GoTxn: Verifying a Crash-Safe, Concurrent Transaction System

by
Mark Theng
B.S. Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2022

Author .....
Department of Electrical Engineering and Computer Science
January 4, 2022

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Professor
Thesis Supervisor

Certified by .....
Tej Chajed
Doctoral Student
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

GoTxn: Verifying a Crash-Safe, Concurrent Transaction System

by

Mark Theng

Submitted to the Department of Electrical Engineering and Computer Science
on January 4, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Bugs related to concurrency and crash safety are infamous for being subtle and hard to reproduce. Formal verification provides a way to combat such bugs through the use of machine-checked proofs about program behavior. However, reasoning about concurrency and crashes can be tricky, especially when scaling up to larger systems that must also have good performance.

This thesis discusses the verification of GoTxn, the concurrent, crash-safe transaction system underlying the verified Network File System (NFS) server DaisyNFS. It focuses on the specification and proof of the write-ahead log and the automatic two-phase locking interface used to enforce crash and concurrent atomicity in transactions, detailing how the verification framework Perennial can be used to manage assertions about crash behavior across multiple threads. By effectively harnessing concurrency to hide disk access latency, GoTxn enables performance in DaisyNFS similar to the unverified Linux NFS server.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Thesis Supervisor: Tej Chajed
Title: Doctoral Student

Acknowledgments

I would like to thank Tej Chajed, Frans Kaashoek, Nickolai Zeldovich, Joe Tassarotti and Ralf Jung for their contributions to this project. The development of GoTxn was a collaborative effort, and would not have been possible without their efforts. I would especially like to thank Tej for his foundational work on concurrent, crash-safe systems verification, on which this work is based; Joe and Ralf for their work on program reasoning techniques central to this effort; and Frans and Nickolai for their inspiration, direction, and invaluable guidance in developing this thesis. Special thanks also to my friends, family, and everyone else at MIT who have supported me throughout this work.

Contents

List of Figures

Chapter 1

Introduction

Atomicity is a powerful tool for building and reasoning about concurrent, crash safe file systems. Consider the creation of a file f inside a directory d. This involves creating an inode for f, reading d's inode, and then writing d's inode back to disk with a new entry for f. To maintain consistency, a file system requires that these operations behave as a single atomic transaction. Otherwise, it might end up dereferencing an invalid pointer or creating a dead inode.

Two key forms of atomicity that a file system might require are:

- **Crash atomicity.** If the system crashes, writes in a transaction should either be all discarded or all committed to disk.
- **Concurrent atomicity.** Concurrent transactions should be *serializable* in that they should give the same results as if they were executed sequentially.

One way to provide atomicity is to build the file system on top of a transaction system. Bugs in these transaction systems can be subtle, costly, and hard to reproduce, making them a good target for formal verification. However, the subtle reasoning required to verify a transaction system makes doing so a challenging task.

This thesis describes the verification of GoTxn, a transaction system that aims to be a step towards this goal. GoTxn supports the basic features of a practical transaction system, including:

- **Crash safety.** Write-ahead logging provides crash recovery that maintains transaction atomicity.
- **Concurrent execution.** Transactions are executed concurrently but safely to maintain good performance without breaking atomicity.
- **Disk performance.** Disk operations are batched, absorbed and performed concurrently when possible to achieve good performance.
- **Object-based interface.** Transactions operate on the level of objects, which are variable-sized units of data potentially smaller than a disk block. Concurrent transactions can safely access different objects in the same block.
- **Compiled language.** The Go language provides compiled language performance unavailable to verified systems that rely on extraction to a functional language.

On top of this, GoTxn offers a lock management interface that guarantees transaction serializability, allowing developers to reason about concurrently-executed transactions as sequential code. This provides access to the high degree of proof automation available for sequential code, allowing for the faster iteration times necessary to reach practical levels of complexity.

1.1 System Overview

GoTxn is built in layers (left side of Fig. 1-1), each exposing a small set of interface methods to the layer above. The lowest layer interacts directly with the disk, itself modeled as a layer that exposes atomic reads and writes to disk blocks. The layers of GoTxn are:

- **The write-ahead log (WAL) layer.** The WAL layer provides crash atomicity for groups of disk updates, called *multiwrites*, by recording them to an on-disk log before installing them to their target locations.

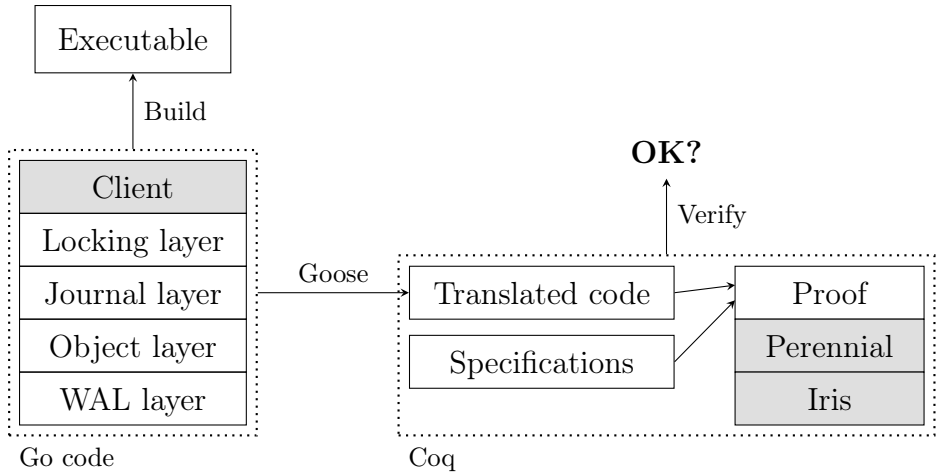

Figure 1-1: Overview of GoTxn (left) and its verification workflow (right).

- **The object layer.** The object layer allows multiwrites to be expressed in terms of object reads and writes instead of operations on entire disk blocks.
- **The journal layer.** The journal layer provides a local transaction buffer for each transaction, allowing transactions to be built up over time before being committed as a single multiwrite.
- **The locking layer.** The locking layer implements automatic two-phase locking to guarantee transaction serializability and isolate concurrent reasoning from the developer.

GoTxn records all disk writes to the on-disk log and guarantees that transactions persist across crashes immediately after they are committed. This means that it does not support log-bypass writes and deferred durability, among other features common in unverified systems like the Linux kernel journaling layer jbd2.

1.2 Verification

To verify GoTxn, we start by writing *specifications* for each layer's interface calls. Each specification is an intuitive but relatively complete description of client-visible behavior, including concurrency and crash-safety guarantees. We then write mathematical *proofs* showing that the implementation of each layer meets its specifications,

given the specifications of the layer below. Clients of GoTxn can similarly be verified as layers on top of GoTxn, with its own specifications and proofs.

We machine-check these proofs in the interactive proof assistant Coq [28] with the help of Perennial [5] (right side of Fig. 1-1), a framework that extends the concurrency verification system Iris [17] to support crash reasoning. As part of this process, we use the Goose [6] translation tool to make GoTxn's Go code accessible to reasoning in Coq.

1.3 Thesis Outline

This thesis focuses on the verification of the WAL and locking layers, with particular attention to the techniques used to manage the complexity of reasoning about crash safety and concurrency in GoTxn. For clarity, the proofs and specifications presented here have been slightly simplified. The full Coq proofs and specifications can be accessed online at the address `https://github.com/mit-pdos/perennial`

The rest of the thesis is organized in the following manner:

- Chapter 2 discusses related work.
- Chapter 3 provides background on the proof techniques used in this work.
- Chapter 4 discusses the verification of the WAL layer.
- Chapter 5 discusses the verification of the locking layer.
- Chapter 6 evaluates the usefulness of GoTxn and its WAL, both in terms of performance and verification guarantees.
- Chapter 7 provides a conclusion to the thesis.

Chapter 2

Related Work

This thesis includes work directly contributing to and extending GoJournal [7], the system making up the WAL, object and journal layers of GoTxn. GoTxn is in turn used as a foundation for DaisyNFS [8], a concurrent Network File System (NFS) server verified with the help of the sequential, verification-aware programming language Dafny [18]. DaisyNFS serves as an example of how the sequential interface of GoTxn's locking layer makes powerful sequential proof automation available to the verification of crash-safe, concurrent programs.

GoTxn relies on Perennial [5] to enable concurrent, crash-safe reasoning. An earlier version of Perennial was used to prove concurrent crash safety specifications for a mail server called Mailboat [5]. However, Mailboat is considerably simpler than GoTxn since it operates on independent file objects and so does not need a journaling system.

2.1 Verified Write-Ahead Logs

FSCQ [10], Argosy [4] and Yxv6 [25] also feature verified WALs to provide crash safety guarantees. However, they use simpler, sequential implementations that only process one multiwrite at a time, limiting their performance. DFSCQ [9] includes a more complex verified WAL that allows multiwrites to be committed in groups, but does not support concurrent execution. GoTxn's WAL accepts, logs and installs

multiwrites concurrently, requiring more advanced proof techniques.

Fault-Tolerant Concurrent Separation Logic (FTCSL) [23] is a framework developed to support proofs involving both concurrency and crash safety by extending the concurrent separation logic framework Views [14]. It was used to prove part of the WAL-based ARIES recovery algorithm [21]. However, FTCSL has only been used in pen-and-paper proofs, leaving open the possibility of errors that could render the proofs invalid and limiting its usefulness in proving large, practical programs. In fact, the FTCSL proofs provided for the ARIES algorithm cover only crash-safety and do not involve concurrent reasoning.

2.2 Verified Concurrency Isolation

Concurrent program verification frameworks such as CSPEC [3], CIVL [16] and Armada [20] can, like GoTxn, be used to reduce the behavior of concurrent code to serially executed sequential code. However, they require the developer to perform concurrent reasoning to complete this reduction. GoTxn instead uses two-phase locking to guarantee atomicity in its specification, completely isolating concurrent reasoning from the developer. This not only makes verification more practical for large programs, but also simplifies reasoning about crash safety.

Client code in IronFleet [15] can be shown to be atomic if the code satisfies a sequentially-checkable ordering constraint. GoTxn does not require a similar constraint on transaction code since it enforces two-phase locking as part of its transaction interface. IronFleet also assumes a distributed system environment where concurrent programs share resources by passing messages over the network, in contrast to GoTxn where threads communicate by directly accessing shared disk and memory resources.

The serializability of two-phase locking itself has been proven with manual and machine-checked proofs in works such as those by Chkliaev et al. [11], Attiya et al. [1] and Pollak [24]. In a similar vein, Lesani et al. [19] developed a system to verify software transactional memory algorithms, which they used to verify an implementation of the NOrec algorithm [13]. However, these proofs were done outside the context of

a larger transaction system that also provides crash safety.

Chapter 3

Background

The proof of GoTxn relies on separation logic [2], a methodology for reasoning about concurrent programs. It is based on the idea that threads can be analyzed independently when they do not share any disk or memory resources. By tracking which resources each thread uses and how, we can prove properties about concurrent programs with proof structures that parallel code. This chapter provides the background necessary to understand proofs described in subsequent chapters.

3.1 Resource Logic

Separation logic tracks resource use by introducing the concept of *ownership*. A thread's ownership over a resource acts as a guarantee that no other thread will access that resource. In the simplest case, a proof assigns different threads ownership over non-overlapping sets of resources, and then steps through their code to show that they only access those resources. By also tracking what happens to the values held by those resources, we can prove assertions about how each thread affects program state.

Concretely, ownership over a resource a holding value v is represented by the *predicate* $a \mapsto v$. These can be combined with the *separation conjunction operator* $*$. For example, the predicate $a_1 \mapsto v_1 * a_2 \mapsto v_2$ represents ownership over both a_1 and a_2. Similarly, constraints over values can be represented by ordinary logical

predicates. For example, $\exists v,\ a \mapsto v * v \geq 10$ asserts ownership over a with value at least 10. A separation logic proof associates each thread execution state with a *proof state* predicate representing assertions about the resources owned by the thread at that point in execution.

While the operator $*$ may seem similar to the classical logical conjunction operator $\wedge$, there is one important difference. When a thread forks, the resources of its proof state P must be distributed to the proof states P_i of its children. In doing so, the proof must show that it does not distribute the same resource to multiple P_i. To make such constraints easy to reason about, we define $P * Q$ to represent not just the logical assertions of both P and Q, but also the *disjoint union* of the ownership claims of P and Q. This means that it is always valid to assign each child i the proof state P_i when forking from a proof state $P := \divideontimes_i P_i$.

Weakening is an important concept when reasoning about and transforming proof state. A predicate Q is said to be *weaker* than a predicate P if the assertions of P together imply the assertions of Q. This is denoted $P \vdash Q$, for example

$$a \mapsto v * v < 5 \vdash a \mapsto v * v < 10$$

In keeping with the definition of $*$, we generally cannot say that $P \vdash P * P$ unless P does not contain any ownership claims.

A *Hoare triple* specifies what resources a piece of code requires and its behavior with respect to those resources. A piece of code C that transforms a *pre-condition* P_{pre} to a *post-condition* P_{post} can be given the Hoare triple $\{P_{pre}\}\ C\ \{P_{post}\}$. For example, the Hoare triple $\{\exists v,\ x \mapsto v\}$ `*x = 10` $\{x \mapsto 10\}$ specifies that the memory write `*x = 10` requires ownership over `x` and changes its value to 10. If a proof state just before C can be written $P * P_{pre}$, then the proof can *apply* the Hoare triple $\{P_{pre}\}\ C\ \{P_{post}\}$ to give the proof state $P * P_{post}$ after C.

More generally, Hoare triples of the form $\{P_{pre}\}\ M\ \{\mathbf{RET}\, r;\ P_{post}(r)\}$ can be specified for a method M, where P_{post} may depend on the return value r. For example, a method that computes square roots might be given the Hoare triple $\{\varnothing\}$ `sqrt`(x) $\{\mathbf{RET}\, r;\ r = \sqrt{x}\}$, where $\varnothing$ is the empty assertion.

To show that a piece of code satisfies a Hoare triple, a proof starts with a proof state containing exactly the resources in the Hoare triple's pre-condition. Then, it steps through the code's execution, using Hoare triples for smaller code elements to transform the proof state. Finally, it shows that the proof state at every exit point satisfies the post-condition.

A natural extension to the concept of ownership is *fractional ownership*, which implements the single-writer, multiple-reader model of concurrency control by allowing threads to have less than full ownership over resources. Fractional ownership over a resource a is represented by a predicate of the form $a \mapsto_q v$, where $0 < q \leq 1$ denotes the amount of ownership the thread has over a with $q = 1$ denoting full ownership. At any time, a proof may split a predicate $a \mapsto_q v$ into multiple pieces $a \mapsto_{q_i} v$ where $\sum_i q_i = q$, or, similarly, combine multiple pieces $a \mapsto_{q_i} v$ into $a \mapsto_q v$. These rules guarantee that the sum of q values across all threads for any resource will always be at most one.

A thread may read from a resource if it has any partial ownership over it, but may only write to resource if it has full ownership over it. This leads to the following properties:

- Full ownership over a resource guarantees exclusive access. In other words, no other thread may concurrently access a resource while a thread is writing to it.
- Partial ownership over a resource guarantees that no other thread may modify its value.
- If a proof state contains multiple predicates $a \mapsto_{q_i} v_i$ for the same resource, then all the v_i must be equal.

3.2 Invariants

Systems often make use of atomic mechanisms to allow concurrent threads to safely access the same resource. For example, GoTxn relies on the fact that disk operations

Proof progress	**Proof state**
Initial state	$\boxed{I}$
After opening I	$\exists v,\ c \mapsto v * v \geq 0$
After incrementing c	$\exists v,\ c \mapsto v + 1 * v \geq 0$
(Weakening)	$\exists v,\ c \mapsto v + 1 * v + 1 \geq 0$
After closing I	$\boxed{I}$

Figure 3-1: Proof that atomically incrementing c maintains the invariant $I := \exists v,\ c \mapsto v * v \geq 0$.

are atomic by specification. We can reason about such mechanisms with the use of *invariants*, which are predicates safely shared across multiple threads.

At any time, a proof may seal a predicate I into an invariant $\boxed{I}$. $\boxed{I}$ is *persistent*, meaning that it can be duplicated and shared across multiple threads. Later, the proof may *open* $\boxed{I}$ to gain access to I, but only for a single atomic step. After the atomic step, the proof must *close* $\boxed{I}$ by giving up I. In analogy to the classical notion of invariants, this guarantees that I remains satisfied for the rest of the program's execution.

