Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems

Travis Hance[†], Yi Zhou[†], Andrea Lattuada[‡,*], Reto Achermann[⋆], Alex Conway[‡], Ryan Stutsman[‡⊕], Gerd Zellweger[‡], Chris Hawblitzel[⊙], Jon Howell[‡], Bryan Parno[†]

[†] *Carnegie Mellon University;* [⋆] *University of British Columbia;*
[‡] *VMware Research;* [⊕] *University of Utah;* [⊙] *Microsoft Research*

Abstract

We present IronSync, an automated verification framework for concurrent code with shared memory. IronSync scales to complex systems by splitting system-wide proofs into isolated concerns such that each can be substantially automated. As a starting point, IronSync's ownership type system allows a developer to straightforwardly prove both data safety *and* the logical correctness of thread-local operations. IronSync then introduces the concept of a *Localized Transition System*, which connects the correctness of local actions to the correctness of the entire system. We demonstrate IronSync by verifying two state-of-the-art concurrent systems comprising thousands of lines: a library for black-box replication on NUMA architectures, and a highly concurrent page cache.

1 Introduction

Despite the importance of concurrent software, it is famously difficult to write correctly. The correctness of any one thread can, in principle, depend on changes from any other thread; developers often struggle to consider all possible thread interleavings. This reasoning becomes more difficult in aggressively optimized systems that use custom synchronization tools beyond standard abstractions like locks. Such systems often appear *anti-modular* in that they entangle synchronization logic with application logic. For example, a concurrent page cache (§5.2) might use a bit both for the synchronization purpose of read-locking a section of memory and for the logical purpose of indicating that an IO write is in progress.

In theory, formal software verification can produce provably correct code, but existing techniques have struggled to reach production-scale shared-memory systems (§9). Some work [19, 20] on verifying concurrent systems carefully focuses on networking and asynchronous disk IO, but avoids *shared-memory* concurrency. Other work [11, 12, 17, 36, 46] does tackle shared memory, but it relies on techniques which require considerable sophistication and manual effort from the developer. Still other work [38] offers simpler tools and greater automation, but the automation does not yet scale, requiring hours of CPU time for tens of lines of code (§7.1).

In contrast, IronSync enables verification of production-scale shared-memory concurrent systems, including those with custom synchronization protocols. Such verification comes with many reasoning challenges, and IronSync succeeds by carefully partitioning the system-level proof so that a developer can ergonomically tackle each challenge by formalizing her existing intuitions, supported at each stage by powerful automation.

At the lowest level, an IronSync developer uses an ownership type system to prove the data safety of her implementation. Oversimplifying, in an ownership type system, an *owned* value must be held (or referenced) by exactly one variable. A fast, deterministic type checker enforces this property, which IronSync, like Rust [28, 39] and other languages, in turn uses to enforce data safety, i.e., basic memory safety plus freedom from conflicting reads and writes. Our experience, and that of the Rust community, suggests using an ownership type system for such reasoning is relatively intuitive.

Going further, IronSync shows how the developer can additionally use ownership types to reason about the *logical correctness* of a thread's local actions on data it owns. Intuitively, in any segment of code where a thread operates only on owned data, that data cannot affect or be affected by other threads. Hence, IronSync can reason about such code using *sequential* reasoning techniques and tools honed over decades of research and development. Reasoning sequentially is more intuitive for the developer as well.

Finally, the IronSync developer must connect the locally correct thread actions to global, *system-wide* correctness. To support this step, IronSync introduces the *Localized Transition System* (LTS), which abstracts each thread's actions and the state they act upon, formalizing the intuition that in efficient concurrent programs, each thread acts only on a small fragment of the program's global state. Using the techniques above, the developer locally proves that the LTS is a sound abstraction of their implementation. Finally, IronSync soundly abstracts the LTS into a simplified program representing the whole system. This new program (with threads, locks, and other implementation details abstracted away) is far simpler to reason about, and it connects naturally to previous automated techniques for, e.g., reasoning about asynchronous IO or distributed nodes [19, 20]. Since each step on this path is proven sound, proofs at this level also apply to the implementation.

To support the advanced read-sharing patterns found in production systems, IronSync also reuses the machinery above (with a small twist) to factor out reasoning about complex synchronization primitives, so that the developer can think about them separately from the core application logic, even if

[*] Work done while at ETH Zurich

the implementation deeply entangles the two concerns.

The developer writes their implementation and performs all of the reasoning above in an extended version of the Dafny language [33] augmented with the trusted axioms and memory primitives in IronSync's framework.

Evaluation. We introduce IronSync via a series of increasingly complex examples, culminating in two production-level case studies (§5). To illustrate IronSync at scale, we verify a Node Replication (NR) library that creates a linearizable NUMA-aware concurrent data structure from a black-box sequential one [9]. To show IronSync working both at scale and with prior work on crash safety [19], we verify SplinterCache [14], a production-scale disk-backed in-memory page cache created for use in commercial products [49].

Each case study's performance matches its unverified production-level counterpart (§7.2), driving workloads of 5M updates/sec. with 192 threads (NR) or 3M ops/sec. (SplinterDB with verified SplinterCache), demonstrating IronSync has not impeded optimization through limited expressiveness or excessive proof burden. We also uncovered severe bugs (§7.3) in the unverified implementations.

Limitations. As in any verification system, the correctness of a verified IronSync program depends on the correctness of its spec, and the verification tool (Dafny [33]). Our encoding of the IronSync framework in Dafny is also trusted (although application-specific definitions are not). IronSync does not verify liveness, termination, or deadlock-freedom. We focus on data safety and functional correctness; i.e., *if* an operation returns a result, it is correct according to its spec. IronSync verifies programs against a high-level memory consistency model that distinguishes data-race-free memory from racy, sequentially consistent atomic memory. This compiles to efficient assembly on modern hardware, but it cannot exploit every optimization afforded by relaxed memory (§4.2).

Contributions. In summary, this paper:

1. Factors the proof of a production-scale concurrent system into intuitive reasoning steps that can each be supported with powerful automation.
2. Illustrates how the application of an ownership type system enables scalable, automated concurrent reasoning about both data safety *and* logical correctness.
3. Introduces Localized Transition Systems, which soundly connect *local* reasoning about a thread's actions to *global* reasoning about the correctness of the full system.
4. Enables developers to verify complex, application-specific read-sharing synchronization tools in isolation from the program's main application logic.
5. Demonstrates, via case studies, that IronSync effectively reasons about practical, complex, high-performance concurrent systems.

2 The Potential Pitfalls of Parallelism

To highlight the challenges of writing and reasoning about concurrent code, we begin with a simple bank application.

```
if accounts[A] ≥ amt {
  accounts[A] = accounts[A] - amt
  accounts[B] = accounts[B] + amt
}
```

Figure 1: Buggy code violating data safety

```
lock(accounts[A]);
sufficient_balance := accounts[A] ≥ amt;
unlock(accounts[A]);

if sufficient_balance {
    lock(accounts[B]);
    accounts[B] = accounts[B] + amt
    unlock(accounts[B]);

    lock(accounts[A]);
    accounts[A] = accounts[A] - amt
    unlock(accounts[A]);
}
```

Figure 2: Buggy code violating logical correctness

```
lock(accounts[A]);
lock(accounts[B]);
if accounts[A] ≥ amt {
    accounts[A] = accounts[A] - amt
    accounts[B] = accounts[B] + amt
}
unlock(accounts[A]);
unlock(accounts[B]);
```

Figure 3: Correct code (assuming that $A < B$, which is necessary to avoid deadlock, although proving deadlock-freedom is out-of-scope for IronSync)

This multi-threaded application maintains a list of accounts, each with an account ID and a balance, supporting one operation, $transfer(A, B, d)$, which moves d dollars from account A to account B.

Here, there are a handful of mistakes an inexperienced developer might make. First, they might write code as in Figure 1, which would be correct in a sequential program but critically flawed in a concurrent one, where, e.g., two different threads might both write to `accounts[A]`, one overwriting the other. In fact, it might be difficult to even fully characterize the buggy behavior on realistic hardware with weaker memory models [1, 2, 45], and in some programming languages, data races may even be undefined behavior [5, 52].

A naive fix would be to protect the code in Figure 1 with a global lock. Such a fix would be correct but embarrassingly inefficient. For efficiency, the developer might employ a finer-grained concurrency strategy by creating a lock for each account, reasoning that most transfers affect disjoint accounts.

Figure 2 uses a discipline of locking an account before reading/writing it, eliminating the *data-safety* problems above, since no other thread can simultaneously read/write the account information. Even with this fix, however, the program is still not *logically correct*: two different transfer operations might each check that account A has sufficient funds and then move forward even when A lacks the funds to complete both, leaving A with a negative balance.

Given enough debugging (or prior experience), the developer might eventually produce the code in Figure 3, which

holds locks on both accounts as it makes the transfer. How would the developer convince herself that she has finally produced a correct implementation?

She might first reason that the program is *data safe* because it holds the corresponding account lock whenever it accesses shared memory. Data safety rules out blatant data corruptions, making it feasible to reason about logical correctness.

Given data safety, she might then informally reason that holding both locks simultaneously gives the thread exclusive access to the portion of the state (accounts A and B) needed to correctly perform the transfer. Although other concurrent threads might simultaneously modify *other* program state, all such modifications are irrelevant to the transfer. Conversely, any changes the thread makes to accounts A and B must be irrelevant to any concurrent transfers, since those transfers must involve other accounts. Hence, she can *reason locally* about the correctness of the transfer implementation. Furthermore, from the perspective of the other threads, the transfer appears to occur atomically.

With IronSync, the developer formalizes her intuitions about data safety and logical correctness in a machine-checked way.

3 The Core IronSync Methodology

IronSync utilizes a variety of techniques, with the philosophy of using the right tool for each job. We introduce the techniques through a series of increasingly complex examples.[1] Here we focus on the toy banking application from §2 to illustrate IronSync's core ideas: **(1)** the use of an ownership-based type system, **(2)** the abstraction of threads via a *localized transition system*, and **(3)** the way we soundly compose a *localized transition system* into a *global state machine* that can soundly reason about global properties of the full concurrent program.

In §4, we introduce increasingly sophisticated features and examples, building up to our production-level case studies in §5. We defer the formalism underpinning IronSync to §6.

