Sound Semantics of a High-Level Language with Interprocessor Interrupts

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Hristo Pentchev
pentchev@wjpserver.cs.uni-saarland.de

Saarbrücken, Januar 2016

Tag des Kolloquiums:	13.06.2016
Dekan:	Univ.-Prof. Dr. Frank-Olaf Schreyer
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Jörg Siekmann
1.Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2.Berichterstatter:	Prof. Dr. Bernhard Beckert
Akademischer Mitarbeiter:	Dr. Johannes Hoffart

Abstract

Pervasive formal verification guarantees highest reliability of complex multi-core computer systems. This is required especially for safety critical applications in automotive, medical and military technologies. A crucial part of formal verification is the profound understanding of all system layers and the correct specification of their computational models and the interaction between software and hardware. The underlying architecture and the semantics of the higher-level programs cannot be considered in isolation. In particular, when the program execution relies on specific hardware features, these features have to be integrated into the computational model of the programing language.

In this thesis, we present an integration approach for interprocessor interrupts provided by multi-core architectures in the pervasive verification of system software written in C. We define an extension to the semantics of a high-level language, which considers interprocessor interrupts. We prove simulation between a multi-core hardware model and the high-level semantics with interrupts. In this simulation, we assume interrupts to occur on the boundary between statements. We justify that assumption by stating and proving an order reduction theorem, to reorder the interprocessor interrupt service routines to dedicated consistency points.

Kurzzusammenfassung

Formale durchdringende Verifikation garantiert die höchste Zuverlässigkeit von komplexen Multi-Core Computersystemen. Das ist insbesondere bei sicherheitskritischen Anwendungen in der Automobil-, Medizin- und Militärtechnik unerlässlich. Ein wesentlicher Bestandteil der formalen Verifikation ist das tiefgründige Verständnis aller Ebenen des Modell-Stacks und die korrekte Spezifikation deren Rechenmodelle und des Zusammenspiels zwischen Software and Hardware. Die zugrunde liegende Hardwarearchitektur und die Semantik der abstrakten Programmiersprache können nicht isoliert von einander betrachtet und analysiert werden. Insbesondere dann, wenn sich die Programmausführung auf Eigenschaften der Hardware stützt, müssen diese Eigenschaften in das Rechenmodell der Programmiersprache integriert werden.

In dieser Arbeit präsentieren wir einen Integrationsansatz für Inter-Prozessor-Interrupts von Multi-Core-Architekturen in die durchdringende Verifikation der Systemsoftware geschrieben in C. Wir definieren eine Erweiterung der Semantik einer Hochsprache, die Inter-Prozessor-Interrupts berücksichtigt. Wir beweisen Simulation zwischen einem Multi-Core-Hardware-Modell und der High-Level-Semantik mit Interrupts. In dieser Simulation nehmen wir an, dass Interrupts an der Grenze zwischen Anweisungen auftreten. Wir rechtfertigen diese Annahme durch die Definition und den Beweis eines Reduktionstheorem, um die Interprozessor-Interrupt-Service-Routinen zu dedizierten Konsistenz-Punkten zu reordern.

Acknowledgements:

First I want to express my gratitude to Prof. Dr. Wolfgang J. Paul for supervising my work, his wise advices, and all the valuable discussions about research and life.

I appreciate very much the cooperation within the Hyper-V verification group of the Verisoft XT project and especially the prolific brainstorming sessions with Mikhail Kovalev, Christoph Baumann and Sabine Schmlatz.

Last but not least, I thank my family for supporting me during the years.

Contents

Chapter 1

Introduction

Modern computer systems have changed our lives completely. They provide a huge pile of services, which not only make everyday activities easier but also create a lot of new possibilities. It is amazing to observe the enormous digitalization in all areas of our society. Smartphones have become an irreplaceable companion for almost two billion people [Sta]. Digital technologies are disrupting conventional businesses in all industry sectors from publishing to banking. All this is only possible due to the rapid evolution of hardware and the simultaneous development of software. The developers of micro-processors put more and more computing cores on a chip. This requires parallel software, to make use of the available computational power. Parallelization however dramatically increases the complexity of the system. A Fraunhofer developer survey in 2010 has shown that less than twenty percent of the software developers claim to have good expertise in multicore programing [Heb10]. These facts lead to the indisputable conclusion, that modern computer systems are faulty.

Apparently a blue screen on a personal PC is not tragical, but software and hardware failures have already caused hundreds of millions of losses (e.g. the result of Intel's well-known pentium bug [Pra95]). In worst cases software failures are even responsible for casualties (e.g. the incidents with the radiation therapy machine Therac-25 [LT93].)

And since modern complex computer systems are embedded in safety critical applications in automotive, medical and military technologies it is obvious that the presence of failures must be eliminated or at least minimized.

One way to increase the safety and reliability of systems is testing. It is widely established in the industry and implements sophisticated methods. Unfortunately the absence of failures in a tested system can only be claimed to the limited set of simulated system executions. In a parallel system with arbitrary interleaving the set of all possible executions is enormous. Thus even extensive tests of complex systems can not guarantee correctness.

Another approach is to apply formal methods and verification tools to provide mathematically based correctness proof. In this approach the systems are analyzed and an abstract model is specified. Then all possible executions of the system are proven to obey the formal specification. That way the correctness of the system is verified against the specification. Two major factors for the consistency of formal verification are the correctness of the specified abstract model and the soundness of the verification tool. The latter means that if a property is evidenced by the tool, than it indeed holds for all executions. In other words, formal methods rely on a precise computational model. If the computational model of the verification tool is wrong, verification results are faulty. If we consider solely software and code-verification tools, then this computational model is defined by the semantics of the programming language. With code-verification tools, one can prove the correctness of programs, but this is not enough to state correctness of a system containing soft- and hardware. We need to examine the behavior of the whole system including soft- and hardware and to define pervasive theory. For instance in low-level system software soft- and hardware are closely coupled, which makes writing system software a challenging task. Soft- and hardware are operating together and hardware characteristics influence the software execution massively. Thus in order to program reliable system software, one has to consider soft- and hardware in a mixed computational model. Moreover the verification of properties of system software is impossible in the absence of a programing language semantics, that incorporate hardware (e.g. interrupts and memory management unit).

System verification was in the focus of the Verisoft XT project [Ver]. One of the goals of the project was to prove the correctness the Microsoft's Hyper-V^{TM} Hypervisor. Hyper-V^{TM} is a hypervisor, for X86-64 virtualization. As most system software in general also Hyper-V^{TM} is written mostly in C. For that purpose Microsoft developed during Verisoft XT with the help of the project partners the concurrent C verification tool VCC. In the scope of the project big portions of Hyper-V^{TM} were verified in VCC. The soundness proof of VCC was sketched but not completed. The development of a sound system verification method and the definitions of the missing semantics continued after the project at the chair of Professor Paul. A multi-core model stack was defined to cover the different levels of abstraction of the system. It contains a hardware model of a multi-core MIPS machine, an ownership based order reduction theorem [Bau14], extended semantics for the C Intermediate Language (C-IL) with Ghost [Sch13], Mixed Low- and High Level Programming Language Semantics [Sha12] and TLB virtualization method [Kov13]. The results are described in several PhD theses and an attempt to summarize everything is being described in the lecture notes of the current Multicore System Architecture Lecture in [PBLS16].

The integration of inter process communication based on inter processor interrupts (IPIs) in the model stack is handled in this work. We define a semantic model of C-IL with interrupts and justify this model against executions of a multi-core MIPS machine with local APICs. With this model we justify the VCC verification of an expressive generic interprocess communication protocol . During the Verisoft XT project the approach was applied, without the justification presented in this thesis, in an academic hypervisor and in Microsoft Hyper-V$^{\text{TM}}$ (see the appendix in Chapter 8).

1.1 Related Work

The only work we are aware of about C semantics with interrupts is presented in [PBLS16]. Previous less general versions of this semantics were presented in [Alk09], [Sta10], [PSS12], and [Sha12].

In [PBLS16] the authors define semantics for C + assembly + interrupts. There interrupts are disabled during the execution of C portions and are only visible during the execution of the assembly portions of the program. The assembly code is pooling for the interrupt. This is sufficient for a special case of interrupt handling. In the work presented in this thesis we define a more general approach in which interrupts are integrated in the C semantics and visible on the border of C statements.

For transferring the properties from C to ISA we use the model stack. A similar model stack but so far without interrupts is used in the formally verified work reported in [App12]. Furthermore the work presented there is based on a sequential compiler correctness and it is not transferred to the concurrent case.

In order to prove simulation we need to reorder ISA executions in a suitable schedule. The starting point for this order reduction theorem lies in [Bau14]. The generic order reduction theorem presented there however does not consider interrupts.

1.2 Outline of the Thesis

In Chapter 2 we present $MIPS_P$ - a simplified multiprocessor $MIPS$ instruction set architecture (ISA) machine. We define the $MIPS_P$ configuration, present its components and the overall $MIPS_P$ transition function in order to define execution sequences of the machine. In Chapter 3 we present an ownership based order reduction theorem to justify the reordering of arbitrary interleaved $MIPS_P$ executions into schedules suitable for compiler correctness theorem application. We define an ownership model, instantiate with $MIPS_P$ a generic model called COSMOS and apply the COSMOS order reduction theorem. In Chapter 4 we introduce interrupt threads and refine our ownership model. We state and prove an order reduction theorem to reorder handler executions such that interrupts occur at consistency points..

In Chapter 5 semantics for a programming language similar to C - the C Intermediate Language (C-IL). We state a concurrent compiler correctness theorem in the absence of interrupts similar to [Sha12] and [Kov13]. In Chapter 6 we extend *C-IL* with a component mirroring the local APIC state. We state and prove a concurrent simulation theorem between $MIPS_P$ executions and executions of the extended *C-IL*. This work is concluded in Chapter 7.

1.3 Notation

Definition 1.1 ▶ Hilbert-Choice-Operator

To choose an arbitrary element of a given set A we use the Hilbert-choice-operator ε.

$$\varepsilon A \in A$$

If the given set consists of a single element, then

$$\varepsilon\{x\} = x \ .$$

Definition 1.2 ▶ Natural Numbers

We denote by $\mathbb{N}$ the set of natural numbers (with zero).

$$\mathbb{N} \overset{def}{=} \{0, 1, 2, \ldots\}$$

We denote by $\mathbb{N}^+$ the set of positive natural numbers.

$$\mathbb{N}^+ \overset{def}{=} \{1, 2, \ldots\}$$

Definition 1.3 ▶ Boolean Values

We denote by $\mathbb{B}$ set of Boolean values.

$$\mathbb{B} \overset{def}{=} \{0, 1\}$$

We also use the term bit to refer to a Boolean value.

Definition 1.4 ▶ Power Set

By 2^A we denote the power set of a given set A, i.e. the set of all subsets of A.

$$2^A \overset{def}{=} \{B \mid B \subseteq A\}$$

Definition 1.5 ▶ Interval of Natural Numbers

We denote by $[i:j]$ the interval of natural numbers from i to j.

$$[i:j] \in 2^{\mathbb{N}} \overset{def}{=} \begin{cases} \emptyset & \textit{if } i > j \\ \{i, i+1, \ldots, j\} & \textit{if } i \leq j \end{cases}$$

Definition 1.6 ▶ Finite Sequences

We denote empty sequences by ε. We index the elements in a finite sequence β of n elements from given set A from left to the right and start indices at 0.

$$\beta \overset{def}{=} \beta_0\beta_1\beta_2 \ldots \beta_{n-1}$$

We denote single elements of the array by β_i or $\beta[i]$, and the subsequence with elements from β_i to β_j by $\beta[i:j]$.

$$\beta[i:j] \overset{def}{=} \begin{cases} \varepsilon & \textit{if } i > j \\ \beta_i & \textit{if } i = j \\ \beta_i\beta[i+1:j] & \textit{otherwise} \end{cases}$$

We denote by $|\beta|$ the number of elements in β (i.e. the length of β).

$$|\beta| \overset{def}{=} n$$

The set of all sequences of elements from A with length n we denote by A^n.

$$A^n \overset{def}{=} \{\beta \mid (|\beta| = n) \wedge \forall i \in [0 : n-1].\ \beta_i \in A\}$$

By A^* we denote the set of arbitrary long sequences.

A finite sequence of boolean values (bits) b of length n we call a bit vector.

◀ **Definition** 1.7
Bit-Strings

$$b \in \mathbb{B}^n$$

For a given bit-string b of length n we define its value as a natural number by

$$\langle b \rangle \overset{def}{=} \sum_{i=0}^{n-1} b_i \cdot 2^i \ .$$

The binary representation of a given natural number k as a n-bit long string, where $k \in [0 : 2^n - 1]$, we define by

$$bin_n(k) \in \mathbb{B}^n \overset{def}{=} \varepsilon\{b \mid (b \in \mathbb{B}^n) \wedge (\langle b \rangle = k)\} \ .$$

A record is a tuple of named components and their types. We define a record R with components a and b of types A and B by

◀ **Definition** 1.8
Records

$$R \overset{def}{=} [a \in A, b \in B]$$

and access the elements by $R.a$ and $R.b$. We update the components of a record $r \in R$ by

$$r' = r[a \mapsto a', b \mapsto b']$$

Chapter 2

Abstract Hardware Model

In this work we present a stripped-down concise model, that on the one hand is functional enough to demonstrate our goals and on the other hand is simple enough not to shift the focus of the thesis.

$MIPS_P$ is a simplified multiprocessor $MIPS$ instruction set architecture (ISA) machine. Similar models have been specified in the scope of the Verisoft XT project and the subsequent research.

Ulan Degenbaev gives in [Deg11] a formal model of $x64$ ISA.

In [Sch13] Sabine Schmaltz defines a multiprocessor model $MIPS$ called $MIPS$-86. $MIPS$-86 is basically a multiprocessor version of the sequential processor model from [KMP14] extended with MMU, TLB, APIC and device models, which are motivated by and similar to $x86$ architectures.

We stay closest to the $MIPS$-86. The main simplifications compared to $MIPS$-86 affect devices and MMUs. We also consider that all processors have already been booted and are running. Readers interested in the omitted details can look them up in [Sch13].

$MIPS$-86 processors run in two modes, user mode with address translation and system mode without address translation. Memory management unit steps can only be made in user mode. We are interested only in properties of system code, that runs in untranslated mode. These properties concern only a small fraction of the MMU state, which allows us to use a strongly simplified model.

Significant parts of the simplifications are based on the results of joint work started in Verisoft XT. In [DPS09] Ulan Degenbaev, Wolfgang Paul and Norbert Schirmer give a sketch of cache, SB, and TLB reduction theorems and basic compiler consistency. In [Kov13] Mikhail Kovalev has shown that, following a programming discipline, store buffers are transparent to the hypervisor and can be taken away. He also provided proofs for memory virtualization covering the behavior of the MMU.

In $MIPS_P$ we omited devices since they do not interfere with IPIs. Although device signals are delivered to the core over the APIC, they do not influence the

IPIs because of their lower priority. Our model can be extended with devices without changing significantly any theorem or proof, that we present in this work.

We refine the specification of $MIPS$-86 concerning guest execution. $MIPS$-86 does not provide instructions for virtualization support and an intercept model, which are necessary for virtualization. We add the VMRUN instruction for virtualization support to the instruction set, but still do not provide detailed formal definitions for guest execution.

In the next section we provide a summary of the instruction set of our model. Then in Section 2.2 we define the configuration of a $MIPS_P$ machine. In the subsequent sections we present the semantics of the different $MIPS_P$ components and their local transition functions. Finally we define the overall transition function of the machine as a composition of the steps of its sub-components.

2.1 $MIPS_P$ Instruction Set

The $MIPS_P$ instruction set consists of instructions which we separate in three types.

- I-Type instructions operate on two registers and an immediate constant.
- R-Type instructions operate on three registers.
- J-Type instructions implement jumps in the program to a given address passed as an immediate constant.

Every instruction type has a specific layout. We define the instruction layouts in Table 2.1, Table 2.2 and Table 2.3. The opcode together with the function code[1] codes the operation. rs, rt and rd define register addresses. sa, imm and $iindex$ store the immediate constant operands.

Bits	31 ... 26	25 ... 21	20 ... 16	15 ... 0
Field Name	opcode	rs	rt	immediate constant imm

Table 2.1: I-Type Instruction Layout.

Bits	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
Field Name	opcode	rs	rt	rd	sa shift amount	fun function code

Table 2.2: R-Type Instruction Layout.

[1]The function code is relevant only for R-Type instructions.

Bits	31 . . . 26	25 . . . 0
Field Name	opcode	instruction index $iindex$

Table 2.3: J-Type Instruction Layout.

The set of $MIPS_P$ ISA instructions is defined in Table 2.4[2], Table 2.6 and Table 2.5. The tables contain the coding of the instructions, their assembler syntax and a coarse overview of their semantics. The full semantics of the given instructions we define later in Section 2.4.2.

opcode	Mnemonic	Assembler-Syntax	Effect
Data Transfer			
100 000	lb	lb rt rs imm	rt = sxt(m$_1$(rs + sxt(imm)))
100 001	lh	lh rt rs imm	rt = sxt(m$_2$(rs + sxt(imm)))
100 011	lw	lw rt rs imm	rt = m$_4$(rs + sxt(imm))
100 100	lbu	lbu rt rs imm	rt = zxt(m$_1$(rs + sxt(imm)))
100 101	lhu	lhu rt rs imm	rt = zxt(m$_2$(rs + sxt(imm)))
101 000	sb	sb rt rs imm	m$_1$(rs + sxt(imm)) = rt[7:0]
101 001	sh	sh rt rs imm	m$_2$(rs + sxt(imm)) = rt[15:0]
101 011	sw	sw rt rs imm	m$_4$(rs + sxt(imm)) = rt
Arithmetic, Logical Operation, Test-and-Set			
001 000	addi	addi rt rs imm	rt = rs + sxt(imm)
001 001	addiu	addiu rt rs imm	rt = rs + sxt(imm)
001 010	slti	slti rt rs imm	rt = (rs < sxt(imm) ? 1 : 0)
001 011	sltui	sltui rt rs imm	rt = (rs < zxt(imm) ? 1 : 0)
001 100	andi	andi rt rs imm	rt = rs $\wedge$ zxt(imm)
001 101	ori	ori rt rs imm	rt = rs $\vee$ zxt(imm)
001 110	xori	xori rt rs imm	rt = rs $\oplus$ zxt(imm)
001 111	lui	lui rt imm	rt = imm0^{16}
Branch			
000 001	bltz	bltz rs imm	pc = pc + (rs < 0 ? imm00 : 4)
000 001	bgez	bgez rs imm	pc = pc + (rs $\geq$ 0 ? imm00 : 4)
000 100	beq	beq rs rt imm	pc = pc + (rs = rt ? imm00 : 4)
000 101	bne	bne rs rt imm	pc = pc + (rs $\neq$ rt ? imm00 : 4)
000 110	blez	blez rs imm	pc = pc + (rs $\leq$ 0 ? imm00 : 4)
000 111	bgtz	bgtz rs imm	pc = pc + (rs > 0 ? imm00 : 4)

Table 2.4: I-Type Instructions of $MIPS$.

[2]To distinguish between branch instructions with the same opcode we additionally use the rt field.

opcode	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j $iindex$	pc = bin_{32}(pc+4)[31:28]iindex00

000 011	jal	jal $iindex$	R31 = pc + 4
			pc = bin_{32}(pc+4)[31:28]iindex00

Table 2.5: J-Type Instructions of $MIPS$.

2.1.1 VMRUN

The virtualization instruction VMRUN implements the context switch from hypervisor to guest. In our model VMRUN is an abstraction of the context switch in $X86$, which in reality consists of more than just a single VMRUN instruction. In $MIPS_P$ VMRUN saves the state of the hypervisor in the memory, restores the guest state and changes the mode of the processor from system to user mode. The formal specification of VMRUN and other virtualization extensions and their integration into the current $MIPS$ model is future work.

2.2 Configuration

Definition 2.1 ▶ Abstract Hardware Configuration/Abstracted machine state

A $MIPS_P$ configuration c of our abstract machine has the type C_M

$$C_M \overset{def}{=} [cpu \in Pid \rightarrow C_{CPU}, m \in C_{MEM}]$$

and contains:

- $c.cpu$ - a mapping from processor identifier to processor configuration, where $Pid = [0 : np - 1]$ and $np \in \mathbb{N}^+$ is a parameter defining the number of processors in the system, and
- $c.m$ - a global memory.

Definition 2.2 ▶ CPU Configuration

A processor configuration

$$C_{CPU} \overset{def}{=} [core \in C_{CORE}, tlb \in C_{TLB}, apic \in C_{APIC}]$$

consists of:

- a core, executing instructions,
- a translation lookaside buffer (TLB), caching address translations,
- an advanced programmable interrupt controller (APIC), sending and receiving interrupts.

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 000	sll	sll $rd\ rt\ sa$	rd = sll(rt,sa)
000000	000 010	srl	srl $rd\ rt\ sa$	rd = srl(rt,sa)
000000	000 011	sra	sra $rd\ rt\ sa$	rd = sra(rt,sa)
000000	000 100	sllv	sllv $rd\ rt\ rs$	rd = sll(rt,rs)
000000	000 110	srlv	srlv $rd\ rt\ rs$	rd = srl(rt,rs)
000000	000 111	srav	srav $rd\ rt\ rs$	rd = sra(rt,rs)
Arithmetic, Logical Operation				
000000	100 000	add	add $rd\ rs\ rt$	rd = rs + rt
000000	100 001	addu	addu $rd\ rs\ rt$	rd = rs + rt
000000	100 010	sub	sub $rd\ rs\ rt$	rd = rs − rt
000000	100 011	subu	subu $rd\ rs\ rt$	rd = rs − rt
000000	100 100	and	and $rd\ rs\ rt$	rd = rs $\wedge$ rt
000000	100 101	or	or $rd\ rs\ rt$	rd = rs $\vee$ rt
000000	100 110	xor	xor $rd\ rs\ rt$	rd = rs $\oplus$ rt
000000	100 111	nor	nor $rd\ rs\ rt$	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt $rd\ rs\ rt$	rd = (rs < rt ? 1 : 0)
000000	101 011	sltu	sltu $rd\ rs\ rt$	rd = (rs < rt ? 1 : 0)
Jumps				
000000	001 000	jr	jr rs	pc = rs
000000	001 001	jalr	jalr $rd\ rs$	rd = pc + 4 pc = rs
Synchronizing Memory Operations				
000000	111 111	rmw	rmw $rd\ rs\ rt$	rd' = m m' = (rd = m ? rt : m)
Virtualization Instructions				
000000	111 110	vmrun	vmrun	
TLB Instructions				
000000	111 101	flush	flush	flushes TLB

Coprocessor Instructions					
opcode	rs	fun	Mnemonic	Assembler-Syntax	Effect
010000	10000	011 000	eret	eret	Exception Return
010000	00100		movg2s	movg2s $rd\ rt$	spr[rd] := gpr[rt]
010000	00000		movs2g	movs2g $rd\ rt$	gpr[rt] := spr[rd]

Table 2.6: R-Type Instruction of $MIPS$.

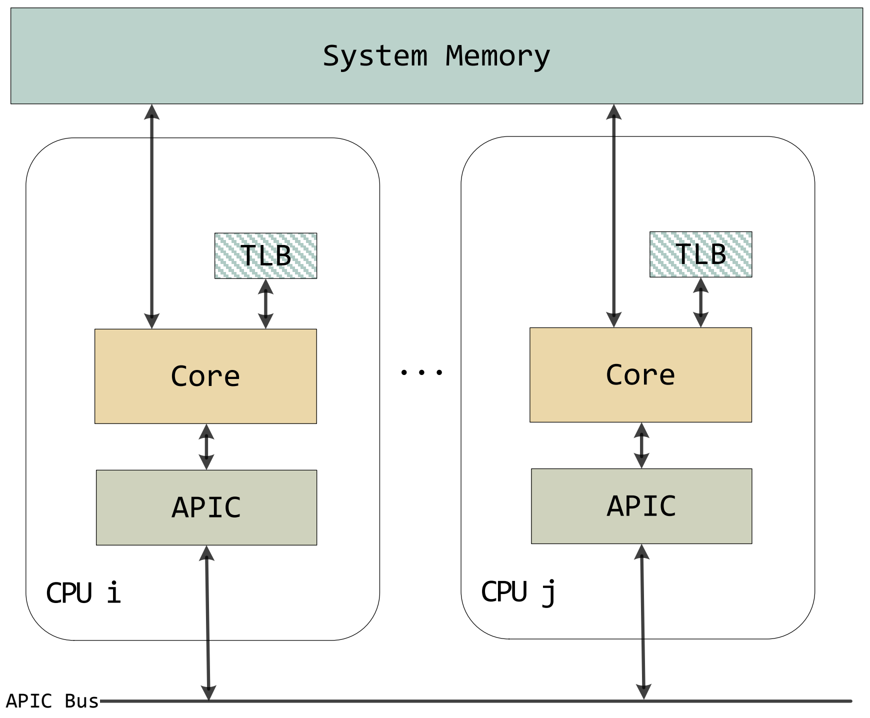

Figure 2.1: Abstract machine architecture

2.3 Memory

Definition 2.3 ▶ Memory Configuration

The global memory is a mapping of thirty two bit wide addresses to bytes.

$$C_{MEM} \overset{def}{=} \mathbb{B}^{32} \rightarrow \mathbb{B}^{8}$$

Definition 2.4 ▶ Reading Memory

The content of $n \in \mathbb{N}^{+}$ consecutive memory cells starting at address $a \in \mathbb{B}^{32}$ of a memory $m \in C_{MEM}$ is defined by the following notation.

$$m_n(a \in \mathbb{B}^{32}) \in \mathbb{B}^{8 \cdot n} \overset{def}{=} \begin{cases} m_{n-1}(a+1) \circ m(a) & n > 0 \\ \epsilon & n = 0 \end{cases}$$

Definition 2.5 ▶ Writing Memory

Changes of the memory content in memory m are applied by the function *write*

$$write(m \in C_{MEM}, a \in \mathbb{B}^{32}, v \in \mathbb{B}^{8^*}, c \in \mathbb{B}^{32} \cup \{\bot\}) \in C_{MEM} \overset{def}{=}$$

$$\begin{cases} m[addr \to byte_i(v)] & c = \bot \\ & \wedge\ addr \in [a : a + (|v|/8) - 1] \\ & \wedge\ i = \langle addr \rangle - \langle a \rangle \\ \\ m[addr \to byte_i(v)] & c \in \mathbb{B}^{32} \wedge v \in \mathbb{B}^{32} \\ & \wedge\ addr \in [a : a + 3] \\ & \wedge\ i = \langle addr \rangle - \langle a \rangle \\ & \wedge\ c = m_4(a) \\ \\ m & otherwise \end{cases}$$

where a is the address of the first byte to be written, v is the new value and c is compare-value in case of read-modify-write access.

◀ **Definition** 2.6 Memory Transition Function

The only changes of the memory state are originating from a memory write and for that reason the memory transition function δ^{mem} is defined completely by the memory write function.

$$\delta^{mem}(m \in C_{MEM}, a \in \mathbb{B}^{32}, v \in \mathbb{B}^{8^*}, c \in \mathbb{B}^{32} \cup \{\bot\}) \in C_{MEM} \overset{def}{=} write(m, a, v, c)$$

2.4 Processor Core

2.4.1 Processor Core Configuration

◀ **Definition** 2.7 Core Configuration

The processor core configuration C_{CORE} consists of:

- a program counter pc storing a 32 bit wide pointer to the next instruction,
- a general purpose register file gpr of 32 registers which are 32 bit wide,
- a special purpose register file spr of 32 registers which are 32 bit wide.

$$C_{CORE} \overset{def}{=} [pc \in \mathbb{B}^{32}, gpr \in \mathbb{B}^5 \to \mathbb{B}^{32}, spr \in \mathbb{B}^5 \to \mathbb{B}^{32}]$$

When the processor is making a step, we have basically two possibilities. It either executes the next instruction, or it performs a jump to the interrupt service routine (JISR). Before defining the transition function of the core we first consider these two cases. Later in Section 2.4.4 we present the core transition function.

Table 2.7: $MIPS$Special Purpose Registers.

Index	Alias	Usage
0	sr	status register (contains masks to enable/disable maskable interrupts)
1	esr	exception sr
2	eca	exception cause register
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data
7	mode	mode register $\in \{0^{31}1, 0^{32}\}$

2.4.2 Instruction Execution

In the following we introduce some additional notations and auxiliary functions which we need in the formal definition of the instruction execution.

2.4.2.1 Auxiliary Definitions for Instruction Decoding and Execution

Definition 2.8 ▶ Instruction Decoding

We define the following instruction predicates that define the instruction type depending on the opcode.

$$rtype(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} I[31:26] \in \{0^6, 010^4, 01^30^2\}$$

$$jtype(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} I[31:26] \in \{0^410, 0^411\}$$

$$itype(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \neg(rtype(I) \vee jtype(I))$$

The instruction opcode (together with the function code for R-type instructions) identifies the corresponding instructions. From the ISA tables in Section 2.1 one can easily define decode predicates that check the opcode (and function code) of the current instruction. The naming convention is to name the predicates with the mnemonic of the corresponding instruction from the tables. Here we list as an example the predicates for jump, store word and addition.

$$j(I) \overset{def}{=} I[31:26] = 000010$$

$$sw(I) \overset{def}{=} I[31:26] = 101011$$

$$add(I) \overset{def}{=} I[31:26] = 001000 \wedge I[5:0] = 10^311$$

Definition 2.9 ▶ Instruction Layout Fields

Furthermore, following the instruction layout we define shorthands for accessing instruction fields.

- Register address of the target (rt), source (rs) and destination (rd) register.

$$rt(I \in \mathbb{B}^{32}) \in \mathbb{B}^5 \overset{def}{=} I[20:16]$$

$$rs(I \in \mathbb{B}^{32}) \in \mathbb{B}^5 \overset{def}{=} I[25:21]$$

$$rd(I \in \mathbb{B}^{32}) \in \mathbb{B}^5 \overset{def}{=} I[15:11]$$

- Immediate constants for I-type instructions.

$$imm(I \in \mathbb{B}^{32}) \in \mathbb{B}^{16} \overset{def}{=} I[15:0]$$

- Immediate constants for J-type instructions.

$$iindex(I \in \mathbb{B}^{32}) \in \mathbb{B}^{26} \overset{def}{=} I[25:0]$$

Memory Operations

The instructions which require memory access are the store instructions

◀ **Definition** 2.10 Memory Operations

$$store(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} sw(I) \vee sh(I) \vee sb(I),$$

the load instructions

$$load(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} lw(I) \vee lh(I) \vee lb(I) \vee lhu(I) \vee lbu(I)$$

and read-modify-write $rmw(I)$.

They access one or several bytes in the memory starting with the byte at the address defined by the function ea, which makes a case distinction on the instruction type.

$$ea(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} c.gpr(rs(I)) & if\ rtype(I) \\ c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & if\ itype(I) \end{cases}$$

The number of accessed bytes might be 1, 2 or 4 and is defined by:

$$d(I \in \mathbb{B}^{32}) \in \mathbb{N} \overset{def}{=} \begin{cases} 1 & if\ lb(I) \vee lbu(I) \vee sb(I) \\ 2 & if\ lh(I) \vee lhu(I) \vee sh(I) \\ 4 & if\ lw(I) \vee sw(I) \vee rmw(I) \end{cases}$$

In case of a store instruction the corresponding bytes of the target register contain the value to be written in the memory. Formally we define the store value by the function sv.

$$sv(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{8 \cdot d(I)} \overset{def}{=} core.gpr(rt(I))[8 \cdot d(I) - 1 : 0]$$

The value read from the memory and passed to the transition function as a parameter $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is extended to the width of a register (i.e. 32) by the function lv.

$$lv(I \in \mathbb{B}^{32}, R \in \mathbb{B}^{8 \cdot d(I)}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} zxt_{32}(R) & if\ lhu(I) \vee lbu(I) \\ sxt_{32}(R) & otherwise \end{cases}$$

Shift Operations

Definition 2.11 ▶ Shift Operations The shift instructions

$$shift(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

perform shift operations on the content of the target register. The shift distance is a number in the interval from 0 to 31 and is defined by:

$$sdist(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in [0:31] \overset{def}{=} \begin{cases} \langle c.gpr(rs(I))[4:0]\rangle mod\ 32 & if\ I[3] \\ \langle I[10:6]\rangle mod\ 32 & otherwise \end{cases}$$

The result of the shift operation is defined by:

$$sres(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} x[32-d-1:0]0^d & if\ I[1:0] = 00 \\ 0^d x[31:d] & if\ I[1:0] = 10 \\ x[31]^d x[31:d] & if\ I[1:0] = 11 \end{cases}$$

where

$$d = sdist(c, I)$$

$$x = c.gpr(rt(I))$$

Arithmetic and Logic Operations

Definition 2.12 ▶ Arithmetic and Logic Operations The arithmetic and logic instructions are denoted by the predicate alu.

$$alu(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \begin{cases} I[31:29] = 001 & if\ itype(I) \\ I[5:4] = 10 & if\ rtype(I) \end{cases}$$

The ALU operations are executed on 32 bit values and compute a 32 bit result. The operands of an ALU computation are defined as follows.

- left operand

$$lop(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32} \overset{def}{=} c.gpr(rs(I))$$

- right operand

$$rop(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} c.gpr(rt(I)) & if\ rtype(I) \\ sxt_{32}(imm(I)) & if\ \neg(rtype(I) \vee I[28]) \\ zxt_{32}(imm(I)) & otherwise \end{cases}$$

The result of the computation is defined by:

$$alures(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32} \overset{def}{=}$$
$$\begin{cases} lop(c,I) +_{32} rop(c,I) & if\ addi(I) \vee addi(I) \\ & \vee addu(I) \vee addui(I) \\ lop(c,I) -_{32} rop(c,I) & if\ sub(I) \vee subu(I) \\ lop(c,I) \wedge_{32} rop(c,I) & if\ and(I) \vee andi(I) \\ lop(c,I) \vee_{32} rop(c,I) & if\ or(I) \vee ori(I) \\ lop(c,I) \oplus_{32} rop(c,I) & if\ xor(I) \vee xori(I) \\ lop(c,I)\ \overline{\vee}_{32}\ rop(c,I) & if\ addi(I) \\ rop[15:0]0^{16} & if\ lui(I) \\ 0^{31}([lop(c,I)] < [rop(c,I)]\ ?\ 1 : 0) & if\ slt(I) \vee slti(I) \\ 0^{31}(\langle lop(c,I)\rangle < \langle rop(c,I)\rangle\ ?\ 1 : 0) & if\ sltu(I) \vee sltui(I) \end{cases}$$

General Purpose Register File Update

Definition 2.13 General Purpose Register File Update

The execution of an instruction in the general case generates a result to be saved in some GPR register. The predicate $gprw$ defines the set of instructions which cause an update of the GPR.

$$gprw(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} alu(I) \vee shift(I) \vee load(I) \vee rmw(I) \\ \vee jal(I) \vee jalr(I) \vee movs2g(I)$$

The 5 bit wide address of the register to be written is defined by:

$$des(I \in \mathbb{B}^{32}) \in \mathbb{B}^{5} \overset{def}{=} \begin{cases} 1^5 & if\ jal(I) \\ rd(I) & if\ rtype(I) \wedge \neg movs2g(I) \\ rt(I) & otherwise \end{cases}$$

The 32 bit input for the register update is computed by the function $gprdin$ and depends on the current core configuration, the current instruction and the input from the memory.

$$gprdin(c \in C_{CORE}, I \in \mathbb{B}^{32}, R \in \mathbb{B}^{8 \cdot d(I)}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} c.pc +_{32} 4 & if\ jal(I) \vee jalr(I) \\ lv(I,R) & if\ load(I) \vee rmw(I) \\ c.gpr(rd(I)) & if\ movs2g(I) \\ alures(c,I) & if\ alu(I) \\ sres(c,I) & if\ shift(I) \end{cases}$$

Branch Instructions

Definition 2.14 ▶ Jump and Branch Instructions

The branch instructions are denoted by the predicate $branch$.

$$branch(I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} beq(I) \vee bne(I) \vee bltz(I) \vee bgez(I) \vee blez(I) \vee bgtz(I)$$

The execution of a branch instruction depends on a branch condition. The evaluation of the corresponding condition for every branch instruction is defined by:

$$btaken(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \begin{cases} x = y & if\ beq(I) \\ x \neq y & if\ bne(I) \\ x < 0^{32} & if\ bltz(I) \\ x \geq 0^{32} & if\ bgez(I) \\ x \leq 0^{32} & if\ blez(I) \\ x > 0^{32} & if\ bgtz(I) \end{cases}$$

where

$$x = c.gpr(rs(I))$$
$$y = c.gpr(rt(I))$$

2.4.2.2 Instruction Execution

Definition 2.15 ▶ Instruction Execution

We define the transition function for instruction execution δ^{instr} based on the current core configuration c, the current instruction I and the input from the memory R.

$$\delta^{instr}(c \in C_{CORE}, I \in \mathbb{B}^{32}, R \in (\mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\perp\})) \in C_{CORE} \overset{def}{=} c'$$

where the components of the new core configuration c' are defined as follows

$$c'.pc = \begin{cases} (c.pc +_{32} 4)[31:28] \circ iindex(I) \circ 00 & if\ j(I) \vee jal(I) \\ c.gpr(rs(I)) & if\ jr(I) \vee jalr(I) \\ c.pc +_{32} sxt(imm(I) \circ 00) & if\ branch(I) \wedge btaken(c, I) \\ c.pc +_{32} 4 & otherwise \end{cases}$$

$$c'.gpr(x) = \begin{cases} gprdin(c, I, R) & if\ x = des(I) \wedge gprw(I) \\ c.gpr(x) & otherwise \end{cases}$$

$$c'.spr(x) = \begin{cases} c.gpr(rt(I)) & if\ x = rd(I) \wedge movg2s(I) \\ c.spr(x) & otherwise \end{cases}$$

The regular instruction execution definition excludes the processing of the instruction $eret$, which is defined later in Section 2.4.3.3.

2.4.3 Interrupt Processing

The instruction execution can be interrupted. This may be due to faulty code or external events. In such a case the processor jumps to the interrupt service routine to handle the interrupt. We classify interrupts on three criteria.

- Depending on their origin interrupts can be internal or external.
  - The internal (software generated) interrupts are triggered due to events in the core or memory, e.g. illegal instruction opcode.
  - The external (hardware generated interrupts) are triggered by external signals, e.g. via the APIC.
- Maskable interrupts can be (temporary) switched off so that the executed program keeps the control. Non maskable interrupts are never ignored.
- The resume type of an interrupt defines how the program execution continues after return from interrupt service routine.
  - abort - The program execution is aborted.
  - repeat - The interrupted instruction is repeated.
  - continue - The program execution continues with the next instruction, i.e. the instruction after the interrupted one.

Furthermore, every interrupt has a priority. The interrupt priority defines the order of interrupt handling in case of simultaneous interrupts. The highest priority is 0 and the lowest one is 7.

In Table 2.8 we present all supported interrupts. Our focus is on the external I/O interrupts, part of which are the IPIs. Before defining formally the interrupt processing in the core we introduce some auxiliary definitions.

interrupt level	**shorthand**	**internal/ external**	**type**	**maskable**	
0	reset	external	abort	no	reset signal
1	I/O	external	repeat	yes	devices
2	ill	internal	abort	no	illegal instruction
3	mal	internal	abort	no	misaligned
7	ovf	internal	continue	yes	overflow

Table 2.8: Interrupt Types

Assumption 1 *Reset Assumption We assume in this thesis the absence of reset interrupts.*

2.4.3.1 Auxiliary Definitions for Interrupt Processing

As already mentioned, the cause of an interrupt can be an internal or an external event. The internal events we denote by the eight bit output value of the (uninterpreted) function iev from the current core configuration and the current instruction.

$$iev :: C_{CORE} \times \mathbb{B}^{32} \to \mathbb{B}^{8}$$

The experienced reader will notice that for page faults we need additional information from the MMU. However we omit the page fault signals here and give the arguments for this simplification at the end of the section.

The external events are computed by the APIC as defined in Definition 2.34 and passed to the core as $eev \in B^{256}$.

Definition 2.16 ▶ Cause of an Interrupt

We define the cause of an interrupt by a 32 bit vector ca, in which the first 8 positions are defined by the internal and external interrupt events and the remaining bits are set to zero. ca is computed from the current core configuration c, the external event vector eev and the current instruction I.

$$ca(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32}$$

$$ca(c, eev, I)[j] \overset{def}{=} \begin{cases} \bigvee\limits_{i=0}^{255} eev[i] & if\ j = 1 \\ iev(c, I)[j] & if\ j \in [2:7] \\ 0 & otherwise \end{cases}$$

The bits $[2:7]$ in the output of ca are used for internal interrupts. The second bit denotes an external interrupt delivered by the APIC, as defined later in this chapter. The first bit denotes a reset interrupt. Since we exclude $reset$ in this thesis we set $ca(c, eev, I)[0]$ to 0.

Since interrupts may be masked, it is possible that an interrupt recorded in ca does not influence the core transition.

Definition 2.17 ▶ Masked cause of interrupt

The masked cause of interrupt defines the raised interrupts visible to the core. Additionally to the cause of interrupt here we take into account the mask bits from the status register. The second status register bit masks device interrupts and the eighth one masks overflow interrupts.

$$mca(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}) \in \mathbb{B}^{32}$$

$$mca(c, eev, I)[j] = \begin{cases} ca(c, eev, I)[j] \wedge c.spr.sr[j] & if \quad j \in \{1, 7\} \\ ca(c, eev, I)[j] & otherwise \end{cases}$$

The least significant bit that is set in mca defines the raised interrupt with the highest priority.

Definition 2.18 ▶ Interrupt Level

We define the interrupt level, i.e. the priority of the triggered interrupt, by the function il.

$$il(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}) \in \mathbb{N} \overset{def}{=} min\{i \mid mca(c, eev, I)[i] = 1\}$$

Based on the previous definition we define the JISR predicate of the core as a disjunction of all masked cause bits.

◀ **Definition** 2.19
JISR event

$$isJISR(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \bigvee_j mca(c, eev, I)[j]$$

In case of an active JISR signal we have an unmasked interrupt, which has to be handled, and we jump to the interrupt service routine.

2.4.3.2 JISR

The JISR transition saves the current execution context of the core, stores information for the triggered interrupt and sets the program counter to the first instruction of the service routine.

We define the JISR transition function δ^{jisr} based on the current core configuration c, the current instruction I, the external event vector eev, and the input from the memory R.

◀ **Definition** 2.20
Jump to Interrupt Service Routine

$$\delta^{jisr}(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}, R \in (\mathbb{B}^{8 \cdot d(I)} \cup \{\bot\})) \in C_{CORE} \overset{def}{=} c'$$

where the components of the new core configuration c' are defined as follows.

$$c'.pc = 0$$

$$c'.spr(x) = \begin{cases} 0^{32} & if\ x = sr \\ 0^{32} & if\ x = mode \\ c.sr & if\ x = esr \\ zxt_{32}(mca(c, eev, I)) & if\ x = eca \\ nextpc & if\ x = epc \\ data & if\ (x = edata) \wedge (il(c, eev, I) = 1) \\ c.mode & if\ x = emode \\ c.spr(x) & otherwise \end{cases}$$

$$c'.gpr = \begin{cases} \delta_{instr}(c, I, R).gpr & if\ continue(c, I, eev) \\ c.gpr & otherwise \end{cases}$$

We denote

- by $nextpc \in \mathbb{B}^{32}$ the program counter to be restored after JISR

$$nextpc = \begin{cases} \delta_{instr}(c, I, R).pc & if\ continue(core, I, eev) \\ core.pc & otherwise \end{cases},$$

- by $data \in \mathbb{B}^{32}$ the interrupt data

  $$data = bin_{32}(k) \ ,$$

  where $k \in \mathbb{B}^8$ is the highest priority of the external signals.

  $$k = min\{j \mid eev[j] = 1\}$$

  In our model we only examine a single type of external interrupts but in general there are different interrupts delivered through the APIC.

2.4.3.3 Return from Interrupt

The execution of the instruction $eret$ is to be considered as a separate case. It is the last instruction to conclude the interrupt service routine and to switch back to the interrupted program.

Definition 2.21 ▶ Return from Interrupt Transition Function

We define the return from interrupt by δ^{eret} based on the current core configuration c.

$$\delta^{eret}(c \in C_{CORE}) \in C_{CORE} \overset{def}{=} c'$$

where the new core configuration c' is defined as follows.

$$c'.pc = c.spr(epc)$$

$$c'.gpr = c.gpr$$

$$c'.spr(x) = \begin{cases} c.spr(esr) & if \ x = sr \\ c.spr(emode) & if \ x = mode \\ c.spr(x) & otherwise \end{cases}$$

2.4.4 Core Transitions

Now we have the auxiliary definitions necessary to define the transition function of the processor core.

Definition 2.22 ▶ Core Transition Function

We define δ^{core} based on the current core configuration c, the current instruction I, the external event vector eev and the input from the memory R. δ^{core} makes a case split on the JISR signal and on the $eret$ predicate. In the absence of interrupts and if the current instruction is not $eret$ the step of the core is defined by δ^{instr}.

$$\delta^{core}(c \in C_{CORE}, eev \in \mathbb{B}^{256}, I \in \mathbb{B}^{32}, R \in (\mathbb{B}^{8 \cdot d(I)} \cup \{\bot\})) \in C_{CORE} \overset{def}{=}$$

$$\begin{cases} \delta^{jisr}(c, eev, I, R) & if \, isJISR(c, eev, I) \\ \delta^{eret}(c) & if \, \neg isJISR(c, eev, I) \wedge eret(I) \\ \delta^{instr}(c, I, R) & otherwise \end{cases}$$

2.5 Advanced Programmable Interrupt Controller

The APIC provides a mechanism for interrupt delivery with two main functions. On the one hand the APIC collects external interrupts and forwards them to the core. On the other hand over the APIC one core can send and receive interrupts to/from other cores. Such interrupts are called inter-processor interrupts (IPI). IPIs are used in multiprocessor systems to implement synchronization points of system wide functions, e.g. during start up. The system software uses IPIs to enforce the execution of given tasks on processors, e.g. flushing TLBs after changes in the page tables. In other words the APIC allows system software running on different processors to communicate using IPIs. The delivery of the IPIs is based on the APIC state, which is defined by the APIC registers. The APIC registers are memory-mapped, i.e. they are accessed using regular memory accesses to a dedicated memory page. $x86$ processors support different types of IPIs and allow different methods of addressing the interrupt targets. IPIs are specified by a vector and/or a type. The combination of both defines different mechanisms of delivery from the target APIC to the processor it belongs to. The set of targets of a given IPI can consist of one, all, or a group of processors. It is specified depending on the IPI's destination mode, i.e. physical destination mode or logical destination mode. In physical destination mode we can either address a single processor by the corresponding APIC or use a shorthand which specifies as targets:

- all APICs,
- or only the issuing APIC,
- or all APICs excluding the issuing one.

The logical destination mode provides on $x86$ architectures more possibilities to address a subset of processors.

Here we present a simplified APIC model, which is limited to a subset of the $x86$ IPIs. We only consider the delivery of level-triggered maskable interrupts in physical destination mode. We leave out APIC components related only to the boot phase of the system, to devices and to delivery failures. In the same time our model is defined in a way that allows us to extend it easily and the missing parts can be just plugged into the following definitions.

2.5.1 APIC configuration

The APIC configuration consists of four registers.

◀ **Definition** 2.23 APIC configuration

$$C_{APIC} \equiv [apicId \in \mathbb{B}^{32}, icr \in \mathbb{B}^{32}, isr \in \mathbb{B}^{256}, irr \in \mathbb{B}^{256}]$$

- The APIC ID register ($apicId$) stores a system wide unique ID of the APIC.

- The interrupt command register (icr) stores the interrupt request, which is to be sent, and all the information needed for its delivery.
- The in-service register (isr) denotes which interrupt requests are currently being handled.
- The interrupt request register (irr) stores the interrupt requests waiting to be handled.

In Table 2.9 we list the APIC registers together with their size and the corresponding byte offset in the APIC page.

$\langle$ **Offset** $\rangle$	**Register**	**Software Read/Write**	**Description**
0	$apicId \in \mathbb{B}^{32}$	Read / Write	APIC ID Register
4	$icr \in \mathbb{B}^{32}$	Read / Write	Interrupt Command Register
16	$isr \in \mathbb{B}^{256}$	Read Only	In-Service Register
48	$irr \in \mathbb{B}^{256}$	Read Only	Interrupt Request Register

Table 2.9: APIC registers

2.5.1.1 APIC Interrupt Command Register

The interrupt command register allows software to specify and send an IPI. Let us take a closer look at the layout of ICR as defined in Table 2.10.

Bits	**Name**	**Meaning**
31 : 24	DEST	destination field
23 : 20	reserved	
19 : 18	DSH	destination shorthand
17 : 13	reserved	
12	DS	delivery status
11	DM	destination mode
10 : 8	MT	message type
7 : 0	VEC	vector

Table 2.10: ICR Bits

We introduce a shorthand notation for accessing fields of register icr.

- $icr.VEC$ denotes the interrupt vector.

$$icr.VEC \in \mathbb{B}^8 = icr[7:0]$$

- $icr.MT$ denotes the interrupt's type.

$$icr.MT \in \mathbb{B}^3 = icr[10:8]$$

We only use interrupts of type **Fixed**, i.e. $MT = 0^3$.

- $icr.DM$ stores the destination mode.

  $$icr.DM \in \mathbb{B} = icr[11]$$

  Physical destination mode is denoted by $DM = 0$.

- $icr.DS$ stores the delivery status, i.e. whether the APIC is currently delivering an IPI ($DS = 1$) or not ($DS = 0$).

  $$icr.DS \in \mathbb{B} = icr[12]$$

- $icr.DSH$ stores the destination shorthand of the receivers of the IPI.

  $$icr.DSH \in \mathbb{B}^2 = icr[19 : 18]$$

  For the set of receivers there are four possibilities

  $$icr.DSH \in \{\mathit{ALL\text{-}BUT\text{-}SELF}, \mathit{ALL}, \mathit{SELF}, \mathit{DESTINATION}\}$$

  where $\mathit{DESTINATION}$ denotes an empty shorthand, i.e. the IPI target is not defined by a shorthand but by the DEST field.

Destination Shorthand Coding	
00	*DESTINATION*
01	*SELF*
10	*ALL*
11	*ALL-BUT-SELF*

Table 2.11: Possible values for $icr.DSH$.

- $icr.DEST$ stores the APIC ID of the target in case of $\mathit{DESTINATION}$ destination shorthand.

  $$icr.DEST \in \mathbb{B}^8 = icr[31 : 24]$$

2.5.1.2 Sending an IPI

To send an IPI, a processor programs the ICR of its APIC with the information about the interrupt and its delivery. The function $\mathit{writeICR}$ stores a given data (of a proper type) into the ICR of the local APIC. All ICR bits are read/write except DS. The DS field is a read only for software and is written only by the APIC hardware. Whenever the ICR is written the DS is automatically set to 1. This indicates that the ICR stores currently a pending interrupt request. When the interrupt request is delivered to the targets the DS bit is cleared by the hardware. Thus the system software should poll for the DS bit before initiating a write to the ICR. Otherwise it may overwrite a previous interrupt request that is still pending.

We define function $\mathit{writeICR}$ based on the current APIC state $apic$ and the passed value to be written into ICR $data$. $\mathit{writeICR}$ models the software driven

◄ **Definition** 2.24
Write to ICR

Interrupt Request irr[i]	**Interrupt in Service isr[i]**	**Description**
0	0	no new request, no request in service
0	1	no new request, request in service
1	0	pending request, no request in service
1	1	pending request, request in service

Table 2.12: States of an incoming interrupt.

update of the register together with the semantics of the APIC itself, i.e. the change of the delivery status.

$$writeICR(apic \in C_{APIC}, data \in \mathbb{B}^{32}) \in C_{APIC} \overset{def}{=} apic[icr \mapsto data']$$

where

$$data'[j] = \begin{cases} 1 & if\ j = 12 \\ data[j] & otherwise \end{cases}$$

Software Condition 1 (Software IPI) *For software generated IPIs we have to write the proper vector $VEC = 0^8$ into the ICR. We have to satisfy our model restrictions and use only physical destination mode $DM = 0$ and fixed interrupts $MT = \{$**Fixed**$\}$. Furthermore for simplicity we only consider IPIs that interrupt all processors but the sender, i.e. destination shorthand $DSH = ALL\text{-}BUT\text{-}SELF$.*

2.5.1.3 Receiving an IPI

For every received interrupt the APIC stores two bits, one in IRR and one in ISR. The index of the corresponding bits is defined by the interrupt vector. Vector $vec = 0^8$ denotes interrupts sent by other processors' APICs. All other vectors are used for device interrupts. The process of handling an IPI can be in one of the four states shown in Table 2.12

The incoming requests are stored into IRR. An IPI request remains pending until the core is ready to service it. The APIC delivers given pending IPI to the core via the external event signal if the core is not servicing an IPI of the same type currently. A pending IPI delivered to the core causes JISR if:

- there is no $reset$ interrupt,
- IPIs are not masked.

