Journal of Systems Architecture 147 (2024) 103049

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

SimplMM: A simplified and abstract multicore hardware model for large scale system software formal verification

Jieung Kim [a,*], Ronghui Gu [b], Zhong Shao [c]

[a] Inha University, No. 1411, HiTech Center, 100, Inha-ro, Michuhol-gu, Incheon, 22212, Republic of Korea
[b] Columbia University, Mudd Building, 500 W 120th St, New York, 10027, NY, USA
[c] Yale University, 51 Prospect St, New Haven, 06511, CT, USA

ARTICLE INFO

Keywords:
Hardware architecture
Multicore hardware
System software
Operating system
Software formal verification
Formal semantics
Software correctness
Bug-free software
Concurrency
Sequential consistency
Shared memory concurrency
Linearizability

ABSTRACT

This paper introduces SimplMM, a novel subsystem within the Certified Concurrent Abstraction Layers (CCAL) modular software verification framework, designed specifically for fine-grained concurrent software. SimplMM aims to provide a generic, practical, and realistic multicore machine model for verifying software within the CCAL framework.

While formal multicore hardware semantics have seen extensive development, their integration with large-scale software verification has received limited attention. To address this gap, we propose a novel approach: a toolkit comprising a generic sequentially consistent multicore semantics, contextual refinement templates, and libraries. These components establish crucial connections between the machine model and verified program modules (layers) using CCAL. We demonstrate the practicality of our framework by successfully integrating it with existing large-scale proofs, specifically for CertiKOS running on top of the x86 hardware architecture.

This research significantly advances the field of accurate and efficient concurrent software verification and development tools for multicore systems. Our provision of a practical and formal multicore machine model, seamlessly integrated within the CCAL framework, equips developers with a powerful toolkit for large-scale concurrent software verification. The effectiveness of our approach, validated through successful integration with existing large-scale proofs such as CertiKOS, establishes a robust foundation for the design and verification of concurrent software in multicore systems.

1. Introduction

Concurrency has become a pervasive feature in modern computing systems, especially with the widespread use of multicore processors. This trend has opened up new opportunities in software development, but it also introduces significant challenges. Designing and implementing reliable concurrent software can be difficult, as even small errors can cause significant issues due to its complexity and importance in the entire software stack. Therefore, it is crucial to have a reliable method to ensure correctness. While static analysis and testing are commonly used to detect and reduce software bugs, they may not be sufficient for guaranteeing bug-free concurrent software due to the possibility of an infinite number of interleavings. As a result, formal verification, the strongest but most expensive method for assuring software correctness, has emerged as a promising approach to addressing this challenge.

Although formal verification has been applied to various types of system software [1], it is still a complex and resource-intensive task due to several requirements. For instance, it needs precise definitions of the runtime environment, such as multicore machines that operating system is running on. It also requires modular and abstract verification interfaces. A modular interface is necessary for scalabiliity by enabling the software to be easily decomposed, proven, and combined with other system modules that have been separately verified. An abstract verification interface provides simple and succinct specifications by abstracting away the interleaving and hiding unnecessary behaviors in implementations, such as local operations that do not affect other instances in concurrent software. Obviously, the verification tool that provides the simplified interface must justify how the simplification can be connected with the low-level concrete definitions. Therefore, it is essential to develop verification methodologies and tools capable of handling these challenges to improve the applicability of formal verification.

In order to address the challenges mentioned above, numerous verification frameworks have been proposed and evaluated [1–17].

* Corresponding author.
E-mail addresses: jieungkim@inha.ac.kr (J. Kim), ronghui.gu@columbia.edu (R. Gu), zhong.shao@yale.edu (Z. Shao).

https://doi.org/10.1016/j.sysarc.2023.103049
Received 9 May 2023; Received in revised form 4 December 2023; Accepted 6 December 2023
Available online 7 December 2023

However, there is still room for improvement in various aspects discussed, including enhanced automation, rigorous hardware semantics or compilations, and demonstration of the tool's applicability to large-scale software. For instance, MPI-SWS has put forward several verification frameworks [10] that facilitate the provision of specifications and proofs for concurrent software. CompcertTSO [17] offers a verification toolkit specifically designed for TSO multicore models, building upon the foundations of Compcert [15], a verified C compiler for sequential user codes. Nevertheless, the current version of these tools is limited to handling small examples, and their extensive testing for large-scale software verification is yet to be carried out.

With this observation, our primary objective is to focus on defining simple but generic, correct, and practical *multicore semantics* and provide a way to use the model in real software verification. To accomplish this goal, we introduce SimplMM, a subsystem of CCAL [2], a modular verification framework for concurrent software that includes the well-known verification case study CertiKOS [5]. CCAL provides a novel layered approach to support modular development and verification of large-scale software. It also supports modular compilation for C programs with CompCertX [18], a variant of the well known verified C compiler CompCert [19]. With the compiler, CCAL compiles each C module into the compiled assembly module, the LAsm module, and links multiple LAsm modules together to form the result of the compilation for the entire software. As a sub system of the framework, SimplMM's main goal is to bridge the gap between the underlying multicore machine model and LAsm, which is primarily targeting the machine model with a single concurrency instance, a single core. Additionally, we address how SimplMM extends the original top-level theorem of CertiKOS, which states that the compiled assembly code of CertiKOS faithfully implements its mathematical specifications, encompassing all behaviors such as memory accesses from users and system calls. This achievement is made possible by utilizing the reasonably concrete and precise x86 machine model based on SimplMM, rather than relying solely on the LAsm interfaces as in the original version.

Obviously, there are multiple challenges that we must tackle to achieve our objective. Firstly, we need to define multicore semantics that are simple yet generic, avoiding the complexities of each instruction and state in the real machine model like x86. Additionally, we must address how to provide a concurrency abstraction from the machine model while hiding the behaviors of other instances, such as other cores. To tackle these challenges, SimplMM employs a multi-step approach. In the first step, we simplify the machine model and concurrency by introducing several intermediate machine models while withholding detailed information about the underlying machines. This approach reduces the complexity of refinement proofs between adjacent machine models, and the proofs are relatively straightforward since they are based on highly abstracted state and instruction definitions. In the second step, we instantiate each machine with precise details, such as state definitions including registers and memories as well as individual hardware instructions of x86. By treating these two steps independently, we can effectively bridge the gap between the underlying x86 multicore machine model and the simplified machine model suitable for formal software verification of large-scale code. As a result, SimplMM can state and prove software correctness not based solely on the LAsm interface, which relies on concurrent interleaving as its trust-based computing base, but also on the full multicore machine model, such as x86.

The main contribution of this paper can be summarized in four parts.

- We introduce a simple abstract multicore machine model with sequential consistency, which connects concrete hardware states and instructions.
- We propose multiple intermediate machine models that abstract concurrency, addressing the challenge of defining multicore semantics as simply as possible while avoiding the complexity of individual instructions and states in real machine models like x86.
- We show contextual refinements between these intermediate models, which are general enough to be reused with the instantiated model using concrete states and definitions.
- We extend the correctness proofs of CertiKOS to show correctness based on our multicore machine model, rather than a single core machine model like LAsm.

We refer to our machine models and contextual refinement proofs collectively as SimplMM.

The structure of the rest of this paper is organized as follows. Section 2 provides a brief overview of CCAL and CertiKOS, two important related works that serve as the foundation for our research. We present an overview of the SimplMM framework in Section 3. The formal syntax, semantics, and refinement proofs of SimplMM are defined in Sections 4, 5, and 6. In Section 7, we explain how SimplMM is connected to the LAsm interface and CertiKOS proofs. Section 8 describes the proof efforts of SimplMM. We discuss related works in Section 9, and present our conclusions in Section 10.

2. Background

2.1. Coq

Coq [20] is a powerful proof assistant tool, and it provides an environment for developing mathematical facts. It enables us to define objects, establish statements related to those objects, and construct proofs to validate the defined statements. Objects in Coq can encompass a wide range of entities, such as integers, sets, trees, functions, programming languages, functional programs, and more. Statements are typically formulated using basic predicates and logical connectives. They can also use objects as their ingredients. Coq's proof-checking engine ensures the correctness of all components, including the validity and consistency of proofs as well as fundamental properties of objects and statements, such as well-formedness and termination conditions. The proof engine architecture in Coq is built upon a small trusted kernel, allowing the utilization of third-party libraries to easily build bigger systems using Coq while maintaining the integrity of the proofs.

Given these capabilities, Coq serves as an excellent tool for software verification. Verification engineers can construct program implementations and specifications using Coq objects. They can then demonstrate and verify the consistency between the specification and its corresponding implementation, leveraging the assistance of Coq. Verification efforts in Coq encompass not only functional correctness, as described earlier, but also the exploration of advanced invariant properties of programs through statements. However, performing verification using Coq still requires considerable effort, including defining formal semantics and compiling the target software's language. It also necessitates the development of practical methodologies to efficiently verify large programs. These challenges have served as inspiration for the development of CCAL.

2.2. *Smallstep library in* CompCert

Our work heavily relies on simulation templates within the Smallstep library of CompCert [21]. Despite the use of the term "simulation" in CompCert, its interpretation slightly deviates from the conventional usage in formal verification. In CompCert, simulation denotes a theorem that concerns every execution of the top-level C code. It establishes the existence of an assembly execution corresponding to the C code and resulting from compilation. However, this method does not capture how every possible low-level assembly execution will unfold, a critical aspect of CompCert's compiler correctness.

To assess this correctness directly, a more conventional simulation approach can be employed, a bottom-up approach (known as backward simulation.) This method examines every possible execution of the low-level code. Subsequently, it necessitates the presence of the

corresponding execution at the high-level specification, specifically, the corresponding C code within CompCert. However, this approach is generally more intricate than the top-down method favored by CompCert. Evaluation steps in low-level machines often outnumber those of high-level machines, necessitating the correlation of small changes in the low-level state with the unchanged state at the high level for a plethora of cases in the proof.

To streamline this process, CompCert's simulation template and method leverage the determinism in their system. Determinism allows them to invert the direction of simulation with a few facts and to provide supplementary lemmas. These lemmas translate their top-down simulation into the conventional bottom-up simulation that others typically define as simulation. The CompCert source code defines and proves this auxiliary theorem, aligning with our work, which primarily involves constructing a multicore machine model compatible with CCAL. Note that CCAL is a verification framework based on a variant of CompCert.

2.3. Certified concurrent abstraction layers

CCAL [2] is a toolkit specifically designed for the verification of concurrent programs written in C and assembly. While our work, SimplMM, is a subcomponent of CCAL, we will treat SimplMM as a separate entity for the sake of simplicity and clarity. CCAL has been successfully applied in various software formal verification projects [5,6,16,22]. With the CCAL toolkit, users can effectively break down large concurrent programs into smaller, manageable pieces and verify each piece individually. Once each piece has been verified, they can be composed to establish a top-level correctness theorem for the entire program. This modular approach empowers users to address the challenges posed by complex concurrent programs. In this section, we will provide a concise overview of CCAL and its remarkable capabilities.

CCAL **overview.** CCAL is a verification approach based on layering. Each verification piece in CCAL consists of a layer implementation and two layer interfaces: the underlay and the overlay. The goal of CCAL is to provide a certified overlay, based on the assumption of the underlay and the proof of a layer implementation on top of it. The certified overlay can also serve as a new underlay to provide another overlay with a corresponding implementation.

To enable this, the underlay provides specifications of interfaces for the instructions and primitives available to the layer implementation, while the overlay provides formal specifications for the procedures that the layer implements. Formally, they can be defined as a tuple (L_1, M, L_2), along with a refinement proof that demonstrates that the code M correctly implements the interface L_2 when executed on a system specified by the interface L_1. The code M consists of a set of functions written in C and/or assembly, which are compiled into executable code using CompCertX [18], a modified version of CompCert [19]. CCAL also supports hierarchical layering, where a higher-level layer (L_2, M', L_3) can run on top of a lower-level layer (L_1, M, L_2). Functions in M' can call functions in M, but the correctness proof only needs to consider the interface specification L_2.

With its hierarchical layering and linking libraries, CCAL enables the verification of composite abstraction layers (L_1, $M \oplus M', L_3$), where $\oplus$ is a composite operator in CCAL. The tuple describes how M and M' together implement the interface L_3 based on the interface L_1. However, the convenience does not come without a cost. CCAL has some limitations, such as disallowing dynamic memory allocation, which complicates the refinement between memory regions and corresponding abstract states, and recursive calls, which can disrupt the layer hierarchy. Despite these limitations, CCAL enables modular and hierarchical verification of concurrent programs while maintaining correctness guarantees throughout the entire program.

Layer interface. Each layer interface L in CCAL is a state transition machine represented as a pair $L = (st, P)$. Here, st represents the state of each layer, and P is a set of named *primitive specifications* that define how each primitive changes the state. The layer state, denoted as st, is defined as a tuple (ρ, mem, A), accurately reflecting the computer's state. In the definition, ρ is a register set, which is a set of values - val, mem is a memory, and A is an abstract state. Accordingly, the CompCertX compiler, which is the compiler of CCAL, also facilitates the same CCAL state definition. Having an abstract state as a component of a state makes a significant difference in CCAL and CompCertX compared to CompCert, where the state definition is (ρ, mem). The abstract state type A is often a record type that can encompass various abstract types as subfields, enabling the representation of hardware features, memory abstractions, and concurrency behaviors. For instance, system-purpose registers (*e.g.*, CR3) and a logical log that maintains a history of system transitions can be included as abstract types within A.

The named primitive specifications in P describe the abstract behavior of C/assembly functions and their transition rules on the state. In this sense, they can be viewed as simplified functional language-style functions, and they serve as formal specifications corresponding to C/Assembly implementations with underlays. Each primitive specification $\sigma \in P$ is a Coq function of type $\sigma : (val^* \times mem \times A) \to \mathsf{option}\ (val \times mem \times A)$, where val and val^* represent the types of C values and lists of values, respectively, for the function's return value and arguments. The return type of the specification is an option type, as it allows for the description of undefined behaviors by returning a $None$ value.

Additionally, each layer can use C and assembly instructions defined in CompCertX. By utilizing all these features, programmers can code C and/or assembly functions by using C and/or assembly instructions and primitives defined in the underlay to implement the overlay.