A similar mechanism is the *lock invariant*, which allows us to apply atomic reasoning to locked regions. A lock invariant $\boxed{I_L}^L$ holds resources protected by its associated lock L. Under $\boxed{I_L}^L$, a thread gains access to I_L when it acquires L and must give up I_L when it releases L. This guarantees that I_L remains satisfied whenever L is not being held.

To illustrate how invariants work, consider a function `Inc`() that atomically increments a signed counter c. If c starts at zero and is only ever modified during calls to `Inc`, we can show that c remains non-negative throughout the execution of the program by constructing the invariant $I := \exists v,\ c \mapsto v * v \geq 0$ and proving (as detailed in Fig. 3-1) the Hoare triple

$$\left\{\boxed{I}\right\}\ \texttt{Inc}()\ \left\{\varnothing\right\} \tag{3.1}$$

Notably, there is no need to include $\boxed{I}$ in the post-condition of Eq. 3.1. The

requirement that $\boxed{I}$ must be closed every time it is open already guarantees that $\boxed{I}$ continues to hold after the call to `Inc`. This is encoded in the persistence of $\boxed{I}$, which allows the proof to retain access to $\boxed{I}$ by duplicating it before applying Eq. 3.1.

We can make stronger assertions about c by relating its value to thread resources. For example, suppose a program performs concurrent reads from c but only ever calls `Inc` from the same thread t. We can reason about the value of c by splitting its ownership between an invariant and t. Specifically, we construct the invariant $I := \exists v,\ c \mapsto_{1/2} v$ and prove the Hoare triple

$$\left\{\boxed{I} * c \mapsto_{1/2} v\right\}\ \texttt{Inc}()\ \left\{c \mapsto_{1/2} v + 1\right\}$$

In this configuration, t may be said to *control* c — while t may open $\boxed{I}$ to gain write access to c, other threads may gain at most read access. After n calls to `Inc`, the proof may use the predicate $c \mapsto_{1/2} n$ from t to conclude that the final value of c is n.

We would like to extend this approach to show that the final value of c is n even if the calls to `Inc` are made concurrently, by different threads. However, we cannot do so directly since multiple threads cannot be given simultaneous control over c. Instead, we use *ghost variables* to redefine the ways in which a thread may control c.

Ghost variables are virtual resources defined solely for the sake of the proof. They do not correspond to actual physical state or code constructs but may be owned by threads and invariants just like ordinary resources. A proof may update a ghost variable γ whenever it has full ownership over γ. If an invariant $\boxed{I}$ relates γ to a physical resource a, then whenever the proof opens $\boxed{I}$ to perform a write to a, it may also be forced to update γ in order to close $\boxed{I}$.

In our example, we give each thread t_i control over an independent additive portion v_i of c through a ghost variable γ_i (Fig. 3-2). Specifically, we split ownership over each γ_i between t_i and the invariant

$$I := \exists v_1 \exists v_2 \ldots \exists v_n,\ \left(\mathop{\Large *}_{i=1}^{n} \gamma_i \mapsto_{1/2} v_i\right) * c \mapsto \sum_{i=1}^{n} v_i$$

by specifying the Hoare triples

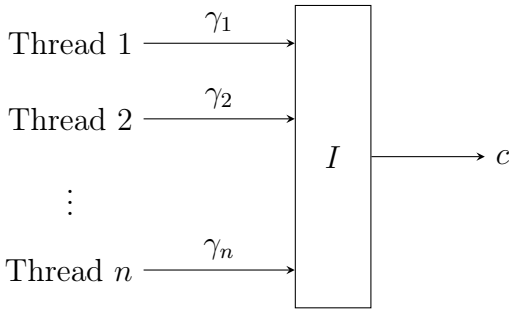

Figure 3-2: n threads share control over c through ghost variables γ_i.

$$\forall i,\ \left\{\boxed{I} * \gamma_i \mapsto_{1/2} v\right\}\ \texttt{Inc}()\ \left\{\gamma_i \mapsto_{1/2} v + 1\right\} \tag{3.2}$$

After incrementing c, the proof of `Inc` for each t_i must increment γ_i in order to close $\boxed{I}$ and prove the post-condition. Combining $\boxed{I}$ with the final value of γ_i from all threads allows us to deduce that the final value of c is n.

A *monotonic counter* is another example of a of virtual resource. Ownership over a monotonic counter γ with value v is denoted $\gamma \mapsto^{+} v$. Similar to a ghost variable, a proof may update v whenever it has full ownership over γ. However, $\gamma \mapsto^{+} v$ comes with the restriction that v can only ever be increased. This means that any partial ownership $\gamma \mapsto^{+}_{q} v$ over γ guarantees that γ remains at least v even after the proof rescinds all ownership over γ. We express this guarantee with a persistent *lower bound predicate* $\gamma \mapsto^{\geq} v$. If the proof later regains access to γ through a predicate of the form $\gamma \mapsto^{+}_{q'} v'$, it can use $\gamma \mapsto^{\geq} v$ to conclude that $v' \geq v$.

Similarly, a *monotonic list* is a virtual resource holding a list that can only ever be appended to. Partial ownership $\gamma \mapsto^{+}_{q} v$ over a monotonic list γ guarantees that the first $\mathbf{len}\, v$ entries of γ remain unchanged even after the proof rescinds all ownership over γ. We express this guarantee with persistent predicates $\gamma \mapsto^{[i]} v[i]$ where $0 \leq i < \mathbf{len}\, v$.

3.3 Logical Atomicity

While invariants can be used to prove powerful assertions about the effects of atomic operations, it is often useful to write specifications that directly expose atomicity itself. For example, GoTxn allows the developer to write arbitrary transactions and call

them in any order. This means that its top-level specifications cannot be full-system assertions like those discussed in the previous section. Rather, its specifications can only guarantee that executing a transaction atomically induces a corresponding transformation on some part of the full system state.

To make this concrete, let us specify the function `Inc()` from the previous section as an atomic transformation $v \leftarrow v + 1$ of the value v of c. We start by constructing the internal invariant

$$I_{int} := \exists v,\ c \mapsto v * P_{conn}(v)$$

where the *connecting predicate* $P_{conn}(v)$ is a caller-defined predicate provided as a parameter to the proof. As the discussion in the previous section shows, our choice of invariant depends on the properties we would like to prove about the larger system. Keeping P_{conn} caller-defined gives us the freedom to adapt the proof of `Inc` to different system specifications. For example, we might choose $P_{conn}(v) := v \geq 0$ to show that v remains non-negative throughout the execution of the program.

Now, in order to close $\boxed{I_{int}}$ after incrementing c, the proof of `Inc` must transform $P_{conn}(v)$ into $P_{conn}(v+1)$. Since P_{conn} is caller-defined, the proof that this transformation is valid must be left as a caller obligation. With this in mind, an initial attempt at a specification for `Inc` might look like

$$\left\{\boxed{I_{int}} * \left(\forall v,\ P_{conn}(v) \vdash P_{conn}(v+1)\right)\right\}\ \texttt{Inc}()\ \left\{\varnothing\right\} \tag{3.3}$$

While this works for simple assertions like $P_{conn}(v) := v \geq 0$, it is often necessary to relate v to thread resources. For example, suppose we would like to show that n concurrent calls to `Inc` increases v by n. Following Sec. 3.2, we give each thread t_i control over a ghost variable γ_i and set

$$P_{conn}(v) := \exists v_1 \exists v_2 \ldots \exists v_n,\ \left(\mathop{\Large *}_{i=1}^{n} \gamma_i \mapsto_{1/2} v_i\right) * v = \sum_{i=1}^{n} v_i$$

Now, when transforming $P_{conn}(v)$ to $P_{conn}(v+1)$ during t_i's call to `Inc`, the proof must also increment γ_i by combining the half-ownership predicates over γ_i from t_i and from $P_{conn}(v)$. In a more complex system, updating $P_{conn}(v)$ might even require

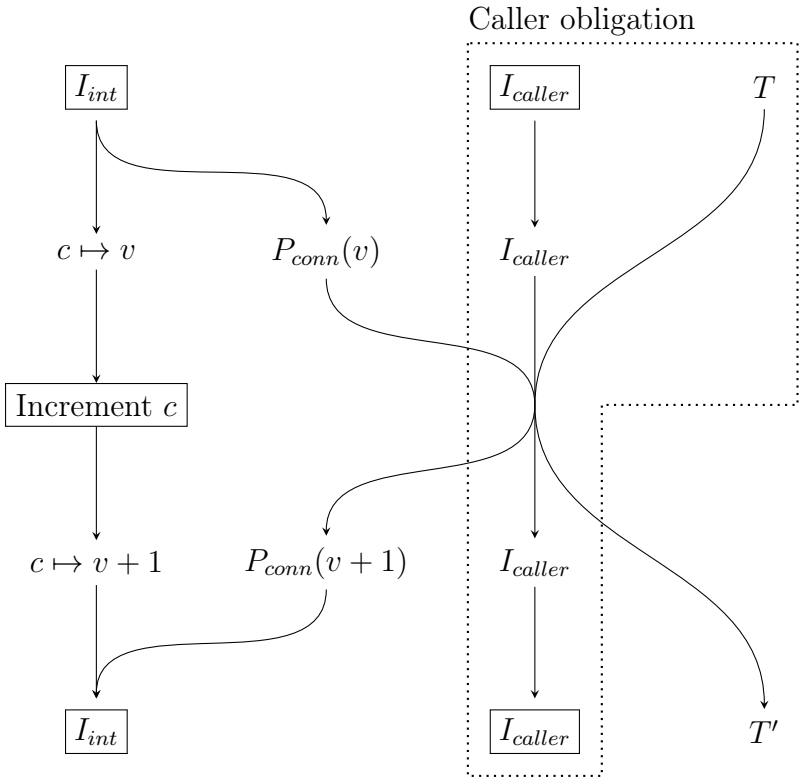

Figure 3-3: Outline of a proof for `Inc`. The caller is responsible for showing that $P_{conn}(v)$ may be transformed into $P_{conn}(v+1)$, possibly by opening and closing some caller invariant $\boxed{I_{caller}}$ and transforming some thread state predicate T into T'. This obligation, expressed by the pre-condition $\forall v,\ P_{conn}(v) \mathrel{\Rrightarrow\!\!\!\!\!\!\ast}\ P_{conn}(v+1) * T'$, corresponds to the section marked out by the dotted lines.

accessing ghost variables from a caller invariant.

To support specifications like these, we define the *fancy update* operator $\Rrightarrow\!\!\!\!\!\!\ast$. The predicate $P_{pre} \Rrightarrow\!\!\!\!\!\!\ast\ P_{post}$ can be thought of a black box holding an unknown predicate P satisfying the constraint that $P_{pre} * P$ can be transformed into P_{post} by updating ghost variables or opening and closing invariants. In other words, it represents a single-use "callback" that can be passed as a pre-condition to a Hoare specification, encoding a transformation to be applied in conjunction with an atomic update. We can now construct the Hoare specification

$$\forall T \forall T',\ \left\{\boxed{I_{int}} * T * \left(\forall v,\ P_{conn}(v) * T \Rrightarrow\!\!\!\!\!\!\ast\ P_{conn}(v+1) * T'\right)\right\}\ \texttt{Inc}()\ \left\{T'\right\}$$

or, more concisely,

$$\forall T',\ \left\{\boxed{I_{int}} * \left(\forall v,\ P_{conn}(v) \Rrightarrow\!\!\!\!\!\!\ast\ P_{conn}(v+1) * T'\right)\right\}\ \texttt{Inc}()\ \left\{T'\right\} \tag{3.4}$$

where $P_{conn}(v) \Rrightarrow\!\!\!\!\!\!\ast\ P_{conn}(v+1) * T'$ can be thought of as the result of "pre-applying" $P_{conn}(v) * T \Rrightarrow\!\!\!\!\!\!\ast\ P_{conn}(v+1) * T'$ to T (Fig. 3-3). Here, T is the caller-defined thread state to be transformed into T' during the update to P_{conn}. In the example of n concurrent calls to `Inc`, the proof sets $T := \gamma_i \mapsto_{1/2} 0$ and $T' := \gamma_i \mapsto_{1/2} 1$ during each thread t_i's call to `Inc` to show that the final value of each γ_i is 1.

Eq. 3.4 is an example of a *HOCAP-style* specification [26]. A HOCAP-style specification reduces the behavior of a piece of code to a single, instantaneous update to caller-visible state applied at some point during the code's runtime. The points at which these updates are applied can be thought of as "commit points" that give an ordering for an equivalent serial execution (Fig. 3-4).

In general, if an atomic operation `M`, possibly non-deterministic, can be described by a *transition relation* $R(\sigma_1, \sigma_2)$ that specifies whether `M` might induce the transition $\sigma_1 \to \sigma_2$ on some caller-visible state σ, we can describe its behavior with a Hoare triple of the form

$$\forall Q,\ \left\{\boxed{I_{int}} * \forall \sigma_1 \forall \sigma_2,\ R(\sigma_1, \sigma_2) \implies \left(P_{conn}(\sigma_1) \Rrightarrow\!\!\!\!\!\!\ast\ P_{conn}(\sigma_2) * Q\right)\right\}\ \texttt{M}\ \left\{Q\right\} \tag{3.5}$$

More broadly, a system interface may define multiple methods that atomically up-

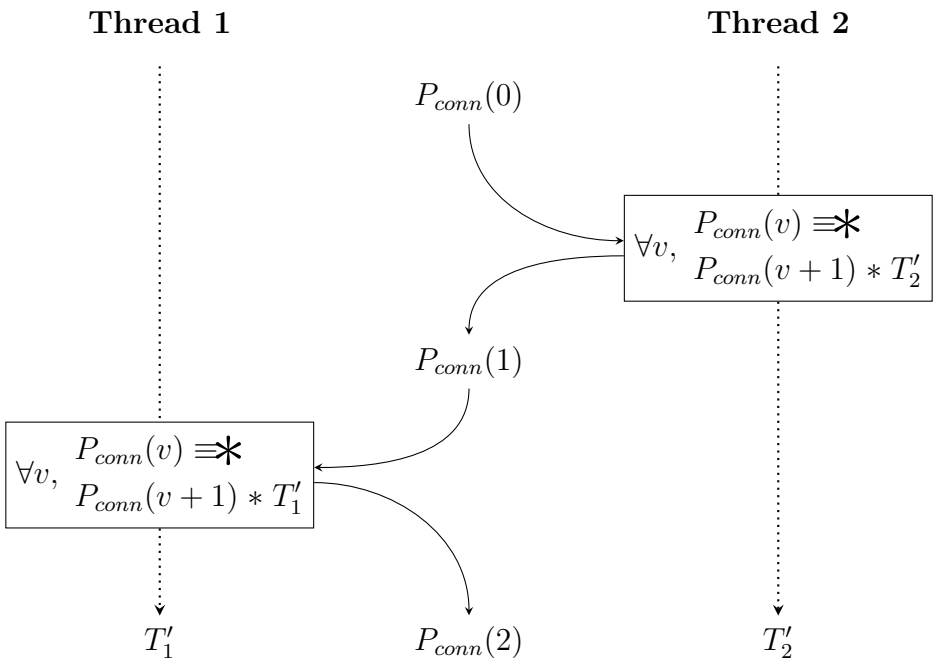

Figure 3-4: The HOCAP specification of `Inc` reduces its behavior to a single atomic update to the caller-visible state variable v. Here, two concurrent calls to `Inc` are modeled as atomic updates each increasing v by one. The specification of `Inc` guarantees that the updates are applied during their respective thread's call to `Inc`, but not that they are applied in any particular order.

date the same caller-visible state. We can model such an interface as an *atomic transition system* by providing HOCAP-style specifications for every interface method.

3.4 Crashes

To specify the crash behavior of a piece of code C, we extend its Hoare triple into a *Hoare quadruple* $\{P_{pre}\}\ C\ \{P_{post}\}\{P_{crash}\}$ by adding a *crash condition* P_{crash} specifying the resources and constraints that can be recovered if a crash occurs at any time during the execution of C. For example, consider a function `Inc()` that atomically increments an on-disk counter c. After a crash, c may hold different values depending on whether the crash happened before or after the operation completes. This gives the natural specification

$$\{c \mapsto v\}\ \texttt{Inc}()\ \{c \mapsto v+1\}\{c \mapsto v \vee c \mapsto v+1\}$$

Just like the proof of a Hoare triple, the proof of a Hoare quadruple for a piece of code C may apply Hoare quadruples for smaller code elements to step through C and transform the proof state. Each time the proof applies a Hoare quadruple $\{P'_{pre}\}\ C'\ \{P'_{post}\}\{P'_{crash}\}$ to a proof state $P_1 * P_2$ where $P_2 \vdash P'_{pre}$, it must additionally show that the crash condition P_{crash} of C remains satisfied throughout the execution of C' by showing that $P_1 * P'_{crash} \Rrightarrow P_{crash}$.