When the jump to the interrupt service routine occurs, the state of the interrupt changes to "in-service", the IRR bit is cleared and the corresponding ISR bit is set to one. After the interrupt is handled the in-service bit is cleared. Figure 2.2 shows the possible transitions.

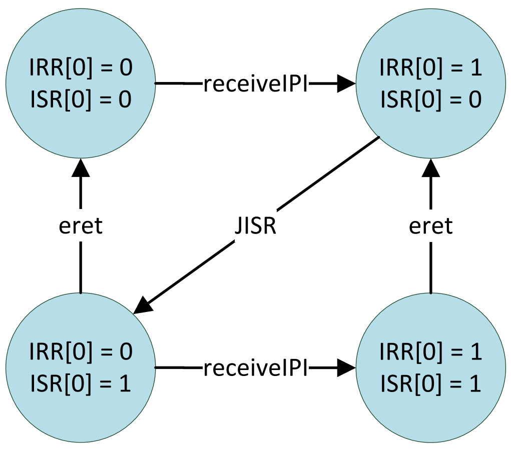

Figure 2.2: APIC receiver state.

It is possible that an interrupt request is delivered to the APIC and the corresponding IRR bit is already set, e.g. if several processors simultaneously send an IPI to the same target. In such a case the IRR state does not change. The APIC does not stack multiple requests of the same type and will deliver only one of the simultaneously arriving interrupts to the processor. The system software must provide data structures and implement a software protocol to ensure that every single IPI is served. An interrupt request can get pending during a previous request of the same type is being serviced.

2.5.2 APIC Transition

For reading the APIC registers we define $read_{APIC}$ based on the current APIC state $apic$ and an offset in the APIC page $offset$.

◀ **Definition** 2.25
APIC reads

$$read_{APIC}(apic \in C_{APIC}, offset \in \mathbb{B}^7) \in (\mathbb{B}^{32} \cup \mathbb{B}^{256} \cup \bot) \overset{def}{=}$$
$$\begin{cases} apic.apicId & if\ \langle offset \rangle = 0 \\ apic.icr & if\ \langle offset \rangle = 4 \\ apic.isr & if\ \langle offset \rangle = 16 \\ apic.irr & if\ \langle offset \rangle = 48 \\ \bot & if\ otherwise \end{cases}$$

We define $write_{APIC}$ based on the current APIC state $apic$, an offset in the APIC page $offset$ and the passed value to be written into corresponding register $data$. If the offset does not define a register writable for software the write is

◀ **Definition** 2.26
APIC writes

ignored.

$$write_{APIC}(apic \in C_{APIC}, offset \in \mathbb{B}^7, v \in \mathbb{B}^{32}) \in C_{APIC} \overset{def}{=}$$
$$\begin{cases} writeAPICID(apic, v) & if\ \langle offset \rangle = 0 \\ writeICR(apic, v) & if\ \langle offset \rangle = 4 \\ apic & otherwise \end{cases}$$

where $writeAPICID$ is trivially defined as follows

$$writeAPICID(apic \in C_{APIC}, data \in \mathbb{B}^{32}) \in C_{APIC} \overset{def}{=} apic[apicId \mapsto data]$$

Definition 2.27 ▶ APIC transition

We define the transition function of the APIC δ^{apic} based on the input alphabet

$$\Sigma_{APIC} \overset{def}{=} (\mathbb{B}^7 \times \mathbb{B}^{32}) \cup (\{\textbf{Fixed}\} \times \mathbb{B}^8) \cup \{\textbf{jisr}, \textbf{eret}\}.$$

The APIC transition has four possible cases.

- A write access to APIC registers initiated by software. The passed data is written to the register with the corresponding offset.
- Receiving an IPI. The interrupt request register bit defined by the interrupt vector is set to one.
- As a part of JISR the pending interrupt with the highest priority gets in-service.
- At the end of the service routine, i.e. at return from interrupt, the ISR bit corresponding to the serviced interrupt is cleared.

$$\delta^{apic}(APIC \in C_{APIC}, in \in \Sigma_{APIC}) \in C_{APIC} \overset{def}{=}$$
$$\begin{cases} write_{APIC}(apic, offset, v) & if\ in = (offset, v) \\ apic[irr[\langle vector \rangle] \mapsto 1] & if\ in = (\textbf{Fixed}, vector) \\ apic[irr[i] \mapsto 0, isr[i] \mapsto 1] & if\ in = \textbf{jisr} \\ apic[isr[j] \mapsto 0] & if\ in = \textbf{eret} \end{cases}$$

where $i = min\{k \mid apic.irr[k] = 1\}$ and $j = min\{k \mid apic.isr[k] = 1\}$.

2.6 Translation Lookaside Buffer

TLBs are an essential part of modern memory systems. They cache address translations for memory virtualization. The content of the TLBs have to be synchronized on several occasions e.g. when a faulty translation is discovered and must be removed. Synchronized TLB flushes are implemented with IPIs.

A detailed model of a TLB with its operations is presented in [Sch13]. However in system mode the processor runs without address translation. Since we examine only processor instruction execution in system mode, i.e. without address translation, we do not consider the concrete translations cached in the TLB. Furthermore we are not interested in proofs of the MMU or TLB virtualization. Both topics have been handled in [Kov13].

Concerning the TLB, we are interested only in properties about TLB flushes that are synchronized over IPIs. Therefore in the scope of this work it is sufficient to have a TLB model that can grow, i.e. by adding new translations to the set of cached ones, and can be emptied(flushed). We think of TLB entries being marked with unique (increasing) identifiers. Since we are not interested in the values of the TLB entries we define our model based purely on the identifiers.

2.6.1 TLB configuration

◀ **Definition** 2.28
TLB Configuration

The TLB configuration

$$C_{TLB} \overset{def}{=} [current \in \mathbb{N}, flushed \in \mathbb{N}]$$

consists of:

- a $current$ counter, which stores the identifier of the translation added last,
- a $flushed$ counter, which stores the identifier of the the last translation added before the last flush.

2.6.2 TLB Transitions

◀ **Definition** 2.29
TLB Transition Function

We define the transition function of the TLB δ^{tlb} based on the input alphabet

$$\Sigma_{TLB} \overset{def}{=} add \cup flush.$$

When we add a new translation in our model, we increase the $current$ counter by one. When we flush the TLB, we copy the value of the $current$ counter into the $flushed$ counter.

$$\delta^{tlb}(tlb \in C_{TLB}, in \in \Sigma_{TLB}) \in C_{TLB} \overset{def}{=}$$
$$\begin{cases} [tlb.current + 1, tlb.flushed] & \text{if } in = add \\ [tlb.current, tlb.current] & \text{if } in = flush \end{cases}$$

2.7 $MIPS_P$ Virtualization Restrictions

Modern processors offer a lot of hardware support for virtualization. Moreover only with the hardware virtualization support we can implement hypervisors for full virtualization without modifying the guest.

Guest execution is initiated by the instruction VMRUN. VMRUN switches the context and starts the instruction execution of guest instructions. During guest execution guest steps are interleaved with MMU steps. This continues until an intercept occurs. In this case the context is switched from guest to hypervisor by the hardware step VMEXIT. VMEXIT is the counter part of the instruction VMRUN. It saves the guest state in the memory, restores the hypervisor state and changes the processor mode. After VMEXIT the processor execution continues with the hypervisor instruction following VMRUN. The hypervisor handles the reason for the intercept and starts the guest again.

However the main topic of this work is on the communication between processors executing hypervisor kernel code. Our statements and proofs are not influenced by guest steps and the implementation of context switches between guest and hypervisor. Therefore we omit the definitions for saving/restoring the context and the specification of the guest intercept mechanism. It is sufficient to know that a pending IPI during guest execution will cause a context switch.

We use a single guest step in our $MIPS_P$ model as an abstraction of all steps in guest mode. This includes guest instruction execution and MMU steps.

We reduce the effect of VMRUN/VMEXIT to mode change and the effect of guest steps to adding new translations into the TLB.

In this way our model is an abstraction in which the core is always containing the hypervisor context. In a simulation relation between $MIPS_P$ and an extended virtualization $MIPS$ machine during guest execution the $MIPS_P$ registers will be mapped not to the real registers of the extended machine, but to the memory area where they were saved during mode switch. Furthermore one would have to prove:

- that the memory assigned to guests is disjoint from the memory of the hypervisor and from the APIC page and
- that guests never access neither hypervisor memory, nor the APIC.

2.8 Auxiliary Definitions

In order to define formally the overall $MIPS_P$ transition we introduce in this section some additional definitions.

Definition 2.30 ▶ Register Accesses

We have shortened the notation for registers or other (sub)components, which are uniquely identifiable. For example for better readability we refer to the program counter of processor i by

$$h.pc_i \overset{def}{=} h.cpu[i].core.pc$$

and to the delivery status register by

$$h.DS_i \overset{def}{=} h.cpu[i].apic.icr.DS.$$

$MIPS_P$ supports two execution modes: a system mode and user mode. We let the hypervisor run in system mode and its guests in user mode.

◀ **Definition** 2.31 Processor Mode

The predicate $guest$ denotes whether a given processor operates in a user mode.

$$guest(core \in C_{CORE}) \in \mathbb{B} \overset{def}{=} core.mode[0] = 1$$

◀ **Definition** 2.32 External Interrupt Request

The predicate irr denotes existence of a pending interrupt request in the current APIC configuration.

$$irr(apic \in C_{APIC}) \in \mathbb{B} \overset{def}{=} \bigvee_{j=0}^{255} apic.irr[j]$$

◀ **Definition** 2.33 Handling an External Interrupt

The predicate isr denotes whether currently an external interrupt is in-service.

$$isr(apic \in C_{APIC}) \in \mathbb{B} \overset{def}{=} \bigvee_{j=0}^{255} apic.isr[j]$$

◀ **Definition** 2.34 External Event Signals

We define the external event vector $eev \in \mathbb{B}^{256}$ provided by the APIC $apic \in C_{APIC}$ to the processor core as follows.

$$eev(apic)[j] \overset{def}{=} \begin{cases} 0 & if \exists k \leq j.\ apic.isr[k] = 1 \\ apic.irr[j] & otherwise \end{cases}$$

◀ **Definition** 2.35 Masked External Interrupt Request

The predicate $exint$ denotes whether in the current configuration an external interrupt is triggered.

$$exint(c \in C_{CORE}, apic \in C_{APIC}) \in \mathbb{B} \overset{def}{=} \bigvee_{i=0}^{255} eev(apic)[i] \wedge c.sr[1]$$

◀ **Definition** 2.36 External $eret$

The predicate $eret_{IPI}$ denotes whether the current instruction is an $eret$ instruction that ends the service of an IPI.

$$eret_{IPI}(c \in C_{CORE}, I \in \mathbb{B}^{32}) \overset{def}{=} eret(I) \wedge c.eca[1]$$

◀ **Definition** 2.37 Internal Interrupt

The predicate $iint$ denotes whether in the current configuration an internal interrupt is triggered.

$$iint(c \in C_{CORE}, I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} isJISR(c, 0^{256}, I)$$

Definition 2.38 ▶ Current Instruction

The function I returns the current instruction. This is the instruction which the core will execute next if there is no interrupt.

$$I(core \in C_{CORE}, m \in C_{MEM}) \overset{def}{=} m_4(core.pc)$$

Software Condition 2 (Aligned Program Counter) *Since $MIPS_P$ instructions are coded in 32 bits we define a software condition for the hypervisor execution, that requires aligned values of the program counter.*

$$\forall core \in C_{CORE}.\ \neg guest(core) \implies core.pc[1:0] = 00$$

Definition 2.39 ▶ APIC Addresses

The APIC registers are mapped to addresses from $1^{20}0^{12}$ to $1^{20}0^51^7$. This region in the memory we call APIC range and denote by

$$\mathbb{A}_{apic} \overset{def}{=} [1^{20}0^{12} : 1^{20}0^51^7].$$

For every address $addr \in \mathbb{A}_{apic}$ the function a_{off} computes the aligned offset in the APIC page.

$$a_{off}(addr \in \mathbb{B}^{32}) \in \mathbb{B}^7 \overset{def}{=} (addr -_{32} 1^{20}0^{12})[6:2]00$$

Definition 2.40 ▶ APIC Ports Read

The predicate $load_{\mathbb{A}_{apic}}$ denotes whether the current instruction is a load from the APIC range.

$$load_{\mathbb{A}_{apic}}(c \in C_{CORE}, I \in \mathbb{B}^{32}) \overset{def}{=} load(I) \wedge ea(c, I) \in \mathbb{A}_{apic}$$

Definition 2.41 ▶ APIC Ports Write

The predicate $store_{\mathbb{A}_{apic}}$ denotes whether the current instruction is a store to the APIC range.

$$store_{\mathbb{A}_{apic}}(c \in C_{CORE}, I \in \mathbb{B}^{32}) \overset{def}{=} store(I) \wedge ea(c, I) \in \mathbb{A}_{apic}$$

Definition 2.42 ▶ Memory System

The function ms defines the processors' view of the memory system. It comprises the memory and the APIC ports.

If the address a belongs to the APIC, we have first to compute the aligned offset in the APIC range by $a_{off}(a)$ and identify the APIC register, to which it belongs. Then we read the entire register, wich is either 32 (i.e. $apicId$ and icr) or 256 (i.e. isr and irr) bit long, by the $read_{APIC}$ function. At the end, if the computed offset belongs to isr or irr, we select the proper word in it. Note, that we allow only to read words from the APIC range. This condition should be maintained by the $MIPS_P$ transition from the next section.

$$ms_d(apic \in C_{APIC}, m \in C_{MEM}) \in C_{MEM}$$

$$ms_d(apic, m)(a) \overset{def}{=}$$
$$\begin{cases} read_{APIC}(apic, 16)[a_{off}(a) - 16 : a_{off}(a) - 13] & if\ a \in \mathbb{A}_{apic} \\ & \quad \wedge a_{off}(a) \in [16 : 47] \\ read_{APIC}(apic, 48)[a_{off}(a) - 48 : a_{off}(a) - 45] & if\ a \in \mathbb{A}_{apic} \\ & \quad \wedge a_{off}(a) \in [48 : 69] \\ read_{APIC}(apic, a_{off}(a)) & if\ a \in \mathbb{A}_{apic} \\ & \quad \wedge a_{off}(addr) \notin [16 : 69] \\ m_d(a) & otherwise \end{cases}$$

2.9 Transition

We model our concurrent $MIPS_P$ machine as an automaton with external inputs. The set of inputs we denote by Σ_M. The transitions function of the automaton we denote by Δ.

$$\Delta : C_M \times \Sigma_M \rightharpoonup C_M$$

Δ is partial since for every input a set of conditions must be fulfilled in order the transition to be taken. We denote the transition guard by the predicate $\mathbb{G}$.

$$\mathbb{G} : C_M \times \Sigma_M \to \mathbb{B}$$

Obviously having in our system multiple components that might be making a step simultaneously we need a way to model a non-deterministic semantics as a deterministic one. We handle this by adding a scheduling information to the input parameters of our automaton. In that way we define execution sequences in which every step is completely defined by the sequence of inputs passed to the system.

Later when proving properties of our models we have to consider all possible input sequences and executions.

As we said in the beginning of this chapter, the overall transition of our model is a composition of the transitions of different components. The transition function makes a case distinction on the stepping component and the kind of particular step it is executing. In some cases only one component is stepping and in some other several components are performing a step simultaneously, e.g. memory and core are involved in the execution of a store instruction. In such cases one component can be determined as the leading or active one, whereas the others do not make a step on their own, but instead respond to the step of the active component. In our model, we assume that the memory can not perform active steps. Under this assumption the possible active component is either a core, or a TLB, or an APIC.

Definition 2.43 ▶ Set of Possible Steps

The input alphabet of the system Σ_M denotes the set of possible inputs.

$$\Sigma_M \overset{def}{=} \textbf{core} \times Pid \cup \textbf{ipi} \times Pid \cup \textbf{guest} \times Pid \cup \textbf{vmexit} \times Pid$$

where

- $(\textbf{core}, pid)$ is the input defining a core step of processor pid.
- $(\textbf{ipi}, pid)$ is the input defining an IPI broadcast step where the APIC of processor pid is the sender.
- $(\textbf{guest}, pid)$ is the input defining a guest step of processor pid.
- $(\textbf{vmexit}, pid)$ is the input defining a VMEXIT step of processor pid.

Definition 2.44 ▶ Transition Function

The definition of the transition function is split into four cases, depending on the input α. We define the new configuration

$$h' = \Delta(h, \alpha)$$

for the components that are changed by the transition and omit listing components that stay unchanged.

- A core step of processor pid.
  The guard of the core step transition:
  - forbids instruction execution in guest mode,
  - forbids fetching of instruction from the APIC page,
  - forbids memory accesses to bytes in the memory and in the APIC page at the same time,
  - forbids read-modify-write accesses to the APIC page,
  - allows only word read/write accesses to the APIC page.

$$\begin{aligned}
\mathbb{G}(h, (\textbf{core}, pid)) \overset{def}{=}\ & (isJISR(c, eev, I) \vee \neg guest(core)) \\
& \wedge h.pc_{pid} \notin \mathbb{A}_{apic} \\
& \wedge ((store(I) \vee load(I)) \wedge ea(h.core_{pid}, I) \notin \mathbb{A}_{apic} \\
& \qquad\qquad \implies (ea(h.core_{pid}, I) + d(I)) \notin \mathbb{A}_{apic} \\
& \wedge (rmw(I) \implies ea(h.core_{pid}, I) \notin \mathbb{A}_{apic} \\
& \qquad\qquad \wedge (ea(h.core_{pid}, I) + 3) \notin \mathbb{A}_{apic}) \\
& \wedge ((store(I) \vee load(I)) \wedge d(I) \neq 4 \implies ea(h.core_{pid}, I) \notin \mathbb{A}_{apic})
\end{aligned}$$

The new configuration

$$h' = \Delta(h, (\textbf{core}, pid))$$

is defined as follows.

$$h'.core_{pid} = \delta^{core}(h.core_{pid}, eev, I, R)$$

$$h'.apic_{pid} = \begin{cases} \delta^{apic}(h.apic_{pid}, \textbf{jisr}) & \textit{if } exint(h.core_{pid}, h.apic_{pid}) \\ \delta^{apic}(h.apic_{pid}, \textbf{eret}) & \textit{if } eret_{IPI}(h.core_{pid}, I) \\ \delta^{apic}(h.apic_{pid}, (a_{\mathit{off}}(addr), data)) & \textit{if } store_{\mathbb{A}_{apic}}(h.core_{pid}, I) \\ & \quad \wedge \neg isJISR(h.core_{pid}, eev, I) \\ h.apic_{pid} & \textit{otherwise} \end{cases}$$

$$h'.tlb_{pid} = \begin{cases} \delta^{tlb}(h.tlb_{pid}, \mathit{flush}) & \textit{if } \mathit{flush}(I) \\ h.tlb_{pid} & \textit{otherwise} \end{cases}$$

$$h'.M = \begin{cases} \delta^{mem}(h.M, (addr, data)) & \textit{if } (rmw(I) \vee store(I)) \wedge addr \notin \mathbb{A}_{apic} \\ & \quad \wedge \neg isJISR(h.core_{pid}, eev, I) \\ h.M & \textit{otherwise} \end{cases}$$

where

$$eev = \begin{cases} eev(h.apic_{pid}) & \textit{if } exint(h.core_{pid}, h.apic_{pid}) \\ 0^{256} & \textit{otherwise} \end{cases}$$

$$I = I(h.core_{pid}, h.M)$$

$$addr = ea(h.core_{pid}, I)$$

$$data = sv(h.core_{pid}, I)$$

$$R = ms_{d(I)}(h.apic_{pid}, h.M)(addr)$$

- Delivery of an IPI request from the local APIC on processor pid.
  The guard of the IPI step transition:
  - requires an IPI send request on processor pid,

 – IPI of type $Fixed$,
 – physical destination mode.

$$\begin{aligned}\mathbb{G}(h, (\textbf{ipi}, pid)) \overset{def}{=} \quad & h.icr_{pid}.DS = 1 \\ & \wedge h.icr_{pid}.MT = 0^3 \\ & \wedge h.icr_{pid}.DM = 0\end{aligned}$$

The new configuration

$$h' = \Delta(h, (\textbf{ipi}, pid))$$

is defined as follows

$$h'.apic_i = \begin{cases} \delta^{apic}(h.apic_i, (mt, vector))[DS \mapsto 0] & \textit{if } i = pid \wedge target(i) \\ h.apic_i[DS \mapsto 0] & \textit{if } i = pid \wedge \neg target(i) \\ \delta^{apic}(h.apic_i, (mt, vector)) & \textit{if } i \neq pid \wedge target(i) \\ h.apic_i & \textit{otherwise} \end{cases}$$

where

$$\begin{aligned} mt &= h.icr_{pid}.MT \\ vector &= h.icr_{pid}.VEC \end{aligned}$$

$$\begin{aligned} target(i) = & h.icr_{pid}.DSH = ALL \\ & \vee h.icr_{pid}.DSH = \textit{ALL-BUT-SELF} \wedge i \neq pid \\ & \vee h.icr_{pid}.DSH = SELF \wedge i = pid \\ & \vee h.icr_{pid}.DSH = ID \wedge h.APIC\_ID_i = h.apic_{pid}.ICR.DEST \end{aligned}$$

- Guest step of processor pid.
 A guest step adds a new translation to the TLB. The guard of the guest step transition requires guest mode of the processor.

$$\mathbb{G}(h, (\textbf{guest}, pid)) \overset{def}{=} guest(h.core_{pid})$$

 The new configuration

$$h' = \Delta(h, (\textbf{guest}, pid))$$

 is defined as follows.

$$h'.tlb_{pid} = \delta^{tlb}(h.tlb_{pid}, add)$$

- VMEXIT of processor pid.

  The guard of the VMEXIT step transition requires guest mode of the processor.

$$\mathbb{G}(h, (\textbf{vmexit}, pid)) \overset{def}{=} guest(h.core_{pid})$$

  The new configuration

$$h' = \Delta(h, (\textbf{vmexit}, pid))$$

  is defined as follows.

$$h'.mode_{pid}[0] = 0$$

2.9.1 Execution Sequences

◀ **Definition** 2.45 Composition of Steps

The composition of several transition steps defined by given input sequence β of length $n \in \mathbb{N}$, starting at initial configuration h we define recursively as follows:

$$\Delta^n(h \in C_M, \beta \in (\Sigma_M)^n) \in C_M \overset{def}{=} \begin{cases} h & \textit{if } n = 0 \\ \Delta^{n-1}(\Delta(h, \beta_0), \beta[1 : n-1]) & \textit{otherwise}. \end{cases}$$

◀ **Definition** 2.46 Execution sequence

Given initial configuration h and an input sequence β we define the execution sequence

$$h^0 \beta_0 h^1 \beta^1 \ldots h^{n-1} \beta_{n-1} h^n$$

where $h^0 = h$ and every other configuration h^i is obtained executing the corresponding input $\beta[0 : i-1]$

$$h^i = \Delta^i(h, \beta[0 : i-1])$$

We denote execution sequences by

$$h \xrightarrow{\beta} h',$$

where $|\beta| = n$ and $h' = h^n$.

In execution sequences we implicitly assume that all step guards hold. This implies for example that if β_k is a core step input, then in the configuration h^k the corresponding processor is in system mode.

Chapter 3

Reordering of Execution Sequences

In the following sections we want to examine $MIPS_P$ execution sequences which execute compiled code. We want to propagate properties of the *C-IL* program to the $MIPS_P$ execution. For that purpose we need a simulation relation. The simulation relation between the program code and the $MIPS_P$ steps we call compiler consistency. Compiler consistency as defined by Andrey Shadrin [Sha12] is sequential and considers the execution of the *C-IL* program on one core. Obviously one *C-IL* step can correspond to several steps of $MIPS_P$. Furthermore, we know that $MIPS_P$ is a concurrent multiprocessor model, thus in $MIPS_P$ execution sequences steps of different processors can interleave arbitrarily. This makes it impossible to define a point in which consistency holds for several threads. We are looking for the possibility to reorder $MIPS_P$ execution sequences in a way that allows to apply a sequential compiler consistency theorem to interleaved multiprocessor executions.

Proving an order reduction theorem for the general case does not fall in the scope of this work. We use a generic order reduction theorem proven by Christoph Baumann [Bau14].

Christoph Baumann defines an abstract ownership based model called COSMOS. COSMOS contains several computational units and a global memory. The memory accesses in COSMOS must follow an ownership-based policy, which guarantees the absence of memory races among computational units on local data. The memory is separated in portions, each of which is either exclusively owned by a computational unit or shared.

The COSMOS order reduction theorem allows the reordering of executions in a sequence of blocks of steps of a single processor. The idea is to apply sequential compiler consistency on the blocks and to interleave only at consistent states. We call these dedicated states interleaving points.

The main argument for the reordering is that most of the steps of a given CPU have no impact on the execution of other CPUs. These steps do not access

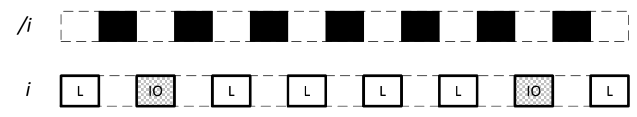

Figure 3.1: Arbitrary interleaving.

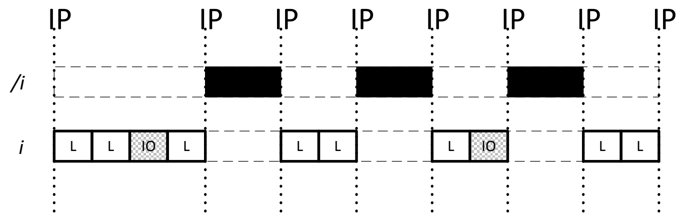

Figure 3.2: Reordered interleaving.

the shared portion of the global memory and we call them local steps. The effect of local steps is visible only to a single processor. In contrast to local steps the effect of so called I/O steps is visible not only to the processing unit but to the whole environment.

I/O steps and interleaving points are defined by the verification engineer and their choice depends on the ISA model, on the particular program and on the compiler. On the one hand we define I/O steps and interleaving points in a way that satisfies the conditions for the application of the COSMOS order reduction theorem. On the other hand later we have to prove that every interleaving point defined by us is a state that satisfies compiler consistency, and that the I/O steps definition covers all non local steps in the execution sequence. We aim at a complete model but our main focus is on execution sequences containing IPI service routine. This includes the execution of compiled *C-IL* hypervisor kernel code and small assembler portions, which are parts of the ISR. For other scenarios the sets of interleaving points and I/O steps defined in this thesis eventually have to be extended.

In order to apply the reordering theorem we have to instantiate COSMOS with MIPS_P. The instantiation interface contains definitions for I/O steps and interleaving points. We identify I/O steps based on the resources they access. The choice of the interleaving points is based on a precondition of the COSMOS reordering theorem, i.e. that between two I/O steps there exist at least one interleaving point. We call this requirement - IOIP condition.

Obviously in MIPS_P APICs have to be considered as shared components, due to the IPI step, which can modify many APICs at the same time. Thus, steps accessing APIC should be considered I/O steps. Unfortunately the APIC is read in every core step, which makes every core step an I/O step. This is a very strong limitation for reordering. In the next section 3.1 we present an extension of the $MIPS_P$ model to solve this problem.

In Section 3.2 we define an ownership discipline that implements the COSMOS model ownership. In Section 3.3 we list some auxiliary definitions. At the end in Section 3.4 we present an instantiation of COSMOS with the new $MIPS_P$ model from Section 3.1.

3.1 $MIPS_P$ Extension

In every core step the APIC is accessed for computing the external interrupt signal. Accessing the APIC in the absence of pending interrupts has no effect to the core step and is in a sense superfluous. We modify $MIPS_P$ in such a way, that core steps access APIC components only if there is a pending interrupt.

◀ **Definition** 3.1 $MIPS_P$ Alphabet Extension

We add to the core step label in our input alphabet a Boolean flag that identifies APIC accesses.

$$\Sigma_M \overset{def}{=} \textbf{core} \times Pid \times \mathbb{B} \cup \textbf{ipi} \times Pid \cup \textbf{guest} \times Pid$$

From now on we denote the input that specifies a core step of processor pid by $(\textbf{core}, pid, ex)$[1] The flag ex identifies the steps where we check for external interrupts. We now define the external event input to the core $eev \in \mathbb{B}^{256}$ by a case distinction. If ex is 1 we read the APIC, otherwise we pass through an empty event vector.

$$eev = \begin{cases} eev(h.apic_{pid}) & \textit{if } ex \\ 0^{256} & \textit{otherwise} \end{cases}$$

We want to show that the extension of the input alphabet of $MIPS_P$ does not change its computation. The input parameter ex has to be bound to the APIC state. On the one hand we want to read the APIC only if it makes sense to, i.e. if there is a pending IPI. On the other hand we can not just ignore pending IPIs by passing an empty event vector.

◀ **Definition** 3.2 $MIPS_P$ Alphabet Invariant

For every input sequence we make the value of the external interrupt flag ex of a core step $(\textbf{core}, pid, ex)$ equivalent to the existence of external interrupt request.

$$\forall \beta \in \Sigma_M, i \in [0 : |\beta| - 1], pid \in Pid.$$
$$\beta_i = (\textbf{core}, pid, ex) \implies (ex \iff exint(h.core_{pid}, h.apic_{pid}))$$

[1]Instead of using as input $(\textbf{core}, pid)$.

With this invariant we can easily prove a simulation between a machine with and a machine without external interrupt flag in its input alphabet. From now on we consider only executions which satisfy that invariant.

We introduce in the following two auxiliary definitions notation, to keep later formalisms concise and readable.

Definition 3.3 ▶ Input Shorthands

For an input $\alpha \in \Sigma_M$

- $\alpha.pid$ denotes the index of the scheduled processor (all IPI steps are denoted by an own ID equal to the number of processors in the system)

$$\alpha.pid \overset{def}{=} \begin{cases} i & \mathit{if}\ \alpha = (\textbf{core}, i, ex) \\ np & \mathit{if}\ \alpha = (\textbf{ipi}, i) \\ i & \mathit{if}\ \alpha = (\textbf{guest}, i) \\ i & \mathit{if}\ \alpha = (\textbf{vmexit}, i) \end{cases}$$

- and $\alpha.ex$ denotes whether α defines core step with an active APIC signal.

$$\alpha.ex \overset{def}{=} \begin{cases} ex & \mathit{if}\ \exists i \in Pid.\ \alpha = (\textbf{core}, i, ex) \\ \bot & \mathit{otherwise} \end{cases}$$

Definition 3.4 ▶ Step Predicates

We define the predicates $core$, $guest$ and ipi to denote whether a given input α defines a core step, a guest step or an IPI step respectively.

$$core(\alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} \exists i \in Pid, ex \in \mathbb{B}.\ \alpha = (\textbf{core}, i, ex)$$

$$guest(\alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} \exists i \in Pid.\ \alpha = (\textbf{guest}, i)$$

$$ipi(\alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} \exists i \in Pid.\ \alpha = (\textbf{ipi}, i)$$

3.2 Ownership

Ownership talks about memory addresses. The set of all addresses included in an ownership setting we denote by $\mathbb{A} \subseteq \mathbb{B}^{32}$. In our ownership model we separate $\mathbb{A}$ into subsets and define access policy for each of them. We distinguish the following three kinds of memory.

- The read only memory contains the addresses that can be read by all processors and written by none.

$$\mathbb{A}^{ro} \subset \mathbb{B}^{32}$$

- The private memory contains the addresses that are assigned to some processor and can be accessed only by this processor.

  $$\mathbb{A}^{pr} \subset \mathbb{B}^{32}$$

  We say that private addresses are owned by the processor they are assigned to.

- The shared memory contains the addresses that can be accessed by all processors.

  $$\mathbb{A}^{sh} \subset \mathbb{B}^{32}$$

During execution the private and the shared addresses can change their ownership state. Private addresses may get shared and vice versa. Contrary to that dynamic behavior the set of all addresses $\mathbb{A}$ and of read only addresses $\mathbb{A}^{ro}$ are fixed. Thus the ownership state consists of a static and dynamic part. We simplify our notation and pass through the execution only dynamic ownership information, whereas the static information is implicitly known.

The dynamic state of the ownership model is defined by a mapping of processor indexes to a set of owned addresses O and by a set of shared addresses.

◀ **Definition** 3.5 Dynamic Ownership Information

$$\mathcal{O} \overset{def}{=} [O \in Pid \to 2^{\mathbb{B}^{32}}, \mathbb{A}^{sh} \in 2^{\mathbb{B}^{32}}]$$

In ownership $o \in \mathcal{O}$ the set of the addresses assigned to processor i is defined by $o.O(i)$. We call this set the processor's owns-set and denote it by the shorthand $o.O_i$. The set of private addresses in o is implicitly defined as the union of all owned addresses.

$$o.\mathbb{A}^{pr} \overset{def}{=} \bigcup_{i \in Pid} o.O_i$$

The set of all private addresses not owned by processor i we denote by:

$$o.\overline{O}_i \overset{def}{=} o.\mathbb{A}^{pr} \setminus o.O_i$$

As shown in Figure 3.3 the shared memory and the private memory may overlap. The intersection of the owned and shared addresses defines addresses that have a single writer but multiple readers. According to that the ownership state of every memory address $a \in \mathbb{A}$ can be either read only, or exclusively owned, or shared owned, or shared unowned.

3.2.1 Ownership Policy

The properties that we maintain on our ownership setting are formally expressed by the following invariant.

The predicate *ownership-inv* denotes whether given ownership state o is valid.

◀ **Definition** 3.6 Ownership Invariant

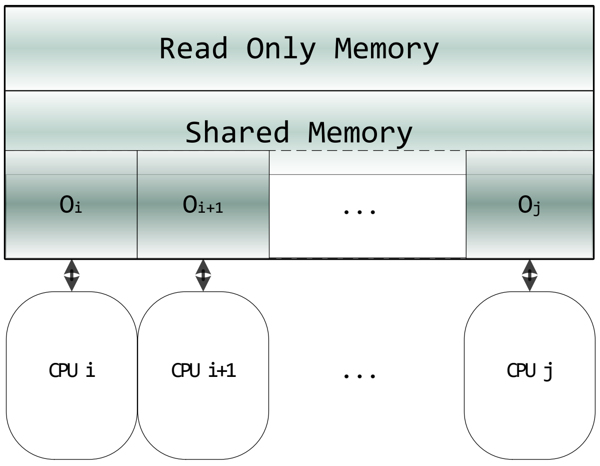

Figure 3.3: $MIPS_P$ memory partitioning into the ownership sets.

$$\begin{aligned}
&\mathit{ownership\text{-}inv}(o \in \mathcal{O}) \in \mathbb{B} \overset{def}{=}\\
&\qquad \mathbb{A} = o.\mathbb{A}^{pr} \cup o.\mathbb{A}^{sh} \cup \mathbb{A}^{ro}\\
&\qquad \wedge\, o.\mathbb{A}^{pr} \cap \mathbb{A}^{ro} = \emptyset\\
&\qquad \wedge\, o.\mathbb{A}^{sh} \cap \mathbb{A}^{ro} = \emptyset\\
&\qquad \wedge\, \forall a, i, j.\ a \in o.O_i \wedge a \in o.O_j \implies i = j
\end{aligned}$$

It states, that:

- the complete address space $\mathbb{A}$ is covered by the ownership,
- the read only memory is disjoint with the private and shared memory,
- the owns-sets of different processors are disjoint.

Every ownership that satisfies *ownership-inv* we call valid.

As we mentioned, the addresses of the shared and the private memory can change their ownership state. They can have different owners and be shared or not during the execution. Thus for an execution sequence with n steps we define a sequence of $n+1$ ownership states o_0 to o_n. The changes in the ownership are represented by the difference between every two consecutive ownership o_i and o_{i+1}. We call these changes ownership transfer. We define the validity of ownership transfer by the following five rules as depicted in Figure 3.4.

1. Shared unowned addresses can be exclusively acquired by some processor and get exclusively owned.

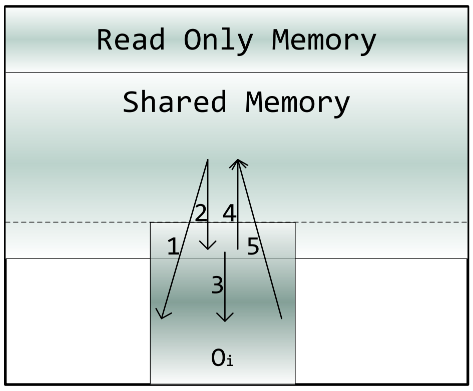

Figure 3.4: Ownership transfer

2. Shared unowned addresses can be acquired by some processor and get shared owned.

3. Shared owned addresses can be exclusively acquired by their owner and get exclusively owned.

4. Shared owned addresses can be released by their owner and become shared unowned.

5. Exclusively owned addresses can be released by their owner and become shared unowned.

◀ **Definition** 3.7
Safe Transfer

Ownership transfer can happen only on I/O steps between the owns-set of the corresponding processor and the set of shared addresses. The predicate *transfer* denotes in a compendious form the transfer rules for an address a, where O and O' denote the old and the new owns-sets, and sh and sh' denote the old and the new sets of shared addresses.

$$\begin{aligned} transfer(O \in 2^{\mathbb{B}^{32}}, O' \in 2^{\mathbb{B}^{32}}, sh \in 2^{\mathbb{B}^{32}}, sh' \in 2^{\mathbb{B}^{32}}, a \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \\ (a \in O \cup sh \Longleftrightarrow a \in O' \cup sh') \\ \wedge\ (a \notin sh \wedge a \in sh' \implies a \in O \wedge a \notin O') \end{aligned}$$

The predicate *safetransfer* denotes whether the ownership pair o and o' satisfies the validity for ownership transfer on steps of processor i, where io is a flag indicating I/O steps.

$$\mathit{safetransfer}(i \in \mathit{Pid}, io \in \mathbb{B}, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B} \overset{def}{=}$$
$$\begin{cases} \forall a \in \mathbb{B}^{32}.\ \mathit{transfer}(o.O_i, o'.O_i, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i, a) & \mathit{if}\ io \\ \quad \wedge \forall j \in \mathit{Pid}.\ j \neq i \implies (a \in o.O_j \iff a \in o'.O_j) & \\ \qquad\qquad\qquad\qquad \wedge (o.O_j \cap o.\mathbb{A}^{sh} = o'.O_j \cap o'.\mathbb{A}^{sh}) & \\ o = o' & \mathit{otherwise} \end{cases}$$

Memory Accesses are considered safe in respect to ownership if they follow the following rules.

- Local steps of a given processor can read the read-only memory and addresses owned by the processor.
- Local steps of a given processor can write only addresses exclusively owned by the processor.
- I/O steps of a given processor can read the read only memory, the shared memory and addresses owned by the processor.
- I/O steps of a given processor can write addresses owned by the processor and all shared unowned addresses.

Definition 3.8 ▶ Memory Access Policy

The predicate $\mathit{safeacc}$ defines our policy for reading and writing memory addresses by a processor i based on the ownership setting o. R and W represent the sets of addresses to be read and/or written. io is a flag denoting whether we are executing an I/O step or not.

$$\mathit{safeacc}(i \in \mathit{Pid}, io \in \mathbb{B}, R \in 2^{\mathbb{B}^{32}}, W \in 2^{\mathbb{B}^{32}}, o \in \mathcal{O}) \overset{def}{=}$$
$$\begin{cases} (R \subseteq o.O_i \cup o.\mathbb{A}^{sh} \cup \mathbb{A}^{ro}) \wedge (W \subseteq o.O_i \cup (o.\mathbb{A}^{sh} \setminus o.\overline{O}_i)) & \mathit{if}\ io \\ (R \subseteq o.O_i \cup \mathbb{A}^{ro}) \wedge (W \subseteq o.O_i \setminus o.\mathbb{A}^{sh}) & \mathit{otherwise} \end{cases}$$

3.2.2 Safe Execution

In order to give a formal definition for safe MIPS_P execution we have to define the memory footprint of MIPS_P steps. Furthermore we have to define the set of addresses accessible to the execution and the read only part of it.

Definition 3.9 ▶ MIPS_P Static Ownership Information

The complete set of addresses of the MIPS_P execution is trivially defined as the full domain of the MIPS_P memory.

$$\mathbb{A}_{\mathit{MIPS}_P} \overset{def}{=} \mathbb{B}^{32}$$

The read only part contains the the memory region where the compiled code resides $\mathbb{A}_{code} \subset \mathbb{B}^{32}$ and the read only data of the compiled program, which we here denote by $\mathbb{A}^{ro}{}_{\textsf{C-IL}} \subset \mathbb{B}^{32}$. Both are defined later in Chapter 5.

$$\mathbb{A}^{ro}{}_{MIPS_P} \overset{def}{=} \mathbb{A}_{code} \cup \mathbb{A}^{ro}{}_{\textsf{C-IL}}$$

The set of accessed addresses depends on the fact, whether the current instruction is interrupted. For convenience we introduce a new predicate to denote external interrupts using the external interrupt flag from our extended alphabet.

The predicate $jisr_{IPI}$ denotes whether the next transition is a core step with an active JISR signal due to a pending IPI request. ◀ **Definition** 3.10 JISR Transition Predicate

$$jisr_{IPI}(\alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} core(\alpha) \wedge \alpha.ex$$

At some places it is important to denote the processor id of the processor being interrupted. Therefore we define a second version of the predicate, which is additionally parametrized with the id of a particular processor.

$$jisr_{IPI}(\alpha \in \Sigma_M, i \in Pid) \in \mathbb{B} \overset{def}{=} core(\alpha) \wedge \alpha.ex \wedge \alpha.pid = i$$

The set of all memory addresses that are read during the execution of a given step α is defined by the function $reads$. ◀ **Definition** 3.11 Read Addresses

$$reads(core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8, \alpha \in \Sigma_M) \in 2^{\mathbb{B}^{32}} \overset{def}{=}$$
$$\begin{cases} f(core) \cup r(core, m) & \text{if } core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \\ & \quad \wedge \neg(iint(core, I) \vee load_{\mathbb{A}_{apic}}(core, I)) \\ f(core) & \text{if } core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \\ & \quad \wedge (iint(core, I) \vee load_{\mathbb{A}_{apic}}(core, I)) \\ \emptyset & otherwise \end{cases}$$

where $f(core)$ denotes the bytes of the instruction being fetched

$$f(core) = \{core.pc, \ldots, core.pc +_{32} 3\} \ ,$$

and $r(core, m)$ denotes the addresses read during intruction execution

$$r(core, m) = \begin{cases} \{ea(core, I), \ldots, ea(core, I) +_{32} (d(I) - 1)\} & \text{if } (load(I) \vee rmw(I)) \\ \emptyset & otherwise \end{cases}$$

and I is the current instruction

$$I = I(h.core_{\alpha.pid}, m) \ .$$

Definition 3.12 ▶ Written Addresses

The set of all memory addresses that are written during the execution of a given step α is defined by the function *writes*.

$$writes(core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8, \alpha \in \Sigma_M) \in 2^{\mathbb{B}^{32}} \overset{def}{=}$$
$$\begin{cases} w(core, m) & \text{if} \;\; core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \\ & \quad \wedge \neg iint(core, I) \\ & \quad \wedge \neg store_{\mathbb{A}_{apic}}(core, I) \\ \emptyset & \text{otherwise} \end{cases}$$

where $w(core, m)$ denotes the set of addresses written during instruction execution and the predicate $swap(core, m, I)$ denotes a read-modify-write instruction with successful compare operation

$$w(u, m) = \begin{cases} \{ea(u, I), \ldots, ea(u, I) +_{32} (d(I) - 1)\} & \text{if} \;\; (store(I) \\ & \quad \vee swap(u, m, I)) \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$I = I(h.core_{\alpha.pid}, m)) \; .$$

Definition 3.13 ▶ Safe $MIPS_P$ Step

A $MIPS_P$ step defined by input α and starting in configuration h is ownership-safe according to the ownership setting pair o and o' if it obeys the memory access policy and maintains the ownership invariant.

$$safestep(h \in C_M, \alpha \in \Sigma_M, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B} \overset{def}{=}$$
$$safeacc(\alpha.pid, IOstep(h, \alpha), rs(h, \alpha), ws(h, \alpha), o)$$
$$\wedge \; safetransfer(\alpha.pid, IOstep(h, \alpha), o, o')$$

where the functions $rs(h, \alpha)$ and $ws(h, \alpha)$ denote the set of addresses which are read and respectively written by α.

$$rs(h, \alpha) = reads(h.core_{\alpha.pid}, h.m, \alpha)$$

$$ws(h, \alpha) = writes(h.core_{\alpha.pid}, h.m, \alpha)$$

The predicate $IOstep(h, \alpha)$ is defined in the next section and denotes whether the next step is an I/O step.

Definition 3.14 ▶ Safe Execution Sequence

A $MIPS_P$ execution sequence defined by initial configuration h and input sequence β is ownership-safe according to initial ownership o and final ownership o' if there exists an ownership sequence such that every step of the execution is ownership-safe. IPI steps are ownership-safe by definition.

$$\begin{aligned}
&\mathit{safeseq}_{MIPS_P}(h \in C_M, \beta \in (\Sigma_M)^*, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \beta = \varepsilon \wedge o = o' \wedge \mathit{ownership\text{-}inv}(o) \\
&\quad \vee \exists o^0, ..., o^n \in \mathcal{O}.\ o^0 = o \wedge o^n = o' \\
&\qquad \wedge \forall i < n.\ \mathit{ownership\text{-}inv}(o^i) \wedge (\mathit{safestep}(h^i, \beta_i, o^i, o^{i+1}) \vee \mathit{ipi}(\beta_i))
\end{aligned}$$

3.3 Auxiliary Definitions for Order Reduction

In order to talk formally about the COSMOS reordering theorem we introduce some definitions. We use them in the formal instantiation of the COSMOS model and in further sections, when we talk about $MIPS_P$ execution sequences.

3.3.1 I/O Steps

Next, we identify the I/O steps in a $MIPS_P$ execution.

The majority of the instructions executed by our machine are generated by compiler from *C-IL* code. In *C-IL* shared variables have "volatile" qualifier, thus the compiler identifies accesses to shared data in the compiled code. We denote the set of addresses of generated instructions with shared data access by A_{io}. In addition we have to identify the I/O steps which are not executing instructions but still access some shared component. Depending on the processors mode we distinguish two kinds of I/O steps. For the execution in system mode we define the set of hypervisor I/O steps.

The next step is hypervisor I/O step if it is JISR or $eret$ due to IPI, or if it accesses shared memory. ◀ **Definition** 3.15 Hypervisor I/O Steps

$$\begin{aligned}
&\mathit{IOstep}_{HV}(c \in C_{CORE}, \alpha \in \Sigma_M, I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \mathit{jisr}_{IPI}(\alpha) \\
&\quad \vee (\mathit{core}(\alpha) \wedge \mathit{eret}_{IPI}(c, I)) \\
&\quad \vee (\mathit{core}(\alpha) \wedge \neg \mathit{iint}(c, I) \wedge (c.pc \in A_{io}))
\end{aligned}$$

In Definition 3.15 we implicitly assume the following software condition and consider therefore only shared accesses originating from the *C-IL* code.

Software Condition 3 (Hypervisor Assembler Portions) *All memory accesses in the assembler portions of the hypervisor implementation are local steps.*

The set of guest I/O steps is defined by the predicate IOstep_G and it consists of all steps executed in guest mode. ◀ **Definition** 3.16 Guest I/O Steps

$$\mathit{IOstep}_G(\alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} \mathit{guest}(\alpha)$$

In our model guest steps does not access any shared component. Still in possible extensions of $MIPS_P$ it may happen that they do, e.g. if MMU steps are accessing shared memory. Since we do not want to restrict the applicability of our theory, we chose the weakest possible condition on guest interleaving points and basically forbid reordering of guest steps.

Definition 3.17 ▶ I/O Steps

The overall set of I/O steps consists of the hypervisor I/O steps, the guest I/O steps and the IPI steps.

$$IOstep(c \in C_{CORE}, \alpha \in \Sigma_M, I \in \mathbb{B}^{32}) \in \mathbb{B} \overset{def}{=} IOstep_{HV}(c, \alpha, I) \vee IOstep_G(\alpha) \vee ipi(\alpha)$$

Definition 3.18 ▶ Predicate Overloading

For convenience when we consider executions of the multiprocessor machine rather than a single core we use the following (overloaded) definitions of I/O steps.

$$IOstep_{HV}(h \in C_M, \alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} core(\alpha) \wedge IOstep_{HV}(h.core_{\alpha.pid}, \alpha, I(h.core_{\alpha.pid}, h.m))$$

$$IOstep(h \in C_M, \alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} IOstep_{HV}(h, \alpha) \vee IOstep_G(\alpha) \vee ipi(\alpha)$$

Next in order to shorten notation we overload the predicate $eret_{IPI}$ from Definition 2.36.

Definition 3.19 ▶ External $eret$ Shorthand

The predicate $eret_{IPI}$ denotes whether the next step is a core step of a given processor i that executes an $eret$ instruction which ends an ISR of an IPI.

$$eret_{IPI}(h \in C_M, \alpha \in \Sigma_M, i \in Pid) \overset{def}{=} core(\alpha) \wedge \alpha.pid = i \wedge eret(I(h.core_{\alpha.pid}, h.m)) \wedge h.core_{\alpha.pid}.eca[1]$$

3.3.2 Interleaving points

The set of interleaving points is defined for a given program and compiler. The compiler identifies consistency points in the *C-IL* execution, points in which compiler consistency holds between the *C-IL* machine and the $MIPS_P$ machine executing the compiled code. Knowing the step that starts in a consistent *C-IL* state, the compiler can determine the instruction that is executed in the corresponding $MIPS_P$ state. Furthermore our choice of interleaving points relates to a condition of the COSMOS reordering theorem. It is required that between every two I/O steps of a given execution there exists at least one interleaving point. Similar to the I/O steps, interleaving points are defined depending on the processors mode. Most of hypervisor interleaving points originate from the compiled program. By

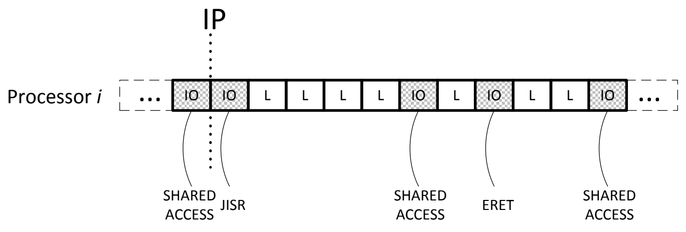

Figure 3.5: Hypervisor execution sequence with IPI handling, where the ISR takes place right after a hypervisor I/O step.

$\mathbb{A}_{cp}$ we denote a set of instruction addresses provided by the compiler. Whenever the program counter in a $MIPS_P$ processor points to an instruction whose address is in $\mathbb{A}_{cp}$, compiler consistency has to hold between the $MIPS_P$ machine and the corresponding *C-IL* configuration. Here we consider $\mathbb{A}_{cp}$ from the point of view of the $MIPS_P$ execution and the required minimal set of interleaving points, but compilers are free to define more interleaving points than the ones listed bellow.

First we consider a hypervisor execution sequence.

In the absence of IPIs all hypervisor I/O steps in a $MIPS_P$ execution are instructions with shared memory accesses. Due to Software Condition 3 we know that all shared memory accesses originate from *C-IL* steps processing a shared access. Thus, in order to satisfy the IOIP condition in the absence of IPIs, it is sufficient to have a requirement on the compiler to place an interleaving point between every two I/O steps. When we include IPIs the situation changes significantly due to the following facts:

- JISR and $eret$ are I/O steps.
- IPIs can occur on the boundary of any two instructions of the program.

These facts imply that JISR may happen immediately after a shared access I/O step (Figure 3.5) and also that $eret$ may be followed by a shared access I/O step (Figure 3.6).

If we consider Figure 3.5 and take into the account that the compiler can not guarantee that JISR occurs only at interleaving points, the only way to satisfy the IOIP condition is to have an interleaving point right after every shared access I/O step. Thus A_{cp} contains the addresses of all instructions that follow the instructions defined by A_{io}. To guarantee this the compiler have to enforce certain programing discipline and exclude optimizations in its code generation algorithm, which may break the requirement, that every shared access ends in a consistent state.

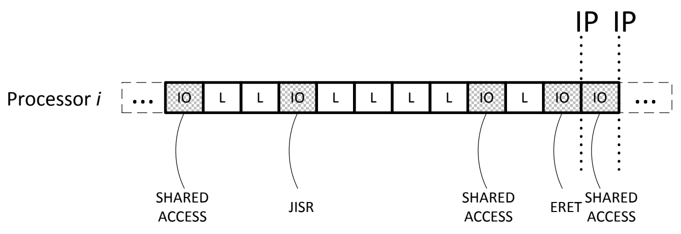

Figure 3.6: Hypervisor execution sequence with IPI handling, where the ISR takes place right before a hypervisor I/O step.

