply that `Wait()` releases and then immediately re-acquires the lock associated with the condition variable. `Signal()` in reality affects the scheduling of `Wait()` calls, but this model captures all executions. Go condition variables actually do provide a stronger guarantee than in some other systems: `Wait()` is guaranteed to return only if *some* thread has issued a `Signal()`, but we did not attempt to model or use this for reasoning purposes.

7.5.3 Structs

One of the most important features of Go to support is structs. Goose support for structs uses a form of *shallow embedding* using a combination of Gallina (Coq's functional language) and GooseLang. Goose encodes higher-level primitives like field access on top of GooseLang primitives like tuples and contiguous memory.

Struct types are represented using a combination of two Gallina types: $t \in \mathit{GooseType}$ gives the type of a struct while $s \in \mathit{StructDecl}$ is an (anonymous) struct declaration that gives its field names and their types. The definitions of these two types are given in fig. 7-6. The exact types are not important in the semantics, but the translation does use the shape of a struct to determine how it is laid out in memory and to support pointers to individual fields. As an example of how approximate these types are, it is sufficient to have a ptrT type for all pointers; if a pointer to a struct is actually dereferenced (or a pointer to a struct field), then the translation uses its type in Go to determine how that dereference should be implemented. Using types to represent struct shapes also makes the translation simpler, since the Goose implementation has access to accurate types from the `go/types` package, which is essentially the Go type checker.

First Goose needs to model struct values. A struct value is represented as a tuple of its fields. The definition of the struct gives an ordering of the fields. This is enough to construct a struct from its fields and to access a field by name. Here the shallow embedding comes in: struct declarations are translated to a definition of the struct type in Gallina, and struct construction

$$
\begin{array}{lrcl}
\textit{GooseType} & t & ::= & \mathsf{uint64T} \mid \mathsf{boolT} \mid \mathsf{unitT} \mid \ldots \\
 & & \mid & t \times t \mid \mathsf{ptrT} \mid \ldots \\
\textit{StructDecl} & s & \in & \mathsf{list}(\mathsf{string} \times \mathsf{GooseType}) \\
\mathsf{structType}([]) & & \triangleq & \mathsf{unitT} \\
\mathsf{structType}((f,t) :: s) & & \triangleq & t \times \mathsf{structType}(s)
\end{array}
$$

Figure 7-6: GooseLang types and struct declarations. These are used in the semantics only to give a "shape" to data for accessing struct fields, and not to represent the Go type system.

and field access are written as Gallina definitions that produce GooseLang expressions. From the perspective of GooseLang then the struct implementations are a type of macro, since they are implemented in the meta language (Gallina) rather than within GooseLang itself.

Structs in memory are more interesting than struct values. Structs could be stored in a single location; due to our non-atomic semantics for memory, this would be sound even for structs larger than a machine word. However, this model would be too restrictive: it is safe for threads to concurrently access *different fields*, just not the same field, and our code actually takes advantage of this property so that code is more natural; working around this restriction requires splitting structs up if they are stored in memory.

To support this concurrency, Goose models a struct in memory with a flattened representation, with each base element in a separate memory cell. The flattening applies recursively to fields that are themselves structs, until a base literal is reached (like an integer or boolean); base elements are at most a machine word in size. When allocating a new pointer, the semantics flattens composite values and stores the elements in a sequence of contiguous addresses.

With a flattened representation the translation needs non-trivial code to read a struct through a pointer, particularly when some of its fields are themselves flattened structs. Any load of a value from memory is translated to a call to `LoadTyped`. This is a Gallina definition of type $\textit{GooseType} \to \textit{Expr}$, where that expression is itself a GooseLang function that takes a pointer. Hence one can think of it as a macro for GooseLang that is represented in Gallina as a meta language. $\texttt{LoadTyped}(t)$ is directed by a type t passed in Gallina to determine how to load and assemble the fields of a struct of type t.

For the purpose of proofs GooseLang's specifications use a *typed points-to fact* of the form $\ell \mapsto_t v$ to describe a pointer to a value of type t. This definition expands to a number of primitive points-to facts, one for each base element. The specification for loading a pointer to a value of type t is

$$\{\ell \mapsto_t v\}\ \texttt{LoadTyped}(t)\ \ell\ \{\mathbf{ret}\ v, \ell \mapsto_t v\},$$

which (much like the primitive load $!\ell$) hides the fact that something non-atomic is happening and looks like an ordinary dereference. Similarly, `StoreTyped` also takes a type, although the

specification requires the caller to prove that the value has the right shape (in reality it always will because the source code in Go is well-typed).

The payoff of structs being many independent locations is that it is possible to model references to individual struct fields. From a pointer to the root of the struct, a field pointer is simply an offset from that pointer (skipping the flattened representations of the previous fields). This offset calculation is much like the code to read a struct from memory, except that it merely computes a single offset rather than iterating over all the fields and offsets.

The Go language reference specifies that each field acts like an independent "variable" (which is stored in the GooseLang heap when it is mutable in Go), so this model accurately reflects the language definition. Moreover modeling structs as independent locations is also justified as being similar to how the implementation works. Structs in memory are in reality represented by contiguous memory, and field access is implemented by computing a pointer from the base of the struct. The main difference between the physical implementation and the model is that the model has a single, abstract memory location for each field, whereas the implementation encodes all data into bytes.

Recall that $\ell \mapsto_t v$ is internally composed of untyped points-to facts for all the base elements of v. In order to reason about v's fields, we introduce a new *struct field points-to fact*, written $\ell \mapsto_{s.f} v$, which asserts ownership of just field f of a struct with descriptor s rooted at ℓ, and gives that field's value as v. A recursive function gives an "exploded" set of struct fields by iterating over t's fields and v simultaneously. Then, we give a proof that $\ell \mapsto_{\mathsf{structType}(s)} v$ is equivalent to the separating conjunction of this exploded list. The result is a convenient lemma for reasoning about a struct using its fields: in the forward direction, the equivalence breaks a large typed points-to into individual fields (with the values computed from v), while in the other direction it allows to prove a $\ell \mapsto_{\mathsf{structType}(s)} v$ by gathering up all the fields.

The struct field points-to is indispensable in proofs, because it is common to write `x.f` in Go when x is a pointer (in C, this would be written `x->f`). The model for loading a struct field is a function $\texttt{loadField}(s, f)\, x$ which is implemented in two steps, first computing the offset to field f and then dereferencing the computed pointer (in both cases the struct descriptor s describes how to interpret field f). Having a field points-to gives a natural specification for this type of load:

$$\left\{\ell \mapsto_{s.f} v\right\} \texttt{loadField}(t, f)\, \ell \left\{\mathbf{ret}\ v, \ell \mapsto_{s.f} v\right\}$$

The lemmas about breaking apart and recombining structs are all proven against a simpler model of structs that only requires flattening and offset calculations. In a sense the model is the trusted code, but the fact that the struct maps-to exploding lemma and all of the expected Hoare triples hold provides strong evidence that the model is also doing the right thing. For

example, the exploding lemma shows that field offsets are disjoint, since the struct maps-to can be broken into field points-to facts for each field.

7.5.4 Slices

One of the most commonly used data structures in Go is the slice `[]T`, which is a dynamically-sized array of values of type `T`. Goose supports a wide range of operations on slices, including appending and sub-slicing; it turns out that the semantics of mutable slices is non-trivial in Go, resulting in an interesting semantics and reasoning principles.

A slice is a combination of a pointer, length, and capacity. Slices are views into a contiguous memory allocation; this view can be narrowed with sub-slicing operations of the form `s[i:j]`, resulting in aliased slices. The elements between the length and capacity are not directly accessible but are used to support efficient amortized appends. Go's built-in slice operations include bounds checks on all slice operations and panic if a memory access or sub-slice operation goes out of bounds.

GooseLang has a primitive for allocating contiguous memory, which Goose uses to model the allocation underlying a slice (though these are not directly accessible to Go code). On top of these Goose models slices as a tuple of a base pointer, length, and capacity.

The GooseLang slice model is directly inspired by the implementation of slices, including modeling slice capacity. Initially Goose used a more abstract model of slices that ignored capacity, but we were surprised to find that this was insufficient to even accurately model subslicing and appending. Directly modeling slice capacity was the simplest solution to obtain a model that is faithful to the Go implementation. The Go language reference isn't specific about what the slice capacity after various operations should be, so our GooseLang model picks a non-deterministic capacity in several places (within appropriate bounds).

The most basic operations on slices are indexing and storing. The GooseLang model of s is a three-tuple, but for clarity this section will refer to its elements as $\texttt{ptr}(s)$, $\texttt{len}(s)$, and $\texttt{cap}(s)$. The translation of `s[i]` is essentially a (typed) load from $\texttt{ptr}(s) + i$ (or undefined behavior if this offset is out-of-bounds). Similarly `s[i] = x` stores to the same location. Goose translates Go's `len(s)` directly to $\texttt{len}(s)$ and `cap(s)` to $\texttt{cap}(s)$.

The Go `append` operation is the most sophisticated to model. The behavior of `append(s, x)` where `s: []T` and `x: T` depends on whether there is extra capacity to store the new element `x`. If there is capacity, then `x` is stored there and the append returns a new slice with the same pointer but a larger length. If there is no capacity, then `append` must allocate a new array in memory, copy the existing elements to it, and then store `x`. In the latter case `append` returns a slice with a fresh pointer.

The difficulty with Go slices arises when supporting subslicing. Consider `s[:i]`, where `i` is less than `len(s)`. Clearly this slice should have the same pointer and length `i`, but what

should its capacity be? Surprisingly, the capacity of this prefix is the full capacity of `s`, which means that the unused elements of `s[:i]` *include the elements of* `s` beyond the index `i`. As a result, `append(s[:i], x)` in fact modifies `s[i]`. GooseLang takes care to model this behavior by implementing `append` exactly as above, taking into account that `append(s, x)` might be an in-place operation.

