Armada: Low-Effort Verification of High-Performance Concurrent Programs

Jacob R. Lorch
Microsoft Research
USA
lorch@microsoft.com

Yixuan Chen
University of Michigan and
Yale University, USA
yixuan.chen@yale.edu

Manos Kapritsos
University of Michigan
USA
manosk@umich.edu

Bryan Parno
Carnegie Mellon University
USA
parno@cmu.edu

Shaz Qadeer
Calibra
USA
shaz@fb.com

Upamanyu Sharma
University of Michigan
USA
upamanyu@umich.edu

James R. Wilcox
Certora
USA
james@certora.com

Xueyuan Zhao
Carnegie Mellon University
USA
xueyuanz@alumni.cmu.edu

Abstract

Safely writing high-performance concurrent programs is notoriously difficult. To aid developers, we introduce Armada, a language and tool designed to formally verify such programs with relatively little effort. Via a C-like language and a small-step, state-machine-based semantics, Armada gives developers the flexibility to choose arbitrary memory layout and synchronization primitives so they are never constrained in their pursuit of performance. To reduce developer effort, Armada leverages SMT-powered automation and a library of powerful reasoning techniques, including rely-guarantee, TSO elimination, reduction, and alias analysis. All these techniques are proven sound, and Armada can be soundly extended with additional strategies over time. Using Armada, we verify four concurrent case studies and show that we can achieve performance equivalent to that of unverified code.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Concurrent programming languages**.

Keywords: refinement, weak memory models, x86-TSO

ACM Reference Format:
Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3385971

1 Introduction

Ever since processor speeds plateaued in the early 2000s, building high-performance systems has increasingly relied on concurrency. Writing concurrent programs, however, is notoriously error-prone, as programmers must consider all possible thread interleavings. If a bug manifests on only one such interleaving, it is extremely hard to detect using traditional testing techniques, let alone to reproduce and repair. Formal verification provides an alternative: a way to guarantee that the program is completely free of such bugs.

This paper presents Armada, a methodology, language, and tool that enable low-effort verification of high-performance, concurrent code. Armada's contribution rests on three pillars: *flexibility* for high performance; *automation* to reduce manual effort; and an expressive, low-level framework that allows for *sound semantic extensibility*. These three pillars let us achieve automated verification, with semantic extensibility, of concurrent C-like imperative code executed in a weak memory model (x86-TSO [35]).

Prior work (§7) has achieved some of these but not simultaneously. For example, Iris [25] supports powerful and sound semantic extensibility but focuses less on automation and C-like imperative code. Conversely, CIVL [19], for instance, supports automation and imperative code without sound extensibility; instead it relies on paper proofs when using techniques like reduction, and the CIVL team is continuously introducing new trusted tactics as they find more users and programs [36]. Recent work building larger verified concurrent systems [6, 7, 17] supports sound extensibility but sacrifices flexibility, and thus some potential for performance optimization, to reduce the burden of proof writing.

In contrast, Armada achieves all three properties, which we now expand and discuss in greater detail:

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06...$15.00
https://doi.org/10.1145/3385412.3385971

Flexibility To support high-performance code, Armada lets developers choose any memory layout and any synchronization primitives they need for high performance. Fixing on any one strategy for concurrency or memory management will inevitably rule out clever optimizations that developers come up with in practice. Hence, Armada uses a common low-level semantic framework that allows arbitrary flexibility, akin to the flexibility provided by a C-like language; e.g., it supports pointers to fields of objects and to elements of arrays, lock-free data structures, optimistic racy reads, and cache-friendly memory layouts. We enable such flexibility by using a small-step state-machine semantics rather than one that preserves structured program syntax but limits a priori the set of programs that can be verified.

Automation However, actually writing programs as state machines is unpleasantly tedious. Hence, Armada introduces a higher-level syntax that lets developers write imperative programs that are automatically translated into state-machine semantics. To prove these programs correct, the developer then writes a series of increasingly simplified programs and proves that each is a sound abstraction of the previous program, eventually arriving at a simple, high-level specification for the system. To create these proofs, the Armada developer simply annotates each level with the proof strategy necessary to support the refinement proof connecting it to the previous level. Armada then analyzes both levels and automatically generates a lemma demonstrating that refinement holds. Typically, this lemma uses one of the libraries we have developed to support eight common concurrent-systems reasoning patterns (e.g., logical reasoning about memory regions, rely-guarantee, TSO elimination, and reduction). These lemmas are then verified by an SMT-powered theorem prover. Explicitly manifesting Armada's lemmas lets developers perform *lemma customization*, i.e., augmentations to lemmas in the rare cases where the automatically generated lemmas are insufficient.

Sound semantic extensibility Each of Armada's proof-strategy libraries, and each proof generated by our tool, is mechanically proven to be correct. Insisting on verifying these proofs gives us the confidence to extend Armada with arbitrary reasoning principles, including newly proposed approaches, without worrying that in the process we may undermine the soundness of our system. Note that inventing new reasoning principles is an explicit non-goal for Armada; instead we expect Armada's flexible design to support new reasoning principles as they arise.

Our current implementation of Armada uses Dafny [27] as a general-purpose theorem prover. Dafny's SMT-based [11] automated reasoning simplifies development of our proof libraries and developers' lemma customizations, but Armada's broad structure and approach are compatible with any general-purpose theorem prover. We extend Dafny with a backend that produces C code that is compatible with ClightTSO [41], which can then be compiled to an executable by CompCert-TSO in a way that preserves Armada's guarantees.

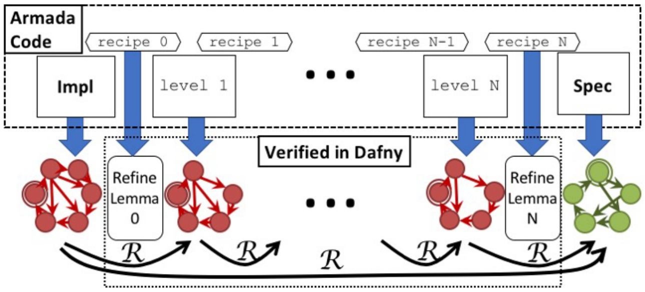

Figure 1. Armada Overview The Armada developer writes a low-level implementation in Armada designed for performance. They then define a series of levels, each of which abstracts the program at the previous level, eventually reaching a small, simple specification. Each refinement is justified by a simple refinement recipe specifying which refinement strategy to use. As shown via blue arrows, Armada automatically translates each program into a state machine and generates refinement proofs demonstrating that the refinement relation $\mathcal{R}$ holds between each pair of levels. Finally, it uses transitivity to show that $\mathcal{R}$ holds between the implementation and the spec.

We evaluate Armada on four case studies and show that it handles complex heap and concurrency reasoning with relatively little developer-supplied proof annotation. We also show that Armada programs can achieve performance comparable to that of unverified code.

In summary, this paper makes the following contributions.

- A flexible language for developing high-performance, verified, concurrent systems code.
- A mechanically-verified, extensible semantic framework that already supports a collection of eight verified libraries for performing refinement-based proofs, including region-based pointer reasoning, rely-guarantee, TSO elimination, and reduction.
- A practical tool that uses the above techniques to enable reasoning about complex concurrent programs with modest developer effort.

2 Overview

As shown in Figure 1, to use Armada, a developer writes an implementation program in the Armada language. They also write an imperative specification, which need not be performant or executable, in that language. This specification should be easy to read and understand so that others can determine (e.g., through inspection) whether it meets their expectations. Given these two programs, Armada's goal is to prove that all finite behaviors of the implementation simulate the specification, i.e., that the implementation *refines* the specification. The developer defines what this means via a *refinement relation* ($\mathcal{R}$). For instance, if the state contains a console log, the refinement relation might be that the log in the implementation is a prefix of that in the spec.

