Certified Concurrent Abstraction Layers

Ronghui Gu
Yale University, USA
Columbia University, USA

Zhong Shao
Yale University
USA

Jieung Kim
Yale University
USA

Xiongnan (Newman) Wu
Yale University, USA

Jérémie Koenig
Yale University, USA

Vilhelm Sjöberg
Yale University, USA

Hao Chen
Yale University, USA

David Costanzo
Yale University, USA

Tahina Ramananandro
Microsoft Research, USA

Abstract

Concurrent abstraction layers are ubiquitous in modern computer systems because of the pervasiveness of multithreaded programming and multicore hardware. Abstraction layers are used to hide the implementation details (e.g., fine-grained synchronization) and reduce the complex dependencies among components at different levels of abstraction. Despite their obvious importance, concurrent abstraction layers have not been treated formally. This severely limits the applicability of layer-based techniques and makes it difficult to scale verification across multiple concurrent layers.

In this paper, we present CCAL—a fully mechanized programming toolkit developed under the CertiKOS project— for specifying, composing, compiling, and linking certified concurrent abstraction layers. CCAL consists of three technical novelties: a new game-theoretical, strategy-based compositional semantic model for concurrency (and its associated program verifiers), a set of formal linking theorems for composing multithreaded and multicore concurrent layers, and a new CompCertX compiler that supports certified thread-safe compilation and linking. The CCAL toolkit is implemented in Coq and supports layered concurrent programming in both C and assembly. It has been successfully applied to build a fully certified concurrent OS kernel with fine-grained locking.

CCS Concepts • **Theory of computation → Logic and verification**; **Abstraction**; • **Software and its engineering → Functionality**; **Software verification**; **Concurrent programming languages**;

1 Introduction

Abstraction layers (e.g., circuits, ISA, device drivers, OS kernels, and hypervisors) are widely used in modern computer systems to help reduce the complex interdependencies among components at different levels of abstraction [3, 48]. An abstraction layer defines an interface that hides the implementation details of its underlying software or hardware components. Client programs built on top of each layer are understood solely based on the interface, independent of the layer implementation.

As multicore hardware and multithreaded programming become more pervasive, many of these abstraction layers also become *concurrent* in nature. Their interfaces not only hide the concrete data representations and algorithmic details, but also create an illusion of *atomicity* for all of their methods: each method call is viewed as if it completes in a single step, even though its implementation contains complex interleavings with operations done by other threads. Herlihy et al. [19, 20] advocated using layers of these atomic objects to construct large-scale concurrent software systems.

Figure 1 presents a few common concurrent layer objects in a modern multicore runtime. Here we use the light gray color to stand for thread-local (or CPU-local) objects, blue (also with round dots in their top-right corner) for objects shared between CPU cores, green for objects exported and shared between threads, and orange for threads themselves. Above the hardware layers, we must first build an efficient and starvation-free spinlock implementation [36]. With spinlocks, we can implement shared objects for sleep and pending thread queues, which are then used to implement the thread schedulers, and the primitives yield, sleep, and wakeup. On
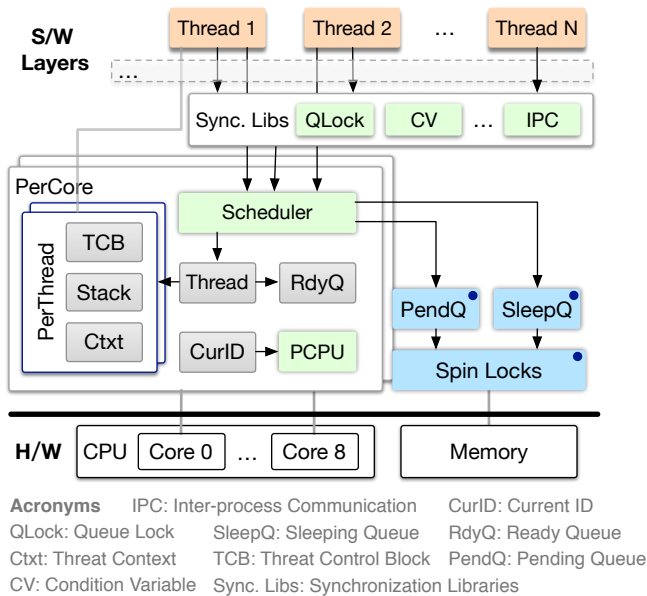
Figure 1. An overview of concurrent abstraction layers in a modern multithreaded and multicore environment (arrow means possible function call from one component to another).

top of them, we can then implement high-level synchronization libraries such as queuing locks, condition variables (CV), and message-passing primitives [2].

Despite the importance of concurrent layers and a large body of recent work on shared-memory concurrency verification [5, 7, 8, 13, 23, 29, 30, 42, 45, 50, 57–59], there are no certified programming tools that can specify, compose, and compile concurrent layers to form a whole system [6]. Formal reasoning across multiple concurrent layers is challenging because different layers often exhibit different interleaving semantics and have a different set of observable events. For example, the spinlock module in Fig. 1 assumes a multicore model with an overlapped execution of instruction streams from different CPUs. This model differs significantly from the multithreading model for building high-level synchronization libraries: each thread will block instead of spinning if a queuing lock or a CV event is not available; and it must count on other threads to wake it up to ensure liveness.

Reasoning across these different abstraction levels requires a general, unified compositional semantic model that can cover all of these concurrent layers. It must also support a general "parallel layer composition rule" that can handle explicit thread control primitives (e.g., sleep and wakeup). It must also support vertical composition [2] of these concurrent layer objects [19] while preserving both the linearizability and progress (e.g., starvation-freedom) properties.

Contributions. In this paper, we present CCAL—a fully mechanized programming toolkit implemented in Coq [55]

and developed under the CertiKOS project [16] for building certified *concurrent* abstraction layers. As shown in Fig. 2, CCAL consists of a novel compositional semantic model for concurrency, a collection of C and assembly program verifiers, a library for building layered refinement proofs, a thread-safe verified C compiler based on CompCertX [15], and a set of certified linking tools for composing multi-threaded or multicore layers.

We define a certified concurrent abstraction layer as a triple $(L_1[A], M, L_2[A])$ plus a mechanized proof object showing that the layer implementation M, running on behalf of a thread set A over the interface L_1, indeed faithfully implements the desirable interface L_2 above. Our compositional semantics model is based upon ideas from game semantics [38]. It enables local reasoning such that the implementation can be first verified over a single thread t by building $(L_1[\{t\}], M, L_2[\{t\}])$ without worrying too much about the concurrency and the guarantees can then be propagated to the whole concurrent machine by parallel compositions.

Following Gu et al. [15], certified concurrent layers enforce *termination-sensitive* contextual correctness property. In the concurrent setting, this means that every certified concurrent object satisfies not only a safety property (e.g., *linearizability*) [10, 20] but also a progress property (e.g., *starvation-freedom*) [33].

The CCAL toolkit has already been used in multiple large-scale verification projects under CertiKOS: Gu et al. [16] have successfully used CCAL to build the world's first fully certified concurrent OS kernel; Sjöberg et al. [53] used CCAL to verify the safety and liveness of a complex MCS lock implementation [36]. Neither of these two papers [16, 53] explained the internals of CCAL and how and why it can work so effectively.

This paper, rather than focusing on the applications of CCAL, gives an in-depth exploration of the CCAL toolkit itself and how it can be used for building various certified concurrent objects. Over Gu et al. [16], this paper presents the following three technical contributions:

- We introduce a new compositional semantic model for shared-memory concurrent abstract machines and prove a general parallel layer composition rule. We show how our new framework is used to specify, verify, and compose various concurrent objects at different levels of abstraction (see Fig. 1).

- We show how to apply standard *simulation* techniques [15, 26] to verify the safety and liveness of concurrent objects in a unified setting. Because our environment context specifies not just the environment's past events but also *future events*, we can readily impose temporal invariants such as fairness requirements (for schedulers) or *definite actions* [30] (for releasing locks). This allows us to give full specifications for lock primitives and support vertical
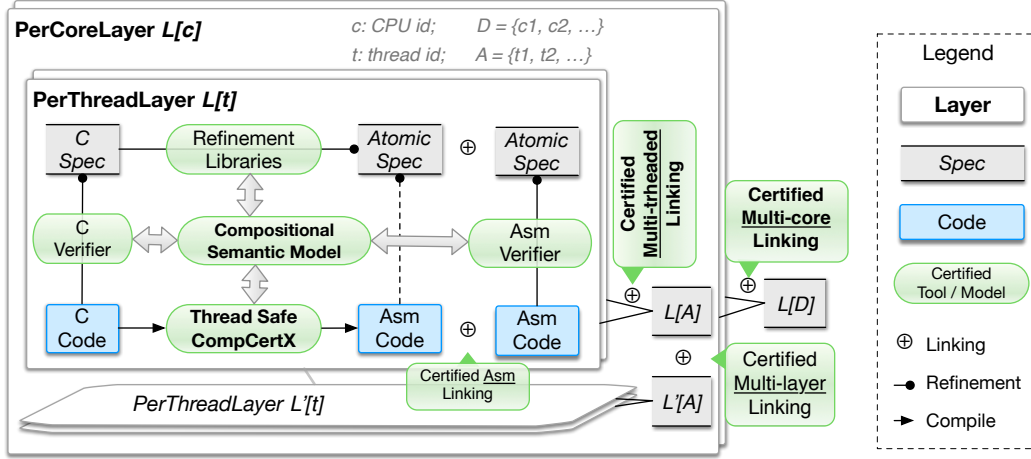
Figure 2. System architecture of the CCAL programming toolkit.

composition of starvation-free atomic objects, none of which have ever been possible before [30].

- We have also developed a new *thread-safe* version of the CompCertX compiler [15] that can compile certified concurrent C layers into assembly layers. To support certified multithreaded linking, we have developed a new extended algebraic memory model (for CompCertX) whereby stack frames allocated for each thread are combined to form a single coherent CompCert-style memory.

Scope and Paper Outline. While the notion of certified concurrent layer can potentially be applied to a more general setting [34, 49], in this paper, we focus on shared-memory concurrent program modules as described in Anderson and Dahlin [2] and Herlihy and Shavit [19], which are sufficient to verify layers as shown in Fig. 1. Section 7 discusses related work and puts our work in broader perspective. Both the CCAL toolkit and all our assembly (or C) machines assume strong sequential consistency for shared primitives. Adding support for relaxed memory models is left as future work.