Proving Hoare quadruples in this manner can impose an unnecessarily high proof burden. One observation that makes them less tedious to prove is that invariants play a similar role to crash conditions. In particular, a proof may reduce a crash specification into a crashless specification by putting its crash condition into an invariant.

An extension of this idea is the *crash borrow* [27]. Whenever its proof state contains a predicate P such that $P \vdash P_c$, a proof may remove P_c from its crash condition by sealing P in a crash borrow $\boxed{P \mid P_c}$. If the proof later needs access to P, it must *open* the crash borrow by adding P_c back to its crash condition.

Unlike an invariant, $\boxed{P \mid P_c}$ is not persistent. However, the proof can still share $\boxed{P \mid P_c}$ between threads by sealing it in an invariant. Another thread may then

interact with P within an atomic step as long as it maintains P_c in its crash condition while doing so. Since P_c is always either satisfied by P or part of some thread's crash condition, it must remain satisfied throughout the execution of the program.

Crash safety for an atomic transition system can be specified by a transition relation $R_{crash}(\sigma_1, \sigma_2)$ describing a crash followed by recovery. To prove such a specification, suppose a system has recovery procedure `Recover`() and system invariant $\boxed{I}$ with connecting predicate $P_{conn}(\sigma)$ exposing caller-visible state σ. We start by defining a crash condition P_{crash} such that $I \vdash P_{crash}$. P_{crash} should generally contain $P_{conn}(\sigma)$ as well as assertions from $\boxed{I}$ about on-disk resources and how they relate to σ.

Since $I \vdash P_{crash}$, if the system crashes any time after $\boxed{I}$ is constructed, P_{crash} holds and can be used as a pre-condition for `Recover`. The proof of `Recover` must then use P_{crash} to reconstruct $\boxed{I}$ and, in a similar vein to Eq. 3.5, apply the atomic update

$$P_{cupd}(Q) := \forall \sigma_1 \forall \sigma_2,\ R_{crash}(\sigma_1, \sigma_2) \implies \left(P_{conn}(\sigma_1) \Rrightarrow P_{conn}(\sigma_2) * Q\right)$$

This is summarized by the Hoare quadruple

$$\forall Q,\ \left\{P_{crash} * P_{cupd}(Q)\right\}\ \texttt{Recover}()\ \left\{\boxed{I} * Q\right\}\left\{P_{crash}\right\}$$

where `Recover` is additionally required to satisfy P_{crash} on crash to ensure that the system recovers safely even if it crashes in the middle of recovery before $\boxed{I}$ is established.

Chapter 4

The WAL Layer

The WAL layer provides crash atomicity for disk multiwrites by atomically recording them to an on-disk log before installing them to disk. After a crash, the log can be used to complete any incomplete multiwrites that were in the process of being installed. Multiwrites recorded to the log are *durable* in the sense that they persist across crashes. Multiwrites not yet recorded to the log are completely discarded on crash.

The log itself, called the *physical log*, is a circular queue located in a dedicated section of disk. It is implemented as a buffer of disk updates along with two header blocks containing its start and end pointers (Fig. 4-1). After a crash, only the updates between the start and end pointers are used for recovery.

This design allows the WAL to append multiwrites in a single atomic step. To do so, the WAL writes the contents of the multiwrite beyond the end of the queue and then advances the end pointer in a single atomic disk write. Since the new updates

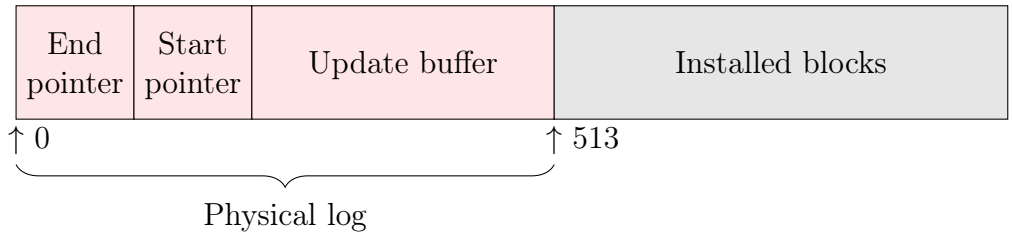

Figure 4-1: The disk layout of the full system, including the physical log.

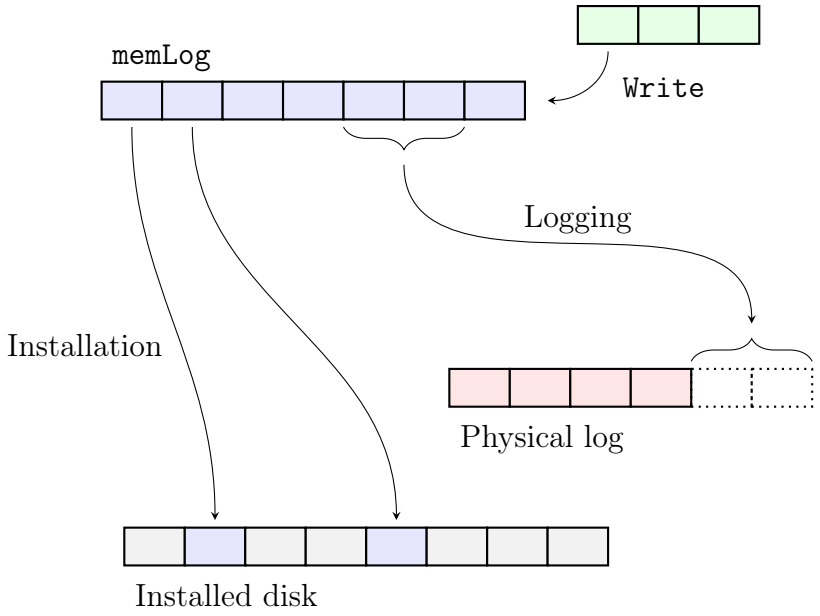

Figure 4-2: Overview of the WAL's implementation.

only become part of the log when the end pointer is updated, the multiwrite is either completely persisted or completely discarded on crash depending on whether the crash happens before or after the last write succeeds.

The WAL hides disk latency by performing logging and installation in background threads synchronized by a locked in-memory work queue called the `memLog` (Fig. 4-2). Threads interact with the WAL primarily through the `memLog`, using the interface methods `Write`, `Flush` and `Read`. A thread performs a multiwrite m by calling `Write`(m) to submit m to the `memLog` and then `Flush`() to wait for the background threads to make m durable. The background threads then install m to disk before removing it from the `memLog` altogether. A thread reads the contents of a disk block at address a by calling `Read`(a), which checks the `memLog` for the latest pending update to a and then reads directly from disk if no pending updates are found.

We model this interface as an atomic transition system (Sec. 3.3) involving the initial disk state $\mathcal{D}_{init}$, a list of multiwrites $\mathcal{G}$ and monotonically increasing lower bounds $\mathcal{B}_{dur}$ and $\mathcal{B}_{ms}$ (Fig. 4-3). In the absences of crashes, $\mathcal{G}$ is the monotonic list of all multiwrites submitted to the WAL. In the event of a crash, only some prefix of $\mathcal{G}$ containing all multiwrites before $\mathcal{B}_{dur}$ persists after recovery. In other words, $\mathcal{B}_{dur}$

Operation	**Behavior**
Initial state	$\mathcal{D}_{init} \leftarrow$ initial disk state; $\mathcal{G} \leftarrow [\,]$; $\mathcal{B}_{dur} \leftarrow 0$; $\mathcal{B}_{ms} \leftarrow 0$
`ReadMem`(a)	One of: • Return $(\mathbf{apply}(\mathcal{G}, \mathcal{D}_{init})^{[a]},\ \texttt{True})$ • $\mathcal{B}_{ms} \leftarrow m$ for some $m \in [\mathcal{B}_{ms}, \mathbf{len}\,\mathcal{G}]$ such that $\forall m' \in [m, \mathbf{len}\,\mathcal{G}]$, $\forall D,\ \mathbf{apply}(\mathcal{G}^{[0:m']}, D)^{[a]} = \mathbf{apply}(\mathcal{G}, D)^{[a]}$; Return $(b,\ \texttt{False})$ for some b
`ReadInst`(a)	Return $\mathbf{apply}(\mathcal{G}^{[0:m]}, \mathcal{D}_{init})^{[a]}$ for some $m \in [\mathcal{B}_{ms}, \mathbf{len}\,\mathcal{G}]$
`Write`(m)	Append m to $\mathcal{G}$
`Flush()`	$\mathcal{B}_{dur} \leftarrow \mathbf{len}\,\mathcal{G}$
Crash	$\mathcal{B}_{dur} \leftarrow m$ for some $m \in [\mathcal{B}_{dur},\ \mathbf{len}\,\mathcal{G}]$; $\mathcal{G} \leftarrow \mathcal{G}^{[0:m]}$

Figure 4-3: The WAL interface specification.

acts as a lower bound for multiwrite durability. `Flush` advances $\mathcal{B}_{dur}$ to the end of $\mathcal{G}$, guaranteeing that all prior multiwrites are durable when it returns.

Exposing separate `Write` and `Flush` calls allows threads to better exploit concurrency. This is true even for GoTxn, which flushes every multiwrite to ensure that committed transactions are durable. To see why, consider two threads concurrently updating different objects on the same disk block. Each thread calls `Read` to retrieve the full contents of the disk block, `Write` to write the block back with the new object contents, and then `Flush`. To prevent these updates from interfering, the threads perform `Read` and `Write` under a lock. However, they may perform `Flush` in parallel with other `Read` and `Write` calls, hiding its latency.

Specifying the behavior of `Read` is tricky. If the `memLog` contains updates to an address a, `Read`(a) returns the most recent contents of a during the atomic step when it accesses the `memLog` under lock. However, if the `memLog` does not contain any updates to a, `Read`(a) must retrieve the installed contents of a by performing a read from disk. When this happens, the block returned by `Read` is only guaranteed to match the most recent contents of a at some point between the read from the `memLog` and the read from disk (Fig. 4-4). An assertion of this form cannot be directly expressed as a HOCAP-style specification since it does not refer to a single atomic

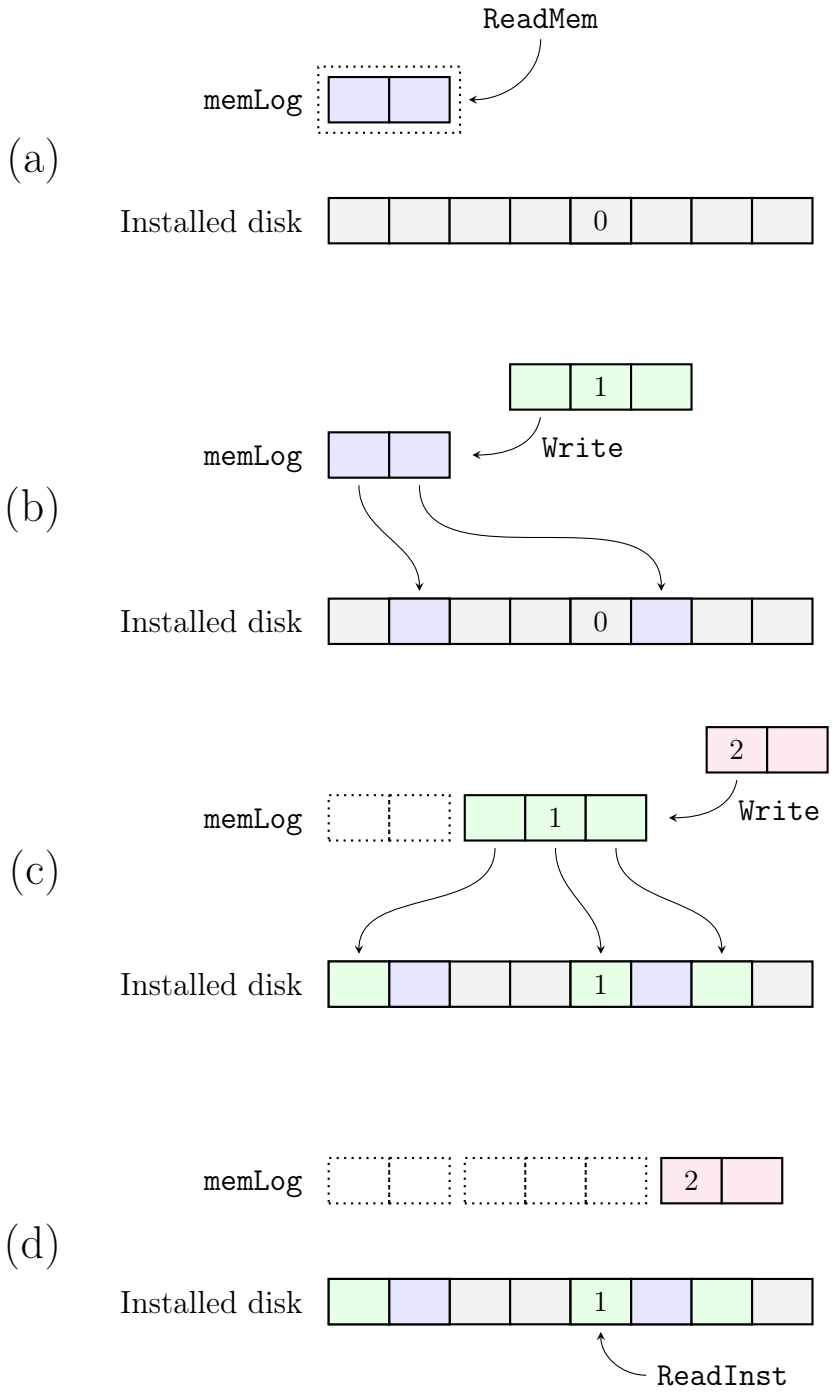

Figure 4-4: Example of `Read`(a) interacting with concurrent calls to `Write`. Updates to a and the installed disk contents at a are labeled with values representing block contents. (a) `ReadMem` does not find any updates to a in the `memLog`. (b-c) Two multiwrites are submitted, each containing an update to a. One of them is installed to disk. (d) `ReadInst` returns the block written to a by the first `Write`. This only reflects the result of applying all of $\mathcal{G}$ to $\mathcal{D}_{init}$ after the first `Write` but before the second.

step.

Instead, we split the specification of `Read` into `ReadMem` and `ReadInst` respectively corresponding to the read from `memLog` and the read from disk. A failed `ReadMem`(a) updates $\mathcal{B}_{ms}$ to the front of the `memLog` and asserts that none of the multiwrites beyond $\mathcal{B}_{ms}$ affect a. In other words,

$$\forall m \in [\mathcal{B}_{ms}, \mathbf{len}\,\mathcal{G}],\ \forall D,\ \mathbf{apply}(\mathcal{G}^{[0:m]}, D)^{[a]} = \mathbf{apply}(\mathcal{G}, D)^{[a]} \tag{4.1}$$

where $\mathbf{apply}(M, D)$ is the disk state after applying all of M to disk state D. Later, `ReadInst`(a) returns $\mathbf{apply}(\mathcal{G}'^{[0:m]}, \mathcal{D}_{init})^{[a]}$ for some $m \in [\mathcal{B}'_{ms}, \mathbf{len}\,\mathcal{G}']$ where $\mathcal{G}'$ and $\mathcal{B}'_{ms}$ are the values of $\mathcal{G}$ and $\mathcal{B}_{ms}$ during the atomic step taken by `ReadInst`. While concurrent calls to `ReadInst` and `Write` from other threads may cause $(\mathcal{B}'_{ms}, \mathcal{G}')$ to differ from $(\mathcal{B}_{ms}, \mathcal{G})$, the monotonicity of $\mathcal{B}_{ms}$ and $\mathcal{G}$ guarantee that $\mathcal{B}'_{ms} \geq \mathcal{B}_{ms}$ and that $\mathcal{G}$ is a prefix of $\mathcal{G}'$. If $m > \mathbf{len}\,\mathcal{G}$, then $\mathbf{apply}(\mathcal{G}'^{[0:m]}, \mathcal{D}_{init})^{[a]}$ reflects the most recent contents of a at some point between the atomic steps taken by `ReadMem` and `ReadInst`, just after $\mathcal{G}'^{[m]}$ was submitted. Otherwise, the caller may conclude from Eq. 4.1 that $\mathbf{apply}(\mathcal{G}'^{[0:m]}, \mathcal{D}_{init})^{[a]}$ reflects the most recent contents of a during the atomic step taken by `ReadMem`.

