Zha JP, Feng XY, Qiao L. Modular verification of SPARCv8 code. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1382–1405 Nov. 2020. DOI 10.1007/s11390-020-0536-9

Modular Verification of SPARCv8 Code

Jun-Peng Zha[1,2], Xin-Yu Feng[1,2,*], *Member*, *CCF*, *ACM*, and Lei Qiao[3], *Member*, *CCF*

[1] *Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China*
[2] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[3] *Beijing Institute of Control Engineering, Beijing 100080, China*

E-mail: jpzha@smail.nju.edu.cn; xyfeng@nju.edu.cn; fly2moon@aliyun.com

Received April 11, 2020; revised October 24, 2020.

Abstract Inline assembly code is common in system software to interact with the underlying hardware platforms. The safety and correctness of the assembly code is crucial to guarantee the safety of the whole system. In this paper, we propose a practical Hoare-style program logic for verifying SPARC (Scalable Processor Architecture) assembly code. The logic supports modular reasoning about the main features of SPARCv8 ISA (instruction set architecture), including delayed control transfers, delayed writes to special registers, and register windows. It also supports relational reasoning for refinement verification. We have applied it to verify that there is a contextual refinement between a context switch routine in SPARCv8 and a switch primitive. The program logic and its soundness proof have been mechanized in Coq.

Keywords Scalable Processor Architecture Version 8 (SPARCv8), assembly code verification, context switch, Coq, refinement verification

1 Introduction

Operating system kernels are at the most foundational layer of computer software systems. To interact directly with hardware, many important components in OS kernels are implemented in assembly, such as the context switch code or the code that manages interrupts. In addition, there is also code written directly in assembly to achieve high performance (e.g., memcpy in linux v2.6.17.10①). Correctness of such assembly code is crucial to ensure the safety and security of the whole system. However, assembly code verification remains a challenging task in existing work on OS kernel verification (e.g., [1–3]), where the assembly code is either unverified or verified based on operational semantics without a general program logic.

SPARC (Scalable Processor Architecture) is a CPU instruction set architecture (ISA) with high-performance and great flexibility②. It has been widely used in various processors for workstations and embedded systems. SPARCv8 ISA has some interesting features, which make it a non-trivial task to design a Hoare-style program logic for assembly code.

• *Delayed Control Transfers.* SPARCv8 has two program counters `pc` and `npc`. The `npc` register points to the next instruction to run. Control-transfer instructions in SPARCv8 change `npc` instead of `pc` to the target program point, while `pc` takes the original value of `npc`. This makes the control transfer happen one cycle later than the execution of the control transfer instructions.

• *Delayed Writes.* The `wr` instruction that writes a special class of registers such as the window invalid mask register `wim` does not take effect immediately. That is, "they may take until completion of the third instruction following the write instruction to consummate

Regular Paper
Special Section on Software Systems 2020
A preliminary version of the paper was published in the Proceedings of APLAS 2018.
This work was supported by the National Natural Science Foundation of China under Grant No. 61632005.
*Corresponding Author
①Linux v2.6.17.10. https://elixir.bootlin.com/linux/v2.6.17.10/source/arch, Sept. 2020.
②The SPARC Architecture Manual Version 8. https://gaisler.com/doc/sparcv8.pdf, Sept. 2020.

their write operation. The number of delay instructions (0 to 3) is implementation-dependent."

• *Register Windows.* SPARCv8 uses register windows and a window rotation mechanism to avoid saving contexts in the stack directly and achieves high performance in context management.

We use a simple example in Fig.1 to show these three features.

```
CALLER :                          ChangeY :

   ⋮
1  mov   1, %o0                   5  rd   Y, %l0
2  call  ChangeY                  6  wr   %i0, 0, Y
3  save  %sp, −64, %sp            7  nop
4  mov   %o0, %l0                 8  nop
   ⋮                              9  nop
                                  10 ret
                                  11 restore  %l0, 0, %o0
```

(a) (b)

Fig.1. Example for SPARC code [4]. (a) Function CALLER. (b) Function ChangeY.

SPARCv8 has 32 general registers, which are split into four logic groups as global (r_0–r_7), out (r_8–r_{15}), local (r_{16}–r_{23}) and in (r_{24}–r_{31}) registers. Correspondingly, we give aliases "%g_0–%g_7", "%o_0–%o_7", "%l_0–%l_7" and "%i_0–%i_7" for these groups respectively.

CALLER in Fig.1(a) calls ChangeY, which updates the "special register" Y and returns its original value.

ChangeY in Fig.1(b) requires an input parameter as the new value for the special register Y. CALLER calls ChangeY at line 2, and pc and npc point to lines 2 and 3 respectively at this moment. The call instruction changes the value of pc to npc and lets npc point to the entry of ChangeY at line 5, which means the control-flow will not transfer to ChangeY in the next cycle, but in the cycle after the execution of the save instruction following the call. Similarly, when ChangeY returns (at line 10), the control is transferred back to the caller after executing the restore instruction at line 11. We call this feature "delayed control transfers".

SPARCv8 uses the save instruction (at line 3 in the example) to save the current context and the restore instruction (at line 10) to restore it. As explained above, among the 32 general registers, the out, local and in registers form the "current" register window. The local registers are for private use in the current context. The in and out registers are shared with adjacent register windows for parameters passing. The save instruction rotates the register window from the current one to the next. Then the local and in registers in the original window are no longer accessible, and the original out registers become the in registers in the current window. In Fig.1, CALLER uses the save instruction (at line 3) to save its context (local and in registers) and rotate the register window (so that the out registers become the in registers). Thus, the %i_0 register assessed in ChangeY at line 6 is the same register as the %o_0 modified in CALLER at line 1. The restore instruction does the inverse. The arguments taken by the save and restore instructions are irrelevant here and can be ignored.

At line 6, the wr instruction tries to update the special register Y with the value of %$i_0 \oplus 0$ (bitwise exclusive OR). Note that the %i_0 register here is the same register as %o_0 at line 1. However, the write is delayed for X cycles, where X is some predefined system parameter that ranges from 0 to 3. For portability, programmers usually do not rely on the exact value of X and assume it takes the maximum value of 3. Therefore three nop instructions are inserted. Reading of Y earlier than line 9 may give us the old value. This feature is called "delayed writes".

These features make the semantics of the SPARCv8 code context-dependent. For instance, a read of a special register (e.g., the register Y in the above example) needs to make sure that there are enough instructions executed since the most recent "delayed write". As another example, the instruction following the call instruction can be any instruction in general, but it is not supposed to update register r_{15}, which contains the return address saved by the call instruction. In addition, the delayed control transfer and the register windows also allow highly flexible calling conventions. Together, they make it a challenging task to have a Hoare-style program logic for local and modular reasoning of the SPARCv8 assembly code.

Working towards a certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8, we try to address these challenges and propose a practical program logic for realistically modelled SPARCv8 code. We have applied our logic to verify the main body of the task context switch routine in the kernel [4].

However, the OS kernel is implemented as C language mainly and SPARCv8 as inline assembly. Just having a traditional Hoare-style program logic for SPARCv8, which can only make sure the safe execution of the SPARCv8 program if the initial state satisfies the precondition, is insufficient. Xu *et al.* [1] proposed a program logic for verifying the correctness of OS kernel implemented in C language with inline assembly, but they used "abstract assembly primitives" to substitute the inline assembly in their verification work. As a supplement to their work, we extend our program logic so

that it can ensure the contextual refinement relation, shown in (1), between the implementations and their corresponding abstract assembly primitives. Here, we use $\mathbb{C}$, A, and C_{as} to represent the C language program, the set of abstract assembly primitives and the implementations of abstract assembly primitives respectively. It means C_{as} refines A under "any context" $\mathbb{C}$:

$$\forall \mathbb{C}.\, \mathbb{C}[C_{\text{as}}] \subseteq \mathbb{C}[A]. \tag{1}$$

Here we use "$\subseteq$" to represent the refinement relation.

However, if we use the C program as a client code to call inline assembly code, we need to define the semantics of C-assembly interaction.

Since the goal of this paper is to verify the correctness of the assembly code with respect to the abstract assembly primitives, we want to avoid the C-assembly interaction (and leave it as future work). We decompose the OS verification tasks into two steps, as shown in Fig.2, and focus on S2 only in this paper.

$$\begin{array}{ll} & [\![\ \mathbb{C}[A]\ ]\!]^{\mathsf{C}} \\ \text{(S1)}\ \cup\!\mid & \Longleftarrow \quad \boxed{C = \mathit{Comp}(\mathbb{C})} \\ & [\![\ C[\Omega]\ ]\!]^{\mathsf{P\text{-}SPARCv8}} \\ \text{(S2)}\ \cup\!\mid & \Longleftarrow \quad \boxed{\vdash C_{\text{as}} : \Omega} \\ & [\![\ C[C_{\text{as}}]\ ]\!]^{\mathsf{SPARCv8}} \end{array}$$

Fig.2. Idea to establish contextual refinement.

The source program of OS kernel $\mathbb{C}[A]$, which executes under the C language semantics (shown as $[\![\_]\!]^{\mathsf{C}}$), is implemented as C language with a set of abstract assembly primitives A. The compiler (Comp) translates the C program $\mathbb{C}$ to the SPARCv8 code C. As S1 shows, we assume the compilation ensures the refinement between $\mathbb{C}[A]$ and $C[\Omega]$ that executes under Pseudo-SPARCv8 semantics shown as $[\![\_]\!]^{\mathsf{P\text{-}SPARCv8}}$. Here, Ω represents the set of abstract assembly primitives in the middle layer. We use distinguished notations to represent the set of abstract assembly primitives in the source and intermediate level, since they execute on different program states and have different semantics. The Pseudo-SPARCv8 language $C[\Omega]$, which uses SPARCv8 as client code, is able to call abstract assembly primitives in Ω. In S2, we verify using our refinement logic that the whole SPARCv8 program $C[C_{\text{as}}]$ executing under the realistically modelled SPARCv8 semantics, represented as $[\![\_]\!]^{\mathsf{SPARCv8}}$, refines the program $C[\Omega]$ executing under the Pseudo-SPARCv8 semantics. Finally, we can get $[\![C[C_{\text{as}}]]\!]^{\mathsf{SPARCv8}} \subseteq [\![\mathbb{C}[A]]\!]^{\mathsf{C}}$. In this work, we focus on S2, and leave S1 as future work.

Our work is based on earlier work on assembly code verification but makes the following contributions.

• We propose a new program logic which supports relational reasoning for refinement verification. It ensures that a verified SPARCv8 function contextually refines its corresponding abstract assembly primitive. Our logic supports all the above features of SPARCv8. We redefine basic blocks to include the instruction following the jump or return as the tail of a block, which models the delayed control transfer. To reason about delayed writes, we introduce a modal assertion $\triangleright_t \mathtt{sr} \mapsto w$, saying that the special register `sr` will hold the value w in up to t cycles. We also give logic rules for `save` and `restore` instructions that do register window rotation.

• In order to support refinement verification, we define a Pseudo-SPARCv8 language as the language to implement the high-level specification. It also hides the details of sophisticated register window mechanism in SPARCv8 by abstraction, and makes its language model simpler than realistic SPARCv8. Therefore, it can provide some convenience to write the abstract assembly primitive and reason in the Pseudo-SPARCv8 level.

• Following SCAP [5], our logic supports modular reasoning of function calls in a direct style. However, we use the traditional pre- and post-conditions as function specifications, instead of the assertion g used in SCAP. This allows us to reuse existing techniques (e.g., Coq tactics) to simplify the verification process. The logic rules for function call and return are general and independent of any specific calling convention.

• We give direct-style semantic interpretation for the logic judgments, based on which we establish the soundness. This is different from previous work, which either does syntactic-based soundness proof (e.g., SCAP [5]) or treats return code pointers as first-class code pointers and gives CPS-style (continuation-passing style) semantics. Those approaches for soundness make it difficult to verify the interaction between the inline assembly and the C code in the kernel, the latter being verified following a direct-style program logic.

• Context switch of concurrent tasks is an important component in OS kernels. It is usually implemented as inline assembly because of the need to access registers and the stack. We apply our logic and verify that there is a contextual refinement between a context switch routine implemented in SPARCv8 and an abstract switch

primitive.

The program logic and its soundness proof have been mechanized in Coq. Coq proofs and a comparison technical report are available③.

This paper extends our conference paper in APLAS 2018 [4]. The program logic there can only verify the partial correctness of SPARCv8 code and we make the following expansions.

• We propose a new program logic to do relational reasoning for refinement verification (Subsection 4.4 for details).

• In order to support refinement verification, this paper presents a new Pseudo-SPARCv8 language as the high-level specification language (Subsection 4.1 for details).

• We also use the new logic to verify the implementation of a context switch routine, by showing that the implementation contextually refines an abstract switch primitive (Section 5 for details).

In the remainder of the paper, we present the program model and operational semantics of SPARCv8 in Section 2. For the clear presentation, we use a simplified version of our program logic that does not support refinement verification to demonstrate how our logic supports the three main features of SPARCv8 in Section 3, which is the main point of our work and irrelevant to refinement verification. We present the Pseudo-SPARCv8 program and the relational program logic for refinement reasoning in Section 4. We show the verification of a context switch routine in SPARCv8 in Section 5. Finally, we discuss more on related work in Section 6 and conclude the paper in Section 7.

2 SPARCv8 Assembly Language

We introduce the key SPARCv8 instructions, the model of machine states, and the operational semantics in this section.

2.1 Language Syntax and States

The machine model and the syntax of SPARCv8 assembly language are defined in Fig.3. Here, we follow the block-based memory [6] introduced in CompCert to define the memory in our work. The memory address l is defined as a pair of its block ID and the offset. Block IDs (Block) are integers in mathematics presented as $\mathbb{Z}$, and offsets (Word) are 32-bit integers (called "words"). The value (Val) is either word w or address l. The whole program configuration (Prog) P includes the code heap (CodeHeap) C, the machine state (State) S, and the program counters `pc` and `npc`. The code heap C is a partial function from labels `f` to commands c. Labels are also 32-bit integers, which can be viewed as locations where the commands are saved in the code heap. The operand expression (OpExp) `o`, which is either a general register `r` or a word w, and the address expression (AddrExp) `a`, which is either an operand expression or a sum of the values of register `r` and an operation expression, are auxiliary definitions used as parameters of commands. Commands (Comm) in SPARCv8 are classified into two categories: 1) the simple instructions (SimpIns) `i`, which do sequential operation, e.g., arithmetic operation "`add`", or memory operations "`ld`" (load) and "`st`" (store), or register window operations "`save`" and "`restore`", or special register operations "`rd`" (read) and "`wr`" (write), or "`nop`", whose execution changes no program state (the program counters `pc` and `npc`); 2) the control-transfer instructions, e.g., `call` and `retl` for function call and return, or `jmp` and `be` for unconditional and conditional branch.

The machine state (State) S consists of three parts: the memory (Mem) M, the register state (Rstate) Q which is a pair of register file (RegFile) R and frame

(Word) $w, \mathtt{f} \in$ Int32 (Block) $b \in \mathbb{Z}$ (Addr) $l \in$ Block × Word (Val) $v ::= w \mid l$

(Prog)	P	::=	$(C, S, \mathtt{pc}, \mathtt{npc})$	(CodeHeap)	C	∈	Word ⇀ Comm
(State)	S	::=	(M, Q, D)	(RState)	Q	::=	(R, F)
(Mem)	M	∈	Addr ⇀ Val	(ProgCount)	$\mathtt{pc}, \mathtt{npc}$	∈	Word
(OpExp)	`o`	::=	`r` | w	(AddrExp)	`a`	::=	`o` | `r` + `o`

(Comm)	c	::=	`i` | `call f` | `jmp a` | `retl` | `be f`
(SimpIns)	`i`	::=	`ld a` $\mathtt{r}_d$ | `st` $\mathtt{r}_s$ `a` | `nop` | `add` $\mathtt{r}_s$ `o` $\mathtt{r}_d$ | `save` $\mathtt{r}_s$ `o` $\mathtt{r}_d$ | `restore` $\mathtt{r}_s$ `o` $\mathtt{r}_d$
			| `rd sr` $\mathtt{r}_d$ | `wr` $\mathtt{r}_s$ `o sr` | ...
(InstrSeq)	$\mathbb{I}$	::=	`i`; $\mathbb{I}$ | `jmp a`; `i` | `call f`; `i`; $\mathbb{I}$ | `retl i` | `be f`; `i`; $\mathbb{I}$

Fig.3. Machine states and language for SPARCv8 Code [4].