Refinement proofs. The connection between two layers is established through a simulation-based refinement proof. To prove this for a specific tuple $((st_1, P_1), M, (st_2, P_2))$, where (st_1, P_1) represents the underlay L_1 and (st_2, P_2) represents the overlay L_2, a relation R is required. This relation describes how sub-fields of the abstract state in st_2 correspond to certain objects stored in the memory mem_1 of st_1. We refer to R as the refinement relation. Specifically, if st_1 is of the form (ρ_1, mem_1, A_1) and st_2 is of the form (ρ_2, mem_2, A_2), the high-level specifications in P_2 reference specific sub-fields of the abstract value in A_2, while functions in M over $L_1 = (st_1, P_1)$ operate on parts of mem_1. The relation R specifies how the sub-fields in A_2 correspond to parts of mem_1 with the following type:

$$R_i : mem_1[i] \to A_2[i] \to \mathbb{P},$$

where R_i represents the definition for a sub-field identified as i, and $mem_1[i]$ and $A_2[i]$ denote the types of sub-fields associated with the identifier i in mem_1 and A_2 respectively. We denote the formal relation and consistency of those two layers via refinement proofs as $L_1 \sqsubseteq_R L_2$. To ensure compatibility with CompCertX, the modified version of CompCert for CCAL, CCAL utilizes simulation templates in the Smallstep library of CompCert [21]. While this approach limits the form of theorems that CCAL can prove, such as requiring termination and deterministic properties, it provides the significant advantage that CCAL is fully compatible with CompCert proofs. CCAL employs the identical methodology to that of CompCert for simulation proofs to demonstrate the refinement between abstract layers. This approach enables the integration of proofs for abstract layers in with those for the compilation process.

Example. Fig. 1 presents an illustrative example that demonstrates how CCAL establishes an abstract layer interface using concrete states and implementations. The main goal of this example is to construct an overlay layer that includes the `incr` primitive. To start, the example builds a `incr` function, `void incr()`, based on the `CNT` variable located in the memory of the underlay layer. The `CNT` variable serves as the key state for the counter, and the `void incr()` function defines a transition on this variable. Since the variable is stored in memory, it

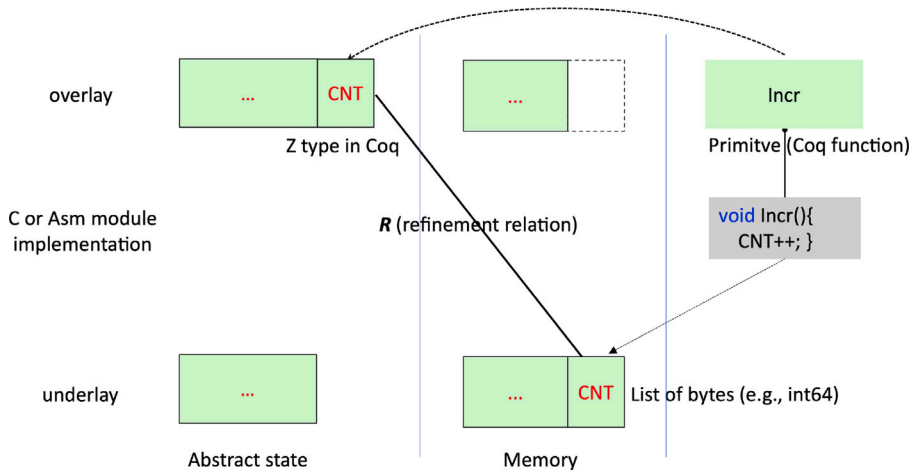

Fig. 1. Counter example in CCAL without considering concurrency.

is represented as a list of bytes. In the case of an `int64` type, the length of the list will be 8.

Next, CCAL outlines a systematic approach for converting the `void incr()` function into the `incr` primitive within the overlay. This process involves several key steps. Firstly, the permission on the `CNT` variable in the overlay's memory is revoked, ensuring that only the `void incr()` function can access it. Instead of directly accessing the `CNT` variable, the overlay introduces a `CNT` field with a $\mathbb{Z}$ type (representing an unbounded integer) in Coq. This field represents the value stored in the `CNT` variable in the underlay's memory. At this stage, both the `CNT` variable in the underlay's memory and the `CNT` field in the overlay's abstract state should contain the same value. To establish this relationship, a refinement relation denoted as R is defined, ensuring the alignment of values between the two layers. Finally, the figure introduces the `incr` primitive within the overlay. The primitive behaves exactly the same as the `void incr()` function but operates on different states, specifically the `CNT` field in the overlay's abstract state. By following these processes, a connection is established between the two layers, enabling coherent behaviors on both layers with all possible transitions.

It is important to note that while CCAL provides a systematic methodology for applying this approach to the verification process, it does not generate definitions or proofs automatically. CCAL is built on Coq, an interactive theorem prover, and it requires verification engineers to provide appropriate state definitions, primitives, refinement relations, and proofs to ensure the coherence of the two layers. However, this methodology can be applied iteratively, making it possible to decompose large-scale software into multiple layers and introduce abstract layers in a granular manner.

Events, logs, and concurrent environmental contexts. Verification frameworks for concurrent programs must consider all possible interleaved behaviors that can occur due to concurrency. Let us consider the verification of correctness and other properties of a concurrent counter as an example. In this scenario, it is crucial to account for situations where other threads can increment the counter an arbitrary number of times between the two increments performed by the current thread. Certain properties of concurrent programs, such as prohibiting two threads from reading the same number from the counter, inherently rely on these interleavings. To address this requirement, it becomes necessary to track how each concurrent instance in the system updates its state. Consequently, the verification framework needs to provide a mechanism to make all interleavings visible. Merely representing program states using more abstract and mathematical data structures instead of low-level states is insufficient. This approach only captures the value that was most recently updated, discarding all previous state updates. For example, representing a C struct with multiple integer fields using a mathematical record type with unbounded integer ($\mathbb{Z}$) fields does not capture the full range of interleavings that can occur among multiple instances in the concurrent system. Fig. 2(a) shows a similar example. It abstracts the shared counter state as a single $\mathbb{Z}$ type variable in Coq, but it has a problem of modeling and visualizing interleavings in Coq definitions and proofs since it directly discards the history of the update when a new `incr` is triggered by one instance of the system.

To tackle this challenge, the CCAL framework leverages the flexibility of abstract states within each layer to model concurrency. Instead of directly representing shared resource states with memory (*mem*) or a simple abstract type in the layer (e.g., $\mathbb{Z}$ type variable in Coq for an integer counter), CCAL utilizes logical logs as part of its abstract state to describe the history of interleaved behaviors. Fig. 2(b) shows how we model a concurrent counter by using events and a log, a collection of events. Instead of discarding the update history, it records all previous updates and who triggered those updates. Calculating the current counter value is also available with this representation by counting the number of `incr` events in the log. With this simple modification, capturing all interleaving details in a concurrent program becomes possible.

This approach also provides a way to consider all possible behaviors of other instances when specifying concurrent programs. Concurrent programs are typically designed with the assumption that each instance executes the same concurrent program sequentially (e.g., concurrent counter, spinlock, memory management module, etc.), while multiple instances can interact with each other at specific points. Therefore, when specifying such programs, it is natural to construct a program specification for a concurrent program in a similar manner to that of a sequential program, but such specifications should also capture how other instances affect the state that current instances can access together. To address this, we introduce the notion of a (concurrent) environmental context, denoted as ε, as shown in Fig. 2(c). The ε is a function that takes a CPU ID and the current log of that CPU ID and returns a log that defines the behavior of other CPUs. Instead of exposing all details of every instance in the system, the example aims to define how the system works based on the view of CPU 1 (i.e., what will be the proper specification in terms of CPU 1.) From this perspective, CPU 0 increments the value of the counter twice. However, there is a possibility for other instances (in this case, CPUs) to update the same counter, as the counter is shared among all concurrent instances in the system. To model this, an environmental context (ε) is queried with the log that CPU 0 has just before CPU 0 invokes the second `incr`. The query then provides one possible scenario in which other instances may perform their actions, such as two `incr` calls from CPU 1 in this case. However, even in this case, the possible behavior by

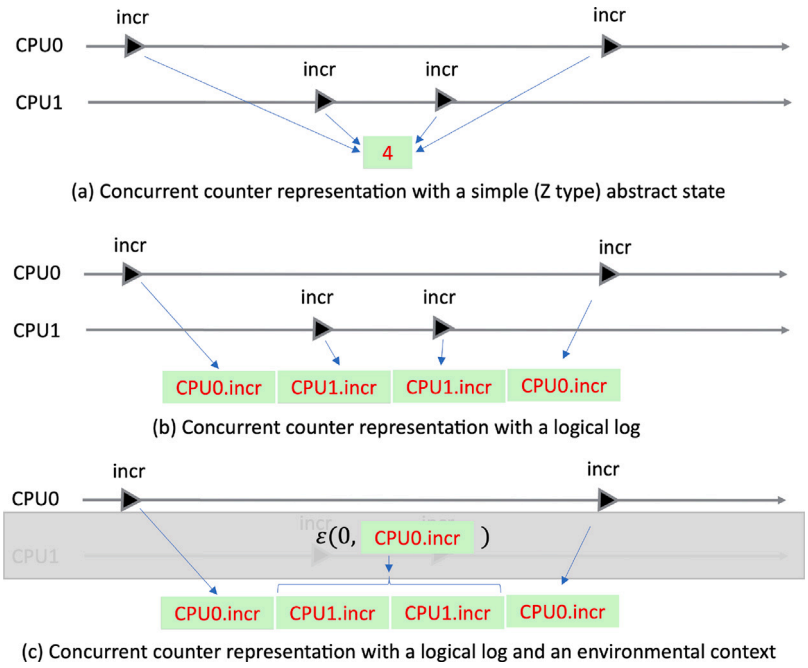

Fig. 2. Events, log, and (concurrent) environmental contexts in CCAL.

CPU 1 is slightly abstracted, as it does not specify exactly when those two `incr` calls are invoked, as seen in Fig. 2(a) and (b). The result from ε only considers who invokes how many transitions that affect the shared state. This approach allows for the consideration of various interleaved behaviors that can arise due to the concurrent nature of the program. By incorporating the environmental context and querying it at relevant points in the program, we can capture the influence of other instances and ensure to build comprehensive specifications of concurrent programs.

The discussed ideas are well fitted into abstract layer approaches and thus work as key concepts to handle concurrency in CCAL. In the abstract state of CCAL, shared resources encompass multiple sub-fields, representing fine-grained sub-data states associated with concurrent modules in the system. Examples include queues for communication channels and thread schedulers in operating systems. Every sub field for shared resources in the abstract state of CCAL must include at least one *log of events* (denoted as l) and an *environmental context* (denoted as ε). These components empower CCAL to handle concurrent programs without the need to modify underlying libraries and the compiler, as previously described in the earlier work of CCAL [2]. An *event* describes any action that has observable consequences for other CPUs. For each primitive specification, events must be defined for all points in the program where it writes to shared resources (excluding accesses to per-CPU private memory). The *log* is a list of events that represents all actions taken by the computer since it began running. To account for concurrency, actions from different CPUs are interleaved in the list. Next, an environmental context models other instances of the concurrent program. During the verification process with CCAL, the framework is always parameterized with a single instance of the concurrent program (similar to Fig. 2(c)), typically a single CPU, due to the underlying machine model LAsm in CompCertX. This necessitates a way to express the behavior of other components, and this is where the environmental context (ε) comes into play.

Even though events and environmental context provide a powerful method to handle concurrent programs, they require assumptions about concurrency interleaving. It is important to select a set of events that are granular enough to capture all possible scheduling interleavings that may occur when writing a specification for a CCAL layer. For instance, events for a simple spinlock layer might include `ACQ_LOCK` and `REL_LOCK`, while a layer with only an `incr` atomic expression may have an event such as `incr`. Therefore, it is crucial for verification engineers to be aware of the interleaving points at the bottom layer of CCAL proofs, as all verified program modules that use CCAL will depend on them. For instance, in the verified MCS Lock module in CertiKOS, memory access operations of the MCS lock data structure serve as interleaving points. However, this approach limits the rigor of formal verification using CCAL. It assumes the validity of the interleaving point, as this validity is not established through a formal connection with the actual x86 machine model which can interleave at every assembly instruction. Instead, it solely relies on the interleaving design by the verification engineer.

2.4. CertiKOS *on* CCAL

CertiKOS [5] is a formally verified operating system that handles fine-grained shared memory concurrency in the proof. It is implemented in 6500 lines of code, including 6100 lines in C and 400 lines in Assembly. CCAL is used to demonstrate the full functional correctness of the system's operating system specifications. Even though the number of lines is much smaller than that of typical commercial programs, verifying this scale and complexity is considered a large-scale verification with tremendous challenges. It is indeed one of the largest formal verification projects in the world.

Theorem 1 describes the correctness theorem of CertiKOS. The theorem is defined under the assumption that $[\![M]\!]_{\mathrm{LAsm}(L[cid,\varepsilon])}$ defines a program M's execution on LAsm with a layer L that is parameterized by CPU ID cid and environmental context ε, $\mathtt{M}_{\mathtt{certikos}}$ implements a CertiKOS kernel, $\mathtt{M}_{\mathtt{user}}$ is a user program, L_{mboot} is the bottom-most layer of all layered stacks in CertiKOS, and L_{syscall} is the top-most layer of all layered stacks in CertiKOS.