The GooseLang model is specifically designed to be sound by sticking to the Go implementation as closely as possible. On the other hand the reasoning principles for slices are intended to be convenient and high-level, and do not involve directly reasoning about slice capacity. The design of GooseLang separates the model from the reasoning principles — Goose has specifications verified against the concrete model, so that the model is trusted and not the separation-logic specifications.

The GooseLang model of slices is based on two abstract predicates: $\texttt{sliceRep}(s, l)$ and $\texttt{sliceCap}(s)$. In the Coq implementation, the model of a slice value is a Gallina record s that must be converted explicitly to a GooseLang value, but this thesis will not make a distinction. Furthermore this section presents just the *untyped* version of this specification where $l : \mathsf{list}(\mathit{Val})$, but GooseLang also has a typed version where $l : \mathsf{list}(T)$ for a type T that has an associated Gallina function $\texttt{to\_val} : T \to \mathit{Val}$. The typed version is convenient in proofs and can be verified as a small extension to the untyped specification.

The first predicate $\texttt{sliceRep}(s, l)$ relates s to the abstract value l that it references. It also represents ownership over the slice's elements and its capacity, in terms of the underlying struct points-to facts. GooseLang's slice specifications use this predicate to describe loading and storing by index:

$$\{\texttt{sliceRep}(s, l) * i < |l|\}$$
$$\texttt{s[i]}$$
$$\{\textbf{ret } v,\ v = l[i] * \texttt{sliceRep}(s, l)\}$$

$$\{\texttt{sliceRep}(s, l) * i < |l|\}$$
$$\texttt{s[i]} = \texttt{v}$$
$$\{\texttt{sliceRep}(s, l[i \mapsto v])\}$$

Next, $\texttt{sliceCap}(s)$ is an abstract predicate that represents *ownership over the capacity* of s. It is necessary to append, since appending might need to write to the capacity, but unneeded to read and write to a slice. We introduce a shorthand $\texttt{sliceFull}(s, l)$ for the append specification, which is like `sliceRep` but also includes ownership over the capacity:

$$\texttt{sliceFull}(s, l) \triangleq$$
$$\texttt{sliceRep}(s, l) * \texttt{sliceCap}(s)$$

$$\{\texttt{sliceFull}(s, l)\}$$
$$\texttt{append(s, x)}$$
$$\left\{\textbf{ret } s',\ \texttt{sliceFull}(s', l +\!\!+ [x])\right\}$$

This specification looks simple but has a non-trivial proof, since in the case where there is

sufficient capacity it moves ownership from $\texttt{sliceCap}(s)$ to $\texttt{sliceRep}(s', l +\!\!+ [x])$ (and $\texttt{ptr}(s') = \texttt{ptr}(s)$), while in the other case where there is no capacity it allocates a new pointer for s' and copies everything over.

The most interesting rules are for subslicing and how they interact with capacity. Consider `s[:i]` again. While Go has no formal notion of ownership, our specifications do. The model for the *value* `s[:i]` is easy enough: it is a pure function that returns a new slice value with a smaller length and the same capacity. This thesis uses $s[:i]$ as the notation for this pure function since the two are so similar. There is one slight complication that `s[:i]` causes a runtime error in Go if $i \geq \texttt{len}(s)$. Thus GooseLang proves the following specification for indexing:

$$\{i < \texttt{len}(s)\}\ \texttt{s[:i]}\ \{\textbf{ret}\ s[:i], \text{True}\}$$

This specification is slightly unusual in that it has a logical precondition but does not refer to the heap (or any other state).

To reason about subslicing from the perspective of memory, the specification needs to relate ownership of $\texttt{sliceRep}(s, l) * \texttt{sliceCap}(s)$ to ownership of $\texttt{sliceRep}(s[:i], \mathsf{take}(l, i))$. It turns out there are two possibilities that the proof engineer can choose between: either give up ownership of the remainder of s in exchange for $\texttt{cap}(s[:i])$, or ignore the capacity of the subslice and keep $\texttt{sliceRep}([i:], \mathsf{drop}(l, i))$. These are incomparable and unexpressed in the Go code being verified: the decision is based on whether subsequently the program will append to the slice prefix but stop using the old slice, or if it will use both $s[:i]$ and $s[i:]$.

To support both ways that $s[:i]$ might be used, GooseLang has two theorems for reasoning about taking a prefix of a slice. Both of these theorems require $i < \texttt{len}(s)$.

SLICE-TAKE-FULL

$$\texttt{sliceFull}(s, l) \vdash \texttt{sliceFull}(s[:i], \mathsf{take}(l, i))$$

SLICE-SPLIT

$$\texttt{sliceRep}(s, l) \dashv\vdash \texttt{sliceRep}(s[:i], \mathsf{take}(l, i)) * \texttt{sliceRep}(s[i:], \mathsf{drop}(l, i))$$

The rule SLICE-TAKE-FULL captures that the program proof can get full ownership of $s[:i]$, in exchange for dropping any ownership over the rest of s. The second rule SLICE-SPLIT instead allows the program proof to split s into two parts at the ith index, but in exchange requires giving up $\texttt{sliceCap}(s[:i])$, the right to append to the first half.

There is one more fact that is sometimes useful, which is that $\texttt{sliceCap}(s) \dashv\vdash \texttt{sliceCap}(s[i:])$ (this is somewhat intuitive; capacity is past the end of the array, so dropping elements from the front has no impact). As a corollary of SLICE-SPLIT, the proof can also split full ownership of

a slice into two (asymmetric) parts:

SLICE-SPLIT-FULL

$$\texttt{sliceFull}(s, l) \dashv\vdash \texttt{sliceRep}(s[:i], \mathsf{take}(l, i)) * \texttt{sliceFull}(s[i:], \mathsf{drop}(l, i))$$

These two reasoning principles correspond to two ways to think about indexing: taking a prefix while using the original slice as capacity for the prefix (the first theorem, SLICE-TAKE-FULL), or splitting the slice into two parts (the second theorem SLICE-SPLIT and its corollary SLICE-SPLIT-FULL). It was interesting that we discovered these principles only while proving theorems about subslicing in GooseLang. Even as experienced Go developers, we had never thought about subslicing and appending in enough detail to appreciate this difference in ownership reasoning.

7.5.5 Maps

After slices, maps are the next most commonly used collection type in Go. GooseLang models maps using an implementation that represents a map as a list of key-value pairs, stored in a single memory location in reverse insertion order. Go's builtin maps are *not* thread-safe, so the model enforces single-threaded access by marking the map as being read while reading from it; this re-uses the race detection for other pointers to ensure that any write to a map while iterating over it is considered undefined behavior, while allowing concurrent read-read access. Maps support all the usual Go operations: insertions, reads (including returning whether the key is present), `len` to get the number of elements in the map, deletion, and iteration.

The implementation of maps is the most involved out of any of the Go primitives. It required directly implementing maps (albeit inefficiently, using an association list) using recursive GooseLang code. We improved our confidence in this implementation both by running it on concrete examples (using the infrastructure described in section 7.6) and proving a specification for the various map operations. Both of these essentially rule out type errors, and the specification is a redundant check on the behavior. Simple tests and verification rule out easy-to-make mistakes like reading the oldest write to a key rather than the latest, or duplicate keys during iteration (the implementation must skip over a key after observing it once).

The model represents a Go map as a pointer to an abstract map value, a GooseLang value that encodes the entire map data as a list of key-value pairs. The specification is based on a pure relation $\texttt{mapVal}(v, m)$ that relates this encoded value to a Gallina map m, which uses `gmap` from stdpp;[3] for simplicity m is of type `gmap u64 val` and Goose only supports maps whose key type is `uint64`. Map values are not a visible notion to the Go code, since it always interacts with

[3] https://gitlab.mpi-sws.org/iris/stdpp

maps via their pointer, so the specifications all use $\texttt{mapRep}(\ell, m) = \exists v.\, \ell \mapsto v * \texttt{mapVal}(v, m)$. The indirection is important, since the Go map value `m: map[uint64]V` is in fact a reference to a map that is mutated in-place (unlike a slice, which has both pure data — pointer, length, and capacity — and heap data). Concurrency is simple for these specifications because Go's maps are not thread safe, so all the specifications assume full ownership over the map's reference.

Many of the map specifications relate an implementation to a model over the Gallina map. For example, this is the specification for map deletion:

$$\{\texttt{mapRep}(l, m)\}$$
$$\texttt{mapDelete}(l, k)$$
$$\{\texttt{mapRep}(l, \mathsf{delete}(m, k))\}$$

Map iteration has a more sophisticated implementation and specification. Consider a generic loop over a map in Go like the following:

```
for k, v := range m {
  body(k, v)
}
```

The model for this entire construct is given by $\texttt{MapIter}(m, \mathsf{body})$, where m is a reference to the map and $body$ is an expression for the body of the loop. Goose translates generic loop bodies, so the Go code does not literally need to consist of a call to a separate function. Recall that the representation for a map is a list of all appended key-value pairs in reverse insertion order (insertions do not delete old values). To iterate over the map's contents, the implementation gets the next key-value pair, then deletes that key from the rest of the map before making a recursive call, avoiding iterating over old values that are no longer logically part of the map. The code for the model uses the order of key-value pairs in the order of the GooseLang representation, but Go actually randomizes map iteration order; see the relevant discussion of limitations in section 7.7.