③https://github.com/jpzha/VeriSparc, Sept. 2020.

list (FrameList) F, and the delay buffer (DelayBuff) D. As defined in Fig.4, R is a partial mapping from register names (RegName) to values. Registers include the general registers r, the processor state registers (PsrReg) psr and the special registers (SpeReg) sr. psr contains the integer condition code fields n, z, v and c, which can be modified by the arithmetic and logical instructions and used for conditional control-transfer, and cwp recording the ID of the current register window. We explain the frame list F and the delay buffer D below.

Register Windows and Frame List. SPARCv8 provides 32 general registers that are split into four groups as global (r_0–r_7), out (r_8–r_{15}), local(r_{16}–r_{23}) and in (r_{24}–r_{31}) registers. The latter three groups (out, local and in) form the current register window.

At the entry and exit of functions and traps, one may need to save and restore some of the general registers as execution contexts. Instead of saving them into stacks in memory, SPARCv8 uses multiple register windows to form a circular stack, and does window rotation for efficient context save and restore. As shown in Fig.5 (the figure taken from "The SPARC Architecture Manual Version 8"④), there are N register windows ($N = 8$ here) consisting of $2 \times N$ groups of registers (each group containing eight registers). The cwp register (part of psr) records the ID number of the current window (cwp $= 0$ in this example).

(RegFile)	R	$\in$	RegName $\rightharpoonup$ Val	(RegName)	rn	::=	$r_0 \mid \ldots \mid r_{31} \mid$ psr $\mid$ sr
(PsrReg)	psr	::=	n $\mid$ z $\mid$ v $\mid$ c $\mid$ cwp	(SpeReg)	sr	::=	wim $\mid$ Y $\mid$ $asr_0 \mid \ldots \mid asr_{31}$
(FrameList)	F	::=	nil $\mid$ fm $:: F$	(Frame)	fm	::=	$[v_0, \ldots, v_7]$
(DelayBuff)	D	::=	nil $\mid (t, \mathtt{sr}, w) :: D$	(DelayCycle)	t	$\in$	$\{0, 1, \ldots, X\}$

Fig.4. Register file, frame list and delaybuffer [4].

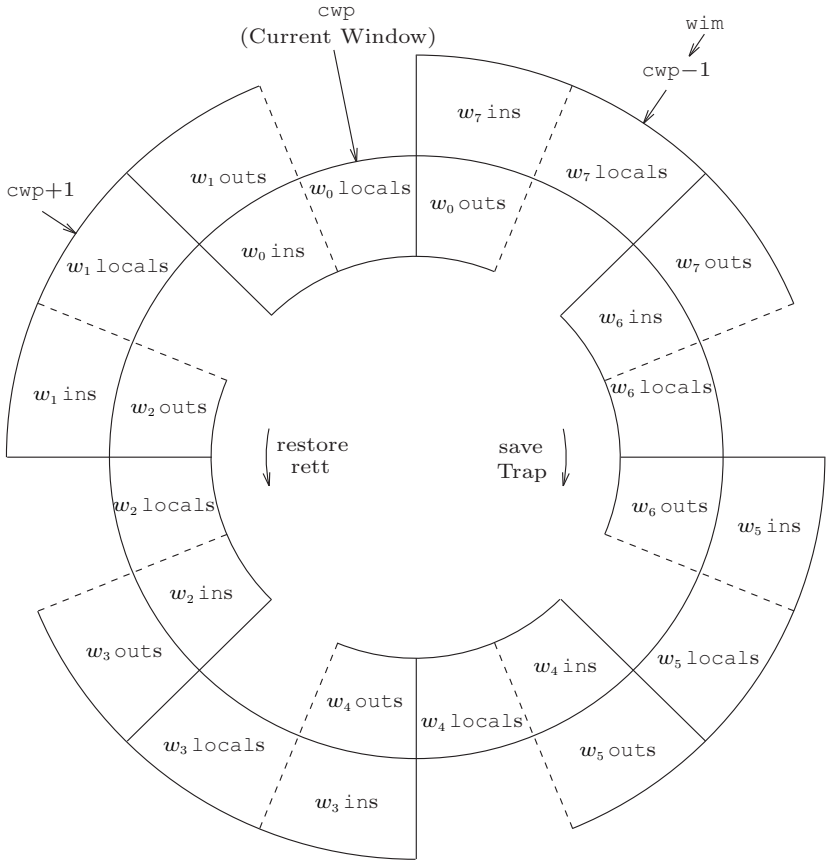


Fig.5. Register windows [4].

④The SPARC Architecture Manual Version 8. https://gaisler.com/doc/sparcv8.pdf, Sept. 2020.

The in and out registers of each window are shared with its adjacent windows for parameter passing. For example, the in registers of w_0 are the out registers of w_1, and the out registers of w_0 are the in registers of w_7. This explains why we need only $2 \times N$ groups of registers for N windows, while each window consists of three groups (out, local and in).

To save the context, the `save` instruction rotates the window by decrementing the `cwp` pointer (modulo N) and w_7 becomes the current window. The out registers of w_0 become the in registers of w_7. The in and local registers of w_0 become inaccessible. This is like pushing them onto the circular stack. The `restore` instruction does the inverse, which is like a stack pop.

The `wim` register is used as a bit vector to record the end of the stack. Each bit in `wim` corresponds to a register window. The bit corresponding to the last available window is set to 1, which means "invalid". All the other bits are 0 (i.e., "valid"). When executing `save` (and `restore`), we need to ensure the next window is valid, in order to avoid the overflow of register window because of the limitation of the number of windows. We use the assertion $\mathsf{win\_valid}(w_{id}, R)$ defined in Fig.6 to say the window pointed to by w_{id} is valid, given the value of `wim` in R.

We use the frame list F to model the circular stack consisting of register windows. As defined in Fig.4, a frame is an array of eight words, modeling a group of eight registers. F consists of a sequence of frames corresponding to all the register windows except the out, local and in registers in the current window. Then `save` saves the local and in registers onto the head of F and loads the two groups of registers at the "tail" of F to the local and out registers (and the original out registers become the in group). The `restore` instruction does the inverse. The operations are defined formally in Fig.6. Here, we use "::" for adding an element at the head of a list, and use "$\cdot$" for appending an element at the tail of a list.

Delay Buffer. The delay buffer D is a sequence of delayed writes. Because the `wr` instruction does not update the target register immediately, we put the write operation onto the delay buffer. A delayed write is recorded as a triple consisting of the remaining cycles t to be delayed, the target special register `sr` and the value w to be written. Note that the value of a special register is restricted to only words, since the special registers are used to record the state of processors, and it is impossible to store memory addresses in them.

Instruction Sequences. We use an instruction sequence $\mathbb{I}$ to model a basic block, i.e., a sequence of commands ending with a control transfer. As defined in Fig.3, we require that a delayed control-transfer instruction must be followed by a simple instruction `i`, because the actual control-transfer occurs after the execution of `i`. The end of each instruction sequence can only be `jmp` or `retl` followed by a simple instruction `i`. Note that we do not view the `call` instruction as

$$\mathsf{out} ::= [\mathtt{r}_8, \ldots, \mathtt{r}_{15}] \qquad \mathsf{local} ::= [\mathtt{r}_{16}, \ldots, \mathtt{r}_{23}] \qquad \mathsf{in} ::= [\mathtt{r}_{24}, \ldots, \mathtt{r}_{31}]$$

$$R([\mathtt{r}_i, \ldots, \mathtt{r}_{i+k}]) ::= [R(\mathtt{r}_i), \ldots, R(\mathtt{r}_{i+k})]$$

$$R\{[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} ::= R\{\mathtt{r}_i \rightsquigarrow v_0\} \ldots \{\mathtt{r}_{i+7} \rightsquigarrow v_7\} \quad \text{where} \quad \mathrm{fm} = [v_0, \ldots, v_7]$$

$$\mathsf{win\_valid}(w_{id}, R) ::= 2^{w_{id}} \,\&\, R(\mathtt{wim}) = 0$$
where & is the bitwise AND operation.

$$\mathsf{next\_cwp}(w_{id}) ::= (w_{id} + N - 1)\%N \qquad \mathsf{prev\_cwp}(w_{id}) ::= (w_{id} + 1)\%N$$

$$\mathsf{save}(R, F) ::= \begin{cases} (R', F'), & \text{if } w'_{id} = \mathsf{next\_cwp}(R(\mathtt{cwp})), \mathsf{win\_valid}(w'_{id}, R), \\ & \quad F = F'' \cdot \mathrm{fm}_1 \cdot \mathrm{fm}_2,\ F' = R(\mathsf{local}) :: R(\mathsf{in}) :: F'', \\ & \quad R'' = R\{\mathsf{in} \rightsquigarrow R(\mathsf{out}), \mathsf{local} \rightsquigarrow \mathrm{fm}_2, \mathsf{out} \rightsquigarrow \mathrm{fm}_1\}, \\ & \quad R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot, & \text{if } \neg\mathsf{win\_valid}(\mathsf{next\_cwp}(R(\mathtt{cwp})), R). \end{cases}$$

$$\mathsf{restore}(R, F) ::= \begin{cases} (R', F'), & \text{if } w'_{id} = \mathsf{prev\_cwp}(R(\mathtt{cwp})), \mathsf{win\_valid}(w'_{id}, R), \\ & \quad F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'',\ F' = F'' \cdot R(\mathsf{out}) \cdot R(\mathsf{local}), \\ & \quad R'' = R\{\mathsf{in} \rightsquigarrow \mathrm{fm}_2, \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{out} \rightsquigarrow R(\mathsf{in})\}, \\ & \quad R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot, & \text{if } \neg\mathsf{win\_valid}(\mathsf{prev\_cwp}(R(\mathtt{cwp})), R). \end{cases}$$

Fig.6. Auxiliary definitions for instruction `save` and `restore` [4].

the end of a basic block, since the callee is expected to return, following our direct-style semantics for function calls. We define $C[\mathtt{f}]$ to extract an instruction sequence starting from $\mathtt{f}$ in C below.

$$C[\mathtt{f}] = \begin{cases} \mathtt{i};\mathbb{I}, & \text{if } C(\mathtt{f}) = \mathtt{i} \text{ and } C[\mathtt{f}+4] = \mathbb{I}, \\ c;\mathtt{i}, & \text{if } c = C(\mathtt{f}) \text{ and } c = \mathtt{jmp\ a} \text{ or } \mathtt{retl} \\ & \quad \text{and } C(\mathtt{f}+4) = \mathtt{i}, \\ c;\mathtt{i};\mathbb{I}, & \text{if } c = C(\mathtt{f}) \text{ and } c = \mathtt{call\ f} \text{ or } \mathtt{be\ f} \\ & \quad \text{and } C(\mathtt{f}+4) = \mathtt{i} \text{ and } C[\mathtt{f}+8] = \mathbb{I}, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

2.2 Operational Semantics

The operational semantics is taken from Wang *et al.*[7], but we use a block-based memory model and omit features like interrupts and traps. We show the selected rules in Fig.7. The program transition relation $(C, S, \mathtt{pc}, \mathtt{npc}) ::\Rightarrow (C, S', \mathtt{pc}', \mathtt{npc}')$ is defined in Fig.7(a). Before the execution of the instruction pointed by $\mathtt{pc}$, the delayed writes in D with 0 delay cycles are executed first. The execution of the delayed writes is defined in the form of $(R, D) \rightrightarrows (R', D')$ below.

$$\frac{(R, D) \rightrightarrows (R', D') \quad C \vdash ((M, (R', F), D'), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}{(C, (M, (R, F), D), \mathtt{pc}, \mathtt{npc}) ::\Longrightarrow (C, (M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}$$

(a)

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad (M, (R, F), D) \overset{\mathtt{i}}{\bullet\!\!\longrightarrow} (M', (R', F'), D')}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M', (R', F'), D'), \mathtt{npc}, \mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{jmp\ a} \qquad [\![\mathtt{a}]\!]_R = \mathtt{f}}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathtt{npc}, \mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \qquad \mathtt{r}_{15} \in \mathrm{dom}(R)}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\}, F), D), \mathrm{npc}, \mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \quad R(\mathtt{r}_{15}) = \mathtt{f}}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathtt{npc}, \mathtt{f}+8)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{be\ f} \quad R(\mathtt{z}) \neq 0}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathtt{npc}, \mathtt{f})} \qquad \frac{C(\mathtt{pc}) = \mathtt{be\ f} \quad R(\mathtt{z}) = 0}{C \vdash ((M, (R, F), D), \mathtt{pc}, \mathtt{npc}) \circ\!\!\longrightarrow ((M, (R, F), D), \mathtt{npc}, \mathtt{npc}+4)}$$

(b)

$$\frac{(M, R) \xrightarrow{\mathtt{i}} (M', R')}{(M, (R, F), D) \overset{\mathtt{i}}{\bullet\!\!\longrightarrow} (M', (R', F), D)} \qquad \frac{R(\mathtt{r}_s) = w_1 \quad [\![\mathtt{o}]\!]_R = w_2 \quad w = w_1 \oplus w_2 \quad \mathtt{sr} \in \mathrm{dom}(R) \quad D' = \mathsf{set\_delay}(\mathtt{sr}, w, D)}{(M, (R, F), D) \overset{\mathtt{wr\ r}_s\ \mathtt{o\ sr}}{\bullet\!\!\longrightarrow} (M, (R, F), D')}$$

$$\frac{\mathsf{save}(R, F) = (R', F') \quad [\![\mathtt{o}]\!]_R = v \quad v' = R(\mathtt{r}_s) + v}{(M, (R, F), D) \overset{\mathtt{save\ r}_s\ \mathtt{o\ r}_d}{\bullet\!\!\longrightarrow} (M, (R\{\mathtt{r}_d \rightsquigarrow v'\}, F'), D)}$$

$$\frac{\mathsf{restore}(R, F) = (R', F') \quad [\![\mathtt{o}]\!]_R = v \quad v' = R(\mathtt{r}_s) + v}{(M, (R, F), D) \overset{\mathtt{restore\ r}_s\ \mathtt{o\ r}_d}{\bullet\!\!\longrightarrow} (M, (R\{\mathtt{r}_s \rightsquigarrow v'\}, F'), D)}$$

(c)

$$\frac{R(\mathtt{sr}) = w \qquad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\mathtt{rd\ sr\ r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow w\})} \qquad \frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad v = v_1 + v_2 \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\mathtt{add\ r}_s\ \mathtt{o\ r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow v\})}$$

$$\frac{[\![\mathtt{a}]\!]_R = l \quad M(l) = v' \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M, R) \xrightarrow{\mathtt{ld\ a\ r}_d} (M, R\{\mathtt{r}_d \rightsquigarrow v'\})}$$

(d)

$$[\![\mathtt{o}]\!]_R ::= \begin{cases} R(r), & \text{if } \mathtt{o} = r, \\ w, & \text{if } \mathtt{o} = w, \\ & \quad -4\,096 \leqslant w \leqslant 4\,095, \\ \text{undefined}, & \text{otherwise.} \end{cases} \qquad [\![\mathtt{a}]\!]_R ::= \begin{cases} [\![\mathtt{o}]\!]_R, & \text{if } \mathtt{a} = \mathtt{o}, \\ v_1 + v_2, & \text{if } \mathtt{a} = \mathtt{r} + \mathtt{o},\ R(\mathtt{r}) = v_1 \\ & \quad \text{and } [\![\mathtt{o}]\!]_R = v_2, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

(e)

Fig.7. Selected operational semantics rules[4]. (a) Program transition. (b) Control transfer instruction transitions. (c) `save`, `restore` and `wr` instruction transitions. (d) Simple instruction transitions. (e) Expression semantics.

$$\frac{}{(R, \mathrm{nil}) \rightrightarrows (R, \mathrm{nil})}$$

$$\frac{(R, D) \rightrightarrows (R', D')}{(R, (t{+}1, \mathtt{sr}, w) :: D) \rightrightarrows (R', (t, \mathtt{sr}, w) :: D')}$$

$$\frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \in \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R'\{\mathtt{sr} \rightsquigarrow w\}, D')}$$

$$\frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \notin \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R', D')}$$

Note that the write of `sr` has no effect if `sr` is not in the domain of R. Since R is defined as a partial map, we can prove the following lemma.