2 Overview

The key challenge of verifying concurrent systems is how to untangle the complexities of module dependencies and interleaving, and then verify different parts independently and locally at the layers they belong to. To address this issue, we introduce a layer-based approach to formally specify, certify, and compose these (concurrent) layers.

In this section, to illustrate our layered techniques, we will walk through a small example (see Fig. 3) that uses a lock to protect a critical section. In this example, the client program P has two threads running on two different CPUs; each thread makes one call to the primitive foo provided by the concurrent layer interface L_2. The interface L_2 is implemented by the concurrent module M_2, which in turn is built

```
1  struct ticket_lock {
2    volatile uint n, t;
3  };
4  //Methods provided by L_0
5  extern uint get_n();
6  extern void inc_n();
7  extern uint FAI_t();
8  extern void f();
9  extern void g();
10 extern void hold();
11 //M_1 module
12 void acq () {
13   uint my_t = FAI_t();
14   while(get_n()!=my_t){};
15   hold();
16 }
17 void rel () { inc_n(); }
18 //Methods provided by L_1

19 extern void acq();
20 extern void rel();
21 extern void f();
22 extern void g();
23 //M_2 module
24 void foo () {
25   acq();
26   f(); g();
27   rel();
28 }
29 //Methods provided by L_2
30 extern void foo();
31
32 //Client program P
33 //Thread running on CPU 1
34 void T1 () { foo(); }
35 //Thread running on CPU 2
36 void T2 () { foo(); }
```

Figure 3. Certified concurrent layers involving ticket locks.

on top of the interface L_1. The method foo calls two primitives f and g in a critical section protected by a lock. The lock is implemented over the interface L_0 using the ticket lock algorithm [36] in module M_1. The lock maintains two integer variables n (the "now serving" ticket number) and t (i.e., next ticket number). The lock acquire method acq fetches-and-increments the next ticket number (by FAI_t) and spins until the fetched number is served. The lock release method rel simply increments the "now serving" ticket number by inc_n. These primitives are provided by L_0 and implemented using x86 atomic instructions. L_0 also provides the primitives f and g that are later passed on to L_1, as well as a no-op primitive hold called by acq to announce that the lock has been taken.

Certified Abstraction Layers. Gu et al. [15] defines a certified sequential abstraction layer as a predicate "$L' \vdash_R M : L$" plus a *mechanized proof object* for the predicate, showing that the layer implementation M, built on top of the interface L' (which we call the *underlay* interface), indeed faithfully *implements* the desirable interface L above (which we call the *overlay* interface) via a simulation relation R.

Here, the implementation M is a program module written in assembly (or C). A layer interface L consists of a set of abstract states and primitives. An abstract layer machine based on L is just the base assembly (or C) machine extended with abstract states and primitives defined in L. The *implements* relation ($\sqsubseteq_R$) is formally defined as a forward simulation [27, 35, 37, 43] with the (simulation) relation R.

A certified layer enforces a *contextual* correctness property: a correct layer is like a "certified compiler," converting any *safe* client program P running on top of L into one that has the same behavior but runs on top of L' (i.e., by "compiling" abstract primitives in L into their implementation in M). If we use "$[\![\cdot]\!]_L$" to denote the behavior of the layer machine based on L, the correctness property of "$L' \vdash_R M : L$" is written formally as "$\forall P. [\![P \oplus M]\!]_{L'} \sqsubseteq_R [\![P]\!]_L$" where $\oplus$ denotes a linking operator over programs P and M.

Certified Concurrent Layers. To support concurrency, each layer interface L is parameterized with a "focused" thread set A (where $A \subseteq D$ and D is the domain of all thread/CPU IDs). The layer machine based on a concurrent layer interface $L[A]$ specifies the execution of threads in A (with threads outside A considered as the *environment*). For the example in Fig. 3, the domain D is $\{1, 2\}$. If we treat $\{1\}$ as the focused thread set, the environment contains thread 2. For readability, we often abbreviate $L[\{i\}]$ as $L[i]$ where $i \in D$.

A concurrent layer interface extends its sequential counterpart with a set of abstract *shared* primitives and a global log l. Unlike calls to thread-local primitives which are not *observable* by other threads, each shared primitive call (together with its arguments) is recorded as an observable event appended to the end of the global log. For example, FAI_t (see Fig. 3) called from thread i takes a log l to a log "$l \cdot (i.\text{FAI\_t})$" with the symbol "$\cdot$" means "cons-ing" an event to the log.

To define the semantics of a concurrent program P in a *generic* way, we develop a novel compositional (operational) model based upon ideas from game semantics [38]. Each run of P over $L[D]$ is viewed as playing a game involving members of D (plus a scheduler): each participant $i \in D$ contributes its play by appending events into the global log l; its *strategy* φ_i is a deterministic partial function from the current log l to its next move $\varphi_i(l)$ whenever the last event in l transfers control back to i.

For example, suppose thread i only invokes FAI_t, its strategy φ_i can be represented as an automaton:

$$\xrightarrow{} \bigcirc \xrightarrow{?l, !i.\text{FAI\_t}, \downarrow t} \circledcirc \circlearrowright\, {?l', !\epsilon}$$
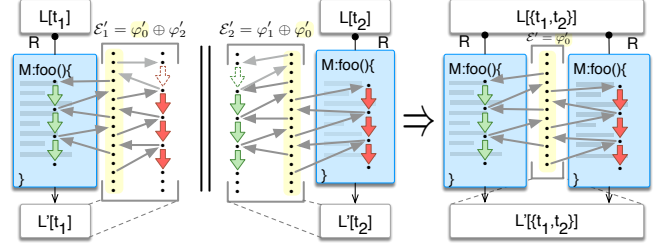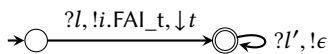

Figure 4. Environment contexts and parallel layer composition.

Suppose the global log is equal to l when the control is transferred to i (denoted as "$?l$"). Thread i first generates the event $i.\text{FAI\_t}$ (denoted as "$!i.\text{FAI\_t}$") and returns the ticket number t (denoted as "$\downarrow t$") calculated from l. It then becomes idle (denoted as a reflexive edge labeled with "$?l', !\epsilon$") and will not produce any more events. The ticket number t is calculated by a function that counts the fetch-and-increment events in l. Such functions that reconstruct the current shared state from the log are called *replay* functions.

The scheduler (denoted as φ_0) acts as a judge of the game. At each round, it picks one thread to make a move (and generate events) using its strategy. The *behavior* of the *whole* layer machine (denoted as "$[\![\cdot]\!]_{L[D]}$") is then just the set of logs generated by playing the game under all possible schedulers.

When focusing on a subset of threads A, the semantics (or execution) of the (concurrent) layer machine based on an interface $L[A]$ is defined over its set of *valid environment contexts*. Each environment context (denoted as $\mathcal{E}$) provides a strategy for its "environment," i.e., the union of the strategies by the scheduler plus those participants *not* in A.

For example, Fig. 4 shows a system with two threads (t_1 and t_2) and a scheduler. On the left, it shows one execution of method foo over the layer machine $L'[t_1]$ under a specific environment context $\mathcal{E}_1'$. Here, $\mathcal{E}_1'$ is the union of the strategy φ_0' for the scheduler and φ_2' for thread t_2. In the middle, it shows the execution of foo (invoked by t_2) over $L'[t_2]$ under the environment context $\mathcal{E}_2'$. On the right, it shows the interleaved execution of two invocations to foo over $L'[\{t_1, t_2\}]$ where the environment context $\mathcal{E}'$ is just the scheduler strategy φ_0'.

Given an environment context $\mathcal{E}$ which also contains a specific scheduler strategy, the execution of P over $L[A]$ is deterministic; the concurrent machine will run P when the control is transferred to any member of A, but will ask $\mathcal{E}$ for the next move when the control is transferred to the environment.

To enforce the safety of environmental moves, each layer interface also specifies its set of valid environment contexts. This validity corresponds to a generalized version of the "rely" (or "assume") condition in rely-guarantee-based reasoning [7, 8, 11, 51, 58]. Each layer interface can also provide its own
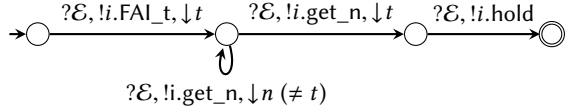
"guarantee" condition. These conditions are simply expressed as invariants over the global log.

Local Layer Interface. Consider the case where the focused thread set is a singleton $\{i\}$. Since the environmental executions (including the interleavings) are all encapsulated into the environment context, $L[i]$ is actually a sequential-like (or *local*) interface parameterized over $\mathcal{E}$. Before each move of a client program P over this local interface, the layer machine first repeatedly asks $\mathcal{E}$ for environmental events until the control is transferred to i. It then makes the move based on received events. Thus, the semantics of running P over $L[i]$ (denoted as $\langle\!\langle P \rangle\!\rangle_{L[i]}$) can also be viewed as a strategy.

The correctness property asserting that a concurrent module on top of a local layer interface indeed satisfies its specification (i.e., a more abstract strategy) is defined by the *strategy simulation* via a simulation relation R for logs.

Definition 2.1. ($\leq_R$) We say a strategy φ is simulated by another strategy φ' with a simulation relation R and write "$\varphi \leq_R \varphi'$", if, and only if, for any two related (by R) environmental event sequences and any two related initial logs, we have that for any log l produced by φ, there must exist a log l' that can be produced by φ' such that l and l' also satisfy R.

