Journal of Systems Architecture 147 (2024) 103046

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

ThreadAbs: A template to build verified thread-local interfaces with software scheduler abstractions☆

Jieung Kim [a,*], Jérémie Koenig [b], Hao Chen [b], Ronghui Gu [c], Zhong Shao [b]

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

ARTICLE INFO

Keywords:
Multithreaded program
Scheduler
Thread abstraction
Assembly model
System software
Operating system
Verified compiler
Software formal verification
Formal semantics
Formal verification framework
Software engineering
Software reliability
Secure software
Software correctness
Bug-free software
Concurrency
Parallel computing
Shared memory concurrency
Linearizability

ABSTRACT

This paper presents ThreadAbs, an extension of the layer-based software formal verification toolkit CCAL (Gu et al., 2018). ThreadAbs is specifically designed to provide better expressiveness and proof management for *thread abstraction* in multithreaded libraries. Thread abstraction isolates the behavior of each thread from others when providing a top-level formal specification for software. Compared to the original CCAL, ThreadAbs offers significant improvements in this regard.

CCAL is a verification framework that enables a layered approach to building certified software, as demonstrated by multiple examples (Gu et al. 2016; Li et al. 2021; Shin et al. 2019). Obviously, its main targets usually include multithread libraries, which support significant improvement in the utilization and isolation of system resources. However, it poses new challenges for formal verification. Firstly, it requires a sudden change in the granularity of concurrency during the implementation and verification of the target software. Typically, systems are associated with software schedulers that are built on top of several underlying components in the system (*e.g.*, thread spawn, yield, sleep, and wake-up). Due to the software scheduler, these systems can be divided into low-level components consisting of modules that the software scheduler depends on (*e.g.*, allocators for shared resources and scheduling queues) and high-level components that use software schedulers (*e.g.*, condition variables, semaphores, and IPCs). Therefore, software formal verification on those systems has to provide proper method to deal with those distinct features, which is usually abstracting other threads' behavior as much as possible to provide an independent thread model and its formal specification. Secondly, it requires handling side effects from other threads, such as dynamic resource allocation from parents with proper isolation of all threads from each other.

CCAL has limited support for two crucial aspects of formal verification in multithreaded systems. Firstly, its previous thread abstraction method does not handle the side effects caused by a parent thread during dynamic initial state allocation properly. Secondly, the proofs produced by CCAL are tied to CertiKOS, which makes it challenging to use them for similar proofs that use CCAL as their verification toolkit. To address these issues, we introduce ThreadAbs, a new mechanized methodology that provides proper thread abstraction to reason about multithreaded programs in conjunction with CCAL. We also extend the previous CertiKOS proof with ThreadAbs to demonstrate its usability and expressiveness.

1. Introduction

Multithreading is a commonly employed technique to optimize software performance by leveraging the computational capabilities of hardware. However, ensuring the correctness of multithreaded software presents significant challenges. Debugging software for correctness is already demanding, and it becomes even more daunting in concurrency-related scenarios, such as multithreading. The potential execution interleavings makes exhaustive testing impractical, and identifying and reproducing bugs is particularly challenging without precise knowledge of the interleaving order. Formal verification offers a compelling solution by providing assurance that software complies with its specifications, covering all possible interleaving cases. Consequently,

☆ ThreadAbs: A template to build verified thread-local interfaces with software scheduler abstractions.

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

https://doi.org/10.1016/j.sysarc.2023.103046
Received 9 May 2023; Received in revised form 20 October 2023; Accepted 2 December 2023
Available online 6 December 2023

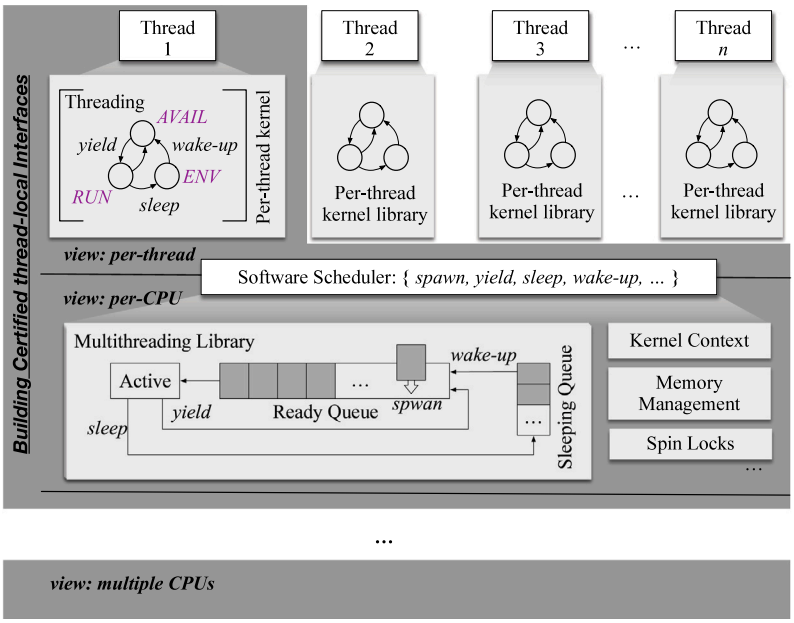

Fig. 1. Top-level Theorem.

we strongly advocate for the use of a verified multithreaded library as the foundation for building trusted threaded programs that fully harness the underlying hardware resources.

Nevertheless, verifying large-scale multithreaded libraries demands significant effort, since interactions among multiple instances within the program amplify its complexity. Additionally, the verification process must include verified compilation to ensure consistency between the verified program and the code running on the machine. To address these challenges, prior research has explored modular and compositional reasoning in various contexts [1–12], including concurrency. Among these approaches, Certified Concurrent Abstraction Layers (CCAL) by the CertiKOS team stands out. CCAL employs a layered approach, replacing lower-level implementations with abstract machines in higher layers. This compositional method has successfully verified large-scale software like CertiKOS. However, CCAL does not completely address another challenge: how to offer a thread abstraction when creating certified specifications tailored to application programmers while still accommodating essential features found in multithreaded libraries. Some operating systems, like CertiKOS, depend on dynamic information to allocate resources (*e.g.*, memory pages) for each thread when it is created. These dynamic behaviors are crucial for ensuring safe and efficient resource management.

Fig. 1 illustrates the typical structure of thread libraries. This structure involves managing the *view change* in thread abstraction, where the execution environment model is adjusted. These libraries vary in their implementations based on their underlying models and intended features. However, a common necessity in these libraries is the inclusion of software schedulers (*e.g.*, thread spawn, yield, sleep, and wake-up). In the absence of a scheduler in the lower parts (up to view:per-CPU), all threads on one CPU can be treated as a single program, managed by lower-level software that provides essential system services (*e.g.*, spinlocks and memory management), without considering multithreading. On the other hand, introducing a software scheduler alters how a program or system component interacts with the rest of the system. With the introduction of software schedulers, a kernel program may be concurrently invoked on behalf of multiple threads due to potential context switches. It is possible to provide a thread-local interface for this part, necessitating accounting for the behaviors of other threads as an abstract environment isolated from the current thread under verification. This approach avoids retaining all concrete details of those evaluations as done in the lower parts. Managing the connection between these two different views requires proper machine models, program modules, concurrent contexts, and contextual refinements. These are aspects that previous CCAL could not address adequately.

Hence, we introduce ThreadAbs as a solution to overcome CCAL's limitations by addressing complexity at its source through several methods. Firstly, ThreadAbs defines multiple intermediate machine models that handle multiple thread scheduling primitives as built-in language-level syntax and semantics. This decouples thread scheduling-related primitives from other operations. Additionally, the introduced machine models can be parameterized by different numbers of concrete and environmental (abstract) instances. These models encompass scenarios ranging from programs containing all threads for each CPU in their configuration to those with only a partial set of threads, including the case of a single-threaded program. Secondly, ThreadAbs provides generic layer templates for these machine models. These templates serve as abstract layers with defined types that can be applied across various machine models. Depending on the verification target, users can instantiate the generic layer template with any layers that meet minimum conditions, thereby enhancing flexibility in our framework. This approach achieves two objectives. It maintains consistency with CCAL by establishing parameterized certified states and transitions for the machine model with a layer definition. It also empowers users to streamline proofs using the templates, reducing the effort required for thread abstraction. Thirdly, ThreadAbs offers contextual refinement proof templates between machine models and their associated layer templates. This feature allows users to construct program-specific proofs based on the proofs already provided by ThreadAbs. Lastly, ThreadAbs extends CompCertX to integrate the thread-local interface with the verified compiler. It handles dynamically initialized thread states correctly and preserves thread isolation properties.

Our thread abstraction methodology offers unique features that make it suitable for verifying large concurrent systems. Firstly, our multithreaded machine semantics are generic and can be applied to any software that meets specific conditions, such as using certain primitives like *yield* and *sleep* for thread context switching. This flexibility allows it to be used with a wide range of programs, especially those compatible with CCAL. Secondly, the library can be seamlessly linked with the Assembly model for CompCertX, establishing a complete formal connection between C and Assembly code while providing data abstraction for detailed memory representation. This connectivity holds regardless of the module's complexity or level. Lastly, our thread abstraction can

handle complex dependencies among multiple data structures in large concurrent programs. This includes dependencies between shared and private objects, with reasonable restrictions. To illustrate the usability and expressiveness of our framework, we extend CertiKOS with ThreadAbs. As part of our case study, we demonstrate the mutual exclusiveness of multiple threads' private data in the system. This implies that private data remains isolated from other threads. For instance, our thread abstraction proof automatically ensures the mutual exclusiveness of dynamically allocated memory pages through two-level page allocation tables. This ensures that each thread cannot directly access the private data of other threads in their private memories.

In conclusion, this paper makes the following contributions:

- We introduce a thread abstraction framework centered around a generic thread configuration. This framework is tailored for the x86 machine and supports cooperative-style scheduling functions like *yield* and *sleep*. It allows for dynamic resource allocation and exhibits the versatility to be applied in multiple programs.
- As part of the proposed framework, we present machine models that establish a connection between a scheduler's implementation and an abstract thread pool semantics.
- We integrate these frameworks with the generic machine model of CompCertX, a variant of CompCert, and demonstrate how to utilize the framework for large-scale program verification.
- We offer guidelines for users, distinguishing between the aspects that the framework automatically guarantees and those that users must adhere to and ensure.
- We provide an evaluation of our framework, including statistics on the lines of code within our framework, the lines of code for our case study, and valuable lessons learned.

The thread abstraction framework introduced here is referred to as ThreadAbs.

The remainder of this article is organized as follows. Section 2 presents the overall idea of ThreadAbs. Sections Section 3, 4, and 5 show all the steps in ThreadAbs by presenting formal rules, proofs, and precise explanations about the parts that users have to provide to use ThreadAbs. Section 6 provides an example demonstrating how to enable ThreadAbs, and Section 7 discusses proof efforts, our experience, and the current limitations. Related work is discussed in Section 8, and Section 9 concludes our paper. Furthermore, the complete proof is accessible at https://zenodo.org/record/8312502, which is an identical version presented in a user-friendly artifact page at https://certikos.github.io/certikos-artifact/.

2. Overview

2.1. Certified concurrent abstraction layers

Certified Concurrent Abstraction Layers (CCAL) [10] is a toolkit explicitly designed for verifying concurrent programs written in C and assembly. Although our work, ThreadAbs, is an extension and a subcomponent of CCAL, for simplicity and clarity, we will treat ThreadAbs as a distinct entity. CCAL has proven successful in various software formal verification projects [13–16]. With the CCAL toolkit, users can effectively dissect large concurrent programs into smaller, manageable components and verify each component individually. After each piece is verified, they can be assembled to establish a top-level correctness theorem for the entire program. This modular approach empowers users to tackle the challenges presented by complex concurrent programs. In this section, we offer a concise overview of CCAL and its remarkable capabilities.

Overview. In CCAL, all layers utilize a state transition machine parameterized with the common underlying machine model LAsm, an extension of CompCert's machine model. To simplify, let us discuss how CCAL operates without delving into the details of LAsm. The primary goal of CCAL is to decompose the verification target aggressively, linearize dependencies between components, and introduce more abstracted layers by substituting low-level implementations with high-level specifications supported by formal proofs. Proofs within CCAL are predicates that span two layers linked by a refinement relation (R). These layers include a participant ID, an environmental context, a program module denoted as $\mathtt{M}_{\mathtt{high}}$ implementing the overlay $L_{\mathtt{high}}$, and an underlay referred to as $L_{\mathtt{low}}$. These proofs can be represented as "$L_{\mathtt{low}}[id, \varepsilon_{low}] \vdash_R \mathtt{M}_{\mathtt{high}} : L_{\mathtt{high}}[id, \varepsilon_{high}]$." This representation encapsulates several key concepts. First, the participant ID links to a specific instance identified by id within concurrent programs, which may correspond to different aspects based on the granularity of concurrency, such as a CPU ID or thread ID. The environmental context (ε) encompasses the abstracted behaviors of other instances within concurrent programs. By following these approaches, layers in CCAL contain a mechanized proof object, typically implemented in Coq. This enables CCAL to ensure that $L_{\mathtt{high}}[id, \varepsilon_{high}]$ exhibits the same observable behaviors as $\mathtt{M}_{\mathtt{high}}$ when applied over $L_{\mathtt{low}}[id, \varepsilon_{low}]$, as specified by the refinement relation R.

