A Certified Thread Library for Multithreaded User Programs

Yu Guo Xinyu Jiang Yiyun Chen Chunxiao Lin
Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
{*guoyu,wewewe,cxlin3*}*@mail.ustc.edu.cn* *yiyun@ustc.edu.cn*

Abstract

Ensuring the safety of multithreaded software is a task both important and challenging. Currently, most approaches focus on the safety of multithreaded programs rather than the runtime based on which those concurrent programs run. In order to fundamentally solve this problem, a method of ensuring the safety of the runtime should be developed. Such a runtime could be organized as a thread library typically.

This paper presents the development and certification of a simple but realistic thread library. The thread library provides common multi-threading features such as dynamic thread creation, termination and joining as well. This library also carries machine-checkable proof which guarantees the library does not violate the safety policies. This paper also presents an approach to link the library to existing certified multithreaded user programs to form an integrated foundational proof-carrying code (FPCC) package. Comparing with the uncertified libraries, our work makes multithreaded applications much more reliable.

1. Introduction

Multithreaded software is widely employed in realistic applications. For example, in a web browser, it is common that while one thread is displaying images or text, another thread is retrieving data from the Internet. However, the safety of multithreaded programs is hard to ensure, since the interference between the simultaneously executing threads must be taken into account.

Many efforts have been devoted to the verification of concurrent programs. Jones [13] introduced the compositional rely-guarantee method (or A-G method) [26, 7, 5] to describe the state changes performed by the environment and by the program respectively. Lamport proposed the Temporal Logic of Action (TLA) [14] as a logic for specifying and reasoning about concurrent programs at the high-language level. Xu *et al.* [24] proposed a logic system to verify deadlock freedom and convergence by rely-guarantee method. Model checkers [9] are developed to verify concurrent programs with a fixed number of threads. Flanagan *et al.* [10] used A-G method to check java multithreaded programs. CCAP [26] applied the A-G method to the assembly code based on a concurrent abstract machine with a built-in thread scheduler. CMAP [7] proposed by Feng and Shao supports thread-modular reasoning with dynamic thread creation and termination.

However, most of the previous work concentrates on safe multithreaded programming, not the runtime (thread library). Nonetheless, only when the safety of underlying thread library is guaranteed, can the programs verified by their methods be safe. The thread library often contains lurking flaws which are subtle and hard to detect and fix due to its complexity, so a fully certified thread library is of urgent necessity. However, the certifying task is full of challenges. A thread library generally involves sophisticated manipulations on memory and machine context. The invariants are subtle and hard to specify. Furthermore, the control flow transfers between threads and the scheduler of thread library increase the certification burden.

In this paper, we propose a simple but realistic certified thread library, named CTL, which implements dynamic thread creation, termination and joining. CTL is written in low-level code and certified thoroughly in the program logic SCAP [8]. The certification convinces us that the semantics of CTL conform to its formal specifications. The code and its specifications, as well as corresponding safety proof are all encoded in a foundational mathematical logic and packed together to form a foundational proof-carrying code (FPCC) package [1, 12], in which a neat and consistent logic system instead of the entire complicated thread library has to be trusted. In this way, not only multithreaded programs but also CTL itself can be reasoned about and verified on a solid and rigorous base.

Even if we have the individual safety proof of them, it is still inadequate to ensure the safety of the interactions be-

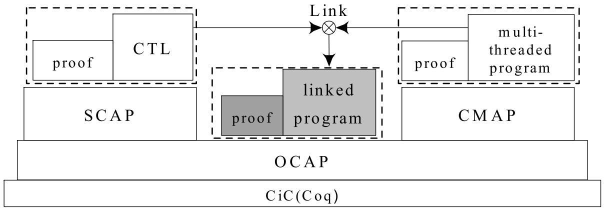

Figure 1. The OCAP framework

tween CTL and multithreaded programs due to the absence of safety proof for the code with respect to linkage. Although the construction of such safety proof is possible, it is non-trivial because of the differences between specification languages, as well as certification methods' variety.

Recently, Feng *et al.* [6] proposed an open certification framework (OCAP) to support inter-operation of different certification systems. OCAP serves as a common layer in which different program logics can be embedded.

Based on OCAP, we demonstrate in this paper the linkage between CTL certified in SCAP and user programs certified in CMAP, and construct the extra safety proof of interaction, as shown in Figure 1.

The main contributions of our work are as follows:

- We describe, specify, and certify a simple but realistic thread library CTL. It provides common multithreading features including thread scheduling, dynamic thread creation, termination and joining.
- We define formal specifications to capture the semantics of CTL routines. This library carries machine-checkable proof which ensures that the library does not violate the safety policies.
- Meanwhile, we adapt and embed CMAP in a simplified OCAP framework. Thus, CTL can be linked to the user programs certified in CMAP to construct an integrated mechanized FPCC package. As far as we know, CTL is the first thread library which can be safely linked to multithreaded user programs.

We have formalized the work presented in this paper, including CTL, OCAP, CMAP, and their soundness proof, in the Coq proof assistant [3]. Interested readers may find them on our web site [11].

The remainder of this paper is organized as follows: Section 2 presents the formalization of basic settings. In Section 3 we describe and certify the CTL library. We discuss the verification framework CMAP in which the multithreaded programs are certified in Section 4. Section 5 shows the linkage of CTL and user programs in a simplified OCAP framework. Section 6 is related work and the conclusion.

2. Basic settings

In order to certify CTL in the FPCC framework, we formalize all the related concepts into a mechanized meta-logic. In other words, the machine model, the code of CTL, its specifications, program logics and related safety proof are all based on a common formal logic, resulting in smaller trusted computing base for safety. In this section, we will present these basic settings.

The mechanized meta-Logic. We use the calculus of inductive constructions (CiC) [21] as our meta-logic. CiC is supported by the Coq proof assistant [3], which we use to implement the work presented in this paper.

The target machine. A MIPS-style [15] target machine (TM) is chosen as our machine model on which our thread library runs. We omit some physical machine features irrelevant to threading, such as address alignment, bits-arithmetic *etc.*. The target machine and its operational semantics are formally defined in Figure 2. A machine program $\mathbb{P}$ contains a code heap $\mathbb{C}$ and an updatable state $\mathbb{S}$. A code heap $\mathbb{C}$ is a segment of memory mapping addresses to machine instructions, but $\mathbb{C}$ is read-only and isolated from the mutable data heap $\mathbb{H}$. The state $\mathbb{S}$, which specifies the execution of the program, consists of a register file, mutable data heap $\mathbb{H}$ and the program counter pc. The target machine has 32 general-purpose registers. Following the MIPS convention, the table below shows the register alias and usage.

$\mathtt{r0}$	$\mathtt{r}_0$	always zero	$\mathtt{v0}-\mathtt{v1}$	$\mathtt{r}_2-\mathtt{r}_3$	return values
$\mathtt{a0}-\mathtt{a3}$	$\mathtt{r}_4-\mathtt{r}_7$	arguments	$\mathtt{t0}-\mathtt{t7}$	$\mathtt{r}_8-\mathtt{r}_{15}$	temporary
$\mathtt{k0}-\mathtt{k1}$	$\mathtt{r}_{26}-\mathtt{r}_{27}$	reserved	$\mathtt{ra}$	$\mathtt{r}_{31}$	return address

The basic TM instruction set covers the common MIPS instructions for arithmetics, jump, conditional branch and load/store. It is easy to add more instructions in our TM.

The relation $\mathbb{P} \longmapsto \mathbb{P}'$ indicates that the program state $\mathbb{P}$ steps to the program state $\mathbb{P}'$. The relation $\mathbb{P} \longmapsto^k \mathbb{P}'$ means that $\mathbb{P}$ reaches $\mathbb{P}'$ in k steps, and $\longmapsto^*$ is the reflexive and transitive closure of the step relation. The auxiliary function $\mathsf{Next}_\iota(\_)$ specifies the state transition according to the operational semantics instruction ι. We use the notation $\hat{\mathbb{S}}_\iota$ to denote $\mathsf{Next}_\iota(\mathbb{S})$.