Theorem 1 (CertiKOS *Correctness With* CCAL)**.**

$$[\![\mathtt{M}_{\mathtt{certikos}} \oplus \mathtt{M}_{\mathtt{user}}]\!]_{\mathrm{LAsm}(L_{\mathrm{mboot}}[cid,\varepsilon])} \sqsubseteq_{R_{certikos}} [\![\mathtt{M}_{\mathtt{user}}]\!]_{\mathrm{LAsm}(L_{\mathrm{syscall}}[cid,\varepsilon])}$$

However, the theorem makes certain assumptions, as discussed above. Firstly, the complete theorem is parameterized by a single CPU (cid), which implies that it describes the multicore machine model symbolically. Secondly, the theorem relies on ε, which allows for

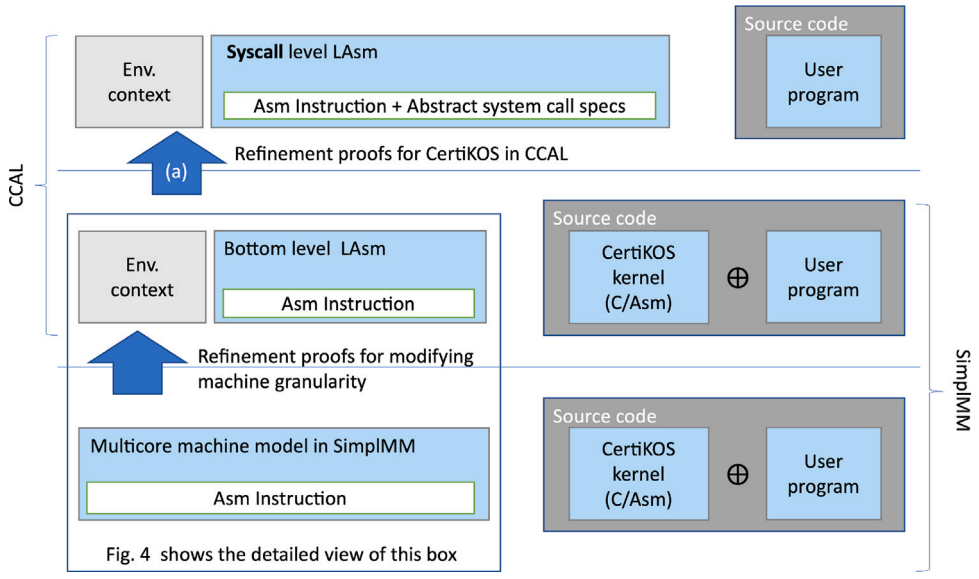

Fig. 3. CertiKOS proofs using CCAL and SimplMM.

concurrent program verification on CCAL, but its correctness must be trusted. Therefore, to strengthen the theorem, we need to address these issues and provide a reasonable multicore machine model as the foundation for the proofs that SimplMM can provide.

3. SimplMM In a nutshell

As discussed, the primary goal of our work is to eliminate the assumptions in Theorem 1 using an efficient and generic approach. Fig. 3 illustrates our focused problem using the example of CertiKOS for simplicity and clarity, although our work is not strictly tied to any specific verification target. The figure depicts three layers. The top-level layer represents the `syscall` layer in the CertiKOS proof stack with CCAL. This layer specifies the abstract layer of the entire CertiKOS kernel and can execute user programs. In Theorem 1, this corresponds to $[\![\mathtt{M}_{\mathtt{user}}]\!]_{\mathrm{LAsm}\left(L_{\mathrm{syscall}}[cid,\varepsilon]\right)}$. The middle level represents the bottom level of the CertiKOS proof stack with CCAL, which executes both the CertiKOS implementation and user programs together. In Theorem 1, this corresponds to $[\![\mathtt{M}_{\mathtt{certikos}} \oplus \mathtt{M}_{\mathtt{user}}]\!]_{\mathrm{LAsm}\left(L_{\mathrm{mboot}}[cid,\varepsilon]\right)}$. CCAL provides contextual refinements between these two layers by introducing multiple layers, refinement relations, and proofs. This is represented by (a) in the figure and aligns precisely with Theorem 1.

However, there are a few assumptions in the approach that need to be addressed. The layers between the top-level and bottom-level layers are all parameterized by a single CPU, which is a limitation of CCAL. These layers also rely on environmental contexts, as depicted in the figure and described in the theorem. However, the validity of the environmental contexts is not justified. Environmental contexts supply actions that directly alter the system's state. However, the method by which these actions are derived from the concrete evaluation of multiple instances within the system remains unclear. Also, the interleaving points at the bottom-level LAsm layer are not justified, even though they can be reviewed and designed by the verification engineers of CertiKOS. To tackle these issues, our work aims to address them by the following two methods. First, we introduce a multicore machine model that enables the execution of CertiKOS implementation and user programs together. This multicore machine model removes the assumptions about interleaving points by allowing interleavings at each assembly instruction. Second, we provide refinement proofs between our defined multicore machine model and the bottom-level LAsm layer. This ensures that the two machine models (layers) are aligned. Unlike the layers in CCAL, the multicore machine model does not require environmental contexts that transition specifications in CCAL layers rely on. Instead of specifying concurrent programs associated with a specific instance (as shown in Fig. 2(c)), the model specifies how each transition affects all states in the system, including per CPU and shared states. In this sense, it eliminates the need for environmental contexts in the same way as the layers in CCAL. The multicore machine model serves as a source for building environmental contexts for all layers in the proof stack.

Clearly, SimplMM needs to address various challenges, including parameterizing the machine model with a single CPU and constructing environmental contexts. Designing these components specifically for each software verification task and building detailed proofs for each case would be impractical. To tackle this, we employ two key ideas: breaking down the problem into multiple subproblems and providing a generic template that can be applied to any software formal verification utilizing CCAL. Fig. 4 provides more detailed information about these ideas.

To simplify the formal semantics and proofs in SimplMM, we initially make several assumptions regarding the underlying machines and the software being verified. First, we assume a fixed number of CPUs, a fixed initial state for all CPUs, and fairness among CPUs. These assumptions are reasonable for general-purpose multicore machine models like multicore x86. Second, we assume that there are no dynamic allocations in the verified software. While this assumption may not be appropriate for all software, it greatly simplifies the verification process in SimplMM and is consistent with the limitations of CCAL regarding resource allocation. Therefore, it is important to note that these assumptions do not restrict the scope of programs that can be verified using CCAL when users aim to extend the proof using SimplMM. Reducing these assumptions is a potential avenue for future work in both CCAL and SimplMM. Additionally, we assume that memory access follows sequential consistency ordering.

With the discussed assumptions, Fig. 4 provides an overview of how SimplMM is designed and how it can be connected with the bottom-level CCAL layer for a specific software verification task. In the figure, all the elements within the SimplMM box represent generic framework components that are not specific to any particular target software being verified. This means that the definitions and proofs inside the box can be reused for multiple verification targets. On the other hand, the red arrows and lines represent target-specific components that need to be instantiated and proven in order to establish the connection between SimplMM and a specific software verification task using CCAL.

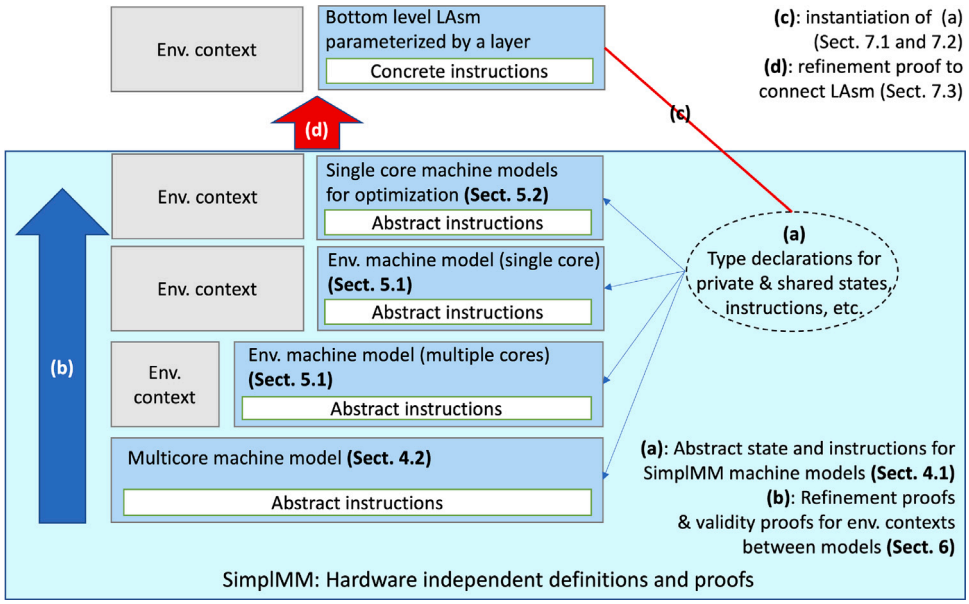

Fig. 4. SimplMM overview.

As discussed, SimplMM addresses two distinct challenges separately: (1) simplifying the multicore machine model to a single-core machine model while abstracting away the detailed machine information, such as state and transition definitions, of the x86 machine model, and (2) providing a generic framework that can be connected with any CCAL layers involved in different verification targets. To tackle these challenges, we begin by defining abstract states and instructions (Fig. 4(a)) that can be shared across multiple machine models in SimplMM, including the multicore and intermediate machine models. This approach not only adds generality to SimplMM but also reduces the complexity associated with multiple machine models. While the different machine models in SimplMM have varying levels of granularity, parameterized by different concurrent instances and having different interleaving points, the evaluation rules for each transition (i.e., hardware instructions performed in each instance) are quite similar. By defining abstract yet generic definitions for these common components, we can effectively reduce the size of machine definitions. Next, we gradually introduce appropriate environmental contexts for each intermediate machine model and establish two key proofs: refinement proofs between the machine models and the validation of the proper context for each model (as depicted in Fig. 4(b)). These proofs ensure that the machine models in SimplMM are refined versions of each other and that the environmental contexts are correctly defined and utilized in the verification process.

This approach leads to the development of a single-core machine model and a concurrent context, which can be connected with the machine model of CCAL, specifically, LAsm parameterized with a layer definition. However, additional work is required to establish the connection between LAsm parameterized by a layer and the models introduced in SimplMM. This is because the introduced models and proofs in SimplMM still rely on abstract definitions that need to be instantiated with specific machine details. To achieve this, we begin by instantiating the abstract definitions with machine details from a specific layer that will be combined with SimplMM. In the case of the CertiKOS proof, this would be $\mathrm{LAsm}\left(L_{\mathrm{mboot}}[cid, \varepsilon]\right)$. This instantiation allows us to incorporate the specific details of the layer into the framework, as depicted in Fig. 4(c). Finally, we establish a one-step connection by providing a refinement proof between the top-level intermediate machine model in SimplMM and LAsm parameterized by a layer, which is depicted in Fig. 4(d). The bottom-level layer of LAsm is entirely determined by verification engineers. One straightforward example is a layer containing atomic instructions for shared objects, expanding the LAsm model to accommodate concurrent behaviors. This proof ensures that the machine models in SimplMM and CCAL are consistent and that the connection between them is valid. The similar idea has been briefly described in previous works [2,5]. However, concrete formal semantics and proofs were not provided as SimplMM has done. Also, it is well-known that there is always a significant gap between a high-level idea and its real formal semantics due to several details that a high-level idea may overlook. We explain the key concepts of each step from Section 4 to Section 7 to provide a better understanding of the process.

4. Multicore machine

In SimplMM, machine models are defined in a modular way by decomposing the necessary components into multiple levels, including the instruction-level, per CPU level, and global machine level. This modular approach enables the sharing and reuse of components across different machine models, leading to more concise proofs. This section begins by introducing a formal definition of abstract states and instruction-level transitions in SimplMM. These abstractions are crucial for representing system states and transitions without delving into the specific details of hardware platforms like x86 or ARM, as discussed in Section 3. By using these abstract definitions, concise proofs can be achieved across multiple formal languages within SimplMM. Next, the section explains the semantics of each CPU by utilizing the abstract states and instruction-level semantics. This allows for a detailed understanding of the behavior of individual CPUs in the system. Finally, the section presents the multicore machine model, which serves as the foundational level of SimplMM based on the abstract definitions introduced earlier. This machine model provides a comprehensive framework for reasoning about the behavior of concurrent systems and enables the verification of complex properties.

4.1. Abstract hardware state and transition rules

The key components of the abstracted hardware states and operations in SimplMM are depicted in Fig. 5(a). The model consists of two types of states: private states (ρ_{mc}) and shared states (ς_{mc} and $e_{\mathtt{atom}}$) which can be instantiated with any types in Coq ($\mathrm{T}_{\mathtt{TYPE}}$ represents the top type in Coq). Private states represent resources that are only accessible by each individual CPU, such as general-purpose

$$
\begin{array}{lcll}
\rho_{mc} & : & \top_{\texttt{TYPE}} & \text{(Private state)} \\
\varsigma_{mc} & : & \top_{\texttt{TYPE}} & \text{(Shared state)} \\
\boldsymbol{e}_{\texttt{atom}} & : & \top_{\texttt{TYPE}} & \text{(Atomic event)} \\
\boldsymbol{cmd}_{\texttt{mc}} & := & \texttt{PRIVATE} \mid \texttt{ATOMIC}\langle id_{rsc} : \mathbb{Z},\ name_{prim} : \text{string}\rangle \mid \texttt{ACQ\_SHARED}\langle id_{rsc} : \mathbb{Z}\rangle \mid \texttt{REL\_SHARED}\langle id_{rsc} : \mathbb{Z}\rangle & \\
\boldsymbol{ev}_{mc} & := & \texttt{EYIELD}\langle from : \mathbb{Z}\rangle \mid \texttt{EBACK}\langle to : \mathbb{Z}\rangle \mid \texttt{EATOMIC}\langle from : \mathbb{Z},\ id_{rsc} : \mathbb{Z},\ e : \boldsymbol{e}_{\texttt{atom}}\rangle & \\
 & & \mid \texttt{EACQ}\langle from : \mathbb{Z},\ id_{rsc} : \mathbb{Z}\rangle \mid \texttt{EREL}\langle from : \mathbb{Z},\ id_{rsc} : \mathbb{Z},\ d : \varsigma_{mc}\rangle & \\
\boldsymbol{log}_{\text{mc}} & := & \text{lst}\ \boldsymbol{ev}_{mc} & \text{(History of shared operations)}
\end{array}
$$

(a) Abstract states, instructions, and events.

$$
\begin{array}{lcl}
\texttt{PC} & : & \rho_{mc} \to \boldsymbol{cmd}_{\text{mc}} \to \mathbb{P} \\
\texttt{Eval}_{\texttt{private}} & : & \mathbb{Z} \to \rho_{mc} \to \rho_{mc} \to \mathbb{P} \\
\texttt{Eval}_{\texttt{get}} & : & \rho_{mc} \to \varsigma_{mc} \to \rho_{mc} \to \mathbb{P} \\
\texttt{Eval}_{\texttt{set}} & : & \rho_{mc} \to \texttt{option}\ \varsigma_{mc} \to \rho_{mc} \to \mathbb{P} \\
\texttt{Eval}_{\texttt{atom}} & : & \mathbb{Z} \to \mathbb{Z} \to \rho_{mc} \to \boldsymbol{log}_{\text{mc}} \to \rho_{mc} \to \boldsymbol{e}_{\texttt{atom}} \to \mathbb{P}
\end{array}
$$

(b) abstract transition rules.