States and transitions. A state transition machine consists of both state and transition definitions. The state of each layer within CCAL is characterized by a state composed of four key elements: a register set (ρ), a memory (m), an abstract state (a), and a global log (l). Transitions in these layers fall into two categories. The first category involves transitions defined in CompCert, which resemble C/Assembly instructions, including function calls. These transitions primarily modify ρ and m in the layer's state, ensuring compilation correctness as CCAL is constructed based on CompCert. The second category covers primitive transitions. Layers define a set of primitives, represented as a partial map from a primitive ID to its specification. These primitives establish *atomic* single-step transition rules that mainly impact a and l in the state. They account for intricate interleavings with participants like other CPUs and threads within the system. Within these primitives, CCAL models all shared states using a single global log. This log is updated by all transitions accessing shared states, whether by the currently focused participant or other participants in the system. The global log is pivotal in concurrent reasoning, as it preserves the history of state updates, unlike a simplistic state structure (*e.g.*, a record type). This historical information is vital for proving and abstracting concurrent properties, including safety, linearizability, and liveness.

To model interleaving with other participants in the global state, a CCAL layer employs an environmental context. This context, a function taking the current shared state and a participant ID (*e.g.*, CPU ID or thread ID), determines how other participants may modify the state before the current participant's action. The transitions within the environmental context are intentionally designed to be adaptable, accommodating a wide range of scenarios. These transitions adhere to the system's invariants, as discussed in earlier works [5], with minimal constraints imposed.

Verified compilation. It is clear that most programmers prefer not to develop large-scale software using Assembly language. Fortunately, CCAL is built on top of CompCertX [17], which is a variation of the verified C compiler CompCert [18]. This platform provides a methodology that enables users to seamlessly integrate C and Assembly code into their software development process. CompCertX also introduces a C model, referred to as clightX, and an assembly model known as LAsm. These models incorporate layer definitions as integral components of their transition mechanisms. As a result, within the framework of CCAL, software modules can be authored in either C or Assembly, and C functions within these modules are automatically compiled into Assembly code.

Refinement proofs. With these definitions, layer building in CCAL abstracts certain aspects of states and transitions within a given low-level program module through contextual refinement proofs. More formally, the module $\mathtt{M}_{\mathtt{high}}$ (or its compiled counterpart, denoted as $(\!|\mathtt{M}_{\mathtt{high}}|\!)$), which operates on $L_{\mathtt{low}}[id, \varepsilon_{low}]$, transforms into $L_{\mathtt{high}}[id, \varepsilon_{high}]$ through the abstraction of states and instructions. Low-level states and transitions–m_{low}, a_{low}, and l_{low}–are abstracted and simplified into high-level abstract states and primitives—a_{high} and l_{high}. This process involves concealing or simplifying certain aspects of the low-level module's behavior while ensuring that the high-level module behaves correctly in relation to the specified abstractions. Contextual refinement, the underlying technique of CCAL, demonstrates that these abstractions are mathematically correct and is based on a refinement relation (R) between two states in the underlay and overlay. Definition 1 provides a formal definition of contextual refinement between two layers while also taking verified compilation into account.

Definition 1 (*Contextual Refinement*)**.** For any participant ID (id), context program ($\mathtt{Ctxt}$), oracles for two layers (ε_{low} and ε_{high}), the following is true:

$$[\![L_{\mathtt{low}}[id, \varepsilon_{low}]\langle(\!|\mathtt{M}_{\mathtt{high}} \oplus \mathtt{Ctxt}|\!)\rangle]\!] \sqsubseteq_R$$
$$[\![L_{\mathtt{high}}[id, \varepsilon_{high}]\langle(\!|\mathtt{Ctxt}|\!)\rangle]\!]$$

The definition implies "Each state transition made by $L_{\mathtt{high}}[id, \varepsilon_{high}]$ based on any context program $\mathtt{Ctxt}$ corresponds to (with the relation R) a sequence of state transitions by $L_{\mathtt{low}}[id, \varepsilon_{low}]$, which has the context of the compilation result from both $\mathtt{M}_{\mathtt{high}}$ and $\mathtt{Ctxt}$ together."

Limitations. CCAL has some limitations. First, it restricts dynamic allocations in the data structures used by verified modules. While this limitation reduces complexity in software verification, it also limits its expressiveness. However, it simplifies the challenges of building ThreadAbs and provides a practical way to create a certified layer for large-scale programs. Secondly, CCAL relies on the simulation library of CompCertX, but this restricts the expressiveness of CCAL layers. Using the library enforces us to use a fixed participant ID for all layers, which poses a problem during layer construction. This limitation becomes evident when considering thread-local (single thread-focused) certified interfaces, which are crucial for system software like operating systems.

Operating systems like CertiKOS initially share kernel modules among all CPU threads, including memory and thread management services. During this setup, CCAL uses a single participant (CPU) for its layers, with concurrency abstractions to model interactions with other CPUs. However, when introducing threads and a software scheduler, isolating each thread becomes crucial. Without this isolation, verifying one thread would require considering all the details and implementations of threads on the same CPU (associated with the same participant ID of the layer). To address this, CCAL introduces thread abstraction for certain system software. This task is complex because it involves dividing a single shared state among all threads into multiple states (*e.g.*, splitting shared registers based on CPU or thread IDs). Additionally, the proof relies on intricate details of CompCertX machine models (LAsm), and the abstraction must handle initial states of all threads, even those dynamically determined based on parent executions and user parameters, while maintaining thread isolation. In conclusion, CCAL and its machine model, LAsm, do not adequately address these challenges, highlighting the need for efficient methods and tools to handle these complexities effectively.

2.2. Thread abstraction in a Nutshell

To overcome CCAL's limitations, we integrate it with ThreadAbs, a framework that allows us to adjust the granularity of participant IDs for each layer as needed. This involves abstracting states and multithreading-related primitives, including scheduling. We demonstrate its utility with CertiKOS to showcase its enhancement of verified software developed using CCAL. Fig. 2 outlines the process of constructing a certified thread-local layer interface using both CCAL and ThreadAbs, with a focus on establishing contextual refinement between $\mathtt{M}_{API}$ on $L_{\mathtt{SHIM}}[cid, \varepsilon_{SHIM}]$ and the top-level layer $L_{\mathtt{API}}[tid, \varepsilon_{API}]$. This figure also provides a detailed perspective akin to Fig. 1. While CCAL forms the foundation for most of the layer construction, it becomes intricate when introducing threads. To address this, we introduce two layers, CSched and TSched, each parameterized by different participant IDs: a CPU ID for one and a thread ID for the other. ThreadAbs plays a pivotal role in refining the interaction between these layers, abstracting the communication among threads. We call this essential abstraction within ThreadAbs *view change*.

To enable CCAL and ThreadAbs to construct a certified thread-local layer interface with the proper view change, several steps are involved. Initially, it entails the creation of a certified CPU-local layer interface, utilizing $L_{\mathtt{SHIM}}[cid, \varepsilon_{SHIM}]$. These layers typically encompass modules essential for building a software scheduler, including components like spinlocks and memory management modules.

Once the software scheduler is introduced and abstracted into a set of primitives (*e.g.*, thread yield, sleep, wake-up, and create), ThreadAbs can be applied to initiate a "view change." In the diagram, CSched initiates the view change, while TSched completes it. Their primary tasks include: (1) building a machine model capable of reusing the CompCertX compiler while allowing for dynamic thread initial states (HAsm), (2) defining concrete CPU-local and thread-local layers that can be formally linked with the appropriate environmental contexts, represented by $\varepsilon_{cid}^{\mathtt{CSched}}$ and $\varepsilon_{tid}^{\mathtt{TSched}}$, respectively, and (3) demonstrating contextual refinement proofs between layers running on their respective machine models. These tasks are intricate and involve software-specific challenges. To address these complexities, ThreadAbs employs abstractions and modular solutions to enhance maintainability, reusability, and generality.

First, ThreadAbs introduces a thread configuration that defines the abstract environment in which the thread-local interface of the target software operates. This configuration includes details like the maximum number of threads, encompassing both active and available threads, the CPU's main thread (which serves as the parent of all CPU threads), and similar parameters. This approach allows ThreadAbs to abstract away from specific hardware or software configurations, making our proofs more adaptable and independent of such specifics.

Second, ThreadAbs introduces an intermediate machine model (MTAsm) and a layer definition (TLink) that can connect with both CSched on LAsm and TSched on HAsm without excessive complexity. The machine model performs two key functions: (1) it decomposes the single state definition and transition rules in LAsm into multiple thread states and transitions for each thread, and (2) it abstracts the behaviors of other threads, creating an environmental context for one thread to construct a thread-local layer interface, ensuring appropriate dynamic initial states for each thread. The intermediate layer, TLink, includes a set of primitives designed for use in MTAsm, while maintaining compatibility with both CSched on LAsm and TSched on HAsm. Although this layered approach within ThreadAbs may expand the size of proofs, it effectively breaks down complex problems into more manageable components. This approach enhances maintainability and reusability by solving these components separately.

Third, we have developed a contextual refinement template to illustrate the refinement process between multiple layers and their machine models within ThreadAbs. This template serves two important purposes: (1) it addresses the challenge of increased proof size in ThreadAbs due to the multiple steps involved in the process, and (2) it allows us to employ ThreadAbs and portions of its proofs across various target software. Initially, we demonstrate contextual refinements using a minimal invariant that each step (each machine model and layer depicted in ThreadAbs) must rely on. This is done without considering

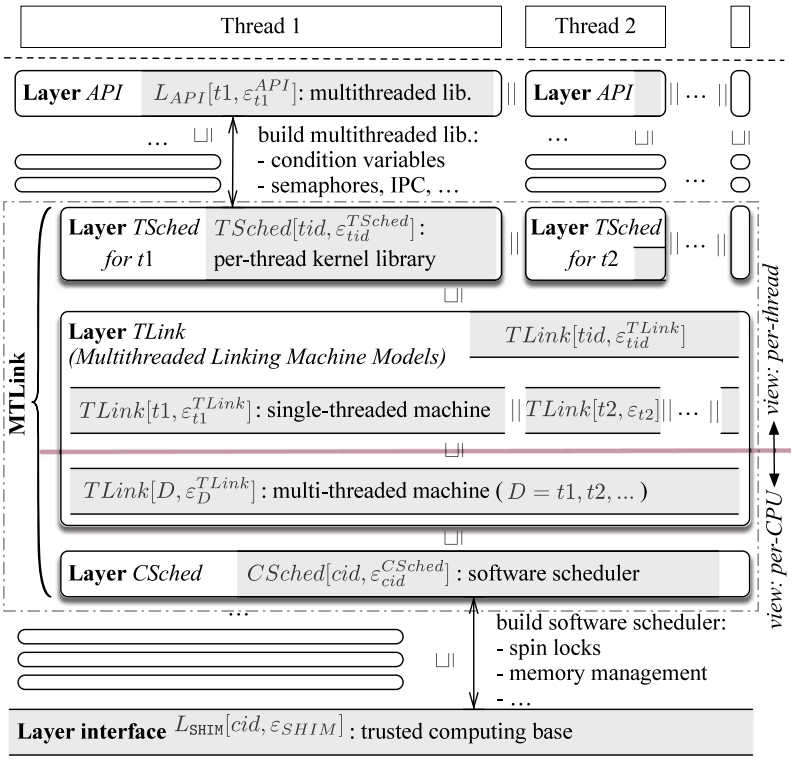

Fig. 2. ThreadAbs structure.

program-specific definitions, such as a list of primitives dependent on the program and memory accessors defining how a thread accesses memory. Subsequently, we fill in the abstracted parts of the proof template with program-specific definitions. However, it is important to note that this approach relies on specific requirements: (1) ThreadAbs assumes the existence of two scheduling primitives–yield and sleep–that transfer control to another thread within the same CPU, (2) thread creation must provide a dynamic initial state for its children, determined during runtime, and (3) threads within the target software must not engage in dynamic memory allocation, adhering to CCAL's restrictions. While this method still necessitates multiple proofs to tailor these proof templates to specific software and machine models, it renders several critical proofs in ThreadAbs generic and reusable. This significantly enhances its practicality and utility.

In summary, we present ThreadAbs, a thread abstraction framework that offers the properties outlined in Definition 2. The framework is applicable to a range of software when the software meets a set of reasonable assumptions and requirements outlined in later sections. In the subsequent sections, we delve into each component depicted in Fig. 2, providing comprehensive insights into how we structure this thread abstraction and how users can leverage the framework for their specific target software.

Definition 2 (*Thread Abstraction*)**.** For any CPU ID (id), thread ID (tid), context program (P), oracles for two layers ($\varepsilon_{cid}^{\texttt{CSched}}$ and $\varepsilon_{tid}^{\texttt{TSched}}$), the following is true:

$$[\![\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]\langle(\!|\texttt{P}|\!)\rangle]\!]_{\text{LAsm}} \sqsubseteq_{R_{link}} [\![\texttt{TSched}[tid, \varepsilon_{tid}^{\texttt{TSched}}]\langle(\!|\texttt{P}|\!)\rangle]\!]_{\text{HAsm}}$$

In this definition, $\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$ is the top layer interface of CPU modules with a machine model for a single CPU, LAsm, and $\texttt{TSched}[tid, \varepsilon_{tid}^{\texttt{TSched}}]$ is the bottom layer interface of thread modules with a machine model for a single thread, HAsm.

3. Bottom level interface and machine model: CSched with LAsm

The $\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$ is a CPU-specific layer interface in ThreadAbs designed to work within the LAsm, which is the machine model used in CompCertX. This layer is tailored to a single CPU and treats all other CPUs in the system as part of the environment. As a result, it provides an isolated and simplified state and evaluation rules for a single CPU in a multicore environment, leveraging CCAL's concurrency abstractions. However, in cases where multiple threads execute on the same CPU, they still share various aspects of their states, including registers, memory, and abstract states. Additionally, this layer incorporates a software scheduler, with scheduler-related primitives that are vital for managing active threads on the CPU. While the specific implementations of these primitives may vary across different operating systems, they share a common purpose in facilitating context switching during the evaluation process.