Consider the acq method of the ticket lock module M_1 running over $L_0[i]$ (see Fig. 3). Its specification can be represented as the following strategy $\varphi'_{\text{acq}}[i]$:

$$\xrightarrow{\quad} \bigcirc \xrightarrow{?\mathcal{E},\,!i.\text{FAI\_t},\downarrow t} \bigcirc \xrightarrow{?\mathcal{E},\,!i.\text{get\_n},\downarrow t} \bigcirc \xrightarrow{?\mathcal{E},\,!i.\text{hold}} \circledcirc$$
$$?\mathcal{E},\,!i.\text{get\_n},\downarrow n\,(\neq t)$$

We write $?\mathcal{E}$ for querying $\mathcal{E}$. We can prove that the simulation "$\langle\!\langle \text{acq} \rangle\!\rangle_{L_0[i]} \leq_{\text{id}} \varphi'_{\text{acq}}[i]$" holds for the identical relation: for any equal $\mathcal{E}$ and equal initial state, if $\varphi'_{\text{acq}}[i]$ takes one step, acq can take one (or more) steps to generate the same event and the resulting states are still equal. This correctness property is also used to define certified concurrent layers:

$$L_0[i] \vdash_{\text{id}} \text{acq} : \varphi'_{\text{acq}}[i] \quad :\equiv \quad \langle\!\langle \text{acq} \rangle\!\rangle_{L_0[i]} \leq_{\text{id}} \varphi'_{\text{acq}}[i]$$

Let "$M_1 := \text{acq} \oplus \text{rel}$" and "$L'_1[i] := \varphi_{\text{acq}}[i]' \oplus \varphi_{\text{rel}}[i]'$". By showing that the lock release satisfies its specification (i.e., "$L_0[i] \vdash_{\text{id}} \text{rel} : \varphi'_{\text{rel}}[i]$") and by the horizontal composition rule (see Sec. 3.3), we have:

$$L_0[i] \vdash_{\text{id}} M_1 : L'_1[i] \quad :\equiv \quad \langle\!\langle M_1 \rangle\!\rangle_{L_0[i]} \leq_{\text{id}} L'_1[i] \qquad (2.1)$$

The notations are extended to a set of strategies, meaning that each strategy of $L'_1[i]$ simulates the one of $\langle\!\langle M_1 \rangle\!\rangle_{L_0[i]}$.

Higher-level Strategies. Although the specifications above (e.g., $\varphi'_{\text{acq}}[i]$) are abstract (i.e., language independent), low-level implementation details and interleavings within the module are still exposed. For example, $\varphi'_{\text{acq}}[i]$ reveals the loop that repeatedly interacts with the environment to check the serving ticket number. To simplify the verification of
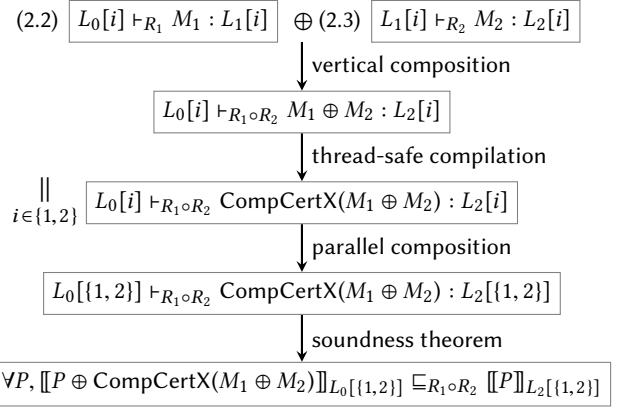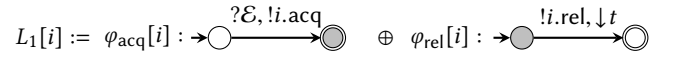
(2.2) $\boxed{L_0[i] \vdash_{R_1} M_1 : L_1[i]}$ $\oplus$ (2.3) $\boxed{L_1[i] \vdash_{R_2} M_2 : L_2[i]}$

$\downarrow$ vertical composition

$\boxed{L_0[i] \vdash_{R_1 \circ R_2} M_1 \oplus M_2 : L_2[i]}$

$\downarrow$ thread-safe compilation

$\displaystyle\parallel_{i \in \{1,2\}} \boxed{L_0[i] \vdash_{R_1 \circ R_2} \text{CompCertX}(M_1 \oplus M_2) : L_2[i]}$

$\downarrow$ parallel composition

$\boxed{L_0[\{1,2\}] \vdash_{R_1 \circ R_2} \text{CompCertX}(M_1 \oplus M_2) : L_2[\{1,2\}]}$

$\downarrow$ soundness theorem

$\boxed{\forall P,\ [\![P \oplus \text{CompCertX}(M_1 \oplus M_2)]\!]_{L_0[\{1,2\}]} \sqsubseteq_{R_1 \circ R_2} [\![P]\!]_{L_2[\{1,2\}]}}$

Figure 5. Layer verification of the ticket lock example using CCAL.

components using locks, we have to *refine* the strategies of $L'_1[i]$ to a higher-level interface $L_1[i]$ that is *atomic*:

$$L_1[i] := \varphi_{\text{acq}}[i] : \xrightarrow{} \bigcirc \xrightarrow{?\mathcal{E},\,!i.\text{acq}} \circledcirc \quad \oplus \quad \varphi_{\text{rel}}[i] : \xrightarrow{} \bullet \xrightarrow{!i.\text{rel},\downarrow t} \circledcirc$$

Here, $\varphi_{\text{acq}}[i]$ simply queries $\mathcal{E}$ and produces a single event $i.\text{acq}$. It then enters a so-called *critical* state (marked as gray) to prevent losing the control until the lock is released. Thus, there is no need to ask $\mathcal{E}$ in critical state.

To prove the strategy simulation between $L'_1[i]$ and $L_1[i]$, we have to pose "rely" (i.e., validity) conditions $\mathcal{R}$ over the environment context of $L'_1[i]$:

- $L'_1[i].\mathcal{R}_{hs}$: the scheduler strategy φ'_{hs} must be *fair*.

- $L'_1[i].\mathcal{R}_j$ $(j \neq i)$: lock-related events generated by φ_j must follow $\varphi_{\text{acq}'}[j]$ and $\varphi_{\text{rel}'}[j]$, and the held locks will eventually be released.

These conditions ensure that the loop (waiting for the ticket to be served) in $\varphi'_{\text{acq}}[i]$ terminates. Also, they can be used to prove that each run of $L'_1[i]$ is captured by $L_1[i]$. For example, if the scheduler strategy φ'_{hs} schedules as "1, 2, 2, 1, 1, 2, 1, 2, 1, 1, 2, 2," running P (see Fig. 3) over $L'_1[D]$ generates the log:

$$l'_g := \quad (1.\text{FAI\_t}) \bullet (2.\text{FAI\_t}) \bullet (2.\text{get\_n}) \bullet (1.\text{get\_n}) \bullet (1.\text{hold}) \bullet (2.\text{get\_n})$$
$$\bullet (1.\text{f}) \bullet (2.\text{get\_n}) \bullet (1.\text{g}) \bullet (1.\text{inc\_n}) \bullet (2.\text{get\_n}) \bullet (2.\text{hold})$$

This interleaving can be captured by a higher-level scheduler φ_{hs} producing "1, 2" (recall that thread 1 is in the critical state while holding the lock), and the generated log at $L_1[D]$ is:

$$l_g := \quad (1.\text{acq}) \bullet (1.\text{f}) \bullet (1.\text{g}) \bullet (1.\text{rel}) \bullet (2.\text{acq})$$

Although logs (and events) at these two layers are different, the order of lock acquiring and the resulting shared state (calculated from logs by replay functions) are exactly the same. By defining the relation R_1 over logs as mapping events $i.\text{acq}$ to $i.\text{hold}$, $i.\text{rel}$ to $i.\text{inc\_n}$ and other lock-related events

to empty ones, we can prove:

$$L_1'[i] \leq_{R_1} L_1[i]$$

Then by the predicate (2.1) and the weakening rule (i.e., the Wκ rule in Fig. 9), we have that:

$$L_0[i] \vdash_{\mathbf{id} \circ R_1 = R_1} M_1 : L_1[i] \qquad (2.2)$$

Similarly, for the foo method (i.e., M_2 in Fig. 3), we can also introduce a low-level strategy $\varphi_{\text{foo}}'[i]$ as the first step:

$$L_2'[i] := \varphi_{\text{foo}}'[i] : \rightarrow \bigcirc \xrightarrow{?\mathcal{E}, !i.\text{acq}} \bullet \xrightarrow{!i.\text{f}} \bullet \xrightarrow{!i.\text{g}} \bullet \xrightarrow{!i.\text{rel}} \circledcirc$$

Then we prove that a high-level atomic interface φ_{foo}:

$$L_2[i] := \varphi_{\text{foo}}[i] : \rightarrow \bigcirc \xrightarrow{?\mathcal{E}, !i.\text{foo}} \circledcirc$$

simulates (with some R_2) φ_{foo}', which in turn simulates foo:

$$L_2'[i] \leq_{R_2} L_2[i] \qquad L_1[i] \vdash_{\mathbf{id}} M_2 : L_2'[i] \qquad (2.3)$$

Based on $L_2'[i]$, we can derive the "guarantee" condition $\mathcal{G}$ of the thread i saying that held locks are always released within three steps, which is consistent but more concrete than the rely condition $\mathcal{R}$ defined above.

Parallel Layer Composition. We say that two layer interfaces $L[t_1]$ and $L[t_2]$ are *compatible* if the guarantee $\mathcal{G}$ of each interface implies the other interface's rely conditions $\mathcal{R}$. The new compositional model allows us to prove a general *parallel layer composition* rule: if $L'[t_1]$ is compatible with $L'[t_2]$, $L[t_1]$ is compatible with $L[t_2]$, and "$L'[t] \vdash_R M : L[t]$" holds for every $t \in \{t_1, t_2\}$, then we have "$L'[\{t_1, t_2\}] \vdash_R M : L[\{t_1, t_2\}]$." Figure 4 shows how to compose certified local layers (one for t_1 and another for t_2) to build a certified layer for the entire machine (with t_1 and t_2 both focused).

Thread-Safe CompCertX and Layer Linking. Since the local layer interface is sequential-like, we can adapt the CompCertX compiler [15] to be thread-safe by merging the stack frames of threads on the same CPU into a single stack. In this way, certified C layers can be compiled into certified assembly layers. We can then apply the horizontal, vertical, and the new parallel layer composition rules (see Sec. 3.3) to construct the certified concurrent layer for the entire system (see Fig. 5). Finally, from "$L'[D] \vdash_R M : L[D]$," the soundness theorem enforces a strong contextual refinement property saying that, for any client program P, we have that for any log l in the behavior $[\![P \oplus M]\!]_{L'[D]}$, there must exist a log l' in the behavior $[\![P]\!]_{L[D]}$ such that l and l' satisfy R.

Theorem 2.2 (Soundness)**.**

$$L'[D] \vdash_R M : L[D] \quad \Rightarrow \quad \forall P, [\![P \oplus M]\!]_{L'[D]} \sqsubseteq_R [\![P]\!]_{L[D]}$$

3 Concurrent Layer Interface and Calculus

In this section we instantiate our compositional model for the x86 multicore hardware and explain the concurrent layer interface and the layer calculus in more detail.

```
1 Function σ'_pull (s: State) (b: Loc) :=
2   match s.a.status b with
3     |free=>ret s{l: s.c.pull(b)::s.l}{a.status.b: own s.c}
4     | _ => None (* get stuck*)
5   end.
```

Figure 6. Pseudocode of the pull specification of $\mathbb{M}_{x86}$ in Coq.