Fig. 5. Abstract state and transitions.

registers and memory locations that are not shared with other CPUs. In contrast, shared states are accessible by all CPUs. Shared states are further divided into two sub-states based on the atomicity of operations performed on them. The first shared state (ς_{mc}) allows non-atomic operations, which means that it allows for the possibility of data race conditions on the state. It is typically the responsibility of programmers to prevent such conditions by implementing protection mechanisms, like spinlocks. The second shared state represents a state that allows only atomic access, such as the CAS instruction, an instruction used in multithreading to achieve synchronization by atomically performing compare-and-swap. This state is indirectly modeled using atomic events ($\boldsymbol{e}_{\texttt{atom}}$), which correspond to the atomic assembly instructions supported by the system. The way to represent resources related to atomic operations follows the concurrent representation concept of CCAL, using events and logs described in Section 2. By doing that, SimplMM and CCAL share common representations for atomic operations that are associated with concurrency interleaving, and providing formal connections between SimplMM and CCAL becomes easy thanks to their similarity.

SimplMM introduces four commands to model operations on the state, consisting of two abstract commands and two logical commands. The abstract commands combine multiple concrete instructions into a single command, while the logical commands capture the logical transfer of ownership of shared memory locations by the system, such as memory regions for thread control blocks in an operating system. The use of only two abstract instructions by grouping multiple instructions significantly simplifies proofs within SimplMM. In contrast, when all machine models in SimplMM explicitly utilize machine-specific instructions, refinement proofs between multiple machine models would have to account for numerous cases and heavily rely on specific hardware details. However, our approach reduces the size of refinement proofs due to the succinct design of the language achieved through abstract definitions. Furthermore, we can provide a machine-independent framework by abstracting away machine-specific details, except in cases where they are genuinely necessary. Additional details about this issue are provided in Section 7.

The first two commands in SimplMM are the private command (PRIVATE) and the atomic command (ATOMIC). The private command represents most instructions in x86 hardware, where their behaviors are localized to a single CPU. Examples of private commands include arithmetic instructions and memory load/store instructions. Arithmetic instructions, by their nature, only affect the CPU executing them. However, the effects of memory access depend on the specific memory region being read from or written to. Accessing private memory with private commands is considered safe. However, accessing shared memory locations with a private command requires the accessing CPU to own the memory location to avoid data race conditions. This can lead to unsafe situations, but it is a possible behavior in x86 and other multicore machine models when the memory region is not suitably protected. Similar to x86 and other hardware, our model differentiates between memory access and ownership. Again, it is the responsibility of programmers to develop data race-free programs. On the other hand, atomic commands are mapped to special instructions that guarantee atomicity in state updates, both in registers and memories. An example of an atomic command is the xchg instruction, which is a x86 instruction that atomically exchanges a register/memory with a register. The remaining two commands are logical commands that represent ownership of shared states. These commands play a crucial role in defining data race conditions in shared memory locations and connecting SimplMM to LAsm, the hardware model of CompCertX. The atomic, acquire, and release commands also include additional information such as the id_{rsc} and $name_{prim}$. The id_{rsc} number is a fixed logical identifier assigned to each shared resource (e.g., page table, IPC channels in operating systems). In hardware models for software with dynamic allocation, assigning these identifiers to shared resources can be complex. However, the simplified memory allocation in CCAL and CertiKOS [5] reduces complexity in SimplMM by eliminating the need for dynamic allocation. While this limitation exists in SimplMM, it greatly simplifies the use of shared states in the model. The $name_{prim}$ string represents the primitive name to distinguish each atomic primitive from others.

When defining a multicore machine model, it is crucial to incorporate logical events that go beyond real hardware instructions. These logical events are necessary to capture the interleavings among multiple CPUs and make them visible, thus playing a significant role in representing the system's behavior as a whole. To fulfill this purpose, the logical history ($\boldsymbol{log}_{\text{mc}}$) is introduced as a shared record among all CPUs. It not only includes the updates on atomic states but also captures the logical events that influence the states and control across multiple CPUs. Among the different events generated, two are dedicated to modeling the interleaving between multiple cores and a logical component in our system, logical system scheduler. A logical system scheduler is a conceptual framework designed to represent the non-deterministic interleaving decisions within a log. The yield event (EYIELD) occurs when one core ($from$) relinquishes control to the logical system scheduler, allowing the system to transfer control to another core. Conversely, the yield-back event (EBACK) is generated by the logical scheduler when a core (to) gains control of execution. The remaining three events track shared memory accesses and are parameterized by the executor (core - $from$), the logical identifier of the shared state being accessed (common to all event types), the result of non-atomic updates (d) for EREL, and the executed atomic operation (e) for $\boldsymbol{e}_{\texttt{atom}}$. These events provide

$$\texttt{view}_{\texttt{l}} := \texttt{LView}\ (ls : \rho_{mc})\ (log : \textbf{\textit{log}}_{\texttt{mc}})$$

$$\text{PRIVATE}\quad \frac{\textbf{PC}\ ps\ \texttt{PRIVATE} \qquad \texttt{Eval}_{\texttt{private}}\ curid\ ps\ ps'}{\texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ \texttt{nil})}$$

$$\text{ACQ\_RULE}\quad \frac{\textbf{PC}\ ps\ \texttt{ACQ\_SHARED}\langle id_{rsc}\rangle \qquad \texttt{CalOwner}(l, id_{rsc}) = \texttt{OFREE}\langle d\rangle \qquad \texttt{Eval}_{\texttt{set}}\ ps\ d\ ps'}{\texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ (\texttt{nil} \bullet \texttt{EACQ}\langle curid, id_{rsc}\rangle))}$$

$$\text{REL\_RULE}\quad \frac{\textbf{PC}\ ps\ \texttt{REL\_SHARED}\langle id_{rsc}\rangle \qquad \texttt{CalOwner}(l, id_{rsc}) = \texttt{OWN}\langle curid\rangle \qquad \texttt{Eval}_{\texttt{get}}\ ps\ d\ ps'}{\texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ (\texttt{nil} \bullet \texttt{EREL}\langle curid, id_{rsc}, d\rangle))}$$

$$\text{ATOMIC}\quad \frac{\textbf{PC}\ ps\ \texttt{ATOMIC}\langle id_{rsc}, \texttt{GetID}_{\texttt{atom}}(e)\rangle \qquad \texttt{Eval}_{\texttt{atom}}\ curid\ id_{rsc}\ ps\ l\ ps'\ e}{\texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ (\texttt{nil} \bullet \texttt{EATOMIC}\langle curid, id_{rsc}, e\rangle))}$$

Fig. 6. Local view (state) and local step rules.

essential information for monitoring and coordinating access to shared memory in the system.

The semantic signature of each operation is presented in Fig. 5(b). These operations return $\mathbb{P}$, which represents the return type of propositional logic — a statement that can be either true or false — in Coq. In order to align with the way that LAsm, the assembly model of CCAL, defines semantics for its instructions, we use propositional logic instead of Coq functions for these operations. It is important to note that these operations do not have concrete definitions in SimplMM for the same reason why SimplMM uses abstract commands. They are concretized to establish the connection between SimplMM and layers in CCAL, and will be further discussed in Section 7.

The program counter (**PC**) is a special mechanism that retrieves the current private state of a core and determines the command to be evaluated next. The private command transition rule ($\texttt{Eval}_{\texttt{private}}$) takes the current CPU ID and private state as input and produces the resulting private state after evaluation. The atomic command transition rule ($\texttt{Eval}_{\texttt{atom}}$) relies on the current CPU ID, resource ID, private state, and logical history. It returns the updated private state and the atomic event that results from the evaluation. The get and set rules ($\texttt{Eval}_{\texttt{get}}$ and $\texttt{Eval}_{\texttt{set}}$) define the transitions for EACQ and EREL, respectively. $\texttt{Eval}_{\texttt{get}}$ takes the private and shared states as input and returns the private state, reflecting the snapshot of the shared resources. Conversely, $\texttt{Eval}_{\texttt{set}}$ defines the reverse transition rule. These rules form the foundation for the transition rules in various machine models within SimplMM, as explained in the subsequent sections.

Fig. 6 illustrates how transitions in each core are defined using the instruction-level semantic rules presented in Fig. 5(b). Specifically, the figure describes how these instruction-level semantic rules are employed to define the local step rules (LMach) based on the abstract CPU-local state ($\texttt{view}_{\texttt{l}}$), also known as the local view.

As illustrated in Fig. 6, local step rules are ternary relations involving one CPU ID and two local views ($\texttt{view}_{\texttt{l}}$). Local views play crucial roles in the semantics, and two local views associated with each rule have different meanings and purposes. The first local view of each local step rule represents the initial state from which the current instruction performs its transition. It includes the initial private state of the CPU and the shared log that represents the initial shared state. On the other hand, the second local view in each semantic rule represents the result of the evaluation that needs to be updated in the global state defined by the global machine semantic rules. Since the initial shared log is already known in the global machine semantic rules, it does not need to be included in the left-hand side local view. Instead, the right-hand side local view contains the final private state and the log (or event) that will be added to the shared log in the global state as defined by the global semantic rules, while the notation $l \bullet e$ represents the concatenation of the event e into the list l. By carefully defining and utilizing the local view, the transitions within each core can be effectively captured, allowing for the composition of these local transitions to form the global behavior of the multicore system.

The local step rules take the local view (LView ps l) as input, which represents the pre-state of the evaluation for the corresponding CPU. These rules return an updated private state and the log that is generated during the evaluation, which also needs to be updated in the shared state that is visible to other CPUs. For example, when the current program counter indicates a private instruction, the semantic rule (PRIVATE rule) only updates the private state of the CPU without generating any logs during the evaluation. On the other hand, when evaluating an atomic instruction, the machine updates the private state and generates an event that corresponds to the atomic command (ATOMIC rule). The acquire and release operations (ACQ_RULE and REL_RULE rules) make use of an auxiliary function (CalOwner) to determine if the resource can be acquired or released and to find the appropriate snapshot, which reflects the state achieved by replaying all events in the log of the shared state that needs to be updated in the private state. However, it is crucial to emphasize that these rules alone do not guarantee the absence of data races in the log. To ensure data race freedom, additional program verification mechanisms, such as the use of spinlocks [23], need to be employed. These mechanisms help discharge the obligations for data race freedom and enhance the overall correctness and reliability of the system.

By utilizing the local step rules, the transitions within each core can be properly handled, ensuring the consistency of the private and shared states. However, further verification is necessary to establish data race freedom and other desired properties in the system.

4.2. Multicore machine syntax and semantics

With the rules defined in Section 4.1, modeling a multicore machine is straightforward, and it is described in Fig. 7. The multicore machine model operates by making non-deterministic selections for the next CPU to perform evaluations based on the global state (hstate), which consists of the current CPU ID ($curid$), a set of private states for each CPU (lsp), and a shared log (log). The process of non-deterministic evaluations involves recording two scheduling events that represent transferring the current control from $curid$ to $curid'$. The multicore machine model then retrieves the private state of $curid'$ ($\texttt{getPST}(lsp,\ curid') = \texttt{Some}\ ps$) from the set of private states, performs the evaluation using the local transition rules defined in Fig. 6. Finally, it updates the result, which are the updated private state and the generated log, in the global state ($\texttt{setPST}(curid',\ ps',\ lsp) = lsp'$). In this rule, lsp' represents the updated private state, which changes the private state of the evaluated CPU while leaving the private states of other CPUs unchanged. The updated shared log is represented by $l' +\!\!+ l''$, which concatenates l'' and l'. Overall, the multicore machine model allows for interleaving of CPUs in each evaluation step without exposing detailed states and instructions. It provides a concise and effective way to model the behavior of a multicore machine.

$$\begin{array}{lll} \texttt{psp} & : & \mathbb{Z} \rightharpoonup \rho_{mc} \\ \texttt{hstate} & := & \texttt{HState}\ (curid : \mathbb{Z})\ (lsp : \texttt{psp})\ (log : \textbf{\textit{log}}_{\text{mc}}) \end{array}$$

$$\frac{l' = l \bullet \texttt{EYIELD}\langle curid\rangle \bullet \texttt{EBACK}\langle curid'\rangle \qquad \texttt{getPST}(lsp,\ curid') = \texttt{Some}\ ps \qquad curid' \in \mathbb{S}_{core} \qquad \texttt{LMach}_{\texttt{CPU}}\ curid'\ (\texttt{LView}\ ps\ l')\ (\texttt{LView}\ ps'\ l'') \qquad \texttt{setPST}(curid',\ ps',\ lsp) = lsp'}{\texttt{Mach}_{\texttt{mc}}\ (\texttt{HState}\ curid\ lsp\ l)\ (\texttt{HState}\ curid'\ lsp'\ (l' +\!\!+ l''))}$$

Fig. 7. Multicore machine syntax and semantics.

$$\begin{array}{lll} \texttt{coinp} & : & \mathbb{Z} \rightharpoonup \mathbb{B} \\ \texttt{estate}\ (A : \{\mathbb{Z}\}) & := & \texttt{EState}_{[A]}\ (curid : \mathbb{Z})\ (lsp : \texttt{psp})\ (cp : \texttt{coinp})\ (log : \textbf{\textit{log}}_{\text{mc}}) \end{array}$$

PROGRESS

$$\frac{\texttt{getPST}(curid,\ lsp) = \texttt{Some}\ ps \qquad \texttt{getCoinP}(curid,\ cp) = \texttt{Some false} \qquad \texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ l') \qquad \texttt{setPST}(curid,\ ps',\ psp) = psp' \qquad \texttt{setCoinP}(curid,\ \texttt{true},\ cp) = cp'}{\texttt{Mach}_{\texttt{env}[A]}\ \varepsilon_A\ (\texttt{EState}_{[A]}\ curid\ lsp\ cp\ l)\ (\texttt{EState}_{[A]}\ curid\ lsp'\ cp'\ (l +\!\!+ l'))}$$

YIELD

$$\frac{l_0 = l \bullet \texttt{EYIELD}\langle curid\rangle \qquad \textbf{oget}(l_0, \varepsilon_A) = \texttt{Some}\ \texttt{EBACK}\langle curid'\rangle \qquad l' = l_0 \bullet \texttt{EBACK}\langle curid'\rangle \qquad curid' \in A \qquad \texttt{getCoinP}(curid',\ cp) = \texttt{Some true} \qquad \texttt{setCoinP}(curid',\ \texttt{false},\ cp) = cp'}{\texttt{Mach}_{\texttt{env}[A]}\ \varepsilon_A\ (\texttt{EState}_{[A]}\ curid\ lsp\ cp\ l)\ (\texttt{EState}_{[A]}\ curid'\ lsp\ cp'\ l')}$$

SKIP

$$\frac{\texttt{getPST}(curid,\ lsp) = \texttt{None} \qquad \texttt{getCoinP}(curid,\ cp) = \texttt{None} \qquad \textbf{oget}(l, \varepsilon_A) = \texttt{Some}\ e \qquad l' = l \bullet e \qquad \texttt{GetCID}(l) = curid' \qquad curid' \in A}{\texttt{Mach}_{\texttt{env}[A]}\ \varepsilon_A\ (\texttt{EState}_{[A]}\ curid\ lsp\ cp\ l)\ (\texttt{EState}_{[A]}\ curid'\ lsp\ cp\ l')}$$

Fig. 8. Environment machine model syntax and semantics.