To accommodate various implementations across different types of operating systems, ThreadAbs adopts a flexible approach by incorporating a placeholder within the layer. This placeholder can be customized with diverse abstract states and primitives. As outlined in Section 2, this strategy is designed to break down the complexities of thread abstraction into smaller, more manageable components. This, in turn, simplifies the development process and promotes the reuse of crucial proofs within ThreadAbs for multiple target software systems. Subsequently, users have the liberty to introduce their own implementations to concretize this layer according to their specific needs. For instance, users can integrate hypervisor management modules, encompassing states and primitives, to instantiate CSched with custom definitions tailored to their target software. In Section 6, we illustrate how CSched interfaces with the concrete layer definition in CCAL.

However, creating a versatile framework for the verification of multiple targets necessitates making several generic assumptions that can encompass the diverse targets we aim to address. One crucial assumption within ThreadAbs concerning CSched revolves around the presence of software scheduler primitives. This assumption is fundamental since effectively employing the concept of threads is impossible

$$\frac{\begin{array}{c}\textsc{Yield Rule}\\ lst = (\rho, m, adt) \quad lst' = (\rho', m', adt') \quad m = m' \quad get\_curid(l) = cid \quad l' = l +\!\!+ (\varepsilon_{cid}^{\texttt{CSched}}\ cid\ l) \bullet (cid\ YIELD)\\ get\_curid(l') = cid' \quad \rho' = adt.kctxt[curid'] \quad adt' = adt/[adt.kctxt := adt.kctxt/[cid = \rho]]\end{array}}{\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}](\texttt{yield}) \vdash_{\text{LAsm}} \sigma_{\texttt{yield}}(\{\}, lst, l) \Rightarrow (\{\}, lst', l')}$$

$$\frac{\begin{array}{c}\textsc{Initial State}\\ m_{init} = \texttt{initmem}(P)\\ adt_{init} = \texttt{initabs}(P) \quad PC_{init} = \texttt{symbol}(\texttt{globalenv}(P), \texttt{main}) \quad \rho_{init} := \{\overline{\texttt{VUndef}}\}/[\textbf{PC} := PC_{init}, \textbf{ESP} := 0]\end{array}}{\text{initST}_{\text{LAsm}}(P) = (\rho_{init}, m_{init}, adt_{init}, nil)}$$

Fig. 3. Yield Rule and initial State in CSched with CPU-local Layer Interfaces.

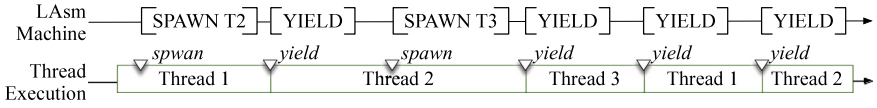

Fig. 4. Evaluation on CSched with LAsm.

in the absence of a software scheduler. ThreadAbs adopts the assumption that CSched incorporates two fundamental scheduling primitives: *yield* and *sleep*. Formally, this assumption is expressed as: "$(yield_{\text{func}}, \_) \in \texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$ and $(sleep_{\text{func}}, \_) \in \texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$." Both of these primitives serve the purpose of determining the next thread to be scheduled, initiating a context switch, and updating the identifier of the currently running thread within the system.

Fig. 3 illustrates the yield specification in the CSched layer. This specification relates input and output states, focusing on a single CPU (cid), while considering the environmental context ($\varepsilon_{cid}^{\texttt{CSched}}$) to account for other CPUs in the system. The state definition comprises two elements: the local CPU state (lst) and a global state (l). The local state (lst) includes CPU-specific data like register values, local memory, and an abstract datum. The yield specification updates the shared state by querying the environmental context ($\varepsilon_{cid}^{\texttt{CSched}}\ l$) to adjust scheduling information and appends a yield event. Furthermore, it updates the local CPU state. This transition involves preserving old register values by adjusting the abstract datum ($adt' = adt/[adt.kctxt := adt.kctxt/[cid = \rho]]$). This action stores the register values (ρ) of CPU cid in the kernel context pool ($adt.kctxt$) of its local abstract state (adt). Subsequently, CPU cid' is scheduled, and its context is reinstated in the registers for execution. This process effectively changes the currently executing thread on the CPU while appropriately preserving and restoring their execution contexts. However, this alone does not offer thread-local interfaces that exclusively focus on a single thread's execution without exposing extraneous details of other threads sharing the CPU.

Fig. 4 shows an example with three threads on LAsm, managed by $\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$. Initially, thread 1 is the main CPU thread, which spawns thread 2, and then thread 2 spawns thread 3. Despite having multiple threads, the machine's evaluation does not fully isolate each thread. It primarily changes the machine's state, which includes shared elements like registers. In this machine, even though multiple threads exist, the scheduling primitives' evaluation does not isolate one thread from others. Instead, it modifies the machine's state. The evaluation starts from a common initial state shared by all threads. Fig. 3 explains how the global initial state is set, involving memory, abstract state, and initial register values. It utilizes the function pointer of the initial thread's main function in the CPU. Later sections will delve into achieving thread abstraction at this level without altering semantics.

Note that transition rules for other primitives that will be later filled out for the placeholders in $\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]$ are also relations on two states: from an initial state to a result state of each transition, as follows:

$$\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}](fid) \vdash_{\text{LAsm}} \sigma_{id}(args, lst, l) \Rightarrow (res \cup \{\}, lst', l').$$

Here, σ_{id} denotes the specification of primitive fid, $args$ is the list of arguments for the primitive call, and $res \cup \{\}$ defines the return type of the primitive call. Figs. 3 and 4 do not include examples for those primitives since they are not the main focus of our work.

4. Thread abstraction

ThreadAbs strives to create an isolated abstract machine for individual threads while accounting for their interleaved interactions with others. These interactions are visible to all threads as shared transition rules and states within CSched on LAsm. We break the problem into smaller components and address them separately. It follows the method to design a state transition-based machine model in the layered manner with CCAL, allowing customization by specifying a participant, environmental context, and layer. This section provides an in-depth explanation of these machine models and their parameterized definitions.

4.1. *Multithreaded machine model:* MTAsm

As the first step, ThreadAbs introduces MTAsm, a multithreaded machine model, and the associated layer TLink. The primary purpose of it is to divide the single state and sequential evaluation for all CPU threads into multiple thread-local states and pseudo-concurrent evaluations. It ensures thread isolation, and each maintains its registers. Fig. 5 illustrates thread evaluation on MTAsm in comparison to that on LAsm. The top three timelines depict the isolated evaluation of three threads on MTAsm, while the bottom timeline shows thread evaluation sharing states on LAsm. In the top three timelines, the only way that each thread explicitly interacts with others is via scheduling primitives. The scheduling transition (yield in the figure) changes the executing thread's ID and logs the event. Introducing MTAsm also brings the challenge of where to define scheduling primitives, which affect multiple threads. Instead of placing them in TLink, MTAsm includes rules for two scheduling primitives within the machine model. It allows ThreadAbs to provide a multi-threaded interface while maintaining isolation between each thread's evaluation and effectively managing scheduling primitives.

However, simply splitting the CPU state and introducing a new layer definition is insufficient for MTAsm to handle dynamic thread initialization with dynamic resource allocation by each thread's parent. Each thread in the model cannot determine the initial local state information, such as the maximum resource capacity and function pointer of each thread when it is spawned, since its initialized state is decided by its parent thread, which dynamically allocates the child's resources and function pointer. However, if each thread is isolated from others, parents cannot directly initialize the states of their children when they are spawned, as even a parent thread cannot directly update the state of its child. To address it, MTAsm implements a lazy initialization

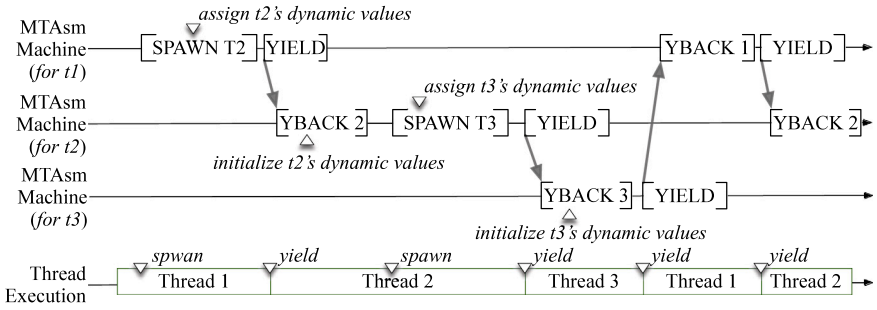

Fig. 5. Evaluation on TLink with MTAsm.

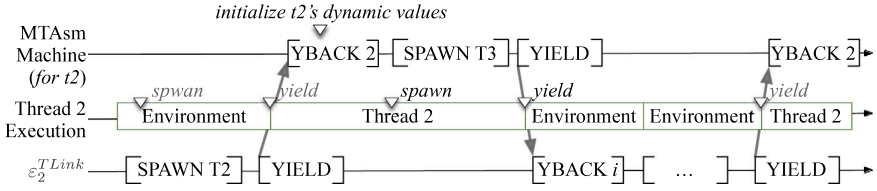

Fig. 6. Single Thread Evaluation on TLink with MTAsm.

semantics that defers updating a thread's initial state until it first reaches its evaluation control, even after it has been spawned.

Fig. 5 illustrates this process. In the figure, Thread 2 begins its initial evaluation after Thread 1 spawns it and passes control to Thread 2. During the spawn, Thread 1 initializes Thread 2's private-state information, such as user context and kernel context, based on dynamically determined information (*e.g.*, arguments of the thread spawn function). Instead of direct modification, the child thread (Thread 2 in this case) initializes itself when it first starts running with the initial log. This log contains a sequence of a spawn event and a yield event generated by Thread 1 in the figure. Through these steps, MTAsm provides a multithreaded interface that isolates each thread's evaluation and handles dynamic thread initial states.

The discussed interface addresses several thread abstraction challenges but remains incomplete. It still exposes concrete evaluations of entire threads in the system. The next step in achieving proper thread abstraction is to isolate the focused thread's evaluation from others. To achieve this, MTAsm utilizes a parameterized environmental configuration (Fig. 6). Unlike a fixed thread set configuration and environmental context in CSched on LAsm, MTAsm can adapt to different sets of threads within the same CPU. TLink handles the environmental context for abstracted threads. For example, in Fig. 6, we see an evaluation of MTAsm with TLink parameterized by a singleton *thread 2* set and an environmental context that abstracts threads 1 and 3. Thread 2 initializes its state using a spawn event and transitions from thread 1, its parent. Then, it continues the evaluation, consulting the environmental context for scheduling primitives. This approach allows ThreadAbs to provide thread abstraction and create a thread-local layer interface. However, it is essential for us to ensure this environmental context accurately represents the underlying evaluation. The following sections provide formal definitions and properties to demonstrate this rigorously.

4.1.1. Multithreaded environment configuration

Fig. 7 outlines essential variables, data types, and functions in ThreadAbs. Within ThreadAbs, there are two states: $\mathcal{P}$ for individual thread states and $\mathcal{S}$ for shared state across all threads, captured as a shared log $\mathcal{L}$. This log records all shared object operations. Four events are introduced in $\mathcal{L}$: YIELD and SLEEP corresponding to `yield` and `sleep`, and YBACK, a logical primitive. PRIM represents all primitive calls accessing shared objects by any thread, whether on the same or different CPUs. This event logs argument information ($args$) and a state snapshot ($snap$) when invoked. Examples in CertiKOS include thread creation and page allocation primitives.

Additionally, ThreadAbs defines thread configuration variables, including D_{thrd} for all CPU threads and an tid_{m} on the same CPU. The initial thread, tid_{m}, initializes the system state and does not need distinguishing from others. ThreadAbs assumes suitable initial values for private and shared states ($\mathcal{P}_{I(m)}$, $\mathcal{P}_{I(nm)}$, and $\mathcal{S}_I$), plus an initial log for each thread ($\mathcal{L}_I$). Private states for the initial thread and others differ to represent dynamic initial state allocation. The initial thread's state is static, determined at system boot, while others' private states dynamically adjust during evaluation based on the shared log. For example, the initial log for thread 1 in Fig. 6 is empty, whereas thread 2's initial log contains events from thread 1, including the creation of thread 2, yield, and yield back events that shift control from thread 1 to thread 2.

Fig. 7 also introduces important auxiliary functions within ThreadAbs. The update function calculates the current shared state based on the log, where all inter-thread operations are recorded as events. Functions like IPD_{thrd} and IPC_{thrd} determine and assign private state and register values, facilitating dynamic thread spawning. STCheck ensures thread validity during evaluations, especially when evaluating `yield`. Additionally, ThreadAbs offers a function (EnvQ) for querying an environmental context based on the log. Formal rules in later sections demonstrate how these functions are utilized in various machine models within ThreadAbs. ThreadAbs also incorporates essential properties pertaining to these definitions, variables, and auxiliary functions, which are crucial for formal thread abstraction proofs. For instance, it enforces constraints on a thread set (*e.g.*, $tid_{\text{m}} \in D_{\text{thrd}}$) and an initial log (*e.g.*, $\mathcal{L}_I(pid)$ must always contain a corresponding thread spawn event for pid). These properties serve as the foundation for defining intermediate machine models and introducing a machine model for thread-local interfaces, as discussed later.