Safety. Safety of the program means that:(*i*) the execution of the programs in CTL will not go stuck; (*ii*) the code of CTL satisfies certain safety specifications. The code heap specification Ψ is a map from code labels $\mathtt{f}$ to code specifications θ, as defined below:

$$\begin{array}{rrcl} (\mathit{CdSpec}) & \theta & ::= & \cdots \\ (\mathit{CHSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \theta\}^* \end{array}$$

$$\begin{array}{rcll}
(\mathit{Program}) & \mathbb{P} & ::= & (\mathbb{C},\mathbb{S}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \iota\}^* \\
(\mathit{State}) & \mathbb{S} & ::= & (\mathbb{R},\mathbb{H},\mathsf{pc}) \\
(\mathit{Memory}) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= & \{\mathtt{r}_k\}^{k\in\{0\ldots 31\}} \\
(\mathit{Labels}) & \mathtt{f},\mathtt{l},\mathsf{pc} & ::= & n \ \ (\mathit{nat\ nums}) \\
(\mathit{Word}) & \mathtt{w} & ::= & i \ \ (\mathit{integers}) \\
(\mathit{Instr}) & \iota & ::= & \mathtt{addu}\ \mathtt{r}_d\ \mathtt{r}_s\ \mathtt{r}_t \mid \mathtt{addiu}\ \mathtt{r}_d\ \mathtt{r}_s\ \mathtt{w} \\
 & & \mid & \mathtt{subu}\ \mathtt{r}_d\ \mathtt{r}_s\ \mathtt{r}_t \mid \mathtt{subiu}\ \mathtt{r}_d\ \mathtt{r}_s\ \mathtt{w} \\
 & & \mid & \mathtt{move}\ \mathtt{r}_d\ \mathtt{r}_s \mid \mathtt{li}\ \mathtt{r}_d\ \mathtt{w} \\
 & & \mid & \mathtt{lw}\ \mathtt{r}_t\ \mathtt{w}(\mathtt{r}_s) \mid \mathtt{sw}\ \mathtt{r}_t\ \mathtt{w}(\mathtt{r}_s) \\
 & & \mid & \mathtt{beq}\ \mathtt{r}_s\ \mathtt{r}_t\ \mathtt{f} \mid \mathtt{bgtz}\ \mathtt{r}_s\ \mathtt{f} \\
 & & \mid & \mathtt{j}\ \mathtt{f} \mid \mathtt{jal}\ \mathtt{f} \mid \mathtt{jr}\ \mathtt{r}_s
\end{array}$$

$(\mathbb{C},(\mathbb{H},\mathbb{R},\mathsf{pc})) \longmapsto (\mathbb{C},\mathsf{Next}_{\mathbb{C}(\mathsf{pc})}(\mathbb{H},\mathbb{R},\mathsf{pc})) =$	
if $\mathbb{C}(\mathsf{pc}) = \iota =$	then $\mathsf{Next}_\iota(\mathbb{H},\mathbb{R},\mathsf{pc}) = \hat{\mathbb{S}}_\iota =$
$\mathtt{addu}\ \mathtt{r}_d\ \mathtt{r}_s\ \mathtt{r}_t$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\},\mathsf{pc}+1)$
$\mathtt{lw}\ \mathtt{r}_t\ \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathsf{pc}+1)$ when $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathit{dom}(\mathbb{H})$
$\mathtt{sw}\ \mathtt{r}_t\ \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\},\mathbb{R},\mathsf{pc}+1)$ when $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathit{dom}(\mathbb{H})$
$\mathtt{beq}\ \mathtt{r}_s\ \mathtt{r}_t\ \mathtt{f}$	$(\mathbb{H},\mathbb{R},\mathsf{pc}+1)$ when $\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)$ $(\mathbb{H},\mathbb{R},\mathtt{f})$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$
$\mathtt{jal}\ \mathtt{f}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathsf{pc}+1\},\mathtt{f})$
$\mathtt{jr}\ \mathtt{r}_s$	$(\mathbb{H},\mathbb{R},\mathbb{R}(\mathtt{r}_s))$

Figure 2. The target machine TM

Note that θ has different form in different program logic and is defined in Section 5.1.

Separation logic. We define some notations common in separation logic [22, 19]. These notations are defined as shorthand and are used in the specifications of CTL to specify the data heap.

$$\begin{array}{rcl}
\mathbb{H} \Vdash A & \triangleq & A\,\mathbb{H} \\
A_1 * A_2 & \triangleq & \lambda\mathbb{H}.\exists\mathbb{H}_1,\mathbb{H}_2.\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H} \wedge \mathbb{H}_1 \Vdash A_1 \wedge \mathbb{H}_2 \Vdash A_2 \\
\mathsf{Top} & \triangleq & \lambda\mathbb{H}.\mathsf{True} \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda\mathbb{H}.\mathtt{l} \neq \mathsf{NULL} \wedge \mathbb{H} = \{\mathtt{l} \rightsquigarrow \mathtt{w}\} \\
\mathtt{l} \mapsto \_ & \triangleq & \lambda\mathbb{H}.\exists\mathtt{w}.(\mathbb{H} \Vdash \mathtt{l} \mapsto \mathtt{w}) \\
\mathtt{l} \mapsto \mathtt{w}_1,\ldots,\mathtt{w}_n & \triangleq & \mathtt{l} \mapsto \mathtt{w}_1 * \mathtt{l}+1 \mapsto \mathtt{w}_2 * \ldots * \mathtt{l}+(n-1) \mapsto \mathtt{w}_n
\end{array}$$

We use $\mathbb{H} \Vdash A$ if the heap predicate A is valid with $\mathbb{H}$. $A_1 * A_2$ asserts the heap $\mathbb{H}_1 \uplus \mathbb{H}_2$ in which $\mathbb{H}_1 \Vdash A_1$ and $\mathbb{H}_2 \Vdash A_2$ hold respectively. Top is valid with any heap. $\mathtt{l} \mapsto \mathtt{w}$ asserts a heap with only one memory cell, at address $\mathtt{l}$ with content $\mathtt{w}$.

3. CTL: a certified thread library

In this section, we give a detailed description of CTL. Because of space limitations, we cannot fully discuss the certifications of every routine of CTL. So we concentrate on certifying the yielding routine, which is an essential part of the thread library.

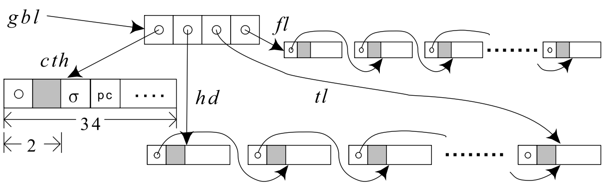

Figure 3. Core data structures of CTL

3.1. Overview

CTL is a lightweight implementation of thread library whose thread model is similar to GNU Pth [4] or FSU pthreads [16], in which threads are implemented in the user-space and the machine context switching is performed by an application library without knowledge of the kernel. This model does not rely on the kernel threads and is adaptable to various platforms. CTL supports non-preemptive scheduling and can directly run on the processors without interrupts handling, for example, the SPIM simulator [20]. CTL is fully certified at the assembly level, so the certification of the compiler correctness can be spared.

3.2. Thread model

We use pseudo C code to illustrate the prototype of the core data structures and API of CTL.

The core data structures in the CTL space are shown in Figure 3. The main data structure is a queue, whose elements are called thread control block(TCB). Each TCB identifies one dynamic thread and contains one thread id and corresponding executing context. The TCB is prototyped by:

```
struct t_tcb {
    word id;   /* unique id */
    word pc;   /* program counter */
    word state; /* ready, dead or wait */
    word wait_id; /* id of thread to wait */
    word regfile[28];
        /* array for saving registers */
};
```

Meanwhile, an isolated TCB identifies the current running thread. Note that the global pointers referring to these data structures are stored in a global pointer array.

CTL provides a threading API:

```
void ctl_yield(void);
word ctl_spawn((* void)());
word ctl_exit();
word ctl_join(int thread_id);
```

$$
\begin{array}{lccl}
(\mathit{StPred}) & \mathtt{p} & ::= & \mathit{State} \to \mathit{Prop} \\
(\mathit{GrPred}) & \mathtt{g} & ::= & \mathit{State} \to \mathit{State} \to \mathit{Prop} \\
(\mathit{CdSpec}) & \theta_{\mathrm{SCAP}} & ::= & (\mathtt{p}, \mathtt{g})
\end{array}
$$

$$
\frac{\forall\, \mathtt{f} \in \mathit{dom}(\Psi') : \quad \Psi;\mathbb{C} \vdash_{\mathrm{SCAP}} \{\Psi'(\mathtt{f})\}\mathtt{f} : \mathbb{C}(\mathtt{f})}{\Psi \vdash_{\mathrm{SCAP}} \mathbb{C} : \Psi'} \ (\mathrm{CDHP})
$$