Lemma 1 [4]. *$(R, D) \rightrightarrows (R', D')$ and $R = R_1 \uplus R_2$, if and only if there exist R'_1 and R'_2, such that $(R_1, D) \rightrightarrows (R'_1, D')$, $(R_2, D) \rightrightarrows (R'_2, D')$, and $R' = R'_1 \uplus R'_2$.*

Here the disjoint union $R_1 \uplus R_2$ represents the union of R_1 and R_2 if they have disjoint domains, and undefined otherwise. This lemma is important to give sound semantics to delay buffer related assertions, as discussed in Section 3.

The transition steps for individual instructions are classified into three categories: the control transfer steps ($\_ \vdash \_ \circ\!\!\longrightarrow \_$), the steps for `save`, `restore` and `wr` instructions ($\_ \bullet\!\!\xrightarrow{-}\!\!\_$), and the steps for other simple instructions ($\_ \xrightarrow{-} \_$). The corresponding step transition relations are defined inductively in Fig.7(b)–Fig.7(d) respectively.

Note that, after the control-transfer instructions, `pc` is set to `npc` and `npc` contains the target code pointer. This explains the one cycle delay for the control transfer. The `call` instruction saves `pc` into register $\mathtt{r}_{15}$, while `retl` uses $\mathtt{r}_{15}+8$ as the return address (which is the address for the second instruction following the `call`). The conditional branch `be f` jumps to `f` (after one-cycle delay) if the value in the register `z` is not 0. Evaluation of expressions `a` and `o` are defined in Fig.7(e). Here, we define the sum of two values v_1 and v_2 below. The result of $v_1{+}v_2$ is legal, if both of v_1 and v_2 are words (Int32), or v_1 is an address and v_2 is an offset. The offset is a word, which acts as an immediate value in the calculation of address.

$$v_1{+}v_2 \ ::= \begin{cases} w_1 + w_2, & \text{if } v_1 = w_1, \text{ and } v_2 = w_2, \\ (b, w_1 + w_2), & \text{if } v_1 = (b, w_1), \text{ and } v_2 = w_2, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

`wr` wants to save the bitwise exclusive OR of the operands into the special register `sr`, but it puts the write into the delay buffer D instead of updating R immediately. The operation $\mathsf{set\_delay}(\mathtt{sr}, w, D)$ is defined below:

$$\mathsf{set\_delay}(\mathtt{sr}, w, D) ::= (X, \mathtt{sr}, w) :: D,$$

where X $(0 \leqslant X \leqslant 3)$ is a predefined system parameter for the delay cycle.

The `save` and `restore` instructions rotate the register windows and update the register file. Their operations over F and R are defined in Fig.6.

3 Program Logic

In this section, we use a simplified version of our program logic that does not support refinement verification to present how our logic handles the features of SPARCv8. The relational program logic for refinement reasoning will be introduced in Section 4.

3.1 Assertions

We define the syntax of assertions (Asrt) in Fig.8, and their semantics in Fig.9. We extend separation logic assertions with specifications of delay buffers and register windows. Registers are like variables in separation logic, but treated as resources. The assertion `emp` says that the memory and the register file are both empty. $l \mapsto v$ specifies a singleton memory cell with value v stored in address l. $\mathtt{rn} \mapsto v$ says that `rn` is the only register in the register file and it contains the value v. Also `rn` is not in the delay buffer. Separating conjunction $p * q$ has the standard semantics as in separation logic [8].

$$\begin{aligned} (\text{Asrt})\ p, q \ ::= \ & \mathsf{emp} \mid l \mapsto v \mid \mathtt{rn} \mapsto v \mid \rhd_t \mathtt{sr} \mapsto w \mid p\!\downarrow \\ & \mid \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) \mid p \wedge q \mid p \vee q \mid p * q \\ & \mid \mathtt{a} =_a v \mid \mathtt{o} = v \mid \forall x.\, p \mid \exists x.\, q \mid \ldots \end{aligned}$$

Fig.8. Syntax of assertions [4].

The assertion $\rhd_t \mathtt{sr} \mapsto w$ describes a delayed write in the delay buffer D. It describes the uncertainty of `sr`'s value in R, which is unknown for now but will become w in up to $t{+}1$ cycles. We use $\_ \rightrightarrows^k \_$ to represent the k-step execution of the delayed writes in D. It also requires that there be at most one delayed write for a specific special register `sr` in D (i.e., $\mathsf{noDup}(\mathtt{sr}, D)$). This prevents more than one delayed write to the same register within four instruction cycles, which practically have no restrictions on programming. By the semantics

$$
\begin{array}{lll}
S \models \mathsf{emp} & ::= & S.M = \emptyset \wedge S.Q.R = \emptyset \\
S \models l \mapsto v & ::= & S.M = \{l \rightsquigarrow v\} \wedge S.Q.R = \emptyset \\
S \models \mathtt{rn} \mapsto v & ::= & S.Q.R = \{\mathtt{rn} \rightsquigarrow v\} \wedge \mathtt{rn} \notin \mathrm{dom}(S.D) \wedge S.M = \emptyset \\
S \models \triangleright_t \mathtt{sr} \mapsto w & ::= & \exists k, R', D'.\, 0 \leqslant k \leqslant t+1 \wedge (R, D) \rightrightarrows^k (R', D') \wedge \\
 & & ((M, (R', F), D') \models \mathtt{sr} \mapsto w) \wedge \mathsf{noDup}(D, \mathtt{sr}) \\
 & & \text{where } S = (M, (R, F), D) \\
S \models p\downarrow & ::= & \exists R', D'.\ ((M, (R', F), D') \models p) \wedge (R', D') \rightrightarrows (R, D) \\
 & & \text{where } S = (M, (R, F), D) \\
S \models \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) & ::= & (S \models \mathtt{cwp} \mapsto w_{id}) \wedge \exists F'.\, F \cdot F' = S.Q.F \\
S \models \mathtt{a} =_a v & ::= & [\![\mathtt{a}]\!]_{S.Q.R} = v \wedge \mathsf{word\_align}(v) \\
S \models \mathtt{o} = v & ::= & [\![\mathtt{o}]\!]_{S.Q.R} = v \\
S \models p_1 * p_2 & ::= & \exists S_1, S_2.\, S_1 \models p_1 \wedge S_2 \models p_2 \wedge S = S_1 \uplus S_2
\end{array}
$$

$$
M_1 \perp M_2 \ ::= (\mathrm{dom}(M_1) \cap \mathrm{dom}(M_2)) = \emptyset \qquad R_1 \perp R_2 \ ::= (\mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)) = \emptyset
$$

$$
S_1 \uplus S_2 \ ::= \begin{cases} (M_1 \cup M_2, (R_1 \cup R_2, F), D), & \text{if } M_1 \perp M_2 \wedge R_1 \perp R_2 \wedge \\ & \quad S_1 = (M_1, (R_1, F), D) \wedge S_2 = (M_2, (R_2, F), D), \\ \text{undefined}, & \text{otherwise.} \end{cases}
$$

$$
\mathrm{dom}(D) \ ::= \begin{cases} \{\mathtt{sr}\} \cup \mathrm{dom}(D'), & \text{if } D = (t, \mathtt{sr}, w) :: D', \\ \emptyset, & \text{if } D = \text{nil}. \end{cases}
$$

$$
\mathsf{noDup}(D, \mathtt{sr}) \ ::= \begin{cases} \mathtt{sr} \notin \mathrm{dom}(D'), & \text{if } D = (t, \mathtt{sr}, w) :: D', \\ \mathtt{sr} \neq \mathtt{sr}' \wedge \mathsf{noDup}(D', \mathtt{sr}), & \text{if } D = (t, \mathtt{sr}', w) :: D', \\ \text{True}, & \text{if } D = \text{nil}. \end{cases}
$$

Fig.9. Semantics of assertions [4].

we have ($p \Rightarrow q$ means for any S, if $S \models p$ holds, then $S \models q$ holds):

$$\mathtt{sr} \mapsto w \ \Rightarrow \ \triangleright_t \mathtt{sr} \mapsto w,$$
$$\triangleright_t \mathtt{sr} \mapsto w \ \Rightarrow \ \triangleright_{t+k} \mathtt{sr} \mapsto w.$$

The assertion $p\downarrow$ allows us to reduce the uncertainty by executing one step of the delayed writes. It specifies states reachable after executing one step of delayed writes from those states satisfying p. Therefore we know:

$$(\triangleright_0 \mathtt{sr} \mapsto w)\downarrow \Rightarrow \ \mathtt{sr} \mapsto w,$$
$$(\triangleright_{t+1} \mathtt{sr} \mapsto w)\downarrow \Rightarrow \ \triangleright_t \mathtt{sr} \mapsto w.$$

Also it is easy to see that if p syntactically does not contain sub-terms in the form of $\triangleright_t \mathtt{sr} \mapsto w$, then $(p\downarrow) \Leftrightarrow p$.

The following lemma shows $(\_)\downarrow$ is distributive over separating conjunction.

Lemma 2 [4]. $(p * q)\downarrow \Leftrightarrow (p\downarrow) * (q\downarrow)$.

The lemma can be proved following Lemma 1. We present the proof of Lemma 2 below.

Proof. "$\Rightarrow$": if $(M, (R, F), D) \models (p * q)\downarrow$, then $(M, (R, F), D) \models (p)\downarrow * (q)\downarrow$. We first get there exist R', D', M_1, M_2, R'_1 and R'_2 such that ($R_1 \uplus R_2$ represents the union of R_1 and R_2 if they have disjoint domains, and $M_1 \uplus M_2$ has the same meaning): (A1) $(R', D') \rightrightarrows (R, D)$; (A2) $R' = R'_1 \uplus R'_2$; (A3) $M = M_1 \uplus M_2$; (A4) $(M_1, (R'_1, F), D') \models p$; (A5) $(M_2, (R'_2, F), D') \models q$. By applying Lemma 1 on (A.1), we get that there exist R_1 and R_2, where $R = R_1 \uplus R_2$, such that: $(R'_1, D') \rightrightarrows (R_1, D)$, $(R'_2, D') \rightrightarrows (R_2, D)$.

Thus, we get $(M_1, (R_1, F), D) \models p\downarrow$; $(M_2, (R_2, F), D) \models q\downarrow$. Finally, we prove that $(M, (R, F), D) \models (p\downarrow) * (q\downarrow)$ holds.

"$\Leftarrow$": if $(M, (R, F), D) \models (p\downarrow) * (q\downarrow)$, then $(M, (R, F), D) \models (p * q)\downarrow$. We first get there exist M_1, M_2, R'_1, R'_2, R_1, R_2, D'_1 and D'_2, where $R = R_1 \uplus R_2$, such that: (B1) $(R'_1, D'_1) \rightrightarrows (R_1, D)$, $(R'_2, D'_2) \rightrightarrows (R_2, D)$; (B2) $(M_1, (R'_1, F), D'_1) \models p$; (B3) $(M_2, (R'_2, F), D'_2) \models q$.

By the definition of the step of the delayed writes, we can prove that $D'_1 = D'_2$. Let $D' = D'_1 = D'_2$. By applying Lemma 1 on (B1), we get there exists R', where $R' = R'_1 \uplus R'_2$, such that $(R', D') \rightrightarrows (R, D)$. Thus, we prove that $(M, (R, F), D) \models (p * q)\downarrow$ holds. □

We use $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ to describe the pointer cwp of the current register window and the frame list as a circular stack. Note that F is just a prefix of the frame list, since usually we do not need to know contents of the full list. Here we use $F \cdot F'$ to represent the concatenation of lists F and F'. Therefore we have $\mathtt{cwp} \mapsto (\!| w_{id}, F \cdot F' |\!) \Rightarrow \mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$.

The assertions $\mathtt{a} =_a v$ and $\mathtt{o} = v$ describe the value of a and o respectively. They are intuitionistic assertions. Since a is used as an address, we also require it to be properly aligned on a 4-byte boundary. We define word_align to represent this restriction below:

$$\mathsf{word\_align}(v) ::= \exists w.\, (v = w \vee v = (\_, w)) \wedge w\%4 = 0.$$

The result of the address expression a may be a word, if it is a pointer in code heap, or a memory address if

it is a location of memory.

3.2 Inference Rules

The code specification θ and the code heap specification Ψ are defined below.

$$(\text{valList})\ \ \iota \in \text{list value} \quad (\text{pAsrt})\ \ \text{fp}, \text{fq} \in \text{valList} \to \text{Asrt}$$
$$(\text{CdSpec})\ \ \theta ::= (\text{fp}, \text{fq}) \quad (\text{CdHpSpec})\ \ \Psi ::= \{\mathtt{f} \rightsquigarrow \theta\}^*$$

The code heap specification Ψ maps the code labels for basic blocks to their specifications θ, which is a pair of pre- and post-conditions. Instead of using normal assertions, the pre- and post-conditions are assertions parameterized over a list of values ι. They play the role of auxiliary variables—feeding the pre- and the post-conditions with the same ι allows us to establish the relationship of states specified in the pre- and post-conditions.

Although we assign a specification θ to each basic block, the post-condition does not specify the states reached at the end of the block. Instead, it specifies the condition that needs to be specified in the future when the "current function" returns. This follows the idea developed in SCAP [5], but we use the standard unary state assertion instead of the binary state assertions used in SCAP so that existing proof techniques (such as Coq tactics) for standard Hoare-triples can be applied to simplify the verification process.

We give a simple example in Fig.10 to show a specification for a function, which simply sums the values of the registers $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$ and writes the result into register $\%\mathtt{l}_7$. The specification (fp, fq) says that, when provided with the same lv as argument, the function preserves the value of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, $\%\mathtt{l}_7$ in the beginning contains any value and at the end contains the sum of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, and the function also preserves the value of $\mathtt{r}_{15}$, which it uses as the return address. To verify the function, we need to prove that it satisfies $(\text{fp}\, lv, \text{fq}\, lv)$ for all lv. Here, $lv[1]$ and $lv[2]$ cannot be a memory address, because a value plus a memory address is illegal. $lv[3]$ should also be a word, because it is a return code pointer whose type is word.

Fig.11 shows selected inference rules in our logic. Our logic divides the proof work into three layers. We define the well-formed code heap in the form of $(\vdash C : \Psi)$ to verify the code heap C, the well-formed instruction sequence in the form of $(\Psi \vdash \{(p, q)\}\ \mathtt{f} : \mathbb{I})$ to verify the instruction sequence $\mathbb{I}$ starting from `f` in code heap, and well-formed instruction in the form of $(\vdash \{p\}\, \mathtt{i}\, \{q\})$ to verify the single simple instruction `i`.

– $\{(\text{fp}, \text{fq})\}$
```
add  %i0, %i1, %l7
add  %l7, %i2, %l7
retl
nop
```

$$\begin{aligned}\text{fp} ::= \lambda\, lv.\ & ((\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\ & * \%\mathtt{l}_7 \mapsto \_ * (\mathtt{r}_{15} \mapsto lv[3])) \\ & \wedge (lv[1], lv[2], lv[3] \in \text{Word}) \\ \text{fq} ::= \lambda\, lv.\ & (\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\ & * (\%\mathtt{l}_7 \mapsto lv[0] + lv[1] + lv[2]) * (\mathtt{r}_{15} \mapsto lv[3])\end{aligned}$$

Fig.10. Example for function specification [4].