3.1 Multiprocessor Machine Model

The multiprocessor machine model $\mathbb{M}_{x86}$ is defined by the machine state, the transition relation, and the memory model.

Machine State. As shown in Fig. 7, the state of $\mathbb{M}_{x86}$ is denoted as a tuple "$s := (c, f_\rho, m, a, l)$," where the components are the current CPU ID c, all CPUs' private states f_ρ (i.e., a partial map from CPU ID to private state ρ), a shared memory state m, an abstract state a, and a global event log l. The private state ρ consists of CPU-private memory pm (invisible to other CPUs) and a register set rs. The shared memory state m is shared among all CPUs. Each location b in both local and shared memories contains a memory value v. The abstract state a is generally used in our layered approach to summarize in-memory data structures from lower layers. It is not just a ghost state, because it affects program execution when making primitive calls. The global log l is a list of observable events, recording all shared operations that affect more than one CPU. Events generated by different CPUs are interleaved in the log, following the actual chronological order of events.

Transition Relation. The machine $\mathbb{M}_{x86}$ has two types of transitions that are arbitrarily and nondeterministically interleaved: program transitions and hardware scheduling.

Program transitions are one of three possible types: instruction executions, private primitive calls, and shared primitive calls. The first two types are "silent", in that they do not generate events. Shared primitives, on the other hand, provide the only means for accessing and appending events to the global log. The transitions for instructions only change ρ, pm, and m, and are defined as standard operational semantics for C or x86-assembly, similar to (and in fact based on) the operational semantics used in CompCert [27]. Primitive calls are specific to our style of verification: they directly specify the semantics of function f from underlying layers as a relation σ_f defined in Coq. This relation specifies how the state is updated after f is called with the given arguments and what the return value is.

Hardware scheduling transitions change the current CPU ID c to some ID c' (recorded as a scheduling event) and can be arbitrarily interleaved with program transitions. In other words, at any step, $\mathbb{M}_{x86}$ can take either a program transition staying on the current CPU, or a hardware scheduling to another CPU. The *behavior* of a client program P over this multicore machine (denoted as $[\![P]\!]_{\mathbb{M}_{x86}}$) is a set of global logs generated by executing P via these two kinds of transitions.

(Id, Loc)	c, b	$\in$	Nat	$(Bytes)$	bl	$\in$	$List\ Byte$	(Val)	v	$:=$	$bl \mid b \mid \mathsf{vundef}$
(Mem)	pm, m	$\in$	$Loc \to Val$	(Reg)	r	$:=$	$\mathsf{EIP} \mid \mathsf{EAX} \mid \cdots$	$(RegSet)$	rs	$\in$	$Reg \to Val$
$(PvtSt)$	ρ	$:=$	(pm, rs)	$(PvtStMap)$	f_ρ	$\in$	$Id \to PvtSt$	(Abs)	a	$\in$	$Type$
$(Event)$	e	$:=$	$\epsilon \mid c.\mathsf{push}(b, v) \mid \cdots$	(Log)	l	$\in$	$List\ Event$	$(State)$	s	$:=$	(c, f_ρ, m, a, l)
$(AsmFn)$	κ_{x86}	$\in$	$List\ x86Instr$	$(AsmModule)$	M_{x86}	$\in$	$Loc \to AsmFn$				
$(Prim)$	σ	$\in$	$State \to List\ Val \to State \to Val \to Prop$								
$(PrimList)$	$\mathcal{L}$	$\in$	$Loc \to Prim$	$(Layer)$	$L[A]$	$:=$	$(\mathcal{L}, \mathcal{R}, \mathcal{G})$	(Inv)	INV	$\in$	$Log \to Prop$
$(Rely, Guar)$	$\mathcal{R}, \mathcal{G}$	$\in$	$Id \rightharpoonup Inv$	$(Strategy)$	φ	$\in$	$Log \rightharpoonup Log$	(EC)	$\mathcal{E}$	$\in$	$Id \rightharpoonup Strategy$

Figure 7. The machine state for the concurrent machine model and the concurrent layer interface.

Memory Model. We introduce a "push/pull" memory model for the shared memory m (the private memory is separately handled in ρ), which encapsulates the shared memory operations into push/pull events and can detect data races.

In this model, each shared memory location b is associated with an ownership status in the abstract state a, which can only be manipulated by two shared primitives called pull and push. The pull operation modifies the ownership from "free" to "owned by c", after which shared memory accesses can be performed by CPU c. The push operation frees the ownership and records its memory updates in the log. Figure 6 shows the specification σ'_{pull}, where "$r\ \{i : v\}$" means updating the record r at field i with value v.

If a program tries to pull a not-free location, or tries to access or push to a location not owned by the current CPU, a *data race* may occur and the machine gets *stuck*. One goal of concurrent program verification is to show that a program is data-race free; in our setting, we accomplish this by showing that the program does not get stuck.

3.2 Concurrent Layer Interface

We now zoom in on the execution of a subset of CPUs A, introducing the *concurrent layer interface* $L[A]$ defined as a tuple $(\mathcal{L}, \mathcal{R}, \mathcal{G})$. The machine based on this concurrent interface is "open" in the sense that it is eligible to capture a subset of the CPUs and then be composed with any acceptable execution of the rest of CPUs. The domain of the private state map f_ρ is also this captured (or focused) subset. The interface $L[A]$ equips this open machine with a collection of primitives that are defined in $\mathcal{L}$ and can be invoked at this level, the rely condition $\mathcal{R}$ that specifies a set of acceptable environment contexts, and the guarantee condition $\mathcal{G}$ that the log l should hold. The instruction transitions are defined as before, but all hardware scheduling is replaced by queries to the environment context.

Environment Context. $\mathcal{E}$ is a partial function from a CPU ID to its *strategy* φ. A strategy is an automata that generates events in response to given logs. When focusing on a CPU set A, all the observable behaviors of the hardware scheduling and the program transitions of other CPUs can be specified as a union of strategies (i.e., $\mathcal{E}$). Thus, whenever there is a potential interleaving, the machine can *query* $\mathcal{E}$ about the events from other CPUs (and the scheduler).

These environmental events cannot influence the behaviors of instructions and private primitive calls. This also applies to shared memory read/write, because the push/pull memory model encapsulates other CPUs' effects over the shared memory into push/pull events. Thus, during the execution of instructions and private primitives, it is unnecessary to query $\mathcal{E}$, and the appended environmental events will be received by the next so-called *query point*, that is, the point just before executing shared primitives.

To be more specific, at each query point, the machine repeatedly queries $\mathcal{E}$. Each query takes the current log l as the argument and returns an event (i.e., "$\mathcal{E}(c', l)$") from a CPU c' not in A. That event is then appended to l, and this querying continues until there is a hardware transition event back to A (assuming the hardware scheduler is fair). In the following, we write "$\mathcal{E}[A, l]$" to mean this entire process of extending l with multiple events from other CPUs.

Rely and Guarantee Conditions. The $\mathcal{R}$ and $\mathcal{G}$ of the layer interface specify the validity of the environment context and the invariant of the log (containing the locally-generated events). After each step of threads in A over interface $L[A]$, the resulting log l must satisfy the guarantee condition $L[A].\mathcal{G}$, i.e., $l \in L[A].\mathcal{G}(c)$ if c is the current CPU ID indicated by l. To prove that guarantee conditions always hold, we not only need to validate the events generated locally but also need to rely on the validity of the environment context. The rely condition $L[A].\mathcal{R}$ specifies a set of valid environment contexts, which take valid input logs and return a valid list of events.

CPU-Local Layer Interface. When focusing on a single CPU c, $L[c]$ is called a CPU-local layer interface. Its machine state is (ρ, m, a, l), where ρ is the private state of the CPU c and m is just a *local copy* of the shared memory.

This m can only be accessed locally by c. The primitives push/pull of $L[c]$ "deliver" the effects of shared memory operations. Figure 8 shows their specifications, which depend on a replay function $\mathbb{R}_{\mathsf{shared}}$ to reconstruct the shared memory value v for some location b and check the well-formedness (i.e., no data race occurs) of the resulting log.

Since σ_{pull} is parameterized with $\mathcal{E}$, it can also be viewed as the following special strategy with private state updates:

```
1  Fixpoint ℝ_shared (l: Log) (b: Loc) (c: Id) :=
2    match l with
3      | nil => ret (vundef, free) (* initial value *)
4      | e :: l' =>   (* l' • a *)
5        do r <- ℝ_shared l' b; (* Haskell syntax sugar*)
6          match r, e with
7            | (v, free), c.pull(b) => ret (v, own c)
8            | (_, own c), c.push(b,v) => ret (v, free)
9            | _ => None (* get stuck *)
10         end
11   end.
12 Function σ_pull 𝓔 (s: State) (b: Loc) :=
13   let l':= s.c.pull(b) :: 𝓔[s.c, s.l] in (* query 𝓔 *)
14   do r <- ℝ_shared l' b s.c; ret s {l: l'} {m.b: fst r}.
15 Function σ_push 𝓔 (s: State) (b: Loc) :=
16   let l':= s.c.push(b,s.m b) :: s.l in (*do not query 𝓔*)
17   do _ <- ℝ_shared l' b s.c; ret s {l: l'}.
```

Figure 8. Pseudocode of push/pull specifications of $L[c]$ in Coq.

$$?\mathcal{E}, !c.\text{pull}(b), \downarrow(v, \text{own } c)$$
$$\{m.b : v\}$$
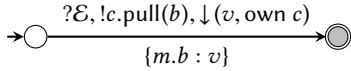
The layer machine enters the *critical state* after calling pull by holding the ownership of a shared location. It exits the critical state by invoking push to free the ownership.

3.3 Concurrent Layer Calculus

To build and compose concurrent layers "$L[A] \vdash_R M : L'[A]$," we introduce a layer calculus shown in Fig. 9. We borrow the notations from Gu et al. [15]: "$\varnothing$" stands for an empty program module, "$\oplus$" computes the union of two modules (or two layers' primitive collections), and "$(i \mapsto \cdot)$" is a singleton map with a pointer or location i as its domain.