Because of the large semantic gap between the implementation and specification, we do not attempt to directly prove refinement. Instead, the developer writes a series of N Armada programs to bridge the gap between the implementation (level 0) and the specification (level $N+1$). Each pair of adjacent levels $i, i+1$ in this series should be similar enough to facilitate automatic generation of a refinement proof that respects $\mathcal{R}$; the developer supplies a short proof *recipe* that gives Armada enough information to automatically generate such a proof. Given the pairwise proofs, Armada leverages refinement transitivity to prove that the implementation indeed refines the specification.

We formally express refinement properties and their proofs in the Dafny language [27]. To formally describe what refinement means, Armada translates each program into its small-step state-machine semantics, expressed in Dafny. For instance, we represent the state of a program as a Dafny datatype and the set of its legal transitions as a Dafny predicate over pairs of states. To formally prove refinement between a pair of levels, we generate a Dafny lemma whose conclusion indicates a refinement relation between their state machines. We use Dafny to verify all proof material we generate, so ultimately the only aspect of Armada we must trust is its translation of the implementation and specification into state machines.

2.1 Example Specification and Implementation

To introduce Armada, we describe its use on an example program that searches for a good, but not necessarily optimal, solution to an instance of the traveling salesman problem.

The specification, shown in Figure 2, demands that the implementation output a valid solution, and it implicitly requires the program not to crash. Armada specifications can use powerful declarations as statements. Here, the `somehow` statement expresses that somehow the program updates `s` so that `valid_soln(s)` holds.

The example implementation, also shown in Figure 2, creates 100 threads, and each thread searches through 10,000 random solutions. If a thread finds a solution shorter than the best length found so far, it updates the global variables storing the best length and solution. The main routine joins the threads and prints the best solution found.

Note that this example has a benign race: the access to the shared variable `best_len` in the first `if (len < best_len)`. It is benign because the worst consequence of reading a stale value is that the thread unnecessarily acquires the mutex.

```
level Specification {
  void main() {
    var s:Solution;
    somehow modifies s ensures valid_soln(s);
    output_solution(s);
  }
}

level Implementation {
  // Global variables
  var best_solution:Solution;
  var best_len:uint32 := 0xFFFFFFFF;
  var mutex:Mutex;

  void worker() { // Thread to search for good solution
    var i:int32 := 0, s:Solution, len:uint32;
    while i < 10000 {
      choose_random_solution(&s);
      len = get_solution_length(&s);
      if (len < best_len) {
        lock(&mutex);
        if (len < best_len) {
          best_len := len;
          copy_solution(&best_solution, &s);
        }
        unlock(&mutex);
      }
      i := i + 1;
    }
  }

  void main() { // Main routine run at start
    var i:int32 := 0;
    var a:uint64[100];
    initialize_mutex(&mutex);
    while i < 100 {
      a[i] := create_thread worker();
      i := i + 1;
    }
    i := 0;
    while i < 100 {
      join a[i];
      i := i + 1;
    }
    print_solution(&best_solution);
  }
}
```

Figure 2. The Armada spec and implementation for our running example, which searches for a not-necessarily-optimal solution to a traveling salesman problem

```
level ArbitraryGuard {
  ...
  len = get_solution_length(&s);
  if (*) {   // arbitrary choice as guard
    lock(&mutex);
    if (len < best_len) {
      best_len := len;
      copy_solution(&best_solution, &s);
    }
    unlock(&mutex);
  }
  ...
}
```

Figure 3. Version of our example program in which the first guard condition is relaxed to an arbitrary choice

2.2 Example Proof Strategy

Figure 3 depicts the program, called `ArbitraryGuard`, at level 1 in our example proof. This program is like the implementation except that it *arbitrarily* chooses whether to acquire the lock, by using `*` in place of the guard condition `len < best_len`.

Our transformation of the `Implementation` program to the `ArbitraryGuard` program is an example of *weakening*, where a statement is replaced by one whose behaviors are a

```
proof ImplementationRefinesArbitraryGuard {
  refinement Implementation ArbitraryGuard
  weakening
}
```

Figure 4. In this recipe for a refinement proof, the first line indicates what should be proved (that the `Implementation`-level program refines the `ArbitraryGuard`-level program) and the second line indicates which strategy (in this case, weakening) generates the proof.

```
level BestLenSequential {
  ...
  if (len < best_len) {
    best_len ::= len; // immediately visible to all threads
    copy_solution(&best_solution, &s);
  }
  ...
}
```

Figure 5. Version of the example program where the assignment to `best_len` is now sequentially consistent

superset of the original. Or, more precisely, a state-transition relation is replaced by a superset of that relation. The two levels' programs thus exhibit *weakening correspondence*, i.e., it is possible to map each low-level program step to an equivalent or weaker one in the high-level program. The proof that `Implementation` refines `ArbitraryGuard` is straightforward but tedious to write, so instead the developer simply writes a recipe for this proof, shown in Figure 4. This recipe instructs Armada to generate a refinement proof using the weakening correspondence between the program pair.

Having removed the racy read of `best_len`, we can now demonstrate an ownership invariant: that threads only access that variable while they hold the mutex, and no two threads ever hold the mutex. This allows a further transformation of the program to the one shown in Figure 5. This replaces the assignment `best_len := len` with `best_len ::= len`, signifying the use of sequentially consistent memory semantics for the update rather than x86-TSO semantics [35]. Since strong consistency is easier to reason about than weak-memory semantics, proofs for further levels will be easier.

Just as for weakening, Armada generates a proof of refinement between programs whose only transformation is a replacement of assignments to a variable with sequentially-consistent assignments. For such a proof, the developer's recipe supplies the variable name and the ownership predicate, as shown in Figure 6.

If the developer mistakenly requests a TSO-elimination proof for a pair of levels that do not permit it (e.g., if the first level still has the racy read and thus does not own the location when it accesses it), then Armada will either generate an error message indicating the problem or generate an invalid proof. In the latter case, running the proof through the theorem prover (i.e., Dafny verifier) will produce an error message. For instance, it might indicate which statement may access the variable without satisfying the ownership predicate or which statement might cause two threads to simultaneously satisfy the ownership predicate.

```
proof ArbitraryGuardRefinesBestLenSequential {
  refinement ArbitraryGuard BestLenSequential
  tso_elim best_len "s.s.globals.mutex.holder == $me"
}
```

Figure 6. This recipe proves that the `ArbitraryGuard`-level program refines the `BestLenSequential`-level program. It uses TSO elimination based on strategy-specific parameters; in this case, the first parameter (`best_len`) indicates which location's updates differ between levels and the second parameter is an ownership predicate.

3 Semantics and Language Design

Armada is committed to allowing developers to adopt any memory layout and synchronization primitives needed for high performance. This affects the design of the Armada language and our choice of semantics.

The Armada language (§3.1) allows the developer to write their specification, code, and proofs in terms of programs, and the core language exposes low-level primitives (e.g., fixed-width integers or specific hardware-based atomic instructions) so that the developer is not locked into a particular abstraction and can reason about the performance of their code without an elaborate mental model of what the compiler might do. This also simplifies the Armada compiler.

To facilitate simpler, cleaner specifications and proofs, Armada also includes high-level and abstract features that are not compilable. For example, Armada supports mathematical integers, and it allows arbitrary sequences of instructions to be performed atomically (given suitable proofs).