5. Introduce CPU local machine model

The connection between the multicore machine model of SimplMM in Section 4 and a particular layer of CCAL is challenging due to the significant gap between the two models. There are three main issues that need to be addressed: (1) defining a transition rule that focuses on a single-core while abstracting away the others and establishing a formal connection between them, since layers of CCAL are parameterized by a single-core. (2) eliminating execution control events that are not present in the machine model of CCAL. (3) mapping each abstract command and transition to its concrete definition of a certain layer of CCAL that users hope to be connected with SimplMM. This section addresses the first two challenges, while the third challenge is tackled in Section 7.

5.1. Environmental machine model

Introducing a single-core machine with a proper environment context, the method to simplify interleavings and to hide non-determinism, requires two components: (1) masking the non-determinism of multicore hardware, and (2) building an environment context that tracks the evaluation history of shared resources by other CPUs. The intermediate machine model, an environment machine model, shown in Fig. 8 handles these two challenges. This model is parameterized by two factors: the current available CPU set (A when $A \subseteq \mathbb{S}_{core}$) and the environment context for the set (ε_A). The state definition (`estate`) is similar to that of a multicore machine model, but with an additional field, cp, to differentiate between states that are eligible for local evaluation and those that are eligible for scheduling.

In comparison to the multicore machine model depicted in Fig. 7, the environmental machine model distinguishes between local evaluations (PROGRESS) and scheduling evaluations (YIELD). This distinction is achieved by introducing a boolean value in the environmental machine model. The `coinp` definition is used to identify this boolean value for each CPU, and the functions `getCoinP` and `setCoinP` are employed to update these values in the machine's state. During a scheduling evaluation, the model first records that the current CPU has triggered the scheduling ($l_0 = l \bullet \texttt{EYIELD}\langle curid\rangle$). Next, it uses the environment context (ε_A), which defines a single scheduling sequence among all possible sequences, to determine the next CPU to be scheduled by calling the context query function ($\textbf{oget}(l_0, \varepsilon_A) = \texttt{Some}\ \texttt{EBACK}\langle curid'\rangle$). The model then records the next scheduled CPU ID in the log ($l' = l_0 \bullet \texttt{EBACK}\langle curid'\rangle$). This deterministic scheduling sequence makes the machine model deterministic, effectively hiding the non-determinism present in multicore machine model. The relationship between the two models is discussed in Section 6.

Since A is a subset of $\mathbb{S}_{core}$ (the full set of CPUs in the system), some CPUs that are in $\mathbb{S}_{core}$ may not be in A. The third rule (SKIP) is applied to these CPUs that are not in A but are in $\mathbb{S}_{core}$. In this case, the rule queries the environment context ($\textbf{oget}(l, \varepsilon_A) = \texttt{Some}\ e$) and adds the result event of the query ($l' = l \bullet e$) to the log. This result represents the abstracted behaviors on shared resources by those CPUs (CPUs that are not in A). It is important to note that there may be multiple private step rules for CPUs that are not in A. These private steps do not affect the observable behaviors of other CPUs, so the rule does not take them into account during the environmental context query. It is important to note that the current available set (A) is a parameter of the step rule. If the set A is the full set of CPUs ($A = \mathbb{S}_{core}$), then this rule is equivalent to the rule in the multicore semantics, except that it hides the non-determinism caused by the multicore environment. If the set A is a single CPU ($A = c$ for some $c \in \mathbb{S}_{core}$), then it becomes a single-core machine model with an environment context for the rule.

5.2. CPU-local machine model and optimizations

The rules in Fig. 8 with a singleton available CPU set can already be considered a machine model with only a single-core. The environment machine model addresses the key challenges of hiding non-determinism and establishing a environment context. However, the rules still differ from transitions in a layer of CCAL in terms of state definition and

$$\texttt{state}_{\texttt{si}} \quad := \quad \texttt{SState}\ (curid : \mathbb{Z})\ (ls : \rho_{mc})\ (coin : \mathbb{B})\ (log : \textbf{\textit{log}}_{\texttt{mc}})$$

$$\text{PROGRESS}\quad \frac{cid = curid \qquad \texttt{LMach}_{\texttt{CPU}}\ curid\ (\texttt{LView}\ ps\ l)\ (\texttt{LView}\ ps'\ l')}{\texttt{Mach}_{\texttt{si}[cid]}\ \varepsilon_{cid}\ (\texttt{SState}\ curid\ ls\ \texttt{false}\ l)\ (\texttt{SState}\ curid\ ls'\ \texttt{true}\ (l +\!\!+ l'))}$$

$$\text{YIELD}\quad \frac{cid = curid \qquad l_0 = l \bullet \texttt{EYIELD}\langle curid\rangle \qquad \textbf{oget}(l_0, \varepsilon_{cid}) = \texttt{Some}\ \texttt{EBACK}\langle curid'\rangle \qquad l' = l_0 \bullet \texttt{EBACK}\langle curid'\rangle}{\texttt{Mach}_{\texttt{si}[cid]}\ \varepsilon_{cid}\ (\texttt{SState}\ curid\ ls\ \texttt{true}\ l)\ (\texttt{SState}\ curid'\ ls\ \texttt{false}\ l')}$$

$$\text{SKIP}\quad \frac{\textbf{oget}(l, \varepsilon_{cid}) = \texttt{Some}\ e \qquad l' = l \bullet e \qquad \texttt{GetCID}(l) = curid' \qquad curid' \neq cid}{\texttt{Mach}_{\texttt{si}[cid]}\ \varepsilon_{cid}\ (\texttt{SState}\ curid\ ls\ \texttt{true}\ l)\ (\texttt{SState}\ curid'\ ls\ \texttt{true}\ l')}$$

Fig. 9. Single-core machine model syntax and semantics.

environment context query. To simplify the model and ease the proofs for connections, we introduced multiple single core machine models by defining a state focused on a single-core and reducing the complexity of recorded logs in our machine model. Fig. 9 shows the first simplification, which reduces the state definition ($\texttt{state}_{\texttt{si}}$) to only allow for a single-core and its associated state. As described, the log and context query have also been simplified, for example, by merging multiple SKIP steps into a single big step style **oget** function, and eliminating unnecessary yield and yield back events. Finally, the yield and yield back events will be merged with the other three events, since changes in execution control at those positions do not affect the state changes caused by arbitrary interleaving.

6. Refinement proofs

SimplMM establishes formal connections between machine models and their concurrent environment contexts by proving two properties. Firstly, we prove that the environment contexts utilized in environmental and single-core machine models can always be obtained from evaluations of multicore and environmental machine models, respectively. This demonstrates that the interleaving hiding with environmental contexts in these machines encompass all necessary interleaving behaviors that can occur in lower-level machines. Secondly, we demonstrate refinements between different machines when appropriate environmental contexts are present, thus confirming that their observable behaviors are equivalent. These two properties can be combined into a single theorem when focusing solely on showing refinement proofs between machine models in SimplMM. However, our primary goal is to provide a multicore machine model that can be linked with layers in CCAL. As a result, our proofs must be crafted to align with the standard simulation library in CompCert. Therefore, two key properties in SimplMM are stated separately. Then, the existence of environmental contexts is verified internally within SimplMM, and refinements between multiple models with the given environmental contexts are verified using the standard simulation library in CompCert. Similar to the previous section, all refinement relations and proofs are mechanized using Coq and can be accessed online [24,25].[1]

Theorem 2 (*Existence of a Scheduling Oracle*). *Assuming a multicore machine evaluation,* $\texttt{Mach}_{\texttt{mc}}{}^{*}\ hst\ hst'$. *Then, there always exists the corresponding consistent environmental machine model steps and the constructed environmental context,* $\varepsilon_{\mathbb{S}_{core}}$, *with the simulation proofs. Also, the constructed environmental context has the same behavior with the generated log* l' *during the evaluation, when* l' *is the log in the state* hst'.[2]

Theorem 2 establishes the existence of $\varepsilon_{\mathbb{S}_{core}}$ that are required in environmental evaluation steps, denoted by $\texttt{Mach}^{*}_{\texttt{env}[\mathbb{S}_{core}]}\ \varepsilon_{\mathbb{S}_{core}}\ est\ est'$. In this context, $\texttt{Mach}^{*}_{\texttt{env}[\mathbb{S}_{core}]}\ \varepsilon_{\mathbb{S}_{core}}\ est\ est'$ represents a valid environmental machine model evaluation that is consistent with $\texttt{Mach}_{\texttt{mc}}{}^{*}\ hst\ hst'$. This theorem is a crucial element in eliminating non-determinism in our proofs. The theorem establishes that it is possible to represent interleavings as a deterministic function by providing events associated with a single execution for any of the possible executions in the multicore machine model. This technique is also applied in other parts of the system where new environmental contexts are introduced, such as when creating an environmental machine model with a subset of $\mathbb{S}_{core}$. The proof of Theorem 2 is simple and relies on induction on the evaluation of the multicore machine. To construct $\varepsilon_{\mathbb{S}_{core}}$, it records the current log as input and outputs scheduling-related events ($\texttt{EYIELD}, \texttt{EBACK}$). This is achieved by recording the log before the evaluation in Fig. 6 (l' in the rule) as input and the scheduling-related events ($\texttt{EYIELD}\langle curid\rangle$ and $\texttt{EBACK}\langle curid'\rangle$) as output.

Theorem 3 (*Env. Step Refines Multicore Step*). *Given a multicore machine evaluation* ($\texttt{Mach}_{\texttt{mc}}{}^{*}\ hst\ hst'$), *a set of all CPUs in the system* ($\mathbb{S}_{core}$), *a valid environment context* ($\varepsilon_{\mathbb{S}_{core}}$), *and an environmental state* (est) *related to* hst *with a refinement relation* ($match\_state\ hst\ est$). *There always exists a valid updated environmental state* (est') *and environmental transition* ($\texttt{Mach}^{*}_{\texttt{env}[\mathbb{S}_{core}]}\ \varepsilon_{\mathbb{S}_{core}}\ est\ est'$) *that satisfy the relationship* ($match\_state\ hst'\ est'$).[3]

Theorem 3 demonstrates the existence of a related environmental machine model evaluation with the core set ($\mathbb{S}_{core}$) for a given multicore machine evaluation, using the refinement relation $match\_state$. The precise definition of the refinement relation $match\_state$ may involve several sub-equations, which can appear tedious. However, at its core, the relation is based on the equality between corresponding subparts of the two machines. For example, consider the case where $hst = \texttt{HState}\ curid_{mc}\ lsp_{mc}\ l_{mc}$ and $est = \texttt{EState}_{[\mathbb{S}_{core}]}\ curid_{e}\ lsp_{e}\ cp_{e}\ l_{e}$. In this scenario, the refinement relation $match\_state\ hst\ est$ checks the equality of the following three subcomponents: (1) $curid_{mc}$ and $curid_{e}$, (2) lsp_{mc} and lsp_{e}, and (3) l_{mc} and l_{e}. Moreover, the relation includes additional checks to ensure the validity of other parts, such as cp_{e}. It is worth noting that similar refinement relations exist for other refinement proofs, although we have omitted them here to avoid excessive detail.

[1] Theorems in mechanized versions, indicated with footnotes, often include additional properties and details that are not explicitly stated in this section. The theorems presented here focus on the key aspects for a simpler presentation, rather than listing all the details found in the corresponding mechanized theorems.

[2] "`Lemma one_step_oracle_refines_hardware`" in [25], where the validity of the oracle is defined with "`valid_oracle`" in the "`hardware_prop`" definition.

[3] "`Lemma one_step_hw_refines_oracle`" in [24].