3.1 Achieving Data Safety in IronSync

IronSync mechanically enforces data-safety via an ownership type system. Such type systems are effective at enforcing data-safety both in unverified programming (e.g., in Rust [29]) and in verified *sequential* programming; e.g., in *Linear Dafny* [35], a version of the Dafny verification language [33] augmented with an ownership (or linear [51]) type system inspired by Rust's. In IronSync, we extend Linear Dafny with tools for logical correctness in *concurrent* settings (3.2). First, though, we overview ownership types, explain how they enforce data safety, and how this aids in verification.

In Linear Dafny, an *owned* value must be held (or referenced) by exactly one variable. Any attempts to duplicate or drop the value are rejected by the type checker. In IronSync, this ensures that data is *uniquely owned*, and hence a thread can read or write to data it owns without interference from other threads. Owned values can, however, be stored in shared (read-only) variables; Dafny's type checker ensures that the scope of such shared variables is contained within the scope of the originating owned variable, and that the owned variable is not modified until the shared variables expire.

```
method M(owned w: int, b: bool) returns (owned z: int)
  owned var x := w; // okay: consumes w
  owned var y := w; // error: w was already consumed
  if b {
    shared var s := x; // okay: shares x read-only
    x := x + 1; // error: borrowed value still live in s
  }
  x := x + 1; // okay: shared variable s has expired
  z := x; // okay: consumes x
```

Figure 4: Ownership in action

IronSync uses the ownership type system to ensure that code cannot read or write shared memory after it gives up permission to access it. Figure 5 shows an example of this approach in a portion of the API for an exclusive lock (which in turn is implemented and proven correct using lower-level IronSync primitives – see §4.2). The API uses owned and shared variables, as introduced above, and generic types (indicated by the type parameter `<T>`) as in Java or C#. Notice that the `Lock` itself is passed **`shared`**; hence, it can be referenced simultaneously from multiple threads, as expected of a lock. The caller obtains an owned *guard* value when it acquires the lock (`AcquireExcl`). The guard object, named after objects like C++'s `std::lock_guard` or Rust's `RwLockWriteGuard`, is an object that exists for the duration the lock is held. Furthermore—and similarly to `RwLockWriteGuard`—the guard object provides a type-safe means to access the data being protected by the lock. That is, after the user relinquishes the guard on release (`ReleaseExcl`), the type system will reject any further attempts to use the guard to access the lock's data.

```
method AcquireExcl<T>(shared m: Lock<T>)
    returns (owned guard: ExclGuard<T>)
method ReleaseExcl<T>(owned guard: ExclGuard<T>)
```

Figure 5: Example lock specification.

With these tools, an IronSync developer can rule out data-safety issues, because shared memory can only be accessed while holding the corresponding permission (e.g., the `ExclGuard`) from a lock (or a fancier primitive – see §4.2). Correct management of the permission is enforced by the ownership type system.

Using an ownership type system to enforce data safety is not unique to verification; e.g., this is a crucial feature of Rust. Novel to IronSync, however, is its further use of its ownership system to help the developer verify logical correctness.

3.2 Local Logical Correctness

IronSync uses ownership to simplify and automate reasoning about the correctness of a thread's actions on data it owns.

[1] All examples are fully verified and available in our open-source release.

3.2.1 Ownership Simplifies Concurrent Correctness

With IronSync, we observe that a type-enforced ownership discipline dramatically simplifies reasoning about *correctness* in concurrent settings. To illustrate, consider this short Dafny program, using its traditional heap model (i.e., without ownership types):

```
method M(data: Data)
  modifies data
  requires data.x == 2
  ensures data.x == 3
{
    data.x := data.x + 1;
}
```

The Dafny verifier can easily prove that if the precondition in the **`requires`** clause holds, then the postcondition in the **`ensures`** clause will always hold. However, if this method were part of a concurrent program, Dafny's standard sequential reasoning would not be sound, since another thread could change the heap-allocated `data` at any time.

With ownership types, however, we can rewrite it as:

```
method M(owned data: Data)
  requires data.x == 2
  ensures data.x == 3
{
    data.x := data.x + 1;
}
```

Because the type system ensures that the `data` value is uniquely owned, the verifier can once again make the assumption that no other thread is concurrently modifying `data`. Hence, the verifier can soundly use the same algorithms as before to easily verify this method.

In short, because IronSync mediates all access to shared resources via Linear Dafny's ownership type system, we can verify the logical correctness of concurrent code operating locally on owned values by using algorithms and tools that have been honed for decades on sequential verification. In our experience, this brings a substantial boost in proof automation.

3.2.2 Maintaining Local Correctness with Invariants

However, even in a program that obeys an ownership discipline, one thread seldom owns a piece of data indefinitely; instead, threads hand off ownership of shared data via synchronization primitives like locks. Hence, the developer needs a mechanism to reason about what value(s) shared data may hold when a thread acquires ownership of that data.

In one such mechanism, IronSync allows the developer to reason about locked data by associating each lock with a *lock invariant*, i.e., a property of the data protected by the lock (Figure 6). The thread can *assume* the property holds when it acquires the lock, and in exchange, it must *prove* the property holds when it releases the lock. This in turn means that the next thread to take the lock may also "take" the assumption.

Again, we see the utility of ownership types: the verifier can soundly assume that the value of `guard.v` is not being modified by other threads while the lock is held, since the thread holding the lock uniquely owns the guard. Hence, we can continue to use the efficient sequential verification techniques from §3.2.1 to prove the correctness of a thread's actions on data it obtains from other threads.

```
function is_even(x: int) { x % 2 == 0 }

// Create a new lock with an invariant that its value
// is always even. Supply a compliant initial value (2).
shared var m := NewLock(2, is_even);
owned var guard := AcquireExcl(m);
assert guard.v % 2 == 0; // Passes
guard.v := guard.v + 1;
// ReleaseExcl(guard); // error: violates invariant `is_even`
guard.v := guard.v + 1;
ReleaseExcl(guard); // okay: satisfies invariant `is_even`
// guard.v := guard.v + 1; // error: guard already consumed
// ReleaseExcl(guard); // error: guard already consumed
```

Figure 6: Example of a lock invariant. Any commented line, if uncommented, would give the resulting error. Note that `ReleaseExcl` consumes the **`owned`** `guard` object, so it cannot be used later.

3.3 From Local to Global Logical Correctness

We have shown IronSync's use of ownership to establish data safety and the logical correctness of a thread's local actions on data it owns. The final step is to explain how threads cooperate to achieve a global (program-wide) logical goal.

This final step would be trivial in a program that protects all of its state with a global lock. In that case, proofs of local correctness would suffice for global correctness, since we would simply specify the program's expected behavior via an invariant on the global lock. Such an approach would be correct but embarrassingly inefficient.

For better efficiency, the developer might employ a finer-grained concurrency strategy using many local locks. At this point, proving a global property directly becomes difficult since no thread has a global view of the system.

For such systems, IronSync provides the developer with tools to build up to global logical correctness in stages. First, the developer creates a simplified abstract model of the threads' local actions (§3.3.1). Second, the developer uses the techniques from §3.2 to *locally* prove that each thread's implementation can be soundly abstracted by the model (§3.3.2). Finally, IronSync reassembles the model's local actions into a single global abstraction of the program (§3.3.3). At this level, the developer can reason about global properties of the system, without the complexity of low-level details like thread interleaving, memory management, or even locking strategy. Because each step above is proven sound, the global correctness properties hold for the implementation as well.

3.3.1 Abstracting Local Actions

As a first step towards proving global correctness, an IronSync developer proves that their implementation corresponds to a simpler program, where threads, locks, and other implementation details are entirely abstracted away. Reasoning about the correctness of the new program is much simpler.

Returning to our bank, we would like the abstract program to operate over a simple state representing *all* of the accounts:

```
State: {accounts: map<AccountId, Balance>}
```

The challenge is to connect the application's concrete state (e.g., as stored in array) to the abstract state above.

As discussed earlier, once the implementation commits to a fine-grained, per-account locking strategy (as in Figure 3), it becomes difficult for an individual thread to reason about the global concrete state. After all, any given thread holds at most two locks at a time, and hence it cannot authoritatively reason about the state of the rest of the accounts.

Hence, IronSync introduces the concept of a *Localized Transition System (LTS)* (formally defined in §6.1), which breaks the abstract program's state into *shards* that match the "granularity" of the concrete implementation. The LTS then defines transitions that apply *locally* to only a subset of the shards. These transitions capture the work a thread performs on its local view of the state. We can later (§3.3.3) reassemble these local views into a global view.

Figure 7 shows the LTS definition for our bank example. The LTS defines a shard to be the information for a single account, matching the granularity of the locking scheme in Figure 3. The localized transition function `transfer` says that a thread that holds two shards (the "pre-state shards"), one for A and one for B, where A's shard holds at least *amt* dollars, can exchange those shards for a new pair of shards (the "post-state shards") with updated balances. This definition directly captures the intuition that a transfer only affects (and is affected by) the state of the two accounts involved. All other accounts (shards) are irrelevant.

Notice the abstraction the LTS provides: the update to the shards occurs atomically without any explicit mention of particular synchronization primitives. For complex systems, this simplification makes the application vastly easier to analyze.

In designing their LTS, a developer will typically choose a "granularity" for their shards and actions that matches the granularity of the implementation's concurrency strategy. As we discuss below, this makes it feasible to use the local correctness techniques from §3.2 to tie the implementation to the LTS. Choosing a coarser granularity would complicate the proof of this connection, while choosing a finer granularity would introduce unnecessary complexity into proofs about the global system (§3.3.3).

In practice, this means that different programs will use different LTS designs. A program with a modest concurrency strategy can afford to use coarse-grained shards, doing most of the proof work "locally" using techniques from §3.2. A program with an aggressive fine-grained concurrency strategy will use finer-grained shards, and thus put more work into spanning the gap from the LTS to the global system.

3.3.2 Tying the Concrete Implementation to the LTS

To make use of the abstraction provided by the LTS, we must soundly (i.e., in a machine-checked way) establish that

```
Shard: {id: AccountId, balance: Balance}

localized transition transfer(A, B, amt):
    for some (bal_1, bal_2) where bal_1 ≥ amt,
    pre-state shards:
        {id: A, balance: bal_1}
        {id: B, balance: bal_2}
    post-state shards:
        {id: A, balance: bal_1 - amt},
        {id: B, balance: bal_2 + amt}
```

Figure 7: Bank LTS

the implementation's behavior matches that of the LTS; thus properties proved about the LTS will meaningfully apply to the real implementation.