The semantics of an Armada program (§3.2), however, are expressed in terms of a small-step state machine, which provides a "lowest common denominator" for reasoning via a rich and diverse set of proof strategies (§4). It also avoids baking in assumptions that facilitate one particular strategy but preclude others.

3.1 The Armada Language

As shown in Figure 1, developers express implementations, proof steps, and specifications all as programs in the Armada language. This provides a natural way of describing refinement: an implementation refines a specification if all of its externally-visible behaviors simulate behaviors of the specification. The developer helps prove refinement by bridging the gap between implementation and specification via intermediate-level programs.

We restrict the implementation level to the core Armada features (§3.1.1), which can be compiled directly to corresponding low-level C code. The compiler will reject programs outside this core. Programs at all other levels, including the specification, can use the entirety of Armada (§3.1.2), summarized in Figure 7. Developers connect these levels together

```
                                 Types
T ::=   uint8 | uint16 | uint32 | uint64
    |   int8 | int16 | int32 | int64                 (primitive types)
    |   ptr<T>                                              (pointers)
    |   T[N]                                                  (arrays)
    |   struct {var ⟨field⟩:T; ...}                          (structs)
    |   int | (T,...,T) | T → T                  (mathematical types)
    |   x : T "|" e                                      (subset types)
    |   ...
                               Expressions
e ::=   ⟨literal⟩ | ⟨variable⟩
    |   ⟨uop⟩ e | e1 ⟨bop⟩ e2                (unary/binary operators)
    |   &e | *e | null                         (pointer manipulation)
    |   e.⟨field⟩                               (struct manipulation)
    |   e1[e2]                                              (indexing)
    |   *                                    (non-deterministic value)
    |   old(e)                    (old value of e in two-state predicate)
    |   allocated(e) | allocated_array(e)                   (validity)
    |   $me | $sb_empty                               (meta variables)
    |   ...
                               Statements
⟨LHS⟩ ::=  ⟨variable⟩ | *e | e.⟨field⟩ | e[e]
⟨RHS⟩ ::=  e | ⟨method⟩(e, ...)
        |  malloc(T) | calloc(T, e)                       (allocation)
        |  create_thread ⟨method⟩(e, ...)                    (threads)
⟨spec⟩ ::= | requires e | modifies e | ensures e
    S ::=  var ⟨variable⟩:T [:= ⟨RHS⟩];
        |  ⟨LHS⟩, ... := ⟨RHS⟩, ...;                      (assignment)
        |  ⟨LHS⟩, ... ::= ⟨RHS⟩, ...;
        |                                   (TSO-bypassing assignment)
        |  if e S1 else S2 | while e1 [invariant e2] S
        |  break; | continue; | assert e; | S1 S2
        |  dealloc e; | join e; | label ⟨label⟩: S
        |  somehow ⟨spec⟩*;                  (declarative atomic action)
        |  explicit_yield {S} | yield;                      (atomicity)
        |  assume e; S                           (enablement condition)
```

Figure 7. Armada language syntax

using a refinement relation (§3.1.3). To let Armada programs use external libraries and special hardware features, we also support developer-defined external methods (§3.1.4).

3.1.1 Core Armada. The core of Armada supports features commonly used in high-performance C implementations. It has as primitive types signed and unsigned integers of 8, 16, 32, and 64 bits, and pointers. It supports arbitrary nesting of `struct`s and single-dimensional arrays, including `struct`s of arrays and arrays of `struct`s. It lets pointers point not only to whole objects but also to fields of `struct`s and elements of arrays. It does not yet support `union`s.

For control flow, it supports method calls, `return`, `if`, and `while`, along with `break` and `continue`. It does *not* support arbitrary control flow, e.g., `goto`.

It supports allocation of objects (`malloc`) and arrays of objects (`calloc`), and freeing them (`dealloc`). It supports creating threads (`create_thread`) and waiting for their completion (`join`).

Each statement may have at most one shared-location access, since the hardware does not support atomic performance of multiple shared-location accesses.

3.1.2 Proof and Specification Support. The full Armada language offers rich expressivity to allow natural descriptions of specifications. Furthermore, all program levels between the implementation and specification are abstract constructs that exist solely to facilitate the proof, so they too use this full expressivity. Below, we briefly describe interesting features of the language.

Atomic blocks are modeled as executing to completion without interruption by other threads. The semantics of an atomic block prevents thread interruption but not termination; a behavior may terminate in the middle of an atomic block. This allows us to prove that a block of statements can be treated as atomic without proving that no statement in the block exhibits undefined behavior (see §3.2.3).

Following CIVL [19], we permit some program counters within otherwise-atomic blocks to be marked as *yield points*. Hence, the semantics of an `explicit_yield` block is that a thread t within such a block cannot be interrupted by another thread unless t's program counter is at a yield point (marked by a `yield` statement). This permits modeling atomic sequences that span loop iterations without having to treat the entire loop as atomic. §4.2.1 shows the utility of such sequences, and Flanagan et al. describe further uses in proofs of atomicity via purity [15].

Enablement conditions can be attached to a statement, which cannot execute unless all its conditions are met.

TSO-bypassing assignment statements perform an update with sequentially-consistent semantics. Normal assignments (using `:=`) follow x86-TSO semantics (§3.2.1), but assignments using `::=` are immediately visible to other threads.

Somehow statements allow the declarative expression of arbitrary atomic-step specifications. A `somehow` statement can have `requires` clauses (preconditions), `modifies` clauses (framing), and `ensures` clauses (postconditions). The semantics of a `somehow` statement is that it has undefined behavior if any of its preconditions are violated, and that it modifies the lvalues in its framing clauses arbitrarily, subject to the constraint that each two-state postcondition predicate holds between the old and new states.

Ghost variables represent state that is not part of the machine state and has sequentially-consistent semantics. Ghost variables can be of any type supported by the theorem prover, not just those that can be compiled to C. Ghost types supported by Armada include mathematical integers; datatypes; sequences; and finite and infinite sets, multisets, and maps.

Assert statements crash the program if their predicates do not hold.

3.1.3 Refinement Relations. Armada aims to prove that the implementation *refines* the specification. The developer defines, via a *refinement relation* $\mathcal{R}$, what refinement means.

```
var snapshot;
if (!precondition_satisfied()) {
  ManifestUndefinedBehavior();
}
havoc_write_set();
snapshot := read_read_set();
while (* || !post_condition_satisfied()) {
  if (snapshot != read_read_set()) {
    ManifestUndefinedBehavior();
  }
  havoc_write_set();
  snapshot := read_read_set();
}
```

Figure 8. Default model for external methods, where the read set is the list of locations in `reads` clauses and the write set is the list of locations in `modifies` clauses

Formally, $\mathcal{R} \subseteq S_0 \times S_{N+1}$, where S_i is the set of states of the level-i program, level 0 is the implementation, and level $N+1$ is the spec. A pair $\langle s_0, s_{N+1} \rangle$ is in $\mathcal{R}$ if s_0 is acceptably equivalent to s_{N+1}. An implementation refines the specification if every finite behavior of the implementation may, with the addition of stuttering steps, simulate a finite behavior of the specification where corresponding state pairs are in $\mathcal{R}$.

The developer writes $\mathcal{R}$ as an expression parameterized over the low-level and high-level states. Hence, we can also use $\mathcal{R}$ to define what refinement means between programs at consecutive levels in the overall refinement proof, i.e., to define $\mathcal{R}_{i,i+1}$ for arbitrary level i. To allow composition into an overall proof, $\mathcal{R}$ must be transitive: $\forall i, s_i, s_{i+1}, s_{i+2} \;.\; \langle s_i, s_{i+1} \rangle \in \mathcal{R}_{i,i+1} \land \langle s_{i+1}, s_{i+2} \rangle \in \mathcal{R}_{i+1,i+2} \implies \langle s_i, s_{i+2} \rangle \in \mathcal{R}_{i,i+2}$.