4.1 The In-Memory Log

To show that the WAL implements its specification, we start by describing the `memLog`. The `memLog` is central to the WAL's operation, acting as an intermediary between client threads making interface method calls, the *logger* thread writing updates to the log, and the *installer* thread installing updates to disk.

The `memLog` manages a list of disk updates `memLog.upds` $\mapsto \mathcal{L}_m$ under the lock `memLock`, divided into three regions $\mathcal{L}_{un}$, $\mathcal{L}_{lo}$ and $\mathcal{L}_{io}$. These are

- The **unstable** region $\mathcal{L}_{un}$, a buffer for recently submitted multiwrites;
- The **logger-owned** region $\mathcal{L}_{lo}$, a work queue for the logger;
- The **installer-owned** region $\mathcal{L}_{io}$, a work queue for the installer.

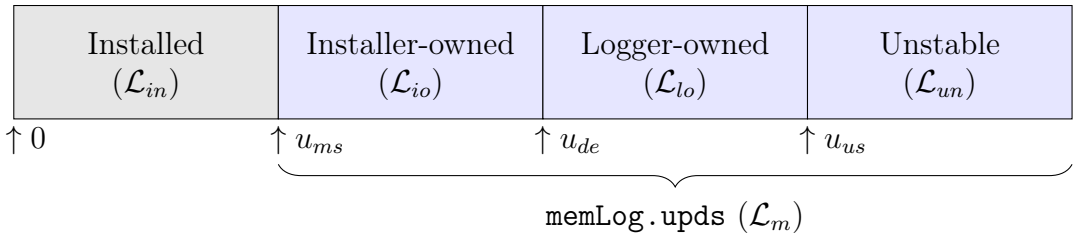

Figure 4-5: The logical log.

`Write` first accumulates updates in $\mathcal{L}_{un}$. `Flush` signals to the logger to move all updates in $\mathcal{L}_{un}$ into $\mathcal{L}_{lo}$. The logger then writes updates in $\mathcal{L}_{lo}$ to the physical log and moves them into $\mathcal{L}_{io}$, queueing them for installation. Finally, the installer installs updates in $\mathcal{L}_{io}$ and truncates them from the `memLog` entirely.

The proof defines an abstract **installed** region $\mathcal{L}_{in}$ containing updates that have been truncated from the `memLog`, and defines the *logical log* (Fig. 4-5)

$$\mathcal{L} = \mathcal{L}_{in} +\!\!+ \mathcal{L}_{io} +\!\!+ \mathcal{L}_{lo} +\!\!+ \mathcal{L}_{un} = \mathcal{L}_{in} +\!\!+ \mathcal{L}_m$$

where $+\!\!+$ is the list concatenation operator. Then, apart from `Write`, all updates to the `memLog` correspond to the rightwards motion of a boundary between regions of $\mathcal{L}$. The `memLog` takes advantage of this by tracking region boundaries with monotonically increasing indices into $\mathcal{L}$. These are

- `memLog.memStart` $\mapsto u_{ms}$, the start of the `memLog`;
- `memLog.diskEnd` $\mapsto u_{de}$, the boundary between $\mathcal{L}_{io}$ and $\mathcal{L}_{lo}$;
- `memLog.unstStart` $\mapsto u_{us}$, the boundary between $\mathcal{L}_{lo}$ and $\mathcal{L}_{un}$.

The WAL may then access the regions of $\mathcal{L}_m$ with the expressions

$$\mathcal{L}_{io} = (\mathcal{L}_m)^{[0:u_{de}-u_{ms}]}$$
$$\mathcal{L}_{lo} = (\mathcal{L}_m)^{[u_{de}-u_{ms}:u_{us}-u_{ms}]}$$
$$\mathcal{L}_{un} = (\mathcal{L}_m)^{[u_{us}-u_{ms}:\mathbf{len}\,\mathcal{L}_m]}$$

Under this scheme, the WAL truncates the `memLog` by truncating `upds` and advancing `memStart`, leaving other indices untouched. This simplifies concurrent reasoning

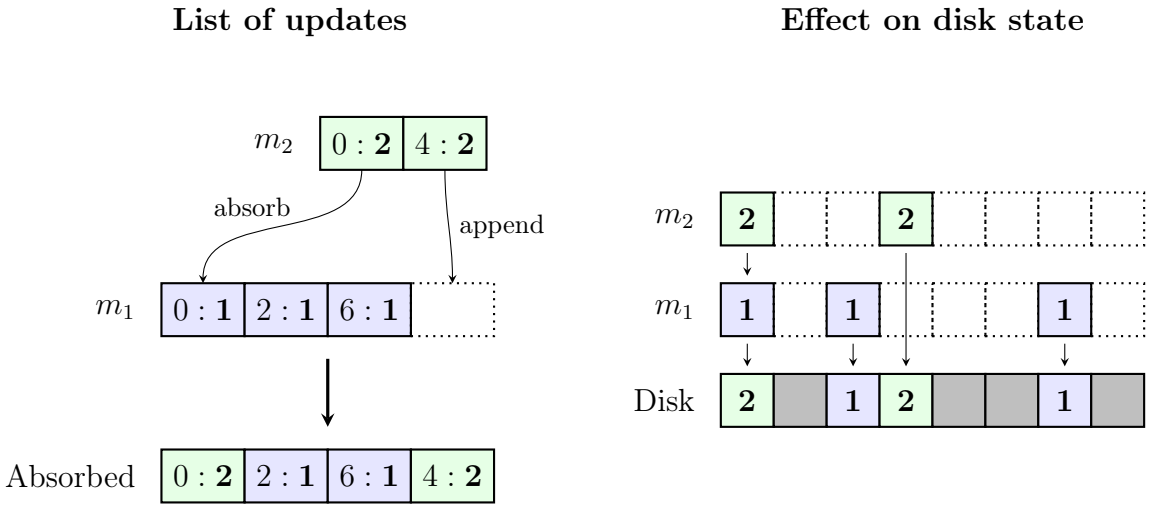

Figure 4-6: An example of multiwrite absorption. **absorb**(m_2, m_1) has the same effect on disk state as $m_1 + m_2$, even though neither m_1 nor m_2 are slices of **absorb**(m_2, m_1).

	Property	Rule
a)	Trivial	$[\,]$ **has** $[\,]$
b)	Absorption	M **has** U $\implies (M +\!\!+ [m])$ **has absorb**(m, U)
c)	Concatenation	$(M_1$ **has** $U_1) \wedge (M_2$ **has** $U_2)$ $\implies (M_1 +\!\!+ M_2)$ **has** $(U_1 +\!\!+ U_2)$

Figure 4-7: Properties of the **has** relation.

because a thread's local indices into `upds` now remain valid evan after it releases `memLock`.

While $\mathcal{L}$ resembles a flattened version of $\mathcal{G}$, there is one important difference. Whenever a multiwrite m is submitted to the WAL, it is *absorbed* into other updates in $\mathcal{L}_{un}$ in order to reduce disk traffic. That is, if an update $\bar{u}'$ in m writes to the same address as another update $\bar{u}$ in $\mathcal{L}_{un}$, $\bar{u}$ is overwritten with $\bar{u}'$ (Fig. 4-6). This takes advantage of the fact that applying just $\bar{u}'$ has the same effect as applying both $\bar{u}$ and $\bar{u}'$ in succession.

Absorption makes reasoning about the `memLog` challenging because we can no longer treat multiwrites as slices of $\mathcal{L}$ (Fig. 4-6). Instead, we introduce the relation

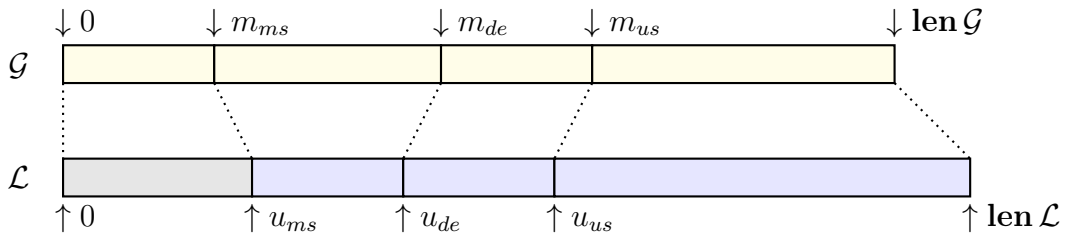

Figure 4-8: Relationship between $\mathcal{L}$ and $\mathcal{G}$.

$$M \textbf{ has } U := \forall D,\ \textbf{apply}(M, D) = \textbf{apply}(U, D)$$

which asserts that performing the multiwrites M has the same effect as performing the updates U. This is useful because $U + m$ has the same effect as $\textbf{absorb}(m, U)$, so if M **has** U, then $M + [m]$ **has** $\textbf{absorb}(m, U)$ (Fig. 4-7b).

Another important property about the **has** relation is that it persists under concatenation (Fig. 4-7c). However, **has** relations cannot be split up: if $M_1 + M_2$ **has** U, there does not necessarily exist U_1 and U_2 such that $U = U_1 + U_2$, M_1 **has** U_1 and M_2 **has** U_2.

We can now use the **has** relation to relate regions of $\mathcal{L}$ to slices of $\mathcal{G}$ (Fig. 4-8). In particular, we track indices m_{ms}, m_{de} and m_{us} into $\mathcal{G}$ corresponding to u_{ms}, u_{de} and u_{us}, and impose the invariant

$$\mathcal{H}_{\mathcal{G},\mathcal{L}}\Big(\big[(0,0),(m_{ms},u_{ms}),(m_{de},u_{de}),(m_{us},u_{us}),(\textbf{len}\,\mathcal{G},\textbf{len}\,\mathcal{L})\big]\Big) \tag{4.2}$$

where

$$\mathcal{H}_{\mathcal{G},\mathcal{L}}\Big(\big[(m_1,u_1),(m_2,u_2),\ldots,(m_n,u_n)\big]\Big) :=$$
$$\forall i,\ m_i \le m_{i+1} \wedge u_i \le u_{i+1} \wedge \mathcal{G}^{[m_i:m_{i+1}]} \textbf{ has } \mathcal{L}^{[u_i:u_{i+1}]}$$

Since this invariant pertains to the `memLog`, which can only be safely accessed under `memLock`, we maintain it as part of the `memLock` lock invariant $\boxed{I_L}^{\texttt{memLock}}$.

Under this invariant, it is not generally possible to move individual multiwrites between regions since there is no guarantee that any given multiwrite can be teased apart from other multiwrites in the same region. It is, however, safe to move an

entire region into an adjacent region in a single atomic step. For example, the WAL may atomically complete the installation of all of $\mathcal{L}_{io}$ by increasing u_{ms} to u_{de} and m_{ms} to m_{de}. This maintains Eq. 4.2 since $\mathcal{G}^{[m_{de}:m_{de}]}$ **has** $\mathcal{L}^{[u_{de}:u_{de}]}$ by Fig. 4-7a and $\mathcal{G}^{[0:m_{de}]}$ **has** $\mathcal{L}^{[0:u_{de}]}$ by Fig. 4-7c.

In general, pairs of indices $b_i = (m_i, u_i)$ protected by an invariant like Eq. 4.2 are *multiwrite boundaries* in the sense that no multiwrite has updates on both sides of any u_i. The proof may only move multiwrites from one region to another by moving both indices of a multiwrite boundary in unison. What we have just shown is that it is always safe to move a multiwrite boundary to an adjacent multiwrite boundary.

4.2 Snapshots

The logger and installer work in a similar fashion, processing updates batch by batch per iteration of an infinite loop. Each cycle, the logger takes a snapshot of $\mathcal{L}_{lo}$, including `unstStart`. After logging the updates in its snapshot, it shifts `diskEnd` to its snapshot's `unstStart`. Similarly, each installer iteration installs a snapshot of $\mathcal{L}_{io}$. After installing the updates in it snapshot, the installer truncates both the `memLog` and the physical log and then updates `memStart` to its snapshot's `diskEnd`.

To hide disk latency effectively, the logger and installer do not hold any locks, including `memLock`, while performing disk operations. In particular, the installer does not hold any locks when installing blocks or truncating the physical log (Fig. 4-9). This means that the logger may concurrently update `diskEnd` in the middle of an active installer cycle, causing `diskEnd` to diverge from the installer snapshot's `diskEnd`. For the sake of flexibility, the proof also does not assert that `unstStart` always matches the logger snapshot's `unstStart`, even though nothing in the WAL's current implementation would cause them to diverge.

We track the installer snapshot's `diskEnd` and the logger snapshot's `unstStart` by defining new multiwrite boundaries b_{des} and b_{uss}. When their respective snapshots are not in use, the proof keeps b_{des} and b_{uss} at b_{ms} and b_{de} respectively. $b_{ms} : b_{des}$ and $b_{de} : b_{uss}$ can be thought of as installer- and logger-controlled windows providing access

```
func (l *Walog) installer() {
  l.memLock.Lock()
  for !l.shutdown {
    // Take a snapshot, including the current diskEnd.
    diskEndSnap := l.memLog.diskEnd
    updsSnap := l.memLog.upds.takeTill(diskEndSnap)
    if len(updsSnap) == 0 {
      // Unlock memLock and wait for the logger to log
      // new updates before re-acquiring the lock.
      l.condInstall.Wait()
      continue
    }
    l.memLock.Unlock()

    // Install updsSnap to disk.
    installBlocks(l.d, updsSnap)
    // Advance the physical log start pointer to the snapshot's diskEnd.
    AdvanceLogStart(l.d, diskEndSnap)

    l.memLock.Lock()
    l.memLog.truncate(diskEndSnap)
  }
  l.memLock.Unlock()
}
```

Figure 4-9: Stripped-down installer code.

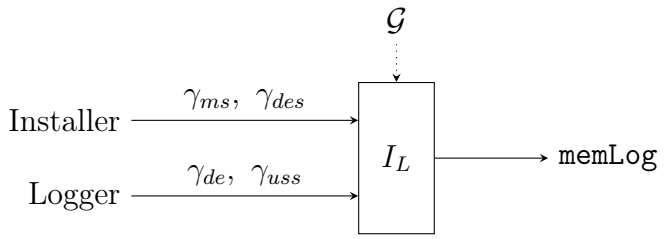

Figure 4-10: Control diagram for $\boxed{I_L}^{\texttt{memLock}}$. Control over the `memLog` is split between the logger, installer and client. The client indirectly controls the updates in the `memLog` through a separate mechanism involving $\mathcal{G}$ (Fig. 4-12).

to non-overlapping ranges of $\mathcal{G}$. We express this in proof by giving the installer and logger control over (b_{ms}, b_{des}) and (b_{de}, b_{uss}) respectively through the ghost variables $(\gamma_{ms}, \gamma_{des})$ and $(\gamma_{de}, \gamma_{uss})$ (Fig. 4-10).

We maintain the assertion that b_{des} and b_{uss} are multiwrite boundaries by extending Eq. 4.2 to define the *logical log boundaries component* $I_{L,b}$ of $\boxed{I_L}^{\texttt{memLock}}$:

$$\begin{aligned} &I_{L,b}(\mathcal{G}, \mathcal{L}, b_{ms}, b_{des}, b_{de}, b_{uss}, b_{us}) \\ &\quad := \mathcal{H}_{\mathcal{G},\mathcal{L}}\Big(\big[(0,0), b_{ms}, b_{des}, b_{de}, b_{uss}, b_{us}, (\mathbf{len}\,\mathcal{G}, \mathbf{len}\,\mathcal{L})\big]\Big) \end{aligned} \tag{4.3}$$

With the help of b_{des} and b_{uss}, we can show that $I_{L,b}$ is maintained throughout the operation of the WAL by stepping through each thread's execution. We summarize the key steps here:

- `Write` absorbs its multiwrite into $\mathcal{L}_{un}$ (Fig. 4-11a). By the absorption property of the **has** relation (Fig. 4-7c), this preserves $\mathcal{G}^{[m_{us}:\mathbf{len}\,\mathcal{G}]}$ **has** $\mathcal{L}^{[u_{us}:\mathbf{len}\,\mathcal{L}]}$.