The top rule CDHP verifies the code heap C. It requires that every basic block specified in Ψ can be verified with respect to the specification, with any argument ι used to instantiate the pre- and post-conditions.

The SEQ rule is applied when meeting an instruction sequence starting with a simple instruction `i`. The instruction `i` is verified by the corresponding well-formed instruction rules, with the precondition $p\!\downarrow$ and some post-condition p'. We use $p\!\downarrow$ because there is an implicit step executing delayed writes before executing every instruction. The post-condition p' for `i` is then used as the precondition to verify the remaining part of the instruction sequence.

Delayed Control Transfers. We distinguish the `jmp` and `call` instructions — the former makes an "intra-function" control transfer, while the latter makes function calls. The JMP rule requires that the target address is a valid one specified in Ψ. Starting from the precondition p, after executing the instruction `i` following JMP and the corresponding delayed writes, the post-condition p' of `i` should satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Since the target instruction sequence of `jmp` is in the same function as the `jmp` instruction itself, the post-condition fq specified at the target address (with the same instantiation ι of the logical variables and the frame assertion p_r) should meet the post-condition q of the current function. As we explained before, the post-condition q does not specify the states reached at the end of the instruction sequence (which are specified by p' instead).

The CALL rule is similar to the JMP rule in that it also requires the post-condition p_2 of the instruction `i` following the `call` satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Here we need to record that the code label `f` is saved in $\mathtt{r}_{15}$ by the `call` instruction. When the callee returns, its post-condition fq (with the same instantiation of auxiliary

$\boxed{\vdash C : \Psi}$ (Well-Formed Code Heap)

$$\frac{\text{for all } \mathtt{f} \in \mathrm{dom}(\Psi),\ \iota : \Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(\mathrm{fp}\,\iota, \mathrm{fq}\,\iota)\}\ \mathtt{f} : C[\mathtt{f}]}{\vdash C : \Psi}\ (\mathsf{CDHP})$$

$\boxed{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}}$ (Well-Formed Instruction Sequences)

$$\frac{\vdash \{p\!\downarrow\}\,\mathtt{i}\,\{p'\} \quad \Psi \vdash \{(p', q)\}\ \mathtt{f}{+}4 : \mathbb{I}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{i}; \mathbb{I}}\ (\mathsf{SEQ})$$

$$\frac{\begin{array}{c} p\!\downarrow \Rightarrow (\mathtt{a} =_a \mathtt{f}') \quad \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \\ \vdash \{p\!\downarrow\!\downarrow\}\,\mathtt{i}\,\{p'\} \quad \exists \iota, p_r.\ (p' \Rightarrow \mathrm{fp}\,\iota * p_r) \wedge (\mathrm{fq}\,\iota * p_r \Rightarrow q) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{jmp\ a}; \mathtt{i}}\ (\mathsf{JMP})$$

$$\frac{\begin{array}{c} \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(p', q)\}\ \mathtt{f}{+}8 : \mathbb{I} \\ p\!\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \_) * p_1 \quad \vdash \{(\mathtt{r}_{15} \mapsto \mathtt{f} * p_1)\!\downarrow\}\,\mathtt{i}\,\{p_2\} \\ \exists \iota, p_r.\ (p_2 \Rightarrow \mathrm{fp}\,\iota * p_r) \wedge (\mathrm{fq}\,\iota * p_r \Rightarrow p') \wedge (\mathrm{fq}\,\iota \Rightarrow \mathtt{r}_{15} = \mathtt{f}) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{call\ f}'; \mathtt{i}; \mathbb{I}}\ (\mathsf{CALL})$$

$$\frac{p\!\downarrow\!\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \mathtt{f}') * p_1 \quad \vdash \{p_1\}\,\mathtt{i}\,\{p_2\} \quad (\mathtt{r}_{15} \mapsto \mathtt{f}') * p_2 \Rightarrow q}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{retl}; \mathtt{i}}\ (\mathsf{RETL})$$

$$\frac{\begin{array}{c} p\!\downarrow \Rightarrow (\mathtt{z} \mapsto w) * \mathrm{true} \quad \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \\ \vdash \{p\!\downarrow\!\downarrow\}\,\mathtt{i}\,\{p'\} \quad \Psi \vdash \{(p' \wedge w = 0, q)\}\ \mathtt{f}{+}8 : \mathbb{I} \\ \exists \iota, p_r.\ ((p' \wedge w \neq 0) \Rightarrow \mathrm{fp}\,\iota * p_r) \wedge (\mathrm{fq}\,\iota * p_r \Rightarrow q) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathtt{be\,f}'; \mathtt{i}; \mathbb{I}}\ (\mathsf{BE})$$

$\boxed{\vdash \{p\}\,\mathtt{i}\,\{q\}}$ (Well-Formed Instructions)

$$\frac{\mathtt{sr} \mapsto \_ * p \Rightarrow (\mathtt{r}_s = w_1 \wedge \mathtt{o} = w_2)}{\vdash \{\mathtt{sr} \mapsto \_ * p\}\,\mathtt{wr\ r}_s\ \mathtt{o\ sr}\,\{(\triangleright_3 \mathtt{sr} \mapsto (w_1 \oplus w_2)) * p\}}\ (\mathsf{WR})$$

$$\frac{}{\vdash \{\mathtt{sr} \mapsto w * \mathtt{r}_d \mapsto \_\}\,\mathtt{rd\ sr\ r}_d\,\{\mathtt{sr} \mapsto w * \mathtt{r}_d \mapsto w\}}\ (\mathsf{RD})$$

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \quad w'_{id} = \mathsf{next\_cwp}(w_{id}) \quad w\,\&\,2^{w'_{id}} = 0 \quad v = v_1 + v_2 \\ p \Rightarrow (\mathtt{cwp} \mapsto (\!|\, w_{id}, F \cdot \_ \cdot \_ \,|\!)) * (\mathsf{out} \mapsto \mathrm{fm}_o) * (\mathsf{local} \mapsto \mathrm{fm}_l) * (\mathsf{in} \mapsto \mathrm{fm}_i) * p_1 \end{array}}{\begin{array}{c} \big(\mathtt{cwp} \mapsto (\!|\, w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F \,|\!)\big) * (\mathsf{out} \mapsto \_) * (\mathsf{local} \mapsto \_) * (\mathsf{in} \mapsto \mathrm{fm}_o) * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2 \\ \vdash \{(\mathtt{wim} \mapsto w) * p\}\,\mathtt{save\ r}_s\ \mathtt{o\ r}_d\,\{(\mathtt{wim} \mapsto w) * (\mathtt{r}_d \mapsto v) * p_2\} \end{array}}\ (\mathsf{SAVE})$$

where $[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \mapsto [w_0, \ldots, w_7] \ ::= \mathtt{r}_i \mapsto w_0 * \cdots * \mathtt{r}_{i+7} \mapsto w_7$

and out, local and in are defined in Fig.6.

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \quad w'_{id} = \mathsf{prev\_cwp}(w_{id}) \quad w\,\&\,2^{w'_{id}} = 0 \quad v = v_1 + v_2 \\ p \Rightarrow (\mathtt{cwp} \mapsto (\!|\, w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F \,|\!)) * (\mathsf{out} \mapsto \_) * (\mathsf{local} \mapsto \_) * (\mathsf{in} \mapsto \mathrm{fm}_i) * p_1 \end{array}}{\begin{array}{c} \big(\mathtt{cwp} \mapsto (\!|\, w'_{id}, F \cdot \_ \cdot \_ \,|\!)\big) * (\mathsf{out} \mapsto \mathrm{fm}_i) * (\mathsf{local} \mapsto \mathrm{fm}_1) * (\mathsf{in} \mapsto \mathrm{fm}_2) * p_1 \Rightarrow \mathtt{r}_d \mapsto \_ * p_2 \\ \vdash \{(\mathtt{wim} \mapsto w) * p\}\,\mathtt{restore\ r}_s\ \mathtt{o\ r}_d\,\{(\mathtt{wim} \mapsto w) * (\mathtt{r}_d \mapsto v) * p_2\} \end{array}}\ (\mathsf{RESTORE})$$

Fig.11. Selected inference rules [4].

variables ι) needs to ensure $\mathtt{r}_{15}$ still contains f, so that the callee returns to the correct address. Also the fq with the frame p_r needs to satisfy the precondition p' for the remaining instruction sequences of the caller.

The RETL rule simply requires that the post-condition q hold at the end of the instruction i following retl. Also i cannot touch the register $\mathtt{r}_{15}$; therefore $\mathtt{r}_{15}$ specified in p must be the same as in q. Since at the calling point we have already required that the post-condition of the callee guarantees $\mathtt{r}_{15}$ contain the correct return address, we know $\mathtt{r}_{15}$ contains the correct value before retl.

The BE rule checks the "current" value of the register z and decides whether the branch will be taken after executing the following instruction i. If z is not zero, the branch is taken and it does the same things as the JMP rule; otherwise, the branch is not taken and the remaining instruction sequence $\mathbb{I}$ should be well-formed.

Delayed Writes and Register Windows. The bottom layer of our logic is for well-formed instructions. The WR rule requires the ownership of the target register sr in the precondition ($\mathtt{sr} \mapsto \_$). Also it implies there is no delayed write to sr in the delay buffer (see the semantics defined in Fig.9). At the end of the delayed

write, we use $\triangleright_3 \mathtt{sr} \mapsto w_1 \oplus w_2$ to indicate the new value will be ready in up to three cycles. Since the maximum delay cycle X cannot be bigger than 3 and the value of X may vary in different systems, programmers usually take a conservative approach to assume $X = 3$ for the portability of code. Our rule reflects this conservative view. The RD rule says the special register can be read only if it is not in the delay buffer. The SAVE and RESTORE rules reflect the save and recovery of the execution contexts, respectively, which is consistent with the operational semantics of the `save` and `restore` instructions given in Figs.6 and 7 respectively.

4 Refinement Verification of SPARCv8

In this section, we present our relational program logic for refinement verification of SPARCv8 code⑤. As an extension of our conference paper, it consists of the following work.

• We define a new Pseudo-SPARCv8 language as the high-level specification language in Subsection 4.1.

• We make some modifications to the SPARCv8 language defined in Section 2 and let it act as the low-level language in our refinement verification. We present our low-level SPARCv8 language and the modifications in Subsection 4.2.

• We define the correctness of abstract assembly primitives in Subsection 4.3, which is formulated as "contextual refinement" between the implementations and their corresponding abstract assembly primitives.

• We present a new program logic to do refinement verification in Subsection 4.4.

• We show that our new program logic is sound in Subsection 4.5. The semantics of judgements, different from the previous work in our conference paper [4] which only ensures the partial correctness of verified programs, is defined as simulation relations between the low- and high-level programs, which guarantees contextual refinement.

4.1 High-Level Pseudo-SPARCv8 Language

The Pseudo-SPARCv8 language contains two parts: the SPARCv8 code as the client and the set of abstract assembly primitives. Here, the execution of the client SPARCv8 code is required to preserve a restriction between register windows and stacks in memory, shown in Fig.12(a) (`cwp` points to the current window and `wim` marks the invalid window, and the details of overlapping of adjacent windows are omitted in the figure).

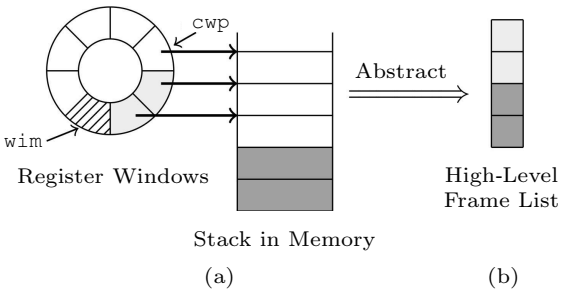


Fig.12. Abstraction of context management. (a) Reigster windows and stack in memory. (b) High-level frame list.

During the execution of the SPARCv8 program, parts of previous procedures' contexts (in Fig.12(a)) are saved in register windows, and the others (the dark gray part in Fig.12(a)) are stored in the stack in memory, since the number of windows is limited. The restriction is that the stack pointer (%`sp`) of each procedure, including the current and previous ones, whose context is saved in register windows currently, should point to the top of its stack frame (shown as the thick arrow in Fig.12(a)), so that the contexts in these windows can be stored correctly in memory when needed. For instance, the context switch routine will check whether the previous window is valid (in clockwise direction), and use the `restore` instruction to set it as the current one and save its contents into the stack (in memory) until the previous one (filled with east north lines in Fig.12(a)) is invalid. The executions of client code are required to preserve such a restriction. Otherwise, some SPARCv8 functions, e.g., context switch routine, whose execution stores the contexts saved in register windows into stacks in memory, cannot be verified if it is unclear where to save the contents of some windows. We do the following when defining the Pseudo-SPARCv8 program to make the client code execution preserve such a restriction.

• In order to ensure that the stack pointer (%`sp`) always points to the top of its stack frame, we require that each instruction, e.g., `add` and `ld`, whose execution does not operate register windows, is prohibited to modify %`sp`. As for the `save` and `restore`, we restrict them to be used in specific forms. We define "`Psave` w" as a macro of "`save` %`sp`, $-w$, %`sp`", whose execution generates a new %`sp` pointing to the stack frame size w allocated newly for the next window. We also define

⑤We give more details about the language definition and the soundness proof in the Coq implementation and the technical report, which is available at https://github.com/jpzha/VeriSparc, Sept. 2020.

⑥In SPARCv8, $\%\mathtt{g}_0$ is always 0, and usually used as a parameter when instructions do not require a specific parameter.

"`Prestore`" as a macro of "`restore` $\%g_0, \%g_0, \%g_0$"⑥, whose execution just restores the previous window and does not modify the value of any register in the previous window. The original `save` and `restore` instructions have no semantics in the high-level client code.

• The special registers in SPARCv8 usually play specific roles and modifying them should be done carefully. For example, `wim` marks which window is invalid.

If we change its value shown in Fig.12(a) to mark another window invalid, as shown in Fig.13, and call context switch routine to save contents of the previous windows into memory until the invalid one at this moment, a problem will arise since we do not know where to save the contents of window marked invalid originally (filled with dots in Fig.13). Therefore, we forbid the client to modify special registers and give *no* semantics to instruction `wr` in high-level client code. Modifying them is hidden in the implementations of abstract assembly primitives in the low level. And the delay buffer can be omitted in high-level program state.

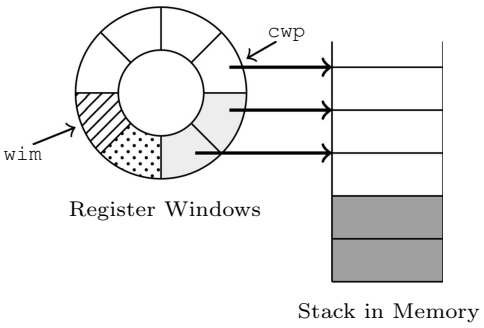


Fig.13. Problem of modifying `wim` arbitrary.

• As shown in Fig.12, we find that we can abstract the register windows and the stack in memory storing contexts into a list (shown in Fig.12(b)). After this abstraction, we do not need to care about whether contexts are saved in register windows or memory, and do not need to describe the contents of the windows unused (the windows in white in Fig.12(a), but excluding the current one pointed by `cwp`) in Pseudo-SPARCv8. The `cwp` register is no longer needed in Pseudo-SPARCv8 since the register windows are abstracted away. The low-level program in our work does not use such abstraction, since the low-level program should be realistically modelled, and the implementations of some primitives need to know the existence of register windows, e.g., the context switch routine that saves the contents of register windows into stacks (in memory).