$$
\frac{\begin{array}{l}
\iota = \mathtt{jal}\ \mathtt{f}' \quad (\mathtt{p}', \mathtt{g}') = \Psi(\mathtt{f}') \quad (\mathtt{p}'', \mathtt{g}'') = \Psi(\mathtt{f}+1) \\
\forall \mathbb{S}.\, \mathtt{p}\, \mathbb{S} \to \mathtt{p}'\, \hat{\mathbb{S}}_\iota \\
\forall \mathbb{S}, \mathbb{S}'.\, \mathbb{S}.\mathsf{pc} = \mathtt{f} \to \mathtt{p}\, \mathbb{S} \to \mathtt{g}'\, \hat{\mathbb{S}}_\iota\, \mathbb{S}' \\
\qquad\qquad \to \mathtt{p}''\, \mathbb{S}' \wedge (\forall \mathbb{S}''.\, \mathtt{g}''\, \mathbb{S}'\, \mathbb{S}'' \to \mathtt{g}\, \mathbb{S}\, \mathbb{S}'') \\
\forall \mathbb{S}, \mathbb{S}'.\, \mathtt{g}'\, \mathbb{S}\, \mathbb{S}' \to \mathbb{S}.\mathbb{R}(\mathtt{ra}) = \mathbb{S}'.\mathbb{R}(\mathtt{ra})
\end{array}}{\Psi;\mathbb{C} \vdash_{\mathrm{SCAP}} \{(\mathtt{p}, \mathtt{g})\}\mathtt{f} : \mathtt{jal}\ \mathtt{f}'} \ (\mathrm{CALL})
$$

$$
\frac{\iota = \mathtt{jr}\ \mathtt{ra} \qquad \forall \mathbb{S}.\, \mathtt{p}\, \mathbb{S} \to \mathtt{g}\, \mathbb{S}\, \hat{\mathbb{S}}_\iota}{\Psi;\mathbb{C} \vdash_{\mathrm{SCAP}} \{(\mathtt{p}, \mathtt{g})\}\mathtt{f} : \mathtt{jr}\ \mathtt{ra}} \ (\mathrm{RET})
$$

Figure 4. SCAP

The routine `ctl_yield()` causes the current thread to yield its execution in favor of another thread with certain scheduling policy. `ctl_yield()` stores the execution context in the queue of the TCBs and picks up one TCB, loads it and switches the control to the new continuation. `ctl_yield()` adopts the round-robin scheduling algorithm.

Thread creation is achieved by using `ctl_spawn()` with a parameter of start code pointer. This function first allocates a new thread control block (TCB). Then the current machine context is cloned and stored into the TCB with a new thread id. Lastly, the new TCB is marked with ready flag and put into the queue. A thread will be running forever if it does not terminate. The role of `ctl_exit()` is stopping the current thread and marking its TCB with dead flag. The `ctl_join()` function suspends the calling thread until the specified thread terminates. It takes a thread id as argument.

3.3. Certification of CTL

SCAP. We use SCAP [8] to certify CTL. SCAP supports modular certification of assembly code with function call/return abstraction, making CTL routines well-organized. The specification constructs of SCAP and some selected SCAP rules are shown in Figure 4. A code specification θ_{SCAP} is a pair of two predicates $\mathtt{p}$ and $\mathtt{g}$. $\mathtt{p}$ is the precondition, while $\mathtt{g}$ is a predicate over the entry state and the future return state after `jr ra`. $\mathtt{g}$ relates the entry state of a code to the return state of the corresponding procedure. $\mathtt{g}$ can also be treated as a general postcondition parameterized by the entry state.

Specification constructs. Figure 5 describes the specification constructs of CTL. We use the word value `w` to specify the thread id σ. The state ts of a thread may be ready, dead or wait . A thread control block Tcb consists of a pc, a

$$
\begin{array}{rccl}
(\mathit{Thread\text{-}id}) & \sigma & ::= & \mathtt{w} \\
(\mathit{RegFileX}) & \tilde{\mathbb{R}} & ::= & \mathbb{R} \setminus \{\mathtt{r0} \leadsto \_, \mathtt{k0} \leadsto \_, \mathtt{k1} \leadsto \_, \mathtt{ra} \leadsto \_\} \\
(\mathit{TCB}) & \mathsf{Tcb} & ::= & (\mathsf{pc}, \tilde{\mathbb{R}}, \mathsf{ts}, \sigma_w) \\
(\mathit{Th\ State}) & \mathsf{ts} & ::= & \mathsf{ready} \mid \mathsf{dead} \mid \mathsf{wait} \\
(\mathit{Th\ Queue}) & \mathsf{Q} & ::= & \{\cdot\} \mid \{\sigma \leadsto \mathsf{Tcb}\} \cup \mathsf{Q}
\end{array}
$$

Figure 5. Specification constructs of CTL

partial registers file $\tilde{\mathbb{R}}$ and a thread state ts. A *partial* registers file $\tilde{\mathbb{R}}$ is a registers file $\mathbb{R}$ excluding `r0`, `k0`, `k1` and `ra`. The registers `k0` and `k1` are preserved for thread scheduling and then unusable in user programs. Because the register `r0` always holds the value of zero, it needn't to be saved. When `ra` is used to call the routines of CTL, its value is saved in the Tcb.pc field. A thread queue Q is a dictionary mapping σ to Tcb and constructed inductively.

Formal core data structures. We formally specify the core data structures that are used in our thread library CTL. As shown in Figure 3, the memory space for CTL is divided into four parts, the global pointer array, the isolated TCB for current thread, the thread queue and the free-block list.

$$
\begin{array}{l}
\mathsf{Core}(\sigma, \mathsf{Tcb}, \mathsf{Q}) \triangleq \exists gbl\,.\, gbl \mapsto cth, hd, tl, fl \\
\qquad\quad * \mathsf{QNode}(cth, \sigma, \mathsf{Tcb}, \mathsf{NULL}) * \mathsf{TQ}(hd, tl, \mathsf{Q}) * \mathsf{GoodL}(fl)
\end{array}
$$

The global pointers (cth, hd, tl, fl) are saved in the memory starting from gbl. cth points to the current thread TCB. hd and tl point to the thread queue. fl points to a memory block list, which is used for dynamic heap allocation.

$$
\begin{array}{l}
\mathsf{QNode}(p, \sigma, (\mathsf{pc}, \tilde{\mathbb{R}}, \mathsf{ts}, \sigma_w), q) \triangleq (p - 2 \mapsto q, 34) \\
\qquad\quad * (p \mapsto \sigma, \mathsf{pc}, \mathsf{ts}, \sigma_w) \\
\qquad\quad * (p + 4 \mapsto \tilde{\mathbb{R}}(\mathtt{r}_1), \ldots, \tilde{\mathbb{R}}(\mathtt{r}_{25}), \tilde{\mathbb{R}}(\mathtt{r}_{28}), \tilde{\mathbb{R}}(\mathtt{r}_{29}), \tilde{\mathbb{R}}(\mathtt{r}_{30}))
\end{array}
$$

$$
\mathsf{Cth}(p, \sigma, \mathsf{Tcb}) \triangleq \mathsf{QNode}(p, \sigma, \mathsf{Tcb}, \mathsf{NULL})
$$

We define the predicate TQ to model the queue for threads. TQ has three arguments, the formal queue Q which is isomorphism to the concrete memory data, a queue-head pointer hd and a queue-tail pointer tl. TQseg models a queue segment in the middle of the queue, with a non-null pointer pointing to the next node. TQseg is defined inductively on the structure of Q.

$$
\begin{array}{l}
\mathsf{TQseg}(\{\}, hd, tl, q) \\
\qquad \triangleq (hd = tl) \wedge (hd = q) \\
\mathsf{TQseg}(\{\sigma \leadsto \mathsf{Tcb}\}, hd, tl, q) \\
\qquad \triangleq (hd = tl) \wedge \mathsf{QNode}(hd, \sigma, \mathsf{Tcb}, q) \\
\mathsf{TQseg}(\{\sigma \leadsto \mathsf{Tcb}\} \cup \mathsf{Q}, hd, tl, q) \\
\qquad \triangleq \exists q'.\, \mathsf{QNode}(hd, \sigma, \mathsf{Tcb}, q') * \mathsf{TQseg}(\mathsf{Q}, q', tl, q) \\
\mathsf{TQ}(\mathsf{Q}, hd, tl) \\
\qquad \triangleq \mathsf{TQseg}(\mathsf{Q}, hd, tl, \mathsf{NULL})
\end{array}
$$

As defined above, a queue segment consists of several nodes modeled by QNode. The routines in the CTL may traverse, search, add a TCB node or delete one in the queue, whose structure should be preserved.

When a thread is created, the scheduler will allocate a heap block from the free block list. If a thread is joined by another one, the scheduler will free its TCB. The definition of free block-list GoodL follows the work by Yu *et al.* [25] and Xiang *et al.* [23], and it can be ported to our framework directly.

Implementation of yielding. The yielding routine is used to schedule threads and perform the context switching. The code body of the `ctl_yield()` are presented in Figure 6.

The first phase of yielding (from `YIELD` to `APPENDTCB`) consists in loading address of the current TCB to `k0`, saving the machine context to the current TCB and loading other global pointers. The code address of the next instruction of the current thread has already been saved in `ra`. `ra` has to be saved into TCB firstly, because `ra` will be used to call `SAVECTX`.