- After a `Flush`, the logger advances `unstStart` to $\mathbf{len}\,\mathcal{L}$. This preserves $I_{L,b}$ because it corresponds to moving the multiwrite boundary b_{us} to the adjacent multiwrite boundary $(\mathbf{len}\,\mathcal{G}, \mathbf{len}\,\mathcal{L})$ (Fig. 4-11b).

- The logger takes a snapshot of $\mathcal{L}_{lo}$, moving b_{uss} to b_{us} (Fig. 4-11c). After the logger releases `memLock`, b_{us} is allowed to diverge from b_{uss} (Fig. 4-11d). At the end of its cycle, the logger advances `diskEnd` to its snapshot's `unstStart`, moving b_{de} to b_{uss} (Fig. 4-11e).

- The installer takes a snapshot of $\mathcal{L}_{io}$, moving b_{des} to b_{de} (Fig. 4-11f). After the installer releases `memLock`, completed logger cycles may cause b_{de} to diverge from b_{des} (Fig. 4-11g). At the end of its cycle, the installer truncates its snapshot from the `memLog`, moving b_{ms} to b_{des} (Fig. 4-11h).

4.3 Durable State

After a crash, the WAL's recovery procedure reconstructs the `memLog` by copying the physical log's contents into `upds`, its start pointer into `memStart` and its end pointer

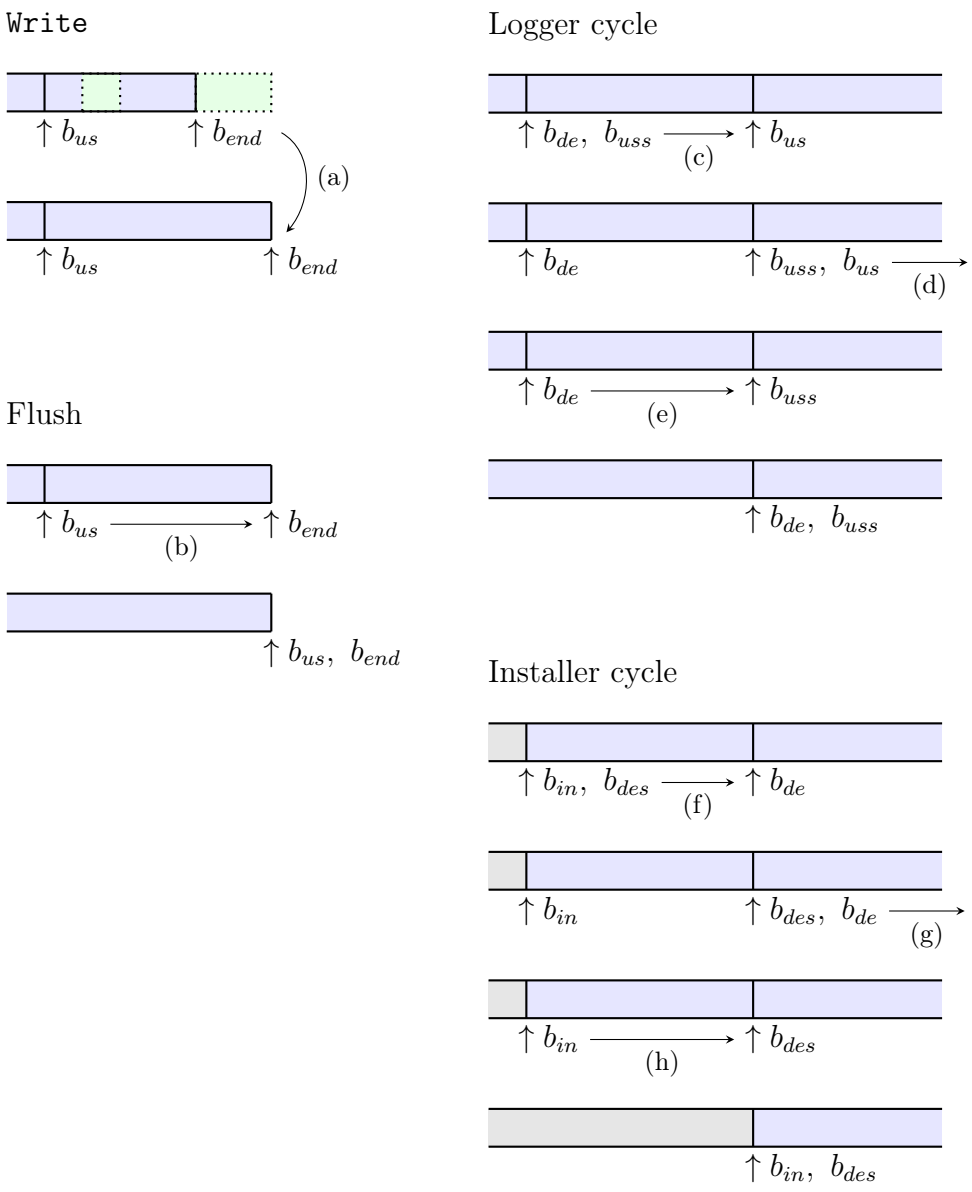

Figure 4-11: Proof that $I_{L,b}$ is maintained throughout the execution of the WAL. $b_{end} = (\mathbf{len}\,\mathcal{G}, \mathbf{len}\,\mathcal{L})$ is the multiwrite boundary connecting the back ends of $\mathcal{G}$ and $\mathcal{L}$.

into `diskEnd` and `unstStart`. It then restarts the logger and installer threads, leaving the installer to complete the installation of interrupted multiwrites as part of its normal operation. In particular, the WAL does not wait until all interrupted multiwrites from before the crash are completely installed before accepting new multiwrites. This simplifies reasoning about crashes during the recovery process itself since recovery does not involve mutating disk state.

The pre-conditions for recovery are the WAL's crash conditions relating to the durable disk state. Since $\boxed{I_L}^{\texttt{memLock}}$ is only guaranteed to hold when no thread holds `memLock`, the proof cannot safely persist assertions in $\boxed{I_L}^{\texttt{memLock}}$ through to recovery. Instead, it captures the WAL's crash obligations in a separate interface invariant $\boxed{I_W}$. Following the discussion in Sec. 3.3, the proof defines

$$I_W := \exists \sigma,\ I_{W,inner}(\sigma) * P_{conn}(\sigma)$$

where $P_{conn}(\sigma)$ is the caller-defined connecting predicate for the caller-visible state $\sigma = (\mathcal{D}_{init}, \mathcal{G}, \mathcal{B}_{dur}, \mathcal{B}_{ms})$. The proof fixes $\mathcal{D}_{init}$ when $\boxed{I_W}$ is sealed at program start, uses a monotonic list $\gamma_{\mathcal{G}} \mapsto^{+} \mathcal{G}$ to synchronize $\mathcal{G}$ between $\boxed{I_W}$ and $\boxed{I_L}^{\texttt{memLock}}$, and maintains in $\boxed{I_W}$ that $\mathcal{B}_{dur} \leq m^*_{pe}$ and $\mathcal{B}_{ms} \leq m^*_{ps}$ where m^*_{ps} and m^*_{pe} are the multiwrite indices corresponding to the start and end of the physical log.

The proof models the physical log as an atomic transition system with state σ_p containing its start index u^*_{ps} and a list of updates $\mathcal{P}_d$. Appending updates to the physical log appends to $\mathcal{P}_d$, while truncating the physical log truncates $\mathcal{P}_d$ and advances u^*_{ps}. The proof relates σ_p to $\boxed{I_W}$ by splitting ownership over a ghost variable $\gamma_p \mapsto \sigma_p$ between $\boxed{I_W}$ and the physical log connecting predicate.

Similar to $\mathcal{L}$, the proof treats $\mathcal{P}_d$ as a suffix starting from u^*_{ps} of a larger $\mathcal{P}$ that includes truncated updates. While the WAL's current implementation guarantees that $\mathcal{P} = \mathcal{L}^{[0:u^*_{pe}]}$, we avoid directly relating $\mathcal{P}$ to $\mathcal{L}$ in $\boxed{I_W}$ or $\boxed{I_L}^{\texttt{memLock}}$ to give us the freedom to adapt the proof to work with different logging schemes. Instead, we relate $\mathcal{P}$ to $\mathcal{G}$ by imposing the *physical log boundaries component* $I_{W,b}$ of I_W:

$$I_{W,b}(\mathcal{G}, \mathcal{P}, b^*_{ps}, b^*_{des}, b^*_{de}, b^*_{pe}) := \mathcal{H}_{\mathcal{G},\mathcal{P}}\Big(\big[b^*_{ps}, b^*_{des}, b^*_{de}, b^*_{pe}\big]\Big) \tag{4.4}$$

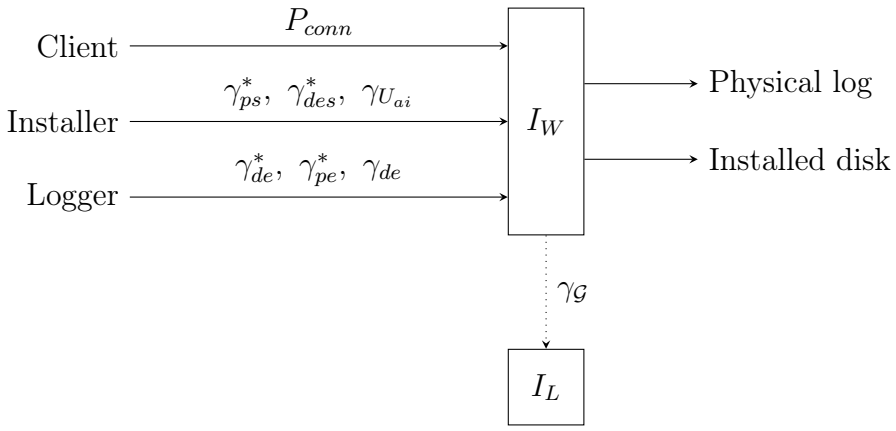

Figure 4-12: Control diagram for $\boxed{I_W}$. Control over disk resources is split between the logger, installer and client through the use of ghost variables and the connecting predicate P_{conn}. $\mathcal{G}$ is in turn synchronized with $\boxed{I_L}^{\texttt{memLock}}$ through the monotonic list $\gamma_{\mathcal{G}}$ (Fig. 4-10).

where b^*_{ps} and b^*_{pe} are the boundaries corresponding to u^*_{ps} and u^*_{pe}, and b^*_{des} and b^*_{de} are the boundaries corresponding to the installer snapshot's and `memLog`'s `diskEnd`.

The installer and logger maintain control over (b^*_{ps}, b^*_{des}) and (b^*_{de}, b^*_{pe}) respectively through the ghost variables $(\gamma^*_{ps}, \gamma^*_{des})$ and $(\gamma^*_{de}, \gamma^*_{pe})$ (Fig. 4-12). Most of the time, $(b^*_{ps}, b^*_{des}, b^*_{de}, b^*_{pe}) = (b_{ms}, b_{des}, b_{de}, b_{de})$. However, after the logger advances the physical log end pointer but before it advances `diskEnd`, $b^*_{pe} = b_{uss}$ instead of b_{de}. Similarly, after the installer truncates the physical log but before it truncates the `memlog`, $b^*_{ps} = b_{des}$ instead of b_{ms}. Like in Sec. 4.2, we show that $I_{W,b}$ holds throughout the operation of the WAL by showing that updates to these boundaries correspond to either concatenating to $\mathcal{P}^{[u^*_{de}:u^*_{pe}]}$ or moving a multiwrite boundary to an adjacent multiwrite boundary.

One complication to this scheme pertains to the point when the installer takes its snapshot. Since the installer snapshot's `diskEnd` comes from the `memLog`, taking the installer snapshot only corresponds to moving b^*_{des} to b^*_{de} if $b^*_{de} = b_{de}$. While we know that $b^*_{de} = b_{de}$ since the logger updates them in unison, we can only communicate this fact to the installer by putting it in an invariant. However, we cannot do so directly since b^*_{de} and b_{de} belong to different invariants. Instead, we split ownership over γ_{de} between not just the logger and $\boxed{I_L}^{\texttt{memLock}}$ but also $\boxed{I_W}$, allowing the proof to assert

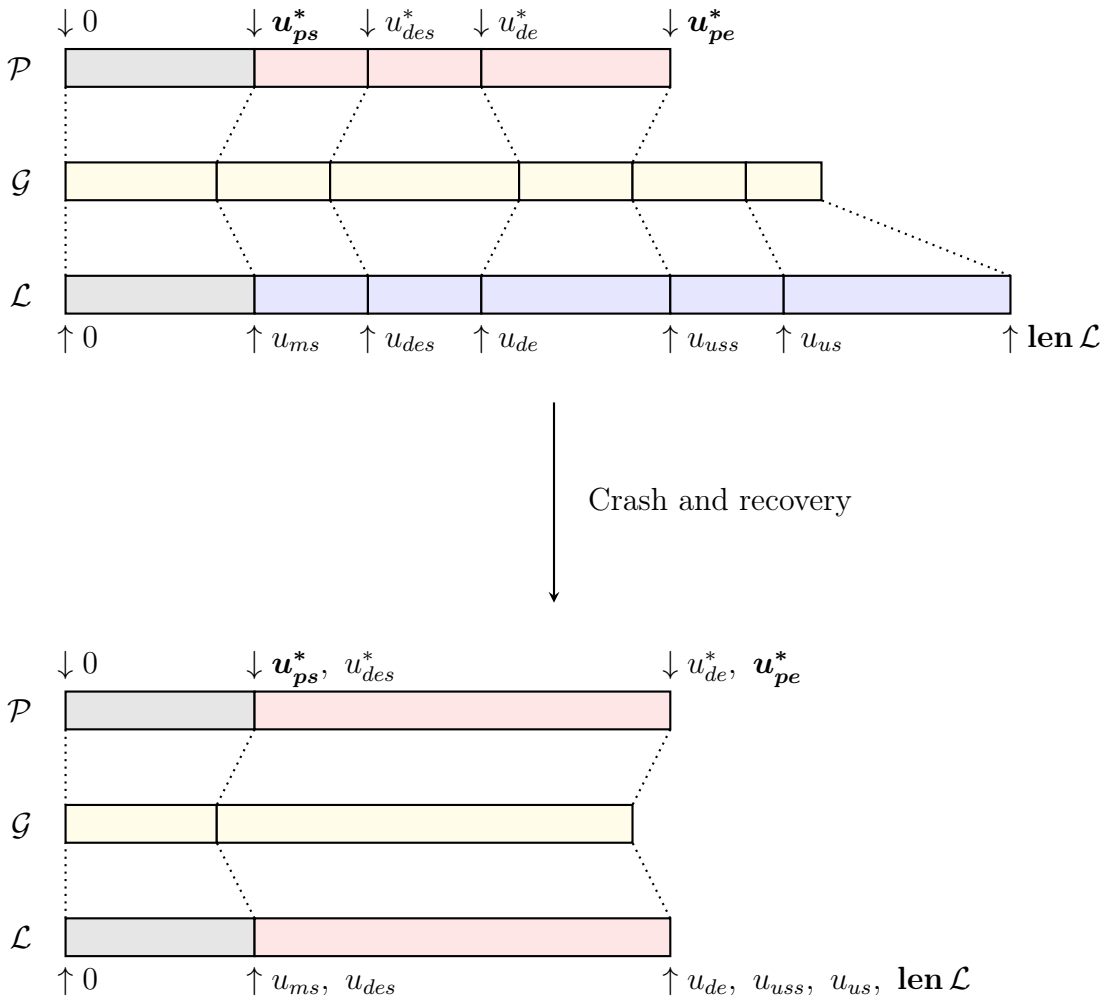

Figure 4-13: $\mathcal{P}$, $\mathcal{L}$ and $\mathcal{G}$ after a crash and recovery.

that $b^*_{de} = b_{de}$ in $\boxed{I_W}$.

$I_{W,b}$ allows the proof of the recovery procedure to show that a crash and recovery is consistent with the WAL's crash specification (Fig. 4-3). When reconstructing $\boxed{I_W}$ and $\boxed{I_L}^{\texttt{memLock}}$, the proof sets $\mathcal{L} \leftarrow \mathcal{P}$ and $(b_{ms}, b_{de}, b_{us}) \leftarrow (b^*_{ps}, b^*_{pe}, b^*_{pe})$ to match the post-recovery contents of the `memLog` (Fig. 4-13). In doing so, $I_{L,b}$ requires the proof to update $\mathcal{G}$ to $\mathcal{G}^{[0:m^*_{pe}]}$. This is consistent with the crash specification since $\mathcal{G}^{[0:m^*_{pe}]}$ is a durable prefix of $\mathcal{G}$.