In Figure 3.6 we depict the fact that a step executing $eret$ due to an IPI can be followed by a shared access I/O step. Since we already require from the compiler to guarantee that shared access I/O steps end in a consistent state and *C-IL* statements with shared accesses can be translated into several $MIPS_P$ instructions we can not define that shared access I/O steps start in a consistent state. Therefore, in order to satisfy the COSMOS condition, we have to define the state immediately after the $eret$ step as an interleaving point.

A JISR step ends in a state, which is obviously not consistent. We have to find a subsequent state in the execution, before the next I/O step, which we define as an interleaving point. In Chapter 5 we assume that the compiler will set an interleaving point at the beginning of every *C-IL* function. We know that our service routine contains the execution of a handler implemented in *C-IL*. Furthermore, due to Software Condition 3 we know that the first I/O step after JISR can only originate from the *C-IL* part of the handler. Thus the required interleaving point after JISR is in our case the one at the beginning of the *C-IL* IPI handler, which is also assumed to be defined by the set $\mathbb{A}_{cp}$.

In Figure 3.7 we depict an execution that contains all possible hypervisor I/O steps. In Chapter 6 we prove that every interleaving point defined by us is a state that satisfies compiler consistency

Definition 3.20 ▶ Hypervisor interleaving points

In an execution sequence defined by the initial configuration h and the input sequence β the configuration h^k is a hypervisor interleaving point if the next step executes an instruction from $\mathbb{A}_{cp}$ or if h^k is the first configuration after IPI service routine.

$$\begin{aligned} hypIP(h \in C_M, \beta \in (\Sigma_M)^*, k \in [0 : |\beta| - 1]) \in \mathbb{B} &\overset{def}{=} \\ (core(\beta_k) \wedge h^k.pc_{\beta_k.pid} \in \mathbb{A}_{cp}) & \\ \vee\, (core(\beta_{k-1}) \wedge eret_{IPI}(h^{k-1}.core_{\beta_{k-1}.pid}, I)) & \end{aligned}$$

where

$$I = I(h^{k-1}.core_{\beta_{k-1}.pid}, h^{k-1}.m)$$

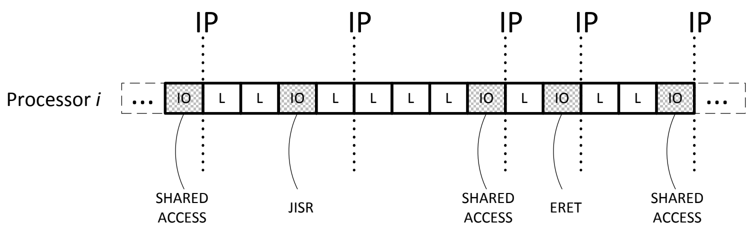

Figure 3.7: Hypervisor execution sequence with IPI handling and program thread and handler I/O steps.

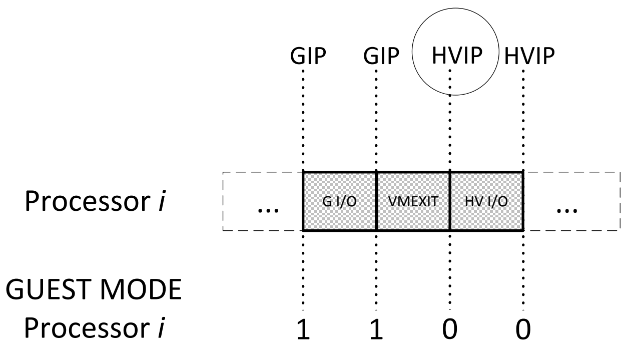

Figure 3.8: Processor execution sequence with guest steps followed by a hypervisor I/O step.

Since every guest step is an I/O step we need to define an interleaving point between consecutive guest steps.

Every configuration in guest mode is a guest interleaving point.

◀ **Definition** 3.21
Guest Interleaving Points

$$guestIP(h \in C_M, \beta \in (\Sigma_M)^*, k \in [0 : |\beta| - 1]) \in \mathbb{B} \overset{def}{=} IOstep_G(\beta_k)$$

Additionally we need one more interleaving point at the end of every guest execution. According to the definition of guest I/O steps (Definition 3.16), VMEXIT is an I/O step. Since there could be a hypervisor I/O step after VMEXIT, we set an interleaving point after VMEXIT (Figure 3.8). From the program point of view the first step after the context switch is defined by the instruction following

VMRUN. VMRUN is implemented by a compiler intrinsic on *C-IL* level. Thus this interleaving point is also identified by the compiler and included in $\mathbb{A}_{cp}$.

Definition 3.22 ▶ Interleaving Points

The overall set of interleaving points consists of

- all hypervisor interleaving points,
- all guest interleaving points,
- all configurations after an IPI step and
- all configurations in which some processor does its first step.

$$\begin{aligned}&IP(h \in C_M, \beta \in (\Sigma_M)^*, k \in [0 : |\beta| - 1]) \in \mathbb{B} \overset{def}{=} \\ &\qquad ipi(\beta_{k-1}) \vee guestIP(h, \beta, k) \vee hypIP(h, \beta, k) \\ &\qquad \vee (\exists i \in Pid.\ k = min\{j \mid i = \beta_j.pid\})\end{aligned}$$

3.3.3 Execution Sequences

Definition 3.23 ▶ Local Steps

Given a step sequence β of length n the function *coresteps* returns the set of indexes of steps of processor i contained in β.

$$coresteps(i \in Pid, \beta \in (\Sigma_M)^*) \in 2^{\mathbb{N}} \overset{def}{=}$$
$$\begin{cases} \emptyset & if\ \beta = \varepsilon \\ coresteps(i, \beta[0 : n-2]) \cup \{n-1\} & if\ \beta \neq \varepsilon \wedge \beta_{n-1}.pid = i \\ coresteps(i, \beta[0 : n-2]) & if\ \beta \neq \varepsilon \wedge \beta_{n-1}.pid \neq i \end{cases}$$

Definition 3.24 ▶ Local Steps Subsequence

Given a step sequence β of length n the function *coreseq* returns the subsequence of steps of processor i contained in β.

$$coreseq(i \in Pid, \beta \in (\Sigma_M)^*) \in (\Sigma_M)^* \overset{def}{=}$$
$$\begin{cases} \varepsilon & if\ \beta = \varepsilon \\ coreseq(i, \beta[0 : n-2]) \circ \beta_{n-1} & if\ \beta \neq \varepsilon \wedge \beta_{n-1}.pid = i \\ coreseq(i, \beta[0 : n-2]) & if\ \beta \neq \varepsilon \wedge \beta_{n-1}.pid \neq i \end{cases}$$

Definition 3.25 ▶ I/O Steps Subsequence

Given a step sequence β of length n and an initial configuration h the function *IOsteps* returns subsequence of I/O steps contained in β.

$$IOsteps(h \in C_M, \beta \in (\Sigma_M)^*) \in (\Sigma_M)^* \overset{def}{=}$$
$$\begin{cases} \varepsilon & if\ \beta = \varepsilon \\ IOsteps(h, \beta[0 : n-2]) \circ \beta_{n-1} & if\ \beta \neq \varepsilon \wedge IOstep(h_{n-1}, \beta_{n-1}) \\ IOsteps(h, \beta[0 : n-2]) & if\ \beta \neq \varepsilon \wedge \neg IOstep(h_{n-1}, \beta_{n-1}) \end{cases}$$

We consider two step input sequences $\beta, \omega \in (\Sigma_M)^*$ equivalent if they contain the same number of steps, the local order of steps of every processor is the same and the global order the I/O steps is the same.

◀ **Definition** 3.26 Equivalent Execution Sequence

$$\begin{aligned} \beta \overset{o}{=} \omega \overset{def}{=} \quad & |\beta| = |\omega| \\ & \wedge \forall i \in Pid.\ coreseq(i, \beta) = coreseq(i, \omega) \\ & \wedge \forall h \in C_M.\ IOsteps(h, \beta) = IOsteps(h, \omega) \end{aligned}$$

The $IOIP$ predicate denotes that in a given execution sequence

◀ **Definition** 3.27 IOIP Condition

- the first step of every processor starts at an interleaving point and
- at least one interleaving point exists between every two I/O steps of every processor.

$$\begin{aligned} IOIP(h \in C_M, \beta \in (\Sigma_M)^*) \in \mathbb{B} \overset{def}{=} & \forall pid \in Pid, k \in [0 : |\beta| - 1]. \\ & (k = min\{i \mid i \in coresteps(pid, \beta)\} \implies IP(h, \beta, k)) \\ & \wedge \forall i, j \in coresteps(pid, \beta).\ IOstep(h^i, \beta_i) \wedge IOstep(h^j, \beta_j) \wedge i < j \\ & \qquad \implies \exists l \in (coresteps(pid, \beta) \cap [i + 1 : j]).\ IP(h, \beta, l) \end{aligned}$$

The predicate $IPsched$ denotes that in a given execution sequences, steps of different units are interleaved only at interleaving points. With other words, every two consecutive steps either belong to the same unit or are separated by an interleaving point.

◀ **Definition** 3.28 IP Schedule

$$IPsched(h \in C_M, \beta \in (\Sigma_M)^*) \in \mathbb{B} \overset{def}{=}$$
$$\begin{cases} 1 & if\ \beta = \varepsilon \\ 1 & if\ |\beta| = 1 \\ IPsched(h, \beta[0 : |\beta| - 2]) & if\ |\beta| > 1 \wedge IP(h, \beta, |\beta| - 1) \\ IPsched(h, \beta[0 : |\beta| - 2]) & \\ \quad \wedge \beta_{n-1}.pid = \beta_{n-2}.pid & otherwise \end{cases}$$

3.4 Instantiation of COSMOS with $MIPS_P$

In the following lines we show how the COSMOS model can be instantiated with $MIPS_P$. Christoph Baumann defines in [Bau14] an instantiation interface for COSMOS and presents an instantiation with a simple $MIPS$ model without APIC. As we show later, the formal instantiation of COSMOS with an APIC-containing model requires some smart technical refinements.

3.4.1 COSMOS Instantiation Interface

The COSMOS instantiation interface is defined by the signature of the model.

$$S \overset{def}{=} [\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP}]$$

It consists of types for memory addresses ($\mathcal{A}$) and values ($\mathcal{V}$), a dedicated set of read-only addresses ($\mathcal{R}$), number of computational units nu, computational unit configuration type $\mathcal{U}$, set of external inputs $\mathcal{E}$, function defining the set of addresses read by the next step $reads$, transition function δ, functions defining the set of I/O steps and interleaving points $\mathcal{IO}$ and $\mathcal{IP}$ respectively. We instantiate COSMOS by instantiating the types and implementing the functions maintaining their signatures. For the instantiated model we have to prove that the reads-set depends only on the memory content of the read addresses. Furthermore, we have to maintain the characteristics of the model. In COSMOS the only shared resource can be the memory. Therefore in the instantiation we have to handle the APIC as part of the memory. Furthermore, in COSMOS every step is performed by some computational unit. Thus we need to add an artificial unit for performing the IPI steps.

Definition 3.29 ► Instantiated COSMOS Machine

The configuration of a COSMOS machine instantiated with $MIPS_P$ is defined by the record S_{MIPS_P}.

$$S_{MIPS_P} \overset{def}{=} [\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP}]$$

- $S_{MIPS_P}.\mathcal{A} = (\mathbb{B}^{32} \setminus \mathbb{A}_{apic}) \cup Pid$
  The set of memory addresses excludes the region mapped to the APICs but includes the set of processor identifiers. The APIC of processor i can be accessed over memory port i, where the memory distinguishes two different addressing modes. Usual 32-bit vectors address memory bytes, whereas natural number processor identifiers refer to the APIC of the corresponding processor.

- $S_{MIPS_P}.\mathcal{V} = \mathbb{B}^8 \cup C_{APIC}$
  The set of memory values contains byte values and APIC configurations.

- $S_{MIPS_P}.\mathcal{R} = \mathbb{A}^{ro}{}_{MIPS_P}$
  We set the read-only memory to be defined by $\mathbb{A}^{ro}{}_{MIPS_P}$.

- $S_{MIPS_P}.nu = np + 1$
  We have $np + 1$ computational units. In addition to the np processors we have a computational unit performing the IPI steps.

- $S_{MIPS_P}.\mathcal{U} = [core \in C_{CORE}, tlb \in C_{TLB}] \cup C_{ipi}$
  The state of the first np units is defined by the core and the TLB of the

corresponding processor. The last unit executes only IPI steps and has an uninterpreted configuration state C_{ipi}.

In the following we use all functions defined in Chapter 2 with unit parameters instead of core parameters, e.g. we define the next instruction by

$$I(u, m) \overset{def}{=} I(u.core, m).$$

In the following definitions we use I as a shorthand for $I(u, m)$.

- $S_{MIPS_P}.\mathcal{E} = \Sigma_M$
  The set of external inputs is defined by the labels of our transition system.

- $S_{MIPS_P}.reads$ defines the set of read addresses for every step. The addresses of the instruction being fetched $f(u)$ and the addresses read during instruction execution $r(u, m)$ are defined as in Section 3.2.

  In case of steps that access the APIC we have to add the APIC addresses to the reads-set.

$$S_{MIPS_P}.reads(u, m, \alpha) = \begin{cases} f(u) \cup r(u, m) & \mathit{if}\ \ core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \\ & \quad \wedge \neg(iint(u, I) \vee load_{\mathbb{A}_{apic}}(u, I)) \\ f(u) \cup \{\alpha.pid\} & \mathit{if}\ \ core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \wedge \neg iint(u, I) \\ & \quad \wedge (load_{\mathbb{A}_{apic}}(u, I) \vee eret_{IPI}(u, I)) \\ f(u) & \mathit{if}\ \ core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \wedge iint(u, I) \\ \{\alpha.pid\} & \mathit{if}\ \ jisr_{IPI}(\alpha) \\ \{pid\} & \mathit{if}\ \ \alpha = (\textbf{ipi}, pid) \\ \emptyset & \mathit{otherwise} \end{cases}$$

- $S_{MIPS_P}.\delta$ defines the transition function for the units of the instantiated COSMOS machine.

  The set of addresses written $w(u, m)$ during instruction execution of instruction I is defined as in Section 3.2. The overall set of addresses written by the transition function is defined as follows.

$$W(u, m, \alpha) = \begin{cases} w(u, m) & \mathit{if}\ \ core(\alpha) \wedge \neg jisr_{IPI}(\alpha) \wedge \neg iint(u, I) \\ & \quad \wedge \neg eret_{IPI}(u, I) \wedge \neg store_{\mathbb{A}_{apic}}(u, I) \\ \{\alpha.pid\} & \mathit{if}\ \ jisr_{IPI}(\alpha) \\ & \quad \vee (core(\alpha) \wedge \neg iint(u, I) \\ & \qquad \wedge (store_{\mathbb{A}_{apic}}(u, I) \vee eret_{IPI}(u, I))) \\ \{0, \ldots, np - 1\} & \mathit{if}\ \ ipi(\alpha) \\ \emptyset & \mathit{otherwise} \end{cases}$$

As described in [Bau14], $S_{MIPS_P}.\delta$ gets as an input the state of the unit executing the step, a partial memory defined by the reads set of the step, and an external input. The transition function defines the new state of the unit and an output partial memory, which represents the updated part of memory for the step.

$$(u', m') = S_{MIPS_P}.\delta(u, m, \alpha)$$

Since the transition function of the $MIPS_P$ machine is defined for a total memory, we define a memory M which is a total function by filling in zeros for the addresses not covered by m.

$$M(a) = \begin{cases} m(a) & \mathit{if}\ m(a) \neq \bot \\ 0^8 & \mathit{otherwise} \end{cases}$$

We define the transition function for the four types of possible inputs, i.e. core steps, IPI steps, guest steps, and VMEXIT steps.

* $(u', m') = S_{MIPS_P}.\delta(u, m, (\textbf{core}, pid, ex))$
  The state u' of the processing unit u is defined by the core and TLB transition functions from previous chapter.

$$u'.core = \delta^{core}(u.core, eev, I, R)$$

$$u'.tlb = \begin{cases} \delta^{tlb}(u.tlb, \mathit{flush}) & \mathit{if}\ \mathit{flush}(I) \\ u.tlb & \mathit{otherwise} \end{cases}$$

  where

$$eev = \begin{cases} eev(m(pid)) & \mathit{if}\ ex \\ 0^{256} & \mathit{otherwise} \end{cases}$$

$$I = M_4(u.pc)$$

$$addr = ea(u.core, I)$$

$$R = ms_{d(I)}(m(pid), M)(addr)$$

  The new memory state is defined by the writes, which do not go to the APIC range, and by the changes of the APIC. We first define M' - the state of the total memory M after the execution of the step, and $apic'$ - the new APIC state. Then we construct the output partial memory m' using M' and $apic'$.

$$M' = \begin{cases} \delta^{mem}(M, (addr, data)) & \text{if } store(I) \wedge addr \notin \mathbb{A}_{apic} \\ & \quad \wedge \neg isJISR(u, eev, I) \\ M & otherwise \end{cases}$$

$$apic' = \begin{cases} \delta^{apic}(m(pid), \textbf{jisr}) & \text{if } jisr_{IPI}(\textbf{core}, pid, ex) \\ \delta^{apic}(m(pid), \textbf{eret}) & \text{if } eret_{IPI}(u, I) \\ \delta^{apic}(m(pid), (apic_{offset}(addr), data)) & \text{if } store_{\mathbb{A}_{apic}}(u, I) \\ & \quad \wedge \neg isJISR(u, eev, I) \\ m(pid) & otherwise \end{cases}$$

$$m'(a) = \begin{cases} M'(a) & \text{if } a \in w(u, m) \\ apic' & \text{if } a = pid \\ \bot & otherwise \end{cases}$$

where

$$data = sv(u.core, I)$$

* $(u', m') = S_{MIPS_P}.\delta(u, m, (\textbf{ipi}, pid))$
  IPI steps change only the APIC state in the memory.

$$u' = u$$

$$m'(i) = \begin{cases} \delta^{apic}(m(i), (mt, vector))[DS \mapsto 0] & \text{if } i = pid \wedge target(i) \\ m(i)[DS \mapsto 0] & \text{if } i = pid \wedge \neg target(i) \\ \delta^{apic}(m(i), (mt, vector)) & \text{if } i \neq pid \wedge target(i) \\ m(i) & otherwise \end{cases}$$

where

$$mt = m(pid).ICR.MT$$
$$vector = m(pid).ICR.VEC$$

$$\begin{aligned} target(i) = & m(pid).ICR.DSH = ALL \\ & \vee m(pid).ICR.DSH = ALL\text{-}BUT\text{-}SELF \wedge i \neq pid \\ & \vee m(pid).ICR.DSH = SELF \wedge i = pid \\ & \vee m(pid).ICR.DSH = ID \wedge h.APIC\_ID_i = m(pid).ICR.DEST \end{aligned}$$

- $(u', m') = S_{MIPS_P}.\delta(u, m, (\textbf{guest}, pid))$
  Guest steps change only the TLB of the current unit.

$$u'.tlb = \delta^{tlb}(u.tlb, add)$$

- $(u', m') = S_{MIPS_P}.\delta(u, m, (\textbf{vmexit}, pid))$
  VMEXIT steps change only the mode of the current unit.

$$u'.core.mode[0] = 0$$

- $S_{MIPS_P}.\mathcal{IO}(u, m, \alpha) \overset{def}{=} IOstep(u.core, \alpha, I(u, m))$
  The set of IO steps is defined in Definition 3.17.

- $S_{MIPS_P}.\mathcal{IP}$ The set of interleaving points is defined in Section 3.3.2.

  Formally the signature of the COSMOS function differs from the signature of the function from Definition 3.22. $S_{MIPS_P}.\mathcal{IP}$ has parameters defining only the next step, whereas IP is defined on the execution sequence. $S_{MIPS_P}.\mathcal{IP}$ is inconvenient for defining interleaving points based on a step that has already been performed. One has to extend the model with a ghost history variable, which tells whether the last step was an $eret$ step due to an IPI or an IPI step. We omit presently here this simple extension according to our policy to keep the model simple and leave details out, which do not contribute neither to better understanding nor to expressing some important property of the proofs.

3.4.2 Instantiation Restriction for $reads$

The restriction for the COSMOS instantiation requires the reads-sets for the next step defined on two different memory states to be the same if the two memories are equal at the addresses of the reads-set. The set defined by $S_{MIPS_P}.reads$ depends on the configuration of the computational unit, the input parameter and in some cases on the fetched instruction. Which instruction we fetch depends only on the processor's program counter and on the four consecutive memory bytes starting at the address stored in the program counter. These are the only memory cells which influence the output of $S_{MIPS_P}.reads$. Since in our instantiation the instruction bytes are included in the reads-set, S_{MIPS_P} obviously satisfies the requirement.

3.4.3 Reordering Theorem

As a result of the instantiation we can now apply the COSMOS reordering theorem to S_{MIPS_P} execution sequences. It says that if all IP schedule execution sequences starting in a given configuration fulfill the IOIP condition and are ownership safe, then we can deduce ownership safety for all execution sequences.

Since we can easily prove that every trace of the $MIPS_P$ can be simulated by an execution of S_{MIPS_P} and vice versa the two models are isomorphic. Thus we can claim the same properties for $MIPS_P$ execution sequences.

For $MIPS_P$ every possible trace fulfill the IOIP condition by our definition of I/O steps and interleaving points. All I/O steps except JISR and guest I/O steps end in an interleaving point.

- After JISR according to Software Condition 3 a processor executes only local steps before the next interleaving point, where it starts executing compiled code.
- Guest interleaving steps are followed by VMEXIT, which ends in an interleaving point.

This allows us to simplify the $MIPS_P$ version of the COSMOS reorder theorem in Definition 3.30. Additionally we expose a lemma from [Bau14], which says that every sequence satisfying the IOIP condition can be reordered into an equivalent IP schedule sequence.

◀ **Definition** 3.30 $MIPS_P$ Order Reduction Theorem

If all $MIPS_P$ IP schedule execution sequences starting in a given configuration h are ownership safe, then we can deduce ownership safety for every execution sequence β, and β can be reordered into an equuivalen IP schedule ω.

$$\begin{aligned}
&\forall h, \beta, o.\\
&\qquad (\mathit{ownership\text{-}inv}(o)\\
&\qquad \wedge \forall \gamma.\ \exists o'.\ (\mathit{IPsched}(h,\gamma) \implies \mathit{safeseq}_{MIPS_P}(h,\gamma,o,o')))\\
&\qquad \implies (\exists o''.\ \mathit{safeseq}_{MIPS_P}(h,\beta,o,o'')\\
&\qquad\quad \wedge\ \exists \omega.\ (\omega \overset{o}{=} \beta \wedge \mathit{IPsched}(h,\omega)))
\end{aligned}$$

As a consequence of the theorem we will consider from now on only IP schedule execution sequences. In that to prove ownership safety of all executions it is sufficient to prove ownership-safety for IP schedules. Furthermore, to prove simulation of a *C-IL* programs we apply compiler consistency on IP schedules.

In the following figures we present examples for reordered execution sequences.

Figure 3.9 shows reordered uninterrupted hypervisor execution on processor i, where interleaving happens only after shared accesses.

Figure 3.10 shows reordered guest execution on processor i, where all guest steps and the VMEXIT step start in an interleaving point.

Figure 3.11 depicts that IPI steps happen from the point of view of processors only at interleaving points. In the reordered execution sequence the IPI steps may take place before local processor steps, which have been before them in the original trace. Still since we have the external interrupt flag in the step labels the JISR will happen at the original place regarding the local execution order of the processors.

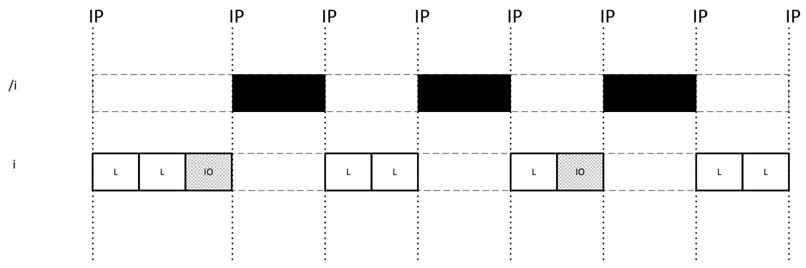

Figure 3.9: Reordered uninterrupted hypervisor execution.

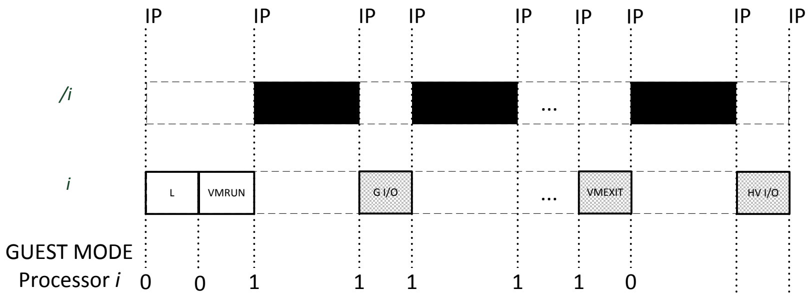

Figure 3.10: Guest execution.

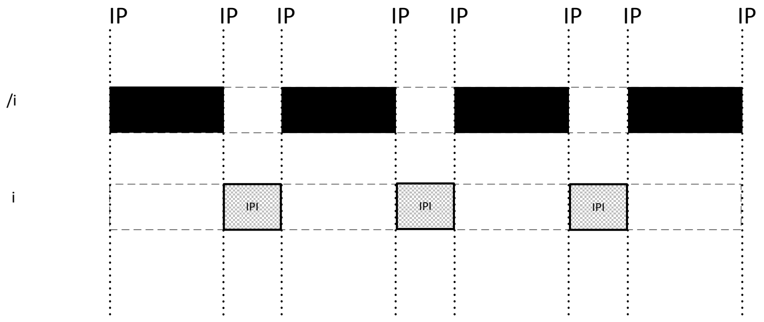

Figure 3.11: Execution of the IPI unit.

Chapter 4

Interrupt Thread

In this chapter we look into the details of reordered processor executions that contain an interrupt service routine. The complete execution of an interrupt service routine starts with JISR and ends with the execution of the $eret$ instruction. In Figure 4.1 we depict an execution sequence which contains interrupt handling obtained after application of the COSMOS reordering theorem. The oval steps belong to the service routine and the square steps belong to the interrupted program. COSMOS provides a competitive technology for generic reordering but it does not consider interrupts. Using COSMOS ownership alone, it is not possible to express and prove that the handler execution does not mess up the program data, since the execution of the interrupt routine has the same access rights as the interrupted program.

As a matter of fact interrupt handlers run in different context than the program. Thus what we considered a processor execution up to here, we split into program thread and handler thread from now on, where the handler thread consists of the interrupt service routine. The first part of the interrupt service routine stores the context of the program thread and dispatches the interrupt to the corresponding handler. The second part is the execution of the particular handler. The third and last part restores the program thread and returns the control to it.

After the service routine the program thread continues from the interrupted place. We consider the IPI handling from the point of view of the interrupted hypervisor. The effect of the TLB flushes is visible to the guest but not to the hypervisor since we do not use address translation in system mode. In order to express and prove such a property we need to guarantee data separation between program thread and handler thread except for the shared data. Therefore we need to refine the ownership model so that we distinguish between handler and program executions.

In the previous chapter we defined the state after $eret$ as an interleaving point. So we expect compiler consistency to hold after the service routine. Now that we consider the interrupt handler a different thread, this can only be guaranteed if compiler consistency holds also for the configuration before JISR. This of course

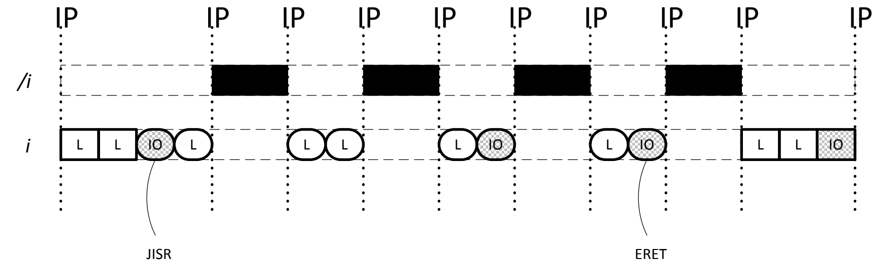

Figure 4.1: Execution sequence of processor i that contains IPI service routine. Oval steps belong to the ISR and the square steps belong to the interrupted program thread.

can not be guaranteed in general. In $MIPS_P$ IPIs can interrupt the program thread at an arbitrary point, e.g. in the middle of a program step, which requires several $MIPS_P$ steps. Thus we need another reorder theorem which allows us to reorder JISR and the service routine to the preceding interleaving point.

The context switch in the service routine is implemented in assembler and contains several instructions. During the execution of these instructions the core state of the interrupted program is partially in registers and partially in memory. This makes it inconvenient to refer to it. Furthermore we want to state that the saved context stays unchanged during the service routine. For that reason we introduce a ghost component. The core state of the program thread is saved atomically and secured during the service routine in this new ghost component.

We extend the $MIPS_P$ configuration and the $MIPS_P$ step function so that the register context of the program thread is stored at JISR in a ghost sub-component of the processor.

Definition 4.1 ▶ $MIPS_P$ Extensions

We extend the processor configuration by a ghost core component. We call this new ghost component old-core and refer to by $oCore$.

$$C_{CPU} \overset{def}{=} [core \in C_{CORE}, tlb \in C_{TLB}, apic \in C_{APIC}, oCore \in C_{CORE}]$$

The changes in the step function are limited to the core step. All original components of the new configuration $h' = \Delta(h, (\textbf{core}, pid, ex))$ are defined as before (Definition 2.44). Additionally in case of JISR we update $oCore$ with the current core state.

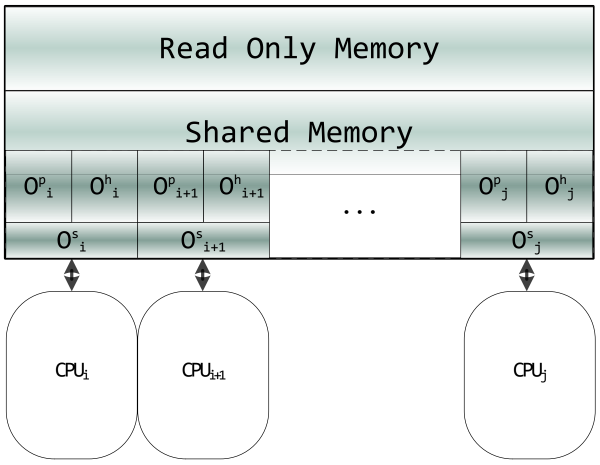

Figure 4.2: OwnershipXT

$$h'.oCore_{pid} = \begin{cases} h.core_{pid} & \mathit{if}\ \mathit{isJISR}(h.core_{pid}, eev, I) \\ h.oCore_{pid} & \mathit{otherwise} \end{cases}$$

In Section 4.1 we define a more fine grained ownership, which distinguishes between program thread and interrupt thread steps. In Section 4.2.1 we list some auxiliary definitions. In Section 4.2 we present a reorder theorem, aiming at a model in which JISR happens only at interleaving points.

4.1 Ownership XT

One property we require of the interrupt handler is that it should not change the memory owned by the program thread. In order to express that, we define a new ownership policy. Additionally we show that the new ownership policy implies the one from Chapter 3.

In Figure 4.2 the enhancement of the ownership model is depicted. The changes concern only the private memory. The set of owned addresses of every processor is separated in three sets representing the stack addresses, the program thread private addresses and the interrupt thread private addresses. The set of stack addresses is fixed and does not change during execution.

We prevent data races on the stack with the access policy based on the stack pointer. During program thread execution we have only valid stack data for the program thread and it resides between the stack base and the stack pointer (Figure 4.3). On JISR we save the stack pointer of the interrupted program.

During the service routine the data belonging to the interrupted program resides between the stack base and the saved stack pointer. The handler stack

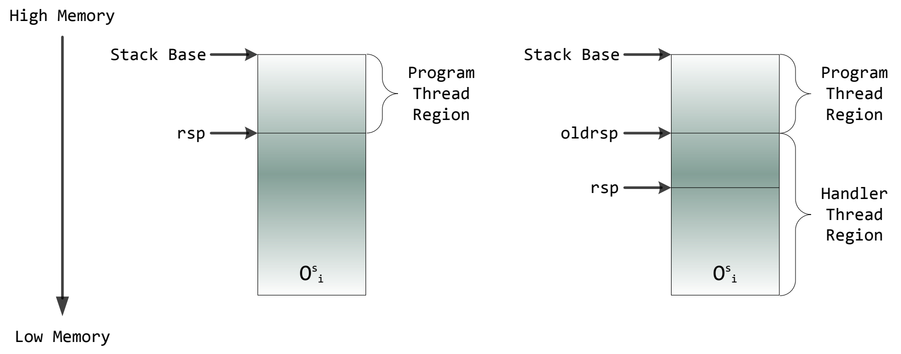

Figure 4.3: Stack memory during program thread execution (left) and handler thread execution (right). The stack is growing downwards.

data resides between the saved stack pointer of the program thread and the current stack pointer. In other words, the saved stack pointer of the program thread serves as a stack base address for the handler thread.

Definition 4.2 ▶ Dynamic Ownership Information XT

The dynamic state of the ownership is defined by a set of shared addresses and three mappings of processor indexes

- to a program thread owns-set O^p,
- to an interrupt thread owns-set O^h and
- to a set of stack addresses O^s.

$$\mathcal{O}_{XT} \overset{def}{=} [O^p \in Pid \to 2^{\mathbb{B}^{32}}, O^h \in Pid \to 2^{\mathbb{B}^{32}}, O^s \in Pid \to 2^{\mathbb{B}^{32}}, \mathbb{A}^{sh} \in 2^{\mathbb{B}^{32}}]$$

We use the following shorthand notation to refer to the different sets of addresses assigned to given processor i in an ownership $o \in \mathcal{O}_{XT}$.

$$o.O^p_i \overset{def}{=} o.O^p[i]$$

$$o.O^h_i \overset{def}{=} o.O^h[i]$$

$$o.O^s_i \overset{def}{=} o.O^s[i]$$

With $o.O_i$ we refer to the union of all addresses assigned to processor i.

$$o.O_i \overset{def}{=} o.O^p_i \cup o.O^h_i \cup o.O^s_i$$

The overall set of private addresses is the union of all owned addresses.

$$o.\mathbb{A}^{pr} = \bigcup_{i \in Pid} o.O_i^p \ \cup \ o.O_i^h \ \cup \ o.O_i^s$$

The set of private addresses not owned by processor i we denote depending on its mode, i.e. program thread or handler thread mode, by:

$$\overline{O}_i^h = \mathbb{A}^{pr} \setminus O_i^h$$

$$\overline{O}_i^p = \mathbb{A}^{pr} \setminus O_i^p$$

From the extended ownership we can build an ownership of the type defined in the previous chapter.

The function $join$ generalizes the information from given extended ownership $ox \in \mathcal{O}_{XT}$ into ownership of type $\mathcal{O}$.

◀ **Definition** 4.3
Join of extended ownership

$$join(ox \in \mathcal{O}_{XT}) \in \mathcal{O} \overset{def}{=} [O, ox.\mathbb{A}^{sh}]$$

where the first element of the result record $O \in Pid \rightarrow 2^{\mathbb{B}^{32}}$ maps processor identifiers to the union of addresses owned by the corresponding processor in the ownership ox.

$$O[i] = ox.O_i^h \cup ox.O_i^p \cup ox.O_i^s$$

We call the ownership $join(ox)$ the *conjoint version of* ox or simply *conjoint* ox.

4.1.1 Ownership Policy XT

The validity conditions of the extended ownership include the ones from the previous chapter and define in addition the properties on the new sets.

The predicate $ownership\text{-}inv_{XT}$ defines the invariant of the extended ownership. It states that:

◀ **Definition** 4.4
Ownership Invariant

- the complete address space $\mathbb{A}$ is covered by the ownership,
- the read only memory is disjoint with the private and the shared memory,
- the owns-sets of different processors are disjoint,
- stack addresses are not shared,
- the set of stack addresses, the program thread owns-set and the interrupt thread owns-set are disjoint.

$$\begin{aligned}
&\mathit{ownership\text{-}inv}_{XT}(o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \mathbb{A} = o.\mathbb{A}^{pr} \cup o.\mathbb{A}^{sh} \cup \mathbb{A}^{ro} \\
&\qquad \wedge\, o.\mathbb{A}^{pr} \cap \mathbb{A}^{ro} = \emptyset \\
&\qquad \wedge\, o.\mathbb{A}^{sh} \cap \mathbb{A}^{ro} = \emptyset \\
&\qquad \wedge\, \forall a, i, j.\ a \in o.O_i \wedge a \in o.O_j \implies i = j \\
&\qquad \wedge\, o.\mathbb{A}^{sh} \cap \bigcup_{i \in Pid} o.O_i^s = \emptyset \\
&\qquad \wedge\, \forall i.\ o.O_i^h \cap o.O_i^p = \emptyset \wedge o.O_i^h \cap o.O_i^s = \emptyset \wedge o.O_i^s \cap o.O_i^p = \emptyset
\end{aligned}$$

Lemma 4.1 (Ownership Invariant Refinement) *If a given extended ownership o is valid then its conjoint version is also valid.*

$$\forall o \in \mathcal{O}_{XT}.\ \mathit{ownership\text{-}inv}_{XT}(o) \implies \mathit{ownership\text{-}inv}(\mathit{join}(o))$$

Proof The proof is trivial, since Definition 4.4 contains literally Definition 3.6.

✔

The rules for ownership transfer in the extended ownership are similar to the one from the previous chapter. The difference is that we need to distinguish which thread is running in order to define which sets are allowed to change, since during program thread execution the handler thread owns-set must stay unchanged and vice versa. The following rules define the admissible transitions during interrupt service routine on a processor i. The handler thread can:

1. non-/exclusively acquire shared unowned addresses to the handler owns-set,
2. exclusively acquire shared addresses which are already in the handler owns-set,
3. release shared/exclusively owned addresses from the handler owns-set.

In case of program thread execution the rules are the same, they just are related to the program thread owns-set.

In Figure 4.4 we depict the transfer rules.

Definition 4.5 ▶ Safe transfer

The predicate $\mathit{transfer}_{XT}$ denotes the transfer rules depending on the processor mode defined by the parameter ih. If ih is one we refer to handler thread execution and if ih is zero we refer to program thread execution.

$$\begin{aligned}
&\mathit{transfer}_{XT}(i \in Pid, ih \in \mathbb{B}, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}, a) \in \mathbb{B} \overset{def}{=} \\
&\qquad \begin{cases} \mathit{transfer}(o.O_i^h, o'.O_i^h, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^h, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i^h, a) & \mathit{if}\ ih \\ \mathit{transfer}(o.O_i^p, o'.O_i^p, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^p, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i^p, a) & \mathit{otherwise} \end{cases}
\end{aligned}$$

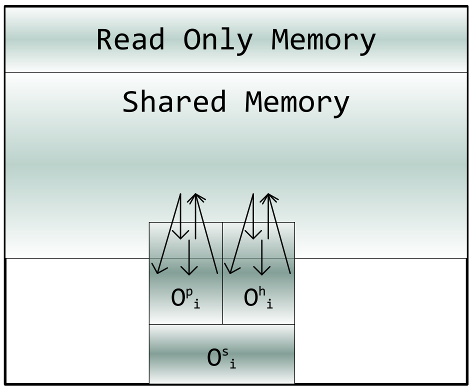

Figure 4.4: Ownership transfer XT.

The predicate $\mathit{safetransfer}_{XT}$ denotes whether the extended ownership pair o and o' satisfies the validity for ownership transfer on steps of processor i, where io is a flag indicating I/O steps and ih indicates handler thread execution.

$$\mathit{safetransfer}_{XT}(i \in \mathit{Pid}, io \in \mathbb{B}, ih \in \mathbb{B}, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}$$
$$\begin{cases} \forall a \in \mathbb{B}^{32}.\ \mathit{transfer}_{XT}(i, ih, o, o', a) & \mathit{if}\ io \\ \quad \wedge \forall j \in \mathit{Pid}.\ j \neq i \implies a \in o.O_j \iff a \in o'.O_j & \\ & \\ o = o' & \mathit{otherwise} \end{cases}$$

Lemma 4.2 (Safe Transfer Refinement) *If the ownership transfer between the extended ownership pair o and o' is safe, then the ownership transfer between their conjoint versions $join(o)$ and $join(o')$ is also safe.*

$$\begin{aligned} &\forall i \in \mathit{Pid}, io, ih \in \mathbb{B}, o, o' \in \mathcal{O}_{XT}. \\ &\qquad \mathit{safetransfer}_{XT}(i, io, ih, o, o') \implies \mathit{safetransfer}(i, io, join(o), join(o')) \end{aligned}$$

Proof The only thing that we have to prove is that transfer that satisfies $\mathit{transfer}_{XT}$ also satisfies the rules defined for the conjoint ownership. We make a case distinction on ih. After unfolding the definitions of $\mathit{safetransfer}_{XT}$, $\mathit{transfer}_{XT}$ and $\mathit{safetransfer}$ we come to similar statements in both cases.

- $ih = 1$

$$\begin{aligned} &\mathit{transfer}(o.O^h_i, o'.O^h_i, o.\mathbb{A}^{sh} \setminus o.\overline{O}^h_i, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}^h_i, a) \implies \\ &\qquad\qquad \mathit{transfer}(o.O_i, o'.O_i, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i, a) \end{aligned}$$

- $ih = 0$

$$\mathit{transfer}(o.O_i^p, o'.O_i^p, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^p, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i^p, a) \implies \mathit{transfer}(o.O_i, o'.O_i, o.\mathbb{A}^{sh} \setminus o.\overline{O}_i, o'.\mathbb{A}^{sh} \setminus o'.\overline{O}_i, a)$$

Both cases are quite similar and can be generalized to the following statement.

$$\begin{aligned} &\mathit{transfer}(O_s, O'_s, sh_s, sh'_s, a) \wedge A(O_s, O'_s, sh_s, sh'_s, O, O', sh, sh') \\ \implies &\mathit{transfer}(O, O', sh, sh', a) \end{aligned}$$

Where O and O_s denote sets of owned addresses, sh and sh_s denote sets of shared addresses, O', O'_s, sh' and sh'_s denote the corresponding sets after the ownership transfer of an address a, and A is a predicate stating that the sets from the left hand side of the implication are subsets of the sets from the right hand side. In that generalization O_s represents $o.O_i^p$ and $o.O_i^h$, which are subsets of $o.O_i$ represented by O. Furthermore sh_s denotes $o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^h$ and $o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^p$, which are subsets of $o.\mathbb{A}^{sh} \setminus o.\overline{O}_i$ represented by sh.

$$\begin{aligned} A(O_s, O'_s, sh_s, sh'_s, O, O', sh, sh') \overset{def}{=} &\ O_s \subseteq O \ \wedge\ O'_s \subseteq O' \ \wedge\ sh_s \subseteq sh \\ &\wedge\ sh'_s \subseteq sh' \ \wedge O \setminus O_s = O' \setminus O'_s \ \wedge\ sh \setminus sh_s = sh' \setminus sh'_s \end{aligned}$$

We introduce the following shorthands for better readability.

- $X = O \setminus O_s = O' \setminus O'_s$
- $Y = sh \setminus sh_s = sh' \setminus sh'_s$

After unfolding the definition of $\mathit{transfer}$ we get:

$$\begin{aligned} &(a \in O_s \cup sh_s \iff a \in O'_s \cup sh'_s) \\ &\wedge (a \notin sh_s \wedge a \in sh'_s \implies a \in O_s \wedge a \notin O'_s) \\ &\wedge A(O_s, O'_s, sh_s, sh'_s, O, O', sh, sh') \\ &\implies (a \in O \cup sh \iff a \in O' \cup sh') \\ &\qquad \wedge (a \notin sh \wedge a \in sh' \implies a \in O \wedge a \notin O') \end{aligned}$$

We split the conjunctions on both sides and prove them separately, which gives us even stronger statement, since

$$(a \implies c) \wedge (b \implies d) \implies (a \wedge b \implies c \wedge d)$$

Case 1: $(a \in O_s \cup sh_s \iff a \in O'_s \cup sh'_s) \wedge A \implies (a \in O \cup sh \iff a \in O' \cup sh')$

$$\begin{aligned} a \in O \cup sh &\iff a \in X \cup Y \cup O_s \cup sh_s \\ &\iff a \in X \cup Y \cup O'_s \cup sh'_s && \mathit{precondition} \\ &\iff a \in O' \cup sh' \end{aligned}$$

Case 2:

$$\begin{aligned}&(a \notin sh_s \wedge a \in sh'_s \implies a \in O_s \wedge a \notin O'_s) \wedge A\\ &\implies (a \notin sh \wedge a \in sh' \implies a \in O \wedge a \notin O')\end{aligned}$$

$$\begin{aligned}
a \notin sh \wedge a \in sh' &\implies a \notin sh \wedge a \notin Y \wedge a \in sh' && Y \subseteq sh\\
&\implies a \notin sh \wedge a \in sh'_s && Y = sh' \setminus sh'_s\\
&\implies a \notin sh_s \wedge a \in sh'_s && sh_s \subseteq sh\\
&\implies a \in O_s \wedge a \notin O'_s && \mathit{precondition}\\
&\implies a \in O_s \wedge a \notin O'_s \wedge a \notin X && X = O \setminus O_s\\
&\implies a \in O_s \wedge a \notin O' && X = O' \setminus O'_s\\
&\implies a \in O \wedge a \notin O' && O_s \subseteq O
\end{aligned}$$

✔

The memory access policy of the extended ownership is close to the one defined in the previous chapter, where a processor i depending on I/O flag can access read only memory, shared memory and its owns-set. For the extended ownership we reduce the set of accessible addresses depending on the processor mode. Instead of the complete owns-set the handler and the program thread running on processor i are allowed to access only the handler owns-set and the program thread owns set respectively. In addition both are allowed to access a portion of the stack.

◄ **Definition** 4.6 Memory Access Policy

The predicate $safeacc_{XT}$ defines our policy for reading and writing memory addresses by a processor i based on the extended ownership setting o. R and W represent the sets of addresses to be read and/or written respectivelly. io and ih are flags denoting the execution of an I/O step and a handler step respectively. rsp is the value of the processor's stack pointer.

$$safeacc_{XT}(i \in Pid, io \in \mathbb{B}, ih \in \mathbb{B}, rsp \in \mathbb{B}^{32}, R \in 2^{\mathbb{B}^{32}}, W \in 2^{\mathbb{B}^{32}}, o \in \mathcal{O}_{XT}) \overset{def}{=}$$

$$\begin{cases} (R \subseteq o.O_i^h \cup o.\mathbb{A}^{sh} \cup o.\mathbb{A}^{ro} \cup stack(o.O_i^s, ih, rsp)) & if\ ih \wedge io \\ \wedge (W \subseteq o.O_i^h \cup (o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^h) \cup stack(o.O_i^s, ih, rsp)) & \\ & \\ (R \subseteq o.O_i^h \cup o.\mathbb{A}^{ro} \cup stack(o.O_i^s, ih, rsp)) & if\ ih \wedge \neg io \\ \wedge (W \subseteq o.O_i^h \setminus o.\mathbb{A}^{sh} \cup stack(o.O_i^s, ih, rsp)) & \\ & \\ (R \subseteq o.O_i^p \cup o.\mathbb{A}^{sh} \cup o.\mathbb{A}^{ro} \cup stack(o.O_i^s, ih, rsp)) & if\ \neg ih \wedge io \\ \wedge (W \subseteq o.O_i^p \cup (o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^p) \cup stack(o.O_i^s, ih, rsp)) & \\ & \\ (R \subseteq o.O_i^p \cup o.\mathbb{A}^{ro} \cup stack(o.O_i^s, ih, rsp)) & if\ \neg ih \wedge \neg io \\ \wedge (W \subseteq o.O_i^p \setminus o.\mathbb{A}^{sh} \cup stack(o.O_i^s, ih, rsp)) & \end{cases}$$

where the function $stack$ defines the stack region of the program thread to contain all addresses above the stack pointer parameter and the stack region for the handler to contain all addresses under the stack pointer parameter[1].

$$stack(S \in 2^{\mathbb{B}^{32}}, ih \in \mathbb{B}, rsp \in \mathbb{B}^{32}) \in 2^{B^{32}} \overset{def}{=} \begin{cases} \{a \mid a \in S \wedge a \leq rsp\} & if\ ih \\ \{a \mid a \in S \wedge a > rsp\} & if\ \neg ih \end{cases}$$

Lemma 4.3 (Safe access refinement) *If a memory access is ownership safe according to an extended ownership* o*, then it is also ownership safe according to the conjoint ownership* $join(o)$.

$$\begin{aligned} &\forall i \in Pid, io \in \mathbb{B}, ih \in \mathbb{B}, rsp \in \mathbb{B}^{32}, R, W \in 2^{B^{32}}, o \in \mathcal{O}_{XT}. \\ &\qquad safeacc_{XT}(i, io, ih, rsp, R, W, o) \implies safeacc(i, io, R, W, join(o)) \end{aligned}$$

Proof The proof considers four cases $\neg ih \wedge \neg io$, $ih \wedge \neg io$, $\neg ih \wedge io$ and $ih \wedge io$. In all of them we show that the largest possible reads and writes sets that satisfy $safeacc_{XT}$ are subsets of the ones defined for the conjoint ownership by $safeacc$.

✔

[1] In later definitions we have to instantiate the rsp parameter of the $stack$ function properly either with the stack register of the core or with the stack register of the old-core.

4.1.2 Safe Execution

A $MIPS_P$ step defined by input α and starting in configuration h is ownership-safe according to extending ownership setting pair o and o' if it obeys the memory access policy and maintains the ownership invariant.

◀ **Definition** 4.7
Safe $MIPS_P$ Step

$$\begin{aligned}&\mathit{safestep}_{XT}(h \in C_M, \alpha \in \Sigma_M, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \overset{def}{=} \\ &\qquad \mathit{safeacc}_{XT}(\alpha.pid, \mathit{IOstep}(h,\alpha), isr, rsp, reads(h,\alpha), writes(h,\alpha), o) \\ &\quad \wedge \mathit{safetransfer}_{XT}(\alpha.pid, \mathit{IOstep}(h,\alpha), isr, o, o')\end{aligned}$$

where functions $reads(h, \alpha)$ and $writes(h, \alpha)$, which denote the set of addresses which are read and respectively written by α, are defined as in Section 3.2. The predicate $\mathit{IOstep}(h, \alpha)$ denotes whether the next step is an I/O step. isr denotes whether we are in an IPI service routine.

$$isr = h.isr_{\alpha.pid}[0] \vee jisr_{IPI}(\alpha)$$

We note, that JISR steps count as the first ISR step according to the ownership safety and transfer.
By rsp we denote the stack pointer used to split the stack.

$$rsp = \begin{cases} h.core_{\alpha.pid}.rsp & \neg isr \\ h.oCore_{\alpha.pid}.rsp & isr \end{cases}$$

A $MIPS_P$ execution sequence defined by initial configuration h and input sequence β is ownership-safe according to initial extended ownership o and final extended ownership o' if there exists an extended ownership sequence such that every step of the execution is ownership-safe.

◀ **Definition** 4.8
Safe Execution Sequence

$$\begin{aligned}&\mathit{safeseq}_{XT}(h \in C_M, \beta \in (\Sigma_M)^*, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\ &\qquad \beta = \varepsilon \wedge o = o' \wedge \mathit{ownership\text{-}inv}_{XT}(o) \\ &\quad \vee \exists o^0, ..., o^n \in \mathcal{O}_{XT}.\ o^0 = o \wedge o^n = o' \\ &\qquad\quad \wedge \forall i < n.\ \mathit{ownership\text{-}inv}_{XT}(o^i) \wedge (\mathit{safestep}_{XT}(h^i, \beta_i, o^i, o^{i+1}) \vee ipi(\beta_i))\end{aligned}$$

Lemma 4.4 (Safe Step Sequence Refinement) *If an execution sequence is ownership safe according to the extended ownership, then it is ownership safe according to the conjoint ownership too.*

$$\begin{aligned}&\forall h \in C_M, \beta \in (\Sigma_M)^*, ox, ox' \in \mathcal{O}_{XT}. \\ &\quad \mathit{safeseq}_{XT}(h, \alpha, ox, ox') \implies \mathit{safeseq}_{MIPS_P}(h, \alpha, join(ox), join(ox'))\end{aligned}$$

Proof We unfold the definitions and apply the Lemmas 4.1, 4.2 and 4.3.