The second phase of yielding is appending the current TCB (by calling `APPENDTCB`) to the thread queue, searching for a ready TCB through the queue, and fetching it out (by calling `FETCHTCB`). Since there is at least one ready TCB (consider the one just appended) in the queue, the routine `FETCHTCB` never fails to return. The algorithm of searching ready thread depends on the scheduling policy. Here the naive FIFO method is used and then the code of searching is a loop over the thread queue.

The role of the last switch phase (from `SWITCH`) of yielding is to switch the machine context from the old context to the new context fetched by `FETCHTCB`. Concretely, it sets `k0` with the address of the new TCB, then makes a tail call to `LOADCTX`. Inside `LOADCTX`, the registers file are restored including `ra`. The last step is tricky, and when program returns, the control flow is transferred to the new thread.

Certification of yielding. Part of the formal specifications of `ctl_yield()` are also presented in Figure 6. Note that the guarantees in the middle of code body are not listed because they are unnecessary for the readers to understand the specification, although indispensable to the certification process. For the same reason, the initial state $(\mathbb{R},\mathbb{H},\mathsf{pc})$ and the final state $(\mathbb{R}',\mathbb{H}',\mathsf{pc}')$, playing their roles as parameters of $\mathtt{p}$ and $\mathtt{g}$, are omitted as well.

The specification of `ctl_yield()` $(\mathtt{p}_y, \mathtt{g}_y)$ is defined below:

$$\begin{aligned}
\mathtt{p}_y &\triangleq \exists \mathsf{Q}\,.\,(\mathbb{H} \Vdash \mathsf{Core}(\_,\_,Q) * \mathsf{Top}) \\
\mathtt{g}_y &\triangleq \lambda \begin{bmatrix} (\mathbb{R},\mathbb{H},\mathsf{pc}) \\ (\mathbb{R}',\mathbb{H}',\mathsf{pc}') \end{bmatrix} .\,\forall A,\mathsf{Q},\sigma,\mathsf{Tcb}\,.\,\exists \sigma_x\,. \\
&\qquad \begin{bmatrix} (\mathbb{H} \Vdash \mathsf{Core}(\sigma,\mathsf{Tcb},Q) * A) \\ (\mathbb{H}' \Vdash \mathsf{Core}(\sigma_x,(\mathsf{pc}',\tilde{\mathbb{R}}',\mathsf{ready},\_),Q') * A) \end{bmatrix}
\end{aligned}$$

where

$$(\mathsf{pc}',\tilde{\mathbb{R}}',\mathsf{ready},\_) = (\mathsf{Q} \cup \{\sigma \rightsquigarrow (\mathbb{R}(\mathtt{ra}),\mathbb{R},\mathsf{ready},\_)\})(\sigma_x)$$

$$\mathsf{Q}' = (\mathsf{Q} \cup \{\sigma \rightsquigarrow (\mathbb{R}(\mathtt{ra}),\tilde{\mathbb{R}},\mathsf{ready},\_)\}) \setminus \{\sigma_x \rightsquigarrow (\mathsf{pc}',\tilde{\mathbb{R}}',\mathsf{ready},\_)\}$$

`ctl_yield()` mainly performs context switching. Its precondition $\mathtt{p}_y$ requires that the memory space of CTL should be well-formed, specified by the predicate Core. $\mathtt{g}_y$ is a condition which specifies the actions performed by the whole `ctl_yield()` routine. We assume that the control flow is transferred to thread σ_x, whose TCB is $\mathsf{Tcb}_x = (\mathsf{pc}_x, \tilde{\mathbb{R}}_x, \mathsf{ready}, \_)$. Then $\mathtt{g}_y$ specifies the postconditions which `ctl_yield()` must satisfies that: (*i*) at the exiting point of `ctl_yield()`, the old machine context is added into Q; (*ii*) a ready control block Tcb_x, which contains the current register file $\mathbb{R}_x$ and the code pointer pc_x, has been fetched out of Q; (*iii*) the memory space of Core is still well-formed while the irrelevant heap A is unchanged. Obviously, the specification of `ctl_yield()` is independent of CTL scheduling algorithm. Core is related to the implementation, which should be hidden from user programs.

By the SCAP rules presented in Figure 4, it can be proved that the yielding routine satisfies these specifications.

Certification of CTL. The certification of `ctl_spawn()`, `ctl_exit()` and `ctl_join()` follows this method of certifying the `ctl_yield()` routine. We take $\mathbb{C}_{\text{CTL}}$ as the code heap of CTL and Ψ_{CTL} as the code heap specification. By the CDHP rule, we have:

$$\Psi_{\text{CTL}} \vdash_{\text{SCAP}} \mathbb{C}_{\text{CTL}} : \Psi_{\text{CTL}}$$

4. CMAP: a concurrent program logic

CMAP is a program logic for certifying multithreaded assembly code with unbounded dynamic thread creation and termination. It assumes that the register files are thread local, *i.e.,* saving and loading registers during context switching. CMAP is based on an abstract machine with built-in thread operations, *e.g.,* `yield`, `spawn` and `exit` are treated as atomic primitives. This approach simplify the certification of user programs. However, the operations are not directly supported by most existing hardwares.

In Figure 7, we present a simplified CMAP system, which is adapted to our thread model. Notice that the whole

$$\{(\mathtt{p}_y, \mathtt{g}_y)\}$$

```
YIELD:       lw k0 cth r0
             sw ra 1 k0
             sw r0 2 k0
             addiu k0 k0 4
             jal SAVECTX
```

$$\begin{aligned}\mathtt{p} = {} & \exists \mathsf{Q}, gbl, \sigma, \mathsf{Tcb}\,.\,\mathsf{Tcb.ts} = \mathsf{ready} \wedge \mathsf{Tcb} = (\mathbb{R}(\mathtt{ra}), \tilde{\mathbb{R}}, \mathsf{ready}, \_) \wedge \\ & \mathbb{H} \Vdash gbl \mapsto p, hd, tl, \_ * \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{TQ}(\mathsf{Q}, hd, tl) * \mathsf{True}\end{aligned}$$

```
LOAD:        jal LOADPTR
```

$$\begin{aligned}\mathtt{p} = {} & \exists \mathsf{Q}, gbl, \sigma, \mathsf{Tcb}\,.\,\mathbb{R}(\mathtt{t0}, \mathtt{t1}, \mathtt{t2}) = (p, hd, tl) \wedge \\ & \mathbb{H} \Vdash gbl \mapsto p, hd, tl, \_ * \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{TQ}(\mathsf{Q}, hd, tl) * \mathsf{True}\end{aligned}$$

```
APPEND:      beq t1 r0 SWITCH
             jal APPENDTCB
```

$$\begin{aligned}\mathtt{p} = {} & \exists \mathsf{Q}, gbl\,.\,\mathbb{R}(\mathtt{t1}, \mathtt{t2}) = (hd, tl) \wedge \\ & \mathbb{H} \Vdash gbl \mapsto \_, \_, \_, \_ * \mathsf{TQ}(\mathsf{Q}, hd, tl) * \mathsf{True}\end{aligned}$$

```
FETCH:       jal FETCHTCB
```

$$\begin{aligned}\mathtt{p} = {} & \exists \mathsf{Q}, gbl, \sigma, \mathsf{Tcb}\,.\,\mathbb{R}(\mathtt{t0}, \mathtt{t1}, \mathtt{t2}) = (p, hd, tl) \wedge \\ & \mathbb{H} \Vdash gbl \mapsto \_, \_, \_, \_ * \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{TQ}(\mathsf{Q}, hd, tl) * \mathsf{True}\end{aligned}$$

```
SAVE:        jal SAVEPTR
```

$$\begin{aligned}\mathtt{p} = {} & \exists \mathsf{Q}, gbl, \sigma, \mathsf{Tcb}\,. \\ & \mathbb{H} \Vdash gbl \mapsto p, hd, tl, \_ * \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{TQ}(\mathsf{Q}, hd, tl) * \mathsf{True}\end{aligned}$$

```
SWITCH:      move k0 t0
             addiu k0 k0 4
             j LOADCTX
APPENDTCB:   ...

FETCHTCB:    ...
```

$$\begin{aligned}\mathtt{p}_s = {} & \exists p, \sigma, \mathsf{Tcb}\,.\,\mathbb{R}(\mathtt{k0}) = p+4 \wedge \mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{True} \\ \mathtt{g}_s = {} & \forall A, p, \sigma, \mathsf{pc}_x, \mathsf{ts}_x\,.\,\mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, (\mathsf{pc}_x, \_, \mathsf{ts}_x, \_)) * A \\ & \quad \rightarrow \mathbb{R}(\mathtt{ra}) = \mathbb{R}'(\mathtt{ra}) \wedge \\ & \qquad \mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, (\mathsf{pc}_x, \tilde{\mathbb{R}}, \mathsf{ts}_x, \_)) * A\end{aligned}$$

```
SAVECTX:   sw r1 0 k0
           sw r2 1 k0
           ...
           sw r25 24 k0
           sw r28 25 k0
           ...
           sw r30 27 k0
           jr ra
```