3.1.4 External Methods. Since we do not expect Armada programs to run in a vacuum, Armada supports declaring and calling *external methods*. An external method models a runtime, library, or operating-system function; or a hardware instruction the compiler supports, like compare-and-swap. For example, the developer could model a runtime-supplied print routine via:

```
method {:extern} PrintInteger(n:uint32) {
  somehow modifies log ensures log == old(log) + [n];
}
```

In a sequential program, we could model an external call via a straightforward Hoare-style signature. However, in a concurrent setting, this could be unsound if, for example, the external library were not thread-safe. Hence, we allow the Armada developer to supply a more detailed, concurrency-aware model of the external call as a "body" for the method. This model is not, of course, compiled, but it dictates the effects of the external call on Armada's underlying state-machine model.

If the developer does not supply a model for an external method, we model it via the Armada code snippet in Figure 8. That is, we model the method as making arbitrary and repeated changes to its write set (as specified in a `modifies` clause); as having undefined behavior if a concurrent thread ever changes its read set (as specified in a `reads` clause); and as returning when its postcondition is satisfied, but not necessarily as soon as it is satisfied.

3.2 Small-Step State-Machine Semantics

To create a soundly extensible semantic framework, Armada translates an Armada program into a state machine that models its small-step semantics. We represent the state of a program as a Dafny datatype that contains the set of threads, the heap, static variables, ghost state, and whether and how the program terminated. Thread state includes the program counter, the stack, and the x86-TSO store buffer (§3.2.1). We represent steps of the state machine (i.e., the set of legal transitions) as a Dafny predicate over pairs of states. Examples of steps include assignment, method calls and returns, and evaluating the guard of an `if` or `while`.

The semantics are generally straightforward; the main source of complexity is the encoding of the x86-TSO model (§3.2.1). Hence, we highlight three interesting elements of our semantics: they are program-specific (§3.2.2), they model undefined behavior as a terminating state (§3.2.3), and they model the heap as immutable (§3.2.4).

3.2.1 x86 Total-Store Order (TSO). We model memory using x86-TSO semantics [35]. Specifically, a thread's write is not immediately visible to other threads, but rather enters a *store buffer*, a first-in-first-out (FIFO) queue. A write becomes globally visible when the processor asynchronously drains it from a store buffer.

To model this, our state includes a store buffer for each thread and a global memory. A thread's local view of memory is what would result from applying its store buffer, in FIFO order, to the global memory.

3.2.2 Program-Specific Semantics. To aid in automated verification of state-machine properties, we tailor each state machine to the program rather than make it generic to all programs. Such specificity ensures the verification condition for a specific step relation includes only facts about that step.

Specificity also aids reasoning by case analysis by restricting the space of program counters, heap types, and step types. Specifically, the program-counter type is an enumerated type that only includes PC values in the program. The state's heap only allows built-in types and user-defined `struct` types that appear in the program text. The global state and each method's stack frame is a datatype with fields named after program variables that never have their address taken.

Furthermore, the state-machine step (transition) type is an enumerated type that includes only the specific steps in the program. Each step type has a function that describes its specific semantics. For instance, there is no generic function for executing an update statement; instead, for each update statement there is a program-specific step function with the specific lvalue and rvalue from the statement.

The result is semantics that are SMT-friendly; i.e., Dafny automatically discharges many proofs with little or no help.

3.2.3 Undefined Behavior as Termination.

Our semantics has three terminating states. These occur when the program exits normally, when asserting a false predicate, and when invoking undefined behavior. The latter means executing a statement under conditions we do not model, e.g., an access to a freed pointer or a division by zero. Our decision to model undefined behavior as termination follows CIVL [19] and simplifies our specifications by removing a great deal of non-determinism. It also simplifies reasoning about behaviors, e.g., by letting developers state invariants that do not necessarily hold after such an undefined action occurs. However, this decision means that, as in CIVL, our refinement proofs are meaningless if (1) the spec ever exhibits undefined behavior, or (2) the refinement relation $\mathcal{R}$ allows the low-level program to exhibit undefined behavior when the high-level program does not. We prevent (2) by adding to the developer-specified $\mathcal{R}$ the conjunct "if the low-level program exhibits undefined behavior, then the high-level program does". Preventing condition (1) currently relies on the careful attention of the specification writer (or reader).

3.2.4 Immutable Heap Structure.

To permit pointers to fields of `structs` and to array elements, we model the heap as a forest of pointable-to objects. The roots of the forest are (1) allocated objects and (2) global and local variables whose addresses are taken in the program text. An array object has its elements as children and a `struct` object has its fields as children. To simplify reasoning, we model the heap as unchanging throughout the program's lifetime; i.e., allocation is modeled not as creating an object but as finding an object and marking its pointers as valid; freeing an object marks all its pointers as freed.

To make this sound, we restrict allowable operations to ones whose compiled behaviors lie within our model. Some operations, like dereferencing a pointer to freed memory or comparing another pointer to such a pointer, trigger undefined behavior. We disallow all other operations whose behavior could diverge from our model. For instance, we disallow programs that cast pointers to other types or that perform mathematical operations on pointers.

Due to their common use in C array idioms, we do permit comparison between pointers to elements of the same array, and adding to (or subtracting from) a pointer to an array element. That is, we model pointer comparison and offsetting but treat them as having undefined behavior if they stray outside the bounds of a single array.

4 Refinement Framework

Armada's goals rely on our extensible framework for automatic generation of refinement proofs. The framework consists of:

Strategies A *strategy* is a proof generator designed for a particular type of correspondence between a low-level and a high-level program. An example correspondence is *weakening*; two programs exhibit it if they match except for statements where the high-level version admits a superset of behaviors of the low-level version.

Library Our *library* of generic lemmas are useful in proving refinements between programs. Often, they are specific to a certain correspondence.

Recipes The developer generates a refinement proof between two program levels by specifying a *recipe*. A recipe specifies which strategy should generate the proof, and the names of the two program levels. Figure 4 shows an example.

Verification experts can extend the framework with new strategies and library lemmas. Developers can leverage these new strategies via recipes. Armada ensures sound extensibility because for a proof to be considered valid, all its lemmas and all the lemmas in the library must be verified by Dafny. Hence, arbitrarily complex extensions can be accommodated. For instance, we need not worry about unsoundness or incorrect implementation of the Cohen-Lamport reduction logic we use in §4.2.1 or the rely-guarantee logic we use in §4.2.2.

4.1 Aspects Common to All Strategies

Each strategy can leverage a set of Armada tools. For instance, we provide machinery to prove developer-supplied inductive invariants are inductive and to produce a refinement function that maps low-level states to high-level states.

The most important generic proof technique we provide is *non-determinism encapsulation*. State-transition relations are non-deterministic because some program statements are non-deterministic; e.g., a method call will set uninitialized stack variables to arbitrary values. Reasoning about such general relations is challenging, so we encapsulate all non-deterministic parameters in each step and manifest them in a *step object*. For instance, if a method M has uninitialized stack variable x, then each step object corresponding to a call to M has a field `newframe_x` that stores x's initial value. The proof can then reason about the low-level program using an *annotated behavior*, which consists of a sequence of states, a sequence of step objects, and, importantly, a function NextState that deterministically computes state $i + 1$ from state i and step object i. This way, the relationship between pairs of adjacent states is no longer a non-deterministic relation but a deterministic function, making reasoning easier.