We define the syntax of the high-level Pseudo-SPARCv8 language in Fig.14. The code (HCode) Π includes the code heap C and the set of abstract primitives (PrimSet) Ω, which is a partial mapping from labels to abstract assembly primitives. The code heap C in Π acts as the client to call abstract assembly primitives. The abstract assembly primitive (Prim) Υ is defined as a relation that takes a list of values as arguments and maps a high-level program state (defined in Fig.15) to another. We add three pseudo instructions in simple instruction (SimpIns). The `Psave` w and `Prestore` restrict the `save` and `restore` instructions to be used in the specific form as mentioned before. We also introduce `print r`, whose execution outputs the value v in general register `r` and generates a message $\mathsf{out}(v)$ as an observable event. The high-level message (HMsg) α can be either an empty message τ, or an output $\mathsf{out}(v)$, or a $\mathsf{call}(\mathtt{f}, \overline{v})$ meaning to call a primitive labelled `f` with arguments $\overline{v}$.

The machine states (HState) of Pseudo-SPARCv8 are defined in Fig.15. The high-level program $\mathbb{P}$

$$
\begin{array}{llll}
\text{(HCode)} & \Pi ::= (C, \Omega) & \text{(CodeHeap)} & C \in \text{Word} \rightharpoonup \text{Comm} \\
\text{(PrimSet)} & \Omega ::= \{\mathtt{f}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathtt{f}_n \rightsquigarrow \Upsilon_n\} & \text{(Prim)} & \Upsilon \in \text{List Val} \to \text{HState} \to \text{HState} \to \text{Prop}
\end{array}
$$

$$
\begin{array}{lll}
\text{(Comm)} & c & ::= \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)} & \mathtt{i} & ::= \mathtt{Psave}\ w \mid \mathtt{Prestore} \mid \mathtt{print\ r} \mid \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{add\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ sr} \\
 & & \quad \mid \mathtt{save\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{restore\ r}_s\ \mathtt{o\ r}_d \mid \ldots \\
\text{(HMsg)} & \alpha & ::= \tau \mid \mathsf{out}(v) \mid \mathsf{call}(\mathtt{f}, \overline{v})
\end{array}
$$

Fig.14. Syntax of Pseudo-SPARCv8 code.

$$
\begin{array}{llllll}
\text{(HProg)} & \mathbb{P} ::= (\Pi, \mathbb{S}) & \text{(HState)} & \mathbb{S} ::= (T, \mathtt{t}, \mathcal{K}, M) & \text{(ThrdPool)} & T ::= \{\mathtt{t} \rightsquigarrow \mathcal{K}\}^* \\
\text{(Tid)} & \mathtt{t} \in \mathbb{Z} & \text{(ThrdLcSt)} & \mathcal{K} ::= (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}) & \text{(HRstate)} & \mathbb{Q} ::= (\mathbb{R}, \mathbb{F}) \\
\text{(HRegFile)} & \mathbb{R} \in \text{HRegName} \to \text{Val} & \text{(HRegName)} & \hat{\mathtt{rn}} ::= \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \mid \mathtt{n} \mid \mathtt{z} \mid \mathtt{c} \mid \mathtt{v} \\
\text{(HFrmList)} & \mathbb{F} ::= \text{nil} \mid (\text{fm}_1, \text{fm}_2) :: \mathbb{F} & \text{(HFrame)} & \text{fm} ::= [v_0, \ldots, v_7]
\end{array}
$$

Fig.15. Machine states for Pseudo-SPARCv8 code.

(HProg) is a pair of the high-level code Π and the high-level state $\mathbb{S}$. The high-level state $\mathbb{S}$ is a tuple including: the thread pool (ThrdPool) T, the current thread ID (Tid) $\mathtt{t}$, the thread local state (ThrdLcSt) $\mathcal{K}$ of the current thread, and the memory (Mem) M.

Thread Local State. The thread local state $\mathcal{K}$ is a triple of high-level register state (HRstate) $\mathbb{Q}$, and program counters pc and npc. The high-level register state $\mathbb{Q}$ consists of the high-level register file (HRegFile) $\mathbb{R}$, and the high-level frame list (HFrmList) $\mathbb{F}$, which is the abstraction of the register windows and the memory storing contexts in the low level. $\hat{\mathtt{rn}}$ is the high-level register names (HRegName), where the cwp register is omitted as introduced before and we also omit special registers for simplicity, because we forbid the high-level client code to modify them[7]. The high-level frame list $\mathbb{F}$ is a list of pairs $(\mathrm{fm}_1, \mathrm{fm}_2)$, which is used to save the contexts (local and in registers) fm_1 and fm_2 of previous procedures. Then, we define the switch primitive as an instantiation of Υ below.

$$
\begin{aligned}
&\mathsf{switch} \ ::= \\
&\quad \lambda\, \overline{v}, \mathbb{S}, \mathbb{S}'.\ \exists\, \mathtt{t}'.\ M(\mathsf{TaskNew}) = (\mathtt{t}', 0) \wedge T(\mathtt{t}') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \\
&\qquad\quad \wedge T' = T\{\mathtt{t} \rightsquigarrow (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})\} \wedge \mathtt{t} \neq \mathtt{t}' \wedge \overline{v} = \mathrm{nil}, \\
&\quad \text{where } \mathbb{S} = (T, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M), \\
&\qquad\quad \mathbb{S}' = (T', \mathtt{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}).
\end{aligned}
$$

The switch primitive takes no arguments ($\overline{v} = \mathrm{nil}$), and changes the identifier of the current thread according to the value in the location TaskNew. We use parameters $\mathbb{S}$ and $\mathbb{S}'$ to represent the machine states before and after the execution of switch respectively.

Operational Semantics in High Level. The operational semantics for Pseudo-SPARCv8 is defined in Fig.16. The high-level program transition relation $(\Pi, \mathbb{S}) : \stackrel{\alpha}{\Longrightarrow} (\Pi, \mathbb{S}')$ is defined in Fig.16(a). In each step, the program may either execute the instruction pointed by pc and generate empty message τ or an output $\mathsf{out}(v)$, or call an abstract assembly primitive in the primitive set. When calling an abstract assembly primitive, a message $\mathsf{call}(\mathtt{f}, \overline{v})$ will be generated. It means that we hope to call the abstract assembly primitive labelled f with the arguments $\overline{v}$ ($\mathsf{args}(\mathbb{Q}, M, \overline{v})$, whose definition is omitted here, means getting the arguments $\overline{v}$ from $\mathbb{Q}$ and M).

The thread local step is defined in Fig.16(b). The step for simple instruction i is represented as "$\mathsf{exec}(\mathtt{i}, \_, \_)$". We show the state transition relation for pseudo instructions Psave w and Prestore in Fig.16(c). Supposing the current register state $\mathbb{Q}$ is $(\mathbb{R}, \mathbb{F})$, the execution of Psave w will save the local and in registers into the high-level frame list $\mathbb{F}$. It also allocates a new block b of size 64-byte to w bytes as a new stack frame in memory (shown as $\boldsymbol{alloc}(M, b, 64, w) = M'$). The reason why it starts from 64-byte is that the 0–64 bytes (16 words) in a stack frame are usually reserved to save the context in window (local and in registers) by convention, and this part of memory is abstracted away in the Pseudo-SPARCv8 program as we have explained and shown in Fig.12. Prestore does the reverse, freeing the block of the current stack frame (shown as $\boldsymbol{free}(b, M) = M'$), and restoring the context of the previous procedure saved in $\mathbb{F}$.

4.2 Low-Level SPARCv8 Program

The global program transition of the low-level SPARCv8 program is defined as the following form.

$$
\frac{\begin{array}{l}(R, D) \rightrightarrows (R', D') \\ C \vdash ((M, (R', F), D'), \mathtt{pc}, \mathtt{npc}) \circ\!\!\xrightarrow{\tau/\mathsf{out}(v)} ((M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')\end{array}}{(C, (M, (R, F), D), \mathtt{pc}, \mathtt{npc}) ::\!\!\xrightarrow{\tau/\mathsf{out}(v)} (C, (M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}
$$

The low-level SPARCv8 program is slightly different from the SPARCv8 program defined in Section 2.

1) The low-level SPARCv8 program uses the instructions defined in Fig.14. It means that we need to give semantics to the pseudo instructions Psave, Prestore and print in the low-level SPARCv8 program. Since Psave and Prestore are simply special forms of save and restore as explained, and print is a primitive responsible for generating observable events, defining their semantics is not a challenge and the translation of programs in this modified language into ones in the standard SPARCv8 language is trivial.

2) The program transition defined in Section 2 does not generate observable events. Here, since we want to support refinement verification and use the event trace refinement [9], each step of the program generates either an empty message τ, or an output $\mathsf{out}(v)$ produced by print.

[7] There is no problem to reserve special registers in the high-level register file and permit the high-level client code to read them.

$$\frac{\Pi = (C, \Omega) \quad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\tau} (\mathcal{K}', M')}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\Longrightarrow^{\tau} (\Pi, (T, \mathtt{t}, \mathcal{K}', M'))} \qquad \frac{\Pi = (C, \Omega) \quad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\mathsf{out}(v)} (\mathcal{K}', M)}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\xRightarrow{\mathsf{out}(v)} (\Pi, (T, \mathtt{t}, \mathcal{K}', M))}$$

$$\frac{\begin{array}{c}\Pi = (C, \Omega) \quad C \Vdash (\mathcal{K}, M) \circ\xrightarrow{\mathsf{call}(\mathtt{f}, \overline{v})} (\mathcal{K}', M) \quad \Omega(\mathtt{f}) = \Upsilon \\ \Upsilon\,(\overline{v})(T, \mathtt{t}, \mathcal{K}', M)(T', \mathtt{t}', \mathcal{K}'', M')\end{array}}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\Longrightarrow^{\tau} (\Pi, (T', \mathtt{t}', \mathcal{K}'', M'))}$$

(a)

$$\frac{C(\mathtt{pc}) = \mathtt{i} \quad \mathsf{exec}(\mathtt{i}, (\mathbb{Q}, M), (\mathbb{Q}', M'))}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} ((\mathbb{Q}', \mathtt{npc}, \mathtt{npc} + 4), M')}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \quad \mathtt{r}_{15} \in \mathrm{dom}(\mathbb{R})}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} (((\mathbb{R}\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\}, \mathbb{F}), \mathtt{npc}, \mathtt{f}), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \quad \mathbb{R}(\mathtt{r}_{15}) = \mathtt{f}}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\tau} ((\mathbb{R}, \mathbb{F}), \mathtt{npc}, \mathtt{f} + 8), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{print\ r} \quad \mathbb{R}(\mathtt{r}) = v}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\mathsf{out}(v)} (((\mathbb{R}, \mathbb{F}), \mathtt{npc}, \mathtt{npc} + 4), M)}$$

$$\frac{\mathtt{pc} \notin \mathrm{dom}(C) \quad \mathtt{npc} = \mathtt{pc} + 4 \quad \mathsf{args}(\mathbb{Q}, M, \overline{v})}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\xrightarrow{\mathsf{call}(\mathtt{pc}, \overline{v})} ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M)}$$

(b)

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, \mathbb{F}) \quad \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \_, \mathsf{local} \rightsquigarrow \_, \mathsf{in} \rightsquigarrow \mathbb{R}([\mathsf{out}])\}\{\%\mathtt{sp} \rightsquigarrow (b, 0)\} \\ \boldsymbol{alloc}(M, b, 64, w) = M' \quad \mathbb{Q}' = (\mathbb{R}', (\mathbb{R}([\mathsf{local}]), \mathbb{R}([\mathsf{in}])) :: \mathbb{F})\end{array}}{\mathsf{exec}(\mathtt{Psave}\ w, (\mathbb{Q}, M), (\mathbb{Q}', M'))}$$

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, (\mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}) \quad \mathbb{R}(\%\mathtt{sp}) = (b, 0) \quad \boldsymbol{free}(b, M) = M' \\ \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \mathbb{R}([\mathsf{in}]), \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{in} \rightsquigarrow \mathrm{fm}_2\} \quad \mathbb{Q}' = (\mathbb{R}', \mathbb{F})\end{array}}{\mathsf{exec}(\mathtt{Prestore}, (\mathbb{Q}, M), (\mathbb{Q}', M'))}$$

(c)

Fig.16. Selected operational semantics rules for high-level program. (a) High-level program transitions. (b) High-level thread local transitions. (c) High-level instruction transitions.

Note that the client code and the implementations of primitives in the low level are both SPARCv8 code heap. There is no need to define their linking in semantics.

4.3 Primitive Correctness

We first establish a state relation between low- and high-level program states. We define this relation below ($\uplus$ means disjoint union), shown as "$S \sim \mathbb{S}$".

$$\frac{\begin{array}{c}M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}, 0)\} \uplus M' \\ (M_c, Q) \Downarrow_{\mathsf{c}} (\mathtt{t}, \mathcal{K}) \quad M_T \Downarrow_{\mathsf{r}} T\backslash\{\mathtt{t}\} \quad D = \mathrm{nil}\end{array}}{(M, Q, D) \sim (T, \mathtt{t}, \mathcal{K}, M')}$$

The low-level memory M is split into four parts: M_c used saving the context of the current thread $\mathtt{t}$; M_T saving the contexts of the ready threads, except the current thread $\mathtt{t}$; a singleton memory cell located $\mathsf{TaskCur}$ saving the current thread ID; and the shared memory M' that is the same as the high-level memory. The delayed buffer D is nil, since the client is not permitted to modify any special register through the `wr` instructions. M_T is "abstracted" as a thread pool in the high-level program. Their relation is represented as "$M_T \Downarrow_{\mathsf{r}} T\backslash\{\mathtt{t}\}$". We use "$(M_c, Q) \Downarrow_{\mathsf{c}} (\mathtt{t}, \mathcal{K})$" to represent the state relation of the current thread in low- and high-level programs.

The correctness of abstract assembly primitives is defined in terms of contextual refinement. We give its formal definition in Definition 1. And we use the "event trace refinement" proposed by Liang *et al.*[9]

Definition 1 (Primitive Correctness). $C_{\mathrm{as}} \sqsubseteq \Omega$ *iff for any* C, S, $\mathbb{S}$, $\mathtt{pc}$ *and* $\mathtt{npc}$, *if* $S \sim \mathbb{S}$ *and* $\mathsf{ProgSafe}(\mathbb{P})$, *then* $P \subseteq \mathbb{P}$, *where* $P = (C \uplus C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc})$, $\mathbb{P} = ((C, \Omega), \mathbb{S})$ *and* $\mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})$.

We use the code heap C_{as} to represent the implementations and Ω to represent the set of corresponding

abstract assembly primitives. The contextual refinement, denoted as $C_{\text{as}} \sqsubseteq \Omega$, says that if and only if for any client code (or context) C, low-level program state S, high-level program state $\mathbb{S}$, program counters `pc` and `npc`, if the low- and high-level program states satisfy the state relation $S \sim \mathbb{S}$ and the high-level program will never get stuck (shown as $\mathsf{ProgSafe}(\mathbb{P})$), then there is an event trace refinement [9], which means that P produces no more observable behaviors than $\mathbb{P}$ and is denoted as $P \subseteq \mathbb{P}$, between low- and high-level programs. $\mathsf{ProgSafe}(\mathbb{P})$ is defined formally below:

$$\mathsf{ProgSafe}(\mathbb{P}) \ ::= \forall \mathbb{P}'.(\mathbb{P} :\Longrightarrow^* \mathbb{P}') \Longrightarrow (\exists \mathbb{P}''.\mathbb{P}' :\Longrightarrow \mathbb{P}'').$$

The client code C is the SPARCv8 code. Therefore, the high-level code is a pair of C and Ω, and the low-level code is just the union of C and C_{as}, shown as $(C \uplus C_{\text{as}})$, because both of C and C_{as} are SPARCv8 code heaps.

$$C \uplus C_{\text{as}} \ ::= C \cup C_{\text{as}} \quad \text{if } \mathrm{dom}(C) \cap \mathrm{dom}(C_{\text{as}}) = \emptyset.$$

4.4 Relational Program Logic for Refinement Verification

Relational Assertion. Fig.17 gives the relational assertion language, and its semantics is given in Fig.18. The relational assertions are interpreted over the relational state $(S, \mathbb{S}, A, w)$, which contains the low-level state S, the high-level state $\mathbb{S}$, the abstract assembly primitive command A defined in Fig.18, and the word w recording the number of the tokens. The high-level primitive command A is either an abstract assembly primitive Υ parameterized over arguments $\overline{v}$, or a $\perp$ meaning the primitive has already been executed. The relational assertion $\mathbb{p}$ reserves the original assertion p describing the low-level state S.

$$\begin{aligned}(\text{RelAsrt}) \quad \mathbb{p}, \mathbb{q} \ ::= \ & p \mid \hat{\mathtt{rn}} \rightarrowtail v \mid l \rightarrowtail v \mid \mathtt{Emp} \\ & \mid \mathtt{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K} \mid \mathtt{t} \rightsquigarrow_{\mathsf{r}} \mathcal{K} \mid (\!|A|\!) \mid \blacklozenge(w) \\ & \mid \mathbb{p}\!\downarrow \mid \mathbb{p} \wedge \mathbb{q} \mid \mathbb{p} \vee \mathbb{q} \mid \mathbb{p} * \mathbb{q} \mid \ldots\end{aligned}$$

Fig.17. Syntax of relational assertion.