Composition Rules. The vertical composition rule (Vcomp) allows us to verify the modules M and N (where N may depend on M) in two separate steps, while the horizontal composition rule (Hcomp) enables local reasoning for independent modules M and N belonging to the same level. These two composition rules can only compose layers over the same CPU set.

Layers on different CPUs can be composed by the parallel composition rule (Pcomp) if simulation relations are the same, and both overlay and underlay interfaces are *compatible*. This compatibility is denoted as "compat($L[A], L[B], L[A \cup B]$)." It says that each guarantee condition of $L[A]$ implies the corresponding rely condition of $L[B]$ and vice versa. The composed interface $L[A \cup B]$ merges the primitives of two layers and is equipped with stronger guarantees and weaker rely conditions. The machine based on this composed layer interface only queries $\mathcal{E}$ about the events not from $A \cup B$.

Multicore Linking Theorem. By composing all the CPUs in the machine (denoted as the set D), the resulting layer interface does not depend on any environmental events except those from the hardware scheduler. We construct such a layer interface $L_{\text{x86}}[D]$ using the primitives provided by the hardware $\mathbb{M}_{\text{x86}}$. We can then prove a contextual refinement from $\mathbb{M}_{\text{x86}}$ to $L_{\text{x86}}[D]$ by picking a suitable hardware scheduler of $L_{\text{x86}}[D]$ for every interleaving (or log) of $\mathbb{M}_{\text{x86}}$.

Theorem 3.1 (Multicore Linking).

$$\forall P, [\![P]\!]_{\mathbb{M}_{\text{x86}}} \sqsubseteq_R [\![P]\!]_{L_{\text{x86}}[D]}$$

This theorem ensures that all code verification over $L_{\text{x86}}[D]$ can be propagated down to the x86 multicore hardware $\mathbb{M}_{\text{x86}}$.

Building Leaf Certified Layers. As the unit of certified concurrent layers, leaf layers can be built by applying the Fun rule, which requires to prove the strategy simulation. Two most common patterns, *fun-lift* and *log-lift*, for this proof have already been shown in Sec. 2. The fun-lift pattern abstracts a concrete implementation into a low-level strategy without changing the potential interleaving. In this pattern, language dependent details (e.g., silent moves changing temporal variables) are hidden and data representation details (e.g., memory values carried by push events) are replaced with abstract state values.

The log-lift pattern always involves the events merging and the interleavings shuffling to form an atomic interface.

4 Building Certified Multicore Layers

In this section, we start to show how to apply our techniques to verify shared objects in the CCAL toolkit. All layers are built upon the CPU-local layer interface $L_{\text{x86}}[c]$.

4.1 Spinlocks

Spinlocks (e.g., the ticket lock algorithm described in Sec. 2) are one of the most basic synchronization methods for multicore machines; they are used as building blocks for shared objects and more sophisticated synchronizations.

A spinlock enforces mutual exclusion by restricting CPU access to a memory location b. Therefore, lock operations can be viewed as "safe" versions of push/pull primitives. For example, when the lock acquire for b succeeds, the corresponding shared memory is guaranteed to be "free", meaning that it is safe to pull the contents to the local copy at this point (line 4 in Fig. 10). We now show how to build layers for the spinlock in Fig. 10, which uses a ticket lock algorithm. Note that query points are denoted as "▷" in pseudocode.

Bottom Interface $L_{\text{x86}}[c]$. We begin with the CPU-local interface $L_{\text{x86}}[c]$ extended with shared primitives FAI_t, get_n, and inc_n. These primitives directly manipulate the lock state t (next ticket) and n (now serving ticket) via x86 atomic instructions. The lock state can be calculated by a replay function $\mathbb{R}_{\text{ticket}}$ counting c.FAI_t and c.inc_n events.

Fun-Lift to $L_{\text{lock\_low}}[c]$. We have shown how to establish the strategy simulation for this low-level interface $L_{\text{lock\_low}}[c]$

$$\frac{}{L[A] \vdash_{\mathbf{id}} \varnothing : L[A]} \text{ EMPTY} \qquad \frac{(\![\kappa]\!)_{L[c]} \leq_R \sigma}{L[c] \vdash_{\mathbf{id}} i \mapsto \kappa : i \mapsto \sigma} \text{ FUN} \qquad \frac{L_1[A] \vdash_R M : L_2[A] \qquad L_2[A] \vdash_S N : L_3[A]}{L_1[A] \vdash_{R \circ S} M \oplus N : L_3[A]} \text{ VCOMP}$$

$$\frac{\begin{array}{c} L[A] \vdash_R M : L_1[A] \qquad L[A] \vdash_R N : L_2[A] \\ L'[A].\mathcal{L} = L_1[A].\mathcal{L} \oplus \mathrm{tt}_2[A].\mathcal{L} \qquad L'[A].\mathcal{R} = L_1[A].\mathcal{R} = \mathrm{tt}_2[A].\mathcal{R} \qquad L'[A].\mathcal{G} = L_1[A].\mathcal{G} = \mathrm{tt}_2[A].\mathcal{G} \end{array}}{L[A] \vdash_R M \oplus N : L'[A]} \text{ HCOMP}$$

$$\frac{L'_1[A] \leq_R L_1[A] \qquad L_1[A] \vdash_S M : L_2[A] \qquad L_2[A] \leq_T L'_2[A]}{L'_1[A] \vdash_{R \circ S \circ T} M : L'_2[A]} \text{ WK}$$

$$\frac{\begin{array}{c} A \perp B \qquad \forall i \in A, L[B].\mathcal{R}(i) \subseteq L[A].\mathcal{G}(i) \qquad \forall i \in B, L[A].\mathcal{R}(i) \subseteq L[B].\mathcal{G}(i) \\ L[A \cup B].\mathcal{L} = L[A].\mathcal{L} = L[B].\mathcal{L} \qquad L[A \cup B].\mathcal{R} = L[A].\mathcal{R} \cap L[B].\mathcal{R} \qquad L[A \cup B].\mathcal{G} = L[A].\mathcal{G} \cup L[B].\mathcal{G} \end{array}}{\mathrm{compat}(L[A], L[B], L[A \cup B])} \text{ COMPAT}$$

$$\frac{L_1[A] \vdash_R M : L_2[A] \qquad L_1[B] \vdash_R M : L_2[B] \qquad \mathrm{compat}(L_1[A], L_1[B], L_1[A \cup B]) \qquad \mathrm{compat}(L_2[A], L_2[B], L_2[A \cup B])}{L_1[A \cup B] \vdash_R M : L_2[A \cup B]} \text{ PCOMP}$$

Figure 9. The fine-grained layer calculus in the concurrent setting.

```
1 void acq (uint b) {
2   uint myt=▷FAI_t(b);
3   while(▷get_n(b)!=myt){}
4   ▷pull(b);//acts as hold()
5 }
```

```
6 void rel (uint b) {
7   push(b);
8   ▷inc_n(b);
9 }
```

Figure 10. Pseudocode of ticket lock using push/pull.

(i.e., $L'_1[c]$, see Sec. 2). Note that $(\![\mathrm{acq}]\!)_{L_{\mathrm{lock\_low}}[c]}$ contains extra silent moves (e.g., assigning myt, line 2 in Fig. 10) compared with $\varphi'_{\mathrm{acq}}[c]$. The simulation relation R_{lock} not only states the equality between logs but also maps the lock state in the memory to the ones calculated by $\mathbb{R}_{\mathrm{ticket}}$. Here we must also handle potential *integer overflows* for t and n. We can prove that, as long as the total number of CPUs (i.e., #CPU) in the machine is less than 2^{32} (determined by uint), the mutual exclusion property will not be violated even with overflows.

Log-Lift to $L_{\mathrm{lock}}[c]$***.*** We then lift the acq and rel primitives to an atomic interface, meaning that each invocation produces exactly one event in the log (see Sec. 2). These atomic lock interfaces (or strategies) are similar to pull/push specifications, except that the former ones are *safe* (i.e., will not get stuck). This safety property can be proved using rely conditions $L_{\mathrm{lock}}[c].\mathcal{R}$ saying that, for any CPU $c' \neq c$, its c'.acq event must be followed by a sequence of its own events (generated in the critical state) ending with c'.rel. The distance between c'.acq and c'.rel in the log is less than some number n.

By enforcing the fairness of the scheduler in rely conditions, saying that any CPU can be scheduled within m steps, we can show the liveness property (i.e., starvation-freedom): the while-loop in acq terminates in "$n \times m \times$ #CPU" steps.

4.2 Shared Queue Object

Shared queues are widely used in concurrent programs, e.g., as the list of threads in a scheduler, etc. In previous work [30],
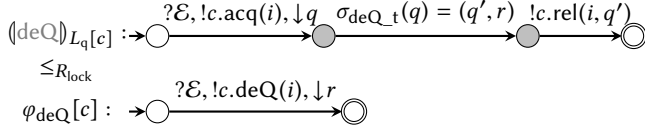
due to the lack of layering support, the verification of any shared object required inlining the lock implementation and duplicating the lock-related proofs. In the following, we illustrate how to utilize concurrent abstraction layers to verify a shared queue module using fine-grained locks.

Fun-Lift to $L_{\mathrm{q}}[c]$***.*** The shared queues are implemented as doubly linked lists, and are protected by spinlocks. For example, the dequeue (deQ) operation first acquires the spinlock associated with queue i, then performs the actual dequeue operation in the critical state, and finally releases the lock. Instead of directly verifying deQ in one shot, we first introduce an intermediate function deQ_t, which contains code that performs the dequeue operation over a local copy, under the assumption that the corresponding lock is held. Since no environmental queries are needed in the critical state, building concurrent layers for deQ_t is similar to building a sequential layer [15]: we first introduce the abstract states a.tcbp and a.tdqp, which stand for the thread control block (i.e., tcb) array and the thread queue array. The abstract tdqp is a partial map from the *queue index* to an *abstract queue*, which is represented as a list of tcb indices. Then we can show that deQ_t meets its specification $\sigma_{\mathrm{deQ\_t}}$:

```
1 Function σdeQ_t ℰ (s: State) (i: Loc) :=
2   do r <- ℝ'shared s.l i s.c; (* replay ownership *)
3   match r with
4   | (_, own s.c) => (*if the lock of queue i is held*)
5     match s.a.tdqp i with (* case over the queue*)
6       | td :: q => ret (s {s.a.tdqp.i: q}, td}
7       | _ => ret (s, -1) (* return -1 for empty queue*)
8     end
9   | _ => None (*get stuck*)
10  end.
```

Fun- and Log-Lift to $L_{\text{q\_high}}[c]$. Finally, we have to show that the deQ function that wraps deQ_t with lock primitives indeed meets an atomic interface. With a simulation relation R_{lock} that merges two queue-related lock events (i.e., c.acq and c.rel) into a single event c.deQ at the higher layer, we can prove the following strategy simulation:

$$
\begin{array}{c}
(\!|\text{deQ}|\!)_{L_q[c]} : \xrightarrow{\quad} \bigcirc \xrightarrow{?\mathcal{E},\,!c.\text{acq}(i),\,\downarrow q} \bullet \xrightarrow{\sigma_{\text{deQ\_t}}(q) = (q',r)} \bullet \xrightarrow{!c.\text{rel}(i,q')} \circledcirc \\[2pt]
\leq_{R_{\text{lock}}} \\[2pt]
\varphi_{\text{deQ}}[c] : \xrightarrow{\quad} \bigcirc \xrightarrow{\quad ?\mathcal{E},\,!c.\text{deQ}(i),\,\downarrow r \quad} \circledcirc
\end{array}
$$

# 5 Building Certified Multithreaded Layers

Multithreaded programs have to deal with interleavings triggered by not only the hardware scheduler but also the explicit invocation of thread scheduling primitives. In this section, we introduce the certified layers dealing with scheduling primitives, a new concept of thread-local layer interfaces equipped with compositional rules, and a thread-safe version of CompCertX.