4.1.1 Regions.

To simplify proofs about pointers, we use *region-based* reasoning, where memory locations (i.e., addresses) are assigned abstract *region ids*. Proving that two pointers are in different regions shows they are not aliased.

We carefully design our region reasoning to be automation-friendly and compatible with any Armada strategy. To assign regions to memory locations, rather than rely on developer-supplied annotations, we use Steensgaard's algorithm [40].

Our implementation of Steensgaard's algorithm begins by assigning distinct regions to all memory locations, then merges the regions of any two variables assigned to each other.

We perform region reasoning purely in Armada-generated proofs, without requiring changes to the program or the state machine semantics. Hence, in the future, we can add more complex alias analysis as needed.

To employ region-based reasoning, the developer simply adds `use_regions` to a recipe. Armada then performs the static analysis described above, generates the pointer invariants, and generates lemmas to inductively prove the invariants. If regions are overkill and the proof only requires an invariant that all addresses of in-scope variables are valid and distinct, the developer instead adds `use_address_invariant`.

4.1.2 Lemma Customization. Occasionally, verification fails for programs that correspond properly, because an automatically-generated lemma has insufficient annotation to guide Dafny. For instance, the developer may weaken `y := x & 1` to `y := x % 2`, which is valid but requires bit-vector reasoning. Thus, Armada lets the developer arbitrarily supplement an automatically-generated lemma with additional developer-supplied lemmas (or lemma invocations).

Armada's lemma customization contrasts with static checkers such as CIVL [19]. The constraints on program correspondence imposed by a static checker must be restrictive enough to ensure soundness. If they are more restrictive than necessary, a developer cannot appeal to more complex reasoning to convince the checker to accept the correspondence.

4.2 Specific Strategies

Our current implementation has eight strategies for eight different correspondence types. We now describe them.

4.2.1 Reduction. Because of the complexity of reasoning about all possible interleavings of statements in a concurrent program, a powerful simplification is to replace a sequence of statements with an atomic block. A classic technique for achieving this is reduction [30], which shows that one program refines another if the low-level program has a sequence of statements $R_1, R_2, \cdots, R_n, N, L_1, L_2, \ldots, L_m$ while the high-level program replaces those statements with a single atomic action having the same effect. Each R_i (L_i) must be a right (left) mover, i.e., a statement that commutes to the right (left) with any step of another thread.

An overly simplistic approach is to consider two programs to exhibit the reduction correspondence if they are equivalent except for a sequence of statements in the low-level program that corresponds to an atomic block with those statements as its body in the high-level program. This formulation would prevent us from considering cases where the atomic blocks span loop iterations (e.g., Figure 9).

```
Low level                        High level
                                 explicit_yield {
lock(&mutex);                      lock(&mutex);
while (condition()) {              while (condition()) {
  do_something();                    do_something();
  unlock(&mutex);                    unlock(&mutex);
                                     yield;
  lock(&mutex);                      lock(&mutex);
}                                  }
unlock(&mutex);                    unlock(&mutex);
                                 }
```

Figure 9. Reduction requiring the use of Cohen-Lamport generalization because the atomic block spans loop iterations

Instead, Armada's approach to sound extensibility gives us the confidence to use a generalization of reduction, due to Cohen and Lamport [9], that allows steps that do not necessarily correspond to consecutive statements in the program. It divides the states of the low-level program into a first phase (states following a right mover), a second phase (states preceding a left mover), and no phase (all other states). Programs may never pass directly from the second phase to the first phase, and for every sequence of steps starting and ending in no phase, there must be a step in the high-level program with the same aggregate effect.

Hence our strategy considers two programs to exhibit the reduction correspondence if they are identical except that some yield points in the low-level program are not yield points in the high-level program. The strategy produces lemmas demonstrating that each Cohen-Lamport restriction is satisfied; e.g., one lemma establishes that each step ending in the first phase commutes to the right with each other step. This requires generating many lemmas, one for each pair of steps of the low-level program where the first step in that pair is a right mover.

Our use of encapsulated nondeterminism (§4.1) greatly aids the automatic generation of certain reduction lemmas. Specifically, we use it in each lemma showing that a mover commutes across another step, as follows. Suppose we want to prove commutativity between a step σ_i by thread i that goes from s_1 to s_2 and a step σ_j from thread j that goes from s_2 to s_3. We must show that there exists an alternate-universe state s_2' such that a step from thread j can take us from s_1 to s_2' and a step from thread i can take us from s_2' to s_3. To demonstrate the existence of such an s_2', we must be able to automatically generate a proof that constructs such an s_2'. Fortunately, our representation of a step encapsulates all non-determinism, so it is straightforward to describe such an s_2' as $\mathsf{NextState}(s_1, \sigma_j)$. This simplifies proof generation significantly, as we do not need code that can construct alternative-universe intermediate states for arbitrary commutations. All we must do is emit lemmas hypothesizing that $\mathsf{NextState}(\mathsf{NextState}(s_1, \sigma_j), \sigma_i) = s_3$, with one lemma for each pair of step types. The automated theorem prover can typically dispatch these lemmas automatically.

4.2.2 Rely-Guarantee Reasoning. Rely-guarantee reasoning [20, 28] is a powerful technique for reasoning about concurrent programs using Hoare logic. Our framework's

Low level	High level
`t := best_len;`	`t := best_len;`
	`assume t >= ghost_best;`
`if (len < t) { ... }`	`if (len < t) { ... }`

Figure 10. In assume introduction, the high-level program has an extra enabling condition. The correspondence might be proven by establishing that best_len ≥ ghost_best is an invariant and that ghost_best is monotonically non-increasing.

generality lets us leverage this style of reasoning without relying on it as our *only* means of reasoning. Furthermore, our level-based approach lets the developer use such reasoning piecemeal. That is, they do not have to use rely-guarantee reasoning to establish *all* invariants *all at once*. Rather, they can establish some invariants and *cement* them into their program, i.e., add them as enabling conditions in one level so that higher levels can simply assume them.

Two programs exhibit the *assume-introduction* correspondence if they are identical except that the high-level program has additional enabling constraints on one or more statements. The correspondence requires that each added enabling constraint *always holds* in the low-level program at its corresponding program position.

Figure 10 gives an example using a variant of our running traveling-salesman example. In this variant, the correctness condition requires that we find the optimal solution, so it is not reasonable to simply replace the guard with * as we did in Figure 3. Instead, we want to justify the racy read of best_len by arguing that the result it reads is conservative, i.e., that at worst it is an over-estimate of the best length so far. We represent this best length with the ghost variable ghost_best and somehow establish that best_len >= ghost_best is an invariant. We also establish that between steps of a single thread, the variable ghost_best cannot increase; this is an example of a rely-guarantee predicate [20]. Together, these establish that t >= ghost_best always holds before the evaluation of the guard.

Benefits. The main benefit to using assume-introduction correspondence is that it adds enabling constraints to the program being reasoned about. More enabling constraints means fewer behaviors to be considered while locally reasoning about a step.

Another benefit is that it cements an invariant into the program. That is, it ensures that what is an invariant now will remain so even as further changes are made to the program as the developer abstracts it. For instance, after proving refinement of the example in Figure 10, the developer may produce a next-higher-level program by weakening the assignment t := best_len to t := *. This usefully eliminates the racy read to the variable best_len, but has the downside of eliminating the relationship between t and the variable best_len. But, now that we have cemented the invariant that t >= ghost_best, we do not need this relationship any more. Now, instead of reasoning about a program that performs a racy read and then branches based on it, we only reason about a program that chooses an arbitrary value and then blocks forever if that value does not have the appropriate relationship to the rest of the state. Notice, however, that assume-introduction can only be used if this condition is already known to *always hold* in the low-level program at this position. Therefore, assume-introduction never introduces any additional blocking in the low-level program.