To connect different machine models, we use a refinement proof library in the CompCert style [26] as discussed earlier. The main purpose of the refinement proofs using the library is to prove the completeness of behavior preservation between a program running on top of a multicore machine model and the same program running on an environmental machine model with a full core set ($\mathbb{S}_{core}$) using an "upward" ("backward" with the term in CompCert) simulation defined in the CompCert small step library [21]. The "upward" simulation allows any behavior of a lower-level machine to be allowed by a higher-level machine. However, "upward" simulations are generally harder to establish than "downward" (*i.e.*, "forward" with the term in CompCert) simulations because lower level machines tend to have more intermediate and/or concrete states than higher level machines as well as require more sub lemmas that are sometimes hard to prove (*e.g.*, the existence of valid transition when any state is provided as an initial state).[4] In this sense, CompCert provides a template to convert a "downward" simulation into an "upward" simulation to reduce the proof's complexity when machine models satisfy certain conditions, such as determinism and receptiveness of two machine models.[5] However, we cannot use this template to prove Theorem 3 due to the non-determinism of a multicore machine model in SimplMM. Therefore, we prove the theorem directly by using an "upward" simulation. In contrast, other simulation proofs in SimplMM use a "downward" simulation.

Theorem 4 (*Existence of an Env. Oracle*). *Assuming an environmental machine evaluation* ($\texttt{Mach}^{*}_{\texttt{env}[A]}\ \varepsilon_A\ est_A\ est'_A$), *where* $B \subseteq A$ *and* $A, B \subseteq \mathbb{S}_{core}$. *Then, there always exist the corresponding consistent environmental machine model steps associated with a core set* B *and the constructed environmental context,* ε_B, *with the simulation proofs. Also, the constructed environmental context exhibits the same behavior as the generated log during the evaluation with the core set* A.[6]

Theorem 4 demonstrates the ability to create an environmental context by capturing behaviors that affect shared states and are triggered by other CPUs in our multicore system ($\texttt{Mach}^{*}_{\texttt{env}[B]}\ \varepsilon_B\ est_B\ est'_B$). When B consists of only one core ($B = cid$), the theorem confirms the possibility of constructing an environmental context specifically for that core. The proof of Theorem 4 is similar to that of Theorem 2, with the additional requirement of constructing the environmental context for both scheduling events and atomic operation events.

Theorem 5 (*Env. Refines Env. with More Cores*). *Assuming there are sets of cores,* A *and* B *(that satisfy* $B \subseteq A$ *and* $A, B \subseteq \mathbb{S}_{core}$*), an evaluation of the environmental machine* ($\texttt{Mach}^{*}_{\texttt{env}[A]}\ \varepsilon_A\ est_A\ est'_A$) *performed within a valid environment context* (ε_B), *and an environmental state* (est_B) *that is connected to the initial state of the evaluation* (est_A) *with the refinement relation* ($match\_eestate\ est_A\ est_B$). *Then, it can be ensured that a valid updated environmental state* (est'_B) *always exists such that the transition* $\texttt{Mach}^{*}_{\texttt{env}[B]}\ \varepsilon_B\ est_B\ est'_B$ *is satisfied and the relationship* ($match\_eestate\ est'_A\ est'_B$) *remains true.*[7]

Theorem 5 establishes the refinement relationship between two environmental machine evaluations that correspond to different sets of cores. Similar to Theorem 3, we utilize the simulation proof library in CompCert to prove this theorem, but with a template that flips a "downward" simulation into an "upward" simulation.

To demonstrate the "downward" simulation between the two machine models, we need to show that given an evaluation step with a larger set of cores (A), there always exists an evaluation step with a smaller set of cores (B). To accomplish this, we utilize induction on each overlaying evaluation rule (evaluation with core set B) to demonstrate this property.

When the evaluation matches the underlying evaluation exactly (evaluation with core set A), it is evident that they both change the state in the same way with the same rule. However, when the evaluation is associated with a core that is not in B but is in A, the evaluation is represented as a SKIP rule in Fig. 9 for the overlaying evaluation. This evaluation matches multiple concrete steps in the underlying evaluation, including at least one step that generates an event and other steps that do not generate events.

Additionally, the theorem intentionally simplifies to obscure intricate technical challenges arising from a specific subtle case. In instances where the high-level model evaluates scheduling events that generate EYIELD, the low-level machine does not proceed with its evaluation, even though the high-level model advances. This introduces a technical challenge in establishing a refinement relation for cases where the source, high-level machine model, has fewer corresponding evaluations compared to the number of steps required in the target, low-level machine model. To address this, we introduce a measure function in the refinement relation, $match_e estate$, which defines the length of the log. This function checks the last event to determine whether it is EYIELD or not. Consequently, the relation specifies how to match low-level and high-level machine model states. The actual statement includes additional propositions to address this technical issue; however, these are omitted in Theorem 5 since they are unnecessary for understanding the high-level statement of the theorem.

Theorem 6 (*Single Core Refines Env. W. Single Core*). *Assuming that there are environmental machine evaluation steps* ($\texttt{Mach}^{*}_{\texttt{env}[\{cid\}]}\ \varepsilon_{\{cid\}}\ est_{\{cid\}}\ est'_{\{cid\}}$) *associated with a core identifier* cid ($cid \in \mathbb{S}_{core}$ *where* $\mathbb{S}_{core}$ *is the entire CPU set of the machine) and an environmental context* ($\varepsilon_{\{cid\}}$), *a valid environmental context* (ε_{cid}) *for a single core machine, and a single-core state* (sst) *related to the initial state of the evaluation* $est_{\{cid\}}$ *with a refinement relation* ($match\_sstate\ est_{\{cid\}}\ sst$). *Then, there always exists a valid updated environmental state* (sst') *such that the transition* $\texttt{Mach}_{\texttt{si}[cid]}\ \varepsilon_{cid}\ sst\ sst'$ *is satisfied and the relationship* ($match\_sstate\ est'_{\{cid\}}\ sst'$) *holds true.*[8]

Theorem 6 establishes the simulation between single-core evaluation and environmental evaluation steps with a singleton core set. This can be straightforwardly verified through a coercion between the environmental machine model state with a single core, cid, and a single-core machine model with the same core. The existence of a valid environmental context can also be shown through a simple coercion between a singleton set and a core identifier, so we do not explicitly state the theorem regarding a valid oracle in this step.

Although it is possible to state and prove the theorem that a single-core machine refines a multicore machine, we omit it in this section. This is because the statement and proofs are straightforward and can be derived by combining Theorems 3, 5, and 6. Furthermore, this refinement theorem is also addressed as a subpart of the theorems in the next section (Section 7). To establish the connection between the machine models and proofs from Sections 4, 5, and 6 to the CertiKOS proof in CCAL, it is necessary to identify appropriate concrete definitions for these abstractions. The next section provides a detailed explanation of how these machine models and proofs can be linked to the concrete layer definition in CCAL.

[4] "Record bsim_properties" in [21].

[5] "Lemma forward_to_backward_simulation" in [21]

[6] "Lemma one_step_single_env_refines_full_env" in [25], where the validity of the oracle is defined with "valid_oracle".

[7] "Lemma single_env_step_refines_full_env_step" in [24].

[8] "Lemma one_step_single_refines_env" in [24].

7. Link with CCAL

The separation of abstract hardware state and semantics in Section 4.1 greatly simplifies our machine models and refinement proofs. It allows us to abstract away from the detailed instructions that the machine must provide and focus on the essential aspects of the system while defining multiple machine syntax and semantics as well as showing refinement proofs between those models.

However, in order to connect the multicore machine model in SimplMM with the verified software that runs on concrete hardware based on a layer in CCAL, we need to introduce more concrete instructions and states into our machine models. To accomplish this, we need to establish a connection between the abstract definitions (abstract states and transition rules) in SimplMM and the verified concurrent operating systems CompCertX and CertiKOS.

This process involves two steps. First, instantiating all abstract states and transition rules in SimplMM, as shown in Fig. 5(a). This involves providing concrete definitions for the abstract states and specifying the behavior of the transition rules. Second, demonstrating that each evaluation of the connected CCAL layer precisely refines the corresponding evaluation of multiple machines in SimplMM. This involves proving that the concrete machine models in CCAL faithfully capture the behavior of the abstract machine models in SimplMM. By completing these steps, we establish a formal and precise link between the machine models and proofs in SimplMM and the verified concurrent operating systems in CCAL. This connection ensures that the verified software in CCAL is correctly aligned with the multicore machine models in SimplMM.

The mechanized version of state and instruction instantiation can be found in [27],[9] while the top-level multicore linking theorem is defined in [28].[10]

7.1. State instantiation

As outlined in Section 2, the state of a CCAL layer is defined as a tuple (ρ, mem, A) that includes registers, memory, and abstract states. The task of instantiating this state involves assigning each component or its sub-components into private and shared states, as depicted in Fig. 5(a). This step is accomplished through a case analysis of all components in the bottom layer of the verified software using CCAL, which is $L_{\text{mboot}}[cid, \varepsilon_{cid}]$, the bottom layer of CertiKOS in our example. For instance, all registers must fall into the private state. However, determining which regions of memory and abstract states belong to the shared and private components is more complex and depends on the definitions used in the L_{mboot} layer. Dividing memory into private and shared states can be challenging, especially when the memory allows for high expressiveness such as dynamic allocation.

Fortunately, CertiKOS, like all verification targets using CCAL, simplifies this process by statically allocating its kernel data structures to each memory block in CompCertX, which is a compiler inside CCAL. Therefore, it is possible to divide memory into multiple blocks and map each block to either a private or shared state. Sub fields of the abstract state in $L_{\text{mboot}}[cid, \varepsilon_{cid}]$ also require a case analysis to determine whether they should be fused into private or shared states.

One important field that should always be connected with the shared state in SimplMM, particularly with $\boldsymbol{log}_{\text{mc}}$, is the logical history field in the abstract state. This field is only a small part of the abstract state definition in $L_{\text{mboot}}[cid, \varepsilon_{cid}]$, as shown in Fig. 10. The logical history field (`log`) keeps track of the history of atomic evaluations on shared resources, which corresponds to $\boldsymbol{log}_{\text{mc}}$ in Fig. 5. Thus, these events are mapped to the shared resources and atomic events in Fig. 5. On the other hand, the CPU ID is considered a private state since we are only focusing on a single CPU at this level.

```
Record RData := mkRData {
  CPU_ID : Z;
  log : list event;
 …
}.
```

Fig. 10. Part of an abstract state in the bottom layer interface of CertiKOS.

```
Definition mboot : compatlayer (cdata RData) :=
 ⊕ get_CPU_ID ↦ gensem get_CPU_ID_spec
 ⊕ atomic_FAI ↦ primcall_atomic_FAI_compatsem
     atomic_FAI_spec
 …
```

Fig. 11. Part of `MBoot` layer interface: the bottom layer interface of CertiKOS.

7.2. Instruction instantiation

Furthermore, the abstract machine models presented in Sections 4 and 5 rely on abstract transition rules (as illustrated in Fig. 6(b)) and must be appropriately instantiated to connect machine models in SimplMM with a specific layer of CCAL, such as $L_{\text{mboot}}[cid, \varepsilon_{cid}]$ for CertiKOS. Similar to state instantiation, instantiating these transition rules requires a case analysis of each rule defined in $L_{\text{mboot}}[cid, \varepsilon_{cid}]$.

Among the various languages involved in the compilation process of CompCertX, we focus on LAsm parameterized by $L_{\text{mboot}}[cid, \varepsilon_{cid}]$, as this level reveals the most hardware-related features. The transition rules in LAsm include instructions and function calls, as well as primitives that trigger transitions in abstract states. Most instructions in LAsm (e.g., arithmetic operations and other basic operations) fall under private command transition rules, while cases for shared, release, and atomic transition rules are more complex. Unfortunately, the built-in instructions in LAsm, which are the same as those in the CompCert machine model, do not define concurrent behaviors. Therefore, $L_{\text{mboot}}[cid, \varepsilon_{cid}]$ includes additional instructions, such as CAS and FAI, as primitives to be used as building blocks for larger concurrent objects (e.g., spinlocks, queuing locks, etc.). However, this extensibility comes with a trade-off. Similar to the assembly instructions in CompCert, the bottom layer primitives have not been verified. Their semantics must be trusted, increasing the vulnerability of the verified software. Reducing the size of the trusted computing base (TCB) is a key goal for future work.

Fig. 11 illustrates components of $L_{\text{mboot}}[cid, \varepsilon_{cid}]$, which includes several primitives that cannot be expressed using bare instructions in CompCert. The definition of `mboot` represents a top-level set that defines these primitives in the mboot layer. Here, `compatlayer (cdata RData)` represents a layer type in CCAL. The $\oplus$ operator is used to compose the transition definitions of these primitives into a set, where each element consists of a primitive name and its specifications.[11] The specifications for the primitives, such as `get_CPU_ID_spec` and `atomic_FAI_spec` (FAI — fetch and increase), are pure Coq functions. These functions take a list of arguments and an initial state as inputs, and produce a result value and a final state as outputs. The specifications serve both as functional semantics and as specifications for the primitives. However, additional wrappers are needed to call these Coq functions in C and Assembly code written at a specific layer in

[9] This relates to multiple definitions, with two important ones being "`Global Instance hdsetting: HardWaredSetting`" for state instantiation and "`Global Program Instance hdsem`" for instruction instantiation.

[10] "`Theorem concurrent_backward_simulation`".

[11] Note that $\oplus$ is a polymorphic operator in CCAL and is used as a composition operator for multiple modules, such as in Theorem 1.

CCAL. These wrappers bridge the gap between Coq and C functions by handling the adjustment of Coq arguments to adhere to the C (or assembly) calling convention, which are the purpose of `gensem` and `primcall_atomic_FAI_compatsem`.

Mapping the abstract transition rules to SimplMM involves checking the primitive name and assigning each primitive to one of the four abstract transition rules. For example, the `get_CPU_ID` primitive is mapped to the private transition rule, while `atomic_FAI` is mapped to the atomic transition rule.