4.1.2. Syntax and semantics

This section presents an intermediate machine model, MTAsm, parameterized by an intermediate layer definition, TLink. Formally, the MTAsm state is defined as a tuple as follows:

$$st_{\text{MTAsm}} := \\ ((cur\_tid : \mathbb{Z}), \{(tid : \mathbb{Z}) \rightharpoonup (lst_{thrd} : \mathcal{P})\}, (l_{thrd} : \mathcal{L}))$$

of which the first element is the currently running thread identifier, $(\{tid \rightharpoonup lst_{thrd}\}, l_{thrd})$, which corresponds to (lst, l) in LAsm, are a set

Event, Thread Configurations, and Data Types

$$\mathsf{ev}_{\mathsf{thrd}} := \mathsf{YIELD}(tid : \mathbb{Z}) \mid \mathsf{SLEEP}\ (tid : \mathbb{Z})\ (i : \mathbb{Z}) \mid \mathsf{YBACK}\ (tid : \mathbb{Z}) \mid \mathsf{PRIM}\ (tid : \mathbb{Z})\ (id : \mathsf{ID}_{\mathsf{prim}})\ (args : \mathrm{lst}\ val)(snap : \mathsf{T}_{\mathsf{TYPE}}) \quad \text{(event)}$$

$\mathcal{P}$	$\in$	$\mathsf{T}_{\mathsf{TYPE}}$	(private state)	$\mathcal{S}$	$\in$	$\mathsf{T}_{\mathsf{TYPE}}$	(shared state)
$\mathcal{L}$	$\in$	lst $\mathsf{ev}_{\mathsf{thrd}}$	(log)	**FAIR**	$\in$	$\mathbb{N}$	(fairness assumption)
D_{thrd}	$\in$	$\{\mathbb{Z}\}$	(thread set of the CPU)	tid_{m}	$\in$	$\mathbb{Z}$	(init thread of the system)
$\mathcal{P}_{I(m)}$	$\in$	$\mathcal{P}$	(initial private state-init)	$\mathcal{P}_{I(nm)}$	$\in$	$\mathcal{P}$	(initial private state-noninit)
$\mathcal{S}_I$	$\in$	$\mathcal{S}$	(initial shared state)	$\mathcal{L}_I(pid : \mathbb{Z})$	$\in$	$\mathcal{L}$	(initial log for pid)

Auxiliary Functions

update	$\in$	$\mathcal{S} \to \mathcal{L} \to \mathcal{S}$	(update function)
get_tid	$\in$	$\mathcal{S} \to \mathbb{Z}$	(get current thread ID)
$\mathsf{IPD}_{\mathsf{thrd}}$	$\in$	$\mathbb{Z} \to \mathcal{L} \to \mathcal{P}$	(initial private data calcuation function)
$\mathsf{IPC}_{\mathsf{thrd}}$	$\in$	$\mathrm{Genv} \to \mathbb{Z} \to \mathcal{L} \to val$	(initial register value calcuation function)
STCheck	$\in$	$\mathsf{ID}_{\mathsf{prim}} \to \mathcal{L} \to \mathcal{P} \to \mathsf{P}$	(check local state)
EnvQ	$\in$	$\mathcal{L} \to \mathsf{ev}_{\mathsf{thrd}}$	(environmental context query function)

Fig. 7. Basic Definitions of Thread Abstraction Framework.

of thread private states and a log shared by all threads. The layer declaration, TLink, is a template layer definition that primitives are defined with abstract signatures with placeholders, and users can fill out placeholders later. MTAsm definitions and proofs do not take care about the details of the layer, but the minimum requirement is necessary to provide the formal template and linking proofs (in Definition 3).

Definition 3 (*Primitive Domain in* TLink *and* CSched)**.** For any primitive ID (fid) which is neither yield nor sleep primitives ($fid \neq \mathtt{yield} \wedge fid \neq \mathtt{sleep}$), the following should hold:

$$(fid, \_) \in \mathtt{CSched}[\_, \_] \leftrightarrow (fid, \_) \in \mathtt{TLink}[\_, \_].$$

Definition 3 specifies that the TLink layer includes a set of primitives similar to the CSched layer (except for two scheduling primitives). However, their specifications are defined on thread-local states rather than CPU-local states. Specifically, the following expression represents the specification:

$$\mathtt{TLink}[\_, \_](fid) \vdash \sigma_{id_{\mathrm{p}}}(args, lst_{tid}, l) \Rightarrow (res \cup \{\}, lst'_{tid}, l')$$

The specification gets the argument ($args$), a local state of the current thread tid (lst_{tid}), and a global state (l), and then returns the result of the evaluation ($res \cup \{\}$) as well as the updated state ((lst'_{tid}, l')).

On the other hand, two scheduling primitives that are not defined in TLink, but in CSched, and their transitions are treated in a different way. MTAsm directly models two scheduling primitives as depicted in Fig. 8 includes the essential definitions to describe their semantics. The first definition concerns a thread's state. In this machine model, which takes an arbitrary thread set on the same CPU as a parameter, thread states on the same CPU are divided into three categories: "running" (RUN), "available" (AVAIL), and "environment" (ENV). The "running" state denotes a thread actively executing and awaiting initialization. The "available" state applies to a thread that may be spawned or scheduled in the future but has not yet started execution. For instance, an already spawned thread can have a AVAIL state before its initial evaluation. The last category includes threads considered part of the environment for the currently specified thread set, such as threads 1 and 3 in the evaluation shown in Fig. 6.

With this definition, the complete state description of MTAsm consists of a tuple of elements: a thread ID that is currently running, a collection of thread states, a set of private states, memory, and a log. The set of private states serves as a partial map from a thread ID to its isolated private state, and all threads that are running and/or available in a set of thread states have their own private state in the set of private states. However, all threads that are indicated as environmental threads in a set of thread states do not have their corresponding private states in a set of private states. A memory and a log are shared by all threads, but we assume that the region accessed by each thread is mutually excluded from other regions owned by other threads. This is a limitation and restriction of ThreadAbs, but it is necessary to simplify our ThreadAbs with the capability of handling our case study, CertiKOS. We plan to extend it to handle more general cases than the current version as part of our future research direction. With these definitions, the figure presents three evaluation rules of MTAsm: the *yield*, *environment*, and *yield back* rules, assuming that the machine is parameterized with a TLink layer and a thread set A (a subset of D_{thrd}). The primary rule of a yield transition rule in MTAsm is to change the currently running thread identifier of the machine state (from $curid$ to $curid'$ in the rule). Specifically, the evaluation first calculates the currently running thread by replying the shared log ($curid = \mathsf{get\_tid}(\mathsf{update}(S_I, \mathrm{log}))$), adds the yield event into the shared log, and then recalculates the next scheduled thread by replying the updated log ($curid' = \mathsf{get\_tid}(\mathsf{update}(S_I, \mathrm{log}'))$).

Before triggering the yield transition, the rule verifies several essential state invariants. Firstly, it assumes that the most recent machine evaluation was neither a yield nor a sleep transition rule. This design choice is a result of how MTAsm handles transitions. Instead of immediately changing the currently running thread and commencing the evaluation with the newly scheduled thread in a single combined yield and sleep transition, MTAsm takes a stepwise approach. Initially, it allows the current thread to relinquish control by merely altering the thread ID responsible for the next scheduled thread. Subsequently, the scheduled thread executes one of the two rules outlined in the figure: the *environment rule* or the *yield back rule*, depending on whether the scheduled thread is part of A. This design serves two primary purposes: (1) it enables the separate handling of dynamically initialized states for each thread during their initial scheduling, a task managed by the *yield back rule*, and (2) it clearly distinguishes the case where the scheduled thread is not in A, addressed by the *environment rule*, from the case where the thread is in A. Additionally, the transition rule checks various properties to ensure safety and alignment with the low-level machine details. For instance, it verifies whether the current thread ($curid$) is a valid running thread and whether the current program counter (PC) correctly indicates the yield function.

The environment rule specifies a transition for when the current thread is not in the current thread set ($tsp[curid] = \mathtt{ENV}$). In this case, evaluations are conducted solely based on a global log (log), employing an environmental context query (EnvQ). The yield back evaluation rule is used to alter the thread state (TS) based on the current thread's status ($curid$). This rule comprises two cases: (1) when the thread is

State

$$\begin{array}{lll} \mathtt{TS} & := & \mathtt{RUN}\ (regset : \rho) \mid \mathtt{AVAIL} \mid \mathtt{ENV} \qquad \text{(thread state)} \\ \mathtt{ST}_{\mathrm{MTAsm}} & \in & (\mathbb{Z}, \{\mathtt{TS}\}, \{\mathcal{P}\}, mem, \mathcal{L}) \qquad \text{(state definition of MTAsm)} \end{array}$$

Evaluation Rules and Yield State

Yield Rule

$$\frac{\begin{array}{c} st = (curid, tsp, psp, mem, log) \quad curid = \mathrm{get\_tid}(\mathrm{update}(S_I, \log)) \quad log \neq \_ \cdot (\mathtt{YIELD}\ \_) \\ log \neq \_ \cdot (\mathtt{SLEEP}\ \_\ \_) \quad tsp[curid] = \mathtt{RUN}\ \rho \quad \mathbf{PC}\ \rho = \mathtt{symbol}(\mathtt{globalenv}(P), \mathtt{yield}) \quad psp[curid] = ps \\ \mathtt{STCheck}(\mathtt{yield}, log, ps) \quad log' = log \cdot (\mathtt{YIELD}\ curid) \quad curid' = \mathrm{get\_tid}(\mathrm{update}(S_I, log')) \quad st' = (curid', tsp, psp, mem, log') \end{array}}{\mathtt{TLink}[A, \varepsilon_A^{\mathtt{TLink}}] \vdash_{\mathrm{MTAsm(yield)}} st \Rightarrow st'}$$

Environment Rule

$$\frac{\begin{array}{c} st = (curid, tsp, psp, mem, log) \quad curid = \mathrm{get\_tid}(\mathrm{update}(S_I, log)) \quad tsp[curid] = \mathtt{ENV} \quad psp[curid] \notin psp \\ ev = \mathrm{EnvQ}(log) \quad log' = log \cdot ev \quad curid' = \mathrm{get\_tid}(\mathrm{update}(S_I, log')) \quad st' = (curid', tsp, psp, mem, log') \end{array}}{\mathtt{TLink}[A, \varepsilon_A^{\mathtt{TLink}}] \vdash_{\mathrm{MTAsm(env)}} st \Rightarrow st'}$$

Yield Back Rule

$$\frac{\begin{array}{c} st = (curid, tsp, psp, mem, log) \\ curid = \mathrm{get\_tid}(\mathrm{update}(S_I, log)) \quad log = \_ \cdot (\mathtt{YIELD}\ \_) \vee log = \_ \cdot (\mathtt{SLEEP}\ \_\ \_) \quad psp[curid] = ps \\ ((tsp[curid] = \mathtt{AVAIL} \wedge PC_{init} = \mathtt{IPC}_{\mathrm{thrd}}(\_, curid, log) \wedge \rho = \{\overline{\mathtt{VUndef}}\}/[\mathbf{RA} := PC_{init}, \mathbf{ESP} := 0]) \quad \vee tsp[curid] = \mathtt{RUN}\ \rho) \\ \rho' = \rho/[\mathbf{PC} := \mathbf{RA}\ \rho] \quad log' = log \cdot (\mathtt{YBACK}\ curid) \quad tsp' = tsp/[curid := \mathtt{RUN}\ \rho'] \quad st' = (curid', tsp', psp, mem, log') \end{array}}{\mathtt{TLink}[A, \varepsilon_A^{\mathtt{TLink}}] \vdash_{\mathrm{MTAsm(yback)}} st \Rightarrow st'}$$

Initial State

$$\frac{\begin{array}{c} m_{init} = \mathtt{initmem}(P) \quad PC_m = \mathtt{symbol}(\mathtt{globalenv}(P), \mathtt{main}) \\ \rho_m := \{\overline{\mathtt{VUndef}}\}/[\mathbf{PC} := PC_m, \mathbf{ESP} := 0] \quad psp_{init}[tid_{\mathrm{m}}] = \mathcal{P}_{I(m)} \quad tsp_{init}[tid_{\mathrm{m}}] = \mathtt{RUN}\ \rho_m \\ tid_{in} \neq tid_{\mathrm{m}} \quad tid_{in} \in A \quad psp_{init}[tid_{in}] = \mathtt{IPD}_{\mathrm{thrd}}(tid_{in}, \mathcal{L}_I(tid_{in})) \quad tsp_{init}[tid_{in}] = \mathtt{AVAIL} \\ tid_{env} \notin A \quad tsp_{init}[tid_{env}] = \mathtt{ENV} \quad psp_{init}[tid_{env}] \notin psp \end{array}}{\mathrm{initST}_{\mathrm{MTAsm}}(A, P) = (tid_{\mathrm{m}}, tsp_{init}, psp_{init}, mem_{init}, nil)}$$

Fig. 8. Parts of Evaluation Rules of MTAsm.

designated as a running thread (RUN ρ), it indicates that the thread has already been initialized and scheduled. Consequently, the rule only modifies the program counter value, and (2) if the thread is marked as AVAIL, it indicates that the thread has been spawned but has not been dispatched for evaluation yet. In this scenario, the appropriate thread state is calculated using $\mathtt{IPC}_{\mathrm{thrd}}$. These rules correspond to a "YBACK" evaluation in Fig. 5.