✔

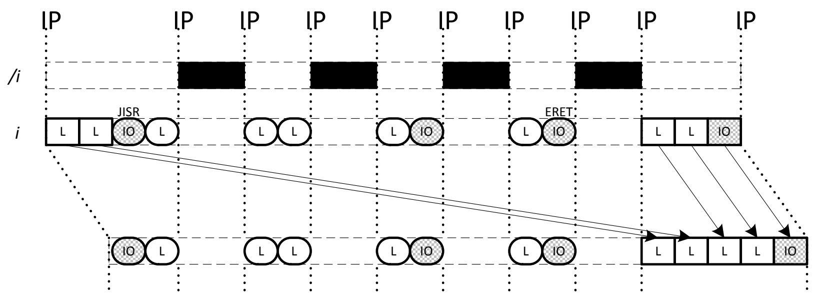

Figure 4.5: Reordering of execution sequence of processor i that contains IPI service routine.

4.2 Reordering Proof

In this section we state and prove a reordering theorem in which local steps of the program placed between JISR and the preceding interleaving point are reordered after the service routine. As Figure 4.5 shows this implies that in the reordered execution JISR happens at an interleaving point.

We want to prove that every $MIPS_P$ execution containing IPI handling can be simulated by an execution in which the JISR step is reordered to the preceding interleaving point. Furthermore we want to prove that if the reordered execution is ownership-safe then also the original execution is ownership-safe. We do not need to examine the complete executions but only the parts including ISR of IPIs. These sub-sequences start at the last interleaving point before JISR and end at the second interleaving point after eret, i.e. the first interleaving point of the program thread after ISR.

We continue in Subsection 4.2.1 with the definitions of some predicates and Lemmas, which we need for stating and proving our reordering theorem. In Subsection 4.2.2 we define a simulation relation between the configurations of the original execution and the one with reordered ISR. In Subsection 4.2.3 we prove that the simulation relations holds for every step of the original execution. In the proof we rely on properties of the ISR. These are stated in the following chapter as Software Conditions 5 and 6

4.2.1 Aux definitions

As we previously mentioned interrupt handling contains a context switch from program thread to interrupt handler thread. The context of the interrupted process is saved and restored to/from a process control block (PCB). In general a PCB is a kernel data structure in which the processor registers are secured. It is used to store the context of users and the kernel on process switched. To avoid ambiguity we call our data structure an $interrupt\ PCB$ or $IPCB$. While PCBs are allocated as C variables IPCBs are not present on the *C-IL* level. They represent a set of memory addresses, which are always assigned to the interrupt service routine.

Software Condition 4 (IPCB Ownership) *IPCB addresses stay always in the handler thread memory and are excluded from the ownership transfer.*

$$\mathbb{A}_{ipcb_i} \subset o.O_i^h,$$

where $\mathbb{A}_{ipcb_i}$ is defined in Definition 4.9.

Later defining the invariant between the ownership model on the different levels in Definition 5.79 and in Software Condition 10 we rely on the IPCB characteristics from above.

Definition 4.9 IPCB

An IPCB offers space for all general and special purpose registers, and the program counter. Thus the IPCB size is then $65 \cdot 4$ bytes. During hypervisor boot a memory region $\mathbb{A}_{ipcb}$ starting at $IPCBS_{BA}$ is allocated for IPCBs, one IPCB per processor.

$$\mathbb{A}_{ipcb} \overset{def}{=} [IPCBS_{BA} : IPCBS_{BA} +_{32} bin_{32}(np \cdot 65 \cdot 4)]$$

where $np \in \mathbb{N}$ denotes the number of processor in the system.

The base address of the IPCB of processor i we denote by

$$ipcb_{ba}(i) \overset{def}{=} IPCBS_{BA} +_{32} bin_{32}(i \cdot 65 \cdot 4)$$

The set of addresses occupied by the IPCB of processor i we denote by

$$\mathbb{A}_{ipcb_i} \overset{def}{=} [ipcb_{ba}(i) : ipcb_{ba}(i) +_{32} bin_{32}(65 \cdot 4)]$$

We refer to the IPCB of processor i by

$$ipcb(h, i) \overset{def}{=} h.m_{65 \cdot 4}(ipcb_{ba}(i))$$

Definition 4.10 IPCB Registers

We define the function reg_i to compute for a given IPCB address a the index of the corresponding GPR registers.

$$reg_i(a \in \mathbb{B}^{32}) \in \mathbb{B}^5 \overset{def}{=} (a[31:2]00 -_{32} ipcb_{ba}(i))[6:2]$$

The function ia_i computes for a given register index r the corresponding aligned address in the IPCB.

$$ia_i(r \in \mathbb{B}^5) \in \mathbb{B}^{32} \overset{def}{=} (0^{25}r00 +_{32} ipcb_{ba}(i))$$

Definition 4.11 ▶ Local Step

The predicate $Lstep$ denotes whether a given step α executed in a configuration h is a local core step of processor i.

$$Lstep_i(h \in C_M, \alpha \in \Sigma_M) \in \mathbb{B} \overset{def}{=} \alpha = (\textbf{core}, i, 0) \wedge \neg IOstep(h, \alpha)$$

Definition 4.12 ▶ Local Steps Sequence

The predicate $Lsteps$ denotes whether a given step sequence β executed in a configuration h^0 contains only local core steps of processor i.

$$Lsteps_i(h^0 \in C_M, \beta \in {\Sigma_M}^*) \in \mathbb{B} \overset{def}{=} \forall k < |\beta|.\ Lstep_i(h^k, \beta_k)$$

Definition 4.13 ▶ Local Equality (Program Thread)

The predicate eq^L denotes whether in two configuration h and d the local components of the program thread on processor i according the given ownership are equal during program thread execution (i.e. clear ISR bit). The local components, i.e. components accessed by local steps, are the core, the owned addresses and the allocated portion of the stack.

$$\begin{aligned} eq_i^L(h \in C_M, d \in C_M, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}\ & h.core_i.pc = d.core_i.pc \\ & \wedge h.core_i.gpr = d.core_i.gpr \\ & \wedge \forall r \in \{sr, mode\}.\ d.core_i.spr(r) = h.core_i.spr(r) \\ & \wedge \forall a \in o.O_i^p \cup stack(o.O_i^s, 0, h.core_i.rsp).\ h.M(a) = d.M(a) \end{aligned}$$

We note, that the only SPRs relevant for steps of the program thread, are the mode and the status registers.

Before defining the local equality for the handler we want to consider an important property of the ISR, i.e. that its execution does not depend on GPR content belonging to the program thread. In other words the GPR content of the configuration, in which JISR happens, does not influence the ISR execution. In the next software condition we express that property.

Software Condition 5 (ISR GPR) *ISR reads to GPR registers are either preceded by an ISR write to the same register or belong to the code saving the context of the program thread. ISR writes to GPR registers can not precede the saving of the destination register in the IPCB*[2].

We rely on this elementary ISR property in our simulation proof. It implies, that the local components of the handler are a subset of what we consider to be the program thread local components. For instance some GPR registers are excluded

[2]For that purpose the corresponding IPCB address should fit in 16 bits. Otherwise the ISR save context should initially load $ipcb_{ba}(i)$ into a dedicated register, whose content must be saved temporary on the stack and then moved into the IPCB, and this software condition must be refined.

corresponding to Software Condition 5. For integrating that property formally in our definitions, we need parameters for both the list of registers saved by the ISR in the IPCB(denoted by $sv \in 2^{\mathbb{B}^5}$), and registers written by the ISR(denoted by $ch \in 2^{\mathbb{B}^5}$). Using these lists we are able to specify the components, which are relevant to the handler steps execution.

◀ **Definition** 4.14
Local Equality ISR

Two configurations h and d are locally equal according the handler thread on processor i and the given ownership if the following properties hold.

- The program counters are equal in both machines.

$$h.core_i.pc = d.core_i.pc$$

- The registers are changed in the same way in both machines.

$$\begin{aligned} &gpreq(h \in C_M, d \in C_M, i \in Pid, ch \in 2^{\mathbb{B}^5}, sv \in 2^{\mathbb{B}^5}) \in \mathbb{B} \overset{def}{=} \\ &\qquad \forall r \in [0:31].\ d.core_i.gpr(r) \neq d.oCore_i.gpr(r) \iff r \in ch \\ &\qquad\quad \wedge r \in ch \implies d.core_i.gpr(r) = h.core_i.gpr(r) \\ &\qquad\quad \wedge r \notin ch \implies h.core_i.gpr(r) = h.oCore_i.gpr(r) \\ &\qquad\quad \wedge ch \subseteq sv \end{aligned}$$

- The SPR stores the same values on both machines, except for epc, esr and $emode$. Throughout the ISR epc, esr and $emode$ store the address of the interrupted instruction, the sr and $mode$ values from the state before JISR respectively.

$$\begin{aligned} &spreq(h \in C_M, d \in C_M, i \in Pid) \in \mathbb{B} \overset{def}{=} \\ &\qquad (\forall r \in \{sr, eca, edata, mode\}.\ d.core_i.spr(r) = h.core_i.spr(r)) \\ &\qquad \wedge d.core_i.spr(esr) = d.oCore_i.spr(sr) \\ &\qquad \wedge d.core_i.spr(emode) = d.oCore_i.spr(mode) \\ &\qquad \wedge d.core_i.spr(epc) = d.oCore_i.pc \\ &\qquad \wedge h.core_i.spr(esr) = h.oCore_i.spr(sr) \\ &\qquad \wedge h.core_i.spr(emode) = h.oCore_i.spr(mode) \\ &\qquad \wedge h.core_i.spr(epc) = h.oCore_i.pc \end{aligned}$$

- The handler thread memory except the IPCB region stores the same values.

$$\forall a \in o.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h.M(a) = d.M(a)$$

We exclude the IPCB addresses from the memory equality, having in mind our intended simulation relation, and handle it separately. In the original and in the reordered executions the ISR begins in different states. This implies that in the reordered execution the values stored in the IPCBs will be different than the ones stored in the original execution.

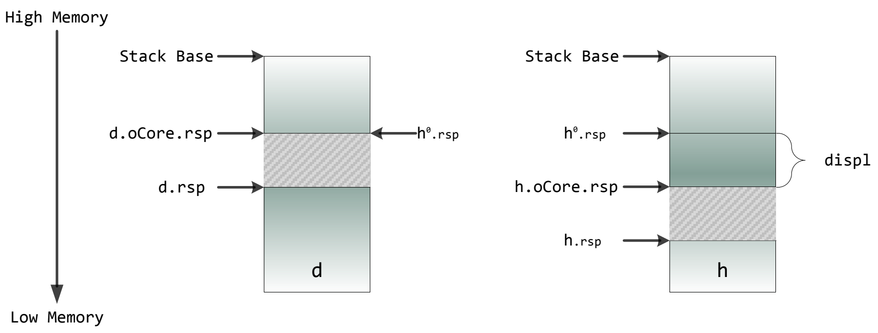

Figure 4.6: Stack region allocation during ISR. On the left side is the stack in the reordered execution. On the right side is the original stack wit some extra space allocated by the postponed steps of the program thread.

- IPCB registers store the GPR content of the configuration before JISR, i.e. IPCB registers are equal to the GPRs in the old-core.

$$\begin{aligned} &ipcbeq(h \in C_M, d \in C_M, i \in Pid, sv \in 2^{\mathbb{B}^5}) \in \mathbb{B} \overset{def}{=} \\ &\quad \forall a \in \mathbb{A}_{ipcb_i}.\ reg_i(a) \in sv \implies \\ &\qquad h.M_4(a[31:2]00) = h.oCore_i.gpr(reg_i(a)) \\ &\qquad \wedge\ d.M_4(a[31:2]00) = d.oCore_i.gpr(reg_i(a)) \end{aligned}$$

- The stack memory of the handler thread stores the same values on both machines. The handler stack might be allocated on different places in both machines. We define this displacement, by the difference between the stack pointers of the interrupted program (Figure 4.6).

$$displ = d.oCore_i.rsp - h.oCore_i.rsp$$

The predicate eq^{LH} denotes handler thread local equality.

$$\begin{aligned} &eq_i^{LH}(h \in C_M, d \in C_M, o \in \mathcal{O}_{XT}, ch \in 2^{\mathbb{B}^5}, sv \in 2^{\mathbb{B}^5}) \in \mathbb{B} \overset{def}{=} \\ &\quad h.core_i.pc = d.core_i.pc \\ &\quad \wedge\ gpreq(h, d, i, ch, sv) \\ &\quad \wedge\ spreq(h, d, i) \\ &\quad \wedge\ \forall a \in o.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h.M(a) = d.M(a) \\ &\quad \wedge\ ipcbeq(h, d, i, sv) \\ &\quad \wedge\ \forall a \in stack(o.O_i^s, 1, h.oCore_i.gpr(rsp)) \wedge a > h.core_i.gpr(rsp). \\ &\qquad h.M(a) = d.M(a + displ) \end{aligned}$$

The predicate eq^E denotes whether in two configuration the components of other processor according the given ownership are equal. The components of other processors, i.e. components accessed by local steps of other units are the processors and the owned addresses.

◀ **Definition** 4.15
Environment
Equality

$$\begin{aligned} eq_i^E(h,d,o) &\overset{def}{=} \forall j \neq i.\ h.cpu_j = d.cpu_j \\ &\wedge \forall a \in o.O_j.\ h.M(a) = d.M(a) \end{aligned}$$

The predicate eq^{Sh} denotes whether in two configuration the shared components according the given ownership are equal. The shared components, i.e. components accessed by I/O steps of processor i, the APIC, the TLB and the shared addresses.

◀ **Definition** 4.16
Shared Equality

$$\begin{aligned} eq_i^{Sh}(h,d,o) &\overset{def}{=} h.apic_i = d.apic_i \\ &\wedge h.tlb_i = d.tlb_i \\ &\wedge \forall a \in o.\mathbb{A}^{sh}.\ h.M(a) = d.M(a) \end{aligned}$$

The following three lemmas express basic properties of local and I/O steps. Similar lemmas are proven in [Bau14]. Since they do not talk about ISR steps, their proofs which are similar to the proofs in [Bau14] and we skip them here. Note that the lemmas require the ownership safety of the considered step. We extend Lemma 4.5 by a condition, that the executed step is a program thread step. Later we define a similar lemma for local steps of the ISR. Lemma 4.6 and Lemma 4.7 do not need a case distinction between a program thread and a handler execution.

Lemma 4.5 (Local Program Thread Steps: Locality) *If we execute the same program thread local step on two configurations, then they are locally equal if and only if they were locally equal before the step.*

$$\begin{aligned} &Lstep_i(h,\alpha) \wedge safestep_{XT}(h,\alpha,o,o) \wedge \neg isr(h.apic_{\alpha.pid}) \implies \\ &\quad (eq_i^L(h,d,o) \implies eq_i^L(\Delta(h,\alpha),\Delta(d,\alpha),o)) \end{aligned}$$

Lemma 4.6 (Local Steps: Environment) *If h' is the configuration obtained after executing a local step of given processor i in configuration h then h' is equal to given configuration d according to shared components and components local to other processors if and only if the same holds for h.*

$$\begin{aligned} &Lstep_i(h,\alpha) \wedge safestep_{XT}(h,\alpha,o,o) \implies \\ &\quad (eq_i^E(h,d,o) \iff eq_i^E(\Delta(h,\alpha),d,o)) \\ &\quad \wedge (eq_i^{Sh}(h,d,o) \iff eq_i^{Sh}(\Delta(h,\alpha),d,o)) \end{aligned}$$

Lemma 4.7 (I/O Steps) *If we execute the same I/O step α on two configurations h and d, then the resulting configurations are locally and shared equivalent if and only if h and d are locally and shared equal. Furthermore the execution of α on one of the machines preserves the equality of components local to other processors.*

$$\begin{aligned}
&\alpha = (\textbf{core}, i, 0) \wedge IOstep(h,\alpha) \wedge safestep_{XT}(h,\alpha,o,o') \implies \\
&\quad (eq_i^L(h,d,o) \wedge eq_i^{Sh}(h,d,o) \\
&\qquad\quad \implies eq_i^L(\Delta(h,\alpha),\Delta(d,\alpha),o') \wedge eq_i^{Sh}(\Delta(h,\alpha),\Delta(d,\alpha),o')) \\
&\quad \wedge (eq_i^E(h,d,o) \iff eq_i^E(\Delta(h,\alpha),\Delta(d,\alpha),o'))
\end{aligned}$$

Next we define the desired execution schedule.

Definition 4.17 ▶ $sched_{JIP}$ Schedule

The predicate $sched_{JIP}$ denotes that every IPI service routine of a given processor i starts at an interleaving point.

$$\begin{aligned}
&sched_{JIP}(h \in MIPS_P, \beta \in (\Sigma_M)^*, i \in Pid) \in \mathbb{B} \overset{def}{=} \\
&\qquad\qquad \forall k \in [0 : |\beta| - 1].\ jisr_{IPI}(\beta_k, i) \implies IP(h,\beta,k)
\end{aligned}$$

Definition 4.18 ▶ Extended IP Schedule Single Processor

The predicate ${sched_{XT}}^i$ denotes that in a given execution sequences, steps of different units are interleaved only at interleaving points and every IPI service routine of a given processor i starts at an interleaving point.

$${sched_{XT}}^i(h \in C_M, \beta \in (\Sigma_M)^*, i \in Pid) \in \mathbb{B} \overset{def}{=} IPsched(h,\beta) \wedge sched_{JIP}(h,\beta,i)$$

Definition 4.19 ▶ Extended IP Schedule

The predicate $sched_{XT}$ denotes that in a given execution sequences, steps of different units are interleaved only at interleaving points and every IPI service routine starts at an interleaving point.

$$sched_{XT}(h \in C_M, \beta \in (\Sigma_M)^*) \in \mathbb{B} \overset{def}{=} IPsched(h,\beta) \wedge \forall i \in Pid.\ sched_{JIP}(h,\beta,i)$$

Definition 4.20 ▶ Program Thread Interleaving Point

The predicate IP^p denotes whether a given interleaving point belongs to a program thread or to an IPI ISR. Interleaving points which belong to an IPI ISR are defined by the APIC state, i.e. by the $apic.isr$ bits, or by the preceding $eret$ step (Figure 4.7).

$$\begin{aligned}
&IP^p(h^0 \in C_M, \beta \in (\Sigma_M)^*, i \in Pid, n \in \mathbb{N}) \in \mathbb{B} \overset{def}{=} \\
&\quad IP(h^0,\beta,n) \wedge \neg isr(h^n.apic_i) \wedge h^n.mode_i[0] = 0 \wedge \neg eret_{IPI}(h^{n-1},\beta_{n-1},i)
\end{aligned}$$

Definition 4.21 ▶ ISR Execution Sequence

The predicate $ISRseq$ denotes whether a given execution (sub-)sequence β of length n is an IPI ISR of processor i.

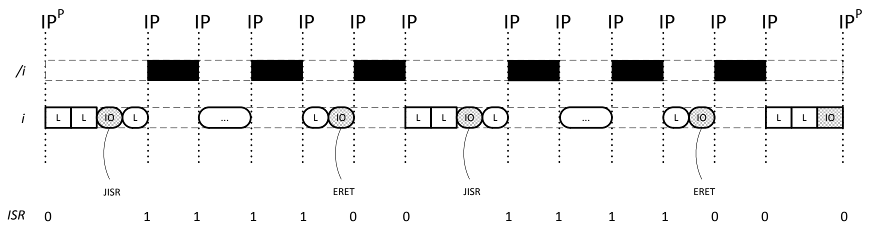

Figure 4.7: Execution sequence with two IPI ISRs of processor i. All interleaving points in which the corresponding ISR bit in the APIC is set and all interleaving points in which an *eret* step ends belong to the ISRs.

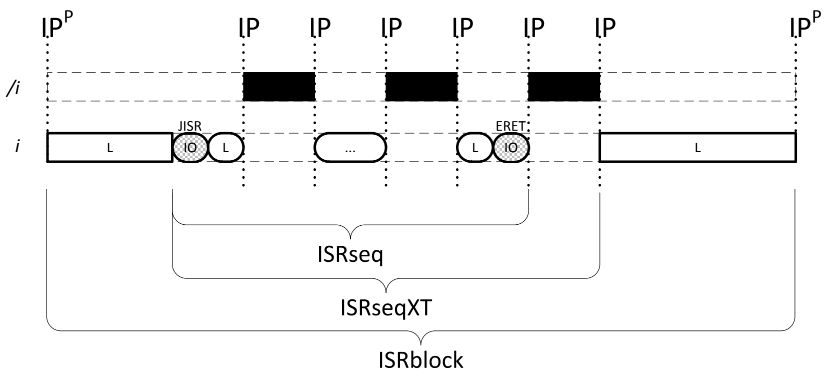

Figure 4.8: ISR block of processor i defined by two subsequent interleaving points of the program thread and an IPI ISR. The IPI ISR steps and the subsequent steps of other processors or IPI will not be reordered.

$$
\begin{aligned}
&\mathit{ISRseq}(h^0 \in C_M, \beta \in (\Sigma_M)^*, i \in \mathit{Pid}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \mathit{jisr}_{IPI}(\beta_0, i) \wedge \mathit{eret}_{IPI}(h^{n-1}, \beta_{n-1}, i) \\
&\qquad \wedge (\neg \exists k \in [1 : n-2].\ (\mathit{jisr}_{IPI}(\beta_k, i) \vee \mathit{eret}_{IPI}(h^k, \beta_k, i)))
\end{aligned}
$$

Since *eret* is an IO step and ends in an interleaving point the subsequent local step of processor i may be preceded by interleaved steps of other processors or IPI steps. These interleaved steps can not be reordered, therefore we define another predicate to cover them together with the ISR sequence preceding them.

Definition 4.22 ▶ ISR Execution Sequence XT

The predicate ISRseq_{XT} denotes whether a given execution (sub-)sequence β of length n is an IPI ISR of processor i followed by steps of other processors or IPI steps.

$$\begin{aligned}
&\mathit{ISRseq}_{XT}(h^0 \in C_M, \beta \in (\Sigma_M)^*, i \in \mathit{Pid}) \in \mathbb{B} \overset{def}{=}\\
&\qquad \exists m \in [0:n-1].\ \mathit{ISRseq}(h^0, \beta[0:m], i)\\
&\qquad \wedge (\forall k \in [m+1:n-1].\ \neg(\beta_k.pid = i \wedge core(\beta_k)))
\end{aligned}$$

Definition 4.23 ▶ ISR Execution Block

The predicate $\mathit{ISRblock}$ denotes whether a given execution (sub-)sequence is an ISR block of processor i. By an ISR block we refer to a sequence of steps of a given processor between two subsequent program thread interleaving points which contains at least one ISR of processor i. An ISR block contains several interleaving points but only two (the first and the final one) are outside the ISR (Figure 4.8).

These conditions on the interleaving points in an ISR imply that the $\mathit{ISRblock}$ predicate is fulfilled by sequences, which contain:

- only complete ISRs of processor i, i.e. ISRs that start and end within the considered sub-sequence,
- only local steps of the same processor before the first JISR step,
- only local steps of the same processor between two IPI ISRs (if the block contains several IPI ISRs),
- a sequence of local steps of i after the last $eret$,
- and not more than one I/O step of i at the end of the sequence.

$$\begin{aligned}
&\mathit{ISRblock}(h^0 \in C_M, \beta \in (\Sigma_M)^*, i \in \mathit{Pid}) \in \mathbb{B} \overset{def}{=}\\
&\qquad \mathit{IPsched}(h^0, \beta)\\
&\quad \wedge \mathit{IP}^p(h^0, \beta, i, 0) \wedge \mathit{IP}^p(h^0, \beta, i, n)\\
&\quad \wedge (\neg \exists l \in [1:n-1].\ \mathit{IP}^p(h^0, \beta, i, l))\\
&\quad \wedge \exists j, k \in [0:n-1].\ \mathit{ISRseq}(h^j, \beta[j:k], i)\\
&\quad \wedge \forall j \in [0:n-1].\ \mathit{jisr}_{IPI}(\beta_j, i) \implies \exists k \in [0:n-1].\ \mathit{ISRseq}(h^j, \beta[j:k], i,)\\
&\quad \wedge \forall j \in [0:n-1].\ \mathit{eret}_{IPI}(h^j, \beta_j, i) \implies \exists k \in [0:n-1].\ \mathit{ISRseq}(h^k, \beta[k:j], i)
\end{aligned}$$

where $n = |\beta|$.

Lemma 4.8 (ISR Block Steps) *Every ISR block of given processor i contains only core steps of processor i in system mode (i.e. no guest or VMEXIT steps of processor i) and not more than one program thread I/O step*[3].

$$
\begin{aligned}
&\forall \beta, h^0, i.\\
&\qquad IPsched(h^0, \beta)\\
&\quad \wedge\, ISRblock(h^0, \beta, i)\\
&\quad \Longrightarrow \forall k \in [0 : n-1].\ (\beta_k \notin \{(\textbf{vmexit}, i), (\textbf{guest}, i)\}\\
&\qquad \wedge\, (IOstep(h_k, \beta_k) \wedge \beta_k.pid = i \Longrightarrow IP^p(h^0, \beta, i, k)))
\end{aligned}
$$

where $n = |\beta|$.

Proof The lemma is trivially proven by the IP schedule condition and the corresponding distribution of I/O steps and IP points.

✔

We aim at reordering all IPI ISR blocks in an original execution, without changing the order of all other steps.

◀ **Definition** 4.24 Reorder ISR Block

The function *reoISRb* returns for a given IPI ISR block β reordered execution sequence equivalent to β according the following rules:

- The output sequence contains the same steps as β.
- The order of ISR steps is unchanged.
- The order of program thread steps is unchanged.
- The order of interleaved steps is unchanged.
- Local steps preceding an ISR are reordered after the ISR (Figure 4.9). In case of several ISRs, all local steps are reordered after the last ISR.

We have to define the function recursively in order to cover the cases with several IPI ISRs. We identify the first ISR in the original sequence β and require that the reordered sequence begins with it. Then we apply recursively the same step on the sub-sequence achieved from β after removing the first ISR and the subsequent interleaved steps. Note that an ISR block with several ISRs still satisfies the *ISRblock* predicate after removing the steps of one ISR. After the

[3]JISR and *eret* step count as interrupt thread steps.

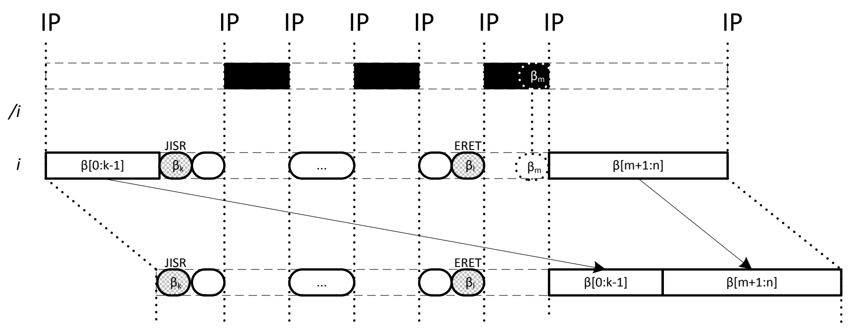

Figure 4.9: ISR block reordering.

last ISR has been removed in the sub-sequences remain only program thread steps.

$$reoISRb(h^0 \in C_M, \beta \in (\Sigma_M)^*, i \in Pid) \in (\Sigma_M)^* \overset{def}{=}$$

$$\begin{cases} \beta[k:m]reoISRb(h^0, \beta[0,k-1]\beta[m+1,n], i) & if\ ISRblock(h^0, \beta, i) \\ & \wedge ISRseq_{XT}(h^k, \beta[k,m], i) \\ & \wedge (k = 0 \\ & \quad \vee Lsteps_i(h, \beta[0, k-1])) \\ & \wedge (m < n \implies \\ & \quad core(\beta_{m+1}) \wedge \beta_{m+1}.pid = i) \\ \beta & otherwise \end{cases}$$

where

$$n = |\beta| - 1$$

Definition 4.25 ▶ Select Processor Steps

The function $steps_i$ returns for a given execution sequence β the subsequence of steps of a given processor i.

$$steps_i(\beta \in (\Sigma_M)^*, i \in Pid) \in (\Sigma_M)^* \overset{def}{=}$$

$$\begin{cases} \varepsilon & if\ \beta = \varepsilon \\ \beta_0 steps_i(\beta[1:|\beta|-1], i) & if\ \beta_0.pid = i \\ steps_i(\beta[1:|\beta|-1], i) & otherwise \end{cases}$$

◀ **Definition** 4.26
Select I/O Steps

The function $steps_{I/O}$ returns for a given execution sequence β and an initial state h^0 the subsequence of I/O steps.

$$steps_{I/O}(h^0 \in C_M, \beta \in (\Sigma_M)^*) \in (\Sigma_M)^* \overset{def}{=}$$
$$\begin{cases} \varepsilon & if\ \beta = \varepsilon \\ \beta_0 steps_{I/O}(h^1, \beta[1 : |\beta| - 1]) & if\ IOstep(h^0 \in C_M, , \beta_0) \\ steps_{I/O}(h^1, \beta[1 : |\beta| - 1]) & otherwise \end{cases}$$

Lemma 4.9 (Reorder ISR Block) *If β is $MIPS_P$ execution sequence starting in h^0, that is an IP schedule and an ISR block of a given processor i and we apply $reoISRb$ to reorder β, then the resulting sequence satisfies the extend schedule predicate for processor i. The order of I/O steps and the local order of steps for other processors are preserved in the reordered schedule. Furthermore all interleaved steps are reordered with the same number of positions, thus the reordering does not insert new interleaving, i.e. interleaved blocks of other processors stay unchanged.*

$$\begin{aligned} &\forall \beta, h^0, i. \\ &\qquad IPsched(h^0, \beta) \\ &\quad \wedge ISRblock(h^0, \beta, i) \\ &\quad \Longrightarrow sched_{XT}{}^i(h^0, reoISRb(h^0, \beta, i)) \\ &\qquad \wedge \forall j \in Pid.\ j \neq i \Longrightarrow steps_i(\beta, j) = steps_i(reoISRb(h^0, \beta, i), j) \\ &\qquad \wedge steps_{I/O}(h^0, \beta) = steps_{I/O}(h^0, reoISRb(h^0, \beta, i)) \\ &\qquad \wedge \exists k. \forall l.\ \beta_l.pid \neq i \Longrightarrow \beta_l = reoISRb(h^0, \beta, i)[l - k] \end{aligned}$$

Proof By unfolding the definition of $reoISRb$ we can easily prove that every ISR block within an IP schedule may be reordered into a step sequence which is as an extended schedule, in which the order of interleaved and I/O steps is preserved. All interleaved steps are reordered with k positions to the left, where k is the number of local program thread steps of processor i, which are reordered to the end of the ISR block.

✔

The interrupt thread and the program thread are running on the same processor and we want to guarantee, that IPI ISR does not change local configuration components related to the execution of the program thread. The memory, i.e. stack and owned memory addresses, is protected by our extended memory ownership model. The core is saved and restored at the beginning and at the end of an ISR respectively.

Software Condition 6 (Valid ISR) *The predicate ISR_V denotes the validity for ISR implementations. It states that the execution of an IPI ISR must be transparent to the interrupted program thread*[4] *and requires that the local state of the program thread before and after an ISR are equivalent. For an execution sequence $h \xrightarrow{\beta} h^n$ with $|\beta| = n$ we define:*

$$ISR_V(h \in C_M, \beta \in (\Sigma_M)^n, o \in (\mathcal{O}_{XT})^n, i \in Pid) \overset{def}{=}$$

$$\begin{cases} \exists m < (n-1).\ jisr_i(h,\beta,m) & \mathit{if} \quad eret_i(h,\beta,n-1) \\ \quad \wedge(\forall k \in [m : n-1].\ o^k.O_i^p = o^m.O_i^p & \\ \qquad \wedge stack(o^k.O_i^s, 0, h^k.oCore_i.rsp) = & \\ \qquad\qquad stack(o^m.O_i^s, 0, h^m.core_i.rsp)) & \\ \quad \wedge(\neg\exists k \in [m+1 : n-2].\ (jisr_i(h,\beta,k) & \\ \qquad\qquad \vee eret_i(h,\beta,k))) & \\ \quad \wedge eq_i^L(h^n, h^m, o^m) & \\ \quad \wedge ISR_V(h, \beta[0:m-1], o[0:m-1], i) & \\ ISR_V(h, \beta[0:n-2], o[0:n-2], i) & \mathit{if} \quad n > 1 \\ & \quad \wedge \neg eret_i(h,\beta,n-1) \\ 1 & \mathit{otherwise} \end{cases}$$

Note on ownership transfer.
Due to Definition 4.7 ownership transfer in an ISR block may happen on five particular occasions.

- If the executing step is the last step of the block and it is an I/O step, then we have transfer of addresses between the program thread owns-set and the shared memory.
- If the executing step is JISR, $eret$, or some other ISR I/O step, then we have transfer of addresses between the interrupt thread owns-set and the shared memory.
- If the executing step is an I/O step and is not a step of the considered processor, then we have transfer of addresses between the owns-set of the stepping processor and the shared memory.

[4]The effect of the TLB flushes is visible to the guest but not to the hypervisor since we do not use address translation in system mode.

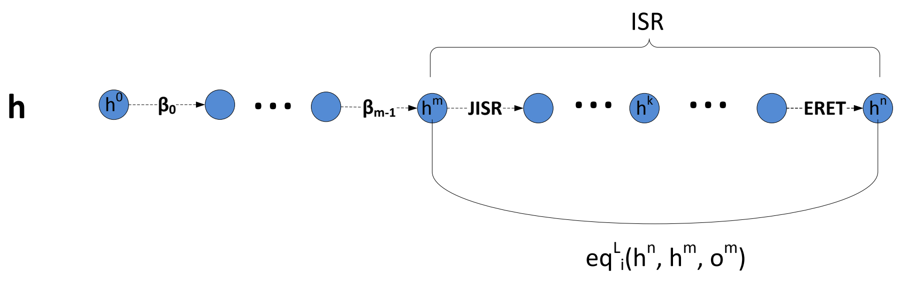

Figure 4.10: The state after a valid ISR is locally equal with respect to the corresponding processor i with the initial state of the ISR.

Software Condition 7 (JISR/ERET Ownership Transfer) *We allow at the begin and at the end of a handler execution ownership transfer according the following two rules.*

- *If the executing step is JISR, then the handler thread may acquire shared addresses.*
- *If the executing step is an $eret$, then the handler thread may release addresses into the shared set.*

4.2.2 Simulation Relation

We define a simulation relation between two machines. The first executes the original ISR block sequence and the second executes the reordered ISR block. We take into account how we express the desired reordering. We postpone all local program thread steps in the reordered execution and process them at once at the corresponding consistency point in the proper order. Thus we need to record the steps which are executed in the original execution and postponed in the reordered execution. The ISR steps in both executions happen simultaneously. In the proof we need to refer to the initial configuration in the interleaving point before JISR. Therefore we pass this configuration also to the coupling relation. Further we refer to the machine executing steps in the original order by h and to the machine with the reordered execution by d.

Definition 4.27 ▶ Simulation Relation

The predicate B defines the simulation relation, which couples the configurations in the executions. It gets as parameters

- the initial configuration h^0,
- the current configuration in the original execution h,
- the current configuration in the reordered execution d,
- the list of program thread steps executed in the original schedule and postponed in the reordered one γ,
- the current ownership o,
- the index of the processor i,
- a flag denoting whether the current state is a program thread interleaving point,
- a list of registers saved by the ISR in the IPCB sv,
- and a list of registers changed by the ISR ch.

Two configurations h and d satisfy our simulation relation if the following conditions hold.

- Environment and shared equality hold.
- During ISR h and d satisfy local ISR equality. Furthermore the state of the interrupted program thread, defined by the program thread owned addresses, the program stack addresses and the old-core, in d is equal to the initial configuration and in h it is defined by the local program thread steps γ executed in h^0.
- During program thread execution the handler thread memory except the IPCB region stores the same values. The relation of the program thread state in program thread IP is different than the one in all other states. In an IP h and d satisfy local equality and all postponed steps are executed. Otherwise the program thread state in d is equal to the initial configuration and in h its defined by the local program thread steps γ executed in h^0.

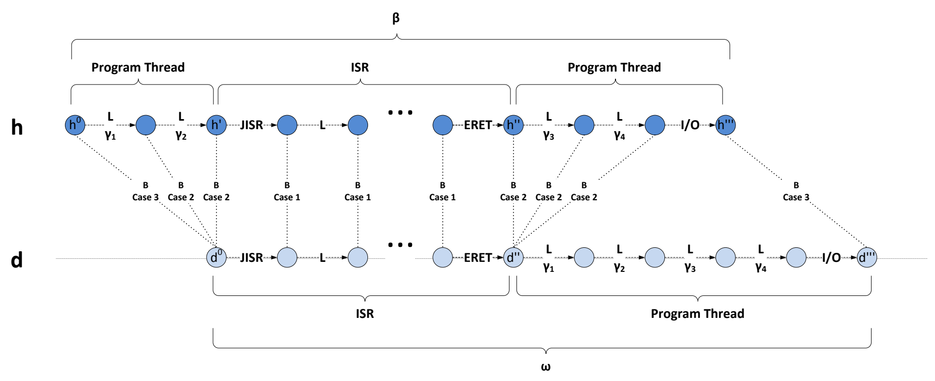

Figure 4.11: Simulation relation.

$$
\begin{aligned}
&B(h^0, h, d \in C_M, \gamma \in (\Sigma_M)^*, o \in \mathcal{O}_{XT}, i \in Pid,\\
&\qquad\qquad\qquad ip \in \mathbb{B}, ch \in 2^{\mathbb{B}^5}, sv \in 2^{\mathbb{B}^5}) \in \mathbb{B} \overset{def}{=}\\
&\quad eq_i^E(h, d, o)\\
&\quad \wedge\, eq_i^{Sh}(h, d, o)\\
&\quad \wedge\, isr(h.apic_i) \implies && (Case1)\\
&\qquad eq_i^{LH}(h, d, o, ch, sv)\\
&\qquad \wedge\, h.oCore_i = \Delta^{|\gamma|}(h^0, \gamma).core_i\\
&\qquad \wedge\, d.oCore_i = h^0.core_i\\
&\qquad \wedge\, \forall a \in o.O_i^p \cup stack(o.O_i^s, 0, h.oCore_i.gpr(rsp)).\\
&\qquad\qquad h.M(a) = \Delta^{|\gamma|}(h^0, \gamma).M(a)\\
&\qquad\qquad \wedge\, (a > d.oCore_i.gpr(rsp) \implies d.M(a) = h^0.M(a))\\
&\quad \wedge\, \neg isr(h.apic_i) \implies\\
&\qquad \forall a \in o.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h.M(a) = d.M(a)\\
&\qquad \wedge\, (\neg ip \implies eq_i^L(h, \Delta^{|\gamma|}(h^0, \gamma), o) && (Case2)\\
&\qquad\quad \wedge\, eq_i^L(h^0, d, o))\\
&\qquad \wedge\, (ip \implies \gamma = \varepsilon && (Case3)\\
&\qquad\quad \wedge\, eq_i^L(h, d, o))
\end{aligned}
$$

4.2.3 Simulation Theorem

Theorem 4.10 (ISR Block Simulation Theorem) *For every ISR block of any processor i in a $MIPS_P$ IP schedule execution defined by initial configuration h^0 and input sequence β, if every extended schedule execution sequence starting from h^0 contains only valid ISRs and is safe with respect to some ownership sequence, and d^m is a configuration equivalent to the initial configuration of the ISR block h^m, there exist an input sequence ω and a step function[5] $s \in \mathbb{N} \to \mathbb{N}$, such that if we execute ω starting in d^m , then:*

- *the execution sequence defined by d^m and ω is an extended IP schedule for processor i and*
- *the simulation relation B is maintained for every step l in the ISR block between configurations h^l and $d^{s(l)}$.*

$$
\begin{aligned}
&\forall h^0, \beta, d^m, i, o, m, n.\\
&\quad IPsched(h^0, \beta)\\
&\quad \wedge ISRblock(h^m, \beta[m:n], i)\\
&\quad \wedge (\forall \rho.\ \exists o'.\ sched_{XT}(h^0, \rho) \implies (ISR_V(h^0, \rho) \wedge safeseq_{XT}(h^0, \rho, o, o')))\\
&\quad \wedge B(h^m, h^m, d^m, \varepsilon, o, i, 1, \emptyset, \emptyset)\\
&\quad \implies \exists \omega, s.\ {sched_{XT}}^i(d^m, \omega)\\
&\qquad \wedge \forall l \in [m : n+1].\ \exists \gamma, o', ch, sv.\ B(h^m, h^l, d^{s(l)}, \gamma, o', i, ip^l, ch, sv)
\end{aligned}
$$

where

$$ip^l = IP^p(h^0, \beta, i, l)$$

We note that by the definition of ISR block only the first and the last configurations are an interleaving points of the program thread.

Proof First we instantiate ω with the reordered version of the ISR block $\beta[m : n]$

$$\omega = reoISRb(h^m, \beta[m:n], i)$$

and apply Lemma 4.9 to conclude that this instantiation results in an extended IP schedule execution for processor i.

As a second step we prove by induction over l the simulation relation for every state h^l and the corresponding state $d^{s(l)}$.

Induction base $\quad l = m$
The simulation relation for the base case

$$B(h^m, h^m, d^{s(m)}, \gamma^m, o^m, i, ip^m, ch^m sv^m)$$

[5]A step function is a monotonically increasing function defined on a subset of integers.

is trivially proven by the hypothesis, when we set $s(m) = m$ and instantiate the parameters as follows.

$$\begin{aligned} \gamma^m &= \varepsilon \\ o^m &= o \\ ip^m &= 1 \\ ch^m &= \emptyset \\ sv^m &= \emptyset \end{aligned}$$

Induction step $\quad l \to l+1$

In the induction step we make a case split on β_l.

The hypothesis of our theorem does not depend on our induction parameter. Thus for proving the claim of the theorem for the state after executing β_l (this we call induction claim) we can rely on the theorem's hypothesis and on the theorem's claim for l(this we call induction hypothesis), i.e. the simulation relation before executing step β_l of the original execution sequence.

$$\begin{aligned} &\forall h^0, \beta, d^m, i, o, m, n. \\ &\quad \mathit{IPsched}(h^0, \beta) \\ &\quad \wedge \mathit{ISRblock}(h^m, \beta[m:n], i) \\ &\quad \wedge B(h^m, h^m, d^m, \varepsilon, o, i, 1, \emptyset, \emptyset) \\ &\quad \wedge \forall \rho.\ \exists o'.\ \mathit{sched}_{XT}(h^0, \rho) \implies \mathit{ISR}_V(h^0, \rho) \wedge \mathit{safeseq}_{XT}(h^0, \rho, o, o') \\ &\quad \wedge \exists \gamma^l, o^l, ch^l, sv^l.\ B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, ip^l, ch^l, sv^l) \\ &\quad \implies \exists \gamma^{l+1}, o^{l+1}, ch^{l+1}, sv^{l+1}.\ B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, ip^{l+1}, ch^{l+1}, sv^{l+1}) \end{aligned}$$

From Lemma 4.8 we can conclude the following properties for β_l.

- β_l can not be a guest step or a VMEXIT step of processor i. Which means that the steps of processor i in $\beta[m:n]$ are only core steps in system mode.
- During ISR we may interleave with IPI steps or steps of other processors (guest and system mode).
- Not more than one program thread step[6] may be an I/O step.

Due to the structure of our simulation relation we have to distinguish on two parameters in the configurations in the induction hypothesis and in the induction claim.

[6] JISR and *eret* step count as interrupt thread steps.

- Are the states h^l and h^{l+1} an ISR or program thread states, i.e. is the ISR bit set?
- Are the states h^l and h^{l+1} a program thread IP?

Based on the ISR state we define five categories of steps.

- Program thread steps: start and end in a state with a clear ISR bit.
- JISR step: starts in a state with a clear ISR bit and ends in a state with a set ISR bit.
- ISR steps: start and end in a state with a set ISR bit.
- $eret$ step: starts in a state with a set ISR bit and ends in a state with a clear ISR bit.
- Other steps, i.e. IPI steps or steps of other processors.

According to the definition of the ISR block we know that we have a program thread IP only at the beginning and at the end of an ISR block. This basically means that we have an additional case distinction on the state before a JISR, i.e. a JISR step may start in both an IP state and a "normal" state, and on the state before and after a program thread step. may:

- The first program thread step in the block (i.e. if the block starts with a local program thread step, which is followed by more program thread steps or JISR) starts in a program thread IP and ends in a "normal" state.
- The last program thread step in the block starts in a "normal" state and ends in a program thread IP.
- All other program thread steps neither start nor end in a program thread IP.

Based on the above observations we define eight proof cases.
Case 1: β_l is the first program thread step in the block.

$$l = m$$

$$s(l) = m$$

Since we assume β_l to be a program thread step, we can deduce

$$\neg(isr(h^l.apic_i) \vee isr(h^{l+1}.apic_i)).$$

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^m, d^m, \varepsilon, o^l, i, 1, \emptyset, \emptyset) = \\
&\quad eq_i^E(h^m, d^m, o^l) \\
&\wedge\, eq_i^{Sh}(h^m, d^m, o^l) \\
&\wedge\, \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^m.M(a) = d^m.M(a) \\
&\wedge\, eq_i^L(h^m, d^m, o^l)
\end{aligned}$$

The execution of β_l in h^l changes only local components of processor i, i.e. the core, the local program thread memory and the allocated portion of the stack. On d we do not execute any step. β_l is attached to the list of postponed steps. All other simulation relation parameters stay the same as in the induction hypothesis.

$$\begin{aligned}
&s(l+1) = s(l) = m \\
&h^{l+1} = \Delta(h^l, (\textbf{core}, i, 0)) \\
&d^{s(l+1)} = d^m \\
&\gamma^{l+1} = \beta_l \\
&o^{l+1} = o^l \\
&ip^{l+1} = 0 \\
&ch^{l+1} = \emptyset \\
&sv^{l+1} = \emptyset
\end{aligned}$$

The intended simulation relation after β_l is :

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, ip^{l+1}, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, \forall a \in o^{l+1}.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l+1)}.M(a) \\
&\wedge\, eq_i^L(h^{l+1}, \Delta^{|\gamma^{l+1}|}(h^m, \gamma^{l+1}), o^{l+1}) \\
&\wedge\, eq_i^L(h^m, d^{s(l+1)}, o^{l+1})
\end{aligned}$$

after instantiation of the parameters

$$\begin{aligned}
&B(h^m, h^{m+1}, d^m, \beta_l, o^l, i, 0, \emptyset, \emptyset) = \\
&\quad eq_i^E(h^{m+1}, d^m, o^l) \\
&\wedge eq_i^{Sh}(h^{m+1}, d^m, o^l) \\
&\wedge \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{m+1}.M(a) = d^m.M(a) \\
&\wedge eq_i^L(h^{m+1}, \Delta(h^m, \beta_l), o^l) \\
&\wedge eq_i^L(h^m, d^m, o^l)
\end{aligned}$$

We can apply Lemma 4.6 which proves the environment and the shared equality between h^{m+1} and d^m(the first two conjuncts). The third one is trivially proven by the hypothesis, since the ownership safety guarantees that addresses owned by the interrupt handler and their memory content are unchanged by the program thread. The local equality $eq_i^L(h^{m+1}, \Delta(h^m, \beta_l), o^l)$ is trivial, since h^{m+1} is the resulting configuration after executing β_l in h^m. The local equality $eq_i^L(h^m, d^m, o^l)$ is contained in the hypothesis.

Case 2: β_l is the last program thread step in the block.

$$l = n$$

Since we assume β_l to be a program thread step, we can deduce

$$\neg(isr(h^l.apic_i) \vee isr(h^{l+1}.apic_i)).$$

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\wedge eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\wedge \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^l.M(a) = d^{s(l)}.M(a) \\
&\wedge eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l,), o^l) \\
&\wedge eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

h^{l+1} is defined by the execution of β_l in h^l and is an IP. On d we execute the list of postponed steps γ^l and β_l to define $d^{s(l+1)}$.

$$\begin{aligned}
&s(l+1) = n + 1 \\
&h^{l+1} = \Delta(h^l, (\textbf{core}, i, 0)) \\
&d^{s(l+1)} = \Delta^{|\gamma^l|+1}(d^{s(l)}, \gamma^l \beta_l)
\end{aligned}$$

We split the execution of $\gamma^l\beta_n$ and define an intermediate configuration

$$d' = \Delta^{|\gamma^l|}(d^{s(l)}, \gamma^l)$$

after executing γ^l in $d^{s(l)}$.

If we look at the last conjunct of the simulation relation from the induction hypothesis

$$eq_i^L(h^m, d^{s(l)}, o^l)$$

and apply ($|\gamma^l|$ times) Lemma 4.5 we get

$$eq_i^L(\Delta^{|\gamma^l|}(h^m, \gamma^l), \Delta^{|\gamma^l|}(d^{s(l)}, \gamma^l), o^l)$$

which in combination with the second last conjunct of the simulation relation from the induction hypothesis

$$eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l)$$

implies

$$eq_i^L(h^l, \Delta^{|\gamma^l|}(d^{s(l)}, \gamma^l), o^l).$$

Hence d' is locally consistent with h^l.

Additionally from our induction hypothesis we know, that h^l and $d^{s(l)}$ are consistent according the shared components and the environment.

$$eq_i^E(h^l, d^{s(l)}, o^l) \wedge eq_i^{Sh}(h^l, d^{s(l)}, o^l)$$

This consistency is maintained by the execution of local steps(Lemma 4.6).

$$eq_i^E(h^l, \Delta^{|\gamma^l|}(d^{s(l)}, \gamma^l), o^l) \wedge eq_i^{Sh}(h^l, \Delta^{|\gamma^l|}(d^{s(l)}, \gamma^l), o^l)$$

Hence d' is consistent according the shared components and the environment with h^l.

Furthermore the execution of program thread steps does not change addresses owned by the interrupt handler.

$$\forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^l.M(a) = d'.M(a)$$

Now let us consider the execution of β_l on both machines. Since β_l might be an I/O step, we need a case distinction. Most of the instantiation parameters for the simulation relation have the same values in both cases, i.e. the execution ends in an IP, all postponed instructions are executed, the lists of changed and stored registers do not change.

$$\begin{aligned}
\gamma^{l+1} &= \varepsilon \\
ip^{l+1} &= 1 \\
ch^{l+1} &= ch^l \\
sv^{l+1} &= sv^l
\end{aligned}$$

The intended simulation relation after β_l is :

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \varepsilon, o^{l+1}, i, 1, ch^l, sv^l) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\ eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\ \forall a \in o^{l+1}.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l+1)}.M(a) \\
&\wedge\ eq_i^L(h^{l+1}, d^{s(l+1)}, o^{l+1})
\end{aligned}$$

Case 2.1: The execution of β_l in h^l changes only local components of processor i, i.e. the core, the local program thread memory and the allocated portion of the stack. The ownership does not change.

$$o^{l+1} = o^l$$

Obviously the execution of β_l on both machines d' and h^l will maintain the consistency . Since all CPUs and all memory addresses (except IPCB and unallocated stack addresses) are equal in d' and h^l and ownership safety only allows accesses to those equal resources, the same will hold also for h^{l+1} and $d^{s(l+1)}$.

$$\begin{aligned}
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^l) \\
&\wedge\ eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^l) \\
&\wedge\ \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l+1)}.M(a) \\
&\wedge\ eq_i^L(h^{l+1}, d^{s(l+1)}, o^l)
\end{aligned}$$

Case 2.2: The execution of β_l might change also shared components. Additionally the ownership might change. As in the previous case the changes in the configurations will be equivalent on both machines and preserve the equality (Lemma 4.7). Still since the ownership is a parameter of our simulation relation we have to look deeper into the consistencies. We have to guarantee that if the ownership changes the predicates still hold. Due to ownership safety, changes in the ownership may appear only in $o.\mathbb{A}^{sh}$ and $o.O_i^p$ (Definition 4.5). But since the union of these two sets of addresses stays unchanged (Definition 3.7), the changes may only lead to redistribution of consistency conditions for particular addresses between the predicates for local or shared equality, which are trivially proven by the hypothesis.
Case 3: β_l is a program thread local step, which does not border on an IP.
Since we assume β_l to be a program thread step, we can deduce

$$\neg(isr(h^l.apic_i) \vee isr(h^{l+1}.apic_i)).$$

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\wedge eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\wedge \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^l.M(a) = d^{s(l)}.M(a) \\
&\wedge eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l) \\
&\wedge eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

The execution of β_l in h^l changes only local components of processor i, i.e. the core, the local program thread memory and the allocated portion of the stack. On d we do not execute any step. β_l is attached to the list of postponed steps. All other simulation relation parameters stay the same as in the induction hypothesis.

$$\begin{aligned}
&s(l+1) = s(l) \\
&h^{l+1} = \Delta(h^l, (\textbf{core}, i, 0)) \\
&d^{s(l+1)} = d^{s(l)} \\
&\gamma^{l+1} = \gamma^l \beta_l \\
&o^{l+1} = o^l \\
&ip^{l+1} = 0 \\
&ch^{l+1} = ch^l \\
&sv^{l+1} = sv^l
\end{aligned}$$

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, 0, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l)}, o^l) \\
&\wedge eq_i^{Sh}(h^{l+1}, d^{s(l)}, o^l) \\
&\wedge \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l)}.M(a) \\
&\wedge eq_i^L(h^{l+1}, \Delta(h^m, \gamma^{l+1}), o^l) \\
&\wedge eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

We can apply Lemma 4.6 which proves the environment and the shared equality between h^{l+1} and $d^{s(l)}$(the first two conjuncts). The third one is trivially proven by the hypothesis, since the ownership safety guarantees that addresses owned by the interrupt handler are unchanged by the program thread. The local equality $eq_i^L(h^{l+1}, \Delta(h^m, \gamma^{l+1}), o^l)$ is trivially proven by the local equality

$eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l)$ from the hypothesis and Lemma 4.5.

$$\begin{aligned}
&eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l)\\
&\qquad \Longrightarrow eq_i^L(\Delta(h^l, \beta_l), \Delta(\Delta^{|\gamma^l|}(h^m, \gamma^l), \beta_l), o^l) \qquad (Lemma 4.5)\\
&\qquad \Longrightarrow eq_i^L(h^{l+1}, \Delta(h^m, \gamma^{l+1}), o^l)
\end{aligned}$$

The local equality $eq_i^L(h^m, d^{s(l)}, o^l)$ is contained in the hypothesis.