Proof generation. The proof generator for this strategy uses rely-guarantee logic, letting the developer supply standard Hoare-style annotations. That is, the developer may annotate each method with preconditions and postconditions, may annotate each loop with loop invariants, and may supply invariants and rely-guarantee predicates.

Our strategy generates one lemma for each program path that starts at a method's entry and makes no backward jumps. This is always a finite path set, so it only has to generate finitely many lemmas. Each such lemma establishes properties of a state machine that resembles the low-level program's state machine but differs in the following ways. Only one thread ever executes and it starts at the beginning of a method. Calling another method simply causes the state to be havocked subject to its postconditions. Before evaluating the guard of a loop, the state changes arbitrarily subject to the loop invariants. Between program steps, the state changes arbitrarily subject to the rely-guarantee predicates and invariants.

The generated lemmas must establish that each step maintains invariants and rely-guarantee predicates, that method preconditions are satisfied before calls, that method postconditions are satisfied before method exits, and that loop invariants are reestablished before jumping back to loop heads. This requires several lemmas per path: one for each invariant, one to establish preconditions if the path ends in a method call, one to establish maintenance of the loop invariant if the path ends just before a jump back to a loop head, etc. The strategy uses these lemmas to establish the conditions necessary to invoke a library lemma that proves properties of rely-guarantee logic.

4.2.3 TSO Elimination.

We observe that even in programs using sophisticated lock-free mechanisms, most variables are accessed via a simple ownership discipline (e.g., "always by the same thread" or "only while holding a certain lock") that straightforwardly provides data race freedom (DRF) [2]. It is well understood that x86-TSO behaves indistinguishably from sequential consistency under DRF [5, 22]. Our level-based approach means that the developer need not prove they follow an ownership discipline for *all* variables to get the benefit of reasoning about sequential consistency. In particular, Armada allows a level where the sophisticated variables use regular assignments and the simple variables use TSO-bypassing assignments. Indeed, the developer need not even prove an ownership discipline for all such variables

```
var x:int32;
ghost var lockholder:Option<uint64>;
...
tso_elim x "s.s.ghosts.lockholder == Some(tid)"
```

Figure 11. Variables in a program, followed by invocation, in a recipe, of the TSO-elimination strategy. The part in quotation marks indicates under what condition the thread `tid` owns (has exclusive access to) the variable `x` in state `s`: when the ghost variable `lockholder` refers to that thread.

at once; they may find it simpler to reason about those variables one at a time or in batches. At each point, they can focus on proving an ownership discipline just for the specific variable(s) to which they are applying TSO elimination. As with any proof, if the developer makes a mistake (e.g., by not following the ownership discipline), Armada reports a proof failure.

A pair of programs exhibits the *TSO-elimination* correspondence if all assignments to a set of locations $\mathcal{L}$ in the low-level program are replaced by TSO-bypassing assignments. Furthermore, the developer supplies an *ownership predicate* (as in Figure 11) that specifies which thread (if any) owns each location in $\mathcal{L}$. It must be an invariant that no two threads own the same location at once, and no thread can read or write a location in $\mathcal{L}$ unless it owns that location. Any step releasing ownership of a location must ensure the thread's store buffer is empty, e.g., by being a fence.

4.2.4 Weakening.

As discussed earlier, two programs exhibit the weakening correspondence if they match except for certain statements where the high-level version admits a superset of behaviors of the low-level version. The strategy generates a lemma for each statement in the low-level program proving that, considered in isolation, it exhibits a subset of behaviors of the corresponding statement of the high-level program.

4.2.5 Non-deterministic Weakening.

A special case of weakening is when the high-level version of the state transition is non-deterministic, with that non-determinism expressed as an existentially-quantified variable. For example, in Figure 4 the guard on an `if` statement is replaced by the `*` expression indicating non-deterministic choice. For simplicity of presentation, that figure shows the recipe invoking the weakening strategy, but in practice, it would use *non-deterministic weakening*.

Proving non-deterministic weakening requires demonstrating a witness for the existentially-quantified variable. Our strategy uses various heuristics to identify this witness and generate the proof accordingly.

4.2.6 Combining.

Two programs exhibit the *combining* correspondence if they are identical except that an atomic block in the low-level program is replaced by a single statement in the high-level program that has a superset of its behaviors. This is analogous to weakening in that it replaces what appears to be a single statement (an atomic block) with a statement with a superset of behaviors. However, it differs subtly because our model for an atomic block is not a single step but rather a sequence of steps that cannot be interrupted by other threads.

The key lemma generated by the combining proof generator establishes that all paths from the beginning of the atomic block to the end of the atomic block exhibit behaviors permitted by the high-level statement. This involves breaking the proof into pieces, one for each path prefix that starts at the beginning of the atomic block and does not pass beyond the end of it.

4.2.7 Variable Introduction.

A pair of programs exhibits the *variable-introduction* correspondence if they differ only in that the high-level program has variables (and assignments to those variables) that do not appear in the low-level program.

Our strategy for variable introduction creates refinement proofs for program pairs exhibiting this correspondence. The main use of this is to introduce ghost variables that abstract the concrete state of the program. Ghost variables are easier to reason about because they can be arbitrary types and because they use sequentially-consistent semantics.

Another benefit of ghost variables is that they can obviate concrete variables. Once the developer introduces enough ghost variables, and establishes invariants linking the ghost variables to concrete state, they can weaken the program logic that depends on concrete variables to depend on ghost variables instead. Once program logic no longer depends on a concrete variable, the developer can hide it.

4.2.8 Variable Hiding.

A pair of programs $\langle L, H\rangle$ exhibits the *variable-hiding* correspondence if $\langle H, L\rangle$ exhibits the variable-introduction correspondence. In other words, the high-level program H has fewer variables than the low-level program L, and L only uses those variables in assignments to them. Our variable-hiding strategy creates refinement proofs for program pairs exhibiting this correspondence.

5 Implementation

Our implementation consists of a state-machine translator to translate Armada programs to state-machine descriptions; a framework for proof generation and a set of tools fitting in that framework; and a library of lemmas useful for invocation by proofs of refinement. It is open-source and available at https://github.com/microsoft/armada.

Since Armada is similar to Dafny, we implement the state-machine translator using a modified version of Dafny's parser and type-inference engine. After the parser and resolver run, our code performs state-machine translation. In all, our state-machine translator is 13,191 new source lines of code (SLOC [42]) of C#. Each state machine includes common Armada definitions of datatypes and functions; these constitute 873 SLOC of Dafny.

Table 1. Example programs used to evaluate Armada

Name	Description
Barrier	Barrier described by Schirmer and Cohen [38] as incompatible with ownership-based proofs
Pointers	Program using multiple pointers
MCSLock	Mellor-Crummey and Scott (MCS) lock [31]
Queue	Lock-free queue from liblfds library [29, 32]

Our proof framework is also written in C#. Its abstract syntax tree (AST) code is a modification of Dafny's AST code. We have an abstract proof generator that deals with general aspects of proof generation (§4.1), and we have one subclass of that generator for each strategy. Our proof framework is 3,322 SLOC of C#.

We also extend Dafny with a 1,767-SLOC backend that translates an Armada AST into C code compatible with CompCertTSO [41], a version of CompCert [4] that ensures the emitted code respects x86-TSO semantics.

Our general-purpose proof library is 5,618 SLOC of Dafny.

6 Evaluation

To show Armada's versatility, we evaluate it on the programs in Table 1. Our evaluations show that we can prove the correctness of: programs not amenable to verification via ownership-based methodologies [38], programs with pointer aliasing, lock implementations from previous frameworks [16], and libraries of real-world high-performance data structures.