5.1 Certified Layers for Scheduling Primitives

Based on the shared thread queues provided by the multicore toolkit (see Sec. 4.2), we introduce a new layer interface $L_{\text{btd}}[c]$ that supports multithreading. At this layer interface, the transitions between threads are done using scheduling primitives, implemented in a mix of C and assembly.

In our multithreaded setting, each CPU c has a private ready queue rdq and a shared pending queue pendq (containing the threads woken up by other CPUs). A thread yield sends the first pending thread from pendq to rdq and then switches to the next ready thread. There are also many shared sleeping queues slpq. When a sleeping thread is woken up, it will be directly appended to the ready queue if the thread belongs to the currently-running CPU. Otherwise, it will be appended to the pending queue of the CPU it belongs to.

Thread switching is implemented by the context switch function cswitch, which saves the current thread's kernel context (i.e., ra, ebp, ebx, esi, edi, esp), and loads the context of the target thread. This cswitch (invoked by yield and sleep) can only be implemented at the assembly level, as it does not satisfy the C calling convention. A scheduling primitive like yield first queries $\mathcal{E}$ to update the log, appends its own event, and then invokes cswitch to transfer the control.
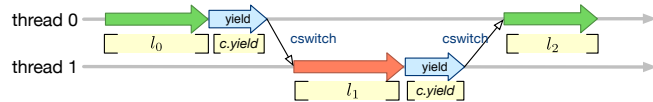

This layer interface introduces three new events c.yield, c.sleep(i, lk) (sleep on queue i while holding the lock lk), and c.wakeup(i) (wakeup the queue i). These events record the thread switches, which can be used to track the currently-running thread by a replay function $\mathbb{R}_{\text{sched}}$.

5.2 Multithreaded Layer Interface

The CPU-local interface $L_{\text{btd}}[c]$ captures the execution of the whole thread set of CPU c and does not support thread-local verification. Ideally, we would like to formally reason about each thread separately and later compose the proofs together obtaining a global property. Thus, we introduce a new layer interface that is compositional and only focuses on a subset of thread running on CPU c.

Let T_c denote the whole thread set running over CPU c. Based upon $L[c]$, we construct a *multithreaded* layer interface "$L[c][T_a] := (L[c].\mathcal{L}, L[c].\mathcal{R} \cup \mathcal{R}_{T_a}, L[c].\mathcal{G}_{|T_a})$," which is parameterized over a focused thread set $T_a \subseteq T_c$. Besides T_q, strategies of other threads running on c form a thread context $\mathcal{E}^t$. Rely conditions of this multithreaded layer interface extend $L[c].\mathcal{R}$ with a *valid set* of $\mathcal{E}^t$ (denoted as "$\mathcal{R}_{T_a}$") and guarantee conditions replace $L[c].\mathcal{G}(c)$ with the invariants held by threads in T_a (denoted as "$L[c].\mathcal{G}_{|T_a}$"). Since our machine model does not allow preemption, $\mathcal{E}^t$ will only be queried during the execution of scheduling primitives, which have two kinds of behaviors depending on whether the *target thread* is focused or not.
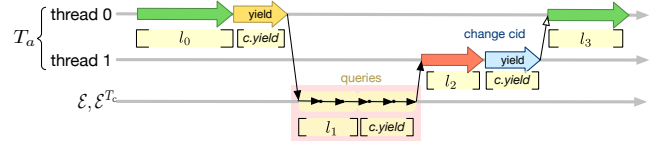
Consider the above execution with $T_a = \{0, 1\}$. Whenever an execution switches (by yield or sleep) to a thread outside of T_a (i.e., the yellow yield above), it takes environmental steps (i.e., inside the red box), repeatedly appending the events returned by the environment context $\mathcal{E}$ and the thread context $\mathcal{E}^t$ to the log until a c.yield event indicates that the control has switched back to a focused thread. Whenever an execution switches to a focused one (i.e., the blue yield above), it will perform the context switch without asking $\mathcal{E}/\mathcal{E}^t$ and its behavior is identical to the one of $L_{\text{btd}}[c]$.

Composing Multithreaded Layers. Multithreaded layer interfaces with disjoint focused thread sets can also be composed in parallel (using an extended Pᴄᴏᴍᴘ rule) if the guarantee condition implies the rely condition for every thread. The resulting focused thread set is the union of the composed ones, and some environmental steps are "replaced by" the local steps of the other thread set. For example, if we compose T_a in the above example with thread 2, the previously yellow yield of thread 0 will then switch to a focused thread.
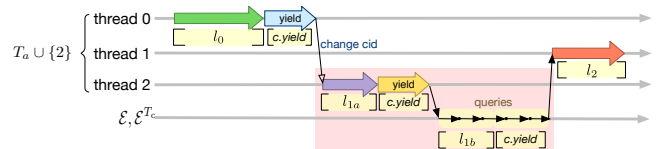
```
1 void acq_q(uint l) {                    8    }
2   ▷ acq(ql_loc(l));                      9  }
3   if (ql_busy[l] != -1) {              10  void rel_q (uint l) {
4     ▷ sleep(l);                        11    ▷ acq(ql_loc(l));
5   } else {                             12    ql_busy[l] =▷ wakeup(l);
6     ql_busy[l] = get_tid();            13    ▷ rel(ql_loc(l));
7     ▷ rel(ql_loc(l));                  14  }
```

Figure 11. Pseudocode of queuing lock.

Here, the event list l_1 generated by $\mathcal{E}$ and $\mathcal{E}^t$ has been divided into two parts: "$l_{1a} \bullet c.\text{yield}$" (generated by thread 2) and l_{1b} (consisting of events from threads outside $\{0,1,2\}$).

Multithreaded Linking. When the whole T_c is focused, all scheduling primitives fall into the second case and never switch to unfocused ones. Thus, its scheduling behaviors are equal to the ones of $L_{\text{btd}}[c]$. By introducing a multithreaded layer interface $L_{\text{htd}}[c][T_c]$ that contains all the primitives of $L_{\text{btd}}[c]$, we can prove the following theorem:

Theorem 5.1 (Multithreaded Linking).

$$L_{\text{btd}}[c] \leq_{\textbf{id}} L_{\text{htd}}[c][T_c]$$

This theorem guarantees that, once the multithreaded machine based on $L_{\text{htd}}[c][T_c]$ captures the whole thread set, the properties of threads running on top can be propagated down to the layer with concrete scheduling implementations.

5.3 Thread-Local Layer Interface

If a multithreaded interface $L[c][t]$ focus only on a single thread $t \in T_c$, yield and sleep primitives always switch to an unfocused thread and then repeatedly query $\mathcal{E}$ and $\mathcal{E}^t$ until yielding back to t.
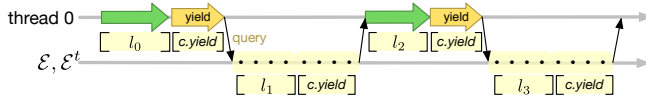
We can prove that this yielding back procedure in our system always terminates. This proof relies on the fact that the software scheduler is *fair* and every running thread gives up the CPU within a finite number of steps. We call $L[c][t]$ a "thread-local" layer interface because scheduling primitives always end up switching back to the same thread; they do not modify the kernel context (i.e., ra, ebp, ebx, esi, edi, esp) and effectively act as a "no-op", except that the shared log gets updated. Thus, these scheduling primitives indeed satisfy C calling conventions.

5.4 Queuing Lock

Based upon thread-local layer interfaces, we build additional synchronization toolkits, such as a queuing lock (see Fig. 11). With queuing locks, waiting threads are put to sleep to avoid busy spinning. Reasoning about this locking algorithm is

particularly challenging since its C implementation utilizes both spinlocks and low-level scheduler primitives (i.e., sleep and wakeup). This verification task can be decomposed into a bunch of layers above $L_{\text{htd}}[c][t]$ using CCAL.

The correctness property of a queuing lock consists of two parts: mutual exclusion and starvation freedom. The lock implementation (Fig. 11) is mutually exclusive because the busy value of the lock (ql_busy) is always equal to the lock holder's thread ID. This busy value is set either by the lock requester when the lock is free (line 6 of Fig. 11) or by the previous lock holder when releasing the lock (line 12). With the atomic interface of the spinlock, the starvation-freedom proof of queuing lock is mainly about the termination of the sleep primitive call (line 4). By showing that all the lock holders will eventually release the lock, we prove that all the sleeping threads will be added to the pending queue or ready queue within a finite number of steps. Thus, sleep will terminate thanks to the *fair* software scheduler. Note that all these properties proved at the C level can be propagated down to the assembly level using the thread-safe CompCertX.