Case 4: β_l is a JISR step, that does not start in an IP.
Since we assume β_l to be a JISR step, we can deduce

$$\neg isr(h^l.apic_i) \wedge isr(h^{l+1}.apic_i).$$

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) =\\
&\quad eq_i^E(h^l, d^{s(l)}, o^l)\\
&\wedge\, eq_i^{Sh}(h^l, d^{s(l)}, o^l)\\
&\wedge\, \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^l.M(a) = d^{s(l)}.M(a)\\
&\wedge\, eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l)\\
&\wedge\, eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

From the shared equality in the simulation relation we know, that the APICs on both machines are equal and the IPI signal is active. Thus we proceed by executing a JISR on h^l and $d^{s(l)}$. The execution of β_l changes local components and the local APIC of processor i. The list of the postponed steps does not change. The ownership may change. The list of changed and stored registers are empty, as this is the initial state of the ISR.

$$\begin{aligned}
&h^{l+1} = \Delta(h^l, (\textbf{core}, i, 1))\\
&d^{s(l+1)} = \Delta(d^{s(l)}, (\textbf{core}, i, 1))\\
&\gamma^{l+1} = \gamma^l\\
&ip^{l+1} = 0\\
&ch^{l+1} = \emptyset\\
&sv^{l+1} = \emptyset
\end{aligned}$$

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, 0, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge eq_i^{LH}(h^{l+1}, d^{s(l+1)}, o^{l+1}, \emptyset, \emptyset) \\
&\wedge h^{l+1}.oCore_i = \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i \\
&\wedge d^{s(l+1)}.oCore_i = h^m.core_i \\
&\wedge \forall a \in o^{l+1}.O_i^p \cup stack(o^{l+1}.O_i^s, 0, h^{l+1}.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^{l+1}.M(a) = \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) \\
&\qquad\qquad \wedge d^{s(l+1)}.M(a) = h^m.M(a)
\end{aligned}$$

The execution of the JISR step, as defined by Definition 4.1, Definition 2.20 and Definition 2.44, changes only the special purpose registers, the program counter, the APIC and copies the core into $oCore$. The possible ownership transfer is reduced by Software Condition 7 to acquiring addresses from the shared memory $o.\mathbb{A}^{sh}$ into the owns-set of the handler thread $o.O_i^h$. Since

$$\forall j \neq i. \;\; o^{l+1}.O_j = o^l.O_j$$

and

$$o^{l+1}.\;\mathbb{A}^{sh} \subseteq o^l.\mathbb{A}^{sh}$$

the ownership transfer does not generate some additional proof obligations for shared and environment equality. Thus automatically the environment equality is preserved. The change in the APICs are equivalent on both machines, therefore the shared equality also is maintained.

Furthermore with respect to the ISR local equality

$$eq_i^{LH}(h^{l+1}, d^{s(l+1)}, o^{l+1}, \emptyset, \emptyset)$$

we know from the $MIPS_P$ transition function and the definition of δ^{jisr}, that the program counter has the same value on both machines, and that no GPR register is changed by the JISR step.

$$\begin{aligned}
&h^{l+1}.pc_i d^{s(l+1)}.pc_i = 0 \\
&\wedge h^{l+1}.gpr = h^l.gpr \\
&\wedge d^{l+1}.gpr = d^l.gpr
\end{aligned}$$

Thus instantiating the sets of changed registers ch with the empty set satisfies the simulation conjuncts about the GPRs. The SPR registers are equal due to the JISR semantics and the hypothesis.

We also know from the hypothesis that in h^l and $d^{s(l)}$ the shared memory is equal. The same holds for the handler memory excluding the IPCB. Since none of them is changed by JISR, this implies that the newly acquired addresses store the same value on both machines, and hence the handler memory excluding the IPCB is equal also in the new configurations

$$\forall a \in o^{l+1}.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l+1)}.M(a)\ .$$

The set of registers stored in the IPCB sv is empty and the corresponding simulation relation conjunct

$$ipcbeq(h^{l+1}, d^{s(l+1)}, i, \emptyset)$$

is trivially proven. The condition about the handler stack region is fulfilled, since no memory address is allocated for it

$$\neg\exists a \in o^l.O_i^s.\ (h^{l+1}.oCore_i.gpr(rsp) \geq a) \wedge (a > h^{l+1}.core_i.gpr(rsp))\ .$$

Therefore the ISR local equality holds after β_l.

The conditions about the $oCore$ components are proven with the local equality from the hypothesis

$$eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l) \wedge eq_i^L(h^m, d^{s(l)}, o^l) \qquad (IH)$$

since $oCore$ stores the core configurations from the pre-state.

$$\begin{aligned} h^{l+1}.oCore_i &= h^l.core_i && (\delta^{jisr}) \\ &= \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i && (IH) \end{aligned}$$

$$\begin{aligned} d^{s(l+1)}.oCore_i &= d^{s(l)}.core_i && (\delta^{jisr}) \\ &= h^m.core_i && (IH) \end{aligned}$$

The equality of the program thread memory is proven by the local equality claimed in the hypothesis.

Case 5: β_l is JISR step, that does start in an IP.
This case is similar to the previous one. Since we start in an IP, i.e. at the beginning of the ISR block, $l = m$. We execute the same step on both machines, the lists of postponed steps before and after JISR are empty.

$$\gamma^l = \gamma^{l+1} = \varepsilon$$

Difference to **Case 4** appears in the local equality condition of the hypothesis:

$$\begin{aligned}
&B(h^m, h^m, d^{s(l)}, \varepsilon, o^l, i, 1, ch^l, sv^l) = \\
&\quad eq_i^E(h^m, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{Sh}(h^m, d^{s(l)}, o^l) \\
&\wedge\, \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\; h^m.M(a) = d^{s(l)}.M(a) \\
&\wedge\, eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, 0, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{LH}(h^{l+1}, d^{s(l+1)}, o^{l+1}, \emptyset, \emptyset) \\
&\wedge\, h^{l+1}.oCore_i = h^m.core_i \\
&\wedge\, d^{s(l+1)}.oCore_i = h^m.core_i \\
&\wedge\, \forall a \in o^{l+1}.O_i^p \cup stack(o^{l+1}.O_i^s, 0, h^{l+1}.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^{l+1}.M(a) = h^m.M(a) \\
&\qquad\quad \wedge\, d^{s(l+1)}.M(a) = h^m.M(a)
\end{aligned}$$

To prove the simulation in this case we use identical arguments with the previous case, but with an empty list of postponed instructions.

Case 6: β_l is an ISR step (between JISR and $eret$).

Since we assume β_l to be an ISR step, we can deduce

$$isr(h^l.apic_i) \wedge isr(h^{l+1}.apic_i).$$

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{LH}(h^l, d^{s(l)}, o^l, ch^l, sv^l) \\
&\wedge\, h^l.oCore_i = \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i \\
&\wedge\, d^{s(l)}.oCore_i = h^m.core_i \\
&\wedge\, \forall a \in o^l.O_i^p \cup stack(o^l.O_i^s, 0, h^l.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^l.M(a) = \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) \\
&\qquad\quad \wedge\, d^{s(l)}.M(a) = h^m.M(a)
\end{aligned}$$

The simulation relation in the induction claim has the same structure and contains the same conjuncts.

ISR steps are executed simultaneously on both machines. According our $MIPS_P$ model ISR steps are non interrupted core steps. They may depend on the shared memory, the interrupt thread's owned memory, on the stack and on the GPR content. We consider three sub-cases - for save context steps, restore context steps and the handler steps in-between. We have the following instantiation of the simulation parameter in all three cases:

$$\begin{aligned}
&s(l+1) = s(l) + 1\\
&h^{l+1} = \Delta(h^l, (\textbf{core}, i, 0))\\
&d^{s(l+1)} = \Delta(d^{s(l)}, (\textbf{core}, i, 0))\\
&\gamma^{l+1} = \gamma^l\\
&ip^{l+1} = 0
\end{aligned}$$

The other parameters we define corresponding to the particular sub-case semantics.

In the induction hypothesis of all three sub-cases we have shared and ISR local equality for processor i , which implies equality of the shared memory, the interrupt thread's owned memory, and the stack (with displacement). According to the GPR content we rely on Software Condition 5. Thus the handler stores the GPR initial values (i.e. from the state before JISR) in the IPCB first and overwrites the registers before reading them outside its save context part. Therefore relevant registers are equal in both configurations during ISR.

Case 6.1: β_l is a save context ISR step.
In this case we are considering a store instruction that saves an unchanged, hence not listed in sv^l and ch^l, register in the IPCB. In the following we denote by r the index of the register which is saved by β_l, where $ia_i(r)$ is the corresponding address in the IPCB. We instantiate the remaining simulation parameters considering the semantics of the store instruction execution. β_l is not an I/O step, hence the ownership does not change. The same is true for the list of changed registers. sv^{l+1} contains sv^l and r.

$$\begin{aligned}
&o^{l+1} = o^l\\
&ch^{l+1} = ch^l\\
&sv^{l+1} = sv^l \cup r
\end{aligned}$$

The only components changed by β_l on both machines are the program counter and the IPCB.

$$h^{l+1}.core_i.pc = h^l.core_i.pc + 4$$
$$h^{l+1}.M_4(ia_i(r)) = h^l.core_i.gpr(r)$$

$$d^{l+1}.pc_i = d^l.pc_i + 4$$
$$d^{l+1}.M_4(ia_i(r)) = d^l.core_i.gpr(r)$$

All simulation relation parts, which do not include these two components, are preserved. The only conjunct that we have to prove is the ISR local equality

$$eq_i^{LH}(h^{l+1}, d^{s(l+1)}, o^{l+1}, ch^{l+1}, sv^{l+1}) \ .$$

The program counter, all registers, and the content of all memory addresses owned the handler except the IPCB ones are trivially equal after β_l on both machines. The instantiation of sv and ch preserves the condition on them embedded in the GPR equality, i.e. $ch^{l+1} \subseteq sv^{l+1}$. The only noticeable argument is required by the IPCB condition.

$$\begin{aligned}
&ipcbeq(h^{l+1}, d^{s(l+1)}, i, sv^{l+1}) \overset{def}{=} \\
&\quad \forall a \in \mathbb{A}_{ipcb_i}.\ reg_i(a) \in sv^{l+1} \implies \\
&\qquad h^{l+1}.M_4(a[31:2]00) = h^{l+1}.oCore_i.gpr(reg_i(a)) \\
&\qquad \wedge\, d^{l+1}.M_4(a[31:2]00) = d^{l+1}.oCore_i.gpr(reg_i(a))
\end{aligned}$$

As we have said, β_l stores the content of register r in the memory word starting at $ia_i(r)$. Since the newly saved register has not yet been overwritten by the ISR (Software Condition 5), its value in the IPCB is equal to the corresponding value in the $oCore$ by the induction hypothesis. And we know that $oCore$ registers are written only by a JISR step and stay unchanged during ISR execution.

$$\begin{aligned}
h^{l+1}.M_4(ia_i(r)) &= h^l.core_i.gpr(r) && (\beta_l) \\
&= h^l.oCore_i.gpr(r) && (IH) \\
&= h^{l+1}.oCore_i.gpr(r) && (\beta_l)
\end{aligned}$$

The same holds also for d.

The only IPCB addresses changed by β_l are the $ia_i(r)$ and the three consecutive ones. For every address a from these four addresses reg_i returns the same value, i.e. the index r, and obviously the aligned version of every of the four addresses is $ia_i(r)$

$$\begin{aligned}
&\forall a \in \{ia_i(r),\ ia_i(r)[31:2]01,\ ia_i(r)[31:2]02,\ ia_i(r)[31:2]03\}. \\
&\qquad\qquad reg_i(a) = r \\
&\qquad\qquad \wedge\, a[31:2]00 = ia_i(r)
\end{aligned}$$

All other IPCB addresses corresponding to registers in sv^{l+1} store the proper $oCore$ registers due to the induction hypothesis. Thus, the IPCB condition is also fulfilled.

Case 6.2: β_l is a restore-context ISR step.

In this case we are considering a load instruction that restores a register value from the IPCB. In the following we denote by r the index of the register which is restored by β_l, where $ia_i(r)$ is the corresponding address in the IPCB. We instantiate the remaining simulation parameters considering the semantics of the load instruction execution. Like in the previous case β_l is not an I/O step, and the ownership does not change $o^{l+1} = o^l$. The list of stored registers does not change $sv^{l+1} = sv^l$. We have to exclude the index of the restored register from ch^l to obtain ch^{l+1}.

$$\begin{aligned} o^{l+1} &= o^l \\ ch^{l+1} &= ch^l \setminus r \\ sv^{l+1} &= sv^l \end{aligned}$$

Similarly to the previous case most of the simulation relation is trivially proven by the hypothesis, since the configuration changes are very limited. The only components changed by β_l are the program counter and the GPR.

$$\begin{aligned} h^{l+1}.core_i.pc &= h^l.core_i.pc + 4 \\ h^{l+1}.core_i.gpr(r) &= h^l.M_4(ia_i(r)) \end{aligned}$$

$$\begin{aligned} d^{l+1}.pc_i &= d^l.pc_i + 4 \\ d^{l+1}.core_i.gpr(r) &= d^l.M_4(ia_i(r)) \end{aligned}$$

The program counter equality is trivial. We only have to take care of the register, that β_l restores.

We restore the register value from the IPCB. By the induction hypothesis IPCB values correspond to the content of $oCore$ registers. We deduce that after β_l the considered register stores value equal to the one in $oCore$ on both machines. We show here only the argumentation for h.

$$\begin{aligned} h^{l+1}.core_i.gpr(r) &= h^l.M_4(ia_i(r)) && (\beta_l) \\ &= h^l.oCore_i.gpr(r) && (IH) \\ &= h^{l+1}.oCore_i.gpr(r) && (\beta_l) \end{aligned}$$

Thus the simulation relation holds.

Case 6.3: β_l is an ISR step, which does not belong to the save context or restore context part.

In this sub-case all components, which are relevant to the step's execution (input data), are equal in both machines. If the instruction reads memory, we rely on ownership safety and the hypothesis. Operands coming from general purpose registers are equal due Software Condition 5 (i.e. any source register rs has been previously overwritten by the ISR and is included in ch^l) and the hypothesis. Based on the same arguments as in Lemma 4.7 and Lemma 4.5, we claim that the changes to the configuration are equal on both machines, taking into account the relaxed local equality for handler thread machines. They basically convey that since the input data for the instruction processing is equal, then also the output data and hence the changes in the GPR, the stack, the handler owned memory or the shared memory are equivalent. IPCB, $oCore$, SPR and the owned memory of the program thread are not changed by the execution of β_l, due to the $MIPS_P$ architecture, ownership safety and our assumption that β_l is not a save context ISR step. With this arguments most of the simulation relation is proven trivially. In the following lines we look into the eq^{LH} conditions, which are not obviously holding.

We instantiate the remaining simulation parameters step by step, since we have to consider some case distinctions.

The list of stored registers does not change.

$$sv^{l+1} = sv^l$$

If the executed instruction does not change the GPR or writes to a destination register rd, which is contained in ch^l, then

$$ch^{l+1} = ch^l \ .$$

Otherwise we add the index of the newly changed register rd to the list of changed registers.

$$ch^{l+1} = ch^l \cup rd$$

This leeds to a new proof obligation

$$rd \in ch^{l+1} \implies d^{s(l+1)}.core_i.gpr(rd) = h^{l+1}.core_i.gpr(rd)$$

coming from the $gpreq$ predicate in the handler local equality. Since rd stores the same new value on h^{l+1} and $d^{s(l+1)}$ the statement is trivially proven.

If β_l is a local step, the ownership does not change

$$o^{l+1} = o^l$$

and the intended simulation relation after β_l is trivially proven.

Last we only have to examine the possible ownership transfer in case of an I/O step. If

$$o^{l+1} \neq o^l$$

the ownership transfer is possible between the shared memory $o.\mathbb{A}^{sh}$ and the owns-set of the handler thread $o.O^h_i$, excluding the IPCB addresses. The union of both sets however stays the same.

$$o^{l+1}.\mathbb{A}^{sh} \cup o^{l+1}.O^h_i = o^l.\mathbb{A}^{sh} \cup o^l.O^h_i$$

According to the induction hypothesis and the simulation relation in the state before β_l, both sets $o^l.\mathbb{A}^{sh}$ and $o^l.O_i^h$ collect addresses of memory, which stores the same values on h^l and $d^{s(l)}$ (i.e. due to eq^{Sh} and eq^{LH}). Thus, the redistribution of these addresses alone does not generate any additional proof obligations, considering the equality of shared and handler owned memory on h^{l+1} and $d^{s(l+1)}$. And as we said above, if β_l changes the memory, then the changes are equivalent on both machines. Therefore the intended simulation theorem is proven.

Case 7: β_l is an $eret$ step. The same step is executed simultaneously on both machines. The execution of β_l changes only the program counter, the SPR and the APIC.

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\quad \wedge eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\quad \wedge eq_i^{LH}(h^l, d^{s(l)}, o^l, \emptyset, sv^l) \\
&\quad \wedge h^l.oCore_i = \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i \\
&\quad \wedge d^{s(l)}.oCore_i = h^m.core_i \\
&\quad \wedge \forall a \in o^l.O_i^p \cup stack(o^l.O_i^s, 0, h^l.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^l.M(a) = \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) \\
&\qquad\qquad \wedge d^{s(l)}.M(a) = h^m.M(a)
\end{aligned}$$

We instantiate the simulation parameters as follows:

$$\begin{aligned}
s(l+1) &= s(l) + 1 \\
h^{l+1} &= \Delta(h^l, (\textbf{core}, i, 0)) \\
d^{s(l+1)} &= \Delta(d^{s(l)}, (\textbf{core}, i, 0)) \\
\gamma^{l+1} &= \gamma^l \\
ip^{l+1} &= 0 \\
ch^{l+1} &= ch^l \\
sv^{l+1} &= sv^l
\end{aligned}$$

Since we are executing an I/O step the ownership may change. If $o^{l+1} \neq o^l$ the ownership transfer is possible between the shared memory $o.\mathbb{A}^{sh}$ and the owns-set of the handler thread $o.O_i^h$, excluding the IPCB addresses (Software Conditions 4 and 7). Formally this is expressed by the following expression:

$$\begin{aligned}
&(\forall j \neq i.\ \ o^{l+1}.O_j = o^l.O_j) \\
\wedge\, &(o^{l+1}.O_i^p = o^l.O_i^p) \\
\wedge\, &(o^{l+1}.O_i^s = o^l.O_i^s) \\
\wedge\, &(o^{l+1}.O_i^h \cup o^{l+1}.\mathbb{A}^{sh} = o^l.O_i^h \cup o^l.\mathbb{A}^{sh})
\end{aligned}$$

Since β_l does not change any memory and both memory regions are equal in the state before β_l according to the simulation relation, the redistribution of addresses does not generate additional proof obligations.

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, ip^{l+1}, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, \forall a \in o^{l+1}.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l+1)}.M(a) \\
&\wedge\, eq_i^L(h^{l+1}, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^{l+1}) \\
&\wedge\, eq_i^L(h^m, d^{s(l+1)}, o^{l+1})
\end{aligned}$$

Since the changes on the APIC (the only non local component changed) are equivalent, the shared and environment equality are maintained. The equality of handler memory is part of the local handler equality eq^{LH} in the hypothesis. Next we examine the local equality between h^{l+1} and the initial configuration h^m with respect to the postponed steps γ^l.

$$eq_i^L(h^{l+1}, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^{l+1})$$

All memory addresses of the program thread memory and of the stack

$$\forall a \in o^{l+1}.O_i^p \cup stack(o^{l+1}.O_i^s, 0, h^{l+1}.core_i.rsp)$$

store the same values in the h^l and h^{l+1} due to the semantics of the $eret$ step and the induction hypothesis.

$$\begin{aligned}
h^{l+1}.M(a) &= h^l.M(a) && (\delta^{eret}) \\
&= \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) && (IH)
\end{aligned}$$

The IH says also that the GPR in the $oCore$ is equal to the GPR after executing γ^l on h^m. From Software Condition 6 we know that after β_l the GPR content must be equal to the one in the $oCore$. As part of the ISR local equality in the induction hypothesis the GPR of h^l is equal to the the GPR in the $oCore$ for all registers not listed in ch^l. Thus we rely on the restore context part of the handler to restore all changed registers and conclude an empty set of changed registers

$$ch^{l+1} = \emptyset \ .$$

When we consider in addition the $eret$ semantics we get:

$$\begin{aligned} h^{l+1}.core_i.gpr &= h^l.core_i.gpr && (\delta^{eret}) \\ &= h^l.oCore_i.gpr && (IH; eq_i^L; ch^{l+1} = \emptyset) \\ &= \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i.gpr && (IH) \end{aligned}$$

Thus we only have to prove the SPR equality and the equality of the program counter. We argue based on the $MIPS_P$ $eret$ definition, the SPR equality condition contained in the ISR local equality and the $oCore$ condition from the induction hypothesis.

$$\begin{aligned} h^{l+1}.core_i.pc &= h^l.core_i.spr(epc) && (\delta^{eret}) \\ &= h^l.oCore_i.pc && (IH; eq_i^L) \\ &= \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i.pc && (IH) \end{aligned}$$

$$\begin{aligned} h^{l+1}.core_i.spr(sr) &= h^l.core_i.spr(esr) && (\delta^{eret}) \\ &= h^l.oCore_i.spr(sr) && (IH; eq_i^L) \\ &= \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i.spr(sr) && (IH) \end{aligned}$$

$$\begin{aligned} h^{l+1}.core_i.spr(mode) &= h^l.core_i.spr(emode)(\delta^{eret}) \\ &= h^l.oCore_i.spr(mode) && (IH; eq_i^L) \\ &= \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i.spr(mode) && (IH) \end{aligned}$$

Thus $eq_i^L(h^{l+1}, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^{l+1})$ holds.

The proof for $eq_i^L(h^m, d^{s(l+1)}, o^{l+1})$ is similar and uses the same arguments.

Case 8: β_l is not a step of processor i but some interleaved step executed on another processor or an IPI step.

We remind the reader, that interleaving in an ISR block may happen only in the ISR sequence or right after an $eret$ step. We execute the same step on both machines. The execution of β_l does not change anything local to processor i. It may change components local to other processors and shared configuration

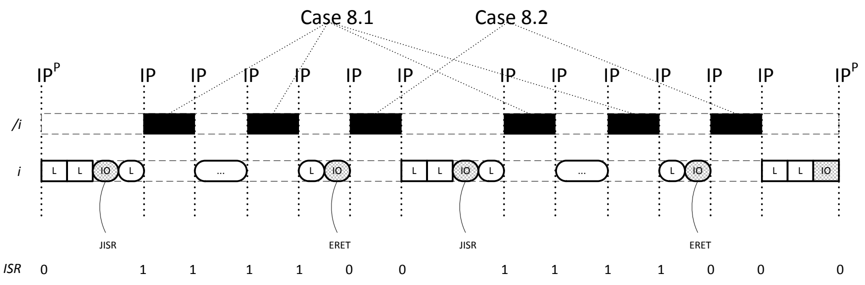

Figure 4.12: The simulation relation for interleaved steps depends on the state of the ISR bit. For interleaving with ISR steps the ISR bit is set (Case 8.1), whereas for interleaved steps right after an *eret* step the ISR bit is cleared.

elements. Thus the simulation relation parameters stay unchanged except for the configurations h and d, and the ownership.

$$\begin{aligned}
&s(l+1) = s(l) + 1\\
&h^{l+1} = \Delta(h^l, \beta_l)\\
&d^{s(l+1)} = \Delta(d^{s(l)}, \beta_l)\\
&\gamma^{l+1} = \gamma^l\\
&ip^{l+1} = 0\\
&ch^{l+1} = ch^l\\
&sv^{l+1} = sv^l
\end{aligned}$$

Since β_l may be an I/O step, the ownership may change but the ownership set of processor i will remain unchanged $o^l.O_i = o^{l+1}.O_i$.

Due the structure of our simulation relation, we need a case distinction on the location of β_l in the ISR block (Figure 4.12).

Case 8.1: β_l is an interleaved step bordered by ISR steps.

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{LH}(h^l, d^{s(l)}, o^l, ch^l, sv^l) \\
&\wedge\, h^l.oCore_i = \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i \\
&\wedge\, d^{s(l)}.oCore_i = h^m.core_i \\
&\wedge\, \forall a \in o^l.O_i^p \cup stack(o^l.O_i^s, 0, h^l.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^l.M(a) = \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) \\
&\qquad\qquad \wedge\, d^{s(l)}.M(a) = h^m.M(a)
\end{aligned}$$

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, 0, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{Sh}(h^{l+1}, d^{s(l+1)}, o^{l+1}) \\
&\wedge\, eq_i^{LH}(h^{l+1}, d^{s(l+1)}, o^{l+1}, ch^l, sv^l) \\
&\wedge\, h^{l+1}.oCore_i = \Delta^{|\gamma^l|}(h^m, \gamma^l).core_i \\
&\wedge\, d^{s(l+1)}.oCore_i = h^m.core_i \\
&\wedge\, \forall a \in o^l.O_i^p \cup stack(o^l.O_i^s, 0, h^{l+1}.oCore_i.gpr(rsp)). \\
&\qquad\qquad h^{l+1}.M(a) = \Delta^{|\gamma^l|}(h^m, \gamma^l).M(a) \\
&\qquad\qquad \wedge\, d^{s(l+1)}.M(a) = h^m.M(a)
\end{aligned}$$

If we apply Lemma 4.5 and Lemma 4.7 we know, that the changes on both machines will be equivalent and the simulation relation preserved.

Case 8.2: β_l is preceded by an $eret$ step and followed by a local program thread step.

The simulation relation in the induction hypothesis is defined by:

$$\begin{aligned}
&B(h^m, h^l, d^{s(l)}, \gamma^l, o^l, i, 0, ch^l, sv^l) = \\
&\quad eq_i^E(h^l, d^{s(l)}, o^l) \\
&\wedge\, eq_i^{Sh}(h^l, d^{s(l)}, o^l) \\
&\wedge\, \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^l.M(a) = d^{s(l)}.M(a) \\
&\wedge\, eq_i^L(h^l, \Delta^{|\gamma^l|}(h^m, \gamma^l), o^l) \\
&\wedge\, eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

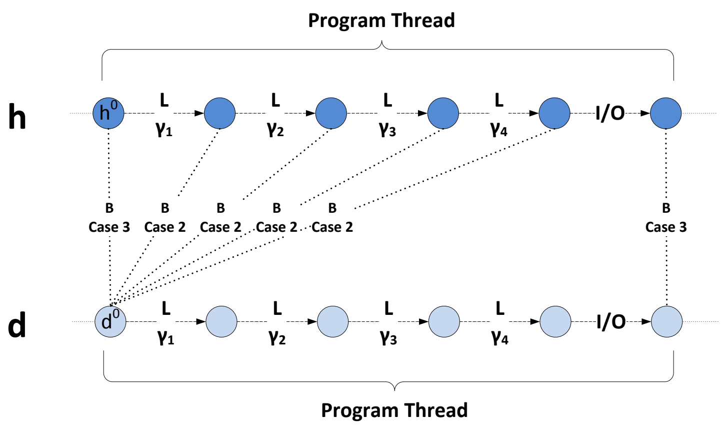

Figure 4.13: Simulation relation for interleaving blocks consisting only of program thread steps.

The intended simulation relation after β_l is:

$$\begin{aligned}
&B(h^m, h^{l+1}, d^{s(l+1)}, \gamma^{l+1}, o^{l+1}, i, 0, ch^{l+1}, sv^{l+1}) = \\
&\quad eq_i^E(h^{l+1}, d^{s(l)}, o^l) \\
&\quad \wedge\, eq_i^{Sh}(h^{l+1}, d^{s(l)}, o^l) \\
&\quad \wedge\, \forall a \in o^l.O_i^h \setminus \mathbb{A}_{ipcb_i}.\ h^{l+1}.M(a) = d^{s(l)}.M(a) \\
&\quad \wedge\, eq_i^L(h^{l+1}, \Delta(h^m, \gamma^{l+1}), o^l) \\
&\quad \wedge\, eq_i^L(h^m, d^{s(l)}, o^l)
\end{aligned}$$

We again apply Lemma 4.5 and Lemma 4.7 to conclude, that the changes on both machines will be equivalent and the simulation relation preserved.

✔

After proving the simulation for ISR blocks, we extend it to complete executions of a single core including non-ISR interleaving blocks. We remind the reader, that in the execution sequence a of given processor next to ISR blocks, we can have interleaving blocks consisting of hypervisor steps without ISR steps (Figure 4.13), VMEXIT steps, and guest steps.

Theorem 4.11 (Execution Simulation Theorem Single Processor) *For every $MIPS_P$ IP schedule execution defined by initial configuration h^0 and input sequence β , if every extended schedule execution sequence starting from h^0 contains only valid ISRs and is safe with respect to some ownership sequence, and d^0 is a configuration equivalent to the initial configuration h^0, there exist an input* *sequence ω and a step function $s \in \mathbb{N} \to \mathbb{N}$, such that if we execute ω starting in d^0, then:*

- *the execution sequence defined by d^0 and ω is an extended IP schedule for a given processor i and*
- *the simulation relation B is maintained for every step l between configurations h^l and $d^{s(l)}$.*

$$\begin{aligned}
&\forall h^0, \beta, d^0, i, o.\\
&\quad \mathit{IPsched}(h^0, \beta)\\
&\quad \wedge (\forall \rho.\ \exists o'.\ \mathit{sched}_{XT}(h^0, \rho) \implies (\mathit{ISR}_V(h^0, \rho) \wedge \mathit{safeseq}_{XT}(h^0, \rho, o, o')))\\
&\quad \wedge B(h^0, h^0, d^0, \varepsilon, o, i, 1, \emptyset, \emptyset)\\
&\quad \implies \exists \omega, s.\ \mathit{sched}_{XT}{}^i(d^0, \omega)\\
&\qquad \wedge \forall l \in [0 : |\beta| - 1].\ \exists \gamma, o', ch, sv, m \in [0 : l].\ B(h^m, h^l, d^{s(l)}, \gamma, o', i, ip^l, ch, sv)
\end{aligned}$$

where

$$ip^l = IP^p(h^0, \beta, i, l)$$

Proof We instantiate ω with the reordered version of β, in which all ISR blocks of processor i are reordered by the $\mathit{reoISRb}$ function and the order of all steps outside of ISR interleaving blocks stays unchanged. We Lemma 4.9 on the reordered blocks to conclude that this instantiation results in an extended IP schedule execution for processor i.

As a second step we prove by induction over l the simulation relation for every state h^l and the corresponding state $d^{s(l)}$.

Induction base $\quad l = 0$
The simulation relation for the base case

$$B(h^0, h^0, d^{s(l)}, \gamma^0, o^0, i, ip^0, ch^0 sv^0)$$

is trivially proven by the hypothesis, when we instantiate the parameters as follows.

$$\begin{aligned}
&s(l) = 0\\
&\gamma^0 = \varepsilon\\
&o^0 = o\\
&ip^0 = 1\\
&ch^0 = \emptyset\\
&sv^0 = \emptyset
\end{aligned}$$

Induction step $l \rightarrow l+1$
Similarly to the proof of Theorem 4.10 we have a case split on β_l. The proof is quite similar to the previous one. We can distinguish in general between steps inside and outside of ISR blocks. Steps within ISR blocks of processor i are described in the previous proof. Steps outside of the ISR blocks can be program thread steps of processor i, guest steps or VMEXIT steps of processor i, IPI steps, and steps of other processors.

The major difference in the proof is that now the reference initial configuration of the simulation relation h^m have to be updated every time, when we pass an interleaving point of the program thread, so that h^m is always equal to the last configuration, which is a program thread interleaving point. Thus, for β_l in an ISR block h^m is always equal to the initial configuration of the ISR block like in the pervious theorem.

- For steps within ISR blocks of processor i the proof is identical with the previous theorem.
- If β_l is a hypervisor step (i.e. core step in system mode) of processor i outside of the ISR blocks we have following cases:
  - β_l is the first step in an interleaving block. The proof is identical to **Case 1** of the previous theorem.
  - β_l is the last step in an interleaving block. The proof is identical to **Case 2** of the previous theorem.
  - β_l is a step, which does not border on an IP. The proof is identical to **Case 3** of the previous theorem.
- If β_l is a guest step or a VMEXIT step it is executed simultaneously on both machines and its execution does not change any configuration components related to the simulation relation. The simulation relation parameters also do not change. Thus the simulation relation follows from the induction hypothesis.
- If β_l is an IPI step or a step of another processor, the proof is identical to **Case 8.2** of the previous theorem.

✔

Finally we are ready to state and proof a theorem about reordering of complete execution sequences.

Theorem 4.12 (Execution Simulation Theorem) *For every $MIPS_P$ IP schedule execution defined by initial configuration h^0 and input sequence β , if every extended schedule execution sequence starting from h^0 contains only valid ISRs and is safe with respect to some ownership sequence, and d^0 is a configuration* *equivalent to the initial configuration h^0, there exist an input sequence ω and a step function $s \in \mathbb{N} \to \mathbb{N}$, such that if we execute ω starting in d^0, then:*

- *the execution sequence defined by d^0 and ω is an extended IP schedule and*
- *the simulation relation B is maintained for every step l between configurations h^l and $d^{s(l)}$.*

$$\begin{aligned}
&\forall h^0, \beta, d^0, o.\\
&\quad \mathit{IPsched}(h^0, \beta)\\
&\quad \wedge (\forall \rho.\ \exists o'.\ \mathit{sched}_{XT}(h^0, \rho) \implies (\mathit{ISR}_V(h^0, \rho) \wedge \mathit{safeseq}_{XT}(h^0, \rho, o, o')))\\
&\quad \wedge B(h^0, h^0, d^0, \varepsilon, o, i, 1, \emptyset, \emptyset)\\
&\quad \implies \exists \omega, s.\ \mathit{sched}_{XT}(d^0, \omega)\\
&\qquad \wedge \forall l \in [0 : |\beta| - 1], i \in \mathit{Pid}.\ \exists \gamma, o', ch, sv, m \in [0 : l].\\
&\qquad\qquad B(h^m, h^l, d^{s(l)}, \gamma, o', i, ip^l, ch, sv)
\end{aligned}$$

where

$$ip^l = \mathit{IP}^p(h^0, \beta, i, l)$$

Proof We prove the theorem by applying inductively Theorem 4.11 for all processors. In doing so we have to guarantee that applying Theorem 4.11 for a given processor i will preserve the properties concluded by preceding applications of the theorem for other processors. Due to Lemma 4.9 we can conclude that the reordering of the steps of one processor will neither change the order of steps nor split interleaved blocks of all other processors. In particular already reordered extended schedules for one processors will not be destroyed by subsequent reordering and application of Theorem 4.11.

Considering the simulation relation, we know that subsequent reordering of ISR blocks of a given processor i preserve shared equality and environment (according to i) equality. This is sufficient to conclude, that the reordering of local steps of processor i preserves the simulation relation of other processors.

✔

Chapter 5

C-IL Semantics

In this chapter we introduce semantics for the C Intermediate Language (C-IL), which is a simplified programming language similar to C. C-IL was developed and defined by Sabine Schmaltz and Andrey Shadrin [SS12] as part of the Verisoft XT verification technology. In this work we omit some definitions, which can be looked up in [SS12], e.g. technical details of the expression evaluation. We are aiming at stating a simulation relation between a *C-IL* computation and a $MIPS_P$ execution. Thus our focus concerning C-IL is on the definition of the step function and compiler correctness theorem. Our compiler correctness theorem is based on the work of Andrey Shadrin [Sha12] and Mikhail Kovalev [Kov13], which we adapt to $MIPS_P$. Furthermore, we define ownership safety for C-IL computations. We assume C-IL safety in order to deduce safety of the $MIPS_P$ execution.

In Section 5.1 we present the sequential C-IL semantics, which we later in Section 5.2 enhance to the concurrent case. In Section 5.3.4 we define ownership safety. Finaly in Section 5.3 we define the compiler correctness theorem.

In the this chapter we do not consider interrupts. They are added to the semantics in the next chapter.

5.1 Sequential C-IL Semantics

5.1.1 C-IL Types

As the most program languages *C-IL* supports primitive, composed and pointer types.

5.1.1.1 Primitive types

◀ **Definition** 5.1
Primitive types

The set of primitive types is defined by the void type **void** and six types for signed (**int**n) and unsigned (**uint**n) integer values, where n defines the size of the type in bits, i.e. 8, 16 and 32 bits. The Boolean type we implement using integers.

$$\mathbb{T}_P \overset{def}{=} \{\textbf{void}\} \cup \{\textbf{int}n \mid n \in \{8, 16, 32\}\} \cup \{\textbf{uint}n \mid n \in \{8, 16, 32\}\}$$

The range for each integer type is defined as follows:

- **int**n - n-bit signed integers

$$\textbf{int}n \overset{def}{=} \{-2^{n-1}, \dots,\ 2^{n-1} - 1\}$$

- **uint**n - n-bit unsigned integers

$$\textbf{uint}n \overset{def}{=} \{0, \dots,\ 2^{n} - 1\}$$

5.1.1.2 Complex types

The array type **array**(t, n) is defined by the type of the array elements t and their number n.

We define individually for each C_{IL}-program a set of struct type names $\mathbb{T}_C$. A struct type has a unique name in $\mathbb{T}_C$ and consists of several fields.

We define two types of pointers types in *C-IL*. By **ptr**(t) we denote the set of typed pointers to values of type t. A function pointer of type **fptr**(t, T) points to a function with return value of type t, where T is a list of the types of the functions' parameters.

Definition 5.2 ▶ C-IL types

The set of *C-IL* types $\mathbb{T}$ is defined inductively and consists of primitive types, pointer types, array types and struct types.

- primitive types: $t \in \mathbb{T}_P \implies t \in \mathbb{T}$,
- struct types: $t_c \in \mathbb{T}_C \implies$ **struct** $t_c \in \mathbb{T}$,
- array types: $t \in \mathbb{T}, n \in \mathbb{N} \implies$ **array**$(t, n) \in \mathbb{T}$,
- (regular) pointer types: $t \in \mathbb{T} \implies$ **ptr**$(t) \in \mathbb{T}$,
- function pointer types: $t \in \mathbb{T}, T \in \mathbb{T}^* \implies$ **fptr**$(t, T) \in \mathbb{T}$.

5.1.1.3 Type qualifiers

C-IL provides two type of qualifiers: **volatile** and **const**. Type qualifiers are used to provide additional information to the compiler in order to allow or forbid some optimizations of the code. They partially define the behavior of instances of the different types and especially how they can be accessed.

Definition 5.3 ▶ Set of type Qualifiers

We define the set of type qualifiers $\mathbb{Q}$ as:

$$\mathbb{Q} \overset{def}{=} \{\textbf{volatile}, \textbf{const}\}.$$

- The qualifier **volatile** defines shared data. It is used as a compiler directive that prevents the compiler from reordering volatile memory accesses and some other optimizations, i.e. volatile data is never cached in registers. This is highly useful in memory-mapped-I/O and in concurrent applications.

- The qualifier **const** defines constant values. A memory access performed on a variable of a type qualified with the const qualifier can not be a write. This may enable the compiler to perform additional optimizations on the code, relying on the fact that a given value in memory is never overwritten.

The set of qualified types $\mathbb{T}_{\mathbb{Q}}$ we define inductively as a composition of qualifiers and unqualified types from $\mathbb{T}$. $\mathbb{T}_{\mathbb{Q}}$ concists of:

◀ **Definition** 5.4 Qualified C-IL types

- qualified primitive types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_P \implies (q, t) \in \mathbb{T}_{\mathbb{Q}}$,
- qualified struct types: $q \subseteq \mathbb{Q} \wedge t_c \in \mathbb{T}_C \implies (q, \textbf{struct } t_c) \in \mathbb{T}_{\mathbb{Q}}$,
- qualified array types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}}, n \in \mathbb{N} \implies (q, \textbf{array}(t, n)) \in \mathbb{T}_{\mathbb{Q}}$,
- qualified (regular) pointer types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}} \implies (q, \textbf{ptr}(t)) \in \mathbb{T}_{\mathbb{Q}}$,
- qualified function pointer types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}}, T \in \mathbb{T}_{\mathbb{Q}}^* \implies (q, \textbf{fptr}(t, T)) \in \mathbb{T}_{\mathbb{Q}}$.

We allow that the set of qualifiers $q \subseteq \mathbb{Q}$ is empty. This makes it easy to convert unqualified types to qualified types, i.e. just adding an empty qualifiers set.

To turn a qualified type into an unqualified one, we throw away all the qualifiers.

◀ **Definition** 5.5 Converting Qualified Types to Unqualified

$$qt2t(x) \overset{def}{=}$$

$$\begin{cases} t & x = (q, t) \wedge t \in \mathbb{T}_P \\ \textbf{ptr}(qt2t(x')) & x = (q, \textbf{ptr}(x')) \\ \textbf{array}(qt2t(x'), n) & x = (q, \textbf{array}(x', n)) \\ \textbf{funptr}(qt2t(x'), [qt2t(X_1), \ldots, qt2t(X_n)]) & x = (q, \textbf{funptr}(x', [X_1, \ldots, X_n])) \\ \textbf{struct } t_c & x = (q, \textbf{struct } t_c) \end{cases}$$

In some cases, e.g. case distinctions in definitions, we want to distinguish among pointer and array types. Therefor we define the following type predicates.

The predicates *isptr*, *isarray* and *isfptr* denote that given type is a pointer type, an array type, or a function pointer type respectively.

◀ **Definition** 5.6 Type Predicates

$$isptr, isarray, isfptr :: \mathbb{T} \to \mathbb{B}$$

$$isptr(t) \overset{def}{=} \exists t'.\ t = \textbf{ptr}(t')$$
$$isarray(t) \overset{def}{=} \exists t', n'.\ t = \textbf{array}(t', n')$$
$$isfptr(t) \overset{def}{=} \exists t', T.\ t = \textbf{fptr}(t', T)$$

5.1.2 Values

Values in *C-IL* are bit-strings paired with a type specifier. In the following lines we define each of the value sets to the corresponding types, where $\mathbb{V}$ denotes the set of variables, and $\mathbb{F}_{name}$ denotes the set of function names.

With $size_{ptr} \in \mathbb{N}$ we refer to the archticture dependent size of pointers in bytes. For $MIPS_P$ we set $size_{ptr} = 4$.

- Primitive values:

$$n \in \{8, 16, 32\} \;\wedge\; i \in \mathbb{B}^n \Rightarrow val(i, \textbf{int}n) \in val_{\textbf{int}n}$$

$$n \in \{8, 16, 32\} \;\wedge\; i \in \mathbb{B}^n \Rightarrow val(i, \textbf{uint}n) \in val_{\textbf{uint}n}$$

- Structs:

$$t_C \in \mathbb{T}_C \;\wedge\; B \in (\mathbb{B}^8)^* \Rightarrow \textbf{val}(B, \textbf{struct } t_C) \in val_{\textbf{struct}}$$

- Pointer and array values:

$$a \in \mathbb{B}^{8 \cdot size_{ptr}} \wedge t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t)) \Rightarrow \textbf{val}(a, t) \in val_{\textbf{ptr(t)}} \subseteq val_{\textbf{ptr}}$$

- Local references values:

$$v \in \mathbb{V} \wedge o \in \mathbb{N} \wedge i \in \mathbb{N} \wedge t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t)) \Rightarrow \textbf{lref}((v, o), i, t) \in val_{\textbf{lref}}$$

- Function pointers values:

$$a \in \mathbb{B}^{8 \cdot size_{ptr}} \wedge t \in \mathbb{T} \wedge isfptr(t) \Rightarrow \textbf{val}(a, t) \in val_{\textbf{fptr}}$$

- Symbolic function values:

$$f \in \mathbb{F}_{name} \wedge isfptr(t) \Rightarrow \textbf{fun}(f, t) \in val_{\textbf{fun}}$$

Definition 5.7 ▶ *C-IL* values

We define the set of all possible values val as an union of the possible values of all types.

$$val \overset{def}{=} val_{\textbf{int}n} \cup val_{\textbf{uint}n} \cup val_{\textbf{struct}} \cup val_{\textbf{ptr}} \cup val_{\textbf{lref}} \cup val_{\textbf{fptr}} \cup val_{\textbf{fun}}$$

5.1.3 Expressions

C-IL provides the set of unary operators $\mathbb{O}_1$ and the set of binary operators $\mathbb{O}_2$ defined in the following way.

◀ **Definition** 5.8
Operators

$$\mathbb{O}_1 \subset \{\oplus \mid \oplus \in val \rightharpoonup val\}$$
$$\mathbb{O}_1 \overset{def}{=} \{-, \sim, !\}$$

$$\mathbb{O}_2 \subset \{\oplus \mid \oplus \in val \times val \rightharpoonup val\}$$
$$\mathbb{O}_2 \overset{def}{=} \{+, \text{-}, *, /, \%, <<, >>, <, >, <=, >=, ==, != , \&, |, \hat{}, \&\&, ||\}$$

We define the set of *C-IL* expressions $\mathbb{E}$ inductively. It contains:

◀ **Definition** 5.9
Expressions

- Constants: $c \in val \Rightarrow c \in \mathbb{E}$,
- Variable names: $v \in \mathbb{V} \Rightarrow v \in \mathbb{E}$,
- Function names: $f \in \mathbb{F}_{name} \Rightarrow f \in \mathbb{E}$,
- Unary operations on expressions: $e \in \mathbb{E} \wedge \oplus \in \mathbb{O}_1 \Rightarrow \oplus e \in \mathbb{E}$,
- Binary operations on expressions: $e_0, e_1 \in \mathbb{E} \wedge \oplus \in \mathbb{O}_2 \Rightarrow (e_0 \oplus e_1) \in \mathbb{E}$,
- Conditional: $e, e_0, e_1 \in \mathbb{E} \Rightarrow (e\ ?\ e_0 : e_1) \in \mathbb{E}$,
- Type cast: $t \in \mathbb{T}_Q \wedge e \in \mathbb{E} \Rightarrow (t)e \in \mathbb{E}$,
- Dereferencing pointer: $e \in \mathbb{E} \Rightarrow *e \in \mathbb{E}$,
- Address of expression: $e \in \mathbb{E} \Rightarrow \&e \in \mathbb{E}$,
- Field access: $e \in \mathbb{E} \wedge f \in \mathbb{F} \Rightarrow (e).f \in \mathbb{E}$, where $\mathbb{F}$ denotes the set of field names,
- Size of Type: $t \in \mathbb{T}_Q \Rightarrow$ **sizeof**(t),
- Size of Expression: $e \in \mathbb{E} \Rightarrow$ **sizeof**(e).

We define the following two shorthands in correspondence to the established C syntax.

- Access of array elements of a given array a.

$$a[i] \overset{def}{=} *(a + i)$$

- Access to fields in structs from a pointer a.

$$a \rightarrow f \overset{def}{=} (*(a)).f$$

5.1.4 Statements

Definition 5.10 ▶ *C-IL* Statements

We define the set of *C-IL* statements $\mathbb{S}$ inductively as follows:

- Assignment: $e_0, e_1 \in \mathbb{E} \Rightarrow e_0 = e_1 \in \mathbb{S}$
- Goto: $l \in \mathbb{N} \Rightarrow$ **goto** $l \in \mathbb{S}$
- If-Not-Goto: $e \in \mathbb{E} \ \wedge \ l \in \mathbb{N} \Rightarrow$ **ifnot** e **goto** $l \in \mathbb{S}$
- Function Call: $e_0, e \in \mathbb{E} \wedge E \in \mathbb{E}^* \Rightarrow e_0 =$ **call** $e(E) \in \mathbb{S}$, where the expression list E denotes the sequence of parameters passed to the function.
- Procedure Call: $e \in \mathbb{E} \wedge E \in \mathbb{E}^* \Rightarrow$ **call** $e(E) \in \mathbb{S}$
- Return: $e \in \mathbb{E} \Rightarrow$ **return** $e \in \mathbb{S}$ and **return** $\in \mathbb{S}$

5.1.5 Program

Definition 5.11 ▶ *C-IL* Program

A *C-IL* program π consists of a set of global variables $\pi.\mathcal{V}_G$, a type table $\pi.T_F$ for struct types, and a function table $\pi.\mathcal{F}$.

$$\mathit{prog}_{\textit{C-IL}} \overset{def}{=} [\mathcal{V}_G \in (\mathbb{V} \times \mathbb{T}_\mathbb{Q})^*, T_F \in \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_\mathbb{Q})^*, \mathcal{F} \in \mathbb{F}_{name} \rightharpoonup fun_{\textit{C-IL}}]$$

- $\pi.\mathcal{V}_G$ – Denotes the sequence of global variable names and their types.
- $\pi.T_F$ – The type table maps names of struct types to a list of field names and the corresponding field types.
- $\pi.\mathcal{F}$ – The function table maps function names to function table entries (Definition 5.12).

Definition 5.12 ▶ Function Table Entry

Function table entries are defined by the type $fun_{\textit{C-IL}}$.

$$fun_{C-IL} \overset{def}{=} [rettype \in \mathbb{T}_\mathbb{Q}, npar \in \mathbb{N}, \mathcal{P} \in \mathbb{S}^* \cup \{\textbf{extern}\}, \mathcal{V} \in (\mathbb{V} \times \mathbb{T}_\mathbb{Q})^*]$$

Function table entries consist of:

- $rettype$ – type of the function's return value,
- $npar$ – number of the function's parameters,
- $\mathcal{P}$ – function body, i.e. list of statements, or **extern**,
- $\mathcal{V}$ – local variables and parameters of the function paired with their types.

In *C-IL* programs assembly functions and compiler intrinsics are declared as external functions. This is expressed by the special value **extern** of the function body in function table entries to assembly functions. The first $npar$ entries in $\mathcal{V}$ denote function parameters.

5.1.6 Configuration

As next step we define the state of *C-IL* programs.

A *C-IL* configuration consists of a byte-addressable global memory $c.\mathcal{M}$ and a stack $c.s$.

◀ **Definition** 5.13 *C-IL*-Configuration

$$C_{C\text{-}IL} \overset{def}{=} [\mathcal{M} \in \mathbb{B}_{gm} \to \mathbb{B}^8, s \in frame^*_{\textsf{C-IL}}]$$

$\mathcal{M}$ denotes the global memory of the program, where

$$\mathbb{B}_{gm} \subset \mathbb{B}^{8 \cdot size_{ptr}}$$

denotes the set of global memory addresses. It stores the global variables.A *C-IL* stack is a list of *C-IL* stack frames (Definition 5.14). Local variables are stored in the stack.

C-IL stack frames are defined by the type $frame_{\textsf{C-IL}}$. It consists of a local memory, a return destination, a function name and a location.

◀ **Definition** 5.14 *C-IL* Stack Frame

$$frame_{\textsf{C-IL}} \overset{def}{=} [\mathcal{M}_{\mathcal{E}} \in \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*, rds \in val_{\textbf{ptr}} \cup val_{\textbf{lref}} \cup \{\bot\}, f \in \mathbb{F}_{name}, loc \in \mathbb{N}]$$

- $\mathcal{M}_{\mathcal{E}}$ – The local memory for local variables and parameters maps variable names to a byte-string representation of the value of the corresponding variable.

- *rds* – The return value destination describes where the return value of a function call has to be stored. $\bot$ denotes the absence of a return value and a return value destination.

- f – The name of the function to which the frame belongs.

- *loc* – The location counter defines the next statement to be executed.