$$\begin{aligned}\mathtt{p}_l = {} & \exists p, \sigma, \mathsf{Tcb}\,.\,\mathbb{R}(\mathtt{k0}) = p+4 \wedge \mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * \mathsf{True} \\ \mathtt{g}_l = {} & \forall A, p, \sigma, \mathsf{pc}_x, \mathbb{R}_x\,.\,\mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * A \\ & \quad \rightarrow \mathbb{R}'(\mathtt{ra}) = \mathsf{Tcb.pc} \wedge \tilde{\mathbb{R}}' = \mathsf{Tcb}.\tilde{\mathbb{R}} \wedge \\ & \qquad \mathbb{H} \Vdash \mathsf{Cth}(p, \sigma, \mathsf{Tcb}) * A\end{aligned}$$

```
LOADCTX:   lw r1 0 k0
           lw r2 1 k0
           ...
           lw r25 24 k0
           lw r28 25 k0
           ...
           lw r30 27 k0
           subiu k0 k0 3
           lw ra 0 k0
           jr ra

SAVEPTR:   ...

LOADPTR:   ...
```

Figure 6. Code and specification of yield(part)

system is unchanged except for the primitives of `yield`, `spawn` and `exit` are replaced with function calls in CTL.

The code specification θ_{CMAP} is a quadruple $(\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})$, The predicate $\mathtt{p}$ and the local guarantee $\breve{\mathtt{g}}$ describe the states and transitions of the program. When checking concurrent properties during the interleaving execution, we rely on the A-G method, in which the assumption $\mathtt{A}$ and the gurantee $\mathtt{G}$ are used. In one thread, the assumption $\mathtt{A}$ gives information of what atomic transitions may be performed by other threads, while the guarantee $\mathtt{G}$ holds on every atomic transition performed by the thread itself. So long as the environment (*i.e.,* the collection of all the rest of the threads) satisfies $\mathtt{A}$, the thread's behavior to the environment will satisfy its $\mathtt{G}$. Furthermore, every thread should be verified to ensure that the guarantee of any other thread satisfies its assumption.

By this method we could then certify each thread separately without worrying about the rest of the threads. That is how we achieve thread-modular reasoning. And if all threads satisfy their specifications, the following non-interference property results in correct collaboration between threads.

$$\begin{aligned}& \mathsf{NI}\,([(\mathtt{A}_1, \mathtt{G}_1, \sigma_1, \tilde{\mathbb{R}}_1), \ldots, (\mathtt{A}_n, \mathtt{G}_n, \sigma_n, \tilde{\mathbb{R}}_n)]) \triangleq \\ & \qquad \forall i \neq j\,.\,\sigma_i \neq \sigma_j \wedge \forall \mathbb{H}, \mathbb{H}'.\,(\mathtt{G}_i\ \tilde{\mathbb{R}}_i\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{A}_j\ \tilde{\mathbb{R}}_j\ \mathbb{H}\ \mathbb{H}')\end{aligned}$$

Suppose the program will yield its control by executing `jal yield`, when pc points to `f`. In order to ensure the yielding safety, the premises of YIELD rule are: (*i*) the precondition and assumption at the address $\mathtt{f}+1$ are the same in each thread; (*ii*) the local guarantee $\breve{\mathtt{g}}$ at the address $\mathtt{f}+1$ is equal to the guarantee $\mathtt{G}$; the state of yielding thread always satisfies the current precondition if it would satisfy the assumption $\mathtt{A}$ after any state transition; (*iii*) the current thread completes the required state transition specified by the guarantee $\mathtt{G}$.

Example. We give an example to explain how to certify user multithreaded programs. The example adapted from [7] is a program for unbounded dynamic thread creation as shown in Figure 8. When running, the main thread initialize 100 pieces of data with 0 to 99 respectively, and distributes them to 100 child threads. These child threads add their own data by *one* in parallel. To ensure the safety property of the program, each child thread assumes that no other threads will touch its own working data and guarantees that it will not change other threads' data. The assumptions and guarantees of the main thread and its child threads, defined in Figure 8, reflect these ideas.

Suppose the example code is $\mathbb{C}_{\mathrm{EX}}$ and its specification is Ψ_{EX}, we have the safety of the code heap $\mathbb{C}_{\mathrm{EX}}$ with CMAP inference rules.

$$
\begin{array}{rccl}
(\mathit{StPred}) & \mathtt{p} & ::= & \mathit{RegFileX} \to \mathit{Heap} \to \mathit{Prop} \\
(\mathit{Guar}) & \breve{\mathtt{g}}, \mathtt{G} & ::= & \mathit{RegFileX} \to \mathit{Heap} \to \mathit{Heap} \to \mathit{Prop} \\
(\mathit{Assume}) & \mathtt{A} & ::= & \mathit{RegFileX} \to \mathit{Heap} \to \mathit{Heap} \to \mathit{Prop} \\
(\mathit{CdSpec}) & \theta_{\text{CMAP}} & ::= & (\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})
\end{array}
$$

$$
\frac{\forall\, \mathtt{f} \in \mathit{dom}(\Psi') : \quad \Psi;\mathbb{C} \vdash_{\text{CMAP}} \{\Psi'(\mathtt{f})\}\mathtt{f} : \mathbb{C}(\mathtt{f})}{\Psi \vdash_{\text{CMAP}} \mathbb{C} : \Psi'} \; (\text{CDHP})
$$

$$
\frac{\begin{array}{l}\Psi(\mathtt{f}+1) = (\mathtt{p}, \mathtt{G}, \mathtt{A}, \mathtt{G}) \qquad \forall \tilde{\mathbb{R}}, \mathbb{H}\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \breve{\mathtt{g}}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H} \\ \forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}'\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \mathtt{A}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}' \to \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H}'\end{array}}{\Psi;\mathbb{C} \vdash_{\text{CMAP}} \{(\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})\}\mathtt{f} : \mathtt{jal\ yield}} \; (\text{YIELD})
$$

$$
\frac{\begin{array}{l}\Psi(\mathtt{f}+1) = (\mathtt{p}, \mathtt{G}, \mathtt{A}, \mathtt{G}) \quad \forall \tilde{\mathbb{R}}, \mathbb{H}\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \breve{\mathtt{g}}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H} \\ \forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}'\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \mathtt{A}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}' \to \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H}' \\ \forall \tilde{\mathbb{R}}, \mathbb{H}\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \Psi(\tilde{\mathbb{R}}\{\mathtt{a0}\}) = (\mathtt{p}', \mathtt{G}', \mathtt{A}', \mathtt{G}') \wedge \mathtt{p}'\; \tilde{\mathbb{R}}\; \mathbb{H} \\ \forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}'\,.\, \mathtt{p}'\; \tilde{\mathbb{R}}\; \mathbb{H} \to \mathtt{A}'\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}' \to \mathtt{p}'\; \tilde{\mathbb{R}}\; \mathbb{H}' \\ \mathtt{A} \overset{\circ}{\Rightarrow} \mathtt{A}' \quad \mathtt{G}' \overset{\circ}{\Rightarrow} \mathtt{G} \quad \mathtt{G} \overset{\circ}{\Rightarrow} \mathtt{A}' \quad \mathtt{G}' \overset{\circ}{\Rightarrow} \mathtt{A}\end{array}}{\Psi;\mathbb{C} \vdash_{\text{CMAP}} \{(\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})\}\mathtt{f} : \mathtt{jal\ spawn}} \; (\text{SPAWN})
$$

$$
\frac{\forall \tilde{\mathbb{R}}, \mathbb{H}\,.\, \mathtt{p}\; \tilde{\mathbb{R}}\; \mathbb{H} \to \breve{\mathtt{g}}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}}{\Psi;\mathbb{C} \vdash_{\text{CMAP}} \{(\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})\}\mathtt{f} : \mathsf{jal}\ \mathtt{exit}} \; (\text{EXIT})
$$

where

$$
\mathtt{G} \overset{\circ}{\Rightarrow} \mathtt{A} \quad \triangleq \quad \forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}'\,.\, \mathtt{G}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}' \to \mathtt{A}\; \tilde{\mathbb{R}}\; \mathbb{H}\; \mathbb{H}'
$$