A key idea in IronSync is that we tie the implementation to the LTS by explicitly manifesting and manipulating the state shards of the LTS *abstraction* in the *implementation* code. The code then uses the local correctness techniques from §3.2 to prove that its manipulation of its concrete state correctly reflects LTS-defined actions on the corresponding shards.

In more detail, the implementation holds LTS shards in owned *ghost* variables. Ghost variables act like normal variables, but they serve only as "proof constructs" and are absent from the compiled executable. Making the shards owned ensures that they cannot be duplicated, preventing the implementation from holding on to two potentially contradictory shards (e.g., one that claims account A holds v dollars and another that claims it holds $v+x$ dollars). Indeed, unique ownership prevents those shards from existing anywhere in the system, even spread across different threads.

In practice, an IronSync developer will typically embed the ghost shards into their implementation and tie the ghost state to physical state via invariants. The top of Figure 8 illustrates this idea for our bank example. The implementation stores each account's concrete balance in an `Entry` datatype (similar to a `struct`) that also holds a ghost owned shard defined by the LTS. The account entries live in a sequence, each protected with a lock with an invariant that the ghost state in the shard matches the physical state of the implementation.

Looking at this program, a reader might understandably wonder what is accomplished by redundantly "doubling up" the state into a physical account balance and a ghost account balance. The key is that by doing so, we establish the formal correspondence between the concrete implementation state and the abstract state of the LTS.

The final step is to connect the implementation's *actions* to the transitions in the LTS. To do so, IronSync provides a trusted, axiomatic API for the ghost shards, with API calls that update the shards by performing valid transitions of the verified LTS. Each call *consumes* the old owned shards, and *produces* new owned shards. As shown in Figure 8, this means that during a transfer, the developer can update the physical state of the account balances and then atomically exchange (via `LTS_transition`) the old shards for a new pair representing the LTS state after performing the abstract `transfer` transition. These new shards match the concrete

```
datatype Entry = Entry(bal: int, ghost owned shard: Shard)

shared var accts : seq<Lock<Entry>> where
  ∀i, accts[i] has lock invariant: (entry: Entry)
    ⇒ entry.shard == Shard({id: i, balance: entry.bal})

method DoTransfer(A: AccountId, B: AccountId, amt: Dollars)
  // Acquire locks on both accounts.
  owned var guardA := AcquireExcl(accts[A]);  lock A
  owned var guardB := AcquireExcl(accts[B]);  lock B

  // These follow from the lock invariant.
  assert guardA.v.shard.bal == guardA.v.bal;
  assert guardB.v.shard.bal == guardB.v.bal;

  // Physically move `amt` from one account to the other.
  guardA.v.bal -= amt;  debit A
  guardB.v.bal += amt;  credit B

  // Invariant is temporarily broken.
  assert guardA.v.shard.bal != guardA.v.bal;

  // Perform the transfer transition of the LTS
  // as a ghost operation.
  guardA.v.shard, guardB.v.shard := ghost transition
    LTS_transition("transfer", guardA.v.shard, guardB.v.shard, amt);

  // Lock invariants have been restored.
  assert guardA.v.shard.bal == guardA.v.bal;
  assert guardB.v.shard.bal == guardB.v.bal;

  // We can now release the locks.
  ReleaseExcl(guardA);  unlock A
  ReleaseExcl(guardB);  unlock B
```

Figure 8: An implementation of our bank example. Figure 9 illustrates one possible execution.

state, satisfying the corresponding lock invariants and hence allowing the locks to be released.

Hence, we can soundly reason about the implementation's concrete actions on concrete state using the LTS's abstract transitions on its abstract shards. IronSync's trusted API allows the programmer to make this connection locally in the implementation code by showing that a sequence of physical steps are consistent with the "large" atomic steps of the LTS.

We illustrate this process in Figure 9a, which shows one invocation of the `DoTransfer` method from Figure 8. The illustration depicts the relationship between the ghost shards of the LTS (dashed blue boxes) and the physical values stored in memory. Time runs along the x-axis; each vertical gradation represents a fine-grained period, such as a single instruction.

Initially, Thread 2 holds no locks. It receives a client request to transfer \$7 from A to B. The developer knows that the relevant LTS transition requires atomically interacting with the A and B shards, so the thread's first step is to acquire lock A. Lock acquisition brings into Thread 2's scope both permission to observe the physical value of A (via a pointer; the physical value does not move from the heap, of course) as well as the ghost shard for A.

Later, Thread 2 likewise acquires the physical B state and its ghost shard. The ghost LTS transition requires that shard A has a value greater than the \$7 transfer. Thread 2 confirms this by checking the physical value of A, which it knows (from the lock invariant) matches the ghost shard for A.

Then Thread 2 debits \$7 from the physical value of A. Note that the ghost shard has not changed; no LTS transition allows debiting A all by itself. The lock invariant is temporarily false, which is fine, since Thread 2 still holds the lock. Next, Thread 2 credits \$7 to the physical value of B.

Thread 2 cannot release the locks until it restores their invariants. Hence, it invokes `transfer(A,B,7)`, the ghost LTS transition from Figure 7, which consumes the shards $A : 9, B : 1$. As a ghost transition, this happens instantaneously. The transition yields (by postcondition) the new shards $A : 2$, $B : 8$, which the thread proves match the corresponding physical values. Having restored the lock invariants, the thread completes its work by releasing its locks, one at a time.

For simplicity, we do not illustrate any activity on Thread 1. However, observe that it could, with any interleaving, acquire noncontending locks and interact with their associated state.

3.3.3 Global Logical Correctness With the GSM

To reason about the logical correctness of the entire multi-threaded program, we reassemble the shards of the LTS into a representation of the program's global state, and similarly, we translate the local transitions of the LTS into global transitions over the global state. We call the result (formally defined in §6.1) a Global State Machine (GSM).

Figure 10 shows the developer's definition of the GSM for the bank example. The `State` now holds all of the accounts. The `transfer` transition is an atomic step that reads and writes the global `state`. As in Figure 9b, the GSM's state only changes—atomically—at the moment Thread 2 invokes the ghost LTS transition. Hence, regardless of low-level thread interleaving, from GSM's global perspective, the state advances through a sequence of atomic global transitions, even as the physical values are updated asynchronously. This greatly simplifies reasoning about global correctness.

Indeed, since the GSM is a standard state machine, we can employ standard reasoning techniques honed by decades worth of research [32], including prior work automating such reasoning [19, 20]. For example, we can easily prove that account balances never go negative, or that the total amount of money across all accounts is always preserved. In both cases, the proof proceeds by showing that the property holds in an initial state, and then showing that if it holds before an atomic transition, then it also holds afterwards. For the bank example, these proofs are produced fully automatically.

In contrast, it would be impossible to talk about such global properties from within the implementation, where a given thread only ever holds at most two locks.

Soundly Assembling the GSM. To compose the LTS shards into the GSM's state, the IronSync developer must declare a datatype that can hold one or more shards. §6.1 discusses the formal rules the datatype must obey, but a common pattern is to use a (partial) dictionary from an application-specific identifier (e.g., an account ID) to the corresponding shard. The developer then proves the soundness of each LTS

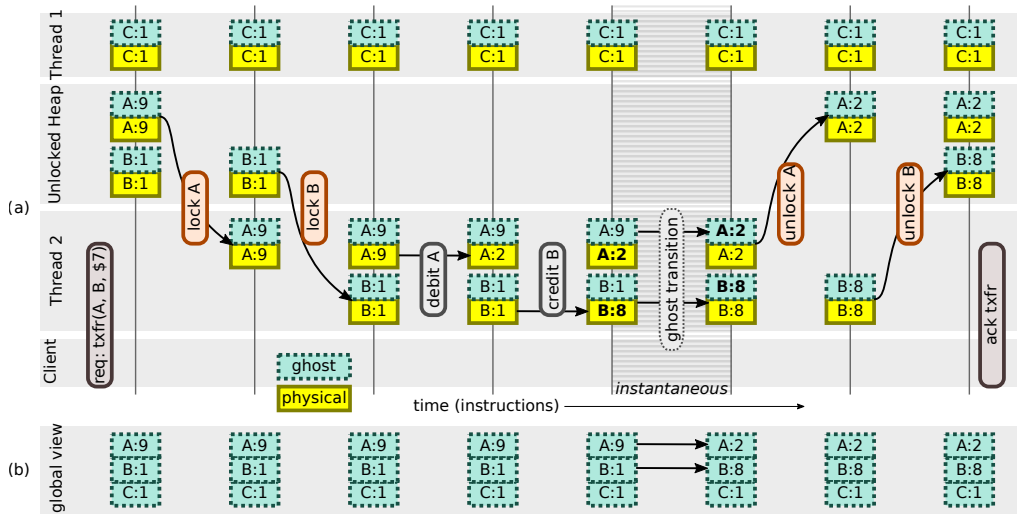

Figure 9: **(a)** Ghost values (dashed cyan boxes) travel alongside physical values (solid yellow). Ghost values are atomically updated according to the LTS rules (see §3.3.2 for details). **(b)** LTS transitions, in turn, are abstracted into the GSM (§3.3.3).

```
State: {accounts: map<AccountId, Balance>}

atomic transition transfer(A, B, x):
    if state.accounts[A] ≥ x {
        state.accounts[A] -= x;
        state.accounts[B] += x;
    }
```

Figure 10: Bank GSM

transition, with respect to this dictionary. Oversimplifying, this proof proceeds as follows. We imagine starting with a dictionary that contains at least the transition's incoming shards (in Figure 7, the initial account information for *A* and *B*), and possibly some others as well. We remove the *incoming* shards from the dictionary and hand them to the localized transition. We then try to combine the *outgoing* shards with the dictionary. If this results in a well-formed dictionary (i.e., no keys are duplicated), the transition is valid and can be lifted to an atomic transition of the GSM (as shown in Figure 10).

4 Advanced IronSync Techniques

Verification of real concurrent programs often has additional challenges beyond those of our "toy" banking example. This section illustrates, via examples, IronSync's solutions to two situations that arise in a real concurrent system:

- An abstraction of the program state (like the GSM) might still not be abstract enough to be a useful specification.
- Developers employ custom synchronization tools (beyond simple locks), plus optimizations like read-sharing.

We apply these solutions to the complex case studies in §5.