5.1.7 Context

For the execution of *C-IL* program is required information, which is specific to the compiler and the underlying machine. This additional information includes specification of the global data of the program, i.e. addresses, offset, size. We call this environmental information *context* of the *C-IL* program and parametrize *C-IL* executions with it.

Definition 5.15 ▶ C-IL Context

The context of a *C-IL* program is defined by the type $\mathit{context}_{\mathit{C\text{-}IL}}$.

$$\begin{aligned}\mathit{context}_{\mathit{C\text{-}IL}} \overset{def}{=} [&size_{ptr} \in \mathbb{N},\\ &alloc_{gvar} \in \mathbb{V} \rightharpoonup \mathbb{B}^{8\cdot size_{ptr}},\\ &\mathcal{F}_{addr} \in \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{8\cdot size_{ptr}},\\ &size_{struct} \in \mathbb{T}_C \rightharpoonup \mathbb{N},\\ &size\_t \in \mathbb{T}_P,\\ &\mathit{offset} \in \mathbb{T}_C \times \mathbb{F} \rightharpoonup \mathbb{N},\\ &cast \in val \times \mathbb{T}_{\mathbb{Q}} \rightharpoonup val,\\ &endianness \in \{\textbf{little}, \textbf{big}\},\\ &intrinsics \in \mathbb{F}_{name} \rightharpoonup fun_{C-IL},\\ &R_{extern} \in \mathbb{F}_{name} \rightharpoonup 2^{val \times C_{C\text{-}IL} \times C_{C\text{-}IL}}]\end{aligned}$$

- $size_{ptr}$ denotes the size of pointer types in bytes.
- $alloc_{gvar}$ maps global variable names to addresses.
- $\mathcal{F}_{addr}$ maps function names to addresses.
- $size_{struct}$ returns the size of struct names.
- $size\_t$ denotes the type of the value returned by the **sizeof** operator.
- offset returns the byte-offset of a given field in a struct.
- $cast$ does type casting of a given value to a given type.
- $endianess$ denotes the order in which bytes are stored in the memory.
- $intrinsics$ returns a function table entry for compiler intrinsic functions.
- R_{extern} defines the transition relation for external procedures, i.e. functions that are not implemented on the *C-IL* level.

Intrinsics are pre-defined functions provided by the compiler. A function call to an intrinsic does not create a new stack frame but only inlines the intrinsic implementation into the program code [Sch13]. On the *C-IL* level the effect of compiler intrinsics is defined together with the effect of external function calls by R_{extern}. We consider two compiler intrinsics: for read-modify-write operations and VMRUN. The only external procedure, that we define, implements the send-ipi operation. We define R_{extern} later in this chapter, when we are defining the *C-IL* transition function.

The VMRUN intrinsic does not have a return value or parameters. It is replaced by the compiler by the VMRUN $MIPS_P$ instruction.

The function table entry for the VMRUN intrinsic is defined as follows. ◀ **Definition** 5.16 VMRUN FTE

$$\theta.intrinsics(vmrun) \overset{def}{=} [(\emptyset, void), 0, \textbf{extern}, \varepsilon]$$

The read-modify-write intrinsic does not have a return value. It gets four parameters: an address where the value from the memory location defined by rds to be stored, an address of volatile memory location $dest$, a compare value cmp and a value to be written in case of a successful comparison $exchng$.

The function table entry for the read-modify-write intrinsic is defined as follows. ◀ **Definition** 5.17 Read-Modify-Write FTE

$$\begin{aligned}\theta.intrinsics(rmw) \overset{def}{=} [(\emptyset, void), 4, \textbf{extern}, \ & (rds, (\emptyset, \textbf{ptr}(\{\textbf{volatile}\}, \textbf{int}32)))\\ & \circ (dest, (\emptyset, \textbf{ptr}(\emptyset, \textbf{int}32)))\\ & \circ (cmp, (\emptyset, \textbf{int}32))\\ & \circ (exchng, (\emptyset, \textbf{int}32))]\end{aligned}$$

The predicate $rmw_c^{\pi,\theta}$ denotes whether a given expression e is a reference to the read-modify-write intrinsic. ◀ **Definition** 5.18 Read-Modify-Write Predicate

$$\begin{aligned}rmw_c^{\pi,\theta}(e \in \mathbb{E}) \in \mathbb{B} \overset{def}{=} \ & [e]_c^{\pi,\theta} = \textbf{fun}(rmw, t)\\ \vee \ & [e]_c^{\pi,\theta} = \textbf{val}(\theta.\mathcal{F}_{addr}(rmw), t)\end{aligned}$$

where

$$t = \textbf{funptr}(\textbf{void}, \textbf{ptr}(\textbf{int}32) \circ \textbf{ptr}(\textbf{int}32) \circ \textbf{int}32 \circ \textbf{int}32)$$

Having the information from the context it is straightforward to calculate the size of a given *C-IL* type.

The function $size_\theta$ returns the number of bytes required to store a value of the given type. ◀ **Definition** 5.19 Size of Types

$$size_\theta(t \in \mathbb{T}) \in \mathbb{N} = \begin{cases} n/8 & t = \textbf{int}n \vee t = \textbf{uint}n\\ \theta.size_{ptr} & isptr(t) \vee isfunptr(t)\\ n \cdot size_\theta(t') & t = \textbf{array}(t', n)\\ \theta.size_{struct}(t_C) & t = \textbf{struct}\ t_C \end{cases}$$

5.1.8 Memory

We model the memory as a flat byte-addressable memory. In global memory we map addresses to bytes. In the local memory we map variable names to byte-strings. In the same time *C-IL* values are typed bit-strings. In order to interpret memory bytes as *C-IL* values and vice versa we define the following two functions.

Definition 5.20 ▶ Converting Values

The function $val2bytes_\theta$ converts *C-IL* values to byte-strings.

$$val2bytes_\theta :: val \rightharpoonup (\mathbb{B}^8)^*$$

$$val2bytes_\theta(v) \overset{def}{=} \begin{cases} bytes(b) & v = \textbf{val}(b,t) \land \theta.endianness = \textbf{little} \\ rev(bytes(b)) & v = \textbf{val}(b,t) \land \theta.endianness = \textbf{big} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function $bytes2val_\theta$ converts byte-strings to *C-IL* values.

$$bytes2val_\theta :: (\mathbb{B}^8)^* \times \mathbb{T} \rightharpoonup val$$

$$bytes2val_\theta(B,t) \overset{def}{=}$$

$$\begin{cases} \textbf{val}(bits(B),t) & t \neq \textbf{struct}\ t_C \land \theta.endianness = \textbf{little} \\ \textbf{val}(bits(rev(B)),t) & t \neq \textbf{struct}\ t_C \land \theta.endianness = \textbf{big} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The functions $bits$ and $bytes$ convert byte-strings to bits-strings and bits-strings to byte-strings respectively. The function rev reverses the order in the strings. In dependence of the endianness of the underlying architecture we possibly reverse the order of bytes.

Definition 5.21 ▶ Reading from Global Memory

The function $read_\mathcal{M}$ reads a byte-string of length s from a global memory $\mathcal{M}$ starting at address a.

$$read_\mathcal{M} :: (\mathbb{B}_{gm} \to \mathbb{B}^8) \times \mathbb{B}^{8\cdot size_{ptr}} \times \mathbb{N} \to (\mathbb{B}^8)^*$$

$$read_\mathcal{M}(\mathcal{M},a,s) \overset{def}{=} \begin{cases} read_\mathcal{M}(\mathcal{M}, a +_{8\cdot size_{ptr}} 0^{8\cdot size_{ptr}-1}1, s-1) \circ \mathcal{M}(a) & s > 0 \\ \varepsilon & s = 0 \end{cases}$$

Definition 5.22 ▶ Writing to Global Memory

The function $write_\mathcal{M}$ writes a byte-string B to global memory $\mathcal{M}$ starting at address a.

$$write_\mathcal{M} :: (\mathbb{B}_{gm} \to \mathbb{B}^8) \times \mathbb{B}^{8\cdot size_{ptr}} \times (\mathbb{B}^8)^* \to (\mathbb{B}_{gm} \to \mathbb{B}^8)$$

$$write_\mathcal{M}(\mathcal{M},a,B)(x) = \begin{cases} B[\langle x\rangle - \langle a\rangle] & \langle x\rangle - \langle a\rangle \in \{0,\dots,|B|-1\} \\ \mathcal{M}(x) & \text{otherwise} \end{cases}$$

Definition 5.23 ▶ Reading from Local Memory

The function $read_{\mathcal{M}_\mathcal{E}}$ reads a byte-string of length s from local memory $\mathcal{M}_\mathcal{E}$ of variable v starting at offset o.

$$read_{\mathcal{M}_\mathcal{E}} :: (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times \mathbb{N} \rightharpoonup (\mathbb{B}^8)^*$$

$$read_{\mathcal{M}_\mathcal{E}}(\mathcal{M}_\mathcal{E},v,o,s) \overset{def}{=} \mathcal{M}_\mathcal{E}(v)[o+s-1] \circ \ldots \circ \mathcal{M}_\mathcal{E}(v)[o]$$

If $s + o > |\mathcal{M}_\mathcal{E}(v)|$ or $v \notin dom(\mathcal{M}_\mathcal{E})$, the function is undefined for the given parameters.

Definition 5.24 Writing to Local Memory

The function $write_{\mathcal{M}_\mathcal{E}}$ writes byte-string B to variable v of local memory $\mathcal{M}_\mathcal{E}$ starting at offset o.

$$write_{\mathcal{M}_\mathcal{E}} :: (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times (\mathbb{B}^8)^* \rightharpoonup (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*)$$

$$write_{\mathcal{M}_\mathcal{E}}(\mathcal{M}_\mathcal{E}, v, o, B)(w)[i] \overset{def}{=} \begin{cases} B[i-o] & w \neq v \vee i \in \{o, \ldots, o + |B| - 1\} \\ \mathcal{M}_\mathcal{E}(w)[i] & otherwise \end{cases}$$

If $|B| + o > |\mathcal{M}_\mathcal{E}(v)|$ or $v \notin dom(\mathcal{M}_\mathcal{E})$, the function is undefined for the given parameters.

Definition 5.25 Reading from a *C-IL* Configuration

The function $read$ returns for a given *C-IL* configuration c and a given *C-IL* context θ the value from the memory addressed by the pointer x.

$$read_\theta :: C_{C\text{-}IL} \times val \rightharpoonup val$$

$$read_\theta(c, x) \overset{def}{=}$$
$$\begin{cases} bytes2val_\theta(read_\mathcal{M}(c.\mathcal{M}, a, size_\theta(t)), t) & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ bytes2val_\theta(read_{\mathcal{M}_\mathcal{E}}(c.s[i].\mathcal{M}_\mathcal{E}(v), o, size_\theta(t)), t) & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ read_\theta(c, \mathbf{val}(a, \mathbf{ptr}(t))) & x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ read_\theta(c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t))) & x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ undefined & otherwise \end{cases}$$

Definition 5.26 Writing to a *C-IL* Configuration

The function $write$ writes the value y to the memory of a *C-IL* configuration c in a context θ at the address defined by pointer x.

$$write_\theta :: C_{C\text{-}IL} \times val \times val \rightharpoonup C_{C\text{-}IL}$$

$$write_\theta(c, x, y) \overset{def}{=}$$
$$\begin{cases} c[\mathcal{M} := write_\mathcal{M}(c.\mathcal{M}, val2bytes_\theta(x), & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ \qquad\qquad val2bytes_\theta(y))] & \wedge\, y = \mathbf{val}(b, t) \\ c' & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ & \wedge\, y = \mathbf{val}(b, t) \\ write_\theta(c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ write_\theta(c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t)), y) & x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where c' differs from c only in the local memory of the i-th stack frame.

$$c'.s[i].\mathcal{M}_{\mathcal{E}} = write_{\mathcal{M}_{\mathcal{E}}}(c.s[i].\mathcal{M}_{\mathcal{E}}, v, o, val2bytes_{\theta}(y))$$

5.1.9 Auxiliary Definitions

In the following lines we define some auxiliary functions and notations, which we later use for computing the next *C-IL* configuration in the execution of *C-IL* programs.

We index the frames on the stack of a *C-IL* configuration, e.g. the first frame is indexed by zero and the top-most frame is indexed by the length of the stack minus one. In the subsequent definitions we assume a non-empty stack. Only for a non-empty stack the auxiliary functions are total and well-define.

Definition 5.27 ▶ Accessing Frame Elements

To easily talk about components of the frame $i \in [0 : (|c.stack| - 1)]$ of the C-IL configuration $c \in C_{C\text{-}IL}$ we define the following notation:

$$\begin{aligned} c.\mathcal{M}_i &\overset{def}{=} c.s[i].\mathcal{M}_{\mathcal{E}} \\ c.rds_i &\overset{def}{=} c.s[i].rds \\ c.f_i &\overset{def}{=} c.s[i].f \\ c.loc_i &\overset{def}{=} c.s[i].loc. \end{aligned}$$

For the top-most frame on the stack we additionally define:

$$\begin{aligned} c.\mathcal{M}_{top} &\overset{def}{=} c.s[|c.s| - 1].\mathcal{M}_{\mathcal{E}} \\ c.rds_{top} &\overset{def}{=} c.s[|c.s| - 1].rds \\ c.f_{top} &\overset{def}{=} c.s[|c.s| - 1].f \\ c.loc_{top} &\overset{def}{=} c.s[|c.s| - 1].loc. \end{aligned}$$

Definition 5.28 ▶ Topmost Frame Index

The function top returns the index of the top-most stack frame in a given *C-IL* configuration.

$$top(c \in C_{C\text{-}IL}) \in \mathbb{N} \overset{def}{=} |c.s| - 1$$

Definition 5.29 ▶ Removing the Topmost Frame

The function $drop_{frame}$ removes the top-most stack frame from a *C-IL* configuration.

$$drop_{frame}(c \in C_{C\text{-}IL}) \in C_{C\text{-}IL} \overset{def}{=} c[s \mapsto c.s[0 : |c.s| - 2]]$$

The function inc_{loc} increments the location counter of the top-most frame. ◀ **Definition** 5.30 Incrementing Location Counter

$$inc_{loc}(c \in C_{C\text{-}IL}) \in C_{C\text{-}IL} \overset{def}{=} c[loc_{top} \mapsto c.loc_{top} + 1]$$

The function set_{loc} assigns a given value to the location counter of the top most stack frame. ◀ **Definition** 5.31 Setting Location Counter

$$set_{loc}(c \in C_{C\text{-}IL}, l \in \mathbb{N}) \in C_{C\text{-}IL} \overset{def}{=} c[loc_{top} \mapsto l]$$

The function $stmt_{next}$ computes the next statement to be executed. ◀ **Definition** 5.32 Next Statement

$$stmt_{next}(c \in C_{C\text{-}IL}, \pi \in \textsf{prog}_{\textsf{C-IL}}) \in \mathbb{S} \overset{def}{=} \pi.\mathcal{F}(c.f_{top}).\mathcal{P}[c.loc_{top}]$$

The function set_{rds} sets the return destination field of the topmost stack frame to a given value v. ◀ **Definition** 5.33 Setting Return Destination

$$set_{rds}(c \in C_{C\text{-}IL}, v \in val_{\textbf{ptr}} \cup val_{\textbf{lref}} \cup \{\bot\}) \in C_{C\text{-}IL} \overset{def}{=} c[rds_{top} \mapsto v]$$

The function $frame_{new}$ creates a new stack frame and puts it on the top of the stack. ◀ **Definition** 5.34 Creating a New Frame

$$frame_{new}(c \in C_{C\text{-}IL}, \pi \in \textsf{prog}_{\textsf{C-IL}}, f \in \mathbb{F}_{name}) \in C_{C\text{-}IL} \overset{def}{=} c[s \mapsto (\mathcal{M}'_{\mathcal{E}}, \bot, f, 0) \circ c.s]$$

where $\mathcal{M}'_{\mathcal{E}}$ has to offer enough space for the local variables and the parameters of f.
$\mathcal{V}' = \pi.\mathcal{F}(f).\mathcal{V}$
$npar' = \pi.\mathcal{F}(f).npar$

$$\forall i \in [npar' : |\mathcal{V}'| - 1].\ \mathcal{V}'[i] = (v_i, t_i) \Longrightarrow |\mathcal{M}'_{\mathcal{E}}(v_i)| = size_\theta(t_i)$$

$$\forall i \in [0 : npars' - 1].\ \mathcal{M}'_{\mathcal{E}}(v_i) = val2bytes_\theta([E[i]]_c^{\pi,\theta})$$

The predicate *is-function* denotes whether a given function pointer is valid, i.e. the pointer corresponds to a function name. ◀ **Definition** 5.35 Is-function

$$\begin{aligned} is\text{-}function(v \in val_{fptr}, f \in \mathbb{F}_{name}, \theta \in \textsf{context}_{\textsf{C-IL}}) \in \mathbb{B} \overset{def}{=} \\ v = \textbf{val}(b, \textbf{fptr}(t, T)) \wedge \theta.\mathcal{F}_{addr}(f) = b \\ \vee\ v = \textbf{fun}(f, \textbf{fptr}(t, T) \wedge f \in dom(\pi.\mathcal{F})) \end{aligned}$$

5.1.10 Operational semantics

The *C-IL* operational semantics presented in this section define the execution of *C-IL* programs. The transition from one *C-IL* configuration to the next we define with a case distinction on the next statement. Important part of the semantics is the evaluation of expressions. Expressions are evaluated to *C-IL* values in dependence of current *C-IL* configuration $c \in C_{C\text{-}IL}$, the given program $\pi \in \mathit{prog}_{\text{C-IL}}$ and a context $\theta \in \mathit{context}_{\text{C-IL}}$. In expression evaluation the type of the evaluated expression must be determined. The function

$$\tau_c^{\pi,\theta} :: \mathbb{E} \rightharpoonup \mathbb{T}_Q$$

returns to a given expression its type.

The expression evaluation function

$$[\ ]_c^{\pi,\theta} :: \mathbb{E} \rightharpoonup val$$

returns the value of a given expression.

Both functions are defined by case split on the different expressions from Definitions 5.9 and by structural induction on the evaluated expression. The functions are partial and return $\perp$ for expressions which can not be evaluated correctly, e.g. pointer dereferencing $*(e)$ can be evaluated only if e is of pointer type or an array.

Neither the simulation relation between $MIPS_P$ and *C-IL* nor the extensions to the *C-IL* semantics that we introduce in the next chapter require changes in the expression evaluation. Thus we omit here the formal definitions. They can be found in [SS12] and [Sch13].

Depending on the type of the next statement one of the following rules defines the next step in the execution of a *C-IL* program. In the next definitions, when we say location counter, we refer to the location counter of the top most stack frame.

Definition 5.36 ▶ Assignment

The execution of an assignment statement consists of writing the value of the right hand expression at the location defined by the left hand expression and increasing the location counter.

$$\frac{stmt_{next}(c,\pi) = (e_0 = e_1)}{\pi,\theta \vdash c \rightarrow inc_{loc}(write_\theta(c, [\&e_0]_c^{\pi,\theta}, [e_1]_c^{\pi,\theta}))}$$

Definition 5.37 ▶ Goto

The execution of a goto statement consists of updating the value of the current location counter with the provided value.

$$\frac{stmt_{next}(c,\pi) = \textbf{goto } l}{\pi,\theta \vdash c \rightarrow set_{loc}(c,l)}$$

Definition 5.38 ▶ If-Not-Goto (success)

The execution of an if-not-goto statement depends on the evaluation of the condition expression. If the expression e evaluates to zero we set the location counter to the provided value l.

$$\frac{stmt_{next}(c,\pi) = \textsf{\textbf{ifnot}}\ e\ \textsf{\textbf{goto}}\ l \qquad \textsf{\textbf{zero}}_\theta([e]_c^{\pi,\theta})}{\pi,\theta \vdash c \to set_{loc}(c,l)}$$

◀ **Definition** 5.39 If-Not-Goto (failure)

If the expression e does not evaluate to zero we increase the location counter.

$$\frac{stmt_{next}(c,\pi) = \textsf{\textbf{ifnot}}\ e\ \textsf{\textbf{goto}}\ l \qquad \neg\textsf{\textbf{zero}}_\theta([e]_c^{\pi,\theta})}{\pi,\theta \vdash c \to inc_{loc}(c)}$$

◀ **Definition** 5.40 Function Call or Procedure Call

The execution of a function call statement (with or without return value) is defined if the expression e evaluates to a function f, f is not an external function, the local memory of the created frame offers enough space for local variables and parameters, expression E is a valid parameter list according the function declaration. The execution of the statement pushes the new stack frame on the stack and increases the location counter in the old topmost frame.

$$\frac{\begin{array}{c} stmt_{next}(c,\pi) = \textsf{\textbf{call}}\ e(E) \vee stmt_{next}(c,\pi) = (e_0 = \textsf{\textbf{call}}\ e(E)) \\ \textit{is-function}([e]_c^{\pi,\theta}, f) \qquad \theta.\mathcal{F}(f).\mathcal{P} \neq \textsf{\textbf{extern}} \end{array}}{\pi,\theta \vdash c \to frame_{new}(inc_{loc}(set_{rds}(c,rds)),\pi,f)}$$

where rds' denotes the return destination for the function call.

$$rds = \begin{cases} \bot & stmt_{next}(c,\pi) = \textsf{\textbf{call}}\ e(E) \\ [\&e_0]_c^{\pi,\theta} & stmt_{next}(c,\pi) = (e_0 = \textsf{\textbf{call}}\ e(E)) \end{cases}$$

◀ **Definition** 5.41 Function Return with Result

The execution of a return statement with return value and return destination is defined by dropping the top stack frame and writing the return value to the return destination.

$$\frac{stmt_{next}(c,\pi) = \textsf{\textbf{return}}\ e \wedge c.rds_{top(c)-1} \neq \bot}{\pi,\theta \vdash c \to write_\theta(set_{rds}(drop_{frame}(c),\bot), drop_{frame}(c).rds_{top}, [e]_c^{\pi,\theta})}$$

◀ **Definition** 5.42 Function Return without Result

The execution of a return statement without return value or without return destination is defined by dropping the top stack frame.

$$\frac{stmt_{next}(c,\pi) = \textsf{\textbf{return}} \vee (stmt_{next}(c,\pi) = \textsf{\textbf{return}}\ e \wedge c.rds_{top(c)-1} = \bot)}{\pi,\theta \vdash c \to drop_{frame}(c)}$$

◀ **Definition** 5.43 External Procedure Call

The execution of a external procedure call statement is defined if the expression e evaluates to a function f, f is an extern function, the expression E is valid parameter list according the function declaration, the transition relation for extern functions defines the transition from c to c' under the parameters E, in the new configuration c' all stack frames but the topmost one are unchanged, the location counter in topmost frame is increased and the function stays the same.

$$\frac{stmt_{next}(c,\pi) = \textbf{call } e(E) \qquad \textit{is-function}([e]_c^{\pi,\theta}, f) \qquad \theta.\mathcal{F}(f).\mathcal{P} = \textbf{extern} \qquad (([E_0]_c^{\pi,\theta}, \cdots, [E_{|E|-1}]_c^{\pi,\theta}), c, c') \in \theta.R_{extern}(f)}{\pi, \theta \vdash c \to c'}$$

In the next lines we present the transition relation for the external functions that we define.

Definition 5.44 ▶ *rmw* Transition Relation

The definition of the compiler intrinsic read-modify-write has two cases. The difference is that if the compare operation is successful we additionally write the exchange value in the destination memory.

$$\begin{aligned}
&((rds, dest, cmp, exchng), c, c') \in \theta.R_{extern}(rmw) \implies \\
&\qquad read_\theta(c, dest) \neq cmp \wedge c' = inc_{loc}(write_\theta(c, rds, read_\theta(c, dest))) \\
&\quad \vee\, read_\theta(c, dest) = cmp \\
&\qquad \wedge\, c' = inc_{loc}(write_\theta(write_\theta(c, rds, read_\theta(c, dest)), dest, exchng))
\end{aligned}$$

Definition 5.45 ▶ *vmrun* Transition Relation

A function call to the intrinsic *vmrun* appears on the *C-IL* level as a NOOP.

$$(\perp, c, c') \in \theta.R_{extern}(vmrun) \implies c' = inc_{loc}(c)$$

Definition 5.46 ▶ *sendipi* Transition Relation

The external function *sendipi* is implemented in $MIPS_P$ assembler and writes to the APIC interrupt command register. The implementation consists of several instructions and the last is a store word.

The written word $data \in \mathbb{B}^{32}$ is defined according the ICR layout and Software Condition 1 as follows:

$$\begin{aligned}
&data.DEST = 0^8 \\
&data.reserved = 0^4 \\
&data.DSH = 1^2 \\
&data.reserved = 0^5 \\
&data.DS = 1 \\
&data.DM = 0 \\
&data.MT = 0^3 \\
&data.VEC = 0^8.
\end{aligned}$$

The written address $ICR_a = 1^{20}0^9100$ is computed from the APIC base address and the ICR offset.

$$(\bot, c, c') \in \theta.R_{extern}(sendipi) \implies c' = inc_{loc}(write_\theta(c, ICR_a, data))$$

For the moment we consider $sendipi$ as an ordinary memory write. In the next chapter we will extend its semantics and model its effect in the $MIPS_P$ machine.

5.2 CC-IL Semantics

For the execution of several *C-IL* programs in a multiprocessor system we need extended semantics. We define the Concurrent *C-IL* semantics to talk about several *C-IL* threads running at the same time. Every thread has an unique ID $tid \in Tid$ ($Tid \subset \mathbb{N}$).

A concurrent C-IL configuration consists of a single shared global memory and multiple stacks, i.e. one per each thread.

◀ **Definition** 5.47
Concurrent C-IL configuration

$$C_{CC\text{-}IL} \overset{def}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, s \in Tid \mapsto frame^*_{\textsf{C-IL}}]$$

From the configuration $c \in C_{CC\text{-}IL}$ we can construct a sequential configuration $c(t) \in C_{C\text{-}IL}$ for given thread with an ID $t \in Tid$.

$$c(t) \overset{def}{=} (c.\mathcal{M}, c.s(t))$$

A $CC\text{-}IL$ execution is the composition of threads' steps interleaved at statements. The threads execute *C-IL* steps and can access both their local resources (stack) and the shared memory.

◀ **Definition** 5.48
$CC-IL$
Operational Semantics

$$\frac{\begin{array}{l}\pi, \theta \vdash c(t) \to (\mathcal{M}', s') \\ c' = (\mathcal{M}', c.s[t \mapsto s'])\end{array}}{\pi, \theta \vdash c \to c'}$$

5.2.0.1 Auxiliary Functions and Notation

We denote a single $CC\text{-}IL$ step by

$$\pi, \theta \vdash c \to c'.$$

In some case we want to be precise and say that the given $CC\text{-}IL$ step is made by given thread t

$$\pi, \theta \vdash c \to_t c'.$$

A sequence of $CC\text{-}IL$ steps of thread t we denote by

$$\pi, \theta \vdash c \to^*_t c'.$$

Definition 5.49 ▶ Accessing Frame Elements

Since in the concurrent configuration we have several stacks we have to refine our notation for accessing frame elements. To refer to components of the i-th frame of given stack s ($i \in [0 : (|s| - 1)]$) we define:

$$\begin{aligned} s.\mathcal{M}_i &\overset{def}{=} s[i].\mathcal{M}_{\mathcal{E}} \\ s.rds_i &\overset{def}{=} s[i].rds \\ s.f_i &\overset{def}{=} s[i].f \\ s.loc_i &\overset{def}{=} s[i].loc. \end{aligned}$$

For the top-most frame on the stack we additionally define:

$$\begin{aligned} s.\mathcal{M}_{top} &\overset{def}{=} s[|s| - 1].\mathcal{M}_{\mathcal{E}} \\ s.rds_{top} &\overset{def}{=} s[|s| - 1].rds \\ s.f_{top} &\overset{def}{=} s[|s| - 1].f \\ s.loc_{top} &\overset{def}{=} s[|s| - 1].loc. \end{aligned}$$

5.3 Compiler Correctness

In this work we talk about execution of assembler code generated by a compiler for a given *C-IL* program. We want to verify properties on the *C-IL* level and transfer them to the $MIPS_P$ level. Thus we want to define a simulation relation between both. Then we can state that computations of the *C-IL* machine simulate computations of the $MIPS_P$ machine. Having this we can prove properties for the *C-IL* computations which will hold for the $MIPS_P$ computations. The compiler consistency is the simulation relation between the *C-IL* program and the generated instruction sequence.

Compiler consistency for *C-IL* and *MIPS-86* in the absence of interrupts has been examined and a formal definition has been given by Andrey Shadrin [Sha12] building on previous work by W. J. Paul and others [LPP05a, DPS09].

We note that in the context of a hypervisor program, the compiler correctness theorem talks only about the execution of the hypervisor code. $MIPS_P$ processors execute compiled hypervisor code in system mode . In guest mode $MIPS_P$ processors execute guest steps, while the hypervisor *C-IL* machine is not proceeding. If we think of an extended $MIPS_P$ in which guest steps also execute instructions, obviously the processor state will change during guest mode. Thus guest execution would break the consistency between the hypervisor *C-IL* machine and the $MIPS_P$ configuration. The consistency has to be defined in dependence on the processor mode. In guest mode instead of the processor registers one should look up the saved kernel processor state in the memory. We omit the case split in the consistency relation here, since $MIPS_P$ does not provide detailed model for guest executions and the switch between kernel context and guest context. Still we state the compiler correctness theorem excluding guest steps. This way it is applicable not only on $MIPS_P$ but also on a an $MIPS_P$ extended version, which provides more functionality in guest mode.

In this chapter we define compiler consistency close to [Sha12]. In the next chapter we extend the *C-IL* semantics by inter-processor interrupts, extend the compiler correctness theorem for the corresponding interrupt cases, and prove it.

5.3.1 Consistency Points

Non optimizing compilers translate every *C-IL* step into one ore more assembler steps. A step function maps the *C-IL* steps numbers to assembler steps numbers. For non optimizing compilers consistency holds for every *C-IL* step.

For an optimizing compiler consistency holds only at several points of the execution which we call consistency points. The consistency points are defined on both levels *C-IL* and $MIPS_P$. The compiler defines the set of *C-IL* consistency points and the corresponding $MIPS_P$ states. Additionally on the $MIPS$ level we have more consistency points than on the *C-IL* level. So more than one hardware configuration may be consistent with a single software configuration.

Defining the set of consistency points is an art itself and depends strongly on the particular compiler and consistency relation. The more consistency points a given compiler guarantees the less optimizations on the generated code it can do. In this work we define a set of consistency points, which is sufficient for our proofs. It satisfies the conditions of our reordering theorem and allows us to prove compiler correctness theorem inductively.

5.3.1.1 Software Consistency Points

The *C-IL* states where consistency holds we call software consistency points. A *C-IL* state is a consistency point if the next statement is:

- a function call[1],
- the first statement in a given function,
- a return statement,
- a statement performing a shared memory access,
- the first statement after a function call,
- the first statement after a statement performing a shared memory access.

[1]The compiler sets a consistency point before/after function calls to *C-IL* functions, external functions and compiler intrinsics.

Definition 5.50 ▶ C-IL Consistency Locations

Given a program π and a context θ the predicate cp denotes whether a given location loc in the execution of function f is a consistency point.

$$\begin{aligned} cp^{\pi,\theta}(f \in \mathbb{F}_{name}, loc \in \mathbb{N}) \in \mathbb{B} \overset{def}{=} & (loc = 0) \\ & \vee \pi.\mathcal{F}(f).\mathcal{P}[loc] \in \{\textbf{return}, \textbf{return } e\} \\ & \vee \pi.\mathcal{F}(f).\mathcal{P}[loc] \in \{e_0 = \textbf{call } e(E), \textbf{call } e(E)\} \\ & \vee \pi.\mathcal{F}(f).\mathcal{P}[loc-1] \in \{e_0 = \textbf{call } e(E), \textbf{call } e(E)\} \\ & \vee vol_c^{\pi,\theta}(\pi.\mathcal{F}(f).\mathcal{P}[loc-1]) \end{aligned}$$

Definition 5.51 ▶ C-IL Consistency Point

The predicate $CP^{\pi,\theta}_{\textsf{C-IL}}$ denotes whether given configuration c is a consistency point.

$$CP^{\pi,\theta}_{\textsf{C-IL}}(c \in C_{C\text{-}IL}) \in \mathbb{B} \overset{def}{=} cp^{\pi,\theta}(c.f_{top}, c.loc_{top})$$

If we consider a function call to an external function and relate it to the set of *C-IL* consistency point, we can conclude that on the *C-IL* level function calls to external functions are bordered by consistency points and there is no other consistency point in-between.

Thus following software condition arises.

Software Condition 8 (Consistent External Functions) *All external functions preserve compiler consistency. External functions do not access shared data.*

5.3.1.2 $MIPS_P$ Consistency Points

We define the $MIPS_P$ consistency points to overlap with the interleaving points defined in Chapter 3. We recall that we distinguish among different types of interleaving points.

In Chapter 3 we already mentioned the set of instruction addresses $\mathbb{A}_{cp}$. It contains pointers to the first instruction to be executed after a consistency point that originates from the compiled program. In the absence of interrupts, all hypervisor interleaving points are defined by the compiler by $\mathbb{A}_{cp}$.

Definition 5.52 ▶ Hypervisor Consistency Point of Processor i

The predicate $hypCP_{\textsf{C-IL}}$ denotes whether the k-th step in the execution $h \xrightarrow{\beta} h'$ is a hypervisor consistency point for processor i.

$$\begin{aligned} & hypCP_{\textsf{C-IL}}(h \in C_M, i \in Pid, \beta \in (\Sigma_M)^*, k \in \mathbb{N}) \in \mathbb{B} \overset{def}{=} \\ & \qquad k \in [0 : |\beta| - 1] \wedge i = \beta_k.pid \wedge core(\beta_k) \wedge h^k.pc_{\beta_k.pid} \in \mathbb{A}_{cp} \end{aligned}$$

We recall that according the definitions in Chapter 3 each interleaving block in a reordered $MIPS_P$ execution starts and ends at an interleaving point. Hypervisor blocks, which we consider in our simulation theorem, start at hypervisor interleaving point but may end at any kind of an interleaving point. The next hypervisor block of the same processor starts then at another hypervisor interleaving point. This implies that consistency holds at the beginning and at end of a hypervisor block. Thus we also include the guest and IPI interleaving points in the set of consistency points.

The predicate CP_{MIPS_P} denotes whether the k-th step in the execution $h \xrightarrow{\beta} h'$ is a consistency point.

◀ **Definition** 5.53 $MIPS_P$ Consistency Point

$$CP_{MIPS_P}(h \in C_M, \beta \in (\Sigma_M)^*, k \in \mathbb{N}) \in \mathbb{B} \overset{def}{=} \\ k \in [0 : |\beta| - 1] \wedge \exists i \in Pid.\ hypCP_{\textsf{C-IL}}(h, i, \beta, k) \vee guestIP(h, \beta, k) \vee ipi(\beta_{k-1})$$

Despite the fact that each interleaving block in a reordered $MIPS_P$ execution is bordered by interleaving points, still it might be that in the considered execution some threads do not end in a consistent state. Therefore we have to distinguish two cases for given consistency point h. Threads that are running, i.e. processing steps, after h have been interleaved properly at their own last consistency point. Threads that are not running after h are possibly interrupted before reaching their own consistency point.

The predicate $running$ denotes that a given processor i will perform further steps after k in the given transition sequence.

◀ **Definition** 5.54 Running Thread

$$running_i(\beta, k) \overset{def}{=} \exists l \in [k : |\beta| - 1] : \beta_l.pid = i$$

Our compiler correctness theorem is stated inductively on interleaving blocks between two consistency points. Thus we need to identify the next consistency point in a execution.

The function $nextCP$ returns the index of the next consistency point in the execution sequence $h \xrightarrow{\beta} h'$ starting from configuration h^k.

◀ **Definition** 5.55 Next Consistency Point

$$nextCP(h \in C_M, \beta \in (\Sigma_M)^*, k \in \mathbb{N}) \in \mathbb{N} \cup \perp \overset{def}{=} \\ \begin{cases} m & CP_{MIPS_P}(h, \beta, m) \wedge \forall l \in [k+1, m-1] : \neg CP_{MIPS_P}(h, \beta, l) \\ \perp & \text{otherwise.} \end{cases}$$

5.3.2 Compiler Information

In order to formalize the compiler correctness theorem we need to examine some details of the compiler and the code generation function. We need to know where local variables and function parameters are saved on the stack. A compiler calling convention defines how functions receive their parameters and return a result. Furthermore the compiler assigns to some GPRs special functions, e.g. stack

Index	Alias	Usage
0	$zero$	always zero
1 ... 13	$t_0 \dots t_{12}$	temporary
14	rv	return value
15 ... 18	$i_0 \dots i_3$	input arguments
19 ... 28	$t_{13} \dots t_{22}$	temporary
29	rsp	stack pointer
30	rbp	stack frame base pointer
31	t_{23}	temporary

Table 5.1: Special Usage of GPRs

pointer. We stick to the calling convention from [Sha12]. In Table 5.1 we list registers alongside their function and alias.

Furthermore, we introduce two disjoint lists of register indexes $caller$ and $callee$. The exact separation of registers into these two lists is for us an irrelevant detail. The important fact is, that together $caller$ and $callee$ contain all registers except $zero, rsp, rbp$.

Definition 5.56 ▶ Calling Convention

We define the following calling convention:

- All GPR registers from $caller$ are caller saved.If the caller relies on their values it has to save them, because the callee may change them.
- All GPR registers from $callee$ are callee saved.The callee guarantees to the caller that after return this registers have the same values as before the call.
- A Function's result is always passed through register rv.
- The first four parameters of a given function are passed to it by its caller in registers $i_0, \dots, i_3$.
- If a given function has more than four parameters, then all parameters except the first four are pushed by the caller on the stack.
- The stack pointer rsp points to the first byte of the last occupied word of the topmost stack frame.
- The stack frame base pointer rbp points to the first byte of the previous base pointer field of the topmost stack frame.

In Figure 5.1 we present the stack layout. For every called function a function frame is allocated on the stack, which grows downwards. Not only the number of frames on the stack but also the frame's layout changes dynamically on function calls.

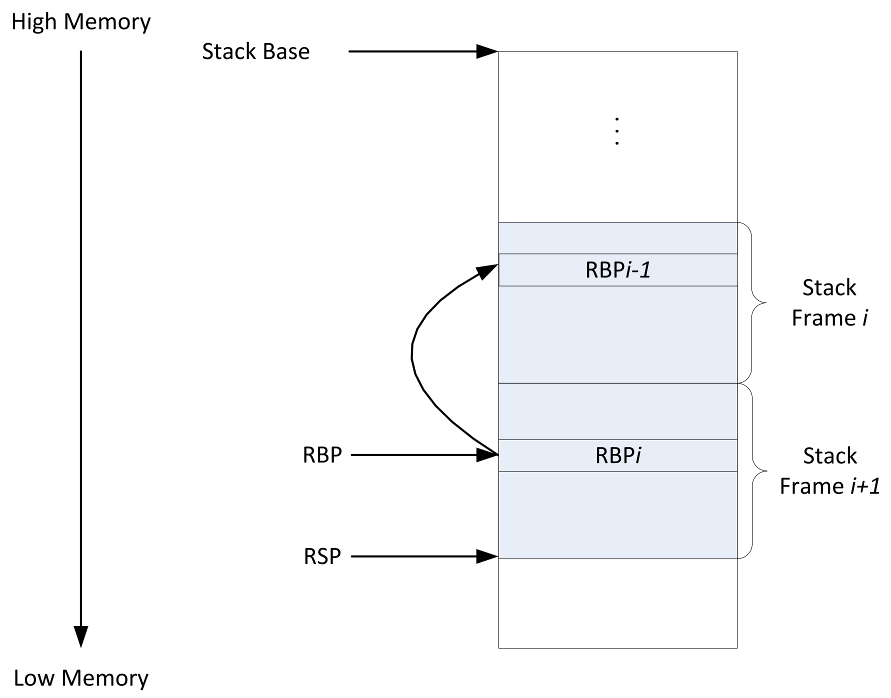

Figure 5.1: C-IL stack layout. The stack is growing downwards.

All frames offer place for function arguments, the return address, the base pointer of the previous frame, callee save registers, local variables and temporaries(Figure 5.2). Additionally some frames contain caller save registers. These frames belong to functions which are executing a function call statement and are waiting for a callee to return. For simplicity we assume that always all callee save registers are stored on the stack. The region allocated for function parameters offers space for all parameters, despite the fact that the first four parameters are passed in registers.

The allocation and the deallocation of stack space is done both on the caller and on the callee side. This is done on the one hand by *pre-call-code* and *post-call-code* generated for function calls on the caller side (5.3). On the other hand the compiler generates for every function f a *prolog* code and an *epilog* code. The *prolog* and the *epilog* are executed by the callee. First the *pre-call-code* pushes caller save registers on the stack, allocates the parameter region, stores parameters (if more then four) and pushes the return address on the stack. Then *prolog* allocates further stack space and sets up the frame for the function execution. At the end of the function the *epilog* deallocates the frame of the function, restores the base pointer and the program counter. At that point the caller save registers are still on the stack and the *post-call-code* restores them and deallocates the corresponding stack region.

Definition 5.57 ▶ Static Compiler Information $CC\text{-}IL$

We define the record $info_{CC\text{-}IL}$ to consist of the static information about the compiled program.

$$
\begin{aligned}
info_{CC\text{-}IL} \overset{def}{=} [&code \in (\mathbb{B}^{32})^*,\\
&cba \in \mathbb{B}^{8\cdot size_{ptr}},\\
&csize \in \mathbb{N},\\
&stmtcode \in \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}^{8\cdot size_{ptr}},\\
&pcode \in \mathbb{F}_{name} \to \mathbb{B}^{8\cdot size_{ptr}},\\
&pcode_{size} \in \mathbb{F}_{name} \to \mathbb{N},\\
&ecode \in \mathbb{F}_{name} \to \mathbb{B}^{8\cdot size_{ptr}},\\
&ecode_{size} \in \mathbb{F}_{name} \to \mathbb{N},\\
&pccode \in \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}^{8\cdot size_{ptr}},\\
&stack_{ba} \in Tid \to \mathbb{B}^{8\cdot size_{ptr}},\\
&stack_{size} \in \mathbb{N},\\
&fsize \in \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{N},\\
&fsize_t \in \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{N},\\
&fsize_p \in \mathbb{F}_{name} \to \mathbb{N},\\
&fsize_v \in \mathbb{F}_{name} \to \mathbb{N},\\
&lvar_{num} \in \mathbb{F}_{name} \to \mathbb{N},\\
&lvar_{reg} \in \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}^5 \cup \bot,\\
&lvar_{off} \in \mathbb{V} \times \mathbb{F}_{name} \to \mathbb{N},\\
&par_{off} \in \mathbb{V} \times \mathbb{F}_{name} \to \mathbb{N},\\
&reg_{off} \in \mathbb{F}_{name} \times \mathbb{N} \times \mathbb{B}^5 \to \mathbb{N},\\
&cp \in \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}]
\end{aligned}
$$

- $info.code$ the compiled code.
- $info.cba$ code region base address.
- $info.csize$ code size in bytes.
- $info.stmtcode$ maps a function name and a location to the address of the first instruction of the compiled statement.
- $info.pcode$ maps a function name to the address of the first instruction of the function's prolog.

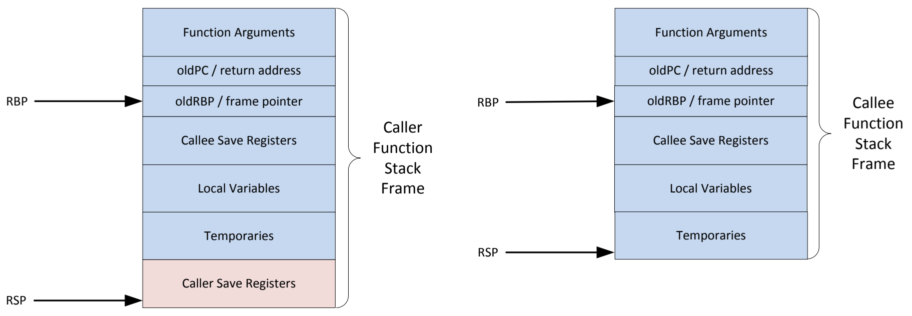

Figure 5.2: C-IL stack frame layout. The stack is growing downwards.

- $info.pcode_{size}$ maps a function name to the size in bytes of the function's prolog.
- $info.ecode$ maps a function name to the address of the first instruction of the function's epilog.
- $info.ecode_{size}$ maps a function name to the size in bytes of the function's epilog.
- $info.pccode$ maps a function name and a location of a function call in it to the address of the first instruction of the post-call-code of the function call.
- $info.stack_{ba}$ returns the stack base address for a given thread.
- $info.stack_{size}$ maximum stack size in bytes.
- $info.fsize$ maps a function and a location to a number of bytes allocated on the stack for the function's frame.
- $info.fsize_t$ maps a function and a location to the number of bytes allocated on the stack for temporaries.
- $info.fsize_p$ maps a function to a number of bytes allocated on the stack for its parameters.
- $info.fsize_v$ maps a function to the number of bytes allocated on the stack for its local variables.
- $info.lvar_{num}$ maps a given function to the number of its local variables.
- $info.lvar_{reg}$ maps the local variables of a given function and a location to the register index where they are stored.

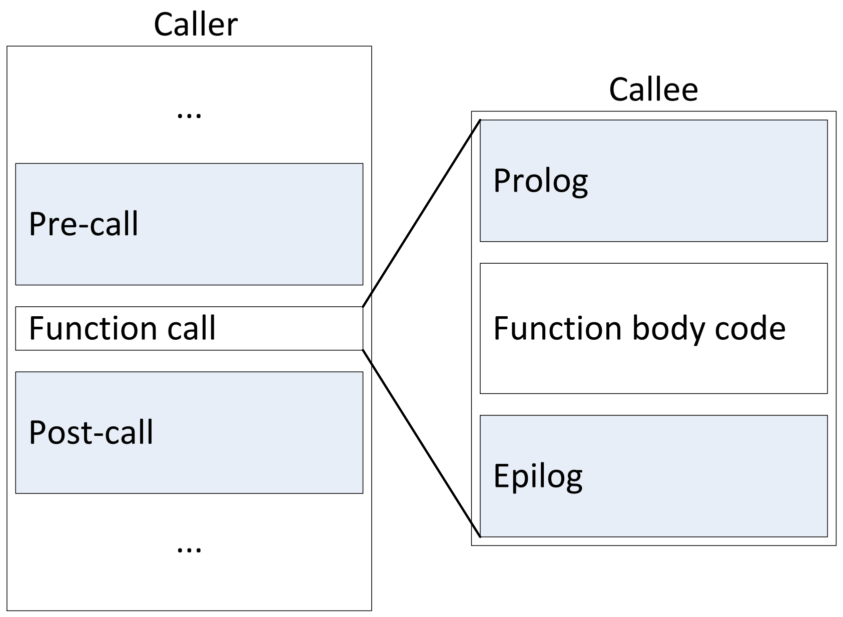

Figure 5.3: Function call code.

- $info.lvar_{\mathit{off}}$ maps the local variables(excluding parameters) of a given function to their offset in bytes within the corresponding stack frame (below the frame base address).
- $info.par_{\mathit{off}}$ maps the parameters of a given function to their offset in bytes within the corresponding stack frame (above the frame base address).
- $info.reg_{\mathit{off}}$ maps a function name, a location and an index of a callee save or a caller save register to an offset in bytes according the frame base pointer. The callee save registers, which contain local variables of f_i are saved in the frame $i+1$, thus the offset defined by $info.reg_{\mathit{off}}$ for callee save registers reaches the next frame.
- $info.cp$ denotes the consistency points for a given function and a program location.

Definition 5.58 ▶ Static Compiler Information *C-IL*

Since in all the cases we consider only the stack of a single thread we define $info_{\text{C-IL}}$ a single-thread version of $info_{CC\text{-}IL}$ with the only difference that it stores a single stack base address.

Definition 5.59 ▶ Static Compiler Information Transformation

The function $sinfo$ returns for a given thread and a given concurrent static information the corresponding single-thread static information.

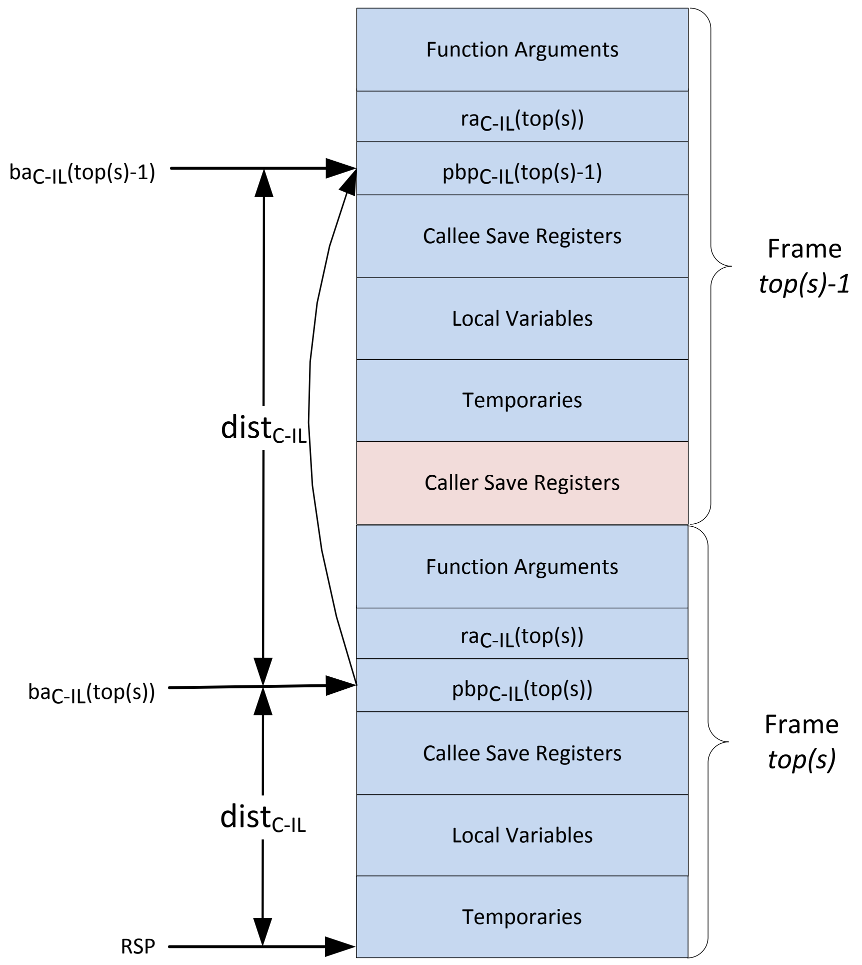

Figure 5.4: C-IL stack frame distance.

$$sinfo :: Tid \times info_{CC\text{-}IL} \rightarrow info_{\text{C-IL}}$$

$$sinfo(t, info).xxx = \begin{cases} info.xxx & if\ \ xxx \neq stack_{ba} \\ info.stack_{ba}(t) & otherwise \end{cases}$$

◀ **Definition** 5.60 Base Pointer Distance

The function $dist_{\text{C-IL}}$ returns the number of bytes between the base pointers of frames on the stack s (Figure 5.4). For the topmost frame it returns the number of bytes between the base pointer and the stack pointer.

$$dist_{\textsf{C-IL}}(i \in \mathbb{N}, s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}) \in \mathbb{N} \overset{def}{=} \begin{cases} info.fsize_v(s.f_i) + 4 * |callee| + info.fsize_t(s.f_i, s.loc_i) & if\ i = top(s) \\ info.fsize(s.f_i, s.loc_i) - info.fsize_p(s.f_i) + info.fsize_p(s.f_{i+1}) & otherwise \end{cases}$$

Definition 5.61 ▶ Base Address

The function $ba_{\textsf{C-IL}}$ returns the base address of the i-th frame on the stack s.

$$ba_{\textsf{C-IL}}(i \in \mathbb{N}, s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}) \in \mathbb{B}^{32} \overset{def}{=} \begin{cases} info.stack_{ba} - bin_{32}(info.fsize_p(s.f_i) - 4) & if\ i = 0 \\ ba_{\textsf{C-IL}}(0, s, info) - bin_{32}(\sum^{j<i} dist_{\textsf{C-IL}}(j, s, info)) & otherwise \end{cases}$$

Definition 5.62 ▶ Return Address

The function $ra_{\textsf{C-IL}}$ returns the return address of the i-th frame on the stack s.

$$ra_{\textsf{C-IL}}(i \in \mathbb{N}, s \in frame^*_{\textsf{C-IL}}, m \in \mathbb{B}^{32} \to \mathbb{B}^8, info \in info_{\textsf{C-IL}}) \in \mathbb{B}^{32} \overset{def}{=} m_4(ba_{\textsf{C-IL}}(i, s, info) + 4)$$

Definition 5.63 ▶ Previous Base Pointer

The function $pbp_{\textsf{C-IL}}$ returns for given frame i on the stack s the base address of the previous frame.

$$pbp_{\textsf{C-IL}}(i \in \mathbb{N}, s \in frame^*_{\textsf{C-IL}}, m \in \mathbb{B}^{32} \to \mathbb{B}^8, info \in info_{\textsf{C-IL}}) \in \mathbb{B}^{32} \overset{def}{=} m_4(ba_{\textsf{C-IL}}(i, s, info))$$

5.3.2.1 Memory Layout

Definition 5.64 ▶ Code Region

With the compiler information we can now define the code region in the memory.

$$CR(info \in info_{CC\text{-}IL}) \in 2^{\mathbb{B}^{32}} \overset{def}{=} [info.cba : info.cba + bin_{32}(info.csize - 1)]$$

Definition 5.65 ▶ Stack Region

The stack region of thread i is defined by the thread's stack base address and the maximum stack size.

$$StR(i \in Tid, info \in info_{CC\text{-}IL}) \in 2^{\mathbb{B}^{32}} \overset{def}{=} [info.stack_{ba}(i) - bin_{32}(info.stack_{size} + 1) : info.stack_{ba}(i)]$$

Definition 5.66 ▶ Code Region Address

The predicate cr denotes whether given address a belongs to the region, where the code resides.

$$cr(a \in \mathbb{B}^{8 \cdot size_{ptr}}, info \in info_{CC\text{-}IL}) \in \mathbb{B} \overset{def}{=} a \in CR(info)$$

The predicate sr denotes whether given address a belongs to the local stack memory of some thread.