Finally, we describe the installed disk state. The installed disk state is the result of applying some prefix $\mathcal{L}^{[0:u]}$ to $\mathcal{D}_{init}$, where $u_{ms} \leq u \leq u_{des}$ depending on the progress of the installer in installing its snapshot. Since $\boxed{I_W}$ does not have access to $\mathcal{L}$, we

instead express this by maintaining

$$\forall a,\ \exists b,\ d_a \mapsto b * b = \mathbf{apply}(\mathcal{P}^{[0:u^*_{ps}]} +\!\!+ U_{ai}, \mathcal{D}_{init})^{[a]} \tag{4.5}$$

where $d_a \mapsto b$ is the ownership predicate asserting that the disk block at address a has contents b, and U_{ai} is a list of updates controlled by the installer through the ghost variable $\gamma_{U_{ai}}$ representing the updates that the installer has already installed from its snapshot.

The specification of `ReadInst` (Fig. 4-3) additionally requires us to maintain that

$$\exists m \in [\mathcal{B}_{ms}, \mathbf{len}\,\mathcal{G}],\ b = \mathbf{apply}(\mathcal{G}^{[0:m]}, \mathcal{D}_{init})^{[a]}$$

for all a, since `ReadInst`(a) may be called for any a at any time. We do so by augmenting Eq. 4.5 to also enforce

$$\exists m \in [m^*_{ps}, m^*_{des}],\ \mathbf{apply}(\mathcal{P}^{[0:u^*_{ps}]} +\!\!+ U_{ai}, \mathcal{D}_{init})^{[a]} = \mathbf{apply}(\mathcal{G}^{[0:m]}, \mathcal{D}_{init})^{[a]} \tag{4.6}$$

The proof of the installer must now show that this remains satisfied whenever the installer installs an update. This is only interesting when a is affected by the installer snapshot $\mathcal{L}_{isnap} := \mathcal{L}^{[u_{ms}:u_{des}]}$, because otherwise U_{ai} has no effect on a and so

$$\mathbf{apply}(\mathcal{P}^{[0:u^*_{ps}]} +\!\!+ U_{ai}, \mathcal{D}_{init})^{[a]} = \mathbf{apply}(\mathcal{P}^{[0:u^*_{ps}]}, \mathcal{D}_{init})^{[a]} = \mathbf{apply}(\mathcal{G}^{[0:m^*_{ps}]}, \mathcal{D}_{init})^{[a]}$$

If $\mathcal{L}_{isnap}$ affects a, then the proof must show that Eq. 4.6 holds for all prefixes U_{ai} of $\mathcal{L}_{isnap}$. To do so, it first shows that $\mathcal{G}_{isnap}$ **match** $\mathcal{L}_{isnap}$ where

$$M \textbf{ match } U := \forall(\bar{u}_a :\ \bar{u}_b) \in U,\ \exists \bar{m} \in M,\ \forall D,\ \mathbf{apply}(\bar{m}, D)^{[\bar{u}_a]} = \bar{u}_b$$

and $\mathcal{G}_{isnap} := \mathcal{G}^{[m_{ms}:m_{des}]}$. In other words, for any a, each $\bar{u} \in \mathcal{L}_{isnap}$ affecting a can be matched to an $\bar{m} \in \mathcal{G}_{isnap}$ such that $\bar{u}$ has the same effect on a as $\bar{m}$. Then, by choosing $\bar{u}$ to be the last update of U_{ai} affecting a, the proof may recover an $\bar{m} \in \mathcal{G}_{isnap}$ with the same effect as $\bar{u}$ on a, so that

$$\mathbf{apply}(\mathcal{P}^{[0:u^*_{ps}]} +\!\!+ U_{ai}, \mathcal{D}_{init})^{[a]} = \mathbf{apply}(\mathcal{G}^{[0:m+1]}, \mathcal{D}_{init})^{[a]}$$

where m is the index of $\bar{m}$ in $\mathcal{G}$ (Fig. 4-14).

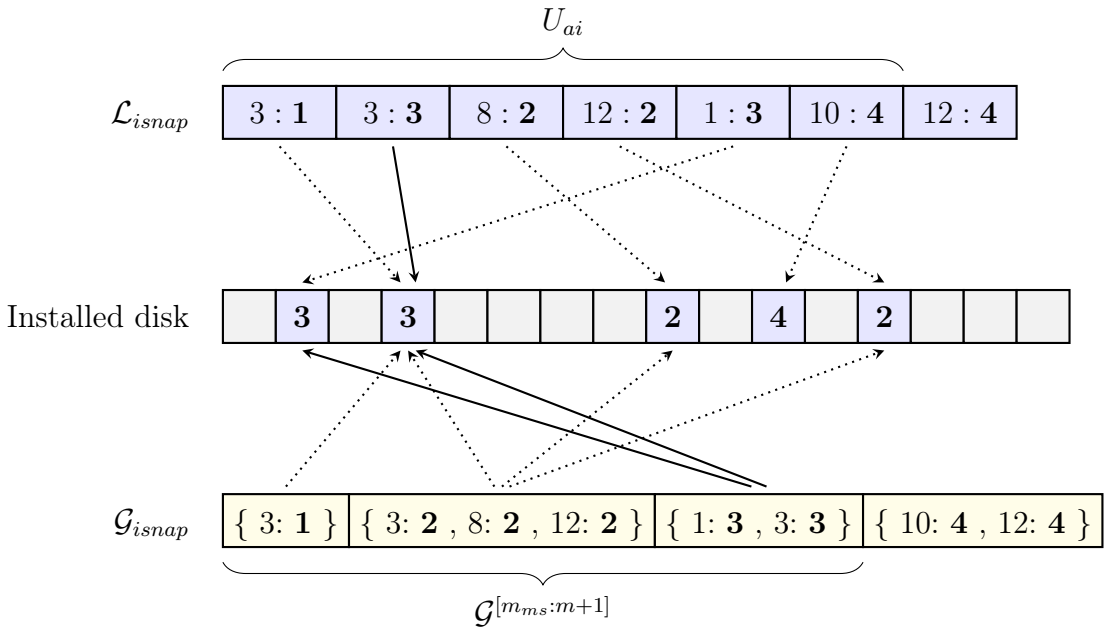

Figure 4-14: Motivation for the **match** relation. If U_{ai} affects a, then U_{ai} has the same effect on a as a prefix $\mathcal{G}^{[m_{ms}:m+1]}$ of $\mathcal{G}_{isnap}$ if $\bar{u}$, the last update of U_{ai} affecting a, has the same effect on a as $\mathcal{G}^{[m]}$. This is illustrated here for $a = 3$, with the solid arrows representing the effects of $\bar{u}$ and $\mathcal{G}^{[m]}$.

Unfortunately, the invariants so far only guarantee that $\mathcal{G}_{isnap}$ **has** $\mathcal{L}_{isnap}$, not $\mathcal{G}_{isnap}$ **match** $\mathcal{L}_{isnap}$. In particular, M **has** U does not imply M **match** U since U may contain multiple updates to the same address. For example, $M = [\{(0:\ 2)\}]$ and $U = [(0:\ 1), (0:\ 2)]$ satisfy M **has** U but not M **match** U since the effect of $(0:\ 1)$ is not reflected by the multiwrite $\{(0:\ 2)\}$. This distinction is important: if $\mathcal{G}_{isnap} = [\{(0:\ 2)\}]$ and $\mathcal{L}_{isnap} = [(0:\ 1), (0:\ 2)]$, then `ReadInst`$(0)$ may return 1, which does not reflect the complete application of any durable prefix of $\mathcal{G}$.

The proof shows that $\mathcal{G}_{isnap}$ **match** $\mathcal{L}_{isnap}$ by asserting in $\boxed{I_L}^{\texttt{memLock}}$ that the **match** relation holds for every region of $\mathcal{L}$:

$$\mathcal{G}^{[0:m_{ms}]}\ \textbf{match}\ \mathcal{L}^{[0:u_{ms}]} \wedge \cdots \wedge \mathcal{G}^{[m_{us}:\textbf{len}\,\mathcal{G}]}\ \textbf{match}\ \mathcal{L}^{[u_{us}:\textbf{len}\,\mathcal{L}]} \tag{4.7}$$

This is essentially the same as $I_{L,b}$ (Eq. 4.3), but with **match** instead of **has**. Since **match** satisfies the same algebraic properties as **has** with respect to absorption and concatenation (Fig. 4-7), the proof for $I_{L,b}$ can be directly reused to show that Eq. 4.7 remains satisfied throughout the execution of the WAL.

Chapter 5

The Locking Layer

The locking layer provides an interface for *transactions*, which are atomic operations on durable objects. A transaction is treated as an atomic whole not just across crashes but also with respect to other concurrent transactions. The role of the object, journal and locking layers is to close the gap between transactions and the disk reads and multiwrites offered by the WAL.

One key step is to allow object writes to be distributed across the body of a transaction. The journal layer enables this by assigning each transaction a *local transaction buffer*. A transaction accumulates object writes in its local transaction buffer and *commits* on completion by submitting the accumulated writes as a single atomic group.

The journal layer alone is not sufficient to guarantee atomicity for transactions that include reads. For example, consider two concurrent instances of a transaction that creates a file in a directory d by creating a file, reading d's inode, appending an entry for the file to the inode and then writing the inode back to disk. If both transactions perform the read before either one commits, the resulting state of d's inode will only contain an entry from one of the two transactions.

Locks can be used to prevent such conflicts. However, verifying the correctness of a locked implementation requires concurrent reasoning, which can be inhibitively tedious for complex transactions. The locking layer mitigates this with an automatic lock management scheme that guarantees transaction atomicity. Its specifications

reduce the behavior of transactions into atomic operations on the durable object state, allowing for verification free from concurrent reasoning.

5.1 Locking Layer Interface

The locking layer exposes wrappers around the journal layer interface calls `Begin`, `Read`, `Write` and `Commit`. A client calls `Begin` to create a new local transaction buffer, `Read` to perform object reads, `Write` to add an object write to its local transaction buffer and `Commit` to submit the accumulated object writes. `Commit` also flushes the WAL to ensure that the object writes are durable when it returns.

The locking layer associates every object with its own lock, allowing transactions that operate on non-overlapping sets of objects to run in parallel. Whenever a `Read` or `Write` first accesses an object, the locking layer acquires its associated per-object lock. On `Commit`, the locking layer releases all the locks acquired for the transaction throughout its execution. This design allows for a simple specification that does not require developers to explicitly specify lock sets for each transaction.

In fact, the specification for the locking layer completely hides concurrent reasoning from the caller by reducing a transaction's execution to a single atomic update applied on `Commit` to the caller-visible committed object state $\mathcal{C}$. We represent the transition relation for this atomic update as a *transaction map* $\mathcal{M}$ mapping all objects accessed by the transaction to initial and modified values. Concretely, the transaction map $\mathcal{M}$ represents the transition relation

$$R(\mathcal{C}, \mathcal{C}') := \forall a,\ \mathcal{C}[a] = \mathcal{C}'[a] \vee (a : \mathcal{C}[a], \mathcal{C}'[a]) \in \mathcal{M} \tag{5.1}$$

where $\mathcal{C}[a]$ and $\mathcal{C}'[a]$ are the contents of object a in the committed object state before and after `Commit`, and $(a : v, v')$ denotes an entry of $\mathcal{M}$ mapping object a to initial value v and modified value v'.

The proof builds $\mathcal{M}$ through the specifications for `Read` and `Write` (Fig. 5-1), starting with an empty map on `Begin`. For example, a transaction that copies the contents of an object a_i into another object a_j has the transaction map $\mathcal{M} :=$

Operation	Constraint	Behavior
`Read`(a)	$a \notin \mathcal{M}$	Define a new variable v; $\mathcal{M} \leftarrow \mathcal{M} \cup (a : v, v)$; Return v
`Read`(a)	$(a : v, v') \in \mathcal{M}$	Return v'
`Write`(a, x)	$a \notin \mathcal{M}$	Define a new variable v; $\mathcal{M} \leftarrow \mathcal{M} \cup (a : v, x)$
`Write`(a, x)	$(a : v, v') \in \mathcal{M}$	$\mathcal{M} \leftarrow \mathcal{M} \cup (a : v, x)$

Figure 5-1: Specifications for `Read` and `Write`. $M \cup (k : v)$ denotes the operation that inserts the mapping $k \to v$ into map M, possibly overwriting an older mapping for k.

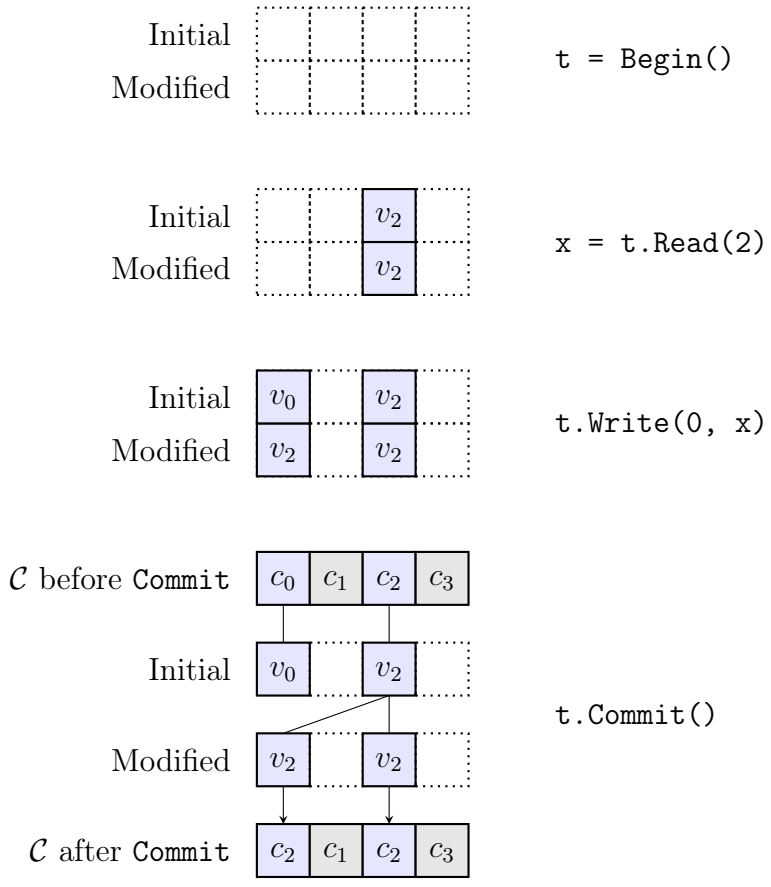

Figure 5-2: Constructing and using $\mathcal{M}$ in a transaction that copies the contents of object 2 into object 0. The value of $\mathcal{M}$ after each interface call is represented by two rows showing the initial and modified values of its entries.

$\{(a_i : v_i, v_i),\ (a_j : v_j, v_i)\}$ (Fig. 5-2). Importantly, the specifications for `Read` and `Write` make no reference to $\mathcal{C}$. Instead, they represent initial object values as algebraic variables to be reconciled with $\mathcal{C}$ on `Commit`. From the perspective of the proof, this delays a transaction's logical object reads until `Commit`, allowing the transaction to be treated in its entirety as a single atomic update applied on `Commit`.

In the same vein as Sec. 3.3, the proof exposes $\mathcal{C}$ through caller-defined per-object connecting predicates $a \mapsto_{\mathcal{C}} v$. The caller could, for example, define $a \mapsto_{\mathcal{C}} v := \gamma_a \mapsto_{1/2} v$ for some ghost variables γ_a, and then share γ_a with an invariant of their own in order to prove assertions about how their transactions affect $\mathcal{C}$. The Hoare triple for `Commit` now takes the form

$$\begin{array}{l} \forall \mathcal{M} \forall Q_{succ} \forall Q_{abort}, \\ \left\{ \mathcal{T}_L(\mathcal{M}) * \left(P_{upd}(\mathcal{M}, Q_{succ}) \wedge Q_{abort} \right) \right\} \\ \quad \texttt{Commit}() \\ \left\{ \mathbf{RET}\, r;\ \mathbf{if}\ r = \texttt{Success}\ \mathbf{then}\ Q_{succ}\ \mathbf{else}\ Q_{abort} \right\} \end{array} \tag{5.2}$$

where $\mathcal{T}_L(\mathcal{M})$ is a predicate encapsulating the local transaction resources managed by the locking layer, and

$$P_{upd}(M, Q) := \left(\mathop{\Large *}_{(a:v,v') \in M} a \mapsto_{\mathcal{C}} v \right) \mathrel{-\!\!*\!\!\!*} \left(\mathop{\Large *}_{(a:v,v') \in M} a \mapsto_{\mathcal{C}} v' \right) * Q \tag{5.3}$$

The construction $P_1 \wedge P_2$ can be thought of as a black box holding an unknown predicate that satisfies both P_1 and P_2 but not necessarily $P_1 * P_2$. Here, it is used to allow the caller-defined thread state to transform differently depending on whether the transaction aborts or succeeds.