4.1 Specification via Refinement

For trivial programs like our bank, one might accept an invariant as the definition of correctness. For substantial programs, we prefer to express correctness via a trusted specification that precisely defines the program's expected behavior, and then prove that the implementation refines it; i.e., every execution

```
Shard: {idx: nat, entry: (Key, Value)?}

localized transition insert(key, value):
    for some (i, j, k⃗, v⃗)
        where i = hash(key) and key ∉ {k_i, ..., k_j}
    pre-state shards:
        { for m = i .. j | {idx: m, entry: (k_m, v_m)} }
        {idx: j+1, entry: null} // First empty slot
    post-state shards:
        { for m = i .. j | {idx: m, entry: (k_m, v_m)} }
        {idx: j+1, entry: (key, value)} // Holds (key, value)
```

Figure 11: **A Hash Table LTS Transition** that inserts a new `(key,value)` pair at index `j+1`; the other shards are unmodified, but serve to justify that `j+1` is the correct index.

of the implementation is an execution of the spec.

A hash table's spec, for example, is a simple dictionary, succinctly expressible in 10-20 lines. Its implementation obeys the spec while providing good performance. For instance, a Robin Hood Hash Table (RHHT) [10] stores key-value pairs in an array and locates keys via *linear-probing* [30]: given a key, probing starts with the key's hash index and continues sequentially until the key or an empty slot is found.

To exploit concurrency, a developer might add multiple threads and create a lock for each slot in the array. A straightforward concurrency strategy would have a thread lock the entire range of slots needed for each linear probe, complete its operation, and then release the locks. To express this strategy in IronSync, per §3, the developer defines an LTS (Figure 11) with shards at the granularity of the locking scheme: each shard of the LTS represents a single array slot.

Once the LTS is proven sound, the developer uses IronSync to reassemble the LTS into the GSM comprising the full sequence of optionally-occupied slots. They then prove that the GSM refines the spec by establishing invariants. One RHHT invariant is that each key in the table can be found in a contiguous range of non-empty slots starting from the key's hash index. Proving refinement via such invariants is

straightforward using standard techniques [32] previously encoded in Dafny [19, 20].

4.2 Lower-Level Memory Primitives

The previous examples are built from locks, which help maintain data safety. In practice, many advanced concurrent systems do not use locks, but rather custom synchronization tools built from lower-level primitives. Supporting such advanced systems is a core IronSync goal, and hence IronSync makes these lower-level primitives its base and then verifiably constructs locks and other synchronization tools from them. In this section, we introduce IronSync's primitives and, as a warm-up, see how they let us verify a basic mutual-exclusion lock.

Consider a lock implemented with two fields: a boolean *flag* indicating whether the lock is taken, and a slot for the *data* being protected by the lock. Defining operations on these fields must be done in terms of a *memory-ordering model*, which dictates when different threads may disagree on the ordering of reads and writes. Developers must take care to use special, slower instructions to synchronize threads when necessary, and such subtleties are notoriously difficult to handle correctly, especially since the details depend on the hardware platform (e.g., x86-TSO [45] or ARM [1]).

IronSync's memory model is based on the C++11 memory model [5, 7], which abstracts over these hardware differences by providing a distinction between *non-atomic* memory (the most common, "normal" memory) and *atomic* memory. Non-atomic memory access compiles to fast instructions, while atomic memory (depending on how it is used) may compile to slower instructions, possibly involving memory fences. To make this dichotomy sound, the C++ model requires all non-atomic accesses be data-race-free (a burden placed on the programmer); however, the atomic memory allows contended access. In the lock example, multiple threads might contend to access the *flag*, but the thread that wins will have exclusive access to the *data* field, making its accesses data-race-free.

IronSync supports data-race-free non-atomic memory and sequentially-consistent atomic memory. Specifically, it takes advantage of the C++11 memory model's *DRF-SC property*, which states that if all non-SC memory accesses are data-race-free, then the entire execution is sequentially-consistent. By allowing data-race-free memory for the common cases, IronSync takes advantage of much of the speed afforded by modern hardware, although it does not take advantage of the weaker atomic memory orderings (e.g., *release-acquire* ordering or *relaxed*).

In particular, IronSync supports these two modes of shared-memory-access through two of its trusted primitives, `Atomic` for word-sized atomic memory and `Cell<T>` for non-atomic memory storing arbitrary types `T`. To ensure that access to a shared `Cell<T>` is data-race-free, IronSync requires a thread to own a special ghost object of type `Permission<T>` for reading and writing. Meanwhile, IronSync treats the sequentially-consistent atomics as if they were "virtual locks" that can be unlocked for a single atomic operation; they can then use lock invariants, as before, to verify code that manipulates ghost objects in the virtual lock. `Atomic` supports common atomic operations, like compare-and-swap and atomic addition.

With these tools, the developer can verify a lock as follows: they declare the *flag* field as an `Atomic` and the *data* field as a `Cell`. They store the ghost `Permission` object for the `Cell` in *flag*'s virtual lock. By reading and writing to *flag* (e.g., with an atomic compare-and-swap), threads can transfer ownership of the `Permission`, allowing them to access the *data* field in a data-race-free manner. This process is verified by IronSync, which checks that the invariant on the virtual lock is maintained.

4.3 Read Sharing

Crucially, data-race-freedom does not preclude all simultaneous access. While it prohibits a write from occurring simultaneously with a read or another write, it does permit multiple simultaneous readers. This read-sharing is crucial for performance in many applications; however, to make use of it, the developer must still ensure that threads obey some single-writer, multiple-reader protocol. In such a protocol, the developer ensures that there can be a single writer or multiple readers at any given time, but never both at the same time (and of course never more than one writer).

The challenge with read-sharing protocols is that there is no optimal way of accounting for the shared state. For example, a particularly common protocol uses reference-counting, e.g., in a reader-writer lock, but even here, there is no universal way to implement a reader-writer lock. Our case studies (§5), for example, employ two different custom-built reference-counting-based locks, and locks aren't even the end of the story. Our NR case study (§5.1) uses a lock-free cyclic buffer, where multiple threads share read-access to entries, and where the safety is guaranteed by a protocol of head and tail pointers.

In IronSync, the developer can implement and verify a read-sharing protocol, including any of the above, by designing a particular kind of LTS, which we call a *guard protocol*, and proving that it enforces safe access to shared state. A guard protocol is an LTS whose state has an explicit notion of a stored (ghost) object, along with a notion of depositing and withdrawing that object. Intuitively, the program begins with a unique reference to an object (e.g., the ghost `Permission` for a `Cell` – see §4.2). To create read-shared references that it can give to other threads, the program "deposits" the object into the guard protocol, and in exchange it can obtain one or more *guards*. A guard is simply an LTS state shard that acts as a "witness" that the object has been deposited (and not yet withdrawn). Once all the guards (i.e., read-shared references) are returned, IronSync allows the program to "withdraw" the reference from the LTS and use it once again for mutation.

To demonstrate the soundness of their guard protocol, the library developer must show that it satisfies two obligations.

First, they show that guard shards only exist when an object has been deposited (and not yet withdrawn). This prevents the library from synthesizing bogus read-shared references. Second, they show that the LTS' withdrawal transition only occurs when an object is in fact deposited and there are no outstanding guards.

Once the guard protocol is proven sound, IronSync provides the library developer with an extended version of the trusted shard API from §3.3.2. Recall that the standard API consumes and produces owned shards. The API for guard protocols, however, allows a thread holding a read guard to acquire a shared version of the protected data; e.g., a shared `Permission` for a `Cell`, which the `Cell` API requires for read access to its concrete memory, but which doesn't suffice to use the API for writing. Hence, the developer can ergonomically manipulate shared data using Dafny's shared variables, with the assurance that all accesses are data-race free.

Crucially, IronSync's general approach to read sharing enables a developer to devise protocols that are drastically different from a read/write lock. For example, in the cyclic buffer (§5.1), threads read entries (via ghost guard objects) and use a head pointer to indicate when they are done; other threads look at these head pointers to determine when it is safe to garbage collect the entries and overwrite them (requiring a withdraw).

5 Case Studies

IronSync, we have seen, comprises a collection of tools: (ghost) ownership types, LTS abstraction, state-machine refinement, and automated verification. To test that this collection suffices to verify modern production-scale systems (i.e., systems notable for their performance, which they achieve through non-trivial concurrency patterns), we select two such systems and produce verified implementations within IronSync. These particular systems were chosen, in part, because there was independent interest in verifying their correctness from the systems' designers. We compare our case studies to those in prior work in §9.

By producing implementations that match the originals in design and performance (§7), we show that writing a system in IronSync does not sacrifice performance-critical concurrency patterns. Of course, our implementations are not identical to the originals: ours are written in Dafny (and compiled to machine-generated C++ code), and they make a few minor deviations from the originals (§7.2). Nonetheless, the exercise does, as a bonus, yield some insight into the originals (§7.3).

Overview. Both case studies are complex; for each, correctness depends on myriad interlocking moving parts. Hence, we show how an IronSync developer divides the proof work into manageable subtasks, and chooses the right IronSync tool for each.

Specifically, NR (§5.1) shows how to pull together all the IronSync features discussed earlier. With SplinterCache (§5.2), we also verify the program's use of an external disk.

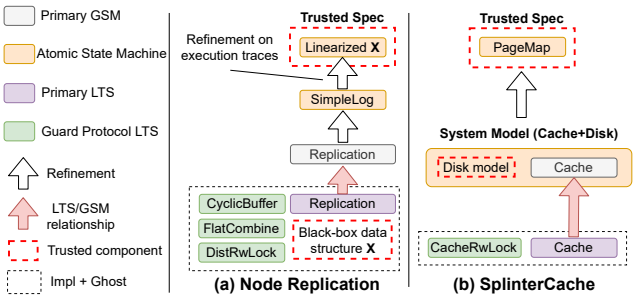

Figure 12: Proof architecture of our case studies.

Common Architecture. Each case study follows a similar high-level structure. Each has a *primary LTS* (and a corresponding GSM), which is used, via state-machine refinement (§4.1), to establish that the program meets its specification. In addition, each program uses complex synchronization logic that makes data safety challenging, so each case study also uses several secondary LTSes as guard protocols (§4.3). Figure 12 summarizes the architectures for the case studies in terms of these components.

5.1 Node Replication NR

NR [9] is a concurrency library that transforms a black-box, sequential data structure into a linearizable, NUMA-aware concurrent version. NR works by replicating the sequential data structure on each NUMA node, using an operation log to maintain consistency. Replicas benefit from read concurrency using a reader-writer lock designed to minimize reader contention [50] and from write concurrency using *flat combining* [22], which batches operations from multiple threads to be executed by a single *combiner* thread. The combiner appends the batched operations to the log; other nodes read the log and update their local replica copies. The original, unverified NR implementation is ∼1000 lines of Rust. NR has recently been adopted by NrOS [6], which uses it to implement scalable versions of a wide range of OS subsystems.