We define $\hat{\mathtt{rn}} \rightarrowtail v$ and $l \rightarrowtail v$ to describe the state of register files and memory in the high level. The assertion `Emp` says that the high-level memory and the thread pool are both empty, and the low-level state satisfies `emp` defined in Fig.9. The assertions $\mathtt{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}$ and $\mathtt{t} \rightsquigarrow_{\mathsf{r}} \mathcal{K}$ represent the thread local state of the current and ready thread respectively. Note that the threads in the thread pool are viewed as resources and can be separated by separation conjunction.

The assertion $(\!|A|\!)$ means the current high-level primitive command is A. And the assertion $\blacklozenge(w)$ takes a word w, which can also be separated by separation conjunction $(*)$, to state that the number of tokens in current state is "no less" than w. Tokens are used to avoid infinite loops and recursive function calls to make sure the termination preserving refinement.

The assertion $\mathbb{p}\!\downarrow$, which is similar to $p\!\downarrow$ defined in Fig.9, describes the state after executing one step of

$$\begin{aligned}
&(S,\mathbb{S},A,w) \models \mathtt{Emp} && ::= \ \mathbb{S}.M = \emptyset \wedge \mathbb{S}.T = \emptyset \wedge S \models \mathsf{emp} \\
&(S,\mathbb{S},A,w) \models p && ::= \ S \models p \wedge \mathbb{S}.M = \emptyset \wedge \mathbb{S}.T = \emptyset \\
&(S,\mathbb{S},A,w) \models \hat{\mathtt{rn}} \rightarrowtail v && ::= \ \mathbb{S}.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\hat{\mathtt{rn}}) = v \wedge (S,\mathbb{S},A,w) \models \mathtt{Emp} \\
&(S,\mathbb{S},A,w) \models l \rightarrowtail v && ::= \ \mathbb{S}.M = \{l \rightsquigarrow v\} \wedge \mathbb{S}.T = \emptyset \wedge S \models \mathsf{emp} \\
&(S,\mathbb{S},A,w) \models \mathtt{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K} && ::= \ \mathbb{S}.T \backslash \{\mathtt{t}\} = \emptyset \wedge \mathbb{S}.\mathtt{t} = \mathtt{t} \wedge \mathbb{S}.\mathcal{K} = \mathcal{K} \wedge \mathbb{S}.M = \emptyset \wedge S \models \mathsf{emp} \\
&(S,\mathbb{S},A,w) \models \mathtt{t} \rightsquigarrow_{\mathsf{r}} \mathcal{K} && ::= \ \mathbb{S}.T = \{\mathtt{t} \rightsquigarrow \mathcal{K}\} \wedge \mathbb{S}.M = \emptyset \wedge \mathtt{t} \neq \mathbb{S}.\mathtt{t} \wedge S \models \mathsf{emp} \\
&(S,\mathbb{S},A,w) \models (\!|A'|\!) && ::= \ A = A' \wedge (S,\mathbb{S},A,w) \models \mathtt{Emp} \\
&(S,\mathbb{S},A,w) \models \blacklozenge(w') && ::= \ w' \leqslant w \wedge (S,\mathbb{S},A,w) \models \mathtt{Emp} \\
&(S,\mathbb{S},A,w) \models \mathbb{p}\!\downarrow && ::= \ \exists S'.((S',\mathbb{S},A,w) \models \mathbb{p}) \wedge (R',D') \rightrightarrows (R,D) \\
& && \qquad \text{where} \ \ S = (M,(R,F),D),\ S' = (M,(R',F),D') \\
&(S,\mathbb{S},A,w) \models \mathbb{p} * \mathbb{q} && ::= \ \exists S_1,S_2,\mathbb{S}_1,\mathbb{S}_2,w_1,w_2.\, S = S_1 \uplus S_2 \wedge \mathbb{S} = \mathbb{S}_1 \uplus \mathbb{S}_2 \wedge \\
& && \qquad w = w_1 + w_2 \wedge (S_1,\mathbb{S}_1,A,w_1) \models \mathbb{p} \wedge (S_2,\mathbb{S}_2,A,w_2) \models \mathbb{q}
\end{aligned}$$

$$T_1 \perp T_2 \ ::= (\mathrm{dom}(T_1) \cap \mathrm{dom}(T_2)) = \emptyset$$

$$\mathbb{S}_1 \uplus \mathbb{S}_2 \ ::= \begin{cases} (T_1 \cup T_2, \mathtt{t}, \mathcal{K}, M_1 \cup M_2), & \text{if } T_1 \perp T_2 \wedge M_1 \perp M_2 \wedge \\ & \quad \mathbb{S}_1 = (T_1, \mathtt{t}, \mathcal{K}, M_1) \wedge \mathbb{S}_2 = (T_2, \mathtt{t}, \mathcal{K}, M_2), \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$(\text{HPrimCom}) \quad A \ ::= \Upsilon(\overline{v}) \mid \perp \qquad \frac{\Upsilon\,(\overline{v})(\mathbb{S})(\mathbb{S}')}{(\Upsilon(\overline{v}),\mathbb{S}) \dashrightarrow (\perp,\mathbb{S}')}$$

Fig.18. Semantics of relation assertion.

delayed writes.

Inference Rules in Relational Program Logic. The code specification $\hat{\theta}$ and the code heap specification Ψ in relational program logic are defined below.

$$\begin{array}{l}(\text{valList})\iota \in \text{list value}\\ (\text{pAsrt})\ \mathtt{fp}, \mathtt{fq} \in \text{valList} \to \text{RelAsrt}\\ (\text{CdSpec})\hat{\theta} ::= (\mathtt{fp}, \mathtt{fq})\\ (\text{CdHpSpec})\ \Psi\ ::= \{\mathtt{f} \rightsquigarrow \hat{\theta}\}^*\end{array}$$

Here, $\mathtt{fp}$ and $\mathtt{fq}$ are relational assertions parameterized over a list of values ι. Fig.19 shows selected inference rules for refinement verification in our logic. The top rule WfPrim verifies the contextual refinement between the code heap C_{as} and the corresponding abstract assembly primitive set Ω. It requires that each code block specified in Ψ can be verified with respect to its specification, shown as $(\vdash C_{\text{as}} : \Psi)$, and the specifications of the implementations of abstract assembly primitives need to meet some restrictions, shown as $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$, which we will discuss in more details following. The inference rules for `jmp` and `call` in relational program logic will consume a token, shown as $\blacklozenge(1)$, in order to avoid infinite loops and recursive function calls and ensure termination-preserving. $\mathsf{wf}(\mathtt{p}_r)$, whose definition is omitted here, means there is no subterm in form of $(\mathtt{t} \rightsquigarrow_{\mathtt{c}} \mathcal{K})$, $(\hat{\mathtt{rn}} \rightarrowtail v)$ and $(\!|A|\!)$ in the frame $\mathtt{p}_r$, because the state they described is not separated by separation conjunction $*$. The ABSCSQ rule allows us to execute the high-level primitive command specified in precondition. The implication $\mathtt{p} \Rrightarrow \mathtt{p}'$ is defined below formally:

$$\begin{array}{l}(\mathtt{p} \Rightarrow \mathtt{p}') \vee\\ (\forall S, \mathbb{S}, A, w.\ ((S, \mathbb{S}, A, w) \models \mathtt{p}) \Longrightarrow\\ \quad (\exists \mathbb{S}', A', w'.\ ((A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')) \wedge ((S, \mathbb{S}', A', w') \models \mathtt{p}')).\end{array}$$

The inference rules for instructions are not shown here, since they are not different from the rules in Fig.11.

Well-Defined Specification. $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ is defined formally in Definition 2. It contains three properties that the specification needs to satisfy.

Definition 2 (Well-Defined Specification). $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ *holds, iff*:

1) *for any* $\overline{v}$, $\mathbb{S}$, $\mathbb{S}'$, $\mathbb{S}_r$, *if* $\Upsilon(\overline{v})(\mathbb{S})(\mathbb{S}')$, *and* $\mathbb{S} \perp \mathbb{S}_r$, *then the following holds* (*where* $\mathtt{f} = \mathbb{S}'.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\mathtt{r}_{15})$) :

- $\mathbb{S}'.\mathcal{K}.\mathtt{pc} = \mathtt{f}+8$, $\mathbb{S}'.\mathcal{K}.\mathtt{npc} = \mathtt{f}+12$;
- *there exist* $\mathbb{S}''$ *and* $\mathbb{S}'_r$, *such that* $\Upsilon(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, *and* $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$;

2) *for any* ι, *there exists* $\overline{v}$, *such that* $\mathtt{fp}\ \iota \Rightarrow (\!|\Upsilon(\overline{v})|\!) * \text{true}$, *and* $\mathtt{fq}\ \iota \Rightarrow (\!|\bot|\!) * \text{true}$;

3) *for any* $\overline{v}, S, \mathbb{S}$, *if* $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\Upsilon(\overline{v}), \overline{v})$, *there exist* $\iota, \mathtt{p}_r$ *and* w, *such that*

$\boxed{\Psi \vdash C_{\text{as}} : \Psi}$ (Well-Formed Primitive)

$$\frac{\begin{array}{l}\vdash C_{\text{as}} : \Psi\\ \text{for all } \mathtt{f} \in \text{dom}(\Omega) :\\ \quad \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}) \quad \Omega(\mathtt{f}) = \Upsilon \quad \mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)\end{array}}{\Psi \vdash C_{\text{as}} : \Omega}\ (\mathsf{WfPrim})$$

$\boxed{\vdash C_{\text{as}} : \Psi}$ (Well-Formed Code Heap)

$$\frac{\text{for all } \mathtt{f} \in \text{dom}(\Psi), \iota :\ \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}) \quad \Psi \vdash \{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}]}{\vdash C_{\text{as}} : \Psi}\ (\mathsf{WfInt})$$

$\boxed{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}$ (Well-Formed Instruction Sequences)

$$\frac{\Vdash \{\mathtt{p}\downarrow\}\,\mathtt{i}\,\{\mathtt{p}'\} \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f}+4 : \mathbb{I}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{i}; \mathbb{I}}\ (\mathsf{SEQ})$$

$$\frac{\begin{array}{c}\mathtt{p}\downarrow \Rightarrow (\mathtt{a} =_a \mathtt{f}') \quad \mathtt{f}' \in \text{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathtt{fp}, \mathtt{fq})\\ \Vdash \{\mathtt{p}\downarrow\downarrow\}\,\mathtt{i}\,\{\mathtt{p}' * \blacklozenge(1)\} \quad \exists \iota, \mathtt{p}_r.\ (\mathtt{p}' \Rrightarrow \mathtt{fp}\ \iota * \mathtt{p}_r) \wedge (\mathtt{fq}\ \iota * \mathtt{p}_r \Rightarrow \mathtt{q}) \wedge \mathsf{wf}(\mathtt{p}_r)\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{jmp\ a}; \mathtt{i}}\ (\mathsf{JMP})$$

$$\frac{\begin{array}{c}\mathtt{f}' \in \text{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathtt{fp}, \mathtt{fq}) \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f}+8 : \mathbb{I}\\ \mathtt{p}\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \_) * \mathtt{p}_1 \quad \Vdash \{(\mathtt{r}_{15} \mapsto \mathtt{f} * \mathtt{p}_1)\downarrow\}\,\mathtt{i}\,\{\mathtt{p}_2 * \blacklozenge(1)\}\\ \exists \iota, \mathtt{p}_r.\ (\mathtt{p}_2 \Rightarrow \mathtt{fp}\ \iota * \mathtt{p}_r) \wedge (\mathtt{fq}\ \iota * \mathtt{p}_r \Rightarrow \mathtt{p}') \wedge (\mathtt{fq}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f}) \wedge \mathsf{wf}(\mathtt{p}_r)\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{call\ f}'; \mathtt{i}; \mathbb{I}}\ (\mathsf{CALL})$$

$$\frac{\mathtt{p}\downarrow\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_1 \quad \Vdash \{\mathtt{p}_1\}\,\mathtt{i}\,\{\mathtt{p}_2\} \quad (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_2 \Rrightarrow \mathtt{q}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{retl}; \mathtt{i}}\ (\mathsf{RETL})$$

$$\frac{\mathtt{p} \Rrightarrow \mathtt{p}' \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q}')\}\,\mathtt{f} : \mathbb{I} \quad \mathtt{q}' \Rrightarrow \mathtt{q}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}\ (\mathsf{ABSCSQ})$$

Fig.19. Selected inference rules for refinement verification.

$(S, \mathbb{S}, \Upsilon(\overline{v}), w) \models (\mathtt{fp}\ \iota * \mathtt{p}_r)$, $(\mathtt{fq}\ \iota * \mathtt{p}_r) \Rightarrow \mathsf{INV}(\bot, \_)$, *and* $\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r)$ *hold.*

First, the program counters should equal $\mathtt{f}+8$ and $\mathtt{f}+12$ respectively, where $\mathtt{f}$ is contained in the $\mathtt{r}_{15}$ register after the execution of the abstract assembly primitive Υ. It ensures that low-level implementations and corresponding high-level abstract assembly primitives return to the same code pointers. We also require that if an abstract assembly primitive can execute safely on a part of program state, it can also execute safely on the whole program state, and keeps the additional program state unchanged. $\mathbb{S} \perp \mathbb{S}_r$ is defined formally below:

$$\begin{aligned}\mathbb{S} \perp \mathbb{S}_r \ ::= & \ T \perp T' \wedge M \perp M' \wedge \mathtt{t} = \mathtt{t}' \wedge \mathcal{K} = \mathcal{K}' \\ \text{where} \ \ & \mathbb{S} = (T, \mathtt{t}, \mathcal{K}, M),\ \mathbb{S}_r = (T', \mathtt{t}', \mathcal{K}', M').\end{aligned}$$

Second, the abstract assembly primitive should be specified in the precondition, and its execution should be done in the final state. Third, an "invariant" holds between low- and high-level programs at the entry of the function. Our logic needs to ensure that such invariant can be reestablished when the function returns. We define such invariant as INV formally below:

$$\begin{aligned}\mathsf{INV}(A, \overline{v}) ::= \{(S, \mathbb{S}, A, w) \mid & S \sim \mathbb{S} \wedge (\exists \mathbb{S}'.(\mathbb{S}) \dashrightarrow^* (\bot, \mathbb{S}')) \\ & \wedge \mathsf{args}(\mathbb{S}.\mathcal{K}.\mathbb{Q}\mathbb{S}.M, \overline{v})\}.\end{aligned}$$

The invariant consists of the state relation between low- and high-level program states, shown as $S \sim \mathbb{S}$ and defined in Subsection 4.3, and the safe execution of the primitive command A, which means that A can execute zero (if $A = \bot$) or one step (if $A = \Upsilon(\overline{v})$) from the current state, shown as $\exists \mathbb{S}'.(A, \mathbb{S}) \dashrightarrow^* (\bot, \mathbb{S}')$. Including the safe execution of A in the invariant is essential, since we can get some knowledge of the high-level program state from it. For example, if $\mathsf{INV}(\mathsf{switch}(\mathrm{nil}), \mathrm{nil})$ holds, we know that the location TaskNew must save a pointer pointing to a ready thread in the thread pool; otherwise the switch primitive cannot execute.

$$\begin{aligned}&\mathsf{INV}(\mathsf{switch}(\mathrm{nil}), \mathrm{nil}) \implies \\ &\quad \exists \mathtt{t}, \mathcal{K}.\, (\mathtt{t} \rightsquigarrow_r \mathcal{K}) * (\mathsf{TaskNew} \mapsto (\mathtt{t}, 0)) * \mathrm{true}\end{aligned}$$

We use the frame $\mathtt{p}_r$ for local reasoning. $\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r)$, whose definition is omitted here, says that $\mathtt{p}_r$ still holds after the execution of $\Upsilon(\overline{v})$.