5.5 Thread-Safe Compilation and Linking

In this section, we show how to adapt Gu et al.'s CompCertX verified separate compiler [15, §6] to handle programs that call scheduling primitives. Section 5.3 shows how thread-local layer interfaces allow us to give *C style* specifications to scheduling primitives (yield and sleep) which are partly implemented in assembly. Thus, code of each thread can be verified at the C level over $L_{\text{htd}}[c][t]$ and individual threads can then be composed into programs on $L_{\text{btd}}[c]$ by Thm. 5.1. However, it is still challenging to show that the compiled programs at the assembly level are also compatible with this parallel composition because of a small snag which we glossed over until now: *stack frames*. In the CompCert memory model [28], whenever a function is called, a fresh memory block has to be allocated in the memory for its stack frame. This means that, on top of the thread-local layer $L_{\text{htd}}[c][t]$, a function called within a thread will allocate its stack frame into the thread-private memory state, and conversely, a thread is never aware of any newer memory blocks allocated by other threads. In comparison, on top of the CPU-local layer $L_{\text{btd}}[c]$, all stack frames have to be allocated in the CPU-local memory (i.e., *thread-shared* memory) regardless of which thread they belong to; thus, in the thread composition proof, we need to account for all such stack frames.

Our solution is based on a special *memory extension* [28, §5.2] that only removes the access permissions of some memory blocks. To enable the thread composition, we extended the semantics of yield and sleep on the thread-local layer $L_{\text{htd}}[c][t]$. Besides generating a $c.\text{yield}/c.\text{sleep}$ event, such a scheduling primitive also allocates *empty* memory blocks as "placeholders" for other threads' new stack frames during this yield/sleep. These empty blocks are the ones without

$$\frac{m_1 \circledast m_2 \simeq m}{\mathsf{nb}(m) = \max(\mathsf{nb}(m_1), \mathsf{nb}(m_2))}\ (\textsc{Nb}) \qquad \frac{m_1 \circledast m_2 \simeq m}{m_2 \circledast m_1 \simeq m}\ (\textsc{Comm}) \qquad \frac{m_1 \circledast m_2 \simeq m \quad \mathsf{ld}(m_2, \ell) = \lfloor v \rfloor}{\mathsf{ld}(m, \ell) = \lfloor v \rfloor}\ (\textsc{Ld})$$

$$\frac{m_1 \circledast m_2 \simeq m}{m_1 \circledast \mathsf{st}(m_2, \ell, v) \simeq \mathsf{st}(m, \ell, v)}\ (\textsc{St}) \qquad \frac{m_1 \circledast m_2 \simeq m \quad \mathsf{nb}(m_1) \le \mathsf{nb}(m_2)}{m_1 \circledast \mathsf{alloc}(m_2, l, h) \simeq \mathsf{alloc}(m, l, h)}\ (\textsc{Alloc}) \qquad \frac{m_1 \circledast m_2 \simeq m \quad \mathsf{nb}(m_1) \le \mathsf{nb}(m_2)}{m_1 \circledast \mathsf{liftnb}(m_2, n) \simeq \mathsf{liftnb}(m, n)}\ (\textsc{Lift-R})$$

$$\frac{m_1 \circledast m_2 \simeq m \quad \mathsf{nb}(m_1) \le \mathsf{nb}(m_2)}{\mathsf{liftnb}(m_1, n) \circledast m_2 \simeq \mathsf{liftnb}(m, n - (\mathsf{nb}(m) - \mathsf{nb}(m_1)))}\ (\textsc{Lift-L})$$

Figure 12. Algebraic memory model

any access permissions. We write "$\mathsf{nb}(m)$" to denote the total number of blocks in m, and write "$\mathsf{liftnb}(m, n)$" as the memory extended from m by allocating n new empty blocks.

With the extended semantics for scheduling primitives, we can prove that a ternary relation "$m_1 \circledast m_2 \simeq m$" holds between the private memory states m_1, m_2 of two disjoint thread sets and the thread-shared memory state m after the parallel composition. This relation among memory states is called the "algebraic memory model", which is defined by the axioms shown in Fig. 12.

Rule N$\textsc{b}$ states that the block number of the composed memory m is equal to "$\max(\mathsf{nb}(m_1), \mathsf{nb}(m_2))$." Rule C$\textsc{omm}$ says that the parallel memory composition is commutative. Rule L$\textsc{d}$ and S$\textsc{t}$ state that the behaviors of memory load and store (over m_1 or m_2) are preserved by the composed memory m. It is because that every non-shared memory block of m_1 either does not exist in m_2 or corresponds to an empty block in m_2, and vice versa.

All the remaining rules in Fig. 12 share the condition "$\mathsf{nb}(m_1) \le \mathsf{nb}(m_2)$." This condition indicates that thread 2 is "more-recently scheduled/running" because only *running* thread can allocate memory blocks. Thus, memory allocations on m_2 can be preserved by the composed memory m (see Rule A$\textsc{lloc}$). In addition, if thread 2 is still the next scheduled thread and there are n new stack frames allocated by threads other than $\{1, 2\}$, we can then simply allocate n empty blocks in m_2, which will be preserved by m (see Rule L$\textsc{ift}$-R). If thread 1 is the next thread to run, after allocating n new empty blocks to m_1, the composed memory m only need to allocate the blocks that have not been captured by m_2 (see Rule L$\textsc{ift}$-L).

Based on the parallel composition for two memory states, we can use Rule L$\textsc{ift}$-R and L$\textsc{ift}$-L to generalize to N threads by saying that m is a composition of the private memory states "$m_1, \dots, m_N$" of N threads (on a single processor) if, and only if, there exists a memory state m' such that m' is a composition of "$m_1, \dots, m_{N-1}$" and $m_N \circledast m' \simeq m$ holds.

6 Evaluation and Experience

We have implemented the CCAL toolkit (see Fig. 2) in the Coq proof assistant. Table 1 presents the number of lines

Component	LOC	Component	LOC
Auxiliary library	6,200	Multilayer linking	17,000
C verifier	2,200	Multithread linking	10,000
Asm verifier	800	Multicore linking	7,000
Simulation library	1,800	Thread-safe CompCertX	7,500

Table 1. Lines of proofs in Coq for the toolkit.

	C&Asm Source	Spec.	Invariant Proof	C & Asm Proof	Simulation Proof
Ticket lock	74	615	1,080	1,173	2,296
MCS lock	287	1,569	2,299	1,899	3,049
Local queue	377	554	748	2,821	3,647
Shared queue	20	107	190	171	419
Scheduler	62	153	166	1,724	2,042
Queuing lock	112	255	992	328	464

Table 2. Statistics for implemented components.

(in Coq) for each component in Fig. 2. The auxiliary library contains the common tactics and lemmas for 64 bit integers, lists, maps, integer arithmetic, etc.

Case Studies. To evaluate the framework itself, we have implemented, specified, and verified various concurrent programs in the framework. Table 2 presents some of the statistics with respect to the implemented components. As for lock implementations, their source code contains not only the code of the associated functions, but also the data structures and their initialization. In addition to the top-level interface, the specification contains all the specifications used in the intermediate layers. For both the ticket and MCS locks, the simulation proof column includes the proof of starvation freedom (about 1,500 lines) in addition to the correctness proof. The gap between the underlying C implementation and the high-level specification of the locks also contributes to the large proof size for these components. For example, intermediate specification of the ticket lock uses an unbounded integer for the ticket field, while the implementation uses a binary integer which wraps back to zero. Similarly, the queue is represented as a logical list in the specification, while it is implemented as a doubly linked list.

Our development is compositional. Both ticket and MCS locks share the same high-level atomic specifications (or strategies) shown in Sec. 2. Thus the lock implementations can be freely interchanged without affecting any proof in the higher-level modules using locks. When implementing the shared queue library, we also reuse the implementation and proof of the local (or sequential) queue library: to implement the atomic queue object, we simply wrap the local queue operations with lock acquire and release statements. As shown in Table 2, using verified lock modules to build atomic objects such as shared queues is relatively simple and does not require many lines of code.

Following the same philosophy, Gu et al. [16] has further extended our work with paging-based dynamically allocated virtual memory, device drivers with in-kernel interrupts, a synchronous inter-process communication (IPC) protocol using the queuing lock, a shared-memory IPC protocol with a shared page, and Intel hardware virtualization support; our CCAL toolkit was used to produce the world's first fully certified concurrent OS kernel with fine-grained locking.

Performance Evaluation. We have measured the performance of the ticket lock on an Intel 4-Core i7-2600S (2.8GHz) processor with 16GB memory. Initially, the ticket lock implementation incurred a latency of 87 CPU cycles in the single core case. After a short investigation, we found that we forgot to remove some function calls to "logical primitives" used for manipulating ghost abstract states. After we removed these extra null calls, the latency dropped down to only 35 CPU cycles. Gu et al. [16] also presented performance evaluations of their OS kernel built using CCAL.

Limitations. Our concurrent machine models assume strong sequential consistency (SC) for atomic primitives. Previous work [52] demonstrated that race-free programs on a TSO model do indeed behave as if executing on a sequentially consistent machine. Since safe programs on our push/pull model are race-free, we believe extending our work from SC to TSO is promising. In our future work, we will formalize and integrate this proof in Coq. Furthermore, the current event-based contextual refinement proofs still require quite a bit of manual proof. We are working on developing more automation tactics to further cut down the proof effort. In addition to this general toolkit that can support a broad range of concurrent programs, we also plan to provide more aggressive automation for commonly-used concurrent programming patterns, either through additional tactic libraries or using specific program logics targeting such patterns.

7 Related Work and Conclusions

Certified Abstraction Layers. Gu et al. [15] presented the first formal account of certified abstraction layers and showed how to apply layer-based techniques to build certified system software. The layer-based approach differs from Hoare-style program verification [4, 21, 40, 46] in several significant ways. First, it uses the termination-sensitive forward simulation techniques [26, 35] and proves a stronger contextual correctness property rather than simple partial or total correctness properties (as done for Hoare logics). Second, the overlay interface of a certified layer object completely removes the internal concrete memory block (for the object) and replaces it with an abstract state suitable for reasoning; this abstract state differs from auxiliary or ghost states (in Hoare logic) because it is actually used to define the semantics of the overlay abstract machine and the corresponding contextual refinement property. Third, as we move up the abstraction hierarchy by composing more layers, each layer interface provides a new programming language that gets closer to the specification language—it can call primitives at higher abstraction levels while still supporting general-purpose programming in C and assembly.

Our CCAL toolkit follows the same layer-based methodologies. Each time we introduce a new concrete concurrent object implementation, we replace it with an abstract atomic object in its overlay interface. All shared abstract states are represented as a single global log, so the semantics of each atomic method call would need to *replay* the entire global log to find out the return value. This seemingly "inefficient" way of treating shared atomic objects is actually great for compositional specification. Indeed, it allows us to apply game-semantic ideas and define a general semantics that supports parallel layer composition.