The possibility of *iterator invalidation* adds a subtlety to Go's map iteration — it would be incorrect for the body of the loop to modify the map (it might be sound to write to the map without modifying the domain, but Goose does not attempt to model this). The model of maps puts the entire contents of a map in one heap location. In order to ensure iterator invalidation is undefined behavior, the model marks the map's reference as being read for the entire duration of iteration, using the StartRead and FinishRead GooseLang primitives at the beginning and end of `MapIter`, respectively. If the map value in the heap is $\mathsf{reading}(n, v)$, these operations increment and decrement (respectively) the reader count n, so that any writes within the body have undefined behavior.

Iteration has a *higher-order* specification that assumes a specification for the body, showing it preserves a loop invariant P over the part of the map consumed so far:

$$\frac{\begin{array}{c}\forall m_0, k, v.\, k \notin m_0 \wedge m[k] = v \to \\ \{P(m_0)\}\ \mathsf{body}\ k\ v\ \{P(m_0[k \mapsto v])\}\end{array}}{\{\mathtt{mapRep}(\ell, m) * P(\emptyset)\}\ \mathtt{MapIter}(\ell, \mathsf{body})\ \{\mathtt{mapRep}(\ell, m) * P(m)\}}$$

GooseLang includes some alternate specifications, proven on top of this one, that express the invariant in slightly different ways — for example, it is often useful for a proof to express its loop invariant in terms of both the map iterated over so far and the remaining subset of the map.

Getting the size of a map has a relatively simple specification but a somewhat complex implementation. Like map iteration, the code for `MapLen` (modeling `len(m)` in Go) must delete old keys from the map's association list in order to avoid counting overwritten values. A subtlety is that the map length is a 64-bit integer, and this counter could in principle overflow while iterating over the map. This situation is unrealistic since a map with 2^{64} elements would take an unreasonable amount of memory, so the model *assumes* that the map size stays below 2^{64}. The specification for `MapLen` is:

$$\begin{array}{l}\{\mathtt{mapRep}(\ell, m)\} \\ \quad \mathtt{MapLen}(\ell) \\ \{\mathbf{ret}\ s, s = |m| * \mathtt{mapRep}(\ell, m)\}\end{array}$$

The fact that the type of the return value s is a 64-bit integer reflects the bounded-size assumption.

7.6 Testing Goose

Goose is a trusted component in the entire verification process. For the overall system's proof to be sound, we rely on the model to produce all of the behaviors of the Go code; that is, the behaviors of the Go code (which depend on the Go compiler) should be a subset of the behaviors of its translated GooseLang (according to the Coq semantics). As long as this is case, the proof is sound in that if the modeled system always satisfies some property the code will, too.

One subtlety to Goose soundness is that the system is automatically sound if Goose fails to translate some code, or the code does not compile in Coq, or the model has undefined behavior; in each of these cases it is impossible to verify incorrect code. These might still reflect a bug in Goose, or at least an unsupported feature, but they do not compromise soundness. Therefore

the most important bugs are those where the translation is well-defined but its behavior differs from that of Go; these can compromise soundness of the system and lead to a proof that is not borne out in practice.

To increase out confidence in Goose, we implemented a large suite of unit tests. On their own these tests check that Goose continues to translate existing code (and check that the translation has not unexpectedly change). To test the soundness of the translation, the relevant comparison is between Go and GooseLang. Unfortunately GooseLang is not natively an executable language. Its semantics is expressed as a Coq relation that describes the valid executions of an expression, but not how to run a particular expression.

To test GooseLang code, we implemented an interpreter in Coq, which can run GooseLang code and produce either an error due to undefined behavior or a result. The interpreter is itself verified to match the semantics. The specification for the interpreter is slightly subtle in that the interpreter produces only one possible execution rather than all the executions allowed by the semantics, but the non-determinism is only due to the choice of what locations to use for pointers, which should not affect any visible behavior.

GooseLang is a lambda calculus, so its semantics is expressed as a transition system between expressions. It is easy to interpret *pure* reductions like `x + y` where `x` and `y` are values, since the semantics of these pure expressions on their own is already given as a Gallina function rather than a relation. The semantics of each core primitive is given by a transition relation, which the interpreter implements as a function and its correctness theorem shows this function produces an allowed transition. For example, the semantics of allocation stores a value in an unused address, whereas the interpreter concretely identifies an unused location (the maximum used address, plus one).

The challenge in the interpreter's correctness theorem comes from *context* reductions, which specify how to find a sub-expression within `e` to reduce if the head is not immediately a value. The semantics follows a standard presentation of context reduction using *evaluation contexts*. The idea is to define a type of evaluation contexts $E \in \mathscr{E}$ that represent an expression with a hole; $E[e]$ represents filling that hole with the expression e. The possible evaluation contexts give all the context reductions in one compact rule, CONTEXT-REDUCE: if e can step to e', then $E[e]$ can step to $E[e']$. This rule applies whenever such an E exists, while the interpreter recurses through an expression (in the right order) and evaluates a sub-expression, then fills it into the context. The interpreter's proof shows that this traversal is correct, proving that the interpreter and semantics agree on an evaluation order. Specifically, the interpreter proof shows the interpreter produces a valid evaluation order, and a separate proof shows that evaluation contexts are unique.[4] There is other non-determinism in the semantics that the interpreter does not fully explore, though, such as for allocation.

[4] See the lemma `head_redex_unique` in src/goose_lang/lang.v.

The test suite is structured as a number of test functions, each producing a boolean that should be true. To check that the test itself is written correctly, a Go test checks that it produces `true`. Then to check the semantics of the translation, the GooseLang test infrastructure translates the test and runs it through the interpreter, checking that this produces `true` in GooseLang. While the interpreter is not extremely efficient, it is fast enough to run the tests in the test suite.

The GooseLang interpreter and test framework was designed and implemented by Sydney Gibson, and is described in greater detail in her master's thesis [33]. That thesis includes more details on evaluating the interpreter itself, for example documenting bugs caught by the test suite and other bugs that are now part of Goose's regression tests.

7.7 Limitations

Notably missing in Goose but prominent in Go is support for interfaces and channels. We believe both are easy enough to support, but interfaces were not necessary to implement GoTxn, and rather than channels GoTxn use mutexes and condition variables for more low-level control over synchronization.

Control flow is slightly tricky since a Go function is translated to a single GooseLang expression that should evaluate to the function's return value. Goose supports many specific patterns, especially common cases like early returns inside `if` statements and loops with `break` and `continue`, but more complex control flow — particularly returning from within a loop — is not supported. It would be easiest to express general control flow in *continuation-passing style*, (n which every GooseLang takes a continuation, and calling this continuation corresponds to returning from the function in Go, but this would complicate every specification and the translation of function calls.

Goose does not support Go's `defer` statement. It would be nice to support some common and simple patterns, particularly for unlocking, by translating `defer` statically. The behavior of Go's `defer` statement in general is to push the deferred function to a stack of calls associated with the current function that are executed in reverse order at return time. GooseLang does not have a first-class notion of a Go function to associate the stack of deferred functions with, nor the concept of returning from a function. However, it would be useful to support simple static uses of `defer` at the top-level of a function.

Named return values are recommended to document return parameters, and sometimes simplify and clarify the body of a function.[5] However, in general they are quite subtle, due to interaction with `defer` statements and concurrency [9]. One source of difficulty is that the return values are treated like local variables declared at the top of the function, and it is easy to accidentally have races on these variables if they are accessed concurrently.

[5] See the description in Effective Go.

Goose does not support mutual recursion between Go functions, and additionally requires the code to be written in topological order so definitions appear before they are used. Files in a package are especially prone to this issue, since they are processed in alphabetical order; we sometimes name files something like "0constants.go" to make the order work out correctly. The subtlety here is that definition management in Go, as in most imperative languages, conceptually treats all top-level definitions as simultaneous, whereas Coq processes definitions sequentially. Using Coq definition management to model Go definition management imposes a limitation compared to Go, but is much simpler to work with compared to modeling a Go package as a set of mutually recursive definitions. Reasoning about code written in such a model would require setting up specification for all the definitions, then proving them in a recursive way, all while ensuring that no specification is used before it is proven.

GooseLang does have one extant bug related to evaluation contexts. The contexts $e\ E$ and $E\ v$ define a right-to-left evaluation order for functions, which is the opposite of Go. We haven't yet fixed this, either by adjusting the GooseLang semantics or changing the translation to emit code that explicitly evaluates all the arguments in the correct order before calling the function.

Map iteration in Go happens in a non-deterministic order.[6] Thus, strictly speaking, the model of map iteration described in section 7.5.5 given by `MapIter` should shuffle the elements of the map before iterating over it, in order to model the non-determinism of the implementation. Goose does not (currently) do this, simply because the shuffle would be hard to implement in GooseLang. However, the specification for map iteration does not expose an iteration order and would apply unchanged to this more non-deterministic model. All proofs go through a common iteration specification, so our proofs should remain unchanged if `MapIter` started modeling non-deterministic ordering.

7.8 Implementation

The Goose translator is implemented in Go, in about 4,000 lines of code. It takes advantage of a family of packages under the `go/` prefix in the Go standard library, such as `go/ast` and `go/types`, to parse and type-check Go code; the `golang.org/x/tools/go/packages` package even makes it possible to work with Go packages and modules. These greatly simplify implementing Goose and also make it more trustworthy, since at least the source code is parsed in exactly the same way as the compiler (though we can still introduce bugs in interpreting what the AST means). The translator has an intermediate representation of the Coq source for GooseLang, and splits translation into generating the Go structs for this representation and subsequently printing this representation as a Coq file.

[6] In fact the runtime randomizes the starting position each time a map is iterated over, to avoid code that unintentionally relies on any particular behavior. See `mapiterinit` from src/runtime/map.go.