7.3. Connected theorems

With the proposed approaches and proofs, we have successfully connected SimplMM with a layer of CCAL, specifically LAsm parameterized with $L_{\text{mboot}}[cid, \epsilon_{cid}]$. The formal connection is defined in Theorem 7, which establishes the relationship between the machine model with the full CPU set ($\mathbb{S}_{core}$) and an environmental context for a specific CPU (cid). The theorem assumes that $[\![M]\!]_{\mathtt{Mach_{mc}}\left(L[\mathbb{S}_{core},\emptyset]\right)}$ defines the execution of program M on $\mathtt{Mach_{mc}}$, instantiating its abstract states and instructions with LAsm parameterized by layer L. On the other hand, $[\![M]\!]_{\text{LAsm}\left(L[cid,\varepsilon]\right)}$ defines the execution of program M on LAsm, using layer L parameterized by CPU ID cid and concurrent context ϵ. In the context of the theorem, $\mathtt{M_{certikos}}$ represents a CertiKOS kernel, $\mathtt{M_{user}}$ is a user program, and L_{mboot} is the bottom-most layer of the layered stacks in CertiKOS. Given them, the theorem shows that two models are formally consistent with each other. Demonstrating consistency involves employing the simulation proof denoted by $\sqsubseteq$, coupled with the refinement relation formed through the concatenation of multiple refinement relations between intermediate machines, denoted as R_{SimplMM}. Theorem 7 does not abstract any concrete code in $\mathtt{M_{certikos}}$ as abstracted layers, but rather converts the machine model from $\mathtt{Mach_{mc}}$ to LAsm.

The proofs follow the theorems presented in Section 6, with the addition of concrete definitions for $L_{\text{mboot}}[cid, \epsilon_{cid}]$ over LAsm in this section. However, it is not currently possible to formally connect Theorem 2 with Theorem 7 due to limitations in CompCert's proof libraries. The CompCert simulation proof library restricts the global state, in which our environmental context resides, as a static definition. We plan to explore alternative approaches to establish this connection in future work. While it may require modifications to the design of both SimplMM and CompCertX, we believe that with some adjustments, the desired connection can be achieved, although it may entail several proof modifications due to the complexity of the underlying tool, CompCert.

Theorem 7 (SimplMM *Refines* LAsm)**.**

$$[\![\mathtt{M_{certikos}} \oplus \mathtt{M_{user}}]\!]_{\mathtt{Mach_{mc}}\left(L_{\text{mboot}}[\mathbb{S}_{core},\emptyset]\right)} \sqsubseteq_{R_{\text{SimplMM}}} [\![\mathtt{M_{certikos}} \oplus \mathtt{M_{user}}]\!]_{\text{LAsm}\left(L_{\text{mboot}}[cid,\varepsilon_{cid}]\right)}$$

With the discussions and analysis, we have established the theorem of CertiKOS's correctness with both CCAL and SimplMM in Theorem 8. The theorem states that the behaviors of user program executions, running on a LAsm machine parameterized by a L_{syscall} layer, are formally consistent with behaviors of CertiKOS code and user program executions running on $\mathtt{Mach_{mc}}$. This consistency is achieved by instantiating all abstract states and instructions in $\mathtt{Mach_{mc}}$ using LAsm with a L_{mboot} layer. Establishing consistency also requires the utilization of the simulation proof, represented by $\sqsubseteq$, in conjunction with the refinement relation created through the concatenation of refinement relations for SimplMM and CertiKOS. The proof of the theorem is straightforward, as it combines Theorems 1 and 7.

Theorem 8 (CertiKOS *Correctness With* SimplMM)**.**

$$[\![\mathtt{M_{certikos}} \oplus \mathtt{M_{user}}]\!]_{\mathtt{Mach_{mc}}\left(L_{\text{mboot}}[\mathbb{S}_{core},\emptyset]\right)} \sqsubseteq_{(R_{\text{SimplMM}} \circ R_{certikos})} [\![\mathtt{M_{user}}]\!]_{\text{LAsm}\left(L_{\text{syscall}}[cid,\varepsilon]\right)}$$

Table 1
Statistics for the linking framework.

Components	LOC	
	Spec.	Proof
Abstract states & machine definitions	979	1164
Refinement proofs	353	1211
Oracle validity proofs	246	648
Instantiating states and instructions	788	1984
Instantiating refinement proofs	778	2743

8. Evaluation

Table 1 provides statistical information on the proof efforts for SimplMM and its integration with CertiKOS. The `coqwc` command was used to count the number of specification, proof, and comment lines in the Coq files. The number of specification lines includes all objects and statements in the Coq files, such as functions, propositional logics, and theorem and lemma statements. These specification lines define the structure and behavior of the formal system. The number of proofs corresponds to the number of tactic lines in the Coq files, as tactics are used to construct and guide the proofs that establish the validity of the specified properties. The proofs provide formal evidence of the correctness and consistency of the system. In our project, we have employed various user-defined Coq tactics and libraries from the CompCert and CertiKOS projects, and the details of these libraries are not explicitly listed in the table. Given the fact that these existing libraries can handle most proofs within our system, we do not need to introduce any significant user-defined libraries specific to this project.

The implementation of auxiliary definitions and multiple machine models comprises 969 lines of code. Furthermore, the demonstration of basic properties for these abstract machine models involves 1164 lines of code. The refinement proofs based on abstract definitions require 1564 lines of code — 353 lines for specifications and 1211 lines for proofs, while demonstrating the validity of multiple oracles accounts for 894 lines of code — 246 lines for specifications and 648 lines for proofs.

The most significant contribution to the lines of code comes from connecting the abstract states and instructions with concrete instances in the CCAL layer ($L_{\text{mboot}}[cid, \epsilon_{cid}]$) of CertiKOS. This process involves performing case analyses on each instruction and memory block in the machine model of the connected layer. Additionally, the definition and proof of wrappers at each level of the multiple machine models in SimplMM contribute to the overall lines of code. Integrating SimplMM with CompCert libraries and simulation proofs also requires conforming to the strict requirements of CompCert, including demonstrating responsiveness and determinism. This further adds to the lines of code in the implementation. The instantiation of refinement proofs also suffers from similar issues, resulting in redundant proofs at each level. Future work will focus on removing these redundancies to enhance the efficiency of the implementation.

There are also areas for potential improvement to consider. CompCertX and CertiKOS are constructed on an older version of Coq, specifically Coq 8.4.6. This limitation restricts our ability to leverage certain cutting-edge tactics such as 'lia', 'nia', etc. However, based on our previous experience with similar projects, we believe that their incorporation would not significantly reduce the overall complexity. Additionally, exploring the use of tools like Dafny, TLA, and/or Why3 could be advantageous. It is important to note, though, that the general reordering of concurrent behaviors, as expressed in our work through event reordering, remains a significant challenge that these tools have encountered thus far.

9. Related work and conclusion

Program Logics for Shared-Memory Concurrency. Multiple program logics [10,11,29–46] have been developed to support modular verification of concurrent programs. However, many of these logics, such as Turon [37] and Iris [10], are not suitable for our work as they focus on higher-order functions and complex non-blocking synchronization, which our approach does not address. Instead, we offer a unique contribution by mechanically connecting an explicit formal multicore machine model to a toolkit for large-scale formal verification. Our approach uses a global log, which is similar to the concept of compositional subjective history traces proposed by Sergey et al. [43]. While Total-TaDA [45] is capable of proving the total correctness of concurrent programs, it has not been automated with a proof assistant.

Parallel Composition in Concurrent Program Verification. Most concurrent languages utilize a parallel composition operator, $(C_1|C_2)$, to create and terminate instances. In our approach to connecting SimplMM and LAsm, we also deal with composition, but our method differs from previous works. Our parallel composition must always be carried out across the entire program P and all CPUs. This distinction enables us to analyze the behavior of CPUs within the concurrent environment, considering both past and future events. Our composition considers the semantics of running P on the environmental machine model, even if a CPU c has not been executed before. In such cases, the CPU will always consult its environment context to construct a global log.

Liang et al. [47–50] have proposed several rely-guarantee-based simulation (RGSim) approaches that support parallel composition and contextual refinement of concurrent objects. Contextual simulation proof in CCAL and SimplMM is based on the extended version of RGSim, which includes auxiliary states such as environmental contexts and shared logs. Specifically, the proofs in CCAL and SimplMM rely on the validity of environmental context behaviors, which are justified by the behaviors of possible updates from the current instances of the program (e.g., current CPU) on the shared log. This is possible since we assume that all cores in the kernel execute the same code, and the possible behaviors of other instances (e.g., CPUs) will be exactly the same as the possible behaviors of the current instance (e.g., CPU). For example, we assume that all cores/threads use the same spinlock algorithm to achieve mutual exclusion and use the same page memory management codes. The validity of the environmental context is a slightly related but separate concern from showing the existence of the environmental context, as demonstrated in Theorems 2 and 4. The environmental context behaviors capture the interactions and effects of the multicore system, which are crucial for reasoning about the system's behavior and ensuring correctness. CCAL and SimplMM have one more distinctive feature compared to other works. Existing RGSim systems are limited to reasoning about atomic objects in a single layer and cannot vertically connect different machine models as we do in our work. This vertical connection allows us to reason about the behavior of the entire system across different layers and machine models, providing a more comprehensive and flexible approach to formal verification. The Bedrock [51] project provides a verified toolkit for multithreaded programs, allowing dynamic allocation and connecting thread-local services with underlying library components. However, it does not introduce and vertically connect different machine models. CCR [52] also presents a new method for compositional software formal verification. However, they do not support concurrency. We expect that SimplMM can be easily connected with the tool when they support concurrency in the future.

Extending CompCert and Verified Compilation. A previous study [53] extends the original CompCert compiler [19] to support a compositional, thread-safe compilation of concurrent Clight programs. They adopt the interaction semantics proposed by Beringer et al. [54] which handles synchronization-primitive calls as external calls. However, this work is not suitable for large-scale software verification as it does not support a layered ClightX language like CompCertX. Additionally, the authors do not explicitly introduce multicore semantics in a formal manner, despite their interaction model being designed to support shared-memory concurrency.

Other studies have also modified the CompCert compiler to support separate compilation and composition, such as Kang et al. [55] and Ramananandro et al. [56], but they do not address concurrency. Other works in verified compilation [57–60] also lack support for concurrent and/or compositional systems in a layered manner. CompCertTSO [17] presents the CompCert extension that supports x86 TSO model. However, they do not provide compositional approach as CCAL and CompCertX does, thus there are no large-scale verified software with the extension yet. Extending SimplMM to support TSO model is one of our future directions. There are also recent extensions of CompCert [7,8], but they focus on compositionality rather than the underlying model like SimplMM.

CCAL As our target CPU-local machine model, we use CCAL, which is an extended version of CAL [18]. Both CCAL and our machine model connections provide contextual correctness properties using termination-sensitive forward simulation techniques [19,61], which is also described in Section 2.2. This makes it easy to establish a connection between the verification on CCAL and SimplMM. Furthermore, this property is stronger than the partial or total correctness properties guaranteed by Hoare logic-style verification [62–65]. Two previous works, CIVL [44] and FCSL [11], proposed a way to build and prove concurrent programs in a layered manner, similar to CCAL. To the best of our knowledge, neither of them supports the formal linking of different machine models that we provide. However, we believe that our approach for defining formally connected multicore machine models could also be applied in their tools.

Multithreaded Library (Kernel) Verification. A significant amount of work has been done in the area of kernel verification, with several efforts aimed at proving the correctness of multithreaded library (kernel) implementations. For example, seL4 [66,67], Verve [68], and hyperkernel [69] have tackled the challenge of verifying kernels. Among them, Xu et al. [46] developed a new verification framework that combines RGSim and Feng et al.'s program logic [70] to reason about interrupts. They applied the framework to verify multiple key modules written in C in μC/OS-II, a preemptive kernel. Other works, such as seL4 [66], Verve [68], and hyperkernel [69], have aimed to prove several properties using a per-core big kernel lock, which provides limited concurrency support. Additionally, the Verisoft team used VCC [71] to verify spinlocks in a hypervisor, directly postulating a Hoare logic instead of relying on operational semantics for C. However, none of these works have focused on the formal linking between a multicore machine model and a local machine model, which is a crucial aspect of our approach. SeKVM [22] verify parts of KVM based using CCAL with assuming relaxed memory model as their underlying hardware. However, they omit providing concrete semantics and connections for those underlying model as SimplMM does. One future direction is to extend SimplMM to support more complex hardware models, such as the underlying hardware model of SeKVM.

10. Conclusion and future works

This paper presents SimplMM, a framework for intermediate and multicore machine models, that serves as a subsystem of CCAL, a toolkit for formal verification of C/Asm programs. We also demonstrate the relationship between the multicore machine model in SimplMM and LAsm, the machine model used by CCAL to build their verified software stack with CompCertX. The main difference between the multicore machine and LAsm is that SimplMM explicitly represents other CPUs, while LAsm abstracts them as a concurrent context that requires validation. To establish the connection between SimplMM and LAsm, we

use state and command abstractions, along with multiple intermediate machine models, and provide a formal connection between all models. To showcase the effectiveness of SimplMM in conjunction with CCAL, we extend an existing operating system verification, CertiKOS, with fine-grained shared memory concurrency.

In the future, we plan to expand our approach to support more relaxed machine models, including weak memory models and dynamic memory allocation, and integrate it with other existing verification tools.

CRediT authorship contribution statement

Jieung Kim: Conceptualization, Methodology, Software. **Ronghui Gu:** Conceptualization, Methodology, Software. **Zhong Shao:** Conceptualization, Methodology.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jieung Kim reports financial support was provided by National Science Foundation. Zhong Shao reports financial support was provided by National Science Foundation. Ronghui Gu reports financial support was provided by National Science Foundation. Jieung Kim reports financial support was provided by Defense Advanced Research Projects Agency. Zhong Shao reports financial support was provided by Defense Advanced Research Projects Agency. Ronghui Gu reports financial support was provided by Defense Advanced Research Projects Agency. Ronghui Gu and Zhong Shao are co-founders of CertiK. Jieung Kim was previously employed by Google.

Data availability

No data was used for the research described in the article.

Acknowledgments

This research is based on work supported in part by National Science Foundation (NSF, USA) grants 1521523 and 1715154 and DARPA, USA grants FA8750-12-2-0293, FA8750-16-2-0274, and FA8750-15-C-0082. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work was also supported by INHA UNIVERSITY Research Grant.

References

[1] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, J. Szefer, Survey of approaches and techniques for security verification of computer systems, J. Emerg. Technol. Comput. Syst. 19 (1) (2023) http://dx.doi.org/10.1145/3564785.

[2] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, T. Ramananandro, Certified concurrent abstraction layers, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2018, 2018.