Figure 7. CMAP

```
MAIN:   (True, A1, G1)
        move t0 r0
        addiu t1 r0 100
LOOP:   (p, G2, A2, G2)
        beq t0 t1 CONT
        jal YIELD
        sw t0 data t0
        jal YILED
        move a0 CHLD
        move a1 t0
        jal SPAWN
        (p, G3, A3, G3)
        jal YIELD
        addiu t0 t0 1
        jal YIELD
        j LOOP
CONT:   (p', G4, A4, G4)
        jal EXIT
CHLD:   (p'', A, G)
        lw t0 data a1
        jal YIELD
        addiu t0 t0 1
        jal YIELD
        sw t0 data a1
        jal EXIT
```

$$
\begin{array}{l}
\mathtt{A}_1 \triangleq \forall i. 0 \le i < 100 \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\mathtt{G}_1 \triangleq \mathsf{True} \\
\\
\mathtt{A}_2 \triangleq \forall i. (0 \le i < 100 \wedge i \ge [t_0]) \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\mathtt{G}_2 \triangleq \forall i. (0 \le i < 100 \wedge i < [t_0]) \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\\
\mathtt{A}_3 \triangleq \forall i. (0 \le i < 100 \wedge i > [t_0]) \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\mathtt{G}_3 \triangleq \forall i. (0 \le i < 100 \wedge i \le [t_0]) \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\\
\mathtt{A}_4 \triangleq \mathsf{True} \\
\mathtt{G}_4 \triangleq \mathtt{A}_1 \\
\\
\mathtt{A} \triangleq \mathit{data}[a_1] = \mathit{data}'[a_1] \\
\mathtt{G} \triangleq \forall i. (0 \le i < 100 \wedge i \ne [a_1]) \\
\qquad \to (\mathit{data}[i] = \mathit{data}'[i]) \\
\\
\mathtt{p} \triangleq 0 \le [t_0] < 100 \wedge [t_1] = 100 \\
\mathtt{p}' \triangleq [t_0] = 100 \\
\mathtt{p}'' \triangleq 0 \le [a_1] < 100
\end{array}
$$

Figure 8. Unbounded thread creation

$$
\Psi_{\text{EX}} \vdash_{\text{CMAP}} \mathbb{C}_{\text{EX}} : \Psi_{\text{EX}}
$$

5. Linking CTL to user programs

The main purpose of CTL is to provide a certified runtime for multithreaded user programs. In Section 3.3, we have already certified a thread library CTL and a multithreaded program. As to build safety multithreaded programs, just providing a certified library is insufficient. In this section, we will complete our work by linking the certified multithreaded user programs with CTL.

As stated in Section 1, we choose OCAP as our common certification framework. OCAP uses an extensible and heterogeneous program specification. OCAP rules are expressive enough to embed most existing verification logic systems for low-level code. We can embed a program logic and prove system specific rules/axioms as lemmas based on an interpretation function and OCAP rules. Thus, the safety program certified in foreign logic systems can be translated down to the OCAP level and then linked with CTL.

5.1. OCAP-light

For simplicity, we present a light-weight OCAP (OCAP-light) framework in Figure 9. OCAP-light uses heterogeneous code specifications θ to support specification languages of both SCAP and CMAP. The code heap specification Ψ is defined as a map from code labels $\mathtt{f}$ to their specifications θ. Note that code labels $\mathtt{f}$ may be mapped to θ_{SCAP} or θ_{CMAP}.

In OCAP-light, the code specifications written in different languages should have interaction to form a cooperative system. Accordingly, the interpretation function $[\![\_]\!]$ is used to translate the specification θ to assertion a which is used at OCAP-light level. Each assertion a is a CiC predicate over Ψ and machine state $\mathbb{S}$. With interpretation functions, the specific inference rules of program logics can be proved as lemmas based on a thin layer of Hoare-style inference rules over meta-logic. Then the soundness of a program logic is reduced to the soundness of OCAP-light.

The soundness and correctness theorem of OCAP-light ensures safety, stated in Section 2. It says that any well-formed program will run forever without reaching any undefined state in Figure 2, and any reachable states satisfy the corresponding assertions in Ψ.

Theorem 1 (OCAP-light - Soundness and Correctness). If $\Psi \vdash (\mathbb{C}, \mathbb{S})$, for all natural number n there exists a $\mathbb{S}' = (\mathbb{R}', \mathbb{H}', \mathsf{pc}')$, such that $(\mathbb{C}, \mathbb{S}) \longmapsto^{n} (\mathbb{C}, \mathbb{S}')$; and if $\mathsf{pc}' \in \Psi$, then $[\![\Psi(\mathtt{f})]\!]\ \Psi\ \mathbb{S}'$.

5.2. Embedding SCAP in OCAP-light.

As stated in Section 3.3, θ_{SCAP} is a pair of predicates $(\mathtt{p}, \mathtt{g})$. The predicate $\mathtt{p}$ is the precondition and the guarantee $\mathtt{g}$ specifies the state at the (function-call) return point and the relationship between the current point and return point. In our TM, the jal instruction is used to perform function

$\boxed{V ::=}$ ***Specification Constructs***

$$
\begin{array}{rccl}
(\textit{CdSpec}) & \theta & ::= & \theta_{\text{SCAP}} \mid \theta_{\text{CMAP}} \\
(\textit{CHSpec}) & \Psi & ::= & \{\texttt{f} \leadsto \theta\}^{*} \\
(\textit{Assert}) & \mathtt{a} & \in & \textit{CHSpec} \rightarrow \textit{State} \rightarrow \textit{Prop} \\
(\textit{Interp}) & \llbracket \_ \rrbracket & \in & \textit{CdSpec} \rightarrow \textit{Assert}
\end{array}
$$

$\boxed{\llbracket \theta_{\text{SCAP}} \rrbracket \triangleq}$ ***SCAP Interpretation***

$$\llbracket (\mathtt{p}, \mathtt{g}) \rrbracket \triangleq \lambda \Psi, \mathbb{S} \,.\, \mathtt{p}\, \mathbb{S} \wedge \exists n \,.\, \mathsf{WFST}(n, \breve{\mathtt{g}}\, \mathbb{S}, \Psi)$$

$$
\begin{array}{l}
\mathsf{WFST}(0, \mathtt{q}, \Psi) \triangleq \forall \mathbb{S}' \,.\, \mathtt{q}\, \mathbb{S}' \\
\qquad\qquad \rightarrow \exists \theta_{\text{CMAP}} \,.\, \Psi(\mathbb{S}'.\mathbb{R}(\texttt{ra})) = \theta_{\text{CMAP}} \wedge \llbracket \theta_{\text{CMAP}} \rrbracket\, \Psi\, \mathbb{S}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{WFST}(n+1, \mathtt{q}, \Psi) \triangleq \forall \mathbb{S}' \,.\, \mathtt{q}\, \mathbb{S}' \\
\qquad\qquad \rightarrow \exists \mathtt{p}', \mathtt{g}' \,.\, \Psi(\mathbb{S}'.\mathbb{R}(\texttt{ra})) = (\mathtt{p}', \mathtt{g}') \\
\qquad\qquad \wedge\, \mathtt{p}'\, \mathbb{S}' \wedge \mathsf{WFST}(n, \mathtt{g}'\, \mathbb{S}', \Psi)
\end{array}
$$

$\boxed{\llbracket \theta_{\text{CMAP}} \rrbracket \triangleq}$ ***CMAP Interpretation***

$$
\begin{array}{l}
\llbracket (\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G}) \rrbracket \triangleq \lambda \Psi, (\mathbb{R}, \mathbb{H}, \mathsf{pc}) \,.\, \exists \mathbb{H}_1, \mathbb{H}_2, \sigma, \mathsf{Q} \,. \\
\qquad\qquad \mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2 \wedge \mathtt{p}\, \tilde{\mathbb{R}}\, \mathbb{H}_1 \\
\qquad\qquad \wedge\, \mathbb{H}_2 \Vdash \mathsf{Core}(\sigma, \mathsf{Tcb}, \mathsf{Q}) \\
\qquad\qquad \wedge\, \mathsf{WFTQ}\, (\Psi, \mathsf{Q}, (\breve{\mathtt{g}}\, \tilde{\mathbb{R}}\, \mathbb{H}_1), \mathtt{A}, \mathtt{G}, \sigma, \tilde{\mathbb{R}})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{WFTQ}\, (\mathsf{Q}, \mathtt{q}, \mathtt{A}, \mathtt{G}, \sigma, \tilde{\mathbb{R}}) \triangleq \\
\quad \forall \sigma', \mathsf{pc}', \tilde{\mathbb{R}}' \,.\, \mathsf{Q}(\sigma') = (\mathsf{pc}', \tilde{\mathbb{R}}', \_) \\
\quad\quad \rightarrow \exists \mathtt{p}', \mathtt{A}', \mathtt{G}' \,.\, \Psi(\mathsf{pc}') = (\mathtt{p}', \mathtt{G}', \mathtt{A}', \mathtt{G}') \\
\quad\quad\quad \wedge\, \mathsf{NI}\, ([(\mathtt{A}, \mathtt{G}, \sigma, \tilde{\mathbb{R}}), \ldots, (\mathtt{A}', \mathtt{G}', \sigma', \tilde{\mathbb{R}}'), \ldots]) \\
\quad\quad\quad \wedge\, (\forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}' \,.\, \mathtt{p}'\, \tilde{\mathbb{R}}\, \mathbb{H} \rightarrow \mathtt{A}'\, \tilde{\mathbb{R}}\, \mathbb{H}\, \mathbb{H}' \rightarrow \mathtt{p}'\, \tilde{\mathbb{R}}\, \mathbb{H}') \\
\quad\quad\quad \wedge\, (\forall \mathbb{H} \,.\, \mathtt{q}\, \mathbb{H} \rightarrow \mathtt{p}'\, \tilde{\mathbb{R}}'\, \mathbb{H})
\end{array}
$$