◀ **Definition** 5.67
Stack Region

$$sr(a \in \mathbb{B}^{8 \cdot size_{ptr}}, info \in info_{CC\text{-}IL}) \in \mathbb{B} \overset{def}{=} \exists i \in Tid.\ a \in StR(i, info)$$

5.3.3 Compiler Consistency

The compiler consistency relation comprises the coherency between the sub-components of given *C-IL* and $MIPS_P$ configurations, the *C-IL* program and the compiled code.

- Control consistency defines the values of the program counter on the one hand and the return address of the function frames on the other hand.
- Code consistency states that the compiled program is placed in a memory region disjoint from the data, and that the compiled program corresponds to the original code.
- Stack consistency talks about the local memory region where the stack resides, the stack and base pointer and registers that store local variables. All stack frames are properly stored and follow the calling convention.
- Memory consistency talks about the shared global memory of the program, e.g. the global variables.

Defining compiler consistency we distinguish global and local properties.

- The global consistency consists of the memory consistency and the code consistency.
- The local consistency consists of the control consistency and the stack consistency.

The **code consistency** states that the program is stored at the proper memory region and is disjoint from the data. In the absence of self modifying code this is a read only region of the memory. For simplicity we omit here the formal definition and only declare the function $consis^{code}_{CC\text{-}IL}$, which denotes code consistency.

$$consis^{code}_{CC\text{-}IL}(info \in info_{CC\text{-}IL}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B}$$

The memory consistency defines the equivalence of the global data memory in both machines. We denote it with the predicate:

◀ **Definition** 5.68
Memory
Consistency

$$consis^{mem}_{CC\text{-}IL}(info \in info_{CC\text{-}IL}, \mathcal{M} \in \mathbb{B}_{gm} \to \mathbb{B}^8, h \in C_M) \in \mathbb{B} \overset{def}{=}$$
$$\forall a \in \mathbb{B}^{8 \cdot size_{ptr}}.\ \neg(sr(a, info) \vee cr(a, info) \vee a \in \mathbb{A}_{ipcb}) \implies \mathcal{M}(a) = h.m(a)$$

Definition 5.69 ▶ Global Consistency

The predicate $consis^{global}_{CC\text{-}IL}$ denotes the global consistency.

$$consis^{global}_{CC\text{-}IL}(c \in C_{CC\text{-}IL}, info \in info_{CC\text{-}IL}, h \in C_M) \in \mathbb{B} \overset{def}{=} consis^{code}_{CC\text{-}IL}(info, h.m) \wedge consis^{mem}_{CC\text{-}IL}(info, c.\mathcal{M}, h)$$

Definition 5.70 ▶ Control Consistency

The predicate $consis^{control}_{\textsf{C-IL}}$ denotes control consistency. It states that the value of the program counter is the address of the current *C-IL* instruction. The value of the return address in all stack frames is the address of the first post-call-code instruction after the function call to the topmost function.

$$\begin{aligned}&consis^{control}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}, core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \overset{def}{=}\\ &\quad core.pc = info.stmtcode(s.f_{top}, s.loc_{top})\\ &\quad \wedge \forall i \in \mathbb{N}.0 < i < |s| \implies\\ &\qquad m_4(ra_{\textsf{C-IL}}(i, s, m, info)) = info.pccode(s.f_{i-1}, s.loc_{i-1})\end{aligned}$$

Definition 5.71 ▶ Register Consistency

The predicate $consis^{regs}_{\textsf{C-IL}}$ denotes register consistency. It states that the value of the stack pointer register rsp refers to the top of the stack and the value of the frame base pointer register rbp refers to the base address of the topmost frame.

$$\begin{aligned}&consis^{regs}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}, core \in C_{CORE}) \equiv\\ &\quad core.gpr(rbp) = ba_{\textsf{C-IL}}(top(s), s, info)\\ &\quad \wedge core.gpr(rsp) = ba_{\textsf{C-IL}}(top(s), s, info) - bin_{32}(dist_{\textsf{C-IL}}(top(s), s, info))\end{aligned}$$

Definition 5.72 ▶ Previous Base Pointer Consistency

The predicate $consis^{pbp}_{\textsf{C-IL}}$ denotes previous base pointer consistency. It states that the first four bytes of every stack frame contain the base address of the preceding stack frame.

$$\begin{aligned}&consis^{pbp}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}\\ &\quad \forall i \in \mathbb{N}.\ 0 < i < |s| \implies pbp_{\textsf{C-IL}}(i, s, m, info) = ba_{\textsf{C-IL}}(i-1, s, info)\end{aligned}$$

Definition 5.73 ▶ Local Variables Consistency

The predicate $consis^{var}_{\textsf{C-IL}}$ denotes local variables consistency. It states that the local variables and parameters are stored properly. The variables that are kept in registers according the compiler information are stored in the registers only for the topmost frame. For the other frames they are on the stack either in the callee save or in the caller save regions of the corresponding frame.

$$consis^{var}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, \pi \in \textsf{prog}_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}, core \in C_{CORE},$$
$$m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}$$
$$\forall i, j \in \mathbb{N}.\ i < |s| \wedge j < info.lvar_{num}(s.f_i) \implies s.\mathcal{M}_i(v_j) =$$
$$\begin{cases} core.gpr(reg) & if\ reg \neq \bot \wedge i = top(s) \\ m_4(ba_i - bin_{32}(info.reg_{off}(f, reg))) & if\ reg \neq \bot \wedge i < top(s) \\ m_{size_\theta(qt2t(t_j))}(ba_i + bin_{32}(4 + info.par_{off}(v_j, s.f_i))) & if\ reg = \bot \wedge j < npar \\ m_{size_\theta(qt2t(t_j))}(ba_i - bin_{32}(info.lvar_{off}(v_j, s.f_i))) & if\ reg = \bot \wedge j \geq npar \end{cases}$$

where

$$(v_j, t_j) = (\pi.\mathcal{F}(s.f_i)).\mathcal{V}[j]$$
$$reg = info.lvar_{reg}(v_j, s.f_i, s.loc_i)$$
$$npar = (\pi.\mathcal{F}(s.f_i)).npar$$
$$ba_i = ba_{\textsf{C-IL}}(i, s, info)$$

◀ **Definition** 5.74
Stack Consistency

The predicate $consis^{stack}_{\textsf{C-IL}}$ denotes stack consistency as a wrapper for register, previous base pointer and local variables consistency.

$$consis^{stack}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}},$$
$$core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}$$
$$consis^{regs}_{\textsf{C-IL}}(s, info, core)$$
$$\wedge\ consis^{pbp}_{\textsf{C-IL}}(s, info, m)$$
$$\wedge\ consis^{var}_{\textsf{C-IL}}(s, info, core, m)$$

◀ **Definition** 5.75
Local Consistency

The predicate $consis^{local}_{\textsf{C-IL}}$ denotes local compiler consistency.

$$consis^{local}_{\textsf{C-IL}}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}},$$
$$core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}$$
$$consis^{control}_{\textsf{C-IL}}(s, info, core, m)$$
$$\wedge\ consis^{stack}_{\textsf{C-IL}}(s, info, core, m)$$

◀ **Definition** 5.76
Compiler
Consistency

The predicate $consis_{CC\text{-}IL}$ denotes compiler consistency.

$$consis_{CC\text{-}IL}(c \in C_{CC\text{-}IL}, info \in info_{CC\text{-}IL}, h \in C_M, i \in Pid) \in \mathbb{B} \overset{def}{=}$$
$$consis^{global}_{CC\text{-}IL}(c, info, h)$$
$$\wedge\ consis^{local}_{\textsf{C-IL}}(c(i).s, info(i), h.core_i, h.m)$$

5.3.4 Safety

Next, we define a *C-IL* program discipline that allows us to prove simulation between the program and its $MIPS_P$ execution. Programs that follow this discipline we call ownership safe. For a safe *C-IL* program the compiler must guarantee safety of the $MIPS_P$ execution.

In our work we assume that the considered *C-IL* programs have the same number of threads as the number of processors in the underlying $MIPS_P$, i.e. $Tid = Pid$. Defining ownership for threads we also recall that the layout of the *C-IL* memory is the same as the layout of the $MIPS_P$ memory, i.e. both memories are flat and byte addressable. Basically we define the ownership for *C-IL* threads equivalent to the ownership from Section 3.2. We define *C-IL* ownership-safety following the same policy but for a subset of the complete $MIPS_P$ address space $\mathbb{A}_{MIPS_P}$.

We recall that the set of read only addresses for $MIPS_P$ contains the code region in addition to $\mathbb{A}^{ro}{}_{\textit{C-IL}}$. The code region is not part of the *C-IL* memory.

Definition 5.77 ▶ *C-IL* Read Only Memory

Thus the set of *C-IL* read only addresses is defined by the set of constant variables.

$$\mathbb{A}^{ro}{}_{\textit{C-IL}} = \{a \in \mathbb{B}^{32} \mid \exists (v,(q,t)) \in \pi.\mathcal{V}_G.\ q \in \textbf{const} \wedge a \in A_v\}$$

where A_v denotes the set of addresses occupied by the variable v

$$A_v = [\theta.alloc_{gvar}(v) : \theta.alloc_{gvar}(v) + bin_{32}(size_\theta(t) - 1)]$$

Furthermore the set of owned addresses for *C-IL* contains only global memory addresses, since local variables are thread-local by default. We also have to exclude the IPCB region from the *C-IL* ownership. That way we guarantee that the IPCB addresses are constantly assigned to the corresponding processor. In the same time we exclude accesses to IPCBs on the *C-IL* level.

Definition 5.78 ▶ *C-IL* Ownership Address Space

According to these conditions in the ownership address space for *C-IL* we exclude from $\mathbb{A}_{MIPS_P}$ the IPCB region, the code region and the stack region as defined in Section 5.3.2.1.

$$\mathbb{A}_{\textit{C-IL}} \overset{def}{=} \mathbb{A}_{MIPS} \setminus (\mathbb{A}_{code} \cup \mathbb{A}_{stack} \cup \mathbb{A}_{ipcb})$$

Alongside the definitions of the static part of the ownership setting we require in our simulation theorem that the dynamic part of the ownership settings on the *C-IL* and on the $MIPS_P$ levels satisfy the following invariant.

Definition 5.79 ▶ Ownership Consistency

Given *C-IL* ownership o_c and $MIPS_P$ ownership o_m the predicate $consis^{\mathcal{O}}$ states that:

- processor i owns the same addresses as thread i plus the corresponding stack and IPCB regions,

- and the sets of shared addresses for *C-IL* and $MIPS_P$ are equal.

$$\begin{aligned}
&consis^{\mathcal{O}}(o_c \in \mathcal{O}, o_m \in \mathcal{O}) \in \mathbb{B} \overset{def}{=}\\
&\qquad \wedge\ ownership\text{-}inv(o_c)\\
&\qquad \wedge\ ownership\text{-}inv(o_m)\\
&\qquad \wedge\ o_c.\mathbb{A}^{sh} = o_m.\mathbb{A}^{sh}\\
&\qquad \wedge\ \forall i \in Tid.\ (o_m.O_i \setminus o_c.O_i = StR(i, info) \cup \mathbb{A}_{ipcb_i})
\end{aligned}$$

In order to complete the ownership setting for *C-IL*, we have to define the set of *C-IL* I/O steps, e.g. steps which are allowed to access shared data.

5.3.4.1 I/O Steps

We consider the following steps I/O steps:

- steps with accesses to volatile variables,
- the read-modify-write step,
- and the send-ipi step.

We define the following predicates to detect volatile accesses.

The predicate $evol_c^{\pi,\theta}$ denotes whether a given expression e contains volatile accesses.

◀ **Definition** 5.80 Volatile Expression

$$evol_c^{\pi,\theta}(e \in \mathbb{E}) \in \mathbb{B} \overset{def}{=}$$

$$\begin{cases}
\textbf{volatile} \in q & if\ \ e \in \mathbb{V} \wedge \tau_c^{\pi,\theta}(e) = (q,t)\\
evol_c^{\pi,\theta}(e_0) & if\ \ (\exists \oplus \in \mathbb{O}_1.\ e = \oplus e_0)\\
& \quad \vee e = (t)e_0 \vee e = \textbf{sizeof}(e_0)\\
& \quad \vee e = \&(*(e_0))\\
evol_c^{\pi,\theta}(e_0) \vee evol_c^{\pi,\theta}(e_1) & if\ \ \exists \oplus \in \mathbb{O}_2.\ e = e_0 \oplus e_1\\
evol_c^{\pi,\theta}(e_0) \vee evol_c^{\pi,\theta}(e_1) \vee evol_c^{\pi,\theta}(e_2) & if\ \ e = (e_0\ ?\ e_1 : e_2)\\
evol_c^{\pi,\theta}(e_0) \vee \textbf{volatile} \in q & if\ \ e = *(e_0) \wedge \tau_c^{\pi,\theta}(e_0) = (q', \textbf{ptr}(q,t))\\
& \quad \vee e = (e_0).f \wedge \tau_c^{\pi,\theta}(e) = (q,t)\\
0 & otherwise
\end{cases}$$

Definition 5.81 ▶ Volatile Statement

The predicate $vol_c^{\pi,\theta}$ denotes whether a given statement s contains volatile accesses.

$$vol_c^{\pi,\theta}(s \in \mathbb{S}) \in \mathbb{B} \overset{def}{=}$$

$$\begin{cases} evol_c^{\pi,\theta}(e_0) \vee evol_c^{\pi,\theta}(e_1) & \textit{if } s = (e_0 = e_1) \\ evol_c^{\pi,\theta}(e) & \textit{if } s \in \{\textbf{ifnot } e \textbf{ goto } l, \textbf{return } e\} \\ evol_c^{\pi,\theta}(e_0) \vee evol_c^{\pi,\theta}(e) \vee \exists e' \in E.\ evol_c^{\pi,\theta}(e') & \textit{if } s = (e_0 = \textbf{call } e(E)) \\ evol_c^{\pi,\theta}(e) \vee \exists e' \in E.\ evol_c^{\pi,\theta}(e') & \textit{if } s = \textbf{call } e(E) \\ 0 & \textit{otherwise} \end{cases}$$

In order to keep subsequent definitions concise we limit the accesses to volatile data to the following cases:

- Volatile variables may only be accessed in assignment statements or by the intrinsic function rmw.
- Per assignment we allow only one access to a volatile variable, i.e. either a volatile read on the right hand side or a volatile write on the left hand side.
- In case of a volatile read, the right hand side of assignments is either a volatile variable identifier, or it is dereferencing a pointer expression which is either volatile or pointing to a volatile variable.
- In case of a volatile write, the left hand side of assignments is either a volatile variable identifier, or it is dereferencing a pointer expression which is either volatile or pointing to a volatile variable.

We implement these rules in our definition of the I/O steps. Thus every other access to volatile data will not be safe.

Definition 5.82 ▶ *C-IL* I/O steps

The predicate $iostep^{\pi,\theta}$ denotes whether the next step in given *C-IL* configuration c is an I/O step.

$$\begin{aligned} iostep^{\pi,\theta}(c \in C_{C\text{-}IL}) \in \mathbb{B} \overset{def}{=}\ & stmt_{next}(c,\pi) = call\ e(E) \wedge rmw_c^{\pi,\theta}(e) \\ \vee\ & stmt_{next}(c,\pi) = (e = e') \wedge vol_c^{\pi,\theta}(e) \wedge \neg vol_c^{\pi,\theta}(e') \\ & \quad \wedge (e \in \mathbb{V} \vee (e = *(e'')) \wedge no2vol(e'')) \\ \vee\ & stmt_{next}(c,\pi) = (e = e') \wedge vol_c^{\pi,\theta}(e') \wedge \neg vol_c^{\pi,\theta}(e) \\ & \quad \wedge (e' \in \mathbb{V} \vee (e' = *(e'')) \wedge no2vol(e'')) \end{aligned}$$

where

$$no2vol(e) \overset{def}{=} \exists q, q', t.\ \tau_c^{\pi,\theta}(e) = (q', \textbf{ptr}(q,t)) \wedge \textbf{volatile} \notin q \cap q'.$$

The number of accessed bytes during a *C-IL* step varies in dependence of the type of the accessed data. For safety we need to identify the all accessed bytes of the global memory. We have to examine the executed statement and define recursively which memory addresses are accessed.

For a given configuration c, program π and context θ, the functions

$$R_c^{\pi,\theta} :: \mathbb{S} \to 2^{\mathbb{B}^{32}}$$

and

$$W_c^{\pi,\theta} :: \mathbb{S} \to 2^{\mathbb{B}^{32}}$$

return the reads set and the writes set of global addresses of the execution of a given statement respectively. We omit here the formal definitions. The set of read or written byte addresses are defined in Chapter 4.3 of [Bau14].

The predicate $safestep^{\pi,\theta}_{\textsf{C-IL}}$ denotes whether, given a program π and a context θ, the next step of thread i in configuration c is ownership safe according the ownership pair o, o'.

◀ **Definition** 5.83
Safe *C-IL* Step

$$\begin{aligned}
&safestep^{\pi,\theta}_{\textsf{C-IL}}(c \in C_{C\text{-}IL}, c' \in C_{C\text{-}IL}, i \in Tid, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B} \overset{def}{=} \\
&\qquad safeacc_{\textsf{C-IL}}(i, iostep^{\pi,\theta}(c), R_c^{\pi,\theta}(stmt_{next}(c,\pi)), W_c^{\pi,\theta}(stmt_{next}(c,\pi)), o) \\
&\quad \wedge\, safetransfer_{\textsf{C-IL}}(i, iostep^{\pi,\theta}(c), o, o')
\end{aligned}$$

Where the predicates

$$safeacc_{\textsf{C-IL}}(i \in Tid, io \in \mathbb{B}, R \in 2^{\mathbb{B}^{32}}, W \in 2^{\mathbb{B}^{32}}, o \in \mathcal{O}) \in \mathbb{B}$$

and

$$safetransfer_{\textsf{C-IL}}(i \in Tid, io \in \mathbb{B}, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B}$$

are defined equivalent to $safeacc$ and $safetransfer$ from Section 3.2.

The predicate $safeseq^{\pi,\theta}_{\textsf{C-IL}}$ denotes whether a sequence of steps of thread i is safe.

◀ **Definition** 5.84
Safe *C-IL*
Sequence

$$\begin{aligned}
&safeseq^{\pi,\theta}_{\textsf{C-IL}}(c \in C_{C\text{-}IL}, c' \in C_{C\text{-}IL}, i \in Tid, o \in \mathcal{O}, o' \in \mathcal{O}) \in \mathbb{B} \overset{def}{=} \\
&ownership\text{-}inv(o) \\
&\qquad \wedge ((c = c' \wedge o = o') \\
&\qquad\quad \vee (\forall c''.\ (\pi, \theta \vdash c \to_i c'') \implies \exists o''.\ safestep^{\pi,\theta}_{\textsf{C-IL}}(c, c'', i, o, o'') \\
&\qquad\qquad\qquad\qquad \wedge\, safeseq^{\pi,\theta}_{\textsf{C-IL}}(c'', c', i, o'', o')))
\end{aligned}$$

5.3.5 Compiler Correctness Theorem

The compiler consistency defines the simulation relation between *C-IL* and $MIPS_P$. We call a compiler correct if it guarantees that in every consistency point the compiler consistency holds between the *C-IL* and the $MIPS_P$ machine. We define the compiler correctness theorem iteratively for system mode execution fragments of

$MIPS_P$ extended IP schedule execution sequences. It states simulation between *C-IL* and $MIPS_P$ executions. Since we are talking about optimizing compilers our simulation step can consists of several *C-IL* steps. For us a simulation step is a sub-sequence of the execution, which is bordered by two subsequent consistency points

Basically our theorem says that for every hypervisor execution portion of a reordered $MIPS_P$ execution between two consistency points there exists a number of steps on the *C-IL* level, which simulate the considered $MIPS_P$ steps and lead the *C-IL* machine into a consistent state.

Theorem 5.1 (*CC-IL* Compiler Correctness) *The compiler correctness theorem says that, if*

- $\pi \in$ *prog*$_{C\text{-}IL}$ *is a* C-IL *program,*
- $\theta \in$ *context*$_{C\text{-}IL}$ *is the context of* π*,*
- $info \in info_{CC\text{-}IL}$ *is the static compiler information,*
- $h^0 \in C_M$ *is the initial configuration of the machine executing the compiled program,*
- $h^0 \xrightarrow{\beta} h'$ *is an arbitrary IP-schedule execution of the compiled program,*
- h^k *is some hypervisor consistency point of any processor* i *in* $h^0 \xrightarrow{\beta} h'$*,*
- h^l *is the next consistency point after* h^k*,*
- c *is a* C-IL *configuration which is a consistency point of thread* i*,*
- *consistency holds for all running threads between* c *and* h^k*,*
- $\mathbb{A}_{\text{C-IL}}$, $\mathbb{A}^{ro}{}_{\text{C-IL}}$, $\mathbb{A}_{MIPS_P}$ *and* $\mathbb{A}^{ro}{}_{MIPS_P}$ *are defined as in Section 5.3.4 and Section 3.2.2 and* o_m *is a* $MIPS_P$ *ownership consistent with* o_c*,*

then there exists a configuration c'*, between which and* h^l *consistency holds for all running threads.* c' *is either equal to* c *or it is the next consistency point of thread* i *and is obtained from* c *by processing some steps of thread* i*.*

Furthermore, if

- o_c *and* o'_c *are a valid* C-IL *ownership pair, such that the execution from* c *to* c' *is ownership-safe according* o_c *and* o'_c*,*
- *and* o_m *and* o'_m *are a* $MIPS_P$ *ownership pair consistent with* o_c *and* o'_c*,*

then also the corresponding $MIPS_P$ execution from h^k to h^l is ownership-safe according to o_m and o'_m.

If the last instruction of the considered $MIPS_P$ sub-sequence is a store word to the APIC, then the statement to be executed in c is a function call to the external function $sendipi$. This implies that writes to the APIC originate only from the $sendipi$ function.

$$
\begin{aligned}
&\forall h^0, \beta.\ IPsched(h^0, \beta)\\
&\quad \wedge \forall c, o_c, o_m, i, k \in \mathbb{N} \wedge k < |\beta|.\\
&\qquad hypCP_{\textsf{C-IL}}(h^0, i, \beta, k)\\
&\qquad \wedge\, l = nextCP(h^0, \beta, k)\\
&\qquad \wedge\, CP^{\pi,\theta}_{\textsf{C-IL}}(c(i))\\
&\qquad \wedge\, (\forall j \in Pid.\ running_j(\beta, k) \implies consis_{\textsf{C-IL}}(c, info, h^k, j))\\
&\qquad \wedge\, consis^{\mathcal{O}}(o_c, o_m)\\
&\qquad \implies \exists c'.\ \pi, \theta \vdash c \rightarrow^*_i c'\\
&\qquad\quad \wedge\, CP^{\pi,\theta}_{\textsf{C-IL}}(c'(i))\\
&\qquad\quad \wedge\, (\forall j \in Pid.\ running_j(\beta, l) \implies consis_{\textsf{C-IL}}(c', info, h^l, j))\\
&\qquad\quad \wedge\, (\forall o'_c, o'_m.\ consis^{\mathcal{O}}(o'_c, o'_m) \wedge safeseq^{\pi,\theta}_{\textsf{C-IL}}(c(i), c'(i), i, o_c, o'_c)\\
&\qquad\qquad\qquad\qquad\qquad \implies safeseq_{MIPS_P}(h^k, \beta[k : l-1], o_m, o'_m))\\
&\qquad\quad \wedge\, (store_{\mathbb{A}_{apic}}(h^{l-1}.core_i, I(h^{l-1}.core_i, h^{l-1}.m))\\
&\qquad\qquad\qquad\qquad\qquad \implies stmt_{next}(c(i), \pi) = \textbf{call}\ sendipi())
\end{aligned}
$$

Proof The proof of this great theorem is not in the scope of this thesis. Similar simulatino theorems have been proven previously for C (or languages close to C). In [LP08, Lei08] is presented a compiler correctness theorem for a non-optimizing compiler. The proof of a compiler correctness theorem for an optimizing compiler is presented in [Ler09].

In [Bau14] a sequential simulation theorem has been applied in the proof of a general simulation theorem between two concurrent systems including the transfer of ownership-safety.

Chapter 6

CC-IL+IPI Semantics

In the previous chapter we presented the $CC\text{-}IL$ semantics in which define the execution of concurrent *C-IL* code. A hypervisor implemented in *C-IL* interacts with the underlying hardware. Its execution is influenced by hardware features like IPIs. The hypervisor uses IPIs and relies on their semantics. In hypervisors we send an IPI from *C-IL*, which is then broadcasted on the $MIPS_P$ level, and causes the execution of a given *C-IL* IPI handler as part of the IPI service routine. It appears that programmers have in their mind semantics which combine $CC\text{-}IL$ and (parts of) $MIPS_P$. In this chapter we define such a system semantics which lift the IPI semantics to the program language level. We call this new semantics $CC\text{-}IL{+}IPI$.

In order to figure out the kind of IPI component required for our $CC\text{-}IL$ extension we consider the following abstract modeling of the IPI mechanism. A thread i executes the *C-IL* external function $sendipi$. It is implemented by a write to the APIC interrupt command register, which sets the DS bit of the processor's APIC. The $MIPS_P$ IPI transition broadcasts the IPI, clears the DS bit and sets the corresponding interrupt request bit in the APICs of the targets. The targets are interrupted and start executing the *C-IL* IPI handler, which is indicated by clearing the corresponding interrupt request bit and setting the corresponding interrupt in-service bit in the APIC. When the handler terminates the in-service bit is cleared and the interrupted thread continues from the point, where it was interrupted.

We define $CC\text{-}IL{+}IPI$ in a way, which allows to talk about the complete scenario from above. $CC\text{-}IL{+}IPI$ and $CC\text{-}IL$ differ in the configuration and the operational semantics.

◄ **Definition** 6.1 Sequential $C\text{-}IL{+}IPI$ Configuration

We extend the C-IL configuration with a hardware related component $apic \in C^{apic}_{C\text{-}IL}$, which contains three Boolean flags.

$$C_{C\text{-}IL+IPI} \overset{def}{=} [\mathcal{M} \in \mathbb{B}_{gm} \to \mathbb{B}^8, s \in frame^*_{\textsf{C-IL}}, apic \in C^{apic}_{C\text{-}IL}].$$

$$C^{apic}_{C\text{-}IL} \overset{def}{=} [DS \in \mathbb{B}, IRR \in \mathbb{B}, ISR \in \mathbb{B}]$$

DS denotes a send-IPI request. IRR denotes a pending IPI request. ISR denotes that the thread executes the *C-IL* IPI handler.

Definition 6.2 ▶ Concurrent *C-IL+IPI* Configuration

A concurrent *C-IL+IPI* configuration is defined by the record $C_{CC\text{-}IL+IPI}$.

$$C_{CC\text{-}IL+IPI} \overset{def}{=} [\mathcal{M} \in \mathbb{B}_{gm} \to \mathbb{B}^8, s \in Tid \to frame^*_{\text{C-IL}}, apic \in C^{apic}_{CC\text{-}IL}]$$

where the *CC-IL+IPI* APIC component contains one *C-IL+IPI* APIC per thread.

$$C^{apic}_{CC\text{-}IL} \overset{def}{=} Tid \to C^{apic}_{C\text{-}IL}$$

From a concurrent configuration $c \in C_{CC\text{-}IL+IPI}$ we can construct the sequential *C-IL+IPI* configuration $c(t) \in C_{C\text{-}IL+IPI}$ of given thread $t \in Tid$.

$$c(t \in Tid) \in C_{C\text{-}IL+IPI} \overset{def}{=} [c.\mathcal{M}, c.s(t), c.apic(t)]$$

In later statements and proofs we will need to convert a given concurrent *CC-IL+IPI* configuration to *CC-IL* configuration and vice versa. Therefore we define the following two functions.

Definition 6.3 ▶ Converting *CC-IL+IPI*

The function $chw2cil$ converts a given *CC-IL+IPI* configuration to *CC-IL* configuration. The function $cil2chw$ unites a given *CC-IL* configuration and a *C-IL* APIC to a *CC-IL+IPI* configuration.

$$chw2cil(c \in C_{CC\text{-}IL+IPI}) \in C_{CC\text{-}IL} \overset{def}{=} [c.\mathcal{M}, c.s]$$

$$cil2chw(c \in C_{CC\text{-}IL}, apic \in C^{apic}_{CC\text{-}IL}) \in C_{CC\text{-}IL+IPI} \overset{def}{=} [c.\mathcal{M}, c.s, apic]$$

6.1 CC-IL+IPI operational semantics

As previously mentioned, we intend to mirror steps of the hardware in our semantics. Thus *CC-IL+IPI* operational semantics consists of

- steps of the threads that correspond to the *C-IL* program, which we denote by $\xrightarrow{cil}$ and
- two additional hardware steps for the invocation of the IPI handler (which we call Jump to IPI Service Routine, or JIPISR) and for the broadcasting of IPIs, which we denote by $\xrightarrow{jipisr}$ and $\xrightarrow{ipi}$ respectively.

We define $\xrightarrow{cil}$ in Section 6.1.1, $\xrightarrow{jipisr}$ in Section 6.1.2 and $\xrightarrow{ipi}$ in Section 6.1.3. The local steps $\xrightarrow{cil}$ and $\xrightarrow{jipisr}$ are executed on a sequential $C\text{-}IL\text{+}IPI$ configuration of a given thread and $\xrightarrow{ipi}$ is a global step executed on the whole $CC\text{-}IL\text{+}IPI$ configuration.

We define the $CC\text{-}IL\text{+}IPI$ semantics top-down. A step in the $CC\text{-}IL\text{+}IPI$ semantics is either a *C-IL* step of some thread, or a local JIPISR step, or an IPI step.

◀ **Definition** 6.4
CC-IL + IPI step

$$\frac{\pi, \theta \vdash c \xrightarrow{cil}_t c' \vee \pi, \theta \vdash c \xrightarrow{jipisr}_t c' \vee \pi, \theta \vdash c \xrightarrow{ipi} c'}{\pi, \theta \vdash c \to c'}$$

The execution of local $\xrightarrow{cil}$ and $\xrightarrow{jipisr}$ steps in the concurrent context is defined by the following two rules.

$$\frac{\begin{array}{c}\pi, \theta \vdash c(t) \xrightarrow{cil} (\mathcal{M}', s', apic') \\ c' = (\mathcal{M}', c.s[t \mapsto s'], c.apic[t \mapsto apic'])\end{array}}{\pi, \theta \vdash c \xrightarrow{cil}_t c'}$$

$$\frac{\begin{array}{c}\pi, \theta \vdash c(t) \xrightarrow{jipisr} (\mathcal{M}', s', apic') \\ c' = (c.\mathcal{M}, c.s[t \mapsto s'], c.apic[t \mapsto apic'])\end{array}}{\pi, \theta \vdash c \xrightarrow{jipisr}_t c'}$$

We introduce the following shorthands for $CC\text{-}IL\text{+}IPI$ steps.

$$c \xrightarrow[\pi,\theta]{cil(t)} c' \overset{def}{=} \pi, \theta \vdash c \xrightarrow{cil}_t c'$$

$$c \xrightarrow[\pi,\theta]{jipisr(t)} c' \overset{def}{=} \pi, \theta \vdash c \xrightarrow{jipisr}_t c'$$

$$c \xrightarrow[\pi,\theta]{ipi} c' \overset{def}{=} \pi, \theta \vdash c \xrightarrow{ipi} c'$$

A sequence of steps we denote by

$$c \xrightarrow[\pi,\theta]{\beta} c',$$

where

$$\forall i \in [0 : |\beta| - 1].\ \beta_i = ipi \vee \exists t \in Tid.\ \beta_i = cil(t) \vee \beta_i = jipisr(t).$$

We define a predicate to denote for a given step $c \xrightarrow[\pi,\theta]{\alpha} c'$ whether it is a local JIPISR step, or an IPI step.

◀ **Definition** 6.5
Hardware step

$$hw\text{-}step(\alpha) \overset{def}{=} \alpha \in \{jipisr(t), ipi\}$$

6.1.1 C-IL Steps

C-IL steps in the $CC\text{-}IL{+}IPI$ semantics are mostly the same as the steps in the *C-IL* semantics as presented in 5.1.10.

If the next statement is an assignment, goto, if-not-goto, function call, return with result

$$stmt_{next}(c,\pi) \in \{e_0 = e_1, \textbf{goto } l, \textbf{ifnot } e \textbf{ goto } l,\\ \textbf{call } e(E), e_0 = \textbf{call } e(E), \textbf{return } e$$

we only strengthen the condition on the transition.

In the above cases we require that either there is no interrupt request or the thread is executing the service routine.

$$(c.apic.IRR = 0) \vee (c.apic.ISR = 1)$$

We shown as an example the rule for executing an assignment statement in $CC\text{-}IL{+}IPI$.

$$\frac{stmt_{next}(c,\pi) = (e_0 = e_1) \quad ((c.apic.IRR = 0) \vee (c.apic.ISR = 1))}{\pi, \theta \vdash c \to inc_{loc}(write_\theta(c, [\&e_0]_c^{\pi,\theta}, [e_1]_c^{\pi,\theta}))}$$

The execution of the return statement without return value is different in *C-IL* and $C\text{-}IL{+}IPI$.

Definition 6.6 ▶ $C\text{-}IL{+}IPI$ Function Return without Result

We distinguish between three types of return statements without return value:

- return statements executed by the program thread,
- return statements of handler functions called by the $ipihandler$,
- the return statement of the $ipihandler$.

For return statements executed by the program thread and return statements of handler functions called by the $ipihandler$ we drop the frame as usually.

$$\frac{stmt_{next}(c) = \textbf{return} \qquad c.rds_{top} = \bot \\ (\neg((c.apic.IRR = 1 \wedge c.apic.ISR = 0) \vee (c.apic.ISR = 1 \wedge c.f_{top} = ipihandler)))}{\pi, \theta \vdash c \to drop_{frame}(c)}$$

The return statement of the $ipihandler$ ends the servicing of an IPI and we need to extend the transition. In that case additionally to dropping the frame we clear the ISR bit.

$$\frac{stmt_{next}(c) = \textbf{return} \quad c.rds_{top} = \bot \quad c.apic.ISR = 1 \quad c.f_{top} = ipihandler}{\pi, \theta \vdash c \to drop_{frame}(c)[apic.ISR \mapsto 0]}$$

The function call to the external $sendipi$ function changes the $MIPS_P$ APIC. In order to keep the *C-IL* APIC consistent with $MIPS_P$ we redefine in our extended semantic the $sendipi$ transition relation defined by $\theta.R_{extern}$.

In addition to the functionality from Definition 5.46 the write to the APIC ICR on the $C\text{-}IL{+}IPI$ level sets the DS flag.

◀ **Definition** 6.7
$sendipi$ Transition Relation

$$(\perp, c, c') \in \theta.R_{extern}(sendipi) \implies c' = inc_{loc}(write_\theta(c, ICR_a, data))[DS \mapsto 1]$$

6.1.2 JIPISR Step

The JIPISR step in our $C\text{-}IL{+}IPI$ semantics defines the invocation of the interrupt handler. It is processed if there is a pending IPI request. It changes only the stack and the APIC of a single thread.

The execution of the *C-IL* program is interrupted by pushing the frame of the IPI handler on the stack. The effect is similar to a (non-existing) function call to the IPI handler. Furthermore the IRR and the ISR flags are flipped.

◀ **Definition** 6.8
IPI Handler Call

$$\frac{c.apic.IRR = 1 \qquad c.apic.ISR = 0}{\pi, \theta \vdash c \to frame_{ipi}(c)[apic.IRR \mapsto 0, apic.ISR \mapsto 1]}$$

The function $frame_{ipi}$ creates a new frame for the IPI handler and pushes it on the top of the stack.

$$frame_{ipi}(c \in C_{C\text{-}IL}) \in C_{C\text{-}IL} \overset{def}{=} c[s := (\mathcal{M}'_{\mathcal{E}}, \perp, ipihandler, 0) \circ c.s]$$

where $\mathcal{M}'_{\mathcal{E}}$ has to offer enough space for the handler's local variables

$$\mathcal{V}' = \pi.\mathcal{F}(ipihandler).\mathcal{V}.$$

The $ipihandler$ has no parameters,

$$\pi.\mathcal{F}(ipihandler).npar = 0$$

thus $\mathcal{V}'$ contains only local variables. According to that fact the condition on the size of $\mathcal{M}'_{\mathcal{E}}$ is the following:

$$\forall i \in [0 : |\mathcal{V}'| - 1].\ \mathcal{V}'[i] = (v_i, t_i) \Longrightarrow |\mathcal{M}'_{\mathcal{E}}(v_i)| = size_\theta(t_i).$$

6.1.3 IPI Step

All the steps defined up to here are local and only change the $apic$ of a single thread. The $CC\text{-}IL{+}IPI$ IPI step is the only step where we do access in one step the $apic$ components of all threads. It implements the broadcasting of an interrupt.

Definition 6.9 ▶ Broadcast IPI Step

If the DS flag of some thread indicates send-ipi request, the IPI is broadcasted to all other threads.

$$\frac{c.apic(t).DS = 1}{\pi, \theta \vdash c \to c[apic \mapsto send_{ipi}(c.apic, t)]}$$

The function $send_{ipi}$ defines the changes of the configurations of the all threads. It clears the DS flag of the sending thread and sets the IRR flags of all other threads.

$$send_{ipi}(apic \in C^{apic}_{CC\text{-}IL}, t \in Tid) \in C^{apic}_{CC\text{-}IL}$$

$$send_{ipi}(apic, t)(t') \overset{def}{=} \begin{cases} (0, apic(t').IRR, apic(t').ISR) & if\ t' = t \\ (apic(t').DS, 1, apic(t').ISR) & otherwise \end{cases}$$

6.2 CC-IL+IPI Safety

In the previous chapter we have defined ownership safety for *C-IL* program thread steps (Definition 5.83) and for *C-IL* steps sequences of a given thread (Definition 5.84). In the ownership safety of $CC\text{-}IL\text{+}IPI$ executions we additionally consider the new hardware steps and also distinguish between *C-IL* steps of a program thread and *C-IL* steps of an interrupt thread. Furthermore we extend the ownership to the concurrent context where an execution sequence contains steps of different threads. We want to use the $\mathcal{O}_{XT}$ ownership from $MIPS_P$ level on the $CC\text{-}IL\text{+}IPI$ level. Since in $\mathcal{O}_{XT}$ the stack addresses are visible, we actually need another ownership type for $CC\text{-}IL\text{+}IPI$. Our workaround for this is to define an empty stack addresses set on the $CC\text{-}IL\text{+}IPI$ level.

The static part of the ownership, i.e. the address space and the read only memory, is defined by $\mathbb{A}_{C\text{-}IL}$ and $\mathbb{A}^{ro}{}_{C\text{-}IL}$.

The dynamic part of $\mathcal{O}_{XT}$ contains program thread addresses, handler thread addresses and stack addresses. Thus we need to redefine the the ownership consistency for our extended simulation.

Definition 6.10 ▶ Ownership Consistency XT

Given $CC\text{-}IL\text{+}IPI$ ownership o_c and $MIPS_P$ ownership o_m the predicate $consis^{\mathcal{O}}_{XT}$ states that:

- o_c and o_m satisfy the ownership invariant,
- the sets of shared addresses for *C-IL* and $MIPS_P$ are equal,
- the set of program thread addresses for processor i is equal to the set of addresses owned by the *C-IL* program thread i,

- the set of handler thread addresses for processor i is equal to the set of addresses owned by the *C-IL* handler thread i plus the corresponding IPCB regions,
- the set of stack addresses for processor i is equal to the set of addresses in the stack region of the *C-IL* thread i,
- and the set of stack addresses for $CC\text{-}IL\text{+}IPI$ thread i is empty.

$$
\begin{aligned}
&consis^{\mathcal{O}}_{XT}(o_c \in \mathcal{O}_{XT}, o_m \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}\\
&\quad \wedge\ ownership\text{-}inv_{XT}(o_c)\\
&\quad \wedge\ ownership\text{-}inv_{XT}(o_m)\\
&\quad \wedge\ o_c.\mathbb{A}^{sh} = o_m.\mathbb{A}^{sh}\\
&\quad \wedge\ \forall i \in Tid.\ (o_m.O^p_i = o_c.O^p_i)\\
&\quad \wedge\ \forall i \in Tid.\ (o_m.O^h_i = o_c.O^h_i \cup \mathbb{A}_{ipcb_i})\\
&\quad \wedge\ \forall i \in Tid.\ (o_m.O^s_i = StR(i, info))\\
&\quad \wedge\ \forall i \in Tid.\ (o_c.O^s_i = \emptyset)
\end{aligned}
$$

◀ **Definition** 6.11
Safe $CC\text{-}IL\text{+}IPI$ Step

The predicate $safestep^{\pi,\theta}_{C\text{-}IL+IPI}$ denotes whether for a given program π and a context θ the next step of thread i in configuration c is ownership safe according the ownership pair o, o'.

$$
\begin{aligned}
&safestep^{\pi,\theta}_{C\text{-}IL+IPI}(c \in C_{C\text{-}IL+IPI}, c' \in C_{C\text{-}IL+IPI}, i \in Tid, ih \in \mathbb{B},\\
&\qquad\qquad\qquad\qquad\qquad\qquad o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}\\
&safeacc_{C\text{-}IL+IPI}(i, iostep^{\pi,\theta}(c), ih, R^{\pi,\theta}_c(stmt_{next}(c,\pi)), W^{\pi,\theta}_c(stmt_{next}(c,\pi)), o)\\
&\wedge\ safetransfer_{C\text{-}IL+IPI}(i, iostep^{\pi,\theta}(c), ih, o, o')
\end{aligned}
$$

where

$$
\begin{aligned}
&safetransfer_{C\text{-}IL+IPI}(i \in Tid, io \in \mathbb{B}, ih \in \mathbb{B}, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}\\
&\qquad\qquad\qquad\qquad\qquad\qquad safetransfer_{XT}(i, io, ih, o, o')
\end{aligned}
$$

and the predicate

$$safeacc_{C\text{-}IL+IPI}(i \in Tid, io \in \mathbb{B}, ih \in \mathbb{B}, R \in 2^{\mathbb{B}^{32}}, W \in 2^{\mathbb{B}^{32}}, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}$$
$$\begin{cases} (R \subseteq o.O_i^h \cup o.\mathbb{A}^{sh} \cup o.\mathbb{A}^{ro}) & \textit{if } ih \wedge io \\ \wedge (W \subseteq o.O_i^h \cup (o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^h)) & \\ & \\ (R \subseteq o.O_i^h \cup o.\mathbb{A}^{ro}) & \textit{if } ih \wedge \neg io \\ \wedge (W \subseteq o.O_i^h \setminus o.\mathbb{A}^{sh}) & \\ & \\ (R \subseteq o.O_i^p \cup o.\mathbb{A}^{sh} \cup o.\mathbb{A}^{ro}) & \textit{if } \neg ih \wedge io \\ \wedge (W \subseteq o.O_i^p \cup (o.\mathbb{A}^{sh} \setminus o.\overline{O}_i^p)) & \\ & \\ (R \subseteq o.O_i^p \cup o.\mathbb{A}^{ro}) & \textit{if } \neg ih \wedge \neg io \\ \wedge (W \subseteq o.O_i^p \setminus o.\mathbb{A}^{sh}) & \end{cases}$$

is defined similarly to $safeacc_{XT}$ (Definition 4.6) from Section 4.1.

The only difference appears in the accesses to local variables, which on the $C\text{-}IL{+}IPI$ level are safe by default due to the separate frames for the program thread and the handler thread. Therefore we have to remove the conditions for the stack addresses in the definition of $safeacc_{C\text{-}IL+IPI}$.

Definition 6.12 ▶ Safe $C\text{-}IL{+}IPI$ Sequence

The predicate $safeseq^{\pi,\theta}_{C\text{-}IL+IPI}$ denotes whether a sequence of C-IL steps of thread i in a $C\text{-}IL{+}IPI$ execution is safe.

$$\begin{aligned} & safeseq^{\pi,\theta}_{C\text{-}IL+IPI}(c \in C_{C\text{-}IL+IPI}, c' \in C_{C\text{-}IL+IPI}, i \in Tid, o \in \mathcal{O}_{XT}, o' \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\ & ownership\text{-}inv_{XT}(o) \\ & \quad \wedge ((c = c' \wedge o = o') \\ & \quad\quad \vee (\forall c''.\ (\pi, \theta \vdash c \xrightarrow{cil}_i c'') \implies \exists o''.\ safestep^{\pi,\theta}_{C\text{-}IL+IPI}(c, c'', i, o, o'') \\ & \quad\quad\quad\quad \wedge\ safeseq^{\pi,\theta}_{C\text{-}IL+IPI}(c'', c', i, o'', o'))) \end{aligned}$$

In order to be able to talk about a complete $C\text{-}IL{+}IPI$ execution and cover also the new hardware steps in the semantic we need another definition. We note that $CC\text{-}IL{+}IPI$ hardware steps are safe by definition and do not transfer ownership.

Definition 6.13 ▶ Safe $CC\text{-}IL{+}IPI$ Sequence

The predicate $safeseq^{\pi,\theta}_{CC\text{-}IL+IPI}$ denotes the ownership safety of an execution

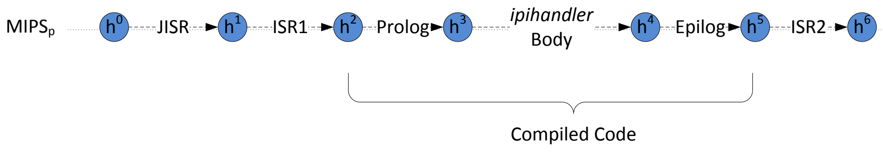

Figure 6.1: $MIPS_P$ execution sequence with IPI handling.

sequence $c \xrightarrow[\pi,\theta]{\beta} c'$.

$$\begin{aligned}
&safeseq^{\pi,\theta}_{CC\text{-}IL+IPI}(c,\beta,c',o,o') \in \mathbb{B} \overset{def}{=} ownership\text{-}inv_{XT}(o) \wedge ownership\text{-}inv_{XT}(o') \\
&\quad \wedge (\beta = \varepsilon \wedge c = c' \\
&\quad \vee \forall c''.\ c \xrightarrow[\pi,\theta]{\beta_0} c'' \implies \\
&\quad\quad (hw\text{-}step(\beta_0) \implies safeseq^{\pi,\theta}_{CC\text{-}IL+IPI}(c'',\mathbf{tl}(\beta),c',o,o') \\
&\quad\quad \wedge \beta_0 = cil(t) \implies \exists o''.\ safestep^{\pi,\theta}_{C\text{-}IL+IPI}(c(t),c''(t),t,c.apic(t).ISR,o,o'') \\
&\quad\quad\quad\quad\quad\quad \wedge safeseq^{\pi,\theta}_{CC\text{-}IL+IPI}(c'',\mathbf{tl}(\beta),c',o'',o')))
\end{aligned}$$

A given program π is ownership safe according to an ownership state o if all possible executions of it starting in configuration c are safe. ◀ **Definition** 6.14 Safe Program

$$\begin{aligned}
&safeprog^{\pi,\theta}_{CC\text{-}IL+IPI}(c \in C_{CC\text{-}IL+IPI}, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\
&\quad \forall c',\beta.\ c \xrightarrow[\pi,\theta]{\beta} c' \implies \exists o'.\ safeseq^{\pi,\theta}_{CC\text{-}IL+IPI}(c,\beta,c',o,o')
\end{aligned}$$

6.3 IPI Service Routine

We want to prove that a trace of the $CC\text{-}IL+IPI$ machine corresponds to a trace of the $MIPS_P$ that includes handling of an IPI. Therefore we need to take a deeper look at the IPI service routine execution sequences on $MIPS_P$ (Figure 6.1).

We relate the execution of the interrupt service routine on $MIPS_P$ to the *C-IL* program and the compiler and distinguish the following parts in it. The IPI service routine starts with JISR and continues with the execution of the ISR code. The ISR in case of an IPI consists of portions written in assembler and a the *C-IL* function $ipihandler$. In Figure 6.2 we sketch the assembler implementation of the ISR. It begins with a list of instructions $ISR1$. The instructions in $ISR1$ store the

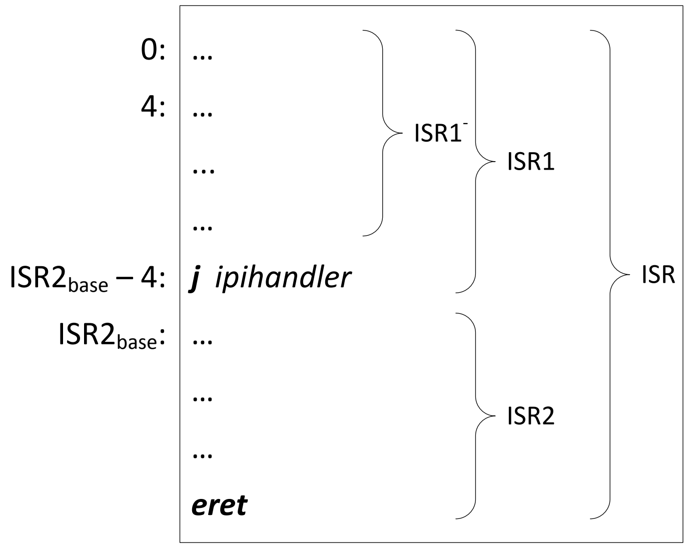

Figure 6.2: ISR implementation consisting of *ISR1* and *ISR2*. *ISR1* starts at address 0 and ends with a jump instruction. *ISR2* starts at address $ISR2_{base}$ and ends with an *eret* instruction

complete context in the processor's IPCB and dispatch the interrupt to the IPI handler. *ISR1* ends with a function call to the *C-IL* handler. The implementation of the call decreases the stack pointer by four (allocating space for return address), stores the program counter (return address) in that newly allocated stack region by a *sw* instruction and sets the program counter to the beginning of *ipihandler*. Since store word can not directly access the program counter, we need first to store it in the link register by a jump-and-link instruction and of course add the corresponding offset to the linked program counter before storing it in the stack. The last instruction in *ISR1* is a jump instruction. We refer to the instructions before the jump instruction by $ISR1^{-}$. The label to the *C-IL* function in the jump instruction is replaced by the compiler with the address of the first prolog instruction of the function [Lev99]. After the prolog has set up the stack frame for the function we have reached a consistency point. The execution continues with the instructions that implement the body of *ipihandler*. Then the epilog is executed and we continue with the instruction after the jump to *ipihandler*. The list of ISR instructions, that follow the jump, restore the context from the IPCB. We refer to this list of instructions by *ISR2*. The last instruction in *ISR2* is *eret*, which concludes the service routine.

The corresponding execution in *CC-IL+IPI* is depicted in Figure 6.3. In c^0 we have a pending IPI request, which according to the *CC-IL+IPI* semantics leads

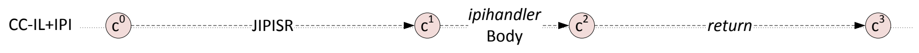

Figure 6.3: $CC\text{-}IL{+}IPI$ execution sequence with IPI handling.

to the execution of $JIPISR$ step, followed by the execution of $ipihandler$. An important state is the configuration before the return statement of the $ipihandler$. The execution of this return statement changes also the APIC component.

In Figure 6.4 we depict both executions together. Significant states in the $MIPS_P$ execution, in consideration of compiler consistency, are h^0, h^3, h^4, and h^6.