Component	Lines of Coq
GooseLang definition	3,847
Libraries (implementation)	951
Libraries (proof)	9,010
GooseLang total	13,808

Figure 7-7: Lines of code (including proofs) for GooseLang.

The translator makes as much of an attempt as possible to identify and report errors, identifying where a feature is unsupported. Each function is translated almost independently, allowing translation to move on to the next function and report a whole batch of errors. Some issues are not checked by the translation, such as the topological order on definitions mentioned in limitations above (in section 7.7); in these cases the resulting Coq code does not compile, which preserves soundness but results in a worse user experience.

GooseLang is implemented in Coq, in a total of about 14,000 lines of code; a breakdown is given in fig. 7-7. This includes the core language definition as well as libraries implemented in the core language to model Go features. Each library has associated reasoning principles verified in Coq.

GooseLang has a notion of "external" operations and state, which is generally instantiated with a disk. The translator checks that the source package imports only one of the allowed external imports, including through its transitive dependencies. It also supports code that uses no external operations, translating it to a form that is itself parameterized, allowing the result to be used as part of any other code.

7.9 Conclusion

Goose is an approach for verifying Go code. We define GooseLang as a model of Go and automatically translate a subset of Go to this language. GooseLang comes with a number of reasoning principles for handling features of Go. The benefit of this approach is the ability to write high-performance code in a productive language, with convenient reasoning while verifying that code. Several aspects of the design contribute to making the approach sound, ranging from the subset of Go supported, to the design of GooseLang, and the use of standard Go tools for analyzing the source code.

Our main use case for Goose for this thesis was to verify GoTxn, but the tool and approach are more generally applicable, even without concurrency or crash-safety reasoning. These ideas could also be productively applied to languages other than Go — I am personally excited about the prospect of having a version for Rust.

Chapter 8

Implementation

This chapter describes some details of the DaisyNFS implementation. Implementation details on the Perennial framework are given in section 3.6 and details on Goose are in section 7.8.

The implementation of DaisyNFS is split into several public code repositories:

- `https://github.com/mit-pdos/daisy-nfsd` builds the overall file-system binary. The repository has the Dafny code and proofs. Most of the evaluation and benchmarking code is also here.
- `https://github.com/mit-pdos/go-journal` has the Go code for GoTxn (the name is a remnant from GoJournal, which was only a journaling system). This repository is just a Go library, since the proofs are elsewhere.
- `https://github.com/mit-pdos/go-nfsd` is an unverified NFS server which we used to evaluate GoJournal [13]. Some of the evaluation code relies on scripts here, even if not comparing against this server.
- `https://github.com/mit-pdos/perennial` has the Perennial framework, GooseLang, and GoTxn's proofs. The repository also holds several other proofs built using Perennial not described in this thesis.
- `https://github.com/tchajed/goose` has the Goose translator and some small Go support libraries, in particular the library that exposes the disk to verified code.
- `https://github.com/tchajed/marshal` is the code for a verified serialization library used in GoTxn. The proofs are in the Perennial repository.
- `https://github.com/zeldovich/go-rpcgen` is a library for generating code from XDR and SUN RPC specifications.

DaisyNFS has some unverified code wrapping the verified implementation of the NFS operations. This glue code uses `go-rpcgen`, a library that parses XDR struct definitions [28]; XDR is a generic serialization protocol similar to protobuf, Cap'n Proto, or FlatBuffers. The XDR code is unverified, but we did use fuzzing to test both memory safety and that deserialization is

	Lines of code (Go)	Lines of proof (Coq)	Ratio
CIRCULAR	109	1,907	17×
WAL-STS	563	13,013	28×
WAL	—	2,854	—
OBJ	250	4,467	18×
JRNL-STS	172	1,729	19×
JRNL	—	1,616	—
LOCKMAP	118	1,778	15×
Misc.	211	1,707	8×
TXN	255	2,153	8×
Transaction refinement	—	6,293	—
Simulation transfer	—	1,854	—
GoTxn total	1,674	39,371	—

Figure 8-1: Lines of code and proof for the components of GoTxn. Ratio is the proof:code ratio, a rough measure of verification overhead.

the inverse of serialization. We use XDR definitions for SUN RPC [89] and the NFSv3-specific definitions from RFC 1813 [7].

The file-system code interacts with GoTxn via an axiomatized interface in Dafny that is substituted with a real implementation that calls the GoTxn library. A similar approach is used to represent Go byte slices.

Similar to VeriBetrKV [39], we followed a discipline of identifying and addressing timeouts in the Dafny proofs. As a result, the overall build is fast: compiling the Dafny proofs takes only 12 minutes on a slow machine in continuous integration and 4 minutes on a laptop using eight CPU cores.

GoTxn is implemented using fairly standard Go code. Goose imposes some restrictions, but we nonetheless have fairly idiomatic code, split into modules and multiple files as expected. One interesting library in GoTxn is its `lockmap` library, which logically implements per-address locking but does not materialize all of these locks for better memory usage (this is similar to Guava's "striped" locking library [2]). The specification for the library mirrors the specification for the usual locks, with a lock invariant associated with each possible address, hiding how the data structure is implemented.

8.1 Lines of code

The lines of proof, code, and specification for the layers of GoTxn are summarized in fig. 8-1. The GoTxn simulation transfer theorem's proof is relatively large because code executed in

	proof	code	spec
GoTxn (Go + Coq)	39,371	1,674	932 (Thm 6.2)
File system (Dafny)	6,787	4,051	630 (Thm 6.3)
Trusted interfaces (Dafny)	—	—	558
`daisy-nfsd`	*unverified*	1,144	—

Figure 8-2: Lines of proof, code, and trusted specification for DaisyNFS.

atomically blocks can include many Go operations modeled by Perennial, and the proof has cases to handle each operation. However the result of the proof is a relatively concise specification as a plain Coq statement that doesn't refer to the Perennial logic.

The file-system operations are implemented in Dafny, which helped us verify a relatively complete system without too much tedium. A breakdown of DaisyNFS's lines of code is given in fig. 8-2. The proof-to-code ratio (where code is the number of lines extracted by Dafny's `/printMode:NoGhost` flag) is about 2× for the file system code. The proof summarizes the implementation well, with about 6× fewer lines of specification as code (about half that specification is quite verbose and concerns error codes and attributes). For efficiency, the Dafny code has trusted interfaces to primitives like byte slices and integer-to-byte encoding. Together these are written in 558 lines of trusted Dafny code. Finally, to complete the NFS server required around 1,000 lines of Go code, about half of which bridge between the Dafny method signatures and the actual NFS structs.

8.2 Achieving good performance for DaisyNFS

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

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

The on-disk data structures—inodes, indirect blocks, and directories—are represented in memory in their serialized form and modified by updating this representation directly, a "zero-

copy" implementation that avoids the cost of an extra memory copy between representations. These were first written as ordinary parsers that produced immutable data structures, which was then migrated to this more efficient implementation. The proofs for this code are ad-hoc rather than a systematic; DaisyNFS only has a handful of formats which did not justify something more general. There is interesting work on verified parsing with zero-copy support [91], albeit not in Dafny.

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

Chapter 9

Evaluation

In this chapter we evaluate DaisyNFS and GoTxn in terms of performance (section 9.1), correctness (section 9.2), and ease of change (section 9.3).

9.1 Performance

The first question answered by the evaluation is, can DaisyNFS and GoTxn achieve good performance, compared to an unverified baseline? This section evaluates performance both with a single and multiple clients.

9.1.1 Experimental setup

To evaluate the performance of DaisyNFS, we ran three benchmarks: the LFS smallfile and large-file benchmarks, and a development workload that consists of `git clone` from a local repository followed by running `make`. These are the same benchmarks used by DFSCQ [18] (a state-of-the art sequential verified file system). As a baseline, this evaluation compares against a Linux NFS server exporting the widely-used ext4 file system. The NFS server lets us compare fairly since both go through the Linux NFS client and use the same underlying protocol. The ext4 file system is run in the non-default `data=journal` mode, forces all data to go through the journal and disables log-bypass writes, which gives the same crash-safety guarantees and makes the designs more comparable. In its default `data=ordered` mode Linux achieves better performance on some benchmarks but can lose recently written data if the system crashes; we also present some results to show the benefit of providing this weaker guarantee.

All of these benchmarks were run using Linux 5.13, Dafny 3.5.0, Go 1.18.1 on an Amazon EC2 i3.metal instance, which has two 36-core Xeon processors running at 2.7 GHz, 512 GB of RAM, and a local 1.9 TB NVMe SSD. To reduce variability we use a bare-metal instance, limit

experiments to a single 36-core socket, disable turbo boost, and disable processor sleep states; the coefficient of variation for all experiments is under 5% so we omit error bars for visual clarity.

The evaluation uses three benchmarks, `smallfile`, `largefile`, and `app`, which test different aspects of file system performance.

Smallfile. The `smallfile` benchmark is a metadata-heavy benchmark that repeatedly creates a file, writes 100 bytes to it, calls `fsync()`, and then deletes the file. The performance reported is the throughput in terms of files/s for this whole process. At the NFS level each iteration results in four RPCs issued to the server: `LOOKUP` for the file being created (which fails), `CREATE`, `WRITE`, and `REMOVE`. Because both DaisyNFS and the Linux NFS server run these same RPCs, the comparison is meaningful.

The `smallfile` benchmark is also used to measure the scalability of the file system. We run several concurrent clients, each repeatedly running the create-write-delete sequence above, for several seconds, then report the aggregate throughput of all of these clients. A concurrent file system should be able to process these RPCs simultaneously — concurrent threads have a chance to prepare new transactions while the system is committing older transactions. Furthermore on a physical disk the journaling for simultaneous operations can be combined (an optimization called *group commit*), resulting in less I/O than separate commits.