4.5 Semantics and Soundness

We first define the simulation relation for instruction sequences. It says C_{as} can execute safely from S, pc and npc until reaching the end of the current instruction sequence ($C_{\mathrm{as}}[\mathtt{pc}]$), and q holds if $C_{\mathrm{as}}[\mathtt{pc}]$ ends with the return instruction retl, and for each step of the low-level execution, the high-level program will execute zero or one step. It is formally defined in Definition 3. Here we use "$\_ \longmapsto^n \_$" to represent n-step execution. The w in simulation records the number of tokens. It will be consumed when meeting the jmp and call instructions to avoid infinite loops and recursive in low level for termination preserving, and reset when the high-level abstract assembly primitive executes.

Definition 3 (Simulation for Instruction Sequence). $\mathtt{q}; \Psi \models (C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_w (A, \mathbb{S})$ *holds if and only if the followings are true* (*we omit the case for* be *here, which is similar to* jmp).

1) *If* $C_{\mathrm{as}}(\mathtt{pc}) = \mathtt{i}$ *then*:

- *there exist* $S', \mathtt{pc}', \mathtt{npc}'$, *such that* $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$,
- *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$,

  if $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$, *then there exist* $A', \mathbb{S}'$ *and* w', *such that*:
  a) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
  b) $\mathtt{q}; \Psi \models (C_{\mathrm{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_{w'} (A', \mathbb{S}')$.

2) *If* $C_{\mathrm{as}}(\mathtt{pc}) = \mathtt{jmp\ a}$ *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
- *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$,
  if $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, *then there exist* $\mathtt{fp}$, $\mathtt{fq}$, ι, A', $\mathbb{S}'$, w', $w'' < w'$ *and* $\mathtt{p}_r$ *such that the followings hold*:
  a) $\mathtt{npc}' = \mathtt{pc}'+4$, $\Psi(\mathtt{pc}') = (\mathtt{fp}, \mathtt{fq})$,
  b) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
  c) $(S', \mathbb{S}', A', w'') \models (\mathtt{fp}\ \iota) * \mathtt{p}_r$, $(\mathtt{fq}\ \iota) * \mathtt{p}_r \Rightarrow \mathtt{q}$, $\mathsf{wf}(\mathtt{p}_r)$.

3) *If* $C_{\mathrm{as}}(\mathtt{pc}) = \mathtt{be\ f}$ *then* ... .

4) *If* $C_{\mathrm{as}}(\mathtt{pc}) = \mathtt{call\ f}$ *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
- *for any* S', $\mathtt{pc}'$ *and* $\mathtt{npc}'$,
  if $C_{\mathrm{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, *then there exist* $\mathtt{fp}, \mathtt{fq}, \iota, A', \mathbb{S}', w', w'' < w'$ *and* $\mathtt{p}_r$, *such that the followings hold*:
  a) $\mathtt{npc}' = \mathtt{pc}'+4$, $\Psi(\mathtt{pc}') = (\mathtt{fp}, \mathtt{fq})$,
  b) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
  c) $(S', \mathbb{S}', A', w'') \models (\mathtt{fp}\ \iota) * \mathtt{p}_r$, $\mathsf{wf}(\mathtt{p}_r)$,
  d) *for any* S_0, $\mathbb{S}_0$, A_0, w_0,
  if $(S_0, \mathbb{S}_0, A_0, w_0) \models (\mathtt{fq}\ \iota) * \mathtt{p}_r$, *then* $\mathtt{q}; \Psi \models (C_{\mathrm{as}}, S_0, \mathtt{pc}+8, \mathtt{pc}+12) \preccurlyeq_{w_0} (A_0, \mathbb{S}_0)$,

e) $(\mathtt{fq}\ \iota) \Rightarrow (\mathtt{r}_{15} = \mathtt{pc})$.

5) *If* $C_{\text{as}}(\mathtt{pc}) = \mathtt{retl}$ *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
- *for any* S', $\mathtt{pc}'$ *and* $\mathtt{npc}'$, *if* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$, *then there exist* A', $\mathbb{S}'$, and w', *such that*:
  a) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
  b) $(S', \mathbb{S}', A', w') \models \mathfrak{q}$, $\mathtt{pc}' = S'.Q.R(\mathtt{r}_{15}) + 8$, *and* $\mathtt{npc}' = S'.Q.R(\mathtt{r}_{15}) + 12$.

Then we define the semantics for well-formed instruction sequences and well-formed code heap below.

Definition 4 (Judgment Semantics).

• $\Psi \models \{(\mathfrak{p}, \mathfrak{q})\}\ \mathtt{f} : \mathbb{I}$ *if and only if, for all* C_{as}, S, $\mathbb{S}$, A *and* w *such that* $C_{\text{as}}[\mathtt{f}] = \mathbb{I}$ *and* $(S, \mathbb{S}, A, w) \models \mathfrak{p}$, *we have* $\mathfrak{q}; \Psi \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_w (A, \mathbb{S})$.

• $\models C_{as} : \Psi$ *if and only if, for all* $\mathtt{f}$, $\mathtt{fp}$ *and* $\mathtt{fq}$ *such that* $\Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq})$, *we have* $\Psi \models \{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)\}\ \mathtt{f} : C_{\text{as}}[\mathtt{f}]$ *for all* ι.

Next, we define the simulation for functions in Definition 5. It says that if there exists a relational state $(S, \mathbb{S}, A, w)$ satisfying the precondition $\mathfrak{p}$, then we have the simulation $\mathfrak{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})$ defined in Definition 6. The simulation, which ensures the safe execution of the low-level SPARCv8 function and its corresponding high-level abstract assembly primitive, in Definition 6 carries an index i, which is used to ensure the termination preserving, and the depth k of function calls, which increases by the `call` instruction and decreases by `retl` (unless $k = 0$).

Definition 5 (Simulation for Function).

$$(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathfrak{p}, \mathfrak{q})} A \ ::= \forall S, \mathbb{S}, w. (S, \mathbb{S}, A, w) \models \mathfrak{p} \Longrightarrow \exists i \in \mathit{Index}.\ \mathfrak{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S}),$$

where $\mathfrak{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ *is defined in Definition* 6.

Definition 6. $\mathfrak{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ *holds if and only if the followings are true*:

1) *if* $C_{\text{as}}(\mathtt{pc}) \in \{\mathtt{i}, \mathtt{jmp\ a}, \mathtt{be\ f}\}$, *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow} (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
- *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow} (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then one of the followings hold*:
  a) $\exists j < i.\ \mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (A, \mathbb{S})$;
  b) *there exists* $\mathbb{S}'$, $j \in \mathit{Index}$, *such that* $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ *and* $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (\bot, \mathbb{S}')$ *holds*;

2) *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{call\ f}$, *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
- *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then one of the followings holds*:
  a) $\exists j < i.\ \mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (A, \mathbb{S})$;
  b) *there exist* $\mathbb{S}'$, $j \in \mathit{Index}$, *such that* $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ *and* $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (\bot, \mathbb{S}')$ *holds*;

3) *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{retl}$, *then*:

- *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
- *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\overset{\tau}{\Longrightarrow}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then there exist* $j \in \mathit{Index}$, $\mathbb{S}'$ *and* A', *such that the followings hold*:
  a) *either* $j < i$, $\mathbb{S}' = \mathbb{S}$ *and* $A' = A$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$;
  b) *if* $k = 0$, *then there exists* w', *such that* (*where* $\mathtt{f} = S'.Q.R(\mathtt{r}_{15})$):
  $(S', \mathbb{S}', A', w') \models \mathfrak{q}$, $A' = \bot$, $\mathtt{pc}' = \mathtt{f}+8$, *and* $\mathtt{npc}' = \mathtt{f}+12$;
  else $\mathfrak{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k-1} (A', \mathbb{S}')$.

Then we give the semantics for well-formed primitive in Definition 7.

Definition 7 (Well-Defined Primitive Set Semantics).

$$\Psi \models C_{\text{as}} : \Omega \ ::= \forall \mathtt{f} \in \mathrm{dom}(\Omega),\ \iota.\ \exists \Upsilon, \overline{v}, \mathtt{fp}, \mathtt{fq}.\ \mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon) \wedge (\mathtt{fp}\ \iota \Rightarrow (\!|\, \Upsilon(\overline{v}) \,|\!) * \mathrm{true}) \wedge (C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)} \Upsilon(\overline{v}),$$

where $\Omega(\mathtt{f}) = \Upsilon, \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq})$.

It says that for any high-level abstract assembly primitive in the primitive set Ω, we can establish a simulation relation defined in Definition 5 between its low-level implementation in code heap C_{as} and itself.

Theorem 1 shows the soundness of our logic.

Soundness Proof. We show the soundness proof of our logic. We first give Lemma 3 to show that the simulation relation for functions can be decomposed into the simulation relations for individual instruction sequences.

Lemma 3. *If* $\Psi \models \{(\mathfrak{p}, \mathfrak{q})\}\ \mathtt{pc} : C_{\text{as}}[\mathtt{pc}]$ *and* $\models C_{\text{as}} : \Psi$, *then* $(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathfrak{p}, \mathfrak{q})} A$.

Then, in Lemma 4, we show that our logic ensures the simulations.

Lemma 4 (Logic Ensures Simulation).

- $\Psi \vdash \{(\mathfrak{p}, \mathfrak{q})\}\, \mathtt{f} : \mathbb{I} \implies \Psi \models \{(\mathfrak{p}, \mathfrak{q})\}\, \mathtt{f} : \mathbb{I}$;
- $\vdash C_{\text{as}} : \Psi \Longrightarrow \models C_{\text{as}} : \Psi$;
- $\Psi \vdash C_{\text{as}} : \Omega \Longrightarrow \Psi \models C_{\text{as}} : \Omega$.

Next, we give Lemma 7, which says that the simulation for functions implies the primitive correctness. We define a whole program simulation in Definition 8 and divide the proof of Lemma 7 into two steps. First, we prove that the simulation for functions implies the whole program simulation in Lemma 5. Second, we prove that the whole program simulation implies the refinement relation between low- and high-level programs in Lemma 6.

Definition 8 (Whole Program Simulation). *Whenever $P \leqslant^i \mathbb{P}$ holds, the followings are true*:

1) *if $P ::\overset{\tau}{\Longrightarrow} P'$, then*:

- $\exists j < i.\, P' \leqslant^j \mathbb{P}$; *or*
- $\exists j, \mathbb{P}'.\, \mathbb{P} :\overset{\tau}{\Longrightarrow}{}^{+} \mathbb{P}'$, *and* $P' \leqslant^j \mathbb{P}'$;

2) *if $P ::\overset{e}{\Longrightarrow} P'$, then $\exists j, \mathbb{P}'.\, \mathbb{P} :\overset{e}{\Longrightarrow}{}^{+} \mathbb{P}'$, and $P \leqslant^j \mathbb{P}$*;

3) *if $P ::\overset{\tau}{\Longrightarrow}$ **abort**, then $\mathbb{P} :\overset{\tau}{\Longrightarrow}{}^{+}$ **abort***.

Lemma 5. *If $\Psi \models C_{as} : \Omega$, $S \sim \mathbb{S}$, $\mathsf{ProgSafe}((C, \Omega), \mathbb{S})$ and $\mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})$, then there exists $i \in Index$, such that $(C \uplus C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \leqslant^i ((C, \Omega), \mathbb{S})$.*

Lemma 6. $P \leqslant^i \mathbb{P} \implies P \subseteq \mathbb{P}$.

Lemma 7 (Simulation Implies Primitive Correctness).

$$\Psi \models C_{\text{as}} : \Omega \Longrightarrow C_{\text{as}} \sqsubseteq \Omega.$$

Proof. We unfold $C_{\text{as}} \sqsubseteq \Omega$ according to its definition in Definition 1, and finish the proof by applying Lemma 5 and Lemma 6. □

Theorem 1 (Logic Soundness).

$$\Psi \vdash C_{\text{as}} : \Omega \Longrightarrow C_{\text{as}} \sqsubseteq \Omega.$$

Proof. The soundness proof can be done by applying Lemma 4 and Lemma 7. □

5 Verifying Context Switch Routine

We apply our program logic to verify that a context switch routine implemented in SPARCv8, which saves the current task's context and restores the new task's context, contextually refines the switch primitive defined in Subsection 4.1. Fig.20 shows the structure of the context switch routine that we proved.

- `SwitchEntry` is the entry of the context switch routine. It saves the local and in registers of the current window into the current task's stack in memory, and calls `reg_save` to save the other registers into the current task's TCB.

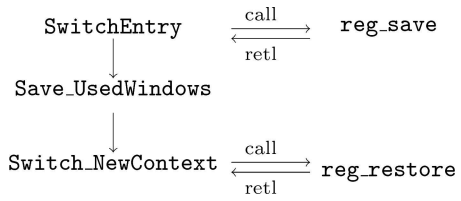


Fig.20. Structure of context switch routine.

- `Save_UsedWindows` saves the register windows (except the current one) into the current task's stack in memory.
- `Switch_NewContext` restores the general registers from the new task's TCB (by calling `reg_restore`) and its stack in memory respectively. Then it sets the new task as the current one.

The main complexity of the verification lies in the code that manages the register window. To save all the used register windows, `Save_UsedWindows` repetitively restores the next window into general registers (as the current window) and then saves them into memory, until all the windows are saved.

Specification. Below we give the pre- and post-conditions (a_{pre} and a_{post}) of the verified module respectively.

$$\begin{aligned}
&a_{\text{pre}}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) \ ::= \\
&\quad \mathsf{Env}(\mathit{env}) * (\mathsf{TaskNew} \mapsto (\mathsf{t}_n, 0)) * \blacklozenge(10) * \\
&\qquad \mathsf{CurT}(\mathsf{t}_c, _, \mathit{env}, \mathcal{K}_c) * \mathsf{RdyT}(\mathsf{t}_n, \mathit{nst}, \mathcal{K}_n) * (\!|\, \mathsf{switch}(\mathrm{nil}) \,|\!) \\
&a_{\text{post}}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) \ ::= \\
&\quad \exists\, \mathit{env}', \mathcal{K}'.\, \mathsf{Env}(\mathit{env}') * (\mathsf{TaskNew} \mapsto (\mathsf{t}_n, 0)) * \\
&\qquad (\mathsf{CurT}(\mathsf{t}_n, \mathit{nst}, \mathit{env}', \mathcal{K}') \wedge \mathsf{p_env}(\mathit{env}') = \mathit{nst}) * \\
&\qquad\quad \mathsf{RdyT}(\mathsf{t}_c, \mathsf{p_env}(\mathit{env}), \mathcal{K}_c) * (\!|\, \bot \,|\!)
\end{aligned}$$

Each of them takes six arguments, the ID of the current task t_c, the ID of the new task t_n, the values *env* of general registers and other register windows saving contexts, the new task's context *nst* to be restored, the current task's thread local state $\mathcal{K}_c$ and the new task's thread local state $\mathcal{K}_n$. In the specification, we use $\mathsf{Env}(\mathit{env})$ to specify the values of general registers and the register windows. We describe the state of the current task using $\mathsf{CurT}(\mathsf{t}_c, \_, \mathit{env}, \mathcal{K}_c)$. It describes the states of the current task t_c in the low and high level, and their state relation. Similarly, $\mathsf{RdyT}(\mathsf{t}_n, \mathit{nst}, \mathcal{K}_n)$ describes the states of the new task t_n in the low and high level, and their state relation. The memory location TaskNew records the identifier of the new

task. $\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0)$, where $l \Mapsto v$ is defined as $(l \mapsto v) * (l \rightarrowtail v)$, denotes that TaskNew saves $(\mathsf{t}_n, 0)$ in both the low- and high-level memory.