In addition to these rules, MTAsm also encompasses other rules for internal instructions like memory load and store, external primitive calls (defined in TLink), and a sleep function call. However, these rules closely resemble the evaluation rules in CompCertX [17] or the yield transition rule presented in the figure, so we have omitted them here. Fig. 8 also illustrates the initial state of MTAsm. The current thread identifier is the init thread (tid_{m}) of the system. The thread state pool (tsp_{init}) includes proper values for **PC** and **ESP** of the main thread but holds AVAIL or ENV depending on the thread identifier and whether the thread is in A or not. The rule also assigns a suitable initial private state to each thread using $\mathtt{IPD}_{\mathrm{thrd}}$. With these features, we accurately model a machine capable of running multithreaded applications using two software scheduling primitives (yield and sleep). It isolates each thread's private state from others, handles dynamic initial states for each thread appropriately, and can be parameterized by a subset of the available full thread set running on top of the CPU. The complete definition of MTAsm is accessible online [19].

4.1.3. Refinement proof

ThreadAbs establishes a formal connection between a context program running on $\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}]$ with LAsm and the same context program on $\mathtt{TLink}[D_{\mathrm{thrd}}, \varepsilon_{D_{\mathrm{thrd}}}^{\mathtt{TLink}}]$ with MTAsm. We employ a simulation library from CompCert [20] for this purpose. This connection relies on various details about states and primitive sets in both interfaces, including abstract definitions (in Fig. 7) and two abstract layer declarations. To establish this proof with minimal restrictions, ThreadAbs introduces an abstract relation AbsRelC, consisting of two rules in Fig. 9, which are dramatically simplified for the readability. First, the yield match relation ensures that the yield evaluation rule in MTAsm corresponds to yield in $\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}]$ based on the $\mathcal{M}_{data}(\mathrm{MTAsm}, \mathrm{LAsm})$ relation. Second, the primitive match relation states that all primitive call transitions in $\mathtt{TLink}[D_{\mathrm{thrd}}, \varepsilon_{D_{\mathrm{thrd}}}^{\mathtt{TLink}}]$ with MTAsm correspond to primitive call transitions in $\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}]$ with LAsm, defining a transitional forward simulation property. Using these abstract relations, ThreadAbs establishes a contextual refinement property, as presented in Lemma 1.

Lemma 1. *For any context program* P*, a CPU ID* cid*, a full thread set* D_{thrd} *on the CPU, two template layer interfaces (*CSched *and* TLink*), environmental contexts (*$\varepsilon_{cid}^{\mathtt{CSched}}$ *and* $\varepsilon_{tid}^{\mathtt{TLink}}$*), and a refinement relation* $R_{(\mathrm{MTAsm,LAsm})}$*, the following holds:*

$$[\![\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}]\langle \mathtt{P} \rangle]\!]_{\mathrm{LAsm}} \sqsubseteq_{R_{(\mathrm{MTAsm,LAsm})}} [\![\mathtt{TLink}[D_{\mathrm{thrd}}, \varepsilon_{D_{\mathrm{thrd}}}^{\mathtt{TLink}}]\langle \mathtt{P} \rangle]\!]_{\mathrm{MTAsm}}$$

With the assumption that two layers with two different machines ($\mathtt{TLink}[D_{\mathrm{thrd}}, \varepsilon_{D_{\mathrm{thrd}}}^{\mathtt{TLink}}] \vdash_{\mathrm{MTAsm}}$ *and* $\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}] \vdash_{\mathrm{LAsm}}$*) satisfies the relation* AbsRelC *and* $R_{(\mathrm{MTAsm,LAsm})}$ *includes* $\mathcal{M}_{data(\mathrm{MTAsm,LAsm})}$ *in it.*

However, users need to provide $\mathcal{M}_{data}(\mathrm{MTAsm}, \mathrm{LAsm})$ and evidence of AbsRelC based on concrete definitions of abstract layers ($\mathtt{CSched}[cid, \varepsilon_{cid}^{\mathtt{CSched}}]$ and $\mathtt{TLink}[D_{\mathrm{thrd}}, \varepsilon_{D_{\mathrm{thrd}}}^{\mathtt{TLink}}]$), which depend on the specific program being verified. While this task can be complex and time-consuming, ThreadAbs offers essential templates for thread abstraction and refinement proofs. These templates, combined with user-provided program-related definitions, enable ThreadAbs to automatically generate a contextual refinement lemma between two layer interfaces. In addition to Lemma 1, ThreadAbs also provides the proof that establishes a connection between a program running with a full thread set and the same program running with a specific thread and its environmental context. The proof is shown in Fig. 6. It is an integral part of

$$\textsc{Yield Match}\quad \frac{\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}] \vdash_{\text{MTAsm(yield)}} st_{\text{MT}} \Rightarrow st'_{\text{MT}} \qquad \mathcal{M}_{data(\text{MTAsm,LAsm})}(st_{\text{MT}}, st_{\text{L}}) \qquad (\exists\ st'_{\text{L}},\ \mathtt{CSched}[cid, \varepsilon^{\mathtt{CSched}}_{cid}](\mathtt{yield}) \vdash_{\text{LAsm}} \sigma_{\mathtt{yield}}(\{\}, st_{\text{L}}) \Rightarrow (\{\}, st'_{\text{L}}) \ \wedge\ \mathcal{M}_{data(\text{MTAsm,LAsm})}(st'_{\text{MT}}, st'_{\text{L}}))}{\mathtt{AbsRelC}(\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}] \vdash_{\text{MTAsm(yield)}}, \mathtt{CSched}[cid, \varepsilon^{\mathtt{CSched}}_{cid}](\mathtt{yield}))}$$

$$\textsc{Primitive Match}\quad \frac{\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}](id) \vdash_{\text{MTAsm}} \sigma_{id}(args, st_{\text{MT}}) \Rightarrow (res_t \cup \{\}, st'_{\text{MT}}) \qquad \mathcal{M}_{data(\text{MTAsm,LAsm})}(st_{\text{MT}}, st_{\text{L}}) \qquad (\exists st'_{\text{L}}\ res_c,\ \mathtt{CSched}[cid, \varepsilon^{\mathtt{CSched}}_{cid}](id) \vdash_{\text{LAsm}} \sigma_{id}(args, st_{\text{L}}) \Rightarrow (res_c \cup \{\}, st'_{\text{L}}) \ \wedge\ \mathcal{M}_{data(\text{MTAsm,LAsm})}(st'_{\text{MT}}, st'_{\text{L}}) \ \wedge\ res_t = res_c)}{\mathtt{AbsRelC}(\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}] \vdash_{\text{MTAsm}} (id), \mathtt{CSched}[cid, \varepsilon^{\mathtt{CSched}}_{cid}](id))}$$

Fig. 9. Abstract Relation (`AbsRelC`).

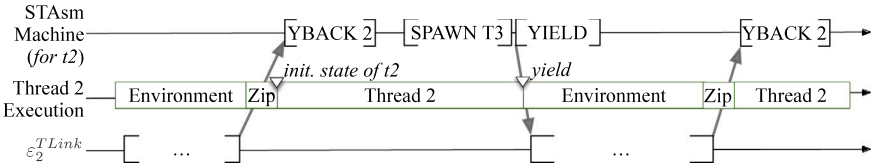

Fig. 10. Single Thread Evaluation on `TLink` with STAsm.

ThreadAbs's thread abstraction, allowing it to hide local evaluations of other threads in a higher layer interface ($\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]$) while maintaining a contextual refinement property with a lower layer interface ($\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}]$), as detailed in Lemma 2.

Lemma 2. *For any context program* P*, a full thread set* D_{thrd}*, a thread identifier* tid*, a template layer* TLink*, environmental contexts* ($\varepsilon^{\mathtt{TLink}}_{cid}$ *and* $\varepsilon^{\mathtt{TLink}}_{tid}$*), and a refinement relation* $R_{(\text{MTAsm,MTAsm})}$*, the following holds:*

$$[\![\mathtt{TLink}[D_{\text{thrd}}, \varepsilon^{\mathtt{TLink}}_{D_{\text{thrd}}}]\langle \mathtt{P}\rangle]\!]_{\text{MTAsm}} \sqsubseteq_{R_{(\text{MTAsm,MTAsm})}} [\![\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]\langle \mathtt{P}\rangle]\!]_{\text{MTAsm}}$$

The proof of Lemma 2 also includes various intermediate steps and definitions, but users do not need to delve into these details. ThreadAbs automatically ensures this property for any concrete layers and configurations meeting minimum requirements (Section 4.1.1). In summary, ThreadAbs, MTAsm, and Lemmas 1 and 2 provide a thread-local layer interface, isolating a thread from others. Definitions and proofs for these lemmas are available online [21] and [22], respectively.

4.2. Single-threaded machine model: Overview

MTAsm already offers a template layer interface focusing on a single thread and hiding others' behavior, as seen in Lemma 2 ($\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]$ with MTAsm). However, this is not sufficient for thread-local layer interfaces due to the incompatibility with CompCertX. CompCertX requires atomic evaluation of each primitive call, including `yield`. However, `yield` in MTAsm uses sequences of yield, environmental, and yield-back transition rules, which do not align with CompCertX. To bridge this gap, ThreadAbs introduces STAsm, an intermediate machine model with simplified scheduling transitions and adjusted state definitions, ensuring compatibility with CompCertX for compilation purposes.

STAsm simplifies scheduling transitions using zipping functions that consolidate events between two key moments: the initiation of scheduling primitive calls and when the callee regains control. Fig. 10 illustrates an example with STAsm based on this concept. The machine commences evaluation with a dynamically allocated initial state computed through IPD_{thrd} and IPC_{thrd} (in Fig. 7). It performs local transitions, generating events as necessary. When Thread 2 invokes `yield`, the machine employs a zipping function for environmental context queries until Thread 2 regains control. However, this relies on an assumption: each thread must undergo at least one evaluation, and the entire system generates a finite number of events (which may not necessarily correspond to the number of steps, as multiple private steps may not produce events).

4.2.1. Syntax and semantics

Fig. 11 provides a machine state definition, a zipping function, and parts of evaluation rules of STAsm. It employs a state definition with five components: thread identifier, register values, thread-local state, memory, and a log, focusing solely on one thread. The EnvZip function combines multiple environmental context queries until the result matches the thread identifier, with a bounded query count (n), which can be customized for typical multithreaded program executions. In STAsm, a `yield` call is abstracted into a single big-step transition. This transition consists of three simultaneous actions: (1) updating the log by appending YIELD, (2) iteratively querying the environmental context until control is returned to the thread, and (3) adding a yield-back event to the log. This abstraction condenses several steps from MTAsm (yield, environment, yield-back rules) into one. The initial state of STAsm is tailored for single-threaded program execution. It contains dynamically allocated values for the program counter and private state, ensuring atomic representation of a yield transition initiated by a specific thread, under reasonable assumptions. Detailed Coq definitions for STAsm are accessible online [23].

4.2.2. Refinement proof

ThreadAbs offers Lemma 3, with the full Coq definition available online [24]. This lemma formally links program evaluations on STAsm and MTAsm. While it relies on thread configurations (*e.g.*, thread ID) and a template layer definition ($\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]$), it does not require additional abstract properties like `AbsRelC`. As a result, users can utilize it without introducing extra concrete definitions, as both MTAsm and STAsm share the same template layer definition, TLink.

Lemma 3. *For any context program* P*, a thread* tid*, a template layer* TLink*, an environmental context* $\varepsilon^{\mathtt{TLink}}_{tid}$*, and a refinement relation* $R_{(\text{STAsm,MTAsm})}$*, the following holds:*

$$[\![\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]\langle \mathtt{P}\rangle]\!]_{\text{MTAsm}} \sqsubseteq_{R_{(\text{STAsm,MTAsm})}} [\![\mathtt{TLink}[tid, \varepsilon^{\mathtt{TLink}}_{tid}]\langle \mathtt{P}\rangle]\!]_{\text{STAsm}}$$

5. Top-level interface

5.1. TSched *With* HAsm

As the final step in achieving thread abstraction and delivering a thread-local layer interface, ThreadAbs introduces HAsm, a machine model tailored for thread-local layer interfaces. This addition is necessary to bridge the gap between the machine model and transition rules in STAsm and the form required to connect with CompCertX.