Largefile. The `largefile` benchmark is a data-heavy benchmark that creates a file, then repeatedly appends to the file until it is 300 MB, and finally calls `fsync()` and closes the file. The performance reported is the throughput in terms of MB/s for these appends. The Linux NFS client buffers the entire append process until the final sync, at which point it issues the writes in many chunks in parallel. This benchmark is challenging to support efficiently because the writes at the NFS server do not come in order, so some are past the end of the file and then the gap is filled in by later RPCs.

App. The application workload `app` consists of running `git clone` on the xv6 repository followed by `make`. The main purpose of including this workload is to demonstrate that DaisyNFS covers a useful range of the NFS interface, since `git clone` especially uses a wider mix of operations and arguments than the other benchmarks, and also that the file system implements these operations efficiently enough to not slow down the workload. The performance reported for app converts this latency to a throughput number "app/s" so that higher is better. The compilation part of this workload does take about 1.2s, out of an overall build that typically takes about 2.5s under Linux NFS, and this time is unaffected by the file system's performance. The xv6 code being compiled is an operating system, so building it requires running the usual development tools—gcc, ld, ar—but also running dd to generate a kernel image.

9.1.2 Single-client performance

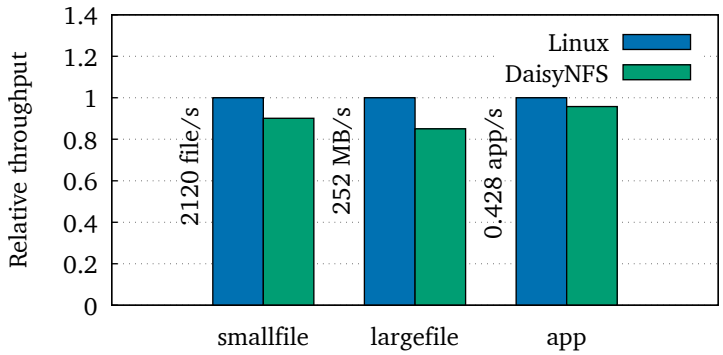


Figure 9-1: Performance of Linux NFS and DaisyNFS for `smallfile`, `largefile`, and `app` workloads, on an in-memory disk. DaisyNFS achieves comparable performance to Linux across these benchmarks.

The first set of results are for the three benchmarks, run on an in-memory disk, all with a single client. These benchmarks are challenging to support efficiently since an in-memory disk means performance isn't dominated by slow I/O, and a single client demands single-core efficiency. The results are shown in fig. 9-1. DaisyNFS achieves comparable performance on these benchmarks after implementing many optimizations (for example, parsing data structures in-place) and features (for example, returning attributes to enable client-side caching). Both CPU and I/O efficiency are important to get good performance.

DaisyNFS gets slightly lower throughput compared to Linux on the `largefile` benchmark. The NFS client's behavior on this benchmark means it issues writes past the end of the file; the semantics of such a write are to fill the gap with zeros. DaisyNFS and Linux get good performance despite this because they implicitly encode those zeros without even allocating a block. This benchmark shows that GoTxn is good at handling concurrent, synchronous writes.

9.1.3 Scalability

The next experiment evaluates DaisyNFS's scalability by measuring the throughput when clients issue operations concurrently. The benchmark used is the `smallfile` benchmark with a varying number of concurrent clients, all running in separate directories. Because this experiment runs on a physical disk, other threads have a chance to prepare transactions while the transaction system is committing to disk.

To evaluate the importance of DaisyNFS's concurrency, we compare against two other configurations of DaisyNFS with reduced concurrency. The "seq WAL" configuration adds locks to the write-ahead log so that the disk operations that normally execute lock-free — installation,

logging, and reading installed data — instead happen under the write-ahead log's in-memory lock. This permits transactions to execute concurrently while not accessing GoTxn but does not allow any parallel disk access. The "seq txn" variant leaves the write-ahead log the same but changes the per-address locking in GoTxn to a *global* transaction lock, preventing any transactions from running concurrently but allowing concurrency between logging and installation in the write-ahead log. Mark Theng came up with the idea of comparing to this seq txn variant in his master's thesis [93].

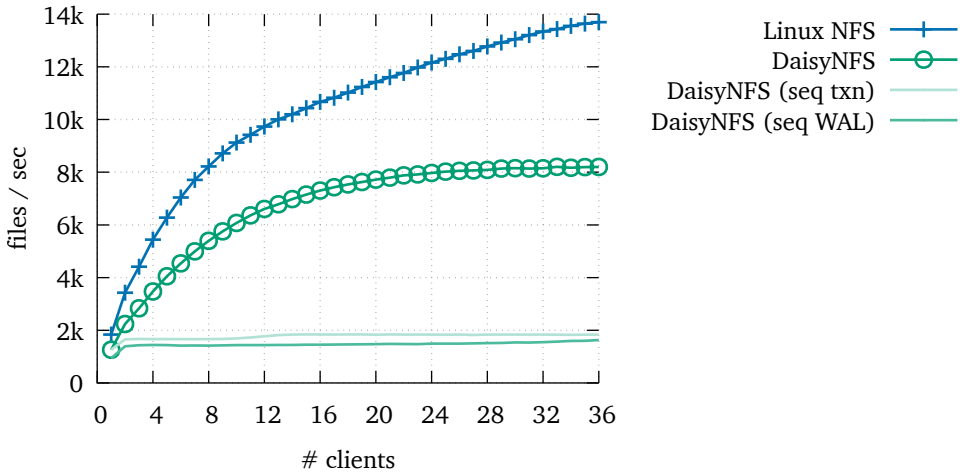


Figure 9-2: Combined throughput of the `smallfile` microbenchmark, varying the number of concurrent clients. DaisyNFS's performance scales with the number of clients, but both eventually scale sub-linearly due to lock contention. DaisyNFS has a single lock that becomes contended more quickly than the contention observed in Linux.

Scaling with number of clients. The scalability experiment varies the number of concurrent smallfile clients, measuring the aggregate throughput of all of the clients. The results are shown in fig. 9-2. Each data point is the result of running for 30 seconds. The graph shows that both DaisyNFS and Linux achieve better performance as the number of clients increase, though Linux has better scalability on this benchmark. The peak throughput of DaisyNFS is about 60% of the throughput of Linux.

With a large number of clients, both file systems stop scaling due to lock contention; they are neither I/O bottlenecked nor are they using the entire CPU. For example, with 30 clients Linux uses only 19% CPU while DaisyNFS uses 19% in userspace and an additional 8% in the kernel; both numbers include the cost of the NFS client, which runs in the kernel. We confirmed that I/O is not a bottleneck with an analogous experiment run on an in-memory disk, whose results are shown in fig. 9-3. DaisyNFS has a common lock for installing sub-block object writes into a full block, which all operations go through to commit. The Go mutex profile points to this

lock as being the biggest source of contention in DaisyNFS for both the NVMe and in-memory benchmarks. The fact that scalability is limited by this common lock indicates the importance of concurrency for performance.

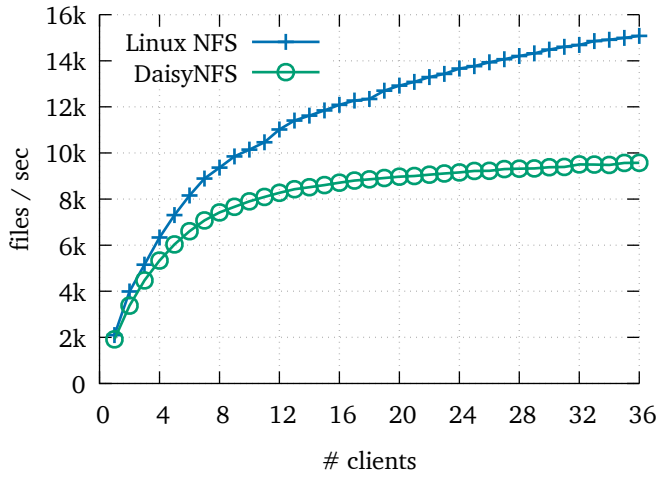
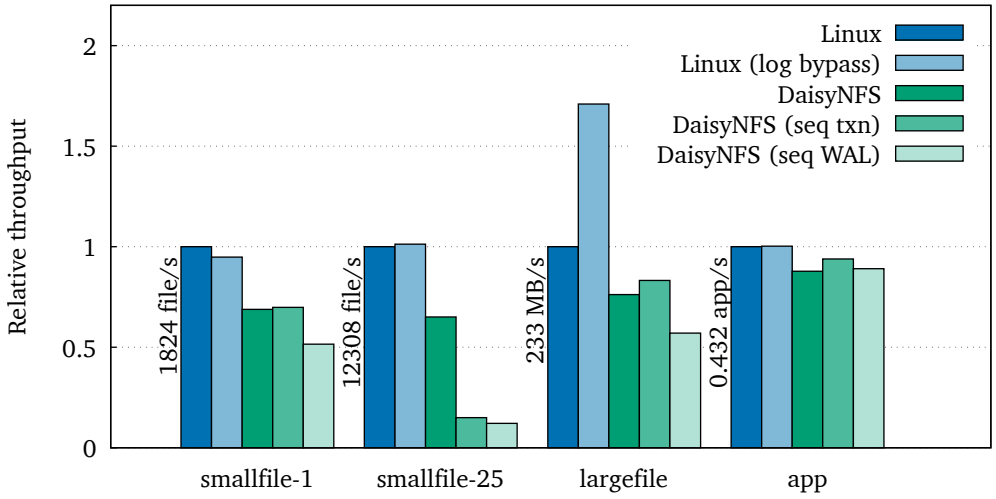


Figure 9-3: Scalability on the `smallfile` benchmark, running on an in-memory disk.