$\boxed{\Psi \vdash \mathbb{P}}$ ***Well-formed program***

$$\frac{\Psi \vdash \mathbb{C} : \Psi \quad (\mathtt{a}\, \Psi\, \mathbb{S}) \quad \mathbb{C} \vdash \{\mathtt{a}\}\mathsf{pc} : \mathbb{C}(\mathsf{pc})}{\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathsf{pc})} \;(\text{PROG})$$

$\boxed{\Psi \vdash \mathbb{C} : \Psi'}$ ***Well-formed code heap***

$$\frac{\text{for all } \texttt{f} \in \textit{dom}(\Psi')\text{:} \quad \mathtt{a} = \langle \llbracket \Psi'(\texttt{f}) \rrbracket \rangle_{\Psi} \quad \mathbb{C} \vdash \{\mathtt{a}\}\texttt{f} : \mathbb{C}(\texttt{f})}{\Psi \vdash \mathbb{C} : \Psi'} \;(\text{CDHP})$$

$\boxed{\mathbb{C} \vdash \{\mathtt{a}\}\texttt{f} : \iota}$ ***Well-formed instruction sequence***

$$\frac{\begin{array}{l} \mathtt{a} \Rightarrow \lambda \Psi, \mathbb{S} \,.\, \mathbb{S}.\mathsf{pc} = \texttt{f} \\ \qquad\qquad \rightarrow \exists \theta' \,.\, \Psi(\texttt{f}') = \theta' \wedge \llbracket \theta' \rrbracket\, \Psi\, \hat{\mathbb{S}}_{\texttt{jal f}'} \end{array}}{\mathbb{C} \vdash \{\mathtt{a}\}\texttt{f} : \texttt{jal f}'} \;(\text{JAL})$$

$$\frac{\mathtt{a} \Rightarrow \lambda \Psi, \mathbb{S}.\, \exists \theta' \,.\, \Psi(\mathbb{S}.\mathbb{R}(\texttt{r}_s)) = \theta \wedge \llbracket \theta \rrbracket\, \Psi\, \hat{\mathbb{S}}_{\texttt{jr r}_s}}{\mathbb{C} \vdash \{\mathtt{a}\}\texttt{f} : \texttt{jr r}_s} \;(\text{JR})$$

$$\frac{\begin{array}{l} \iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{subu}, \mathsf{subiu}, \mathsf{move}, \mathsf{li}, \mathsf{lw}, \mathsf{sw}\} \\ \mathtt{a} \Rightarrow \lambda \Psi, \mathbb{S} \,.\, (\texttt{f}+1 \in \textit{dom}(\Psi) \\ \qquad \rightarrow \exists \theta' \,.\, \Psi(\texttt{f}+1) = \theta' \wedge \llbracket \theta' \rrbracket\, \Psi\, \hat{\mathbb{S}}_{\iota}) \\ \quad \wedge\, (\texttt{f}+1 \notin \textit{dom}(\Psi) \\ \qquad \rightarrow \exists \mathtt{a}' \,.\, \mathtt{a}'\, \Psi\, \hat{\mathbb{S}}_{\iota} \wedge (\mathbb{C} \vdash \{\mathtt{a}'\}\texttt{f}+1 : \mathbb{C}(\texttt{f}+1))) \end{array}}{\mathbb{C} \vdash \{\mathtt{a}\}\texttt{f} : \iota} \;(\text{SEQ})$$

where

$$
\begin{array}{rcl}
\langle \mathtt{a} \rangle_{\Psi} & \triangleq & \lambda \Psi, \mathbb{S} \,.\, \Psi \subseteq \Psi' \wedge \mathtt{a}\, \Psi'\, \mathbb{S} \\
\mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \lambda \Psi, \mathbb{S} \,.\, \mathtt{a}\, \Psi\, \mathbb{S} \Rightarrow \mathtt{a}'\, \Psi\, \mathbb{S}
\end{array}
$$

Figure 9. OCAP-light

call, while the return action is performed by `jr ra`. The invariant of the abstract control stack is captured by the predicate (WFST), which tells us what is a well-form abstract control stack. Certainly, a safe function call jal won't break the well-formedness of the abstract control stack.

We define the interpretation function of SCAP according to these intuitive ideas in Figure 9. Through the interpretation function, we can build SCAP instruction rules as lemmas in OCAP-light. With these lemmas, safety proof can be constructed directly at OCAP-light level, while still reasoning at the SCAP level. Furthermore, we can prove the soundness of SCAP semantically by this interpretation function.

Theorem 2 (SCAP - Soundness).
If $\Psi \vdash_{\text{SCAP}} \mathbb{C} : \Psi'$, then $\Psi \vdash \mathbb{C} : \Psi'$.

5.3. Embedding CMAP in OCAP-light

Like SCAP, an interpretation function of CMAP is also defined in Figure 9. Through the interpretation function, we know that the whole data heap $\mathbb{H}$ is separated into two parts, $\mathbb{H}_1$ and $\mathbb{H}_2$. $\mathbb{H}_1$ is for user programs, while $\mathbb{H}_2$ is for thread library CTL. $\tilde{\mathbb{R}}$ is a registers file excluding the registers `r0`, `k0`, `k1` and `ra`. The remainder of the interpretation function is a complicated predicate WFTQ. Similar to WFST, WFTQ describes the well-formedness of the thread queue. Upon a well-formed thread queue, the thread library can run the scheduling routine safely. The well-formedness of thread queue is specified by the following invariants:

- each Tcb in the thread queue contains a valid code pointer with code specification $(\mathtt{p}, \breve{\mathtt{g}}, \mathtt{A}, \mathtt{G})$;
- assumptions and guarantees of all the threads are non-interference;
- the precondiction of a waiting thread still holds after any state transitions satisfying the assumption, *i.e.,* $\forall \tilde{\mathbb{R}}, \mathbb{H}, \mathbb{H}' \,.\, \mathtt{p}\, \tilde{\mathbb{R}}\, \mathbb{H} \rightarrow \mathtt{A}\, \tilde{\mathbb{R}}\, \mathbb{H}\, \mathbb{H}' \rightarrow \mathtt{p}\, \tilde{\mathbb{R}}\, \mathbb{H}'$;
- when calling the routines in CTL, the state satisfies preconditions of all the waiting thread.

Next, we prove the soundness theorem of CMAP to complete the embedding process. Firstly, we prove that the programs certified by CMAP logic system call the yield routine of CTL safely. Informally, the safety proof of `ctl_yield()` is divided into two subgoals. One subgoal is to prove that the well-formed state before yielding satisfies the precondition of `ctl_yield()` $\mathtt{p}_y$. The alternative is to prove that the state after the program returns from `ctl_yield()` is still well-formed. The difficulties of proving this subgoal come from the indeterminable transfer of control flow. Fortunately, we could solve these difficulties by knowing that the thread queue satisfies `WFTQ`. The following lemma specifies the safety of `ctl_yield()`.

Lemma 1 (Yielding - Safety).
If $\Psi;\mathbb{C} \vdash_{\text{CMAP}} \{(\mathtt{p},\check{\mathtt{g}},\mathtt{A},\mathtt{G})\}\mathtt{f}:\mathtt{jal}\ \mathtt{yield}$, $\Psi(\mathtt{f}+1) = (\mathtt{p},\mathtt{G},\mathtt{A},\mathtt{G})$ then $\langle[\![\,(\mathtt{p},\check{\mathtt{g}},\mathtt{A},\mathtt{G})\,]\!]\rangle_{\Psi} \Rightarrow \langle[\![\,(\mathtt{p}_y,\mathtt{g}_y)\,]\!]\rangle_{\Psi}$

By the JAL rule presented in Figure 9 and Lemma 1, we can prove the CMAP JAL rule presented in Figure 7 as lemmas at OCAP-light level.