To further illustrate how this specification works, consider, as an example of a branching transaction, a transaction that clamps the value of an object a to a value t (Fig. 5-3). At `Commit`, $\mathcal{M}$ takes the form

```
func Clamp(jrnl *Jrnl, a Addr, t int) {
  txn := jrnl.Begin()
  v := txn.Read(a)
  if v > t {
    txn.Write(a, t)
  }
  txn.Commit()
}
```

Figure 5-3: An example of a branching transaction.

$$\mathcal{M} = \begin{cases} \{(a : v, t)\} & \text{if } v > t \\ \{(a : v, v)\} & \text{otherwise} \end{cases}$$

In general, the expression for $\mathcal{M}$ when a transaction completes can look like a sequential version of the transaction's code. This equivalence is made explicit in DaisyNFS where a logical refinement relates GoTxn transactions, through $\mathcal{M}$, to sequential equivalents in Dafny, allowing developers to take advantage of Dafny's powerful proof automation for sequential code.

5.2 Proof Outline

The proof starts by reframing the WAL interface state in terms of per-object *durable predicates* $a \mapsto_{\mathcal{D}} v$ representing ownership over the durable value v of each object a (Fig. 5-4). It does so in the object layer by creating per-object ghost variables and relating them in an invariant to the WAL layer's connecting predicate. Since the object layer flushes after every write, its specification does not expose anything beyond the durable object state.

The journal layer associates each transaction with a *transaction context* $\mathcal{T}(A)$ representing ownership over a local transaction buffer with jurisdiction over the set of objects A. A starts empty on $\texttt{Begin}_{jrnl}$ and grows to include all objects accessed over the course of its transaction. Since the journal layer does not implement any concurrency control, clients may only safely use multiple local transaction buffers

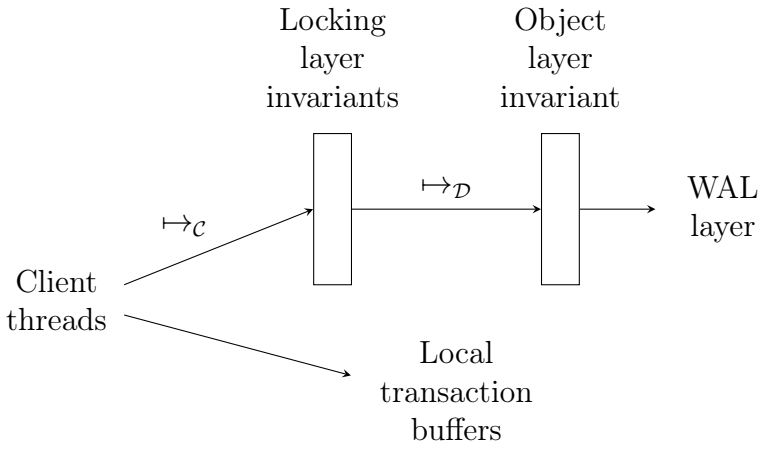

Figure 5-4: Control diagram for the object, journal and locking layers. Control over the WAL is split into per-object ownership predicates $\mapsto_{\mathcal{D}}$ and then distributed to client threads as connecting predicates $\mapsto_{\mathcal{C}}$. There are no invariants associated with the journal layer as it does not require concurrent reasoning. Rather, client threads are given direct ownership over local transaction buffers through the predicates $\mathcal{T}(\mathcal{M})$.

concurrently if the local transaction buffers have jurisdiction over non-overlapping object sets. The journal layer's specification enforces this by introducing a *modification token* Θ_a for each object a. In order to execute $\texttt{Read}_{jrnl}$ or $\texttt{Write}_{jrnl}$ on a, the proof must first *lift* Θ_a into $\mathcal{T}$ to add a to A. Doing so makes Θ_a inaccessible for the rest of the lifetime of $\mathcal{T}$, preventing a from being accessed by other transactions until $\texttt{Commit}_{jrnl}$ is called (Fig. 5-5).

The journal layer exposes the latest value v of an object a in $\mathcal{T}$ through a *local predicate* $a \mapsto_{\mathcal{T}} v$ created when the proof lifts a into $\mathcal{T}$. $\texttt{Read}_{jrnl}$ and $\texttt{Write}_{jrnl}$ only interact with local predicates, leaving durable predicates untouched. At the end of a transaction, $\texttt{Commit}_{jrnl}$ uses local predicates to update durable predicates in a Hoare quadruple of the form

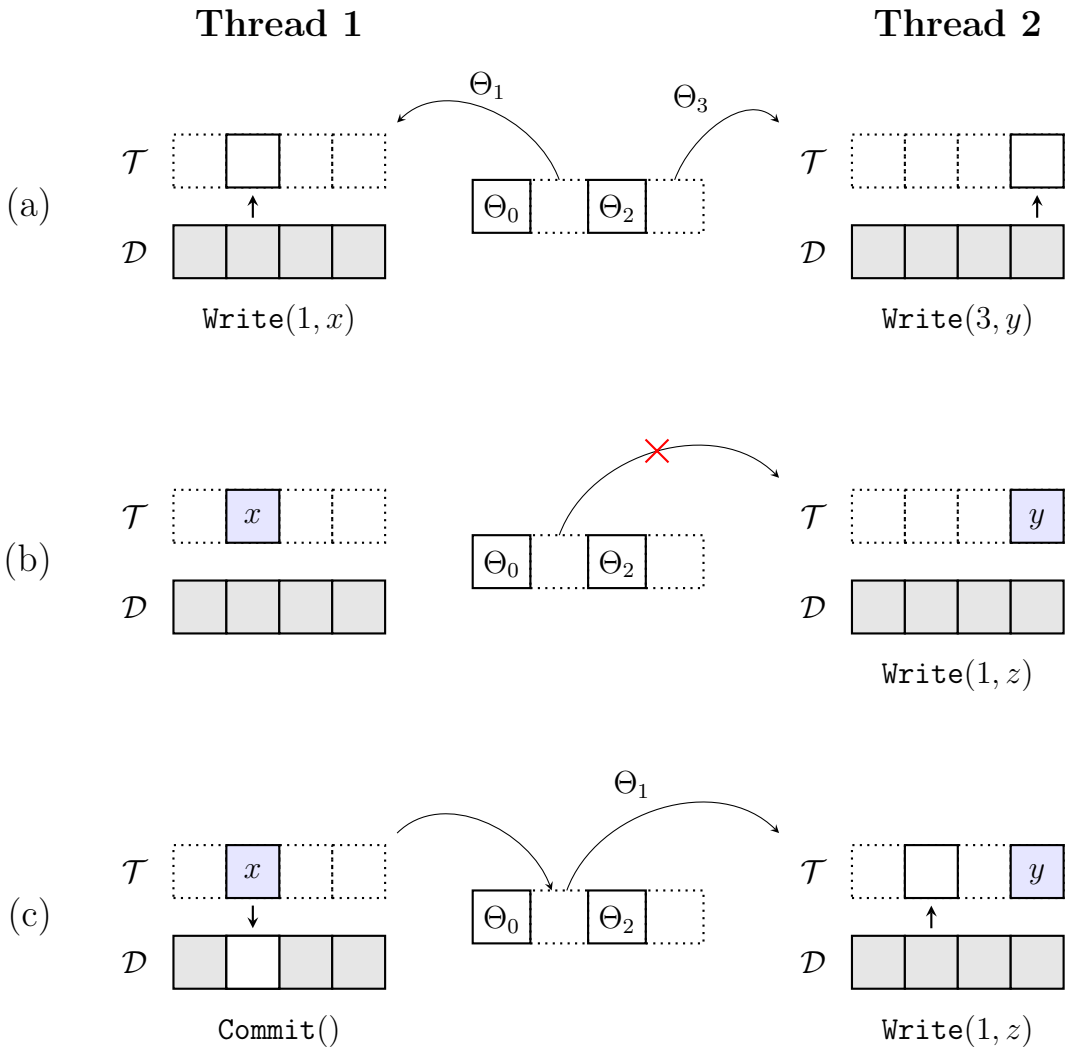

Figure 5-5: Modification tokens in the journal layer specification. (a) Threads may concurrently access different objects since they require different modification tokens. (b) Lifting an object into a thread's transaction context consumes the object's modification token, preventing it from being accessed by other threads. (c) Committing returns all modification tokens for objects associated with the transaction, allowing them to be accessed by other threads.

$$\forall \mathcal{M},\ \left\{ \mathcal{T}(\mathbf{dom}\,\mathcal{M}) * \mathop{\Large *}_{(a:v,v')\in\mathcal{M}} \left(a \mapsto_{\mathcal{D}} v * a \mapsto_{\mathcal{T}} v' \right) \right\}$$

$$\texttt{Commit}_{jrnl}()$$

$$\left\{ \mathbf{RET}\, r;\ \left(\mathop{\Large *}_{(a:v,v')\in\mathcal{M}} \Theta_a \right) * \right. \tag{5.4}$$

$$\left. \mathbf{if}\ r = \texttt{Success}\ \mathbf{then} \mathop{\Large *}_{(a:v,v')\in\mathcal{M}} a \mapsto_{\mathcal{D}} v'\ \mathbf{else} \mathop{\Large *}_{(a:v,v')\in\mathcal{M}} a \mapsto_{\mathcal{D}} v \right\}$$

$$\left\{ \left(\mathop{\Large *}_{(a:v,v')\in\mathcal{M}} a \mapsto_{\mathcal{D}} v \right) \vee \left(\mathop{\Large *}_{(a:v,v')\in\mathcal{M}} a \mapsto_{\mathcal{D}} v' \right) \right\}$$

where $\mathbf{dom}\,\mathcal{M}$ is the set of keys of $\mathcal{M}$.

Eq. 5.4 asserts that a transaction's writes are either all applied or all discarded on crash, satisfying crash atomicity. However, it does not assert that the writes are applied in a single atomic step. In particular, $\texttt{Commit}_{jrnl}$ requires ownership over $a \mapsto_{\mathcal{D}} v$ for all $a \in \mathbf{dom}\,\mathcal{M}$ throughout its execution, preventing them from being used in invariants. Instead, the locking layer relies on locks to make `Commit` appear atomic to the caller.

The locking layer uses a *lock map* implementation that maps addresses to a fixed set of physical locks. This gives the locking layer access to virtual per-object locks L_a that can be assigned lock invariants just like ordinary locks. To construct these lock invariants, the proof defines the crash borrows (Sec. 3.4)

$$B(a, v) := \boxed{a \mapsto_{\mathcal{D}} v * a \mapsto_{\mathcal{C}} v \mid \exists v',\ a \mapsto_{\mathcal{D}} v' * a \mapsto_{\mathcal{C}} v'}$$

These crash borrows, sealed at the start of the program, maintain that an object's connecting predicate always reflects the object's durable state on crash. This captures the intuition that transactions directly transform durable state, allowing developers to write crash specifications solely in terms of the connecting predicates.

Now, for each a, the proof defines the lock invariant

$$I_{L_a} := \exists v,\ \Theta_a * B(a, v)$$

Putting Θ_a in $\boxed{L_a}^{L_a}$ guarantees that a transaction may only access a while holding

L_a.

Whenever a transaction accesses an object a for the first time, the proof opens $\boxed{I_{L_a}}^{L_a}$ and lifts Θ_a into $\mathcal{T}$ to get $a \mapsto_{\mathcal{T}} v$ and $B(a, v)$. It will need these resources on `Commit` to update $a \mapsto_{\mathcal{D}} v$, recover Θ_a, and close $\boxed{I_{L_a}}^{L_a}$ (Fig. 5-6). In the meantime, the proof stores these resources in $\mathcal{T}_L$ alongside $\mathcal{T}$, updating $a \mapsto_{\mathcal{T}} v$ whenever it receives a `Write` to a. In other words, the proof maintains

$$\mathcal{T}_L(\mathcal{M}) := \mathcal{T}(\mathbf{dom}\,\mathcal{M}) * \mathop{\Large\ast}_{(a:v,v')\in\mathcal{M}} \Big(a \mapsto_{\mathcal{T}} v' * B(a, v)\Big)$$

On `Commit`, the proof opens all the $B(a, v)$ in $\mathcal{T}_L(\mathcal{M})$ to access $a \mapsto_{\mathcal{D}} v$ for all $a \in \mathbf{dom}\,\mathcal{M}$ in exchange for adding

$$P_c(\mathcal{M}) := \mathop{\Large\ast}_{a\in\mathbf{dom}\,\mathcal{M}} \Big(\exists v, a \mapsto_{\mathcal{D}} v * a \mapsto_{\mathcal{C}} v\Big)$$

to its crash condition. Since the specification of $\texttt{Commit}_{jrnl}$ does not take an atomic update, the proof uses P_{upd} (Eq. 5.3) to synchronize the connecting predicates to the durable object state only after executing $\texttt{Commit}_{jrnl}$. This is valid because the proof only guarantees that the connecting predicates match the durable object state on crash. If the system crashes after $\texttt{Commit}_{jrnl}$ updates the durable object state but before it returns, the proof applies P_{upd} after the crash to satisfy P_c.

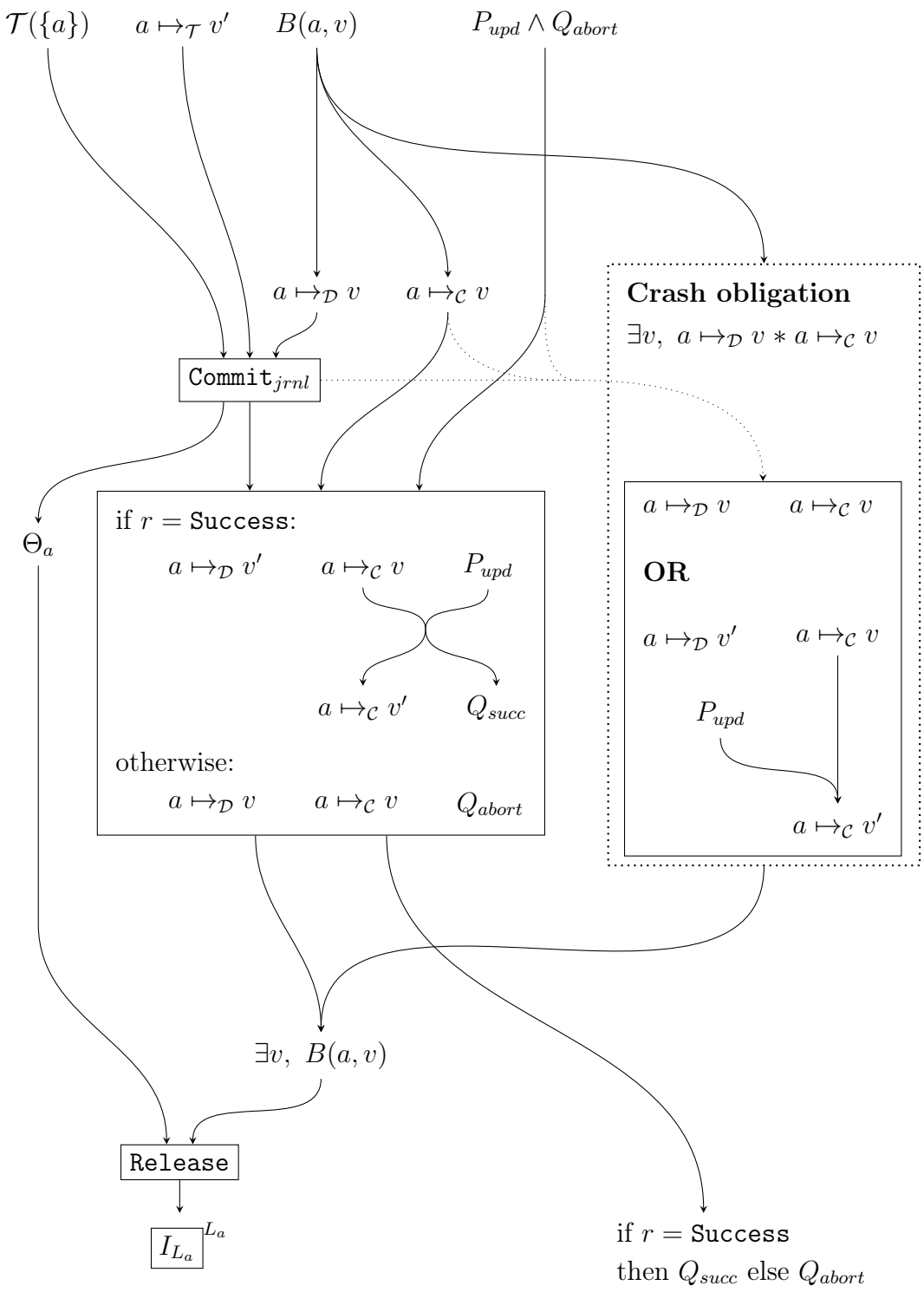

Figure 5-6: Outline of a proof of `Commit` for a transaction $\mathcal{M} := \{(a : v, v')\}$ involving a single object write.