The precondition takes 10 tokens ($\blacklozenge(10)$). As we have explained, verifying `call` and `jmp` instructions will consume a token. Therefore, verifying calling functions `reg_save` and `reg_restore` will both consume a token. And `Save_Usedwindows`, which saves the context of each previous window into memory repetitively until the invalid one, will execute at most eight times, because the upper bound of the number of windows is eight. Therefore, 10 tokens are sufficient (two for `reg_save` and `reg_restore`, and eight for `Save_Usedwindows`).

If we compare a_{pre} and a_{post}, we can see that t_n becomes the current task ($\mathsf{CurT}(\mathsf{t}_n, nst, env', \mathcal{K}')$), and its general registers and stack, specified by $\mathsf{Env}(env')$, are loaded from the saved context nst (i.e., $\mathsf{p\_env}(env') = nst$). Here $\mathsf{p\_env}(env')$ refers to the part of the environment that we want to save or restore as context. Correspondingly, t_c becomes a non-current-thread, and part of its environment env at the entry of the context switch is saved, as specified by $\mathsf{RdyT}(\mathsf{t}_c, \mathsf{p\_env}(env), \mathcal{K}_c)$. The execution of switch should be done in the final state. We use $\mathcal{K}'$ to represent the thread local state of t_n instead of $\mathcal{K}_n$ in the final state, since the execution of switch will modify the program counters in $\mathcal{K}_n$.

Proof Outline. We show how to use our relational program logic defined in Fig.19 to verify the correctness of the context switch routine. We first instantiate the set of abstract assembly primitives (2) and the code heap specification (3) below:

$$\Omega ::= \{\texttt{SwitchEntry} \rightsquigarrow \mathsf{switch}\}. \tag{2}$$

$$\begin{aligned}\Psi ::= \{&\texttt{SwitchEntry} \rightsquigarrow (a_{\text{pre}}, a_{\text{post}}),\\ &\texttt{reg\_save} \rightsquigarrow (\mathtt{fp}_{rs}, \mathtt{fq}_{rs}),\\ &\texttt{reg\_restore} \rightsquigarrow (\mathtt{fp}_{rr}, \mathtt{fq}_{rr}),\\ &\texttt{Save\_Usedwindows} \rightsquigarrow (\mathtt{fp}_{su}, \mathtt{fq}_{su}),\\ &\texttt{Switch\_NewContext} \rightsquigarrow (\mathtt{fp}_{sn}, \mathtt{fq}_{sn})\}.\end{aligned} \tag{3}$$

The set of abstract assembly primitives Ω contains only one abstract assembly primitive switch. The code heap specification Ψ contains the specifications of each code block. We use $(\mathtt{fp}_{rs}, \mathtt{fq}_{rs})$, $(\mathtt{fp}_{rr}, \mathtt{fq}_{rr})$, $(\mathtt{fp}_{su}, \mathtt{fq}_{su})$ and $(\mathtt{fp}_{sn}, \mathtt{fq}_{sn})$ to represent the specifications of `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext` respectively. Since the post-condition in our logic specifies the state when the current function returns, the specification of `SwitchEntry` is $(a_{\text{pre}}, a_{\text{post}})$.

First, we prove that the specification of the context switch routine is well-defined in Lemma 8.

Lemma 8. $\mathsf{wdSpec}(a_{\text{pre}}, a_{\text{post}}, \mathsf{switch})$.

Proof. It is proved by the definition of wdSpec (in Definition 2). □

We use $C_{\texttt{switch}}$ to represent the code heap storing the code of the context switch routine, which includes the code blocks `SwitchEntry`, `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext`. We prove that $C_{\texttt{switch}}$ is well-defined in Lemma 9.

Lemma 9. $\vdash C_{\texttt{switch}} : \Psi$.

Proof. From the WfInt rule in Fig.19, we need to prove that for any ι_1, ι_2, ι_3, ι_4 and ι_5, the followings hold ($\mathtt{f}_{se}$, $\mathtt{f}_{rs}$, $\mathtt{f}_{rr}$, $\mathtt{f}_{su}$ and $\mathtt{f}_{sn}$ represent the starting labels of `SwitchEntry`, `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext` respectively):

- $\Psi \vdash \{(a_{\text{pre}}\ \iota_1, a_{\text{post}}\ \iota_1)\}\ \mathtt{f}_{se} : C_{\texttt{switch}}[\mathtt{f}_{se}]$;
- $\Psi \vdash \{(\mathtt{fp}_{rs}\ \iota_2, \mathtt{fq}_{rs}\ \iota_2)\}\ \mathtt{f}_{rs} : C_{\texttt{switch}}[\mathtt{f}_{rs}]$;
- $\Psi \vdash \{(\mathtt{fp}_{rr}\ \iota_3, \mathtt{fq}_{rr}\ \iota_3)\}\ \mathtt{f}_{rr} : C_{\texttt{switch}}[\mathtt{f}_{rr}]$;
- $\Psi \vdash \{(\mathtt{fp}_{su}\ \iota_4, \mathtt{fq}_{su}\ \iota_4)\}\ \mathtt{f}_{su} : C_{\texttt{switch}}[\mathtt{f}_{su}]$;
- $\Psi \vdash \{(\mathtt{fp}_{sn}\ \iota_5, \mathtt{fq}_{sn}\ \iota_5)\}\ \mathtt{f}_{sn} : C_{\texttt{switch}}[\mathtt{f}_{sn}]$.

The correctness proof of each code block can be done by applying the inference rules for instruction sequences shown in Fig.19. We can choose any place to apply the ABSCSQ rule to execute switch. Here, we apply the ABSCSQ rule in verifying `Switch_NewContext`, when the context switch routine returns, as shown in Fig.21. In Fig.21, we use the solid circle to represent the point applying the ABSCSQ rule, and after the execution of switch, the state relation, defined in Subsection 4.3 and represented as the solid lines in Fig.21, between the low level and the high level can be reestablished. □

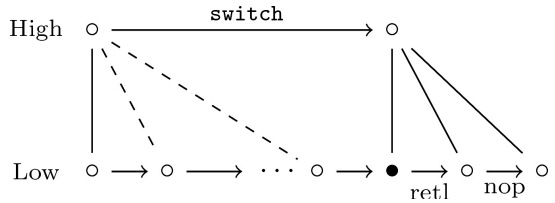


Fig.21. Point doing refinement reasoning.

Theorem 2 (Context Switch Routine Correctness). $\Psi \vdash C_{\texttt{switch}} : \Omega$.

Proof. The proof follows the WfPrim rule in Fig.19, and Lemma 8 and Lemma 9. □

This part of work has not been mechanized in Coq. In our conference paper [4], we show that we apply our

logic for partial correctness to verify the main body of the context switch routine in a realistic embedded OS kernel for aerospace crafts, which consists of around 250 lines of SPARCv8 code, by 6 690 lines of Coq proof scripts. Here, the context switch routine verified by applying our relational program logic is a simplified version of such context switch routine, which omits some details like judging whether the current thread is a valid one. Verifying that each code block is well-defined using the inference rules in our new logic is not different from the previous work. The additional proof efforts include: 1) proving that the specification of context switch routine is well-defined (shown in Lemma 8); 2) applying the ABSCSQ rule to execute the switch primitive and proving that the state relation between low- and high-level programs can be reestablished when the context switch routine returns (as noted in the proof of Lemma 9).

6 Related Work

There has been much work on assembly or machine code verification. Most of them do not support function calls or simply treat function calls in the continuation-passing style where return addresses are viewed as first-class code pointers [10–16]. SCAP [5] supports assembly code verification with various stack-based control abstractions, including function calls and returns. We follow the same idea here. However, SCAP gives a syntactic-based soundness proof by establishing the preservation of the syntactic judgment, which makes it difficult to interact with other modules verified in different logic. Since our goal is to verify inline assembly and link the verified code with the verified C programs, we give a direct-style semantic model of the logic judgments. And it allows us to extend our program logic in conference version [4] to support verifying contextual refinement without many challenges. Also SCAP is based on a simplified subset of assembly instructions, while our work is focused on a realistically modeled subset of SPARCv8 instructions.

In terms of the support of realistic instruction sets, previous work on proof-carrying code (PCC) and typed assembly language (TAL) mostly supports subsets of x86. Myreen and Gordon's work [17] presents a framework for ARM verification based on a realistic model (but it does not support function calls and returns).

As part of the Foundational Proof-Carrying Code (FPCC) project [11], Tan and Appel presented a program logic $\mathcal{L}_c$ for reasoning about control flow in assembly code [16]. Although $\mathcal{L}_c$ is implemented on top of the SPARC machine language, the underlying logic is a type system instead of a full-blown program logic for functional correctness. It reasons about functions in the continuation-passing style. Also handling SPARC features such as delayed writes or delayed control transfers is not the focus of $\mathcal{L}_c$. There has been work on mechanized semantics of the SPARCv8 ISA. Hou *et al.* [18] modelled the SPARCv8 ISA in Isabelle/HOL, and tested their formal model against LENON3 simulation board, which is a synthesisable VHDL model of a 32-bit processor compliant with the SPARCv8 architecture, through more than 100 000 instruction instances. Wang *et al.* [7] formalized its semantics in Coq. Our operational semantics of SPARCv8 follows Wang *et al.* [7] However neither Wang *et al.* [7] nor we validate the formalization against actual hardware, and this remains as future work.

Ni *et al.* [19] verified a context switch module of 19 lines in x86 code to showcase the support of embedded code pointers (ECP) in XCAP [15]. We use our program logic to verify the contextual refinement between a context switch routine in SPARCv8 and switch primitive. The context switch routine implemented in SPARCv8 that we verified is more complicated than that implemented in x86, because of the requirement to save the contexts stored in register window in memory.

Yang and Hawblitzel [20] verified Verve, an x86 implementation of an experimental operating system. Verve has two levels, the high-level TAL code and the low-level "Nucleus" that provides primitive access to hardware and memory. The Nucleus code is verified automatically using the Z3 SMT solver, while the goal of our work is to generate machine checkable proofs. Another key difference is the use of different ISAs. Here we give details to verify specific features of SPARCv8 programs.

There have been many techniques and tools proposed for automated program verification (e.g., [21,22]). It is possible to adapt them to verify SPARCv8 code. We propose a new program logic and do the verification in Coq mainly because the work is part of a big project for a fully certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8. We already have a program logic implemented in Coq for C programs, which allows us to verify C code with Coq proofs. Therefore we want to have a program logic for SPARCv8 so that it can be linked with the correctness proof of C and can generate machine-checkable Coq proofs too. That is, many of the automated verification techniques can be applied to reduce the manual

efforts to write Coq proofs, which we would like to study in the future work.

7 Conclusions

We defined a relational program logic for SPARCv8. Our logic is based on a realistic semantics model and supports the main features of SPARCv8, including delayed control transfer, delayed writes, and register windows. It also supports modular reasoning of function calls in a direct-style and refinement verification. We applied our logic to verify that a context switch routine implemented in SPARCv8 contextually refines the switch primitive for task switching.

Our current work does not consider interrupts in the machine model. We would like to extend it for concurrency verification and finish S1 shown in Fig.2 in the future. S1 shown in Fig.2 says that the compilation ensures that the Pseudo-SPARCv8 code refines the C program with abstract assembly primitives.

References

[1] Xu F, Fu M, Feng X, Zhang X, Zhang H, Li Z. A practical verification framework for preemptive OS kernels. In *Proc. the 28th International Conference*, July 2016, pp.59-79.

[2] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS Kernel. In *Proc. the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009, pp.207-220.

[3] Gu R, Koenig J, Ramananandro T, Shao Z, Wu X N, Weng S C, Zhang H, Guo Y. Deep specifications and certified abstraction layers. In *Proc. the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2015, pp.59-5608.

[4] Zha J, Feng X, Qiao L. Modular verification of SPARCv8 code. In *Proc. the 16th Asian Symposium on Programming Languages and Systems*, December 2018, pp.245-263.

[5] Feng X, Shao Z, Vaynberg A, Xiang S, Ni Z. Modular verification of assembly code with stack-based control abstractions. In *Proc. the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006, pp.401-414.

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

[7] Wang J, Fu M, Qiao L, Feng X. Formalizing SPARCv8 instruction set architecture in Coq. In *Proc. the 3rd International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, Oct. 2017, pp.300-316.

[8] Reynolds J. Separation logic: A logic for shared mutable data structures. In *Proc. the 17th IEEE Symposium on Logic in Computer Science*, July 2002, pp.55-74.

[9] Liang H, Feng X, Shao Z. Compositional verification of termination-preserving refinement of concurrent programs. In *Proc. the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science*, July 2014, Article No. 65.

[10] Necula G C, Lee P. Safe kernel extensions without run-time checking. In *Proc. the 2nd USENIX Symp. Operating System Design and Implementation*, October 1996, pp.229-243.

[11] Appel A W. Foundational proof-carrying code. In *Proc. the 16th Annual IEEE Symposium on Logic in Computer Science*, June 1998, pp.247-256.

[12] Morrisett G, Crary K, Glew N, Grossman D, Samuels R, Smith F, Walker D, Weirich S, Zdancewic S. TALx86: A realistic typed assembly language. In *Proc. the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, May 1996, pp.25-35.

[13] Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language. In *Proc. the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1998, pp.85-97.

[14] Yu D, Nadeem A H, Shao Z. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 2004, 50(1/2/3): 101-127.

[15] Ni Z, Shao Z. Certified assembly programming with embedded code pointers. In *Proc. the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006, pp.320-333.

[16] Tan G, Appel A W. A compositional logic for control flow. In *Proc. the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, Jan. 2006, pp.80-94.

[17] Myreen M O, Gordon M J. Hoare logic for realistically modelled machine code. In *Proc. the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, March 2007, pp.568-582.

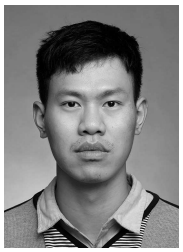
[18] Hou Z, Sanán D, Tiu A, Liu Y, Hoa K C. An executable formalisation of the SPARCv8 instruction set architecture: A case study for the LEON3 processor. In *Proc. the 21st International Symposium on Formal Methods*, November 2016, pp.388-405.

[19] Ni Z, Yu D, Shao Z. Using XCAP to certify realistic systems code: Machine context management. In *Proc. the 20th International Conference on Theorem Proving in Higher Order Logics,* Sept. 2007, pp.189-206.

[20] Yang J, Hawblitzel C. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proc. the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010, pp.99-110.

[21] Berdine J, Calcagno C, O'Hearn P W. Symbolic execution with separation logic. In *Proc. the 3rd Asian Symposium on Programming Languages and Systems*, November 2005, pp.52-68.

[22] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. the 4th International Symposium on Formal Methods for Components and Objects*, November 2005, pp.115-137.



Jun-Peng Zha is a Ph.D. candidate in the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his M.S. degree in computer science from the University of Science and Technology of China, Hefei, in 2019, and his B.S. degree in software engineering from Shandong University at Weihai in 2016. His research interests are in programming languages and formal methods, with a focus on the verification of compiler and low-level assembly code.

Xin-Yu Feng is a professor in Department of Computer Science and Technology at Nanjing University, Nanjing. He obtained his Ph.D. degree in computer science from Yale University, New Haven, in 2007, and his M.E. and B.S. degrees in computer science from Nanjing University, Nanjing, in 2002 and 1999, respectively. His research interests are on theories of programming languages and formal program verification.

Lei Qiao is a professor in the Center of On-Board Computer and Electronics at Beijing Institute of Control Engineering, Beijing. He received his Ph.D. degree in computer science from the University of Science and Technology of China, Hefei, in 2007. His research interests are in operating system design and formal verification.