Verification Objective. Our verified implementation also takes a user-provided black-box data structure, this time one with a functional spec. Verified NR produces the replicated data structure *and* a proof that this replicated system meets the same functional spec *linearizably*.

Proof Overview. As in our earlier examples, we can coarsely divide our tasks into data safety and logical correctness. In some places, the reference implementation uses Rust's `unsafe` code, so these parts pose challenges for our verified implementation. In Rust, `unsafe` means that the code foregoes Rust's usual safe aliasing checks and places the burden of correctness on the programmer. In IronSync, this means that we cannot (solely) rely on the ownership type system to ensure data safety. Luckily, IronSync has a verified alternative: the guard protocol.

As an example, consider the cyclic buffer at the center of NR's coordination, used to broadcast messages from one node to all other nodes. Using a stringent protocol of head and tail

pointers, NR ensures that each node reads each message after it is written but before it is garbage collected, and further, that these reads and writes are properly synchronized. Notice how this custom protocol of head and tail pointers is used in place of a general utility for safe data access (like a mutex). Figure 13 shows pseudocode for parts of this protocol, delineating the read sections and write sections for buffer messages. Data safety, here, approximately amounts to saying that the write section never overlaps any another section. This read-sharing pattern is exactly what guard protocols are designed to support. We can construct such a protocol by identifying the instructions relevant to data safety (marked ★ in Figure 13) and abstracting them into an LTS.

NR has two more places where we need to do something similar. One is a specialized lock that protects the per-node replicas. These locks are designed with multiple reference counters to reduce thread contention; using a guard protocol, we can verify the lock and provide an API similar to the lock API discussed in §4.3. Finally, NR uses a flat-combining algorithm, and so we must reason about the synchronization of data between a client thread and a combiner thread.

With the three guard protocols, DISTRWLOCK, CYCLICBUFFER, and FLATCOMBINE to "patch up" holes in the ownership type system, the only remaining task is to prove a linearizable specification. We use an LTS, REPLICATION, to track **(i)** all in-progress updates and reads, **(ii)** the states of node-local replicas, **(iii)** the full history of event messages, and **(iv)** version numbers for the replicas. Since so much of the synchronization logic is already handled by the guard protocols, the LTS abstraction is dramatically simpler than the NR system as a whole: we have relegated entire subsystems to being "mere implementation details."

Now, from REPLICATION, we can construct the GSM as an abstraction of the global system behavior. Next, we need to establish a state machine refinement to a linearizable specification. This is challenging because REPLICATION has *future-dependent linearization points*; hence, we cannot prove that REPLICATION refines a linearized state machine with a single-state abstraction function. Instead, we have to prove a theorem operating over arbitrary execution traces. To simplify this task, we split it into two steps: first we prove, via a single-state abstraction function, that REPLICATION's GSM refines a simpler state machine, SIMPLELOG, which only tracks a log and the index of the latest linearized operation. Abstracting out replicas makes SIMPLELOG much easier to analyze, and the theorem over execution traces becomes tractable.

Notice that in Figure 13, the CYCLICBUFFER and REPLICATION subsystems are heavily interleaved, with some instructions even being part of both. We discuss this in §8.

5.2 SplinterCache

SplinterCache is a production-grade in-memory page cache used by the key-value store SplinterDB [14], which was created for use in VMware's vSAN [49]. SplinterCache is built

```
fn append(ops):
  tail := globalTail;
  if tail + ops.length > globalHead + SIZE:
    wait and retry;
  compare_and_swap(globalTail, tail, tail + ops.length);

  for i in 0 .. ops.length:
    j := (tail + i) % SIZE;
    live_bit := ((tail + i) / SIZE) % 2;
    log[j].op = ops[i];      // Non-atomic write
    log[j].alive = live_bit; // Synchronizing write

fn dispatch():
  head := nodeLocalVars.localHead;
  tail := globalTail;
  for i in head .. tail:
    j := i % SIZE;
    live_bit := (i / SIZE) % 2;
    wait until:
       log[j].alive == live_bit; // Synchronizing read
    op := log[j].op;             // Non-atomic read
    applyUpdateToReplica(op);
  atomic_max(maxReplicaVersion, tail);
  nodeLocalVars.localHead := tail;
```

Write section

Read section

Figure 13: Pseudocode of key NR algorithms, omitting ghost shards. Shared variables are **bold** for atomics and *italics* for non-atomics. ★ is code relevant to CYCLICBUFFER; ■ to REPLICATION. Ranges show where it is safe to read and write `log[j].op`; CYCLICBUFFER proves disjointness.

for loads where it may have 100GiB of RAM available and for use with low-latency IO devices. Clients can acquire a lock (reader or writer) on a disk page by its address, and the cache abstracts the details from the client. Internally, it loads that page into memory if necessary, and it handles writeback and eviction. Its optimizations include prefetching and batched IOs. It attempts to flush pages to disk before they need to be evicted. The reference code is $\sim$ 2000 lines of C.

Verification Objective. We characterize the behavior of the cache operating together with an external disk, using a trusted model of the disk. Our high-level spec, PAGEMAP, maps each block address to two 4KiB pages, an on-disk and an in-memory version. The system may nondeterministically overwrite the on-disk version with the in-memory one.

A cache makes weaker promises than a key-value store or file system, which might offer snapshot consistency. However, this doubled-up mapping spec still constrains behavior in the event of a crash, demonstrating how IronSync programs can integrate with prior approaches to verifying crash safety [19].

Proof Overview. Once again, we divide our proof into data safety and logical correctness.

The SplinterCache uses a complex locking scheme to protect the in-memory cached pages. The cache implementation needs to acquire a write lock to write to a page or a read lock to read from a page, but there are some subtleties: even if the client intends to only read a page, the cache may still need to load it from disk, which means writing the contents into memory, which requires a write lock on the memory page.

For efficiency's sake, the locking protocol has a variety of special states for handling situations like the above. More specifically, the lock has bit flags for states (not all mutually exclusive): **(i)** `WriteBack`: the page is being written to disk, effectively an extra read-lock; **(ii)** `Loading`: the page is being

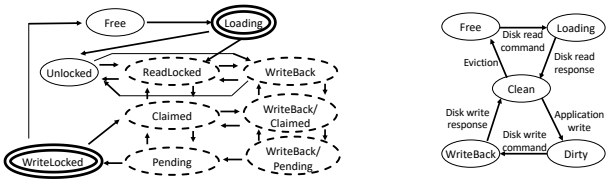

Figure 14: **(a)** Left, transitions for the lock status of a single cache entry in CACHERWLOCK. Dashed ovals represent read-locked states; double ovals represent write-locked states. **(b)** Right, transitions for the status of a single cache entry in CACHE. Both figures are simplified abstractions.

loaded from disk; **(iii)** `Free`: this entry is not assigned to a disk page; and **(iv)** `Claimed`: the claiming thread has the exclusive right to upgrade a read lock to a write lock. These states allow optimizations; e.g., an entry marked `Free` cannot be locked, so a thread that loads a page into a free entry can skip the usual check that there are no readers. Even beyond these states, the lock has the same multiple-reference-counters optimization as NR's DISTRWLOCK described earlier. Figure 14a summarizes all the states.

As usual, we handle the reader-writer lock with a guard protocol. This leaves logical correctness, which we prove with the CACHE LTS. It tracks the information needed to prove consistency properties between cache and disk; e.g., it maintains a two-way mapping between cache entries (indexed by *entry numbers*) and disk pages (indexed by *page numbers*). It tracks the status of each entry, which may be either *empty* (i.e., not corresponding to any page), *loading*, *clean*, *dirty*, or *writeback-in-progress*. This is summarized in Figure 14b.

Next, we can construct the CACHE GSM as a sound abstraction of our implementation. We then apply previous techniques [19] to integrate CACHE with a (trusted) model of the asynchronous disk. This yields a state machine CACHE+DISK which abstracts the behavior of the entire system. Here, we prove relevant invariants: the bidirectional mapping is self-consistent; outstanding IO requests agree with the *loading* and *writeback* statuses; an in-memory clean page matches the on-disk page. We finally prove that CACHE+DISK refines PAGEMAP as the high-level specification.

6 Formalism and Implementation

IronSync's Trusted Computing Base comprises the following:

- A trusted programming language and verifier.
- A trusted library of shared-memory primitives (§4.2).
- A trusted library of formal definitions and axioms for LTSes, guard protocols, and state machine refinement.

The Language. IronSync is built on Linear Dafny [35]; we added ghost ownership types to its existing (non-ghost) ownership types, and we also supply `Atomic` and `Cell` types (§4.2) for shared memory. To prevent unsoundness with concurrent threads, IronSync disallows Dafny's traditional support for aliasable mutable objects.

IronSync code compiles via Dafny's C++ backend, and uses `std::atomic` to implement IronSync's `Atomic`. Therefore, the C++ compiler is also part of the trusted toolchain, notably including its mapping from C++'s memory ordering model to the hardware's. Specifically, we rely on the compiler to insert memory fences appropriately for any `Atomic` memory locations, thereby providing the sequential consistency guarantees for the language runtime.

6.1 Formal Definitions

We briefly summarize the mathematical formalism underpinning IronSync's LTS (§3.3.1) and the guard protocol (§4.3). These definitions are exposed to the IronSync developer via an axiomatically trusted library in Dafny.

IronSync introduces LTSes to formalize the idea that a transition updates and depends on only a portion of the state, while the rest of the state is *irrelevant*. This formalization encodes shards as elements of a *monoid*, an established tool from concurrency reasoning from separation logic [26].

Definition 1 (LTS: Localized Transition System) *A Localized Transition System is a triple* $(M, \text{Init}, \tau_{local})$. *Here, M is a commutative monoid, that is, a set with a composition operator* $(\cdot) : M \times M \to M$ *which is associative and commutative, and with a unit element* $\varepsilon \in M$ *(i.e.,* $\forall m \in M.\, m \cdot \varepsilon = m$*). Meanwhile,* $\text{Init} : M \to \mathsf{bool}$ *represents valid initial states, and* $\tau_{local} : M \times M \to \mathsf{bool}$ *is a "local transition function."*

This essentially says that an element $m \in M$ represents partial information about a state of the system, while the composition of two elements gives us a way to combine the partial information about different components. Thus a transition τ makes sense even on pieces of partial information. In the bank example discussed at the end of §3.3.3, each m is a (partial) dictionary from account IDs to account info.