6.1 Barrier

The Barrier program includes a barrier implementation described by Schirmer and Cohen [38]: "each processor has a flag that it exclusively writes (with volatile writes without any flushing) and other processors read, and each processor waits for all processors to set their flags before continuing past the barrier." They give this as an example that their ownership-based methodology for reasoning about TSO programs cannot support. Like other uses of Owens's publication idiom [34], this barrier is predicated on the allowance of races between writes and reads to the same location.

The key safety property is that each thread does its post-barrier write after all threads do their pre-barrier writes. We cannot use the TSO-elimination strategy since the program has data races, so we prove as follows. A first level uses variable introduction to add ghost variables representing initialization progress and which threads have performed their pre-barrier writes. A second level uses rely-guarantee to add an enabling condition on the post-barrier write that all pre-barrier writes are complete. This condition implies the safety property.

One author took ~3 days to write the proof levels, mostly to write invariants and rely-guarantee predicates involving x86-TSO reasoning. Due to the complexity of this reasoning, the original recipe had many mistakes; output from verification failures aided discovery and repair.

The implementation is 57 SLOC. The first proof level uses 10 additional SLOC for new variables and assignments, and 5 SLOC for the recipe; Armada generates 3,649 SLOC of proof. The next level uses 35 additional SLOC for enabling conditions, loop invariants, preconditions, and postconditions; 114 SLOC of Dafny for lemma customization; and 102 further SLOC for the recipe, mostly for invariants and rely-guarantee predicates. Armada generates 46,404 SLOC of proof.

6.2 Pointers

The Pointers program writes via distinct pointers of the same type. The correctness of our refinement depends on our static alias analysis proving these different pointers do not alias. Specifically, we prove that the program assigning values via two pointers refines a program assigning those values in the opposite order. The automatic alias analysis reveals that the pointers cannot alias and thus that the reversed assignments result in the same state. The program is 29 SLOC, the recipe is 7 SLOC, and Armada generates 2,216 SLOC of proof.

6.3 MCSLock

The MCSLock program includes a lock implementation developed by Mellor-Crummey and Scott [31]. It uses compare-and-swap instructions and fences for thread synchronization. It excels at fairness and cache-awareness by having threads spin on their own locations. We use it to demonstrate that our methodology allows modeling locks hand-built out of hardware primitives, as done for CertiKOS [23].

Our proof establishes the safety property that statements between `acquire` and `release` can be reduced to an atomic block. We use six transformations for our refinement proof, including the following two notable ones. The fifth transformation proves that both `acquire` and `release` properly maintain the ownership represented by ghost variables. For example, `acquire` secures ownership and `release` returns it. We prove this by introducing enabling conditions and annotating the program. The last transformation reduces statements between `acquire` and `release` into a single atomic block through reduction.

The implementation is 64 SLOC. Level 1 adds 13 SLOC to the program and uses 4 SLOC for its recipe. Each of levels 2–4 reduces program size by 3 SLOC and uses 4 SLOC for its recipe. Level 5 adds 33 SLOC to the program and uses 103 SLOC for its recipe. Level 6 adds 2 SLOC to the program and uses 21 SLOC for its recipe. Levels 5 and 6 collectively use a further 141 SLOC for proof customization. In comparison, the authors of CertiKOS verified an MCS lock via concurrent certified abstraction layers [23] using 3.2K LOC to prove the safety property.

6.4 Queue

The Queue program includes a lock-free queue from the liblfds library [29, 32], used at AT&T, Red Hat, and Xen. We use it to show that Armada can handle a practical, high-performance lock-free data structure.

Proof Our goal is to prove that the enqueue and dequeue methods behave like abstract versions in which enqueue adds to the back of a sequence and dequeue removes the first entry of that sequence, as long as at most one thread of each type is active. Our proof introduces an abstract queue, uses an inductive invariant and weakening to show that logging using the implementation queue is equivalent to logging using the abstract queue, then hides the implementation. This leaves a simpler enqueue method that appends to a sequence, and a dequeue method that removes and returns its first element.

It took ~6 person-days to write the proof levels. Most of this work involved identifying the inductive invariant to support weakening of the logging using implementation variables to logging using the abstract queue.

The implementation is 70 SLOC. We use eight proof transformations, the fourth of which does the key weakening described in the previous paragraph. The first three proof transformations introduce the abstract queue using recipes with a total of 12 SLOC. The fourth transformation uses a recipe with 92 SLOC, including proof customization, and an external file with 528 SLOC to define an inductive invariant and helpful lemmas. The final four levels hide the implementation variables using recipes with a total of 16 SLOC, leading to a final layer with 46 SLOC. From all our recipes, Armada generates 24,540 SLOC of proof.

Performance We measure performance in Docker on a machine with an Intel Xeon E5-2687W CPU running at 3.10 GHz with 8 cores and 32 GiB of memory. We use GCC 6.3.0 with `-O2` and CompCertTSO 1.13.8255. We use liblfds version 7.1.1 [29]. We run (1,000 times) its built-in benchmark for evaluating queue performance, using queue size 512.

Our Armada port of liblfds's lock-free queue uses modulo operators instead of bitmask operators, to avoid invoking bit-vector reasoning. To account for this, we also measure liblfds-modulo, a variant we write with the same modifications.

To account for the maturity difference between CompCertTSO and modern compilers, we also report results for the Armada code compiled with GCC. Such compilation is not sound, since GCC does not necessarily conform to x86-TSO; we only include these results to give an idea of how much performance loss is due to using CompCertTSO. To constrain GCC's optimizations and thereby make the comparison somewhat reasonable, we insert the same barriers liblfds uses before giving GCC our generated ClightTSO code.

Figure 12 shows our results. The Armada version compiled with CompCertTSO achieves 70% of the throughput of the liblfds version compiled with GCC. Most of this performance loss is due to the use of modulo operations rather than bitmasks, and the use of a 2013-era compiler rather than a modern one. After all, when we remove these factors, we achieve virtually identical performance (99% of throughput). This is not surprising since the code is virtually identical.

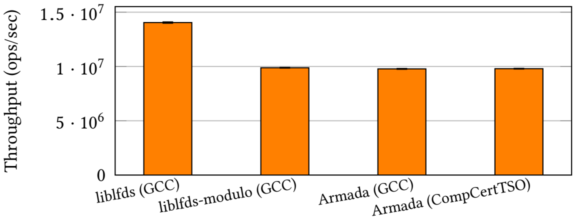

Figure 12. These are performance results for liblfds's lock-free queue vs. the corresponding code written in Armada. The Armada version, and our variant liblfds-modulo, use modulo rather than bitmask operations. Each data point is the mean of 1,000 trials; error bars indicate 95% confidence intervals.

7 Related Work

Concurrent separation logic [33] is based on unique ownership of heap-allocated memory via locking. Recognizing the need to support flexible synchronization, many program logics inspired by concurrent separation logic have been developed to increase expressiveness [10, 12, 13, 21, 26]. We are taking an alternative approach of refinement over small-step operational semantics that provides considerable flexibility at the cost of low-level modeling whose overhead we hope to overcome via proof automation.

CCAL and concurrent CertiKOS [17, 18] propose *certified concurrent abstraction layers*. Cspec [6] also uses layering to verify concurrent programs. Layering means that a system implementation is divided into layers, each built on top of the other, with each layer verified to conform to an API and specification assuming that the layer below conforms to its API and specification. Composition rules in CCAL ensure end-to-end termination-sensitive contextual refinement properties when the implementation layers are composed together. Armada does not (yet) support layers: all components of a program's implementation must be included in level 0. So, Armada currently does not allow independent verification of one module whose specification is then used by another module. Also, Armada only proves properties about programs while CCAL supports general composition, such as the combination of a verified operating system, thread library, and program. On the other hand, CCAL uses a strong memory model disallowing all data races, while Armada uses the x86-TSO memory model and thus can verify programs with benign races and lock-free data structures.