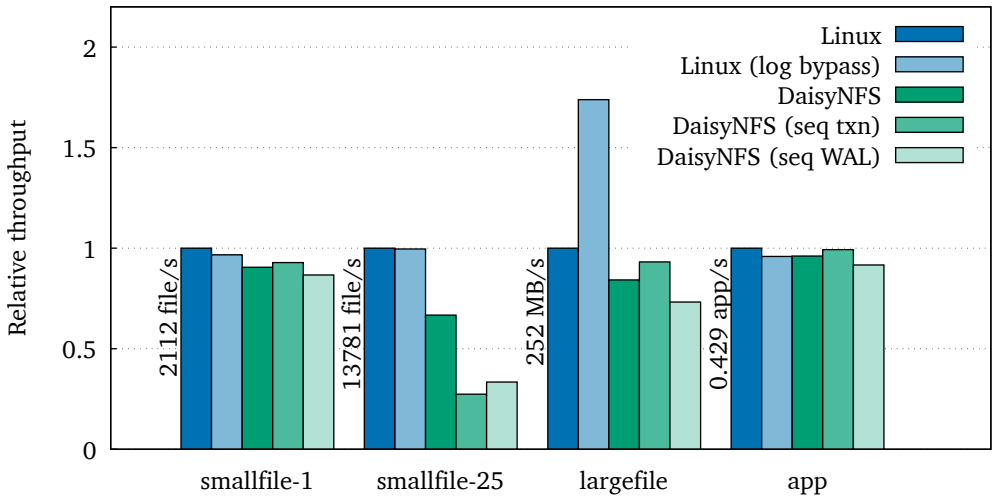
The DaisyNFS seq WAL and seq txn variants show even more clearly that concurrency is important for performance. In both configurations, throughput essentially doesn't increase beyond two clients. Going from one to two clients helps because even though in both of these configurations the server cannot process more than one request at a time, performance improves slightly with a second client since a request can be received while another is being processed.

Concurrency and log bypass for largefile. The final set of experiments aims to understand the performance of DaisyNFS on the largefile and app benchmarks as well. We compare against the same variants of DaisyNFS with reduced concurrency, as well as against Linux with its log-bypass optimization (that is, mounting ext4 with its `data=journal` option). The results are presented in fig. 9-4, on both an NVMe disk and an in-memory disk. The results for the NVMe disk have most of the interesting trends, but there are a few interesting differences worth noting.

The `smallfile-25` and `smallfile-1` numbers repeat individual points from the fig. 9-2, but presented as throughput relative to Linux so that it is easier to compare across file systems. The `smallfile-1` results shown here make it clearer that DaisyNFS has worse performance than Linux with only a single client on a physical disk, even though performance is comparable with an in-memory disk. This is due to less efficient I/O: from the output of `blktrace`, DaisyNFS issues 20 write requests per smallfile iteration whereas Linux only requires 6.5 writes on average; the total I/O is also larger, 108KB vs 80KB; and Linux's journaling design results in sequential writes whereas DaisyNFS repeatedly writes to the same header blocks.



(a) On an NVMe disk



(b) On an in-memory disk

Figure 9-4: Performance across various file-system configurations. The smallfile benchmark is run with a single client (`smallfile-1`) and also with 25 parallel clients (`smallfile-25`). The "log bypass" variant of Linux uses the `data=ordered` ext4 option to write data directly rather than through the journal. The "seq txn" variant of DaisyNFS has a global transaction lock while "seq WAL" adds additional locking to the write-ahead log.

The `smallfile-1` benchmark is interesting because the client issues a single operation at a time, and yet seq WAL performs noticeably worse. This is because ordinarily the write-ahead log parallelizes logging writes and installing them to the disk (rather than alternating between logging and installation, as in seq WAL), and the disk is able to get better throughput by doing these concurrently.

On the `smallfile-25` benchmark DaisyNFS can coalesce concurrent operations and journal them together to reduce total I/O. Unlike with `smallfile-1`, lock contention dominates I/O performance so both file systems perform similarly between disk and memory (even in absolute performance).

The `largefile` benchmark is interesting because the client issues many concurrent operations, but they all conflict since they write to the same file. Linux and DaisyNFS handle this contention fine, still achieving good throughput given that all writes have to be written to both the journal and data region in both systems. However, we can see that a global txn lock actually performs *better* than the per-address locking, showing a case where with enough conflicts two-phase locking has some overhead compared to a simpler locking scheme. The seq WAL case achieves lower performance for similar reasons to `smallfile-1`, since it again alternates logging and installation rather than issues them concurrently.

The `app` benchmark is less heavy on writes, and its performance is barely affected by concurrency in DaisyNFS.

Another comparison shown in the same figure is to Linux with its log bypass optimization (the default `data=ordered` mode), where writes can be written directly to a file's data blocks rather than going through the journal. This optimization makes a difference only for asynchronous ("unstable") NFS writes, since synchronous writes are journaled even with this option. The `smallfile` benchmark has only synchronous writes, so unsurprisingly performance is the same, but on the `largefile` benchmark Linux with log bypass achieves 80% higher throughput. An improvement of up to 2× better throughput is expected here since unlike in all the other configurations, data writes are written only once rather than to both the journal and the data region, and the actual benefit is high since the benchmark is dominated by disk writes. It's worth noting that even Linux does not saturate the disk, which can achieve 500 MB/s in sequential write throughput, due to the overhead of running over NFS.

The log-bypass optimization has been verified in prior work on DFSCQ [18], a verified sequential file system. While the optimization is not out of reach for verification in Perennial, it does break the GoTxn abstraction boundary, since log-bypass writes are not persisted together with other writes in a transaction. What is much more achievable in our design is transactions with asynchronous or deferred durability, where the writes are visible after commit but persisted only later. This optimization does not make much difference for the `largefile` benchmark because the write size is large, but it is useful to combine several small writes in memory into one

large transaction.

9.2 Testing the trusted code and spec

This section answers the question, what are the trusted (assumed correct) aspects of the DaisyNFS correctness theorem? It also discusses what we did to use testing to improve confidence in these components.

For the proof of theorem 6.1 to apply to the actual server, we trust that (1) the Dafny code is a "safe" use of the transaction system, (2) sequential refinement is correctly encoded into Dafny, (3) the libraries for Go primitives are correctly specified in Dafny, and (4) the unverified Go code calling the Dafny methods and implementing the NFS wire protocol is correct. The user must follow the assumed execution model and run initialization from an empty disk and recovery after each boot. Finally, the disk needs to behave according to its assumed specification, which requires that it preserve written data and not corrupt it.

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

To evaluate the file system's correctness and compatibility as a whole, we mounted it using the Linux NFS client and ran the fsstress and fsx-linux tests, two suites used for testing the Linux kernel. In order to look for bugs in crash safety and recovery, we also ran CrashMonkey [69], which found no bugs after running all supported 2-operation tests.

While other experiments in the evaluation interact with DaisyNFS via the Linux client, these tests interact with the server directly from a hand-written client. That client is then tested with symbolic execution. This testing produces a wider range of requests than are possible via the Linux client. The process helped us find and fix several bugs in the unverified parts of DaisyNFS and in the specification itself, which are reported in fig. 9-5.

Most of the bugs in unverified code were bugs in parsing and were readily caught by symbolic execution. The bug when attempting to call REMOVE on a directory was a type-safety issue in an untested code path, which was not caught statically due to limitations in the Go type system. The concurrent write issue is a particularly interesting bug in unverified code. It only triggered when using asynchronous writes (which are unverified), but the symptom was that `git clone` resulted in incorrect data and git complained about consistency errors. The issue was that the unverified code directly passed a byte slice with the `WRITE` data from the RPC library into DaisyNFS. This buffer went through the entire stack and was eventually passed to GoTxn, which retains the buffer. Unfortunately the RPC library *also* re-uses the buffers that hold network requests, and

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

Figure 9-5: Bugs found by testing at the NFS protocol level.

eventually the buffer was corrupted due to a concurrent request. We fixed the bug by copying from the RPC before passing the buffer to the Dafny code. The subtle aspect of this bug is that it is the ownership implicit in the Dafny specification that was violated by the surrounding code, not a straightforward precondition. This was a tricky bug which Mark debugged; it helped to focus on the unverified code, but the invariant that it violated was quite subtle.

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

9.3 Incremental improvements

This section answers the question, how difficult was it to make improvements to the code and update the proofs? Put more broadly, what is the experience of making incremental changes to the proof?

DaisyNFS was implemented and verified over the course of three months by the thesis author, until it had support for enough of NFS to run. We added several features incrementally after the initial prototype worked, both to improve performance and to support more functionality. Some of the interesting changes are listed in fig. 9-6. To improve performance, we switched to operating on the serialized representation of directories directly (decoding fields on demand and encoding in-place) and then added multi-block directories. We added support for attributes

so that the file system stores the mode, uid/gid, and modification timestamp for files and directories. Finally, we implemented the freeing plan described in section 6.5.1, which required additional code through the whole stack (but by design no changes to the file-system invariant). We believe additional features such as symbolic links could be added incrementally with modest effort because of sequential reasoning and proof automation.

Feature	**Time**	**Lines**
In-place directory updates	2 days	600
Multi-block directories	5 days	800
NFS attributes	4 days	500
Freeing space (section 6.5.1)	3 days	1400

Figure 9-6: Incremental improvements to the file-system code were implemented quickly and without much code (which includes both implementation and proof).