The IronSync framework then defines the global state machine (GSM) in terms of the LTS by taking elements of m that represent a complete view of the system (e.g., a dictionary containing all of the bank's accounts). Specifically, we define a transition on a complete state by splitting it in two: one part to be operated on, and one part that is irrelevant to the transition, and then performing the transition on the first part:

Definition 2 (GSM: Global State Machine) *Given an LTS* $(M, \text{Init}, \tau_{local})$, *we define a global transition function,*

$$\tau_{global}(s, s') \triangleq \exists d, d', e.\ \tau_{local}(d, d') \wedge (s = d \cdot e) \wedge (s' = d' \cdot e).$$

We call $(M, \text{Init}, \tau_{global})$ *the* Global State Machine.

In the bank example, d would be a dictionary holding keys for the two accounts involved in a transfer, and e would be a dictionary holding all of the other accounts.

6.2 Guard Protocols

A Guard Protocol consists of **(i)** an LTS with a notion of ghost objects that may be *deposited* or *withdrawn*, and **(ii)** a notion of a *guard*, a state shard that locally guarantees a particular

ghost object is deposited. Concretely, a Guard Protocol is defined by its relation to the following state machine that formalizes "deposited state."

Definition 3 (Safety-Deposit State Machine) *Given a set T, a* Safety-Deposit State Machine *is a state machine defined over the state $T \cup \{empty\}$, with transitions, for all $t \in T$:*

$$empty \xrightarrow{\text{deposit}(t)} t \qquad t \xrightarrow{\text{withdraw}(t)} empty$$
$$empty \xrightarrow{internal} empty \qquad t \xrightarrow{internal} t$$

Here, T is the set of ghost objects that can be deposited.

A developer defines an LTS for their Guard Protocol and proves it sound based on the definition below, and in exchange, IronSync gives them access to a set of ghost shards representing their protocol: a thread can deposit ghost objects from the set T into the protocol and withdraw them later. Crucially, the trusted IronSync API from §4.3 allows code that holds a guard shard to obtain a **`shared`** copy of the deposited value t, which allows the code to ergonomically and soundly manipulate read-shared data.

Formally, we define a Guard Protocol as follows.

Definition 4 (Guard Protocol) *Given a set T, a Guard Protocol is an LTS that has three transition types, $(M, \text{Init}, \tau^{internal}_{local}, \tau^{\text{deposit}(t)}_{local}, \tau^{\text{withdraw}(t)}_{local})$; an invariant* Inv $: M \to$ `bool`*; and an abstraction function* `Abs` $: M \to T \cup \{empty\}$*. We define $\tau^{internal}_{global}, \tau^{\text{deposit}(t)}_{global}, \tau^{\text{withdraw}(t)}_{global}$ of the GSM as in Def. 2.*

We say the Guard Protocol is sound *if* Inv *is an inductive invariant on the GSM, i.e., $\forall$ transition labels ℓ:*

$$\forall m \in M.\ \text{Init}(m) \implies \text{Inv}(m)$$
$$\forall m, m' \in M.\ \text{Inv}(m) \wedge \tau^{\ell}_{global}(m, m') \implies \text{Inv}(m')$$

and if the GSM, as interpreted by `Abs`*, refines the Safety-Deposit State Machine. Given a sound Guard Protocol, we say that $g \in M$ is a* read-guard *of $t \in T$ if,*

$$\forall b \in M.\ \text{Inv}(g \cdot b) \implies \texttt{Abs}(g \cdot b) = t.$$

Here, `Abs` gives the GSM a notion of a "deposited object." The read-guard condition says that in any valid global state (as given by Inv) with g as a sub-shard, t is guaranteed to be the deposited object (as given by `Abs`). This means that a thread holding the guard shard g can soundly read the shared value t, and that all such readers will read the same value.

When the IronSync user defines a new guard protocol and proves it sound, IronSync gives them access to an API to manipulate ghost shards according to the transitions, as with any other LTS. In this case, the functions that perform exchanges can also perform deposits and withdraws; furthermore, there are new functions for the read-guard objects: if g is a read-guard of t, then the user can use a **`shared`** ghost shard g to obtain a **`shared`** ghost shard t. Linear Dafny's type system ensures that the guard reference outlives t.

Major component	trusted LOC	impl LOC	proof LOC	verif time
Common Framework				
LTS def. & ghost axioms	487			15 s
Memory Primitives	310			6 s
Libraries		316	3825	75 s
Bank §3				
Spec	17			0.7 s
LTS			262	9 s
Impl		21	16	2 s
RHHT Hash Table §4.1				
Spec	57			2 s
LTS			687	54 s
Refinement Proofs			168	8 s
Impl		417	1390	68 s
Node Replication §5.1				
Spec	104			4 s
REPLICATION LTS			2329	384 s
FLATCOMBINE LTS			649	97 s
CYCLICBUFFER LTS			1756	182 s
DISTRWLOCK LTS			633	17 s
Refinement Proofs			1291	132 s
Impl		730	1170	80 s
SplinterCache §5.2				
Spec	185			4 s
Disk Model and API	586			14 s
CACHE LTS			1036	159 s
CACHERWLOCK LTS			2015	86 s
Refinement Proofs			2456	372 s
Impl		1579	3163	297 s
Total	1746	3063	22846	34.7 min

Figure 15: Across all case studies the proof:code ratio is 7.5.

In addition to the formulation above, IronSync provides a more advanced version that allows multiple objects to be stored at once. This is useful for NR's cyclic buffer §5.1, for example. Both of these formulations are proved correct, based on a set of low-level axioms for manipulating monoid-based ghost state with **`shared`** variables in Linear Dafny; those axioms in turn are based on a concurrent separation logic for temporary read-sharing called Burrow [18].

7 Evaluation

In our evaluation, we aim to answer the following questions:

- What is the verification effort for IronSync development, both by the developer and the computer verifier (§7.1)?
- Is IronSync suitable for verifying state-of-the-art systems without compromises (§7.2)?
- What does verification tell us about the original reference implementations (§7.3)?

7.1 Verification Effort

Verifying all four concurrent examples consumes under an hour of CPU (5 minutes real time) on an 8-core 64 GiB cloud machine. 88% of files verify in under a minute; the slowest takes less than five. The four examples comprise 2747 lines of non-ghost implementation, plus 316 of shared library.

Figure 15 shows detailed information for each case study. Within implementation files, the proof-to-code ratio is about 4:1, where the proof code includes both the manipulation of ghost shards and standard Dafny proof annotations, like preconditions and postconditions. The full system proofs augment the implementation files with LTS code and refinement proofs, raising the overall proof:code ratio to 7.5.

As two points of comparison, we consider GoJournal [12] and Armada [38] (see §9 for details). GoJournal [12] reports a 19:1 proof-to-code ratio for its shared-memory code, while Armada's largest example takes 4.9 hours of CPU time (about 40 min. of real time) to verify 70 lines of code. These are not direct apples-to-apples comparisons: Armada and GoJournal arguably prove more substantial theorems about machine semantics. Still, verification time and developer effort have historically limited the use of verification tools, and thus IronSync constitutes a major practical improvement.

7.2 Case Study Fidelity

We evaluate IronSync's expressiveness by porting our two production-level case studies, NR (§5.1) and SplinterCache (§5.2), to IronSync to confirm that IronSync does not require sacrificing performance-critical concurrency patterns. We refer to the case studies' existing publications [9, 14] (both within the last 5 years) to justify that they can reasonably be called "state-of-the-art." We evaluate how faithful our IronSync implementations are to the reference implementations both qualitatively and by comparing performance.

First, we report on intentional compromises we made while mimicking the reference code of NR and SplinterCache, from most significant to least. First, in some cases, Reference NR uses release-acquire atomics. IronSync does not support these, so we use sequentially-consistent atomics instead. Second, IronSync does not support callbacks, so we refactored code to avoid them, and we could not implement the secondary, callback-based APIs in SplinterCache or NR. Third, IronSync's SplinterCache adds a runtime check in one method whose correctness was otherwise dependent on properties of SplinterDB's allocator, which was out-of-scope.

As evidence that these artifacts otherwise meet a high degree of fidelity, we benchmark against their references, using methodology similar to the reference publications [9, 14]. Each case study has different hardware requirements.

NR. We evaluate NR's performance against other locks and to its reference Rust implementation from NrOS [6]. We wrap a single-threaded radix tree with IronSync-NR, Reference-NR, or a lock, including a verified DistRwLock (§5.1), an MCS lock [40], a shuffle lock [27], and the standard C++ shared mutex. The benchmark pre-populates the tree with 128M entries (using 8B keys and values) and executes *get* and *update* requests with a uniform key distribution while varying the update ratio and the number of threads.

Figure 16 shows the performance measured on a machine with 4 Xeon Gold 6252 CPUs with 24 cores per NUMA node,

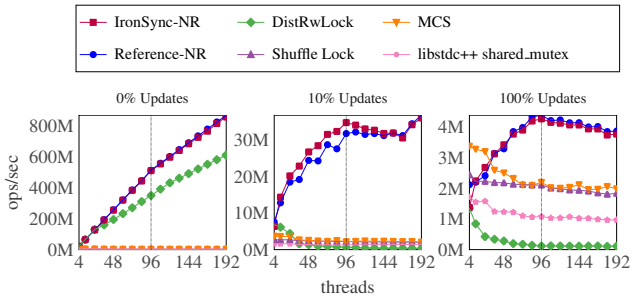

Figure 16: Comparing throughput scalability of IronSync-NR, Reference-NR, and locks. Higher is better.

totaling 96 cores and 192 hardware threads. The threads are pinned to fill up cores on a NUMA node first before moving to the next. NR adds one replica for each NUMA node, so at `x=96` threads, NR uses 4 replicas. Beyond 96 threads, no more replicas are added, and we begin hyperthread-sharing. IronSync-NR and Reference-NR perform similarly and generally outperform the rest, especially for read-heavy workloads.

For 0% updates, IronSync-NR, Reference-NR and the DistRwLock scale linearly, but the other mechanisms suffer under lock contention. NR performs better than DistRwLock due to perfect NUMA locality. With 10% updates, DistRwLock's performance drops to match the other locks, while IronSync-NR and Reference-NR benefit from flat combining. IronSync-NR outperforms Reference-NR slightly, though we do not yet know the cause.