Lemma 2 (Yielding - OCAP-light).
If $\Psi;\mathbb{C} \vdash_{\text{CMAP}} \{\theta_{\text{CMAP}}\}\mathtt{f}:\mathtt{jal}\ \mathtt{yield}$, then:
$\Psi \cup \{\mathtt{yield} \rightsquigarrow (\mathtt{p}_y,\mathtt{g}_y)\};\mathbb{C} \vdash \{[\![\theta_{\text{CMAP}}]\!]\}\mathtt{f}:\mathtt{jal}\ \mathtt{yield}$

Following the same pattern of `ctl_yield()`, we can prove that all the routines are safe to call by the user programs. Then, we can have the soundness theorem of CMAP by the CDHP rule in Figure 9 immediately.

Theorem 3 (CMAP - Soundness).
If $\Psi \vdash_{\text{CMAP}} \mathbb{C}:\Psi$, then $\Psi \cup \Psi_{\text{CTL}} \vdash \mathbb{C}:\Psi$.

Discussion. Benefiting from the underlying OCAP-light, we have bridged the two different program logics, CMAP and SCAP. It is possible to embed other concurrent program logics in OCAP-light with the same method presented in this section. The essential step is to define a corresponding interpretation function, which expresses the global invariants of the program logic in another point of view. In the original paper [7], the soundness of CMAP is proved by PROG and DTHRDS rules, which is similar to our interpretation function of CMAP actually — all of them do the same job checking whether the thread queue is well-formed. The two rule express the global invariants of the CMAP logic.

As observed, the linkage does not increase the cost of proof construction of user program. The safety proof of linkage is hidden from the programmer writing and certifying multithreaded applications.

The thread queue in the original CMAP is abstract and similar to our `Q`, while the interpretation function in our framework interprets the abstract queue `Q` into the concrete one in real TM memory. Between the concrete queue of CTL and the abstract one of the CMAP logic, there is a one-to-one map which makes the bridging possible.

We believe that CTL can be linked with other concurrent certification logics, such as AGL, CSL, *etc.* Because their thread model is similar to ours, they can also be embedded in OCAP-light by the similar techniques we used.

5.4. Example

In Section 3.3, we have proved that CTL is well-formed in SCAP, $\Psi_{\text{CTL}} \vdash_{\text{SCAP}} \mathbb{C}_{\text{CTL}}:\Psi_{\text{CTL}}$. By Theorem 2, we have CTL is well-formed in OCAP-light $\Psi_{\text{CTL}} \vdash \mathbb{C}_{\text{CTL}}:\Psi_{\text{CTL}}$. On the other hand, from Section 4, the *unbounded thread creation* is well-formed in CMAP, $\Psi_{\text{EX}} \vdash_{\text{CMAP}} \mathbb{C}_{\text{EX}}:\Psi_{\text{EX}}$. Hence by Theorem 3, the example is well-formed in OCAP-light $\Psi_{\text{EX}} \cup \Psi_{\text{CTL}} \vdash \mathbb{C}_{\text{EX}}:\Psi_{\text{EX}}$. Finally, CTL and the example are all well-formed at OCAP-light level. By the CDHP rule of OCAP-light, we can link $\mathbb{C}_{\text{EX}}$ with our thread library $\mathbb{C}_{\text{CTL}}$ and have the conclusion that:

$$\Psi_{\text{EX}} \cup \Psi_{\text{CTL}} \vdash \mathbb{C}_{\text{EX}} \cup \mathbb{C}_{\text{EX}}:\Psi_{\text{EX}} \cup \Psi_{\text{CTL}}$$

From this, a complete multithreaded FPCC package can be constructed by the PROG rule of OCAP-light easily.

6. Related work and conclusion

Thread library. Ni *et al.* have certified a thread library named Mth [18], whose aim is quite different from our CTL, they use a machine model that is a strict subset of x86 to ensure that their certified code is runnable on real CPU without any changes, while we concentrate on the simplicity to link our CTL to other certification frameworks with ease, we take an abstract machine model as our platform. But still, our code is MIPS-32 compatible. Mth may be capable of linking to programs certified in other logics, but the linking has not been done yet. The program logic employed by Ni is a variant of XCAP [17], which makes intensive use of general embedded code pointer to support modular reasoning, and results in larger proof size.

FPCC framework. Our work is based on the OCAP framework proposed by Feng *et al.* [6]. In the original OCAP paper, an example was shown to bridge a naive yielding routine to the CCAP logic. Compared to our CTL, the concurrency model of CCAP is rather simple. For example, their model lacks machine context switching and thread management. Our CTL is a big extension to their work, and well illustrates the expressiveness and openness of OCAP.

Chang *et al.* proposed an open verifier for verifying untrusted code [2]. Their framework produces foundational verifiers using untrusted extensions to customize the safety enforcement mechanism. However, it is unclear whether their extensions support concurrent verification.

Concurrency verification. SAGL [5] and concurrent separation logic (CSL) [19] improve the modularity of A-G reasoning method and make the definition of assumptions and guarantees easier. In their machine models, the holding and releasing of locks are primitive operations. So it would be safe to link the programs certified in SAGL or CSL framework with our library just like linking CMAP programs, as long as embedded in OCAP.

Conclusion. We introduce in this paper the design, interfaces and implementation of a certified thread library, which is implemented at assembly level and strictly proved. Dynamic thread management, thread scheduling and synchronization mechanics are also covered, what is more, the modularity of the library endows it with high scalability.

In the open framework OCAP, we show that the thread library can be linked to safe user programs certified in the concurrent certification logic CMAP to form complete multithreaded FPCC packages.

Our long-term goals include the building of a mature thread library with applicative perspectives, as well as the certification of a tiny operating system kernel, whose concurrency model resembles CTL.

Acknowledgments

We would like to thank Zhong Shao and Xinyu Feng for their many inspirational suggestions and comments on ealier drafts of this paper. We would also like to thank Zhaozhong Ni and Hai Fang for help using Coq. Cheng Liu, Yan Guo, Wei Hu and the anonymous reviewers provided many useful suggestions of areas where the paper could be improved.

This work is supported by the National Natural Science Foundation of China under Grant No. 60673126. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of this agency.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] B.-Y. Chang, A. Chlipala, G. Necula, and R. Schneck. The open verifier framework for foundational verifiers. In *Proc. TLDI'05*, pages 1–12, Jan. 2005.

[3] Coq Development Team, INRIA. The Coq proof assistant reference manual. Coq release v8.0, Oct. 2005.

[4] R. S. Engelschall. GNU Pth - the GNU portable threads. `http://www.gnu.org/software/pth/`, 1999-2003.

[5] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. ESOP'07*, pages 173–188, Braga, Portugal, Mar. 2007. Springer-Verlag.

[6] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework to foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, Jan. 2007.

[7] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 International Conference on Functional Programming (ICFP'04)*, September 2005.

[8] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, June 2006.

[9] C. Flanagan and S. Qadeer. Thread modular model checking. In *Proc. of the SPIN Workshop on Software Verification*, 2003.

[10] C. Flanangan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. ESOP'02*, pages 262–277, 2002.

[11] Y. Guo, X. Jiang, Y. Chen, and C. Lin. A certified thread library for multithreaded user programs. `http://ssg.ustcsz.edu.cn/~guoyu/thlib/`, Jan. 2007.

[12] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. LICS'02*, pages 89–100, July 2002.

[13] C. Jones. Tentative steps toward a development method for interfering programs. In *ACM Trans. on Programming Languages and Systems*, pages 596–619, 1983.

[14] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[15] MIPS Technologies, Inc. MIPS32$^{\text{TM}}$ Architecture For Programmers Volume II: The MIPS32$^{\text{TM}}$ Instruction Set, v2.50.

[16] F. Mueller. A library implementation of POSIX threads under unix. In *Proc. of 1993 USEMIX Technical Conf. and Exhib.*, pages 29–41, San Diego, CA, USA, Jan. 1993.

[17] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33nd ACM Symposium on Principles of Programming Languages*, pages 320–333, Jan. 2006.

[18] Z. Ni, D. Yu, and Z. Shao. Technical report for modular verification of machine level thread implementation. `http://flint.cs.yale.edu/publications/mth.html`, Nov. 2006.

[19] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. CONCUR'04*, pages 49–67, 2004.

[20] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, chapter Appendix A: Assemblers, Linkers, and the SPIM Simulator. Morgan Kaufmann, Aug. 2004.

[21] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[22] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, 2002.

[23] S. Xiang, Y. Chen, C. Lin, and L. Li. Molularly certified dynamic storage allocation in scap. In *Proc. 6th International Conference on Quality Software (QSIC'06)*, pages 321–328. IEEE CS press, Oct. 2006.

[24] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[25] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar. 2004.

[26] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. ICFP'04*, pages 175–188, September 2004.