State and Auxiliary (Zipping) Function

$$\mathsf{ST}_{\mathrm{STAsm}} \quad \in \quad (\mathbb{Z}, \rho, \mathcal{P}, mem, \mathcal{L}) \quad \text{(full state definition of STAsm)}$$

$$\mathrm{EnvZip}(n : \mathbb{N})(tid : \mathbb{Z})(l : \mathcal{L}) = \begin{cases} \mathsf{None} & (\text{when } n = O) \\ \mathsf{Some}\ l & (\text{when } n \neq O \wedge tid = \mathsf{get\_tid}(\mathsf{update}(S_I, l))) \\ \mathrm{EnvZip}(n',\ tid,\ (l \bullet \mathrm{EnvQ}(log))) & (\text{when } n = S\ n' \wedge tid \neq \mathsf{get\_tid}(\mathsf{update}(S_I, l))) \end{cases}$$

Evaluation Rules and Yield State

YIELD RULE

$$\frac{\begin{array}{c} st = (tid, \rho, ps, mem, log) \qquad tid = \mathsf{get\_tid}(\mathsf{update}(S_I, log)) \qquad log \neq \_ \bullet (\mathsf{YIELD}\ \_) \\ log \neq \_ \bullet (\mathsf{SLEEP}\ \_\ \_) \qquad \mathbf{PC}\ \rho = \mathtt{symbol}(\mathtt{globalenv}(P), \mathtt{yield}) \qquad \mathrm{STCheck}(\mathtt{yield}, log, ps) \qquad log' = log \bullet (\mathsf{YIELD}\ tid) \\ log'' = \mathrm{EnvZip}(\mathbf{FAIR}, tid, log') \qquad log''' = log'' \bullet (\mathsf{YBACK}\ tid) \qquad \rho' = \rho/[\mathbf{PC} := \mathbf{RA}\ \rho] \qquad st' = (tid, \rho', ps, mem, log''') \end{array}}{\mathtt{TLink}[tid, \varepsilon_{tid}^{\mathtt{TLink}}] \vdash_{\mathrm{STAsm(yield)}} st \Rightarrow st'}$$

INITIAL STATE

$$\frac{\begin{array}{c} mem_{init} = \mathtt{initmem}(P) \\ PC_{init} = \mathrm{IPC}_{\mathsf{thrd}}(\mathtt{globalenv}(P), tid, \mathcal{L}_I(tid)) \qquad \rho_{init} = \{\overline{\mathtt{VUndef}}\}/[\mathbf{PC} := PC_{init}, \mathbf{ESP} := 0]) \qquad ps_{init} = \mathrm{IPD}_{\mathsf{thrd}}(tid, \mathcal{L}_I(tid)) \end{array}}{\mathrm{initST}_{\mathrm{STAsm}}(tid, P) = (tid, \rho_{init}, ps_{init}, mem_{init}, \mathcal{L}_I(tid))}$$

Fig. 11. Parts of Evaluation Rules of STAsm.

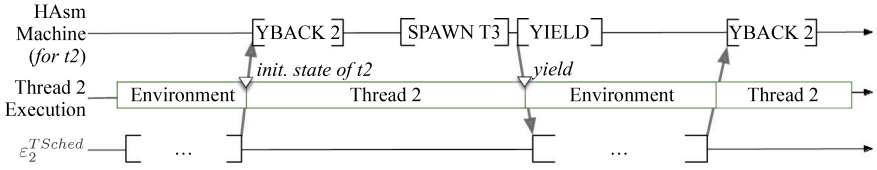

Fig. 12. Evaluation on TSched with HAsm.

HAsm is similar to LAsm but provides additional functionality that is missing in LAsm, enabling the correct assignment of dynamic initial states for individual threads. Moreover, HAsm shares the same machine state and transition rules as LAsm, ensuring effortless integration with the CompCertX compiler without altering the compilation process. This makes HAsm a versatile toolkit for users to incorporate thread-local functionalities into libraries, such as queuing locks and library call dispatchers. Additionally, ThreadAbs provides a template layer declaration, $\mathtt{TSched}[tid, \varepsilon_{tid}^{\mathtt{TSched}}]$, which serves as the bottom layer interface for HAsm. Similar to the constraints of the CSched layer, we rely on a few restrictions that the TSched layer must adhere to. The layer contains the same primitive domain as that of CSched (see Definition 4).

Definition 4 (*Primitive Domain in* CSched *and* TSched)**.** For any primitive ID (fid), the following should hold:

$(fid, \_) \in \mathtt{CSched}[\_, \_] \leftrightarrow (fid, \_) \in \mathtt{TSched}[\_, \_]$.

Both layers share the same set of primitives, but they exhibit distinct behaviors for these primitives. Fig. 12 illustrates the evaluation of a single thread on TSched with HAsm. It starts with the thread's proper initial state, using the initial log of the parameterized thread. Notably, scheduling primitive calls in TSched differ significantly from those in CSched. In TSched, these calls consistently return control to the invoking thread, updating its log with shared events representing potential interleaving by other threads. Fig. 13 defines the yield transition rules and the initial state of HAsm. Unlike the yield transition rules in Fig. 3, which update the state while preserving the local state (except for updating the PC value in the register set), these rules only update the log, leaving the local state unchanged. The initial state of the function relies on two auxiliary functions, $\mathrm{IPD}_{\mathsf{thrd}}$ and $\mathrm{IPC}_{\mathsf{thrd}}$, to accurately assign the initial state for thread tid.

5.2. Refinement proof and top-level theorem

ThreadAbs establishes a formal connection between two template layers, $\mathtt{TLink}[tid, \varepsilon_{tid}^{\mathtt{TLink}}]$ on STAsm, and $\mathtt{TSched}[tid, \varepsilon_{tid}^{\mathtt{TSched}}]$ on HAsm. This step necessitates an additional relation akin to AbsRelC in Lemma 1, bridging the semantic gap between template layer definitions and machine states. To this end, ThreadAbs defines an abstract relation called AbsRelT. Unlike AbsRelC, relations in AbsRelT are mostly straightforward due to the similarity of the two layer interfaces. Both interfaces deal with states for single threads, treating other threads' evaluations as environmental steps, although state representations and rules for these environmental steps may vary slightly due to differences in primitive sets and state/transition definitions between template layers and machine models. For instance, "HAsm" does not have explicit yield evaluation rules; instead, it relies on a user-defined specification in a TSched layer. The contextual refinement between these two layer interfaces is outlined in Lemma 4, with the full definitions and proofs available online [25].

Lemma 4. *For any program* P*, a thread ID* tid*, two template layer interfaces (*TLink *and* TSched*), environmental contexts (*$\varepsilon_{tid}^{\mathtt{TLink}}$ *and* $\varepsilon_{tid}^{\mathtt{TSched}}$*), and a refinement relation* $R_{(\mathrm{HAsm,STAsm})}$*, the following holds:*

$$[\![\mathtt{TLink}[tid, \varepsilon_{tid}^{\mathtt{TLink}}]\langle \mathtt{P} \rangle]\!]_{\mathrm{STAsm}} \sqsubseteq_{R_{(\mathrm{HAsm,STAsm})}} [\![\mathtt{TSched}[tid, \varepsilon_{tid}^{\mathtt{TSched}}]\langle \mathtt{P} \rangle]\!]_{\mathrm{HAsm}}$$

With the assumption that two layers with two different machines ($\mathtt{TSched}[tid, \varepsilon_{tid}^{\mathtt{TSched}}] \vdash_{\mathrm{HAsm}}$ *and* $\mathtt{TLink}[tid, \varepsilon_{itd}^{\mathtt{TLink}}] \vdash_{\mathrm{STAsm}}$*) satisfies the relation* AbsRelT *and* $R_{(\mathrm{HAsm,STAsm})}$ *includes* $\mathcal{M}_{data(\mathrm{HAsm,STAsm})}$ *in it.*

Similar to Lemma 1, users later have to provide $\mathcal{M}_{data(\mathrm{HAsm,STAsm})}$. They also have to provide the evidence of AbsRelT based on program-specific definitions that correspond to template layers, TSched and

YIELD RULE

$$\frac{lst = (\rho, m, adt) \quad l' = l \cdot (tid\ \texttt{YIELD}) \quad l'' = l' +\!\!+ (\varepsilon_{tid}^{\texttt{TSched}}\ tid\ l') \quad l''' = l'' \cdot (tid\ \texttt{YBACK}) \quad \rho' = \rho/[\textbf{PC} := \textbf{RA}\ \rho] \quad lst' = (\rho', m, adt)}{\texttt{TSched}[tid, \varepsilon_{tid}^{\texttt{TSched}}](\texttt{yield}) \vdash_{\text{HAsm}} \sigma_{\texttt{yield}}(\{\}, lst, l) \Rightarrow (\{\}, lst', l''')}$$

INITIAL STATE

$$\frac{m_{init} = \texttt{initmem}(P) \quad a_{init} = \text{IPD}_{\text{thrd}}(tid, l_{init}) \quad PC_{init} = \text{IPC}_{\text{thrd}}(\texttt{globalenv}(P), tid, l_{init}) \quad \rho_{init} := \{\overline{\texttt{VUndef}}\}/[\textbf{PC} := PC_{init}, \textbf{ESP} := 0]}{\text{initST}_{\text{HAsm}}(P) = (\rho_{init}, m_{init}, a_{init}, l_{init})}$$

Fig. 13. Yield Rule and initial State in TSched with Thread-local Layer Interfaces.

TLink. However, when users instantiate placeholders for those template layers and the refinement relation between them, ThreadAbs automatically offers a contextual refinement lemma with their concrete definitions. With all those steps, ThreadAbs provides the top-level theorem of the thread abstraction process, and Theorem 5 depicts the top-level property that ThreadAbs provides. We omit AbsRelC and AbsRelT in the statement for simplicity.

Theorem 5 (*Multithreaded Linking*). *For any program* P, *a CPU ID* cid, *one thread ID* tid *runs on CPU* cid, *two template layer interfaces (*CSched *and* TSched*), environmental contexts* ($\varepsilon_{cid}^{\texttt{CSched}}$ *and* $\varepsilon_{tid}^{\texttt{TSched}}$), *and a refinement relation* $R_{\text{ThreadAbs}}$, *the following holds:*

$$[\![\texttt{CSched}[cid, \varepsilon_{cid}^{\texttt{CSched}}]\langle \texttt{P}\rangle]\!]_{\text{LAsm}} \sqsubseteq_{R_{\text{ThreadAbs}}} [\![\texttt{TSched}[tid, \varepsilon_{tid}^{\texttt{TSched}}]\langle \texttt{P}\rangle]\!]_{\text{HAsm}}$$

with the assumption that $R_{\text{ThreadAbs}} = R_{(\text{HAsm,STAsm})} \circ R_{(\text{STAsm,MTAsm})} \circ R_{(\text{MTAsm,MTAsm})} \circ R_{(\text{MTAsm,LAsm})}$.

Proof. Applying Lemmas 2, 4, 1, and 3 directly justify the theorem.

6. Example

We extend CertiKOS [14] by incorporating ThreadAbs, demonstrating how ThreadAbs enhances verified software by introducing thread abstraction through thread-local layer interfaces. CertiKOS is well-regarded for its comprehensive functional correctness and includes various shared components, such as memory management modules and an IPC module with synchronization capabilities via condition variables, organized within a layered framework. To integrate ThreadAbs into CertiKOS, we have made structural adjustments to align with the format depicted in Fig. 2.

We divided CertiKOS into two parts: $\text{CertiKOS}_{\text{cpu}}$ and $\text{CertiKOS}_{\text{thrd}}$. The first part is built around a shim layer that offers TCB primitives, enabling functionalities beyond what CompCert instructions provide (*e.g.*, fetch and increase operations). $\text{CertiKOS}_{\text{cpu}}$ then offers the top-layer interface PBThrd, which includes thread libraries like thread spawn, yield, sleep, and wake up. $\text{CertiKOS}_{\text{thrd}}$ focuses on delivering thread-local services such as IPC, trap handling, and system call dispatch while concealing the behaviors of other threads within its environmental context. The bottom layer of $\text{CertiKOS}_{\text{thrd}}$ corresponds to PHThrd. These two layers, PBThrd and PHThrd, map to CSched and TSched in Fig. 2, respectively. We employ ThreadAbs to connect them through contextual refinement proofs.

To achieve our goal, a series of tasks must be completed. First, we need to establish precise definitions for the placeholders within template layer definitions, specifically CSched, TLink, and TSched. Following this, we must create abstract definitions essential for intermediate machine models, as depicted in Fig. 7. Additionally, we are required to instantiate the placeholders for template refinement relations, AbsRelC and AbsRelT, with concrete definitions. Lastly, we need to provide the necessary primitive-wise proofs to demonstrate the reliance of ThreadAbs on these definitions, with particular emphasis on AbsRelC and AbsRelT. Among them, our initial task involves defining

```
1  // IPD_thrd for CertiKOS
2  Definition cal_init_pd (ps_thrd : P) (parent: Z) (q: Z)
       (buc : block) (uc_ofs : int): P :=
3    let uctx := ⋯ in (ps_thrd {AC_thrd: (mkContainer q 0
       parent nil true)} {uctxt_thrd: uctx}).
4
5  Fixpoint thrd_init_pd_search (curid: Z) (l : L(curid))
       : P :=
6    match l with
7    | ev::tl =>
8      match ev with
9      | (PRIM parent thread_spawn ((Lint elf_id)::(Lptr
       buc uc_ofs)::(Lint q)::nil) snap)
10     if decide (name=thread_spawn)) && decide (curid =
       snap.res)
11     then match args with
12             | (Vint elf_id)::(Vptr buc uc_ofs)::(Vint
       q)::nil
13               => (cal_init_pd P_I(nm) parent (Int.
       unsigned q) buc uc_ofs) ⋯
14
15 Definition IPD_thrd (curid: Z) (init_log : L) : P :=
16   if (decide (curid= tid_m)) then P_I(m) else
       thrd_init_pd_search curid init_log(curid).
17
18 // update for CertiKOS
19 Definition apply_event (ev : ev_thrd) (ss : S) : option
       S :=
20   match ev with
21   | (PRIM parent thread_spawn ((Lint elf_id)::(Lptr
       buc uc_ofs)::(Lint q)::nil)  _)
22     match thread_spawn_spec_share pid  buc uc_ofs (Int
       .unsigned q) ss with
23     | Some (ss', _) => Some ss' | _ => None end _
24
25 Fixpoint update (ss: S) (l : L) : S :=
26   match l with
27   | nil => ss
28   | hd::tl => let ss' := update ss tl in
29                 match apply_event hd ss' with | Some
       ss'' => ss'' | _ => ss'
30   end end.
```

Fig. 14. Auxiliary function Implementations in CertiKOS.

the PHBThrd layer for CertiKOS, corresponding to TLink, since concrete versions of the CSched and TSched layers already exist. This step also encompasses the instantiation of various components upon which PHBThrd relies, including different data types, thread configurations, and auxiliary functions, all detailed in Fig. 7.

```
1  Definition match_AC_per_pd (id : ℤ) (ss : 𝒮) (ps: 𝒫)(
       adt: (pst_CPU, ℒ_cpu)) :=
2    if decide (id=tid_m) then AC_thrd ps = ZMap.get id (AC_cpu
       adt)
3     if init_cpu sd then if B_GetContainerUsed id (CPU_ID
       _cpu adt) (log_cpu adt)
4       then AC_thrd ps = ZMap.get id (AC_cpu adt) else ⋯
5
6  Record 𝓜_data(MTAsm,LAsm) (ss : 𝒮) (psp: ZMap.t (option
       𝒫)) (adt : (pst_CPU, ℒ_cpu))) :=
7    mkmatch {
8      CPU_ID_re: CPU_ID_thrd ss = CPU_ID_adt adt;
9
10     container_re: forall i,
11       match ZMap.get i psp with
12       | Some ps => match_AC_per_pd i ss ps adt | None
         => True
13       end;
14
15     pperm_disjoint:  forall i j pd1 pd2,  i <> j ->
16       ZMap.get i psp = Some ps1 -> ZMap.get j psp =
         Some ps2 ->
17       (forall k, ZMap.get k (pperm_thrd ps1) <> PGUndef ->
         ZMap.get k (pperm_thrd ps2)= PGUndef)
18         ⋯ }.
```

Fig. 15. Parts of $\mathcal{M}_{data(\text{MTAsm,LAsm})}$ in CertiKOS.