For the `smallfile` benchmark, it took some work to ensure that the Linux NFS client issued exactly four RPCs per iteration. What this involved was returning the correct "weak cache-consistency" data from all operations, so that the client reliably caches attribute information. The RFC says these post-operation attributes are optional but encouraged for the server, but without this information in practice the Linux client issues additional `GETATTR` and `ACCESS` RPCs in every iteration of the benchmark, significantly slowing down DaisyNFS. A big part of the performance improvement on this benchmark involved returning more attribute information (at the same time we did expand the specifications to say the additional metadata is correct). These improvements were carried out in many small steps, but they were similar to the incremental improvements above in terms of taking a small amount of time and few changes to lines of proof; most of the work went into figuring out what was missing in the implementation.

GoTxn had fewer incremental changes to the code and proof after its initial development. Three changes are worth noting. First, the write-ahead log was split into three components: the on-disk circular buffer, the in-memory data structure caching the log, and finally the rest of the code integrating these components. This organization become clear while writing the specification and thinking about the proof. The write-ahead log is the largest library in GoTxn, with the largest lines of code even after splitting off these two data structures, so factoring its complexity was essential to the proof. The second change was an optimization Mark Theng made to improve performance. Formerly installation absorbed multiple writes to the same address before installing, primarily as a way of simplifying the proof, but Mark found this hurt performance since typically absorption had already occurred earlier, and the process had a non-trivial CPU cost. He was able to eliminate it and fix the proof; notably this change incrementally changed an already-complete proof. Finally, the transactional part of GoTxn implemented with two-phase locking was a later idea, which pleasantly enough was verified on top of the existing

lifting-based journaling specification without any changes to the underlying proof.

Chapter 10

Conclusion

This thesis describes an approach to verify software with a combination of concurrency in the implementation and crash-safety guarantees. The approach is applied to the DaisyNFS file system. The work spans from general verification foundations through the design and proof of the file system itself. The foundations include Perennial, a program logic for crashes and concurrency, and Goose, an approach for reasoning about Go code. DaisyNFS is designed around a verified transaction system called GoTxn, which makes it feasible to scale verification by enabling sequential reasoning for the transactions that implement the file-system operations.

10.1 Lessons learned about verification

In the process of conducting this research, we made some broader observations about verification that this section shares.

Verification helps discover accurate, precise specifications. Some systems are extremely reliable, due to extensive testing, real-world usage, and development, but they still lack a clear description of the interface exposed. The process of verification helps discover a precise, mathematical specification, and the proofs confirm that this specification is actually an accurate description of the implementation. Precise and accurate specifications are useful even for unverified software. While documentation is a helpful form of informal specification, mathematical specifications have the advantage of reducing ambiguity.

One example from the work in this thesis came up in the GoTxn write-ahead log specification. When we came up with the idea of a history of multiwrites as the specification, we able to explain absorption, where a value overwrites an older write that hasn't been logged yet, as an optimization and not a detail the caller should be concerned with. Merely writing down the specification also helped clarify some of the internal invariants. Another example was some

difficulty we ran into while interpreting the description of "weak-cache consistency (WCC)" metadata in the NFS protocol. The RFC explains this aspect in several places and combines describing what information should be returned, rationalizing the need for this feature, and describing how the client should use it. A mathematical description teases these apart so that the first is formally described and the rest are commentary on top. The end result was quite simple: the server should fetch and return the old attributes along with the post-operation attributes of whatever file or directory the WCC data pertains to. Because these attributes include modification timestamps, they permit the client to invalidate their local cache when the old timestamp doesn't match the client's cached value.

Verification guides debugging. With a verified system, bugs are still inevitable since there is unverified code surrounding the verified code, the assumptions of the proof can be violated, and the specification can be wrong. However, an advantage of formal verification, particularly fully machine-checked proofs, is that when bugs are discovered it's safe to start debugging by investigating the code *outside* the verification, including carefully looking at the specification. We ran into several bugs while developing DaisyNFS; several are described in section 9.2. Some were due to incorrect specifications; for example, at one point the `CREATE` and `MKDIR` specs did not forbid empty names. Others were due to unverified code violating the assumptions in the verified code.

Verification as an enabler. A real achievement for verification would be to use verification not just as alternative to testing, but to build something with more daring optimizations, features, or speed than would otherwise be possible. Storage systems that holds persistent state have the potential to be just such an application. There aren't that many widely-used file systems — most people use one of ext4, btrfs, or XFS on Linux, NTFS on Windows, and APFS on macOS — and new file systems are generally adopted slowly. APFS was surprising for having only a 3-year development period before Apple widely deployed it. Ext4 is the next newest of that list, introduced in 2008 (by extending ext3, which was first released in 2001). Verification has the potential to create a file system that is more rapidly adopted by virtue of its proof. This might be especially applicable for some new hardware, like a file system for persistent memory or new zoned storage (ZNS) SSDs.

While GoTxn and DaisyNFS don't have any radical optimizations, this observation did come up in our work. Making the logger and installer concurrent complicated the implementation and informal reasoning, and verification gave confidence that this was implemented correctly. Another example was the change Mark made to the installer, removing additional absorption; normally such a change would be considered subtle and would require thinking carefully about whether the code is still correct. Mark made the change and verified it within a couple weeks,

and we could accept the new code readily once the proof was complete.

Choosing what to verify carefully. In prior work on verified file systems, we typically exposed the file system via Filesystem in Userspace (FUSE). This path had two problems that switching to NFS solved. First, FUSE can be inefficient [95], adding overhead before even getting to the underlying file system's performance. Second, the API that FUSE file systems implement is not exactly the same as the POSIX or Linux APIs, so it can be challenging to know what the right specification for the file system should be and how this relates to what clients observe.

The lesson we learned here is that it is important to carefully choose what part of the system to verify, thinking about how it interfaces with the rest of the world. Both NFS and FUSE require unverified code to take system calls and invoke the server or userspace file system respectively, but NFS has the important advantage that the interface the server exposes is written down in the form of an RFC.

10.2 Discussion

Memory safety at interface boundaries. A surprisingly difficult aspect of the proof was addressing memory safety considerations while describing interfaces. DaisyNFS has two important interface boundaries, one between GooseLang and the disk, and another to describe the GoTxn interface. Both APIs involve data in the form of byte slices. The challenging aspect of using bytes in a method is specifying how *ownership* transfers. For example, the `DiskWrite(a, v)` operation passes a buffer `v` to the disk. It could be that ownership of `v` needs to be transferred to the disk, since it uses the buffer later, or that `v` should be read-only for the duration of the call and ownership is returned to the caller, or that if `v` is permanently read-only the disk and caller can share the buffer. It is important for the proof to be sound that the correct ownership discipline is enforced.

Expressing ownership as a logical idea in the logic is relatively easy using separation logic, but the semantics of `DiskWrite(a, v)` has to be given operationally and not just given as a specification. In order to simplify expressing the ownership transfer, the GooseLang `DiskRead` returns a freshly allocated buffer, but it would have been better for performance if the caller supplied a buffer that the `DiskRead` filled (as in the usual `read` system call). Similarly, the GoTxn interface to Dafny makes additional copies to simplify ownership reasoning, especially in the transaction refinement proof. A better solution would have been to develop a specification style for expressing ownership in the semantics of the operations themselves, along with associated reasoning principles.

Rust would assist with better handling of ownership in interfaces, but does not completely solve the problem. Where it would fit in is that if an interface makes *assumptions* about owner-

ship transfer, Rust would help in enforcing those assumptions at compile time. It is convenient for the verified code to know if ownership is violated before attempting a proof, but automatic enforcement would be even more important for ensuring that the code calling into a verified interface respects its ownership assumptions. What Rust doesn't solve is that it is still necessary to express ownership assumptions in the interface's semantics, a challenge independent of the implementation language.

Read-only sharing. Another challenge was reasoning about read-only sharing in the internals of GoTxn. Changing data structures after they were written to support read-only sharing was difficult, since every specification needs to incorporate fractional permissions. As a result read-only locking in GoTxn would be challenging to retrofit support for, even though it might improve read-read concurrency in GoTxn and DaisyNFS. This challenge also shows up in the data structure for the in-memory log, which needs to share read-only data blocks between threads issuing reads to the write-ahead log, the logger thread, and the installer thread.

Modular proofs. Chapter 4 describes a style for specifying a library in Perennial. Modularity was essential to enable the GoTxn proof. Better support for modularity, perhaps formalizing some of the aspects of the specification style, would have more cleanly separated each library's proofs. Where this is particularly important is in making changes to the code that affect an interface, in which case it can be difficult to tell from the code exactly what properties the caller is assuming about the interface. The specification style was developed in parallel with all the proofs, which means that proofs do not all follow the best practices developed along the way.

10.3 Future work

Apply the approach to the kernel. It would be interesting to port the Goose approach to Rust or C and then apply it to code in the kernel. Go already gets good performance, but because it has a runtime a Go library cannot be made to run in the kernel (an interesting exception is to use Goose for Biscuit [24], an OS kernel written in Go). An important challenge for verifying in-kernel code would be modeling the specifications, both those assumed by the verified code and what is promised to the rest of the kernel. Since these are trusted specifications, it would be interesting to more carefully validate them, perhaps with dynamic instrumentation.

Asynchronous model of the disk. The disk model in Goose assumes writes are durable as soon as the write returns, but real disks typically buffer writes internally. Goose has a new asynchronous disk model, and it would be good future work to port GoTxn to this new model.

Above the write-ahead log this should have no effect on the specifications. Going beyond asynchronous durability, it would also be interesting to model a different interface entirely like that of libaio where requests are submitted to a queue and completions are delivered separately.

Improve the write-ahead log. After implementing GoTxn, we made relatively few changes, as reported in section 9.3. Several improvements to the write-ahead log would be interesting to make, which were challenging due to the complexity of the existing proof.

First, it would be worth experimenting to see if the proof of installation could be separated from the rest of the proof. The important task would be identifying the right specification for the logging code which is enough to reason about the combination of installation and logging.