Abstraction for Concurrent Objects. Herlihy and Wing [20] introduced *linearizability* as a key technique for building abstraction over concurrent objects. Developing concurrent software using a stack of shared atomic objects has since become the best practice in the system community [2, 19]. Linearizability is quite difficult to reason about, and it is not until 20 years later that Filipovic et al. [10] showed that linearizability is actually equivalent to a termination-insensitive version of the contextual refinement property. Gotsman and Yang [14] showed that such equivalence also holds for concurrent languages with ownership transfers [42]. Liang et al. [30, 33] showed that linearizability plus various progress properties [19] for concurrent objects is equivalent to various termination-sensitive versions of the contextual refinement property. These results convinced us that we should prove termination-sensitive (contextual) simulation when building certified concurrent layers as well.

RGSim and LiLi. Building contextual refinement proofs for concurrent programs (and program transformations) is challenging. Liang et al. [30−32] developed the Rely-Guarantee-based Simulation (RGSim) that can support both parallel composition and contextual refinement of concurrent objects. Our contextual simulation proofs between two concurrent layers can be viewed as an instance of RGSim if we

extend RGSim with auxiliary states such as environment contexts and shared logs. This extension, of course, is the main innovation of our new compositional layered model. Also, all existing RGSim systems are limited to reasoning about atomic objects at one layer; their client program context cannot be the method body of another concurrent object, so they cannot support the same general vertical layer composition as our work does.

Treatment of Parallel Composition. Most concurrent languages (including those used by RGSim) use a parallel composition command ($C_1 \| C_2$) to create and terminate new threads. In contrast, we provide thread spawn and join primitives, and assign every new thread a unique ID (e.g., t, which must be a member of the full thread-ID domain set D). Parallel layer composition in our work is always done over the whole program P and over all members of D. This allows us to reason about the current thread's behaviors over the environment's full strategies (i.e., both past and future events). Even if a thread t is never created, the semantics for running P over $L[t]$ is still well defined since it will simply always query its environment context to construct a global log.

Program Logics for Shared-Memory Concurrency. A large body of new program logics [5, 7, 8, 13, 18, 22, 23, 29, 39, 42, 44, 45, 50, 51, 56–59] have been developed to support modular verification of shared-memory concurrent programs. Most of these follow Hoare-style logics so they do not prove the same strong contextual simulation properties as RGSim and our layered framework do. Very few of them (e.g., [45]) can reason about progress properties. Nevertheless, many of these logics support advanced language features such as higher-order functions and sophisticated non-blocking synchronization, both of which will be useful for verifying specific concurrent objects within our layered framework. Our use of a global log is similar to the use of compositional subjective history traces [51]; the main difference is again that our environment context can talk about both past and future events but a history trace can only specify past events.

Both CIVL [18] and FCSL [50] attempt to build proofs of concurrent programs in a "layered" way, but their notions of layers are different from ours in three different ways: (1) they do not provide formal foundational contextual refinement proofs of linearizability as shown by Filipovic et al. [10] and Liang et al. [33]; (2) they do not address the liveness properties; (3) they have not be connected to any verified compilers.

Compositional CompCert. Stewart et al. [54] developed a new compositional extension of the original CompCert compiler [26] with the goal of providing thread-safe compilation of concurrent Clight programs. Their interaction semantics also treats all calls to synchronization primitives as external calls. Their compiler does not support a layered ClightX language as our CompCertX does, so they cannot be used to build concurrent layers as shown in Fig. 1.

Game Semantics. Even though we have used game-semantic concepts (e.g., strategies) to describe our compositional semantics, our concurrent machine and the layer simulation is still defined using traditional small-step semantics. This is in contrast to several past efforts [1, 12, 41, 47] of modeling concurrency in the game semantics community which use games to define the semantics of a complete language. Modeling higher-order sequential features as games is great for proving full abstraction, but it is still unclear how it would affect large-scale verification as done in the certified software community. We believe there are great potential synergies between the two communities and hope our work will promote such interaction.

OS Kernel Verification. There has been a large body of recent work on OS kernel verification including seL4 [24, 25], Verve [60], and Ironclad [17]. None of these works have addressed the issues on concurrency with fine-grained locking. Very recently, Xu et al. [59] developed a new verification framework based on RGSim and Feng et al.'s program logic [9] for reasoning about interrupts; they have successfully verified many key modules (in C) in the μC/OS-II kernel, though so far, they have not proved any progress properties.

Conclusions. Abstraction layers are key techniques used in building large-scale concurrent software and hardware. In this paper, we have presented CCAL—a novel programming toolkit developed under the CertiKOS project for building certified concurrent abstraction layers. We have developed a new compositional model for concurrency, program verifiers for concurrent C and assembly, certified linking tools, and a thread-safe verified C compiler. We believe these are critical technologies for developing large-scale certified system infrastructures in the future.

Acknowledgments

References

[1] Samson Abramsky and Paul-Andre Mellies. 1999. Concurrent Games and Full Completeness. In *Proc. 14th IEEE Symposium on Logic in Computer Science (LICS'99)*. 431–442.

[2] Thomas Anderson and Michael Dahlin. 2011. *Operating Systems Principles and Practice*. Recursive Books.

[3] Carliss Y. Baldwin and Kim B. Clark. 2000. *Design Rules: Volume 1, The Power of Modularity*. MIT Press.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. 4th Symposium on Formal Methods for Components and Objects (FMCO'05)*. 364–387.

[5] Stephen Brookes. 2004. A Semantics for Concurrent Separation Logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*. 16–34.

[6] Stephen Chong, Joshua Guttman, Anupam Datta, Andrew Myers, Benjamin Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. 2016. Report on the NSF Workshop on Formal Methods for Security. people.csail.mit.edu/nickolai/papers/chong-nsf-sfm.pdf. (2016).

[7] Xinyu Feng. 2009. Local Rely-Guarantee Reasoning. In *Proc. 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. 315–327.

[8] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Proc. 16th European Symposium on Programming (ESOP'07)*. 173–188.

[9] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. 2008. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation (PLDI'08)*. 170–182.

[10] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398.

[11] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *Proc. 21st International Conference on Concurrency Theory (CONCUR'10)*. 388–402.

[12] Dan R. Ghica and Andrzej S. Murawski. 2008. Angelic Semantics of Fine-Grained Concurrency. *Annals of Pure and Applied Logic* 151, 2-3 (2008), 89–114.

[13] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Proc. 22nd European Symposium on Programming (ESOP'13)*. 249–269.

[14] Alexey Gotsman and Hongseok Yang. 2012. Linearizability with Ownership Transfer. In *Proc. 23rd International Conference on Concurrency Theory (CONCUR'12)*. 256–271.

[15] 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. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. 595–608.

[16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 653–669.

[17] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 165–181.

[18] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Proc. 27th International Conference on Computer Aided Verification (CAV'15)*. 449–465.

[19] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[20] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[21] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[22] Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-grained Concurrency Specification. In *Proc. 38th ACM Symposium on Principles of Programming Languages (POPL'11)*. 133–146.

[23] Ralf Jung, David Swasey, Filip Sieczkowski, Ksper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. 637–650.

[24] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, et al. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. 22nd ACM Symposium on Operating System Principles (SOSP'09)*. 207–220.

[26] Xavier Leroy. 2005–2018. The CompCert verified compiler. http://compcert.inria.fr/. (2005–2018).

[27] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[28] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31.

[29] Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective Auxiliary State for Coarse-Grained Concurrency. In *Proc. 40th ACM Symposium on Principles of Programming Languages (POPL'13)*. 561–574.

[30] Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proc. 43rd ACM Symposium on Principles of Programming Languages (POPL'16)*. 385–399.

[31] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL'12)*. 455–468.

[32] Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional Verification of Termination-Preserving Refinement of Concurrent Programs. In *Proc. Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and 29th IEEE Symposium on Logic in Computer Science (CSL-LICS'14)*. 65:1–65:10.

[33] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *Proc. 24th International Conference on Concurrency Theory (CONCUR'13)*. 227–241.

[34] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc.

[35] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233.

[36] 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.

[37] Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*. 481–489.

[38] Andrzej S. Murawski and Nikos Tzevelekos. 2016. An invitation to game semantics. *ACM SIGLOG News* 3, 2 (2016), 56–67.

[39] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and German Andres Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Proc. 23rd European Symposium on Programming (ESOP'14)*. 290–310.

[40] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. In *Proc. 2006 ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. 62–73.

[41] Susumu Nishimura. 2013. A Fully Abstract Game Semantics for Parallelism with Non-Blocking Synchronization on Shared Variables. In *CSL 2013*. 578–596.

[42] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*. 49–67.

[43] David Michael Ritchie Park. 1981. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*. 167–183.

[44] Pedro Da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *Proc. 28th European Conference on Object-Oriented Programming (ECOOP'14)*. 207–231.

[45] Pedro Da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *Proc. 25th European Symposium on Programming (ESOP'16)*. 176–201.

[46] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS'02)*. 55–74.

[47] Silvin Rideau and Glynn Winskel. 2011. Concurrent Strategies. In *Proc. 26th IEEE Symposium on Logic in Computer Science (LICS'11)*. 409–418.

[48] Jerome H. Saltzer and M. Frans Kaashoek. 2009. *Principles of Computer System Design*. Morgan Kaufmann.

[49] Davide Sangiorgi and David Walker. 2003. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge, England.

[50] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *Proc. 2015 ACM Conference on Programming Language Design and Implementation (PLDI'15)*. 77–87.

[51] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Proc. 24th European Symposium on Programming (ESOP'15)*. 333–358.

[52] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[53] Vilhelm Sjöberg, Jieung Kim, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Proc. 15th Asian Symposium on Programming Languages and Systems (APLAS'17)*. 273–297.

[54] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. 275–287.

[55] The Coq development team. 1999 – 2018. The Coq proof assistant. http://coq.inria.fr. (1999 – 2018).

[56] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-Order Concurrency. In *Proc. 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*. 377–390.

[57] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical Relations for Fine-Grained Concurrency. In *Proc. 40th ACM Symposium on Principles of Programming Languages (POPL'13)*. 343–356.

[58] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *Proc. 18th International Conference on Concurrency Theory (CONCUR'07)*. 256–271.

[59] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Proc. 28th International Conference on Computer Aided Verification (CAV'16), Part II*. 59–79.

[60] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *Proc. 2010 ACM Conference on Programming Language Design and Implementation (PLDI'10)*. 99–110.