Fig. 14 highlights key auxiliary functions employed in CertiKOS when integrated with ThreadAbs. Consider IPD_{thrd} (line 15), which determines how each CertiKOS thread computes its individual thread private state. Initially, it checks if the thread is an init thread (line 16). For init threads, it directly returns an appropriate initial private state ($\mathcal{P}_{I(m)}$) since they start with a predefined system initial state. However, for non-init threads, it invokes `thrd_init_pd_search` (line 5) to search for the corresponding thread spawn event in the initial log. If such an event exists (lines 8–10) and spawns the thread (`curid = snap.res` in line 10), the function calls `cal_init_pd` (line 2) to dynamically allocate the initial private data. This process updates the thread's container with five elements (`AC_{\mathrm{thrd}}: (mkContainer q 0 parent nil true)`), representing maximum page quota, current page usage, parent thread ID, a list of children, and a flag indicating thread initialization. It also assigns user context information from the corresponding thread spawn function call. Additionally, update outlines how the log update function is defined. Given a shared state and a log, it iterates through the log, updating the shared state based on a case analysis of each event. For instance, line 21 utilizes `thread_spawn_spec_share` to specify how a thread spawn operation affects the shared state. We also provide proofs for fundamental properties, such as ensuring the current thread ID always resides in D_{thrd}. These properties are crucial for ThreadAbs to establish a thread abstraction theorem.

Extending CertiKOS with ThreadAbs involves relating primitive call transitions across three layers with different machine models. ThreadAbs provides crucial guidance through abstract relations `AbsRelC` and `AbsRelT` (covered in sections 4.1.3 and 5.2). Establishing these relations also requires defining two underlying refinement relations, $\mathcal{M}_{data(\text{MTAsm,LAsm})}$ and $\mathcal{M}_{data((\text{HAsm,STAsm}))}$, with a segment shown in Fig. 15. Within this framework, the relation (line 10) handles relations for all objects within CertiKOS. While relations for most objects in abstract states, such as the current CPU ID value (line 12), are straightforward, those affected by thread spawning and system initialization, like the container object relation (line 1 and 10), necessitate meticulous case analyses. These analyses depend on thread IDs, the overall system's initialization, and individual thread initialization specifics. Properties related to the multiple private states of all threads are also vital, ensuring proper segmentation of CPU-local states into distinct thread private states. For instance, CertiKOS, supporting dynamic page allocation, mandates mutual exclusiveness of page permission table updates (line 15) to divide a single page permission table within a CPU-local state into multiple tables, each linked to corresponding threads on the CPU.

Concrete definitions for `AbsRelC` and `AbsRelT` are also essential. While these processes are intricate and system-specific, ThreadAbs offers valuable guidance and generic proofs. We envision ThreadAbs as broadly applicable, extending beyond CertiKOS to similar proofs requiring thread abstraction, akin to those within CertiKOS. Lastly, we present the top-level thread abstraction theorem for CertiKOS with ThreadAbs in Theorem 6, accessible online [26].

Table 1
Statistics for ThreadAbs.

Components	LOC	
	Spec.	Proof
Auxiliary Func./Thread Conf.	538	60
Languages	2,017	1,803
Refinement Proofs	2,906	7465

Theorem 6 (*Thread Abstraction in* CertiKOS). *For any context program* P*, a CPU ID* cid*, a thread ID* tid *runs on the CPU, two layer interfaces (*PBThrd *and* PHThrd*), environmental contexts (*$\epsilon_{cid}^{\text{PBThrd}}$ *and* $\epsilon_{tid}^{\text{PHThrd}}$*), and a refinement relation* $R_{\text{ThreadAbs(CertiKOS)}}$*, the following holds:*

$$[\![\text{PBThrd}[cid, \epsilon_{cid}^{\text{PBThrd}}]\langle\langle \text{P} \rangle\rangle]\!]_{\text{LAsm}} \sqsubseteq_{R_{\text{ThreadAbs(CertiKOS)}}} [\![\text{PHThrd}[tid, \epsilon_{tid}^{\text{PHThrd}}]\langle\langle \text{P} \rangle\rangle]\!]_{\text{HAsm}}$$

Proof. With the given CertiKOS specific thread configurations, auxiliary functions, and instances of `AbsRelC` and `AbsRelT`, proving the theorem is straightforward with Theorem 5.

7. Evaluation

7.1. Proof efforts

Table 1 provides statistics regarding the proof efforts associated with ThreadAbs. We have crafted 598 lines of code encompassing auxiliary functions and thread configurations, serving as placeholders for refinement proofs of target software (e.g., CertiKOS). In addition, 3,820 lines of code pertain to language definitions. It is worth noting that a significant portion of these definitions align with `Asm` definitions in CompCert, meaning the actual effort required for defining these languages is not extensive. We have future plans to efficiently manage duplicate parts shared between machine models in ThreadAbs and `Asm` within CompCert. These line numbers also cover code for basic properties of these languages, including *receptiveness* and *determinism*, to support basic small step libraries in CompCert. Employing these definitions, we have created a refinement library capable of linking `TSched` on HAsm with `CSched` on LAsm. Importantly, these proofs and definitions are program-independent, making them highly reusable for users.

Table 2 provides insights into our CertiKOS adaptation with ThreadAbs for thread abstraction in CCAL. We instantiated three key layers: `PBThrd`, `PHBThrd`, and `PHThrd`, corresponding to `CSched`, `TLink`, and `TSched`, respectively.

The table breaks down the lines of code (LOC) for Memory Accessor, Spec., and Layer Def. These aspects pertain to each layer's definition in CCAL. Each layer definition encompasses three crucial components: (1) memory accessor definitions, which detail memory load and store

Table 2
Statistics for linking example with CertiKOS.

Components	LOC		Components	LOC	
	Spec.	Proof		Spec.	Proof
Mem. Accessor (PBThrd)	815	372	Mem. Accessor (PHBThrd)	748	361
Mem. Accessor (PHThrd)	845	402	Spec. (PBThrd)	2,692	N/A
Spec. (PHBThrd)	1,878	N/A	Spec. (PHThrd)	2,582	N/A
Layer Def.[a] (PBThrd)	850	339	Layer Def.[a] PHBThrd)	723	225
Layer Def.[a] (PHThrd)	773	335	Aux. Func./Thrd Conf. Instance	1,628	1,451
Refinement Proofs (Low)[b]	3,487	18,545	Refinement Proofs (High)[c]	975	4,616

[a] Including invariant proofs.
[b] Refinement proofs between PBThrd and PHBThrd (Instance of AbsRelC).
[c] Refinement proofs between PHBThrd and PHThrd (Instance of AbsRelT).

```
Function yield_spec (adt: RData) (rs: KContext) :
    option (RData * KContext) :=
  let curid := (ZMap.get (CPU_ID adt) (cid adt)) in
  …
  Some (adt0 {kctxt: curid_context)}
             {abtcb: new_tcb}
             {cid: scheduled_thread},
       restored_context)
  …
```

(a) Yield specification in the earlier version of CertiKOS

```
Function yield_spec (adt: RData) : option RData :=
  let curid := (ZMap.get (CPU_ID adt) (cid adt)) in
  …
  if IsProper curid then
    Some (adt {big_log: BigDef updated_log}) else None
  …
```

(b) Yield specification in the extended version of CertiKOS

Fig. 16. Top-level yield specifications for two distinct versions of CertiKOS.

```
Record ShareMemInv (pdp: ZMap.t (option privData)) :=
  …
  pperm_disjoint: forall t1 t2 pd1 pd2, t1 <> t2 ->
    ZMap.get t1 pdp = Some pd1 ->
    ZMap.get t2 pdp = Some pd2 ->
    (forall p, ZMap.get p (pperm pd1) <> PGUndef ->
      ZMap.get p (pperm pd2)= PGUndef)
}.
```

Fig. 17. Disjoint property of memory permissions defined in the instance of AbsRelC.

semantics, (2) specifications of layer-specific primitives, and (3) the actual layer definition.

For our thread abstraction task, we meticulously addressed all three components for each layer. These tasks can be demanding due to the need for copying and manual rewriting, mainly driven by inconsistencies in abstract data definitions across layers. Yet, many of these definitions closely resemble each other. For example, memory accessor definitions remain highly similar across layers, differing mainly in the abstract data they employ. The instantiation of auxiliary functions, thread configurations, and basic properties spans 3,079 lines of code, as reflected in the Aux. Func./Thrd Conf. Instance column.

The most intricate part lies in the refinement proofs, demonstrating that our layer definitions align with AbsRelC and AbsRelT. While these sections are extensive, part of the complexity arises from the layers' size. The PHBThrd layer, for instance, boasts 65 primitives. This implies that PBThrd and PHThrd together encompass approximately 67 primitives, encompassing functions like yield and sleep. Furthermore, AbsRelC and AbsRelT incorporate memory accessor assumptions between layers. Despite the volume, the proof efforts for each primitive reasonably match their complexity.

7.2. Experience report

There are two key advantages of the extended version of CertiKOS over its previous version. The enhanced CertiKOS with ThreadAbs offers simplified and thread-local specifications for primitives by concealing interleaving and intertwined behaviors among multiple threads in the system. Additionally, it inherently ensures thread isolation properties during the proof of thread abstraction.

The critical part demonstrating the concealment of interleaving and intertwined behaviors lies in specifications of scheduling primitives that are revealed to user programs. The formal differences have already been discussed, as illustrated in the variances between Figs. 3 and 13, but Fig. 16 highlights the distinctions between them. In Fig. 16(a), the previous version unveils various unnecessary details to users, including context switching ({kctxt: curid_context)} and restored_context), updated information in the thread control block ({abtcb: new_tcb}), and the alteration of the thread ID due to the scheduling ({cid: scheduled_thread}). In this sense, this approach fails to deliver thread-local behavior to each user that runs on top of CertiKOS. At least at the specification level, all user programs on the same CPU expose intricate information updated during their scheduling, revealing the scheduling path and state updates in between their scheduling. This makes specifications for users unnecessarily complex and may require additional proofs to show exclusiveness for parts of resources that are not shared by other threads in the same CPU.

However, the specification of our extended version in Fig. 16(b) solely updates the log during the transition while maintaining the current thread ID unchanged. Also, it does not perform context switching as parts of scheduling. Therefore, this specification offers a much simpler behavior to each user compared to the previous version. It also serves as the key to providing an isolated view for each thread. Unlike the previous version, users are unaware of which threads are scheduled between their executions. Moreover, the context of each thread is concealed from others in the specification itself, enabling an intuitive and natural achievement of the isolation property for kernel resources.

This was not available in the previous version. The same principle for thread isolation is applicable to more complex shared resources, such as memory, even though their properties should be specified as parts of the refinement relation AbsRelC and be shown that all transitions and states satisfy those properties during the proof. For instance, Fig. 17 displays a segment of the refinement relation in the instance of AbsRelC. It indicates that page permission pools pd1 and pd2 for two threads, t1 and t2, possess a disjoint property. This

implies that if one memory page (p) is used in one thread (ZMap.get p (pperm pd1) <> PGUndef), the other does not utilize the page (ZMap.get p (pperm pd2) = PGUndef).

However, incorporating ThreadAbs into CertiKOS also necessitated additional efforts, as elaborated in Section 7.1. Porting the CertiKOS proof to ThreadAbs took three person-months, quite efficient given the massive scale of the CertiKOS proof, which exceeds 250,000 lines of Coq code, including the new thread abstraction proof integrated with ThreadAbs. During this process, we went through multiple iterations, primarily focusing on specifications, refinements, and layer linking proofs.

A major challenge revolved around segregating the abstract data and global log in PBThrd into their counterparts in PHBThrd. With over 35 sub-objects in PBThrd, it was crucial to correctly divide them into thread-local private data and shared data (the global log in PBThrd). This required an in-depth understanding of how each primitive in the layer affected the fields in the abstract data. For instance, redefining abstract data related to certain primitives, like page allocation with a two-level page table or thread spawn, demanded unraveling intricate relationships among multiple fields.