Chapter 6

Evaluation

We evaluate GoTxn to address the following questions:

- Does GoTxn's design give it good performance?
- Does GoTxn's proof grant us confidence in its correctness?

6.1 Performance

We first show that GoTxn achieves performance comparable to unverified systems. Doing so directly is difficult since GoTxn exports a non-standard interface for defining transactions. Instead, we evaluate the performance of DaisyNFS [8], a verified NFS server that depends on GoTxn to achieve good performance. As a baseline, we use a Linux NFS server exporting an ext4 file system, mounted with `data=journal` to guarantee that filesystem operations are committed durably when they return. This allows for a fair comparison since both systems expose the same interface and the same durability guarantees.

We run all experiments on an Amazon EC2 i3.metal instance, which uses an Intel Xeon E5-2686 v4 (Broadwell) processor with 72 logical cores, 512 GB of RAM, and a local 15.2 TB NVMe SSD. To reduce variability, we limit experiments to a single 36-core socket, disable turbo boost, and disable processor sleep states.

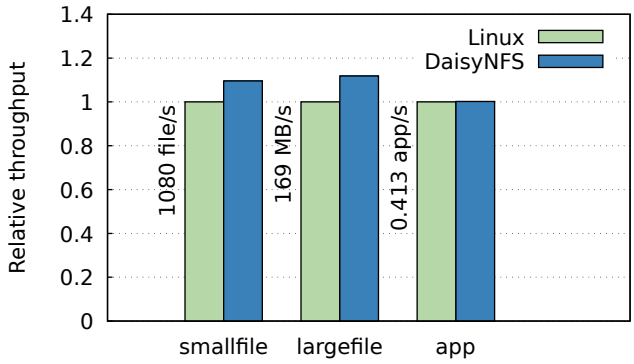

Figure 6-1: Performance comparison between DaisyNFS and the Linux NFS server.

6.1.1 Overall Performance

To evaluate single-client performance, we run benchmarks that interact with DaisyNFS and the Linux NFS server by issuing filesystem operations through the Linux NFS client. We use a benchmark suite that evaluates performance across a range of workloads:

- `smallfile` repeatedly creates, syncs, then deletes a 100-byte file, providing a measure of GoTxn's performance on small transactions.
- `largefile` appends 100 MB to a single file, providing a measure of GoTxn's performance on large transactions.
- `app` downloads and builds the xv6 [12] operating system, providing a measure of GoTxn's performance on a typical, mixed workload.

DaisyNFS achieves a similar throughput to the Linux NFS server in all three benchmarks (Fig. 6-1), demonstrating that GoTxn supports practical performance across different kinds of workloads. Both systems achieve significantly lower throughput in the `largefile` benchmark than the maximum write bandwidth of the NVMe drive (about 3 GB/s), suggesting that disk bandwidth is not a bottleneck in our experiments. Rather, each system's performance is largely determined by its ability to hide disk access latency and manage memory efficiently.

Since DaisyNFS supports fewer features than the Linux NFS server, its slightly larger throughput on the `smallfile` and `largefile` benchmarks should not be taken

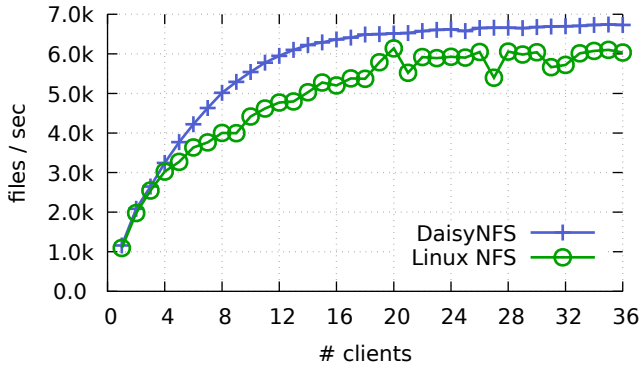

Figure 6-2: Performance of DaisyNFS on the `smallfile` benchmark with multiple concurrent clients, in comparison to the Linux NFS server.

as a sign that it is a better system overall. This result is likely due to DaisyNFS's simpler implementation requiring fewer accesses to in-memory data structures.

We evaluate the multi-client performance of GoTxn with the help of the n-parallel `smallfile` benchmark. n-parallel `smallfile` runs n `smallfile` clients in parallel, each in a separate directory so that their transactions access non-overlapping object sets. Under this benchmark, DaisyNFS achieves performance scaling similar to the Linux NFS server (Fig. 6-2), demonstrating that GoTxn effectively exploits concurrent execution to deliver increased transaction throughput. Like in the single-client experiment, DaisyNFS's slightly better throughput is likely due to its simpler implementation.

6.1.2 GoTxn's Contribution

We now show that GoTxn's design plays an important role in enabling DaisyNFS's single- and multi-client performance. In particular, we focus on the following design features:

- The WAL avoids performing disk operations under a lock, allowing multiwrite submission, logging and installation to happen in parallel.
- The WAL exposes separate `Write` and `Flush` calls, allowing the object layer to submit and flush object writes in parallel even if the writes target different objects in the same disk block.

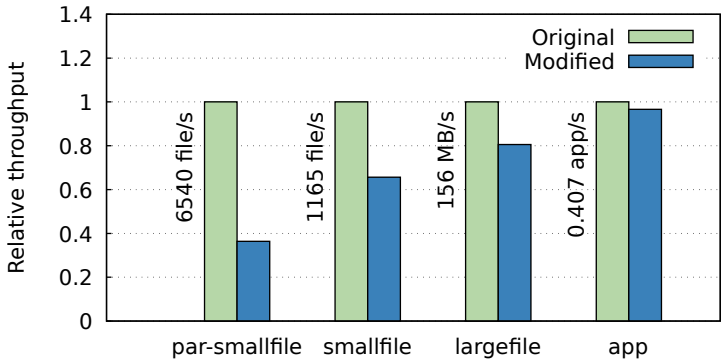

Figure 6-3: Change in the performance of DaisyNFS when the WAL is modifed to perform all disk operations under `memLock`.

- The locking layer provides fine-grained two-phase locking instead of relying on a global transaction lock, allowing it to process transactions that affect non-overlapping object sets in parallel.

- Absorption in the WAL reduces disk traffic during logging and installation.

We evaluate the impact of each design feature by measuring the performance of DaisyNFS on 30-parallel `smallfile` (labeled `par-smallfile`), `smallfile`, `largefile` and `app` with the design feature disabled.

Concurrent logging and installation. The WAL does not hold a lock when reading from or writing to disk, allowing it to accept new multiwrites in parallel with logging and installation. In particular, it does not use a lock to maintain consistency between the `memLog` and the physical log. Modifying the WAL to perform all disk operations under the `memLog` lock `memLock` results in a measurable performance drop across all benchmarks (Fig. 6-3), demonstrating that GoTxn's ability to hide the disk latency of logging and installation is important to DaisyNFS's performance. This is even despite the low latency of the NVMe drives used in our experiments, which is about 25 µs for random writes.

Separate `Write` and `Flush` calls. The object layer uses a lock to prevent writes to objects in the same disk block from interfering. Modifying the object layer to flush the WAL under this lock leads to a considerable reduction in the performance of DaisyNFS on 30-parallel `smallfile` (Fig. 6-4). This demonstrates that separat-

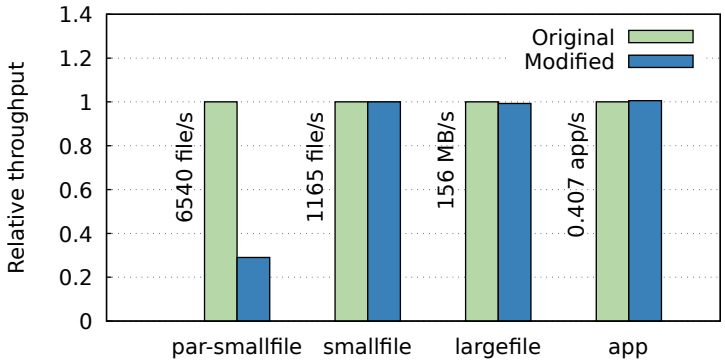

Figure 6-4: Change in the performance of DaisyNFS when the object layer is modified to flush the WAL under the object lock.

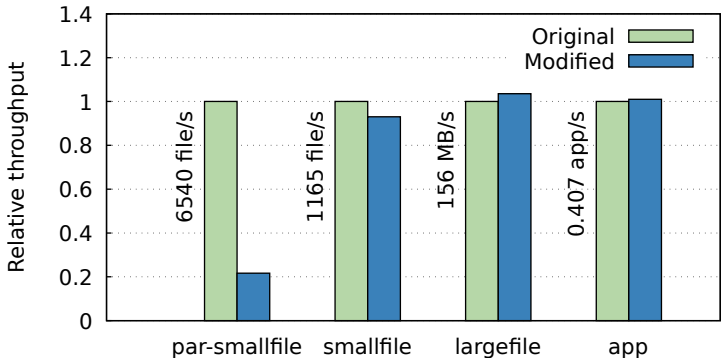

Figure 6-5: Change in the performance of DaisyNFS when the locking layer is modified to use a global transaction lock.

ing `Write` and `Flush` in the WAL's interface is beneficial to GoTxn's multi-client performance even though the WAL is flushed after every transaction.

Fine-grained transaction locks. Modifying the locking layer to use a global transaction lock instead of fine-grained two-phase locking results in a significant drop in the performance of DaisyNFS on 30-parallel `smallfile` (Fig. 6-5). This demonstrates that GoTxn's locking scheme successfully enables DaisyNFS to deliver good multi-client performance by taking advantage of opportunities for concurrent execution.

On the other hand, the slight increase in `largefile`'s throughput under the same modification shows that fine-grained locking can sometimes lead to a small performance loss (4%). This is likely due to the overhead of acquiring and releasing many uncontended locks when processing large transactions from a single client. Neverthe-

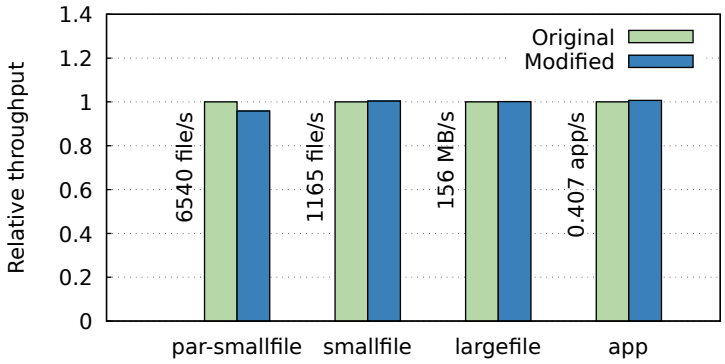

Figure 6-6: Change in the performance of DaisyNFS when the WAL is modified to always append new multiwrites to the `memLog` instead of absorbing them into existing multiwrites in the unstable region.

less, we believe that the significant gain in multi-client performance is well worth this small cost.

Absorption. Disabling multiwrite absorption in the WAL results in a small but measurable performance drop for 30-parallel `smallfile`, suggesting that absorption plays only a small role in improving multi-client performance (Fig. 6-6). This result is likely due to the high bandwidth and low latency of the NVMe drives used in the experiment. A high disk bandwidth makes disk traffic less of a bottleneck for system performance, while a low disk latency results in shorter logging cycles and thus fewer opportunities for absorption. We hypothesize that a larger difference may be observed if DaisyNFS is run on less performant storage disks or concurrently with other, more disk-intensive workloads.

6.2 Correctness

While formal verification gives us a high degree of confidence in GoTxn's functional correctness, it is important to remember that this does not eliminate all potential for bugs. For one, the integrity of GoTxn's proof depends on the correctness of its verification toolchain. This includes the Coq proof system and the Goose translation tool, particularly the models provided by Goose for the disk interface and the Go programming language.

The strength of GoTxn's proof is also limited by the correctness of its specifications. This includes the primitives from the Perennial and Iris frameworks that GoTxn depends on to express its specifications. As with any verified system, GoTxn depends on other mechanisms, such as a manual audit, to check that its specifications correctly express intended guarantees. In an extreme case, a Hoare specification with contradictory pre-conditions does not formally provide any guarantees about system behavior.

The verification of DaisyNFS demonstrates that the functional and crash safety guarantees provided by GoTxn's specifications are complete enough to be useful in reasoning about complex transactions. However, failure modes outside the scope of GoTxn's specifications can be a source for bugs. Most significantly, GoTxn's specifications do not provide any liveness guarantees, which means that they do not exclude the possibility of bugs like infinite loops or deadlocks. They also do not make guarantees about GoTxn's robustness to failure events besides system crashes such as corruption due to disk errors. Guarding against these kinds of bugs is out of scope for this work but is an interesting avenue for future research.

With these limitations in mind, we subject GoTxn to a series of tests to evaluate the effectiveness of our verification methodology. As with the performance evaluation, we use DaisyNFS with the Linux NFS client to support an interface compatible with existing filesystem test suites. We start by validating the functional correctness of DaisyNFS with the Linux regression test suites `fsstress` and `fsx-linux`. While these tests help to confirm correct behavior even under concurrent execution, they do not test crash safety. We complement these tests with the black-box crash testing framework CrashMonkey [22], which found no bugs in any supported two-operation tests.

We also test GoTxn's interface directly by fuzzing it with `gofuzz`. We use a fuzz target that executes transactions generated from the fuzz string, checking that the system does not crash and that reads always reflect the latest write. While our setup does not support testing for concurrency and crash safety, it serves as a basic test for modes of operation that might not be accessible through DaisyNFS.

Despite generating complex inputs with multiple dependent transactions, fuzzing did not reveal any bugs in GoTxn.

Chapter 7

Conclusion

This thesis presents GoTxn, a verified, crash-safe transaction system, and in particular the details of the proof of the WAL and locking layers. The discussion of these proofs demonstrates:

- How crash-aware separation logic with Perennial, including the use of crash borrows, can help proof authors reason effectively about concurrent, crash-safe systems;
- The incremental development of abstractions to manage the complexity introduced by concurrency and performance optimizations;
- The specification and verification of a transaction system that isolates concurrency and crash reasoning from the developer, accelerating the development of complex, verified systems.

Avenues for further work include extending GoTxn's proof to provide liveness guarantees and other aspects of system correctness, as well as reducing its dependence on trusted components like the Goose translation tool. GoTxn itself can also be extended to support performance optimizations common to unverified systems such as deferred durability and log-bypass writes, which depend on weaker notions of crash safety. Despite these limitations, we believe that GoTxn is an important step towards the development of practical, verified systems with crash safety guarantees.

Bibliography

[1] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential verification of serializability. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, page 31–42, Madrid, Spain, January 2011.

[2] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

[3] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018.

[4] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019.

[5] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

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

[7] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a concurrent, crash-safe journaling system using JrnlCert. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[8] Tej Chajed, Joseph Tassarotti, Mark Theng, Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential proofs. 2021.

[9] 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.

[10] 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.

[11] 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.

[12] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] David Harver Pollak. Reasoning about two-phase locking concurrency control. Master's thesis, Imperial College London, June 2017.

[25] 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.

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

[27] Joseph Tassarotti, Tej Chajed, Ralf Jung, Frans Kaashoek, and Nickolai Zeldovich. Separation logic for concurrent storage systems with Peony. 2021.

[28] The Coq Development Team. *The Coq Proof Assistant, version 8.9.0*, January 2019.