Only at very high update rates (e.g., 100%) do MCS and shuffle locks outperform NR at low scale on one NUMA node; otherwise both NR implementations dominate. Hence, we conclude that IronSync-NR provides performance parity with Reference-NR and that it preserves NR's replication and flat combining benefits at all scales.

SplinterCache. We evaluate the performance of IronSync-SplinterCache against the reference implementation both with macrobenchmarks as part of SplinterDB using the YCSB benchmark [15], and with cache-specific microbenchmarks.

Results are from a Dell PowerEdge R630 with a 28-core 2.00 GHz Intel Xeon E5-2660 CPU, 192 GiB RAM and a 960GiB Intel Optane 905p PCI Express 3.0 NVMe device.

Macrobenchmarks. Our YCSB configuration largely follows prior work [14]. We perform the Load, and A-F standard workloads on SplinterDB using either IronSync- or Reference-SplinterCache. Each workload uses 24B keys, 100B values and 14 threads. Run E performs 14M operations and the others each perform 69M operations, so that each workload logically reads/writes roughly 80GiB of data.

We use three target memory sizes: 4 GiB to stress eviction and IO; 20 GiB to reflect a common system configuration; and 100 GiB to stress in-memory and concurrency. Figure 17 shows that SplinterDB with IronSync-SplinterCache is always within 9% of the reference performance.

Microbenchmarks. We first allocate pages and flush them

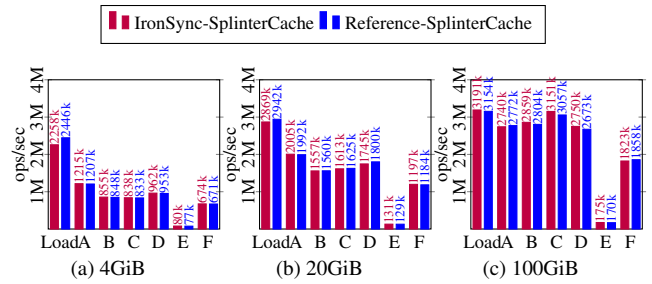

Figure 17: YCSB Benchmark. 69M ops/workload (E is 14M) with 14 threads. Y-axis is mean of 3 runs. Higher is better.

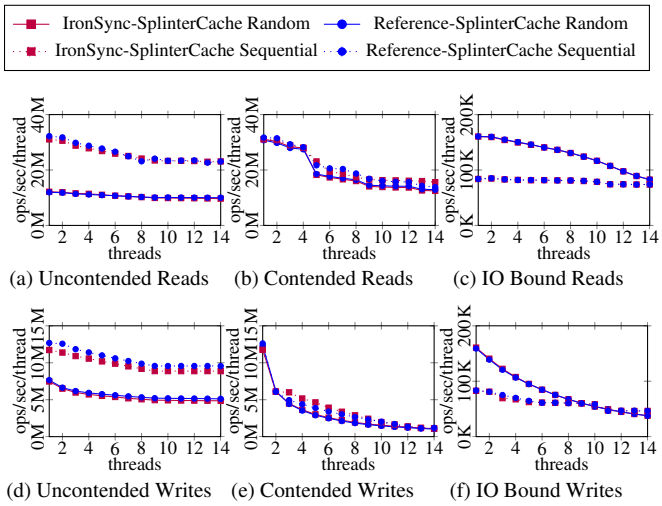

Figure 18: SplinterCache microbenchmark with a 4 GiB cache. Y-axis is mean throughput of 5 runs. Higher is better.

without evicting them. Then each thread performs a fixed number of operations, choosing pages either randomly or sequentially, then either acquiring a read lock or a write lock. We use three configurations in a 4 GiB cache: general "uncontended" in-memory, with 2 GiB of data, (Figures 18a and 18d), "contended" in-memory, with 128 KiB (32 pages) of data (Figures 18b and 18e), and "IO bound", with 8 GiB of data (Figures 18c and 18f). IronSync-SplinterCache is within 11% of the performance of reference on all microbenchmarks.

7.3 Bugs and Insights

We confirmed the 3 bugs below with the original developers.

NR. In the reference code, we identified a bug which could cause a read-read linearizability violation between two different nodes if they took place concurrently with an update. This bug could only occur if a thread dispatched log entries during garbage collection.

This bug surfaced when we realized our first attempt at defining REPLICATION would not be linearizable. We fixed the bug by always holding the lock appropriately, and the verified implementation now puts an extra ghost shard behind the lock to represents the lock's role in REPLICATION.

SplinterCache. We identified two bugs in the reference code. First was a data race on `disk_addr`, which maps cache entries to disk addresses. This race could occur when a read lock races with both eviction and a subsequent load.

Second, the code for batching write IO did not check that `disk_addr` was the expected value after locking a page for writeback. This could result in data written to the wrong location, among other corruptions. We identified these while porting the implementation, as we realized certain ghost shards would not be available following the reference logic.

8 Discussion on Modularity

Concurrent systems can seem dauntingly anti-modular when they entangle low-level synchronization with high-level application logic, making the tasks of ensuring data safety and logical correctness seem inseparably intertwined. By verifying two such case studies, IronSync shows how these two levels of concern can be disentangled *within the proof*.

In NR (§5.1), for example, the `localHead` variables play two distinct roles: **(i)** buffer entries cannot be garbage-collected past any node's `localHead` (relevant to CYCLICBUFFER), ensuring data safety, and **(ii)** `localHead` matches the version of the local replica state (relevant to REPLICATION), ensuring logical correctness. Figure 13 illustrates the overlap of these roles in two methods where the overlap is notably dense. Note that some operations might advance both state machines at the same time; however, this fact is not relevant to proofs associated with either either half.

Likewise in SplinterCache (§5.2), the `WriteBack` flag plays a role in both logical correctness (CACHE) and low-level data safety (CACHERWLOCK). The code ties these two distinct roles together by using the same flag bit: when a thread modifies the physical `WriteBack` flag, it advances *both* state machines (Figure 14), but again, this is an implementation detail to which both abstractions are agnostic.

In short, we modularize proofs of a sophisticated system by abstracting it in *multiple* ways. Difficult concurrent reasoning takes place on simplified abstractions, but IronSync ensures the abstractions compose soundly; thus proofs about the individual components say something meaningful about the whole. The implementation still ties the abstractions together with physical state, but this step is straightforward from a verification standpoint, thanks to the ownership type system and Dafny's automation. Ultimately, this method decouples the modularity structure of the *proof* from that of the *code*.

9 Related Work

Logics for concurrent programs reflect different trade-offs between generality, expressiveness, modularity, and usability. IronSync strikes a balance between very general state machines at high levels of abstraction, while at lower levels leaning on language features like Hoare logic and ownership types for usability. IronSync trusts these language features instead of proving theorems directly against operational semantics, unlike work like Armada [38] or Iris [24].

Concurrent separation logic (CSL) [44] lets threads take temporary ownership of state to perform isolated reasoning.

Propositional CSL is in general undecidable [8], so tools either require manual assistance from the user, as in Iris Proof Mode [31] where the user can manually match hypotheses to goals, use incomplete heuristics and user hints, as in Diaframe [42], or solve restricted fragments, as in Viper [43] and Steel [16]. Meanwhile, IronSync encodes CSL propositions using explicit ownership in the type system. Thus, ownership is directed manually by the user, but this method lets us additionally take direct advantage of automation from standard sequential reasoning tools such as SMT solvers and the weakest-precondition-style encoding used by Dafny.

Recent CSLs are extremely sophisticated. Iris [24] and Steel [16] employ monoids to extend CSL with flexible ownership protocols, used in recent systems like Perennial [11] and GoJournal [12], and Iris can handle future-dependent linearization points with prophecy variables [25]. However, the proof rules in these systems are intricate and may be intimidating to non-experts (e.g. Iris needs the "later modality" to allow impredicative invariants, which allow Iris to express some invariants beyond what IronSync can handle directly). In contrast, IronSync aims to make these concepts approachable by integrating invariants and monoids into its ownership type system, and connecting them with state-machine refinement. As a rough comparison, IronSync's case studies achieve a 7.7:1 proof-to-code ratio, while GoJournal [12] reports a 19:1 ratio for a comparably sized case study.

Like IronSync, Armada [38], IronFleet [20], and VeriBetrKV [19] all employ state-machine refinement. The latter two use Dafny's Hoare reasoning for the implementation of sequential code, whereas IronSync uses it for concurrent shared-memory code. In contrast, Armada verifies concurrent code using state machine refinement throughout the entire proof stack, foregoing Hoare-style reasoning in favor of detailed, low-level state machines. This provides more expressiveness; for example, two of Armada's case studies rely on racy memory accesses using memory ordering weaker than SC, which IronSync does not currently support. However, Armada's expressiveness also imposes costs; e.g., Armada's Pointers case study is 13 LoC and generates 6,997 lines of proof, while in IronSync the proof is trivial, since the correct usage of the owned pointers is automatically determined by type checking. Similarly, Armada's Owicki-Gries counter requires 130 lines of manual proof and generates 169,270 lines of Dafny proof to verify, while in IronSync it requires 230 lines of manual proof that verify directly in 8 seconds. We studied Armada's largest case study, a lock-free queue with 70 lines of code, and implemented an analogous queue in IronSync. Theirs requires 601 lines of proof (compared to 580 for IronSync) and 8 proof layers ($\sim$ 700 LoC), and takes 4.9 hours of CPU time to verify almost 200K lines of generated proof, versus 100 seconds of CPU time for IronSync.

Many other systems utilize ownership types. Cogent [3] and VeriBetrKV [19] use ownership types for systems verification, albeit with no shared-memory concurrency, and with the latter introducing Linear Dafny. CIVL [21] (based mainly on reduction [37]) uses ownership types, but primarily to handle thread identifiers, not general ghost state. Rust [28, 39] uses ownership types to enforce memory safety between threads [23] but lacks verification of deeper correctness properties. GhostCell [53] (an inspiration for our `Cell`) proposes owned "ghost tokens" in Rust to express ownership of groups of objects, though only for memory safety. Tools like Prusti [4] verify single-threaded Rust programs; IronSync can help extend them to multi-threaded Rust code.

Several approaches use Dafny-style automation for concurrent reasoning. Chalice [34] is a Dafny-like language with lock invariants but no tools for global reasoning. GoJournal [12] does integrate Dafny's sequential reasoning into a verified concurrent system, but it performs its shared-memory concurrency reasoning in Iris, so it does not leverage Dafny's automation for *concurrency* reasoning the way IronSync does.