It is worth noting that our level-based approach can be seen as a special case of CCAL's layer calculus. If we consider the special case where specification of a layer is expressed in the form of a program, then refinement between lower level L and higher level H with respect to refinement relation $\mathcal{R}$ can be expressed in the layer calculus as $L \vdash_{\mathcal{R}} \emptyset : H$. That is, without introducing any additional implementation in the higher layer, the specification can nevertheless be transformed between the underlay and overlay interfaces. Indeed, the authors of concurrent CertiKOS sometimes use such $\emptyset$-implementation layers when one complex layer implementation cannot be further divided into smaller pieces [18, 23]. The proofs of refinement in these cases are complex, and might perhaps be more easily constructed using Armada-style levels and strategy-based automatic proof generation.

Recent work [7] uses the Iris framework [25] to reason about a concurrent file system. It too expects developers to write their code in a particular style that may limit performance optimization opportunities and the ability to port existing code. It also, like CertiKOS and CSPEC, requires much manual proof.

QED [14] is the first verifier for functional correctness of concurrent programs to incorporate reduction for program transformation and to observe that weakening atomic actions can eliminate conflicts and enable further reduction arguments. CIVL [19] extends and incorporates these ideas into a refinement-oriented program verifier based on the framework of layered concurrent programs [24]. (Layers in CIVL correspond to levels in Armada, not layers in CertiKOS and CSPEC.) Armada improves upon CIVL by providing a flexible framework for soundly introducing new mechanically-verified program transformation rules; CIVL's rules are proven correct only on paper.

8 Limitations and Future Work

In this section we discuss the limitations of the current design and prototype of Armada and suggest items for future work.

Armada currently supports the x86-TSO memory model [35] and is thus not directly applicable to other architectures, like ARM and Power. We believe x86-TSO is a good first step as it illustrates how to account for weak memory models, while still being simple enough to keep the proof complexity manageable. An important area of future work is to add support for other weak memory models.

As discussed in §7, Armada does not support layering but is compatible with it. So, we plan to add such support to increase the modularity of our proofs.

Armada uses Dafny to verify all proof material we generate. As such, the trusted computing base (TCB) of Armada includes not only the compiler and the code for extracting state machines from the implementation and specification, but also the Dafny toolchain. This toolchain includes Dafny, Boogie [3], Z3 [11], and our script for invoking Dafny.

Armada uses the CompCertTSO compiler, whose semantics is similar, but not identical, to Armada's. In particular, CompCertTSO represents memory as a collection of blocks, while Armada adopts a hierarchical forest representation. Additionally, in CompCertTSO the program is modeled as a composition of a number of state machines—one for each thread—alongside a TSO state machine that models global memory. Armada, on the other hand, models the program as a single state machine that includes all threads and the global memory. We currently assume that the CompCertTSO model refines our own. It is future work to formally prove this by demonstrating an injective mapping between the memory locations and state transitions of the two models.

Because Armada currently emits proofs about finite behaviors, it can prove safety but not liveness properties. We plan to address this via support for infinite behaviors.

Armada currently supports state transitions involving only the current state, not future states. Hence, Armada can encode *history variables* but not *prophecy variables* [1]. Expanding the expressivity of state transitions is future work.

Since we only consider properties of single behaviors, we cannot verify hyperproperties [8]. But, we can verify safety properties that *imply* hyperproperties, such as the unwinding conditions Nickel uses to prove noninterference [37, 39].

9 Conclusion

Via a common, low-level semantic framework, Armada supports a panoply of powerful strategies for automated reasoning about memory and concurrency, even while giving developers the flexibility needed for performant code. Armada's strategies can be soundly extended as new reasoning principles are developed. Our evaluation on four case studies demonstrates Armada is a practical tool that can handle a diverse set of complex concurrency primitives, as well as real-world, high-performance data structures.

Acknowledgments

The authors are grateful to our shepherd, Ronghui Gu, and the anonymous reviewers for their valuable feedback that greatly improved the paper. We also thank Tej Chajed, Chris Hawblitzel, and Nikhil Swamy for reading early drafts of the paper and providing useful suggestions, and Rustan Leino for early discussions and for helpful Dafny advice and support.

This work was supported in part by the National Science Foundation and VMware under Grant No. CNS-1700521, a grant from the Alfred P. Sloan Foundation, and a Google Faculty Fellowship.

References

[1] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theoretical Computer Science* 82, 2 (May 1991), 253–284.

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *Proc. International Symposium on Computer Architecture (ISCA)*. 2–14.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A modular reusable verifier for

object-oriented programs. *Proceedings of Formal Methods for Components and Objects (FMCO)* (2006).

[4] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal verification of a C compiler front-end. In *Proc. International Symposium on Formal Methods (FM)*. 460–475.

[5] Gérard Boudol and Gustavo Petri. 2009. Relaxed memory models: An operational approach. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 392–403.

[6] Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 306–322.

[7] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. 243–258.

[8] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[9] Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *Concurrency Theory (CONCUR)*. 317–331.

[10] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*. 207–231.

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

[12] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 287–300.

[13] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Proc. European Symposium on Programming (ESOP)*. 363–377.

[14] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 2–15.

[15] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2004. Exploiting purity for atomicity. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 221–231.

[16] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 595–608.

[17] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 653–669.

[18] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 646–661.

[19] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and modular refinement reasoning for concurrent programs. In *Proc. Computer Aided Verification (CAV)*. 449–465.

[20] C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (Oct. 1983), 596–619.

[21] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris From the Ground Up: A Modular Foundation for Higher-order Concurrent Separation Logic. *Journal of Functional Programming* 28, e20 (2018).

[22] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 175–189.

[23] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and liveness of MCS lock—layer by layer. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*. 273–297.

[24] Bernhard Kragl and Shaz Qadeer. 2018. Layered concurrent programs. In *Proc. International Conference on Computer Aided Verification (CAV)*. 79–102.

[25] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *Proc. European Symposium on Programming (ESOP)*. 696–723.

[26] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go With the Flow: Compositional Abstractions for Concurrent Data Structures. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 37:1–37:31.

[27] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 348–370.

[28] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 455–468.

[29] LibLFDS. 2019. LFDS 7.11 queue implementation. https://github.com/liblfds/liblfds7.1.1/tree/master/liblfds7.1.1/liblfds711/src/lfds711_queue_bounded_singleproducer_singleconsumer.

[30] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721.

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (Feb. 1991), 21–65.

[32] Maged M. Michael and Michael L. Scott. 2006. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*. 267–275.

[33] Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (2007), 271–307.

[34] Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. European Conference on Object-Oriented Programming*. 478–503.

[35] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proc. Theorem Proving in Higher Order Logics (TPHOLs)*. 391–407.

[36] Shaz Qadeer. 2019. Private Communication.

[37] John Rushby. 1992. Noninterference, Transitivity, and Channel-control Security Policies. Technical Report CSL-92-02, SRI International.

[38] Norbert Schirmer and Ernie Cohen. 2010. From total store order to sequential consistency: A practical reduction theorem. In *Proc. Interactive Theorem Proving (ITP)*. 403–418.

[39] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: a framework for design and verification of information flow control systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 287–305.

[40] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 32–41.

[41] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. 43–54.

[42] David A. Wheeler. 2004. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.