Second, due to the future dependency in the write-ahead log's `Read` (described in section 5.6.1), it is specified as two separate operations; it would be interesting to add *prophecy variables* to Perennial, similar to the Iris implementation [51], and give `Read` a single linearizable specification.

Finally, the write-ahead log uses physical logging where all updates represent full-block overwrites. It would be interesting to add more sophisticated operations like sub-block updates that are natively supported by the write-ahead log (rather than by the layer above). More ambitiously, it would be interesting to verify *logical logging* where updates represent high-level operations like a file write. Logical logging would require a more sophisticated specification and proof because the write-ahead log's entries would be interpreted by the caller, resulting in a higher-order specification where the caller passes a function which might need to be run during recovery, and multiple times.

Verify a file system directly on top of GoJournal. Dafny allows the DaisyNFS proof to take advantage of automation that is enabled by sequential reasoning. It would be interesting to do a direct comparison against a proof in Perennial where due to ownership the proofs would still require only sequential reasoning. The GoJournal paper reports verifying a subset of a file system directly [13], but we never attempted to apply the approach to a full-fledged file system. What would be especially interesting is if the Perennial proof started with the Dafny proof's structure and invariants; perhaps these invariants (developed under the constraints of Dafny and for automation-friendliness) would also be useful in an interactive theorem proving setting.

Do more with the NFS specification. The NFS specification for DaisyNFS is hand-written and integrated with the code. It would be an interesting project in its own right to turn this into a complete, well-tested formalization of the RFC. For example, the Dafny specification could make a serious attempt to capture all the non-determinism allowed by the RFC. As a

standalone artifact the specification could also be used to test servers (comparing to the allowed behaviors). An executable version of the specification, even if inefficient and not durable, would be valuable to test the specification itself and explore what clients do when faced with different server behavior.

Bibliography

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, April 2016. URL https://dl.acm.org/doi/10.1145/2872362.2872404.

[2] Dimitris Andreou. Striped (Guava: Google core libraries for Java 31.1). https://javadoc.io/doc/com.google.guava/guava/31.1/com/google/common/util/concurrent/Striped.html, February 2022.

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

[4] Stefan Bodenmüller, Gerhard Schellhorn, Martin Bitterlich, and Wolfgang Reif. *Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System*, pages 239–265. Springer, Cham, 2021. URL https://doi.org/10.1007/978-3-030-76020-5_14.

[5] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Security Symposium*, pages 917–934, Vancouver, Canada, August 2017.

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

[7] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[8] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1-4):367–422, June 2018.

[9] Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in Golang. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2022.

[10] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018. URL https://www.usenix.org/conference/osdi18/presentation/chajed.

[11] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019. URL https://dl.acm.org/doi/10.1145/3314221.3314585.

[12] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, October 2019. URL https://dl.acm.org/doi/10.1145/3341301.3359632.

[13] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021. URL https://www.usenix.org/conference/osdi21/presentation/chajed.

[14] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, page 418–430, Tokyo, Japan, September 2011. URL https://doi.org/10.1145/2034773.2034828.

[15] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Online, August 2020. URL https://doi.org/10.1145/3408998.

[16] Haogang Chen. *Certifying a Crash-safe File System*. PhD thesis, Massachusetts Institute of Technology, September 2016.

[17] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015. URL https://dl.acm.org/doi/10.1145/2815400.2815402.

[18] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017. URL https://dl.acm.org/doi/10.1145/3132747.3132776.

[19] Dmitri Chkliaev, Jozef Hooman, and Peter van der Stok. Serializability preserving extensions of concurrency control protocols. In *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia,*

July 6-9, 1999, Proceedings, volume 1755 of *Lecture Notes in Computer Science*, pages 180–193. Springer, 1999. URL https://doi.org/10.1007/3-540-46562-6_15.

[20] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, August 2009.

[21] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, Edinburgh, United Kingdom, July 2010.

[22] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. https://pdos.csail.mit.edu/6.828/2021/xv6.html.

[23] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 419–434, Savannah, GA, November 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/curtis-maury.

[24] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–105, Carlsbad, CA, October 2018. URL https://www.usenix.org/conference/osdi18/presentation/cutler.

[25] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 67–78, New York, NY, USA, 2010. Association for Computing Machinery. URL https://doi.org/10.1145/1693453.1693464.

[26] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528. Springer, 2010.

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

[28] Mike Eisler. XDR: External data representation standard. RFC 4506, Network Working Group, May 2006.

[29] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 73–90, San Francisco, CA, May 2019.

[30] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, January 2009.

[31] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.

[32] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, January 2019. URL https://doi.org/10.1145/3290376.

[33] Sydney Gibson. Waddle: A proven interpreter and test framework for a subset of the Go semantics. Master's thesis, Massachusetts Institute of Technology, May 2020.

[34] Google. The Go Programming Language, 2022. https://go.dev/.

[35] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight Go. In *Proceedings of the 2020 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Chicago, IL, November 2020. URL https://doi.org/10.1145/3428217.

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

[37] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. URL https://doi.org/10.1145/3062341.3062363.

[39] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, Banff, Alberta, Canada, November 2020. URL https://dl.acm.org/doi/10.5555/3488766.3488772.

[40] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016. ISBN 1107150302.

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

pages 1–17, Monterey, CA, October 2015. URL https://dl.acm.org/doi/10.1145/2815400.2815428.

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

[43] Dave Hitz, Michael Malcolm, and James Lau. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference*, San Francisco, CA, January 1994. USENIX Association. URL https://www.usenix.org/conference/usenix-winter-1994-technical-conference/file-system-design-nfs-file-server-appliance.

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

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

[46] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, Santa Clara, CA, February 2015. USENIX Association. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

[47] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[48] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015. URL https://dl.acm.org/doi/10.1145/2775051.2676980.

[49] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of the 45th ACM Symposium on Principles of Programming Languages (POPL)*, Los Angeles, CA, January 2018. URL https://dl.acm.org/doi/10.1145/3158154.

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

[51] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic.

In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, pages 45:1–45:32, New Orleans, LA, January 2020. URL https://dl.acm.org/doi/10.1145/3371113.

[52] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019. URL https://dl.acm.org/doi/10.1145/3341301.3359662.

[53] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. PerSeVerE: Persistency semantics for verification under ext4. In *Proceedings of the 48th ACM Symposium on Principles of Programming Languages (POPL)*, Online, January 2021. URL https://doi.org/10.1145/3434324.

[54] Alex Konradi. Performance Optimization of the VDFS Verified File System. Master's thesis, Massachusetts Institute of Technology, June 2017.

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

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

[57] Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, October 2015.

[58] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, January 2017. URL https://dl.acm.org/doi/10.1145/3009837.3009855.

[59] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 77:1–30, St. Louis, MO, September 2018. URL https://dl.acm.org/doi/10.1145/3236772.

[60] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery. URL https://doi.org/10.1145/3341301.3359635.

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

[62] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, page 516–530, Berlin, Heidelberg, 2012. Springer-Verlag. URL https://doi.org/10.1007/978-3-642-32940-1_36.

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

[64] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020. URL https://dl.acm.org/doi/10.1145/3385412.3385971.

[65] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, February 2013. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu.

[66] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems*, 43 (4), October 2021. URL https://doi.org/10.1145/3462205.

[67] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2022. URL https://people.mpi-sws.org/~dreyer/papers/rusthornbelt/paper.pdf.

[68] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

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

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

[71] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 89–102, Nara, Japan, September 2016.

[72] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, January 2019. ISSN 0001-0782. URL https://doi.org/10.1145/3211968.

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

[74] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

[75] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, October 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai.

[76] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. In *Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Indianapolis, IN, October 2013. URL https://doi.org/10.1145/2544173.2509536.

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

[78] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Oxford, United Kingdom, September 2017.

[79] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, January 2019. URL https://doi.org/10.1145/3371079.

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

[81] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending Intel-x86 consistency and persistency: Formalising the semantics of Intel-x86 memory types and non-temporal stores. In *Proceedings of the 49th ACM Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, January 2022. URL https://doi.org/10.1145/3498683.

[82] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Prin-*

ciples (SOSP), pages 38–53, Monterey, CA, October 2015. URL https://dl.acm.org/doi/10.1145/2815400.2815411.

[83] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 158–174, Virtual, June 2021. URL https://doi.org/10.1145/3453483.3454036.

[84] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[85] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional isolation for block storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 23–37, Santa Clara, CA, February 2016. USENIX Association. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/shin.

[86] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson.

[87] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 14–27, 2018. URL https://doi.org/10.1145/3167092.

[88] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: Resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 80–95, New York, NY, USA, 2021. Association for Computing Machinery. URL https://doi.org/10.1145/3453483.3454031.

[89] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification version 2. RFC 1057, Network Working Group, June 1988.

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

[91] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Spiridonova Irina, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*,

San Diego, CA, June 2022. URL https://www.fstar-lang.org/papers/EverParse3D.pdf.

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

[93] Mark Theng. GoTxn: Verifying a crash-safe, concurrent transaction system. Master's thesis, Massachusetts Institute of Technology, January 2022.

[94] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 377–390, Boston, MA, September 2013. URL https://dl.acm.org/doi/10.1145/2500365.2500600.

[95] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, page 59–72, Santa Clara, CA, February–March 2017. URL https://dl.acm.org/doi/10.5555/3129633.3129640.

[96] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 818–834, San Francisco, CA, May 2019. URL https://doi.org/10.1109/SP.2019.00035.

[97] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006. URL https://www.usenix.org/conference/osdi-06/explode-lightweight-general-system-finding-serious-storage-system-errors.

[98] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helper for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019. URL https://dl.acm.org/doi/10.1145/3341301.3359644.