[3] C. Hawblitzel, J. Howell, J.R. Lorch, A. Narayan, B. Parno, D. Zhang, B. Zill, Ironclad apps: End-to-end security via automated full-system verification, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI '14, USENIX Association, Berkeley, CA, USA, 2014, pp. 165–181, URL http://dl.acm.org/citation.cfm?id=2685048.2685062.

[4] C. Hawblitzel, J. Howell, M. Kapritsos, J.R. Lorch, B. Parno, M.L. Roberts, S. Setty, B. Zill, IronFleet: Proving practical distributed systems correct, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 1–17, http://dx.doi.org/10.1145/2815400.2815428, URL http://doi.acm.org/10.1145/2815400.2815428.

[5] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS: An extensible architecture for building certified concurrent OS kernels, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI '16, USENIX Association, Berkeley, CA, USA, 2016, pp. 653–669, URL http://dl.acm.org/citation.cfm?id=3026877.3026928.

[6] H. Chen, X.N. Wu, Z. Shao, J. Lockerman, R. Gu, Toward compositional verification of interruptible OS kernels and device drivers, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, ACM, New York, NY, USA, 2016, pp. 431–447, http://dx.doi.org/10.1145/2908080.2908101, URL http://doi.acm.org/10.1145/2908080.2908101.

[7] J. Koenig, Z. Shao, CompCertO: Compiling certified open c components, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, in: PLDI 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1095–1109, http://dx.doi.org/10.1145/3453483.3454097.

[8] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, C. Hur, CompCertM: CompCert with C-assembly linking and lightweight modular verification, Proc. ACM Program. Lang. 4 (POPL) (2020).

[9] A. Appel, Verified software toolchain, in: G. Barthe (Ed.), ESOP'11: European Symposium on Programming, in: LNCS, vol. 6602, Springer, 2011, pp. 1–17.

[10] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, D. Dreyer, Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning, in: Proc. 42nd ACM Symposium on Principles of Programming Languages, POPL'15, 2015, pp. 637–650.

[11] I. Sergey, A. Nanevski, A. Banerjee, Mechanized verification of fine-grained concurrent programs, in: Proc. 2015 ACM Conference on Programming Language Design and Implementation, PLDI'15, 2015, pp. 77–87.

[12] J.R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M.D. Ernst, T. Anderson, Verdi: A framework for implementing and formally verifying distributed systems, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, ACM, New York, NY, USA, 2015, pp. 357–368, http://dx.doi.org/10.1145/2737924.2737958, URL http://doi.acm.org/10.1145/2737924.2737958.

[13] D. Woos, J.R. Wilcox, S. Anton, Z. Tatlock, M.D. Ernst, T. Anderson, Planning for change in a formal verification of the raft consensus protocol, in: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, in: CPP 2016, ACM, New York, NY, USA, 2016, pp. 154–165, http://dx.doi.org/10.1145/2854065.2854081, URL http://doi.acm.org/10.1145/2854065.2854081.

[14] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M.F. Kaashoek, N. Zeldovich, Using crash hoare logic for certifying the FSCQ file system, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 18–37, http://dx.doi.org/10.1145/2815400.2815402, URL http://doi.acm.org/10.1145/2815400.2815402.

[15] X. Leroy, The CompCert C compiler, 2005–2013, http://compcert.inria.fr/compcert-C.html.

[16] J.Y. Shin, J. Kim, W. Honoré, H. Vanzetto, S. Radhakrishnan, M. Balakrishnan, Z. Shao, WormSpace: A modular foundation for simple, verifiable distributed systems, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 299–311, http://dx.doi.org/10.1145/3357223.3362739.

[17] J. Sevcík, V. Vafeiadis, F.Z. Nardelli, S. Jagannathan, P. Sewell, CompCertTSO: A verified compiler for relaxed-memory concurrency, J. ACM 60 (3) (2013).

[18] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X.N. Wu, S.C. Weng, H. Zhang, Y. Guo, Deep specifications and certified abstraction layers, in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 595–608, http://dx.doi.org/10.1145/2676726.2676975, URL http://doi.acm.org/10.1145/2676726.2676975.

[19] X. Leroy, The CompCert verified compiler, 2005–2023, http://compcert.inria.fr/.

[20] The Coq development team, The Coq proof assistant, 1999–2018, http://coq.inria.fr.

[21] X. Leroy, Module smallstep, the CompCert verified compiler, 2005–2023, https://compcert.org/doc/html/compcert.common.Smallstep.html.

[22] S.W. Li, X. Li, R. Gu, J. Nieh, J. Zhuang Hui, A secure and formally verified Linux KVM hypervisor, in: 2021 IEEE Symposium on Security and Privacy, SP, 2021, pp. 1782–1799, http://dx.doi.org/10.1109/SP40001.2021.00049.

[23] J. Kim, V. Sjöberg, R. Gu, Z. Shao, Safety and liveness of MCS lock–layer by layer, in: 15th Asian Symposium on Programming Languages and Systems, APLAS 2017, in: Lecture Notes in Computer Science, Springer, 2017.

[24] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: SimplMM refinement proofs. https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmclib.Concurrent_Linking_Prop.html.

[25] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS Artifact: SimplMM Oracle Existence proofs, https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmclib.Concurrent_Linking_Additional_Prop.html.

[26] X. Leroy, Formal certification of a compiler back-end or: Programming a compiler with a proof assistant, in: Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, 2006.

[27] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: SimplMM and CertiKOS abstract definition instantiation. https://certikos.github.io/certikos-artifact/html/mcertikos.multicore.semantics.HWSemImpl.html.

[28] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: SimplMM and CertiKOS linking. https://certikos.github.io/certikos-artifact/html/mcertikos.multicore.Concurrent_Linking.html.

[29] T. Dinsdale-Young, M. Dodds, P. Gardner, M.J. Parkinson, V. Vafeiadis, Concurrent abstract predicates, in: ECOOP'10, 2010, pp. 504–528.

[30] P.W. O'Hearn, Resources, concurrency and local reasoning, in: Proc. 15th International Conference on Concurrency Theory, CONCUR'04, 2004, pp. 49–67.

[31] S. Brookes, A semantics for concurrent separation logic, in: Proc. 15th International Conference on Concurrency Theory, CONCUR'04, 2004, pp. 16–34.

[32] X. Feng, R. Ferreira, Z. Shao, On the relationship between concurrent separation logic and assume-guarantee reasoning, in: Proc. 16th European Symposium on Programming, ESOP'07, 2007, pp. 173–188.

[33] V. Vafeiadis, M. Parkinson, A marriage of rely/guarantee and separation logic, in: Proc. 18th International Conference on Concurrency Theory, CONCUR'07, 2007, pp. 256–271.

[34] X. Feng, Local rely-guarantee reasoning, in: Proc. 36th ACM Symposium on Principles of Programming Languages, POPL'09, 2009, pp. 315–327.

[35] B. Jacobs, F. Piessens, Expressive modular fine-grained concurrency specification, in: Proc. 38th ACM Symposium on Principles of Programming Languages, POPL'11, 2011, pp. 133–146.

[36] A. Gotsman, N. Rinetzky, H. Yang, Verifying concurrent memory reclamation algorithms with grace, in: Proc. 22nd European Symposium on Programming, ESOP'13, 2013, pp. 249–269.

[37] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, D. Dreyer, Logical relations for fine-grained concurrency, in: Proc. 40th ACM Symposium on Principles of Programming Languages, POPL'13, 2013, pp. 343–356.

[38] A. Turon, D. Dreyer, L. Birkedal, 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, 2013, pp. 377–390.

[39] R. Ley-Wild, A. Nanevski, Subjective auxiliary state for coarse-grained concurrency, in: Proc. 40th ACM Symposium on Principles of Programming Languages, POPL'13, 2013, pp. 561–574.

[40] A. Nanevski, R. Ley-Wild, I. Sergey, G.A. Delbianco, Communicating state transition systems for fine-grained concurrent resources, in: Proc. 23rd European Symposium on Programming, ESOP'14, 2014, pp. 290–310.

[41] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, H. Yang, Views: Compositional reasoning for concurrent programs, in: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, ACM, New York, NY, USA, 2013, pp. 287–300, http://dx.doi.org/10.1145/2429069.2429104, URL http://doi.acm.org/10.1145/2429069.2429104.

[42] P.D.R. Pinto, T. Dinsdale-Young, P. Gardner, Tada: A logic for time and data abstraction, in: Proc. 28th European Conference on Object-Oriented Programming, ECOOP'14, 2014, pp. 207–231.

[43] I. Sergey, A. Nanevski, A. Banerjee, Specifying and verifying concurrent algorithms with histories and subjectivity, in: Proc. 24th European Symposium on Programming, ESOP'15, 2015, pp. 333–358.

[44] C. Hawblitzel, E. Petrank, S. Qadeer, S. Tasiran, Automated and modular refinement reasoning for concurrent programs, in: Proc. 27th International Conference on Computer Aided Verification, CAV'15, 2015, pp. 449–465.

[45] P.D.R. Pinto, T. Dinsdale-Young, P. Gardner, J. Sutherland, Modular termination verification for non-blocking concurrency, in: Proc. 25th European Symposium on Programming, ESOP'16, 2016, pp. 176–201.

[46] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, Z. Li, A practical verification framework for preemptive OS kernels, in: Proc. 28th International Conference on Computer Aided Verification (CAV'16), Part II, 2016, pp. 59–79.

[47] H. Liang, X. Feng, M. Fu, A rely-guarantee-based simulation for verifying concurrent program transformations, in: Proc. 39th ACM Symposium on Principles of Programming Languages, POPL'12, 2012, pp. 455–468.

[48] H. Liang, X. Feng, Z. Shao, 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, 2014, pp. 65:1–65:10.

[49] H. Liang, X. Feng, A program logic for concurrent objects under fair scheduling, in: Proc. 43rd ACM Symposium on Principles of Programming Languages, POPL'16, 2016, pp. 385–399.

[50] H. Liang, X. Feng, Progress of concurrent objects with partial methods, Proc. ACM Program. Lang. 2 (POPL) (2017) 20:1–20:31, http://dx.doi.org/10.1145/3158108, URL http://doi.acm.org/10.1145/3158108.

[51] A. Chlipala, Mostly-automated verification of low-level programs in computational separation logic, in: PLDI'11, 2011, pp. 234–245.

[52] Y. Song, M. Cho, D. Lee, C.K. Hur, M. Sammler, D. Dreyer, Conditional contextual refinement 7 (popl), 2023, http://dx.doi.org/10.1145/3571232.

[53] G. Stewart, L. Beringer, S. Cuellar, A.W. Appel, Compositional CompCert, in: Proc. 42nd ACM Symposium on Principles of Programming Languages, POPL'15, 2015, pp. 275–287.

[54] L. Beringer, G. Stewart, R. Dockins, A.W. Appel, Verified compilation for shared-memory c, in: Proc. 23rd European Symposium on Programming, ESOP'14, 2014, pp. 107–127.

[55] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, V. Vafeiadis, Lightweight verification of separate compilation, in: Proc. 43rd ACM Symposium on Principles of Programming Languages, POPL'16, 2016, pp. 178–190.

[56] T. Ramananandro, Z. Shao, S.C. Weng, J. Koenig, Y. Fu, A compositional semantics for verified separate compilation and linking, in: Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP '15, ACM, New York, NY, USA, 2015, pp. 3–14, http://dx.doi.org/10.1145/2676724.2693167, URL http://doi.acm.org/10.1145/2676724.2693167.

[57] A. Lochbihler, Verifying a compiler for Java threads, in: ESOP, 2010, pp. 427–447.

[58] J. Ševčík, V. Vafeiadis, F.Z. Nardelli, S. Jagannathan, P. Sewell, Relaxed-memory concurrency and verified compilation, in: POPL, 2011, pp. 43–54.

[59] J. Zhao, S. Nagarakatte, M.M. Martin, S. Zdancewic, Formal verification of SSA-based optimizations for LLVM, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, New York, NY, USA, 2013, pp. 175–186, http://dx.doi.org/10.1145/2491956.2462164, URL http://doi.acm.org/10.1145/2491956.2462164.

[60] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M.D. Shin, Y. Kim, S. Cho, J. Choi, C.K. Hur, K. Yi, Crellvm: Verified credible compilation for LLVM, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2018, ACM, New York, NY, USA, 2018, pp. 631–645, http://dx.doi.org/10.1145/3192366.3192377, URL http://doi.acm.org/10.1145/3192366.3192377.

[61] N.A. Lynch, F.W. Vaandrager, Forward and backward simulations: I. Untimed systems, Inform. and Comput. 121 (2) (1995) 214–233.

[62] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[63] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proc. 17th IEEE Symposium on Logic in Computer Science, LICS'02, 2002, pp. 55–74.

[64] M. Barnett, B.Y.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Proc. 4th Symposium on Formal Methods for Components and Objects, FMCO'05, 2005, pp. 364–387.

[65] A. Nanevski, G. Morrisett, L. Birkedal, Polymorphism and separation in Hoare type theory, in: Proc. 2006 ACM SIGPLAN International Conference on Functional Programming, ICFP'06, 2006, pp. 62–73.

[66] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, et al., seL4: Formal verification of an OS kernel, in: Proc. 22nd ACM Symposium on Operating System Principles, SOSP'09, 2009, pp. 207–220.

[67] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser, Comprehensive formal verification of an OS microkernel, ACM Trans. Comput. Syst. 32 (1) (2014) 2:1–2:70.

[68] J. Yang, C. Hawblitzel, 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, 2010, pp. 99–110.

[69] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, X. Wang, Hyperkernel: Push-button verification of an OS kernel, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, ACM, New York, NY, USA, 2017, pp. 252–269, http://dx.doi.org/10.1145/3132747.3132748, URL http://doi.acm.org/10.1145/3132747.3132748.

[70] X. Feng, Z. Shao, Y. Dong, Y. Guo, Certifying low-level programs with hardware interrupts and preemptive threads, in: Proc. 2008 ACM Conference on Programming Language Design and Implementation, PLDI'08, 2008, pp. 170–182.

[71] D. Leinenbach, T. Santen, Verifying the Microsoft Hyper-V hypervisor with VCC, in: Proc. 2nd World Congress on Formal Methods, 2009, pp. 806–809.