Another formidable task was establishing the refinement relation between the CPU-local abstract data and the thread-local abstract data, which was associated with dynamic initial state allocation and lock operations (IPC channel) for each thread. These sections required exhaustive case analyses to account for all scenarios in which the system modified them. Nevertheless, despite these intricate dependencies resulting in multiple iterations, most proofs followed straightforward paths. Out of 65 primitive cases, only eight posed complex challenges due to intricate state update dependencies. For the rest, providing evidence for AbsRelC and AbsRelT was relatively straightforward, as they impacted only limited portions of the abstract states.

We believe that the challenges we encountered are common in the verification of complex software, especially when people hope to provide strong invariants such as thread isolation on shared resources, as we have demonstrated.

7.3. Limitations of ThreadAbs

ThreadAbs has limitations to consider. A significant issue relates to memory allocation within thread-local layers linked with CCAL proofs. These layers, positioned between TSched and $L_{\texttt{API}}$, cannot allocate new memory blocks. This limitation leads to larger layers linked using ThreadAbs (e.g., CSched and TSched), such as PBThrd and PHThrd in CertiKOS. Users may need to adjust certified layers or refactor code, as we did in CertiKOS. Despite these constraints, we successfully integrated all previously verified services from the original CertiKOS into our ported version.

Additionally, our framework assumes cooperative scheduling, with preemption not fully explored in the current version, as we do not see it as essential. Automation is another challenge, resulting in larger refinement proofs. For instance, we have repetitive routines in refinement proofs, verifying that each primitive adheres to calling conventions. Enhanced proof automation could significantly reduce proof sizes. Nonetheless, our framework and case study demonstrate that creating a certified thread-local library interface is feasible with reasonable effort.

8. Related work and conclusion

Program logics for shared-memory concurrency. Several program logics [3–5,27–41] have been proposed as modular formal verification techniques for shared-memory concurrent programs. Some of these program logics [32,37] support higher-order functions and sophisticated non-blocking synchronization. RefinedC [42] proposes separation logic-based proof automation for complex program design patterns with concurrency. Although neither our tool nor CCAL currently address these patterns, extending ThreadAbs and CCAL to handle them is a promising future direction, as demonstrated by a recent work, CCR [43], which shows how separation logic and contextual refinement proofs can be incorporated for program verification. Several previous works have used a concept similar to logs in our system. For example, Total-TaDA [39] can be used to prove the total correctness of concurrent programs, but it has not been mechanized in any proof assistant.

The recent work by Song et al. [43] presents a novel approach for a modular formal verification framework called Conditional Contextual Refinement (CCR). This approach offers a richer level of expressiveness compared to CCAL and ThreadAbs as it allows for dynamic allocation and mutual recursion. Additionally, it strives to enhance the automation of proofs by combining separation logic with contextual refinement-style proofs. However, it is worth noting that the handling of concurrency is still a work in progress for the CCR tool, and the paper does not delve into investigating the underlying machine model changes caused by software schedulers, unlike ThreadAbs. One of our future research goals is to explore the integration of our approach with CCR. Cuellar [44] introduces a concurrency semantics called the Concurrent Permission Machine (CPM) for CompCert, which enables sequential reasoning for program optimizations. Notably, they address certain limitations of CCAL by allowing dynamic memory allocations. Their model incorporates catch-fire semantics, employing locks to prevent race conditions. Additionally, they present a variant of concurrent separation logic that can be applied to demonstrate the absence of races in a given program. While their work offers a novel approach to verifying concurrent programs, it does not provide a methodology for altering the underlying assembly model during the verification of a single target software, as ThreadAbs does.

Parallel composition in concurrent program verification. Most concurrent languages, for both distributed memory and shared memory, use a parallel composition command, $(C_1|C_2)$, to create and terminate new threads. The composition command combines proofs for multiple threads and is key to demonstrating the correctness of the entire system. ThreadAbs proofs inherently include a composition command and the correctness of composition for multiple threads. Our parallel layer composition occurs in the specific layer TLink and is embedded in Lemma 1, and the composition must always be done over the entire program P and over all members of D_{thrd}.

Liang et al. [7,11,45,46] developed several methodologies based on rely-guarantee-based simulation (RGSim) for supporting parallel composition and contextual refinement of concurrent programs. The contextual refinement proof in ThreadAbs is a variance of RGSim. It extends RGSim by adding auxiliary states, such as environmental contexts and shared logs, which are essential to our ThreadAbs framework. In addition, most existing RGSim systems are limited to reasoning about atomic objects in a single layer since their client program context cannot be the method body of another concurrent object. In this sense, they cannot support view changes in ThreadAbs, which require a vertical composition of multiple layers.

One recent study [46] proposes a method for specifying and verifying the progress of concurrent objects with partial methods, but the mechanized proof is beyond the scope of their research and they do not provide formal linking for view changes. The Bedrock [12] project provides a verified toolkit for multithreaded programs, allowing dynamic allocation and connection of thread-local services and underlying library components. However, it lacks support for building thread-local interfaces that can handle view changes, which is a key feature of ThreadAbs. Furthermore, a recent study [47] showcases the application of linearizability to intricate concurrent software. However, it has not been integrated into any formal verification toolkit, such as CCAL. In another work, Lee et al. [48] introduce a novel generic fairness semantics for concurrent objects, along with illustrative small-scale examples. These developments present an opportunity to potentially streamline proofs within ThreadAbs, although further investigations are needed to confirm this potential benefit.

Extending compcert and verified compilation. Compositional CompCert [49] extends the original CompCert compiler [50] to support compositional thread-safe compilation of concurrent Clight programs. They introduced an interaction semantics approach, following Beringer et al. [51], which treats synchronization-primitive calls as external calls. However, their work does not support a layered language like CCAL, and thus cannot be directly connected to ThreadAbs nor support view changes inside it. Additionally, their work did not investigate concurrency, despite their interaction model being designed for shared-memory concurrency support. While Kang et al. [52] and Ramananandro et al. [53] also modified the CompCert compiler for separate compilation and composition, they did not support concurrency. Similarly, CompCertM[54] and CompCertO[55] propose verified compilers based on CompCert that generalize compositionality of modular compilation, but they do not focus on the view change as ThreadAbs does. Other works on verified compilation [56–59] do not support concurrent and/or compositional programs.

Multithreaded library (kernel) verification. There is a significant amount of work related to kernel verification. For instance, seL4 [60,61], Verve [62], and hyperkernel [63] have addressed multithreaded library (kernel) verification. Xu et al. [40] developed a new verification framework by combining RGSim and Feng et al.'s program logic [64] to reason about interrupts. They verified multiple key modules written in C in μC/OS-II, a preemptive kernel. However, it is important to note that their research has a distinct focus compared to ours. They concentrate on constructing a formal semantics of a C subset and verifying nested interrupts within that formal model. In contrast, our approach takes the verification process down to the assembly level, which is crucial for scheduling primitives due to the necessity of handling context switching that violates C calling conventions and requires direct manipulation of registers. Additionally, their work primarily aims to demonstrate the correctness of their low-level code without incorporating concurrent abstractions related to threads, as is supported by ThreadAbs. In this regard, their work aligns more closely with the earlier version of CertiKOS before the introduction of the ThreadAbs extension. Nevertheless, we believe that their exploration of preemptive scheduling in their work could serve as a valuable reference for extending ThreadAbs to encompass richer scheduling primitives, which are currently absent in our framework.

Some others, such as seL4 [60], Verve [62], and hyperkernel [63], aimed to prove several properties based on a single-threaded or limited concurrency support model using a per-core big kernel lock. The Verisoft team also verified spinlocks in a hypervisor using VCC [65], where they directly postulated a Hoare logic instead of building on operational semantics for C. However, none of these works deal with a *view change* caused by the software scheduler, which we have demonstrated in our case study. Furthermore, SeKVM [16] and its associated research [66] have demonstrated the correctness of a portion of the KVM hypervisor using CCAL. While their work operates within a relaxed memory model environment, it does not primarily emphasize thread abstractions. Incorporating these studies as additional examples within our framework is also a future area of exploration.

9. Conclusion

This paper presents ThreadAbs, a generic framework for constructing certified thread-local interfaces that support dynamic thread initial states, based on the verification toolkit CCAL [10]. ThreadAbs introduces intermediate machine models that decouple the states and transition rules owned by a CPU into separate states and transition rules for each CPU, with abstractions for other threads. It also provides contextual refinement proofs and templates that can be applied to multiple layers with a few restrictions. This enables the gluing of two different certified layers in CCAL, where one layer is parameterized by one CPU and the other by one thread. To achieve this, ThreadAbs separates the core part of thread abstraction from program-specific parts, defines multiple abstract definitions, intermediate languages, and generic thread abstraction proofs that can be instantiated by any program that uses the framework. The methodology is demonstrated by porting CertiKOS, a formally verified operating system with CCAL, to the framework.

As a future direction, we aim to remove the assumptions underlying the framework and proofs and improve the expressiveness and automation of the framework. We also plan to simplify the framework itself.

CRediT authorship contribution statement

Jieung Kim: Conceptualization of this study, Methodology, Software. **Jérémie Koenig:** Conceptualization of this study, Methodology, Software. **Hao Chen:** Conceptualization of this study. **Ronghui Gu:** Conceptualization of this study, Methodology, Software. **Zhong Shao:** Conceptualization of this study, 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. Jieung Kim reports financial support was provided by Defense Advanced Research Projects Agency. Jeremie Koenig reports financial support was provided by National Science Foundation. Jeremie Koenig reports financial support was provided by Defense Advanced Research Projects Agency. Hao Chen reports financial support was provided by National Science Foundation. Hao Chen reports financial support was provided by Defense Advanced Research Projects Agency. Ronghui Gu reports financial support was provided by National Science Foundation. Ronghui Gu reports financial support was provided by Defense Advanced Research Projects Agency. Zhong Shao reports financial support was provided by National Science Foundation. Zhong Shao reports financial support was provided by Defense Advanced Research Projects Agency. Zhong Shao is a founder 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) grants 1521523 and 1715154 and DARPA 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] C.B. Jones, Tentative steps toward a development method for interfering programs, ACM Trans. Program. Lang. Syst. 5 (4) (1983).

[2] S. Ishtiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, in: Proc. 28th ACM Symposium on Principles of Programming Languages, POPL'01, 2001, pp. 14–26.

[3] 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.

[4] 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.

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

[6] M. Fu, Y. Li, X. Feng, Z. Shao, Y. Zhang, Reasoning about optimistic concurrency using a program logic for history, in: Proc. 21st International Conference on Concurrency Theory, CONCUR'10, 2010, pp. 388–402.

[7] 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.

[8] V. Vafeiadis, Concurrent separation logic and operational semantics, in: MFPS, 2011.

[9] H. Yang, Relational separation logic, Theoret. Comput. Sci. 375 (2007) 308–334.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: EAsm in ThreadAbs. https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmtlib.EAsm.html.

[20] X. Leroy, A formally verified compiler back-end, J. Automat. Reason. 43 (4) (2009) 363–446.

[21] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: Refeinement between EAsm and LAsm. https://certikos.github.io/certikos-artifact/html/mcertikos.multithread.lowrefins.AsmE2L.html.

[22] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: Refeinement between two EAsm with different threads. https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmtlib.AsmIIE2IIE.html.

[23] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: TAsm in ThreadAbs. https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmtlib.TAsm.html.

[24] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: Refeinement between TAsm and EAsm. https://certikos.github.io/certikos-artifact/html/mcertikos.conlib.conmtlib.AsmT2IIE.html.

[25] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: Refinement between HAsm and TAsm. https://certikos.github.io/certikos-artifact/html/mcertikos.multithread.highrefins.AsmPHThread2T.html.

[26] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS artifact: ThreadAbs and CertiKOS linking. https://certikos.github.io/certikos-artifact/html/mcertikos.multithread.SingleProcessor_Refinement.html.

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[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.

[42] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, D. Garg, RefinedC: Automating the foundational verification of C code with refined ownership types, 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. 158–174, http://dx.doi.org/10.1145/3453483.3454036.

[43] Y. Song, M. Cho, D. Lee, C.-K. Hur, M. Sammler, D. Dreyer, Conditional Contextual Refinement, Vol. 7, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3571232, POPL.

[44] S. Cuellar, N. Giannarakis, J.-M. Madiot, W. Mansky, L. Beringer, Q. Cao, A.W. Appel, Compiler correctness for concurrency: from concurrent separation logic to shared-memory assembly language, 2020.

[45] 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.

[46] 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.

[47] A. Oliveira Vale, Z. Shao, Y. Chen, A compositional theory of linearizability, Proc. ACM Program. Lang. 7 (POPL) (2023) http://dx.doi.org/10.1145/3571231.

[48] D. Lee, M. Cho, J. Kim, S. Moon, Y. Song, C.-K. Hur, Fair operational semantics, Proc. ACM Program. Lang. 7 (PLDI) (2023) http://dx.doi.org/10.1145/3591253.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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).

[55] 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.

[56] A. Lochbihler, Verifying a compiler for java threads, in: ESOP, 2010, pp. 427–447.
[57] J. Ševčík, V. Vafeiadis, F.Z. Nardelli, S. Jagannathan, P. Sewell, Relaxed-memory concurrency and verified compilation, in: POPL, 2011, pp. 43–54.
[58] 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.
[59] 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.
[60] 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.
[61] 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.
[62] 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.
[63] 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.
[64] 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.
[65] D. Leinenbach, T. Santen, Verifying the Microsoft Hyper-V hypervisor with VCC, in: Proc. 2nd World Congress on Formal Methods, 2009, pp. 806–809.
[66] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, R. Gu, Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 866–881, http://dx.doi.org/10.1145/3477132.3483560.