CertiKOS [17] and SeKVM [36, 46] encapsulate concurrent operations inside modular interfaces, where programmers write proofs about the operations directly in Coq. We expect that IronSync-style ownership could simplify these proofs.

Prior work has verified concurrent hash tables, both bucketing [13] and linear-probing [18]. Prior work has also verified flat-combining [47] and producer-consumer queues [41, 48], but we are not aware of a verified cyclic buffer like NR's, which requires multiple consumers to read each entry.

10 Conclusion

IronSync offers scalable verification of concurrent shared-memory systems by factoring their complex proofs into separate concerns. It automates proofs of data safety and local logical correctness via a fast, deterministic ownership type system combined with powerful tools for *sequential* correctness. IronSync's LTS connects these local techniques to a simplified view of the entire system, where a developer can more easily reason about global properties. Our case studies demonstrate the success of this approach and show that we can tease apart application and synchronization logic for proof purposes, even when the implementation entangles them.

11 Acknowledgments

Work at CMU was supported, in part, by the Alfred P. Sloan Foundation, a Google Faculty Fellowship, a gift from VMware, a grant from the Intel Corporation, and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Andrea Lattuada was supported by a Google PhD Fellowship.

We thank Mihai Budiu, Manos Kapritsos, Jay Lorch, and Oded Padon, along with the anonymous reviewers and our shepherd, Eddie Kohler, for helpful feedback. We also thank Rob Johnson for discussions on SplinterCache.

A Artifact Appendix

Abstract

Our artifact contains everything needed to verify the IronSync projects with Linear Dafny and run the benchmark experiments.

Scope

Our artifact can be used to reproduce the results in Section 7, specifically:

- The table in Figure 15, with line count information and verification times.
- Other figures, such as proof-to-code ratio, mentioned in Section 7.1.
- Benchmark results in Section 7.2, specifically, the claim that for both NR and SplinterCache, the performance of the IronSync applications are comparable to their corresponding reference implementations.

Contents

The artifact contains:

- Linear Dafny source for the IronSync framework.
- Linear Dafny source for the case studies mentioned in the paper (the bank, the hash table, SplinterCache, and NR)
- Our modified version of Linear Dafny needed to run IronSync.
- The open-source reference implementation of NR.
- The open-source reference implementation of SplinterDB (which includes SplinterCache).
- A Docker container with all Dafny dependencies.
- A benchmark harness for each.

Hosting

The artifact is hosted at https://github.com/secure-foundations/ironsync-osdi2023, commit `d361111cbc87b5573d14975227de845e8a717ca5`. See the `README.md` file for instructions.

Requirements

The artifact requires x86 Ubuntu. (Note that, although our artifact includes a Docker container, which may be used on other platforms, the Docker container is only used for running Linear Dafny; it cannot be used for running the benchmarks.)

The ideal hardware for the NR benchmarks is a NUMA machine with 96 cores. The benchmarks in our paper were run on a machine with 4 Xeon Gold 6252 CPUs with 24 cores per NUMA node. However, a machine with fewer cores should still be able to reproduce our graphs up to a certain number of threads.

The ideal hardware for the cache benchmarks is a machine with a low-latency storage device, such as an Intel 905P Optane SSD. The machine also needs at least 100GiB to run the largest benchmark. The benchmarks in our paper were run on a Dell PowerEdge R630 with a 28-core 2.00 GHz Intel Xeon E5-2660 CPU, with 192 GiB RAM.

If the ideal hardware is not used, you will not be able to reproduce the exact performance characteristics of our paper, though we still expect to see that the IronSync implementations perform comparably to the reference implementations. Our artifact contains additional recommendations for selecting hardware.

References

[1] Alglave, J., Fox, A., Ishtiaq, S., Myreen, M. O., Sarkar, S., Sewell, P., and Nardelli, F. Z. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP)* (2009).

[2] Alglave, J., Maranget, L., Sarkar, S., and Sewell, P. Fences in weak memory models. In *Proceedings of Computer Aided Verification (CAV)* (2010).

[3] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[4] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. Leveraging Rust types for modular specification and verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (October 2019).

[5] Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. Mathematizing C++ concurrency. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2011).

[6] Bhardwaj, A., Kulkarni, C., Achermann, R., Calciu, I., Kashyap, S., Stutsman, R., Tai, A., and Zellweger, G. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021).

[7] Boehm, H.-J., and Adve, S. V. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2008).

[8] Brotherston, J., and Kanovich, M. Undecidability of propositional separation logic and its neighbours. *J. ACM 61*, 2 (Apr. 2014).

[9] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[10] CELIS, P. *Robin Hood Hashing*. PhD thesis, University of Waterloo, CAN, 1986.

[11] CHAJED, T., TASSAROTTI, J., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2019).

[12] CHAJED, T., TASSAROTTI, J., THENG, M., JUNG, R., KAASHOEK, M. F., AND ZELDOVICH, N. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2021).

[13] CLAUSEN, E. Verifying hash tables in Iris. Master's thesis, Aarhus University, 2017.

[14] CONWAY, A., GUPTA, A. K., CHIDAMBARAM, V., FARACH-COLTON, M., SPILLANE, R. P., TAI, A., AND JOHNSON, R. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2020).

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing* (2010).

[16] FROMHERZ, A., RASTOGI, A., SWAMY, N., GIBSON, S., MARTÍNEZ, G., MERIGOUX, D., AND RAMANANANDRO, T. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages 5*, ICFP (August 2021).

[17] GU, R., SHAO, Z., CHEN, H., WU, X., KIM, J., SJÖBERG, V., AND COSTANZO, D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (2016).

[18] HANCE, T., HOWELL, J., PADON, O., AND PARNO, B. Burrow: Custom read/write permissions for custom ghost state in concurrent separation logic. Tech. Rep. CMU-CyLab-21-002, Carnegie Mellon University, Cylab, Nov. 2021.

[19] HANCE, T., LATTUADA, A., HAWBLITZEL, C., HOWELL, J., JOHNSON, R., AND PARNO, B. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020).

[20] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2015).

[21] HAWBLITZEL, C., PETRANK, E., QADEER, S., AND TASIRAN, S. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of Computer Aided Verification (CAV)* (2015).

[22] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the synchronization-parallelism tradeoff. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2010).

[23] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (Jan. 2018).

[24] JUNG, R., KREBBERS, R., JOURDAN, J.-H., BIZJAK, A., BIRKEDAL, L., AND DREYER, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming 28* (2018).

[25] JUNG, R., LEPIGRE, R., PARTHASARATHY, G., RAPOPORT, M., TIMANY, A., DREYER, D., AND JACOBS, B. The future is ours: Prophecy variables in separation logic. *Proceedings of the ACM Programming Languages 4*, POPL (Jan. 2020).

[26] JUNG, R., SWASEY, D., SIECZKOWSKI, F., SVENDSEN, K., TURON, A., BIRKEDAL, L., AND DREYER, D. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).

[27] KASHYAP, S., CALCIU, I., CHENG, X., MIN, C., AND KIM, T. Scalable and practical locking with shuffling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2019).

[28] KLABNIK, S., AND NICHOLS, C. *The Rust Programming Language*. No Starch Press, USA, 2018.

[29] KLABNIK, S., NICHOLS, C., AND RUST COMMUNITY. The Rust Programming Language. https://doc.rust-lang.org/book/.

[30] KNUTH, D. E. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[31] KREBBERS, R., TIMANY, A., AND BIRKEDAL, L. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not. 52*, 1 (Jan. 2017), 205–217.

[32] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[33] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).

[34] LEINO, K. R. M., MÜLLER, P., AND SMANS, J. Verification of concurrent programs with Chalice. In *Proceedings of Foundations of Security Analysis and Design (FOSAD)* (2009).

[35] LI, J., LATTUADA, A., ZHOU, Y., CAMERON, J., HOWELL, J., PARNO, B., AND HAWBLITZEL, C. Linear types for large-scale systems verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (November 2022).

[36] LI, S.-W., LI, X., GU, R., NIEH, J., AND HUI, J. Z. A secure and formally verified Linux KVM hypervisor. In *Proceedings of the IEEE Symposium on Security and Privacy* (2021).

[37] LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM, 18*, 12 (1975).

[38] LORCH, J. R., CHEN, Y., KAPRITSOS, M., MA, H., PARNO, B., QADEER, S., SHARMA, U., WILCOX, J. R., AND ZHAO, X. Armada: Automated verification of concurrent code with sound semantic extensibility. *ACM Transactions on Programming Languages and Systems 44*, 2 (June 2022).

[39] MATSAKIS, N. D., AND KLOCK, F. S. The Rust language. *Ada Lett. 34*, 3 (Oct. 2014), 103–104.

[40] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb. 1991).

[41] MÉVEL, G., AND JOURDAN, J.-H. Formal verification of a concurrent bounded queue in a weak memory model. *Proceedings of the ACM on Programming Languages 5*, ICFP (Aug. 2021).

[42] MULDER, I., KREBBERS, R., AND GEUVERS, H. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2022).

[43] MÜLLER, P., SCHWERHOFF, M., AND SUMMERS, A. J. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)* (Berlin, Heidelberg, 2016).

[44] O'HEARN, P. W. Resources, concurrency, and local reasoning. *Theoretical Computer Science 375*, 1–3 (Apr. 2007).

[45] OWENS, S., SARKAR, S., AND SEWELL, P. A better x86 memory model: x86-TSO. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (Aug. 2009).

[46] TAO, R., YAO, J., LI, X., LI, S.-W., NIEH, J., AND GU, R. Formal verification of a multiprocessor hypervisor on Arm relaxed memory hardware. In *Symposium on Operating Systems Principles (SOSP)* (2021).

[47] TURON, A., DREYER, D., AND BIRKEDAL, L. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)* (2013).

[48] TURON, A., VAFEIADIS, V., AND DREYER, D. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)* (2014).

[49] VMWARE. What is VMware vSAN? https://www.vmware.com/products/vsan.html, 2021.

[50] VYUKOV, D. Distributed reader-writer mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2011.

[51] WADLER, P. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods* (1990).

[52] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA., A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Nov. 2013).

[53] YANOVSKI, J., DANG, H.-H., JUNG, R., AND DREYER, D. GhostCell: Separating permissions from

data in Rust. *Proceedings of the ACM on Programming Languages 5*, ICFP (Aug. 2021).