Actually in h^4 the execution should continue with the return code. We expect that for a return statement without return value compilers generate an empty list of instructions. In such cases all required actions for completing the function call are implemented by the function's epilog. Therefore we consider in our figures the return code as part of the epilog for simplicity. Nevertheless in formal definitions we refer to this point in the execution by a program counter pointing to the first instruction of the return statement.

$$info.stmtcode(ipihandler, |\pi.\mathcal{F}(ipihandler).\mathcal{P}| - 1)$$

In case of an empty return code the compiler would assign the epilog to the return statement.

$$info.stmtcode(ipihandler, |\pi.\mathcal{F}(ipihandler).\mathcal{P}| - 1) = info.ecode(ipihandler)$$

The $MIPS_P$ configuration during the ISR depends on the concrete implementation of the assembler portions and the prolog instructions generated by the compiler. We do not consider the details of the step by step execution in this work, but rather give a specification of the ISR. We omit the tedious details of the hypervisor ISR implementation, i.e. the instruction lists $ISR1$ and $ISR2$. A similar example can be found in [Sha12].

In order to formalize our specification of ISR we need to define the execution of instruction lists on $MIPS_P$.

◀ **Definition** 6.15 Instruction List Execution

The predicate $instrexec$ denotes the uninterrupted execution of the instruction list l without branch and jump instructions on the processor i starting in configuration h. It comprises all required conditions on the configurations and the execution sequence, e.g. the program counter of the processor points to the first instruction on the list $h.m_4(h.pc_i) = l_0$ and h' is reached after executing $n = |l|$ (non-interrupted) core steps on processor i.

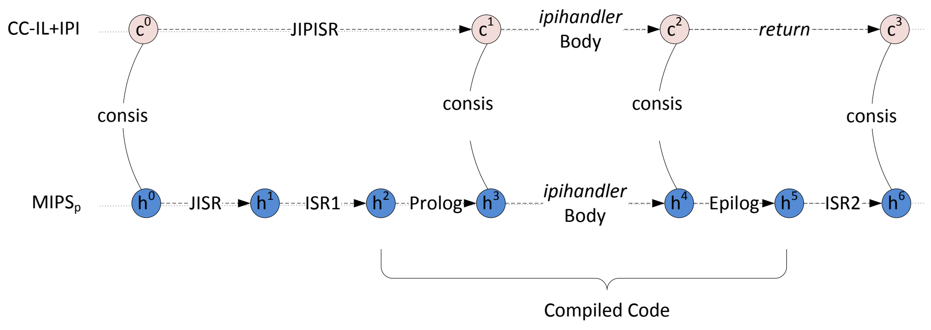

Figure 6.4: Simulation between $CC\text{-}IL{+}IPI$ and $MIPS_P$ execution sequence with IPI handling.

$$
\begin{aligned}
instrexec(h \in C_M, \beta \in (\Sigma_M)^*, l \in (\mathbb{B}^{32})^*, i \in Pid, h' \in C_M) \in \mathbb{B} \overset{def}{=}& \\
h.m_{4\cdot|l|}(h.pc_i) = l& \\
\wedge\, h \overset{\beta}{\to} h'& \\
\wedge\, |\beta| = |l|& \\
\wedge\, \forall j \in [0 : |\beta| - 1].\ core(\beta_j) \wedge \beta_j.pid = i& \\
\wedge\, \forall j \in [0 : |\beta| - 1].\ \neg isJISR(h^j.core_i, eev^j, I(h^j.core_i, h^j.M))& \\
\wedge\, \forall j \in [1 : |\beta| - 1].\ h^j.pc_i = h.pc_i + bin_{32}(4 \cdot (j - 1))&
\end{aligned}
$$

where

$$
eev^j = \begin{cases} eev(h^j.apic_{pid}) & if\ \ exint(h^j.core_{pid}, h^j.apic_{pid}) \\ 0^{256} & otherwise \end{cases}
$$

We also use the notation the following shorthand notation.

$$
h \xrightarrow[l,\ i]{\beta} h' = instrexec(h, \beta, l, i, h')
$$

The following software condition must be satisfied by the concrete hypervisor implementation in order to apply our approach.

Software Condition 9 (ISR) *Given an ISR implementation with structure as in Figure 6.2 we have to prove that after the execution of $ISR1^-$:*

- *the program counter points to the jump to $ipihandler$,*
- *the stack pointer is decrease by four,*

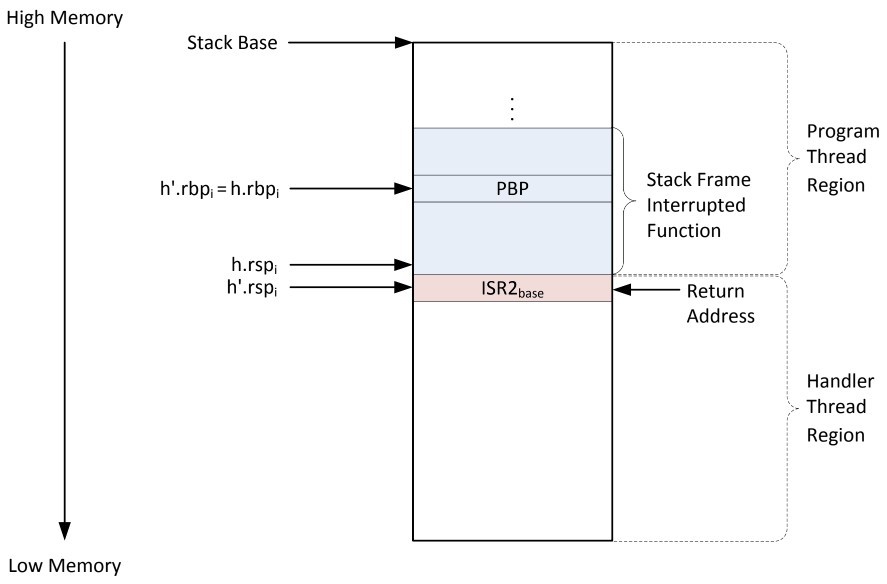

Figure 6.5: Stack layout in h' after executing $ISR1^-$ in h.

- *the base pointer is unchanged,*
- *the memory location pointed to by the stack pointer stores the address of the first instruction of $ISR2$,*
- *the IPCB of the processors stores the initial core state,*
- *all other components are unchanged.*

$$\begin{aligned}
&\forall h, \beta, h', i.\ h \xrightarrow[ISR1^-,\ i]{\beta} h' \implies \\
&\quad h'.pc_i = ISR2_{base} - bin_{32}(4) \\
&\quad \wedge h'.rsp_i = h.rsp_i - bin_{32}(4) \\
&\quad \wedge h'.rbp_i = h.rbp_i \\
&\quad \wedge h'.m_4(h.rsp_i - bin_{32}(4)) = ISR2_{base} \\
&\quad \wedge ipcb(h', i) = h.core_i \\
&\quad \wedge \forall a \in \mathbb{B}^{32}.\ a \notin (\mathbb{A}_{ipcb} \cup [h.rsp_i : h'.rsp_i]) \implies h.m(a) = h'.m(a) \\
&\quad \wedge \forall j \in Pid.\ j \neq i \implies h.core_j = h'.core_j \\
&\quad \wedge \forall j \in Pid.\ h.apic_j = h'.apic_j
\end{aligned}$$

The stack layout is depicted in Figure 6.5.

After the execution of $ISR2$ the machine state has to satisfy the following conditions:

- *the program counter points to the interrupted instruction,*
- *the core state is restored from the IPCB,*
- *the IPI in-service bit in the APIC is cleared,*
- *all other components are unchanged.*

$$\begin{aligned}
&\forall h, \beta, h', i.\ h \xrightarrow[ISR2,\ i]{\beta} h' \implies \\
&\qquad h'.pc_i = ipcb(h,i).epc \\
&\qquad \wedge\, h'.rbp_i = h.rbp_i \\
&\qquad \wedge\, h'.rsp_i = h.rsp_i \\
&\qquad \wedge\, h'.core_i = ipcb(h,i) \\
&\qquad \wedge\, h'.apic_i = h.apic_i[isr[0] \mapsto 0] \\
&\qquad \wedge\, \forall a \in \mathbb{B}^{32}.\ h'.m(a) = h.m(a) \\
&\qquad \wedge\, \forall j \in Pid.\ j \neq i \implies h'.core_j = h.core_j \\
&\qquad \wedge\, \forall j \in Pid.\ j \neq i \implies h'.apic_j = h.apic_j
\end{aligned}$$

Since we heavily rely on the values stored in the IPCB we state the following software condition.

Software Condition 10 (IPCB) *All generated instructions which are executed during IPI handling, i.e. the code for $ipihandler$ and all its callee functions, do not change the processor's IPCB.*

In our ownership setting, i.e. excluding the IPCB from the $CC\text{-}IL{+}IPI$ address space, and for a correct compiler this condition is guaranteed for ownership-safe programs.

In the following two software conditions we state the validity of the prolog and epilog code, that have to be met by the compiler. We constrain our predicates only to the state components relevant for our simulation relation. A complete prolog / epilog specification must at least additionally talk about the saving / restoring of callee-save registers. Since $ISR2$ does not depend on callee-save registers the given conditions are sufficient for our purposes.

Software Condition 11 (Prolog Correctness) *The predicate $pcode_V$ denotes the validity of the prolog of a given function f. It states that if:*

- *$f \in \mathbb{F}_{name}$ is a function name*

- $info \in info_{CC\text{-}IL}$ *is the static compiler information,*
- $pcode_f \in (\mathbb{B}^{32})^*$ *is the prolog code generated for f*

$$pcode_f = h.m_{info.pcode_{size}(f)}(info.pcode(f))$$

- $h \in C_M$ *is the initial configuration of the prolog execution,*
- $h' \in C_M$ *is the final configuration of the prolog execution,*

then the execution of $pcode(f)$ ends in a configuration with program counter pointing to the code of the first statement in f. Furthermore $pcode(f)$ allocates stack space for f's frame (assuming that the space for parameters and the return address is already allocated) and adjusts the stack and base pointer. The previous base pointer is stored in the stack. The prolog does not change anything but the state of executing processor and the portion of the stack which it allocates.

$$pcode_V{}^{\pi,\theta}(h \in C_M, info \in info_{CC\text{-}IL}, f \in \mathbb{F}_{name}, i \in Pid, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=}$$

$$\begin{aligned} \forall \beta, h'.\ h \xrightarrow[pcode_f,\ i]{\beta} h' \implies \\ & h'.pc_i = info.stmtcode(f, 0) \\ & \wedge h'.rbp_i = h.rsp_i - bin_{32}(4) \\ & \wedge h'.rsp_i = h.rsp_i - bin_{32}(info.fsize(f, 0) + info.fsize_p(f) + 4) \\ & \wedge h'.m_4(h.rsp_i - bin_{32}(4)) = h.rbp_i \\ & \wedge \forall a \in \mathbb{B}^{32}.\ a \notin [h.rsp_i : h'.rsp_i] \implies h'.m(a) = h.m(a) \\ & \wedge \forall j \in Pid.\ j \neq i \implies h'.core_j = h.core_j \\ & \wedge \forall j \in Pid.\ h'.apic_j = h.apic_j \end{aligned}$$

Software Condition 12 (Epilog Correctness) *The predicate $ecode_V$ denotes the validity of the epilog of a given function f from program π. It states that if:*

- $f \in \mathbb{F}_{name}$ *is a function name*
- $info \in info_{CC\text{-}IL}$ *is the static compiler information,*
- $ecode_f \in (\mathbb{B}^{32})^*$ *is the epilog code generated for f*

$$ecode_f = h.m_{info.ecode_{size}(f)}(info.ecode(f))$$

- $h \in C_M$ *is the initial configuration of the epilog execution,*
- $h' \in C_M$ *is the final configuration of the epilog execution,*

then the execution of $ecode_f$ frees the stack space allocated for f's frame and restores the program counter and the previous base pointer from the frame header.

$$
\begin{aligned}
&ecode_V{}^{\pi,\theta}(h \in C_M, info \in info_{CC\text{-}IL}, f \in \mathbb{F}_{name}, i \in Pid, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\
&\quad \forall \beta, h'.\ h \xrightarrow[ecode_f,\ i]{\beta} h' \implies \\
&\qquad\qquad h'.pc_i = h.m_4(h.rbp_i + bin_{32}(4)) \\
&\qquad\qquad \wedge\, h'.rbp_i = h.m_4(h.rbp_i) \\
&\qquad\qquad \wedge\, h'.rsp_i = h.rsp_i + bin_{32}(info.fsize(f, |\pi.\mathcal{F}(f).\mathcal{P}| - 1)) \\
&\qquad\qquad \wedge\, \forall a \in \mathbb{B}^{32}.\ h'.m(a) = h.m(a) \\
&\qquad\qquad \wedge\, \forall j \in Pid.\ j \neq i \implies h'.core_j = h.core_j \\
&\qquad\qquad \wedge\, \forall j \in Pid.\ h'.apic_j = h.apic_j
\end{aligned}
$$

6.4 CC-IL+IPI Simulation

Based on *C-IL* compiler consistency we want to prove simulation between $CC\text{-}IL{+}IPI$ and $MIPS_P$ executions, where the extended $CC\text{-}IL{+}IPI$ semantics allow us to prove simulation for more general $MIPS_P$ executions than the compiler correctness for ordinary *C-IL* from the previous chapter. In addition to the executions covered by Theorem 5.1 we now consider $MIPS_P$ executions including IPIs.

6.4.1 CC-IL+IPI Simulation Relation

Since the IPI handler does not have a caller its stack frame header does not have a return address to a *C-IL* location. Thus we have to change the control consistency predicate and exclude the *ipihandler* frame from the return address condition.

Definition 6.16 ▶ *C-IL+IPI* Control Consistency

The predicate $consis^{control}_{C\text{-}IL+IPI}$ states that the value of the program counter is the address of the current *C-IL* instruction. For any function f of the program except the first function ($main$) and $ipihandler$ the value of the return address in the function's stack frame is the address of the first post-call-code instruction after the function call to f.

$$
\begin{aligned}
&consis^{control}_{C\text{-}IL+IPI}(s \in frame^*_{\text{C-IL}}, info \in info_{\text{C-IL}}, core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \overset{def}{=} \\
&\quad core.pc = info.stmtcode(s.f_{top}, s.loc_{top}) \\
&\quad \wedge\, \forall i \in [1 : |s| - 1].\ s.f_i \neq ipihandler \implies \\
&\qquad\qquad m_4(ra_{\text{C-IL}}(i, s, m, info)) = info.pccode(s.f_{i-1}, s.loc_{i-1})
\end{aligned}
$$

With the next software condition we guarantee the proper flow of the program despite the weakening of the control consistency.

Software Condition 13 (Calls $ipihandler$) *In a given program π does not exist a function call statement to $ipihandler$.*

We also have to change the consistency condition for the local variables. The change is related to the top-most frame at the time of interrupt rising, i.e. the frame preceding the $ipihandler$ frame. Since the invocation of $ipihandler$ happens without function call, there is no $pre\text{-}call\text{-}code$ and thus caller-save register are not saved on the stack. Still the interrupted function may have local variables that are stored in caller-save registers, which are then eventually overwritten by the ISR.

◀ **Definition** 6.17 *C-IL+IPI* Local Variables Consistency

The predicate $consis^{var}_{C\text{-}IL+IPI}$ denotes local variables consistency. It states that the local variables and parameters are stored properly. The variables that are passed in registers according to the compiler information are stored in the registers only for the topmost frame. We make a case distinction in our definition of the consistency relation for non-topmost frames. If the subsequent frame belongs to the $ipihandler$, then we compare the values of the local variables in caller-save registers to the registers in the IPCB[1]. For the other frames they are on the stack either in the callee-save or in the caller-save regions of the corresponding frame.

$$consis^{var}_{C\text{-}IL+IPI}(s \in frame^*_{\textsf{C-IL}}, \pi \in \textsf{prog}_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}}, core \in C_{CORE},$$
$$m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}$$
$$\forall i, j \in \mathbb{N}.\ i < |s| \wedge j < info.lvar_{num}(s.f_i) \implies s.\mathcal{M}_i(v_j) =$$
$$\begin{cases} core.gpr(reg) & \text{if } reg \neq \bot \wedge i = top(s) \\ m_4(ba_i - bin_{32}(info.reg_{off}(f, reg))) & \text{if } reg \neq \bot \wedge i < top(s) \\ & \wedge s.f_{i+1} \neq ipihandler \\ m_4(ipcb_{ba}(i) + bin_{32}(reg \cdot 4)) & \text{if } reg \neq \bot \wedge i < top(s) \\ & \wedge s.f_{i+1} = ipihandler \\ m_{size_\theta(qt2t(t_j))}(ba_i + bin_{32}(4 + info.par_{off}(v_j, s.f_i))) & \text{if } reg = \bot \wedge j < npar \\ m_{size_\theta(qt2t(t_j))}(ba_i - bin_{32}(info.lvar_{off}(v_j, s.f_i))) & \text{if } reg = \bot \wedge j \geq npar \end{cases}$$

where

$$\begin{aligned} (v_j, t_j) &= (\pi.\mathcal{F}(s.f_i)).\mathcal{V}[j] \\ reg &= info.lvar_{reg}(v_j, s.f_i, s.loc_i) \\ npar &= (\pi.\mathcal{F}(s.f_i)).npar \\ ba_i &= ba_{\textsf{C-IL}}(i, s, info) \end{aligned}$$

[1]We have already defined IPCB in Section 4.2.

We integrate the required changes of control and local variables consistency in the definitions of stack consistency, local consistency and compiler consistency in the next three definitions.

Definition 6.18 ▶ *C-IL+IPI* Stack Consistency

The predicate $consis^{stack}_{C\text{-}IL+IPI}$ denotes stack consistency as a wrapper for register, previous base pointer and local variables consistency.

$$\begin{aligned}
&consis^{stack}_{C\text{-}IL+IPI}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}},\\
&\qquad core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}\\
&\qquad\qquad consis^{regs}_{\textsf{C-IL}}(s, info, core)\\
&\qquad\qquad \wedge\ consis^{pbp}_{\textsf{C-IL}}(s, info, m)\\
&\qquad\qquad \wedge\ consis^{var}_{C\text{-}IL+IPI}(s, info, core, m)
\end{aligned}$$

Definition 6.19 ▶ *C-IL+IPI* Local Consistency

The predicate $consis^{local}_{C\text{-}IL+IPI}$ denotes local compiler consistency.

$$\begin{aligned}
&consis^{local}_{C\text{-}IL+IPI}(s \in frame^*_{\textsf{C-IL}}, info \in info_{\textsf{C-IL}},\\
&\qquad core \in C_{CORE}, m \in \mathbb{B}^{32} \to \mathbb{B}^8) \in \mathbb{B} \overset{def}{=}\\
&\qquad\qquad consis^{control}_{C\text{-}IL+IPI}(s, info, core, m)\\
&\qquad\qquad \wedge\ consis^{stack}_{C\text{-}IL+IPI}(s, info, core, m)
\end{aligned}$$

Definition 6.20 ▶ *C-IL+IPI* Compiler Consistency

The predicate $consis_{CC\text{-}IL+IPI}$ denotes compiler consistency.

$$\begin{aligned}
&consis_{CC\text{-}IL+IPI}(c \in C_{CC\text{-}IL+IPI}, info \in info_{CC\text{-}IL}, h \in C_M, i \in Pid) \in \mathbb{B} \overset{def}{=}\\
&\quad consis^{global}_{CC\text{-}IL}(chw2cil(c), info, h)\\
&\quad \wedge\ consis^{local}_{C\text{-}IL+IPI}((chw2cil(c))(i).s, info(i), h.core_i, h.m)
\end{aligned}$$

C-IL+IPI compiler consistency exposes some crucial differences between C-IL and *C-IL+IPI*. Obviously in the presence of an *ipihandler* frame on the stack *C-IL+IPI* consistency does not imply C-IL consistency and vice versa. In the context of a simulation proof between *C-IL+IPI* and $MIPS_P$ it is natural to want to make use of the C-IL compiler correctness theorem for the portions in a *C-IL+IPI* execution consisting only of C-IL steps. With the *C-IL+IPI* extensions of compiler consistency we can not do this. Instead we need a theorem that similarly to Theorem 5.1 talks about simulation blocks of steps of a single *C-IL+IPI* thread between two consecutive consistency points. In contrast to the C-IL compiler correctness theorem, we now also want to talk about the execution of interrupt handlers, and maintain the extended ownership safety. Though the special cases of starting and ending an interrupt thread execution remain excluded from the theorem. This limitation of the theorem is expressed by the absence of JISR and $eret$ instruction in the $MIPS_P$ execution. Thus the code neither pushes nor pops an $ipihandler$ frame on/from the stack.

Theorem 6.1 ($CC\text{-}IL{+}IPI$ **Compiler Correctness)** *The compiler correctness theorem for* $C\text{-}IL{+}IPI$ *theorem says that, if*

- $\pi \in prog_{\text{C-IL}}$ *is a* C-IL *program,*
- $\theta \in context_{\text{C-IL}}$ *is the context of* π*,*
- $info \in info_{CC\text{-}IL}$ *is the static compiler information,*
- $h^0 \in C_M$ *is the initial configuration of the machine executing the compiled program,*
- $h^0 \xrightarrow{\beta} h'$ *is an arbitrary IP-schedule execution of the compiled program,*
- h^k *is some hypervisor consistency point of any processor* i *in* $h^0 \xrightarrow{\beta} h'$*,*
- h^l *is the next consistency point after* h^k*,*
- *between* h^k *and* h^l *the* $MIPS_P$ *neither an* $eret$ *instruction nor a JISR step are executed,*
- $\hat{c}$ *is a* $CC\text{-}IL$ *configuration which is a consistency point of thread* i*,*
- $apic$ *is an* $C^{apic}_{CC\text{-}IL}$ *configuration,*
- c *is a* $CC\text{-}IL{+}IPI$ *configuration build of* $\hat{c}$ *and* $apic$*,*
- $CC\text{-}IL{+}IPI$ *consistency holds for all running threads between* c *and* h^k*,*

then there exists a configuration c'*, between which and* h^l *consistency holds for all running threads.* c' *is either equal to* c *or it is the next consistency point of thread* i *and is obtained from* c *by processing some* C-IL *steps of thread* i*.*

Furthermore, for $\mathbb{A}_{\text{C-IL}}$*,* $\mathbb{A}^{ro}{}_{\text{C-IL}}$*,* $\mathbb{A}_{MIPS_P}$ *and* $\mathbb{A}^{ro}{}_{MIPS_P}$ *defined as in Section 5.3.4 and Section 3.2.2 if*

- o_c *and* o_c' *are a valid* $CC\text{-}IL{+}IPI$ *ownership pair, such that the execution from* c *to* c' *is ownership-safe according* o_c *and* o_c'*,*
- *and* o_m *and* o_m' *are a* $MIPS_P$ *ownership pair consistent with* o_c *and* o_c'*,*

then also the corresponding $MIPS_P$ *execution from* h^k *to* h^l *is ownership-safe according to* o_m *and* o_m'*.*

If the last instruction of the considered $MIPS_P$ *sub-sequence is a store word to the APIC, then the statement to be executed in* c *is a function call to the external function* $sendipi$*.*

Formally we state the theorem as follows:

$$
\begin{aligned}
&\forall h^0, \beta.\ \mathit{IPsched}(h^0, \beta)\\
&\quad \wedge \forall \hat{c}, apic, o_c, o_m, i, k \in \mathbb{N} \wedge k < |\beta|.\\
&\qquad hypCP_{\textit{C-IL+IPI}}(h^0, i, \beta, k)\\
&\qquad \wedge\ l = nextCP(h^0, \beta, k)\\
&\qquad \wedge\ consis^{\mathcal{O}}_{XT}(o_c, o_m)\\
&\qquad \wedge\ CP^{\pi,\theta}_{\textsf{C-IL}}(\hat{c}(i))\\
&\qquad \wedge\ c = cil2chw(\hat{c}, apic)\\
&\qquad \wedge\ (\forall j \in Pid.\ running_j(\beta, k) \implies consis_{\textit{CC-IL+IPI}}(c, info, h^k, j))\\
&\qquad \wedge\ (\forall j \in [k : l-1].\ \neg(jisr_{IPI}(\beta_j) \vee iint(h^j.core_i, I(h^j.core_i, h^j.M))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee\ eret(I(h^j.core_i, h^j.M)))\\
&\qquad \implies \exists c'.\ \pi, \theta \vdash c \rightarrow^*_i c'\\
&\qquad\qquad \implies (CP^{\pi,\theta}_{\textsf{C-IL}}(\hat{c}'(i))\\
&\qquad\qquad \wedge\ (\forall j \in Pid.\ running_j(\beta, l) \implies consis_{\textit{CC-IL+IPI}}(c', info, h^l, j))\\
&\qquad\qquad \wedge\ (\forall o'_c, o'_m.\ consis^{\mathcal{O}}_{XT}(o'_c, o'_m) \wedge safeseq^{\pi,\theta}_{\textit{C-IL+IPI}}(c(i), c'(i), i, o_c, o'_c)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \implies safeseq_{XT}(h^k, \beta[k : l-1], o_m, o'_m))\\
&\qquad\qquad \wedge\ (store_{\mathbb{A}_{apic}}(h^{l-1}.core_i, I(h^{l-1}.core_i, h^{l-1}.m))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \implies stmt_{next}(apic(i), \pi) = \textbf{call}\ sendipi()))
\end{aligned}
$$

Proof The proof of this theorem is identical to the proof of Theorem 5.1 and similarly it does not fall into the scope of this thesis. We note, that we do not introduce any additional gap in our theory by skipping the proof of the *C-IL+IPI* compiler correctness theorem since it is based on the same arguments as *C-IL* compiler correctness theorem. It has the same inductive structure with an additional case distinction on the execution context, i.e. execution of an interrupt thread or of a program thread. The condition about the absence of JISR and *eret* steps implies that the number of existing *ipihandler* frames on the stack does not change in the induction step.

$$\forall n.\ c(i).f_n = ipihandler \iff c'(i).f_n = ipihandler$$

Thus we can deduce consistency from the induction hypothesis.

Considering the transfer of the ownership-safety from the *C-IL+IPI* to the $MIPS_P$ level we again use the same argumentation as in Theorem 5.1. Given that the C-IL program obeys the stronger ownership rules we do not need any new type of proof techniques to claim that the compiled code is also ownership-safe according to the extended ownership. We just consider the ownership transfer separately for the execution of the handler and of the program and talk about subsets of the joint ownership address space.

✔

In addition to the consistency conditions in our simulation relation we have to consider the consistency of the $apic$ components.

Definition 6.21 HW Consistency

The predicate $hw\text{-}consis$ denotes that the Boolean flags in the $apic$ component of a given thread i are consistent with the corresponding bits in the APIC of processor i, which are related to software IPIs.

$$hw\text{-}consis(c \in C_{CC\text{-}IL+IPI}, h \in C_M, i \in Tid) \in \mathbb{B} \overset{def}{=} \begin{aligned}[t] & h.irr_i[0] = c.apic[i].IRR \\ \wedge\ & h.isr_i[0] = c.apic[i].ISR \\ \wedge\ & h.icr_i.DS = c.apic[i].DS \end{aligned}$$

Definition 6.22 $CC\text{-}IL{+}IPI$ Simulation Relation

The predicate $SR^{MIPS_P}_{CC\text{-}IL+IPI}$ denotes the simulation relation between $CC\text{-}IL{+}IPI$ and $MIPS_P$.

$$\begin{aligned} & SR^{MIPS_P}_{CC\text{-}IL+IPI}(c \in C_{CC\text{-}IL+IPI}, info \in info_{CC\text{-}IL}, h \in C_M, i \in Pid) \in \mathbb{B} \overset{def}{=} \\ & \qquad consis^{global}_{CC\text{-}IL}(chw2cil(c), info, h) \\ & \qquad \wedge\ consis^{local}_{C\text{-}IL+IPI}((chw2cil(c))(i).s, info(i), h.core_i, h.m) \\ & \qquad \wedge\ hw\text{-}consis(c, h, i) \end{aligned}$$

6.4.2 Consistency Points

Executions with IPIs contain more consistency points than the ones defined in Section 5.3.1. Therefore we extend the old definitions. We define consistency points based on the set of $MIPS_P$ interleaving points. As part of the simulation proof, that we present in the next section, we prove that the $CC\text{-}IL{+}IPI$ simulation relation holds in the state which we define to be a consistency points.

We add to the set of hypervisor consistency points the state after the execution of an $eret$ instruction at the end of IPI service routine.

Definition 6.23 Hypervisor Consistency Point of Processor i

The predicate $hypCP_{C\text{-}IL+IPI}$ denotes whether the k-th step in the execution $h \xrightarrow{\beta} h'$ is a hypervisor consistency point for processor i.

$$\begin{aligned} & hypCP_{C\text{-}IL+IPI}(h \in C_M, i \in Pid, \beta \in (\Sigma_M)^*, k \in [0 : |\beta| - 1]) \in \mathbb{B} \overset{def}{=} \\ & \qquad\qquad i = \beta_k.pid \wedge core(\beta_k) \wedge h^k.pc_{\beta_k.pid} \in \mathbb{A}_{cp} \\ & \qquad\qquad \vee\ eret_{IPI}(h^{k-1}, \beta_{k-1}) \end{aligned}$$

We overload the definition of $MIPS_P$ consistency points corresponding to the new definition of hypervisor consistency points.

Definition 6.24 ▶ $MIPS_P$ Consistency Point

The predicate CP_{MIPS_P} denotes whether the k-th step in the execution $h \xrightarrow{\beta} h'$ is a consistency point.

$$\begin{aligned} CP_{MIPS_P}(h \in C_M, \beta \in (\Sigma_M)^*, k \in [0 : |\beta| - 1]) \in \mathbb{B} \overset{def}{=} \\ \exists i \in Pid.\ hypCP_{C\text{-}IL+IPI}(h, i, \beta, k) \vee k = min\{j \mid i = \beta_j.pid\} \\ \vee\ guestIP(h, \beta, k) \vee ipi(\beta_{k-1}) \end{aligned}$$

In the set of the software consistency points we do not need changes. We would probably expect the initial states of IPI steps, i.e. states in which exists a set DS flag, to be consistency points. According the $CC\text{-}IL{+}IPI$ operational semantics, such configurations follow steps executing the function $sendipi$. As all function calls, also the ones to $sendipi$ start and end in consistent states. Thus we do not have additional conditions here.

6.4.3 CC-IL+IPI Simulation Theorem

The simulation theorem that we define and prove in this section differs from Theorem 5.1 in several aspects.

First of all, as we have already mentioned on numerous occasions, we cover $MIPS_P$ executions with IPI handling. We have to consider, that in $CC\text{-}IL{+}IPI$ IPIs happen on the border between statements. This is definitely not true in the general case for $MIPS_P$ executions. Moreover, it is not true also for IP schedule $MIPS_P$ executions in general. Thus proving a simulation between $CC\text{-}IL{+}IPI$ executions and $MIPS_P$ executions is feasible only for those $MIPS_P$ executions, that satisfies the $sched_{XT}$ predicate.

Second we also consider guest steps. By doing that we prove simulation of complete execution sequences and not only of their hypervisor fragments. Since guest steps in $MIPS_P$ are quite restricted this extension is trivial here. For a $MIPS_P$ that is closer to $X86$ concerning steps in guest mode one would have to consider on the one hand guest instruction execution and context switch and on the other hand MMU steps. We already have sketched how the consistency relation has to be adapted and what are the conditions on the context switch and the guest execution. MMU steps have been examined in [Kov13], where in an approach similar to ours the *C-IL* semantics has been extended and a simulation theorem has been proven. We are convinced that our theory is applicable on a full-size $MIPS$ and can easily be combined with the approach from [Kov13].

According to this simulation we want to transfer ownership safety proven on the $CC\text{-}IL{+}IPI$ level to extended IP schedule $MIPS_P$ executions. Based on the Simulation Theorem 4.12, we can then conclude ownership safety for IP schedule $MIPS_P$ executions. For that purpose the theorem's claim have to be extended by a conjunct saying that the original IP schedule executions is safe. The proof of this extension is trivial, since we only reorder local program thread steps, and the ownership of the program thread does not change within the ISR blocks. Thus, if a given local program thread step is ownership safe at the end of the ISR block, then it is also ownership safe at the beginning of the ISR block. Having safety of IP schedule executions we can use the order reduction theorem form [Bau14] (Definition 3.30) to transfer the ownership safety to arbitrary interleaved $MIPS_P$ executions.

Before stating the simulation theorem we need to define formally some additional conditions.

In the initial state of both machines ($CC\text{-}IL{+}IPI$ and $MIPS_P$) the APIC is empty.

◀ **Definition** 6.25 Empty $CC\text{-}IL{+}IPI$ APIC

The predicate $emptyapic_{CC\text{-}IL+IPI}$ denotes whether a given $CC\text{-}IL{+}IPI$ APIC configuration is in an initial "clear" state for all threads.

$$emptyapic_{CC\text{-}IL+IPI}(a \in C^{apic}_{CC\text{-}IL}) \in \mathbb{B} \overset{def}{=} \forall t \in Tid.\ a(t) = (0,0,0)$$

◀ **Definition** 6.26 Empty $MIPS_P$ APIC

The predicate $emptyapic_{MIPS_P}$ denotes whether in a given $MIPS_P$ configuration the APICs of all processors are in an initial "clear" state.

$$\begin{aligned} emptyapic_{MIPS_P}(h \in C_M) \in \mathbb{B} \overset{def}{=} \forall i \in Pid.\ & \neg irr(h.apic_i) \\ & \wedge \neg isr(h.apic_i) \\ & \wedge \neg h.DS_i \end{aligned}$$

In the proof of the simulation theorem we will rely on properties of the $CC\text{-}IL{+}IPI$ and $MIPS_P$ execution as depicted in Figure 6.4. We capture the important properties of these executions in the following two lemmas.

Lemma 6.2 (Pre-handler Consistency) *The predicate $preIPIH_V$ states that if:*

- *h (h^0 from Figure 6.4) is a $MIPS_P$ configuration with pending IPI,*
- *h' (h^1 from Figure 6.4) is the configuration that we reach when we execute in h JISR,*
- *h'' (h^3 from Figure 6.4) is the configuration that we reach when we execute in h' the assembler code from the ISR for saving the context and a call to* C-IL *$ipihandler$, and the prolog generated by the compiler for $ipihandler$,*
- *c (c^0 from Figure 6.4) is a $CC\text{-}IL{+}IPI$ configuration, such that consistency holds between c and h,*
- *c' (c^1 from Figure 6.4) is the configuration that we reach after executing the JIPISR step in c,*

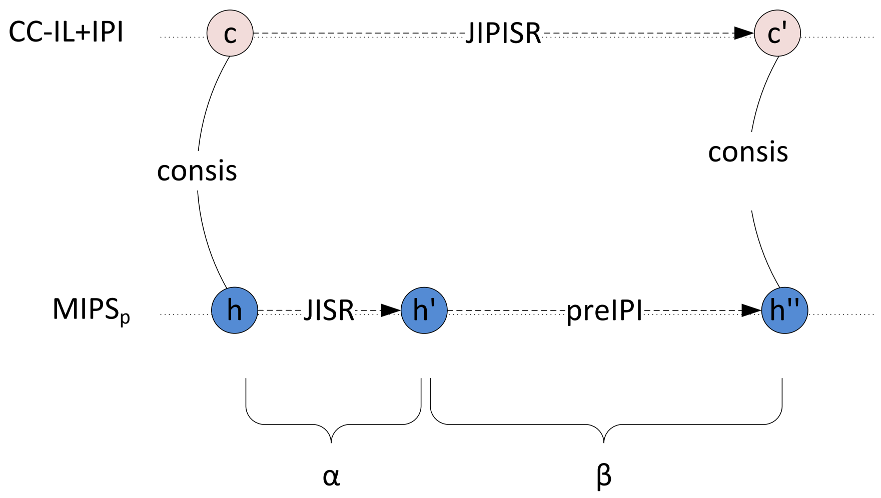

Figure 6.6: If c and h are consistent and we execute IPIJISR step in c and JISR, save context and the compiler generated prolog in h, then the new configurations c' and h'' are also consistent.

- *o is a valid ownership,*

then consistency will hold again between h' and c' and all executed ISA steps are ownership-safe (Figure 6.6).

$$\begin{aligned}
&\mathit{preIPIH}_V(h \in C_M, c \in C_{\mathit{CC\text{-}IL+IPI}}, \mathit{info} \in \mathit{info}_{\mathit{CC\text{-}IL}}, i \in \mathit{Pid}, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \forall \alpha, \beta, h', h''.\ h \overset{\alpha}{\to} h' \\
&\qquad \wedge \mathit{jisr}_{IPI}(\alpha, i) \\
&\qquad \wedge h' \xrightarrow[\mathit{preIPI},\ i]{\beta} h'' \\
&\qquad \wedge SR^{\mathit{MIPS}_P}_{\mathit{CC\text{-}IL+IPI}}(c, \mathit{info}, h, i) \\
&\qquad \wedge c' = \mathit{frame}_{\mathit{ipi}}(c)[\mathit{apic.IRR} \mapsto 0, \mathit{apic.ISR} \mapsto 1] \\
&\qquad \wedge \mathit{ownership\text{-}inv}_{XT}(o) \\
&\qquad \Longrightarrow\ SR^{\mathit{MIPS}_P}_{\mathit{CC\text{-}IL+IPI}}(c', \mathit{info}, h'', i) \\
&\qquad\qquad \wedge \mathit{safeseq}_{XT}(h, \alpha\beta, o, o)
\end{aligned}$$

where

$$\mathit{preIPI} = \mathit{ISR1}, \mathit{pcode}_{\mathit{ipihandler}}$$

Proof To prove the consistency we apply the specification of δ^{jisr}, Software Condition 9 and Software Condition 11. For the safety we rely on the following conditions:

- JISR is safe by definition.
- The ISR code accesses only $\mathbb{A}_{ipcb}$ and the stack memory.
- The prolog code accesses only the stack memory.
- All accesses to $\mathbb{A}_{ipcb}$ and the stack memory are safe because they are owned due to ownership invariant.
- There is no ownership transfer in place. All executed steps access only memory addresses in the IPCB and in the handler stack region (i.e. below $h.rsp$).

✔

Lemma 6.3 (Post-handler Consistency) *The predicate $postIPIH_V$ states that if:*

- *h is a $MIPS_P$ configuration with program counter pointing to the first instruction of $ipihandler$'s return statement (h^4 from Figure 6.4),*
- *h' is the configuration that we reach when we execute in h the epilog generated by the compiler for $ipihandler$ and the assembler code from the ISR for restoring the context (h^6 from Figure 6.4),*
- *c is a $CC\text{-}IL\text{+}IPI$ configuration, such that consistency holds between c and h (c^2 from Figure 6.4),*
- *c' is the configuration that we reach after executing the return step in c (c^3 from Figure 6.4),*
- *o is a valid ownership,*

then consistency will hold again between h' and c' and all executed ISA steps are ownership-safe (Figure 6.7).

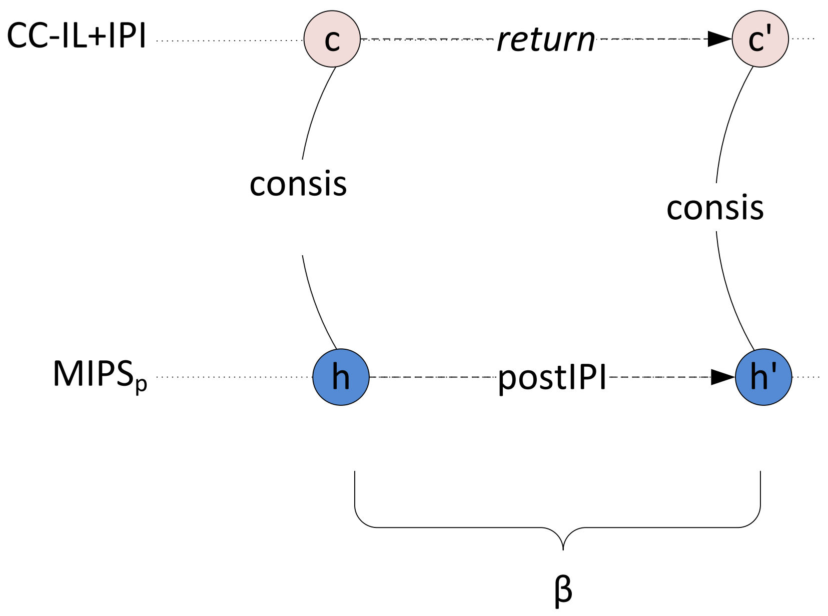

Figure 6.7: If c and h are consistent and we exit the ISR by executing the compiler generated epilog and restore context in h, and the $ipihandler$ return statement step in c, then the new configurations c' and h' are also consistent.

$$\begin{aligned}
&postIPIH_V(h \in C_M, c \in C_{CC\text{-}IL+IPI}, info \in info_{CC\text{-}IL}, i \in Pid, o \in \mathcal{O}_{XT}) \in \mathbb{B} \overset{def}{=} \\
&\qquad \forall \beta, h'.\ h \xrightarrow[postIPI,\ i]{\beta} h' \\
&\qquad \wedge\ ownership\text{-}inv_{XT}(o) \\
&\qquad \wedge\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c, info, h, i) \\
&\qquad \wedge\ c' = drop_{frame}(c)[apic.ISR \mapsto 0] \\
&\qquad \Longrightarrow\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c', info, h', i) \\
&\qquad\quad \wedge\ safeseq_{XT}(h, \beta, o, o)
\end{aligned}$$

where

$$postIPI = ecode_{ipihandler}, ISR2$$

Proof *To prove this lemma we apply the specification of δ^{eret}, Software Condition 12 and Software Condition 9 and argue on the safety similarly to the previous proof.*

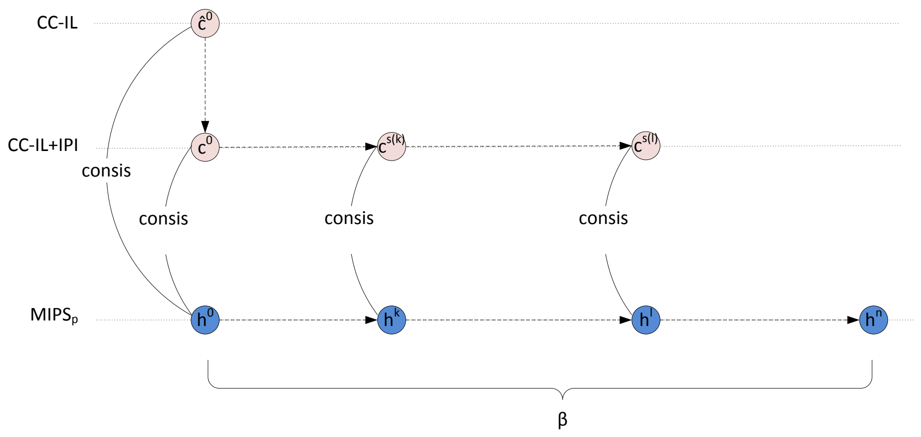

Figure 6.8: Simulation of a $MIPS_P$ execution sequence by a $CC\text{-}IL{+}IPI$ computation.

✔

Theorem 6.4 (Simulation Theorem $CC\text{-}IL{+}IPI$) *The simulation theorem says that, if*

- $\pi \in$ *prog*$_{\text{C-IL}}$ *is a* C-IL *program,*
- $\theta \in$ *context*$_{\text{C-IL}}$ *is the context of* π*,*
- $info \in info_{CC\text{-}IL}$ *is the static compiler information,*
- $h^0 \in C_M$ *is the initial configuration of the machine executing the compiled program,*
- $h^0 \overset{\beta}{\rightarrow} h^n$ *is an arbitrary extended IP-schedule execution of the compiled program,*
- *in* h^0 *all cores are in system mode,*
- *in* h^0 *all APICs are clear,*
- $apic^0$ *is an empty* $CC\text{-}IL{+}IPI$ *APIC record,*
- $\hat{c}^0 \in C_{CC\text{-}IL}$ *is the initial* $CC\text{-}IL$ *configuration of the program execution,*
- *all threads are at a consistency point in* $\hat{c}^0$*,*

- *$c^0 \in C_{CC\text{-}IL+IPI}$ is the initial $CC\text{-}IL{+}IPI$ configuration in which $\hat{c}^0$ is extended by $apic^0$,*
- *the simulation relation holds for all running threads between h^0 and c^0,*
- *$\pi \in prog_{\mathsf{C\text{-}IL}}$ is an ownership-safe* C-IL *program according to an initial ownership o_c,*
- *$\mathbb{A}_{\mathsf{C\text{-}IL}}$, $\mathbb{A}^{ro}{}_{\mathsf{C\text{-}IL}}$, $\mathbb{A}_{MIPS_P}$ and $\mathbb{A}^{ro}{}_{MIPS_P}$ are defined as in Section 5.3.4 and Section 3.2.2 and o_m is a $MIPS_P$ ownership consistent with o_c,*

then there exists a step function s and an execution sequence c^0, c^1, ..., $c^{s(|\beta|)}$ such that for all consistency points h^k the simulation relation holds between h^k and $c^{s(k)}$ for all running threads (Figure 6.8). Furthermore, there exists an ownership o'_m, such that the $MIPS_P$ execution from h^0 to h^k is ownership-safe according to o_m and o'_m.

$$
\begin{aligned}
&\forall h^0, \beta, o_c, o_m, \hat{c}^0, apic^0.\\
&\quad sched_{XT}(h^0, \beta)\\
&\quad \wedge \forall i \in Pid : \neg guest(h^0.core_i)\\
&\quad \wedge emptyapic_{MIPS_P}(h^0)\\
&\quad \wedge emptyapic_{CC\text{-}IL+IPI}(apic^0)\\
&\quad \wedge \forall i \in Pid.\ CP^{\pi,\theta}_{\mathsf{C\text{-}IL}}(\hat{c}^0(i))\\
&\quad \wedge c^0 = cil2chw(\hat{c}^0, apic^0)\\
&\quad \wedge \forall i \in Pid.\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^0, info, h^0, i)\\
&\quad \wedge safeprog^{\pi,\theta}_{CC\text{-}IL+IPI}(c^0, o_c)\\
&\quad \wedge consis^{\mathcal{O}}_{XT}(o_c, o_m)\\
&\quad \Longrightarrow \exists s.\ \forall k \leq |\beta|.\ CP_{MIPS_P}(h^0, \beta, k) \Longrightarrow\\
&\qquad\qquad (\forall i \in Pid.\ running_i(\beta, k) \Longrightarrow CP^{\pi,\theta}_{\mathsf{C\text{-}IL}}(chw2cil(c^{s(k)})(i))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^{s(k)}, info, h^k, i))\\
&\qquad\quad \wedge (\exists o'_m.\ safeseq_{XT}(h^0, \beta[0 : k-1], o_m, o'_m))
\end{aligned}
$$

Proof We prove the simulation between d and c by induction on the number of consistency points reached cpr. In the proof we match the current consistency point index cpr to configuration h^k.

Induction base $\quad cpr = 0$
In this case we set $k = 0$.
$CP_{MIPS_P}(h^0, \beta, 0)$ follows from the definition of hardware consistency points.
From the hypothesis we have:

$$\forall i \in Pid.\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^0, info, h^0, i).$$

Hence, we choose $s(0) = 0$. All running threads are in a consistency point since $chw2cil(cil2chw(c)) = c$.

We choose $o'_m = o_m$. The safety for an empty sequence

$$safeseq_{XT}(h^0, \varepsilon, o_m, o_m)$$

is defined by the ownership invariant of o_m. The ownership invariant of o_m is guaranteed by the hypothesis as conjunct of our ownership consistency predicate.

Induction step $\quad cpr \rightarrow cpr + 1$

In the induction step we consider inductively the execution between two consecutive consistency points h^k and h^l, $k < l$. The hypothesis of our theorem is defined on the initial state and does not depend on our induction parameter. Thus for proving the claim of the theorem for l we can rely on the hypothesis and on the claim for k, i.e. the consistency after the first k steps of the abstract hardware machine.

$$\begin{aligned}
&\exists s, k < |\beta|.\ CP_{MIPS_P}(h^0, \beta, k) \implies \\
&\qquad (\forall i \in Pid.\ running_i(\beta, k) \implies CP^{\pi,\theta}_{\textsf{C-IL}}(chw2cil(c^{s(k)})(i)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^{s(k)}, info, h^k, i)) \\
&\qquad \wedge\ \exists o^k_m.\ safeseq_{XT}(h^0, \beta[0:k-1], o_m, o^k_m)
\end{aligned}$$

We execute in this setting the steps between the current (k) and the next $(l = nextCP(h^0, \beta, k))$ consistency point and define the steps of the *C-IL* machine such that the following holds.

$$\begin{aligned}
&\exists s(l) \geq s(k),\ l < |\beta|.\ CP_{MIPS_P}(h^0, \beta, l) \implies \\
&\qquad (\forall i \in Pid.\ running_i(\beta, l) \implies CP^{\pi,\theta}_{\textsf{C-IL}}(chw2cil(c^{s(k)})(i)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^{s(l)}, info, h^l, i)) \\
&\qquad \wedge\ \exists o'_m.\ safeseq_{XT}(h^0, \beta[0:l], o_m, o'_m)
\end{aligned}$$

We do a case split on the type of the consistency point h^k:

Case 1: h^k is a guest consistency point $(guestIP(h, \beta, k))$.
This means that the next step β_k is either a guest step or VMEXIT.
In this case the next consistency point is the subsequent state, i.e. $l = k + 1$. Both steps are not visible on the *C-IL* level, thus $s(l) = s(k)$.
Both steps do not change relevant parts of the hardware state. From that follows that the execution of a guest step or VMEXIT maintains our consistency.

$$SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^{s(k)}, info, h^k, i) \implies SR^{MIPS_P}_{CC\text{-}IL+IPI}(c^{s(l)}, info, h^l, i)$$

Guest steps and VMEXIT are safe and do not change the ownership. Thus we choose $o'_m = o^k_m$.

$$\begin{aligned} & safeseq_{XT}(h^0, \beta[0:k-1], o_m, o'_m) \\ \wedge\ & safestep_{XT}(h^k, \beta_k, o'_m, o'_m) \\ \implies\ & safeseq_{XT}(h^0, \beta[0:l-1], o, o') \end{aligned}$$

We recall that if the *C-IL* compiler consistency is defined with a case distinction on the processor mode, in system mode we consider the real registers and in guest mode we would have to talk about the saved registers. Such consistency relation requires a more sophisticated ISA model than $MIPS_P$. It would provide semantics for guest instruction execution, guest/hypervisor mode switch and an intercept model. To maintain consistency during guest steps we then have to guarantee:

- that guest steps do not change the memory regions for the stack and for the saved hypervisor context and
- that guest steps do not change the APIC state.

For VMEXIT we would have to guarantee:

- that it restores properly the stored context and
- that it does not change the APIC state.

Details of this argumentation are given in [PBLS16].

Case 2: h^k is a hypervisor consistency point, i.e. $hypCP_{C\text{-}IL+IPI}(h^0, i, \beta, k)$. We have two possible cases:

- The program thread is running (i.e. the ISR bit is not set) and there is no pending IPI. Thus the next step is a program step of the hypervisor.
- The interrupt thread is running (i.e. the ISR bit is set). Thus the next step is a program step is executing $ipihandler$.

We are considering extended IP schedules, thus we know that between the current and next consistency points are executed either only program steps of the hypervisor or only $ipihandler$ steps. We want to apply the $C\text{-}IL{+}IPI$ compiler consistency theorem for the whole execution block starting at h^k and ending in the next consistency point. For that we have first to show that the hypothesis of Theorem 6.1 is satisfied. Then we use the theorem to derive our proof goal.
We apply the theorem with obvious instantiations of variables, e.g. the initial $CC\text{-}IL$ configuration is $\hat{c} = chw2cil(c^{s(k)})$.

- The first four conditions form the compiler correctness theorems's hypothesis are trivially satisfied.
- From the induction hypothesis we know that all running threads are at a consistency point. We are examining steps of processor processor i. This implies that i is running in k and therefore $CP^{\pi,\theta}_{\textsf{C-IL}}(chw2cil(c^{s(k)})(i))$ holds.
- The $C\text{-}IL{+}IPI$ consistency between $c^{s(k)}$ and h^k for all running threads follows from the $C\text{-}IL{+}IPI$ simulation relation between $c^{s(k)}$ and h^k.
- The absence of JISR and $eret$ instruction steps is satisfied by the definition of the current proof-case

Thus we can apply the $C\text{-}IL{+}IPI$ compiler correctness theorem (Theorem 6.1) and claim $C\text{-}IL{+}IPI$ consistency for all running threads between $c^{s(l)}$ and h^l.

For the APIC consistency we have to consider possible changes due to writes to $\mathbb{A}_{apic}$. Such steps are I/O steps and the considered block may only contain a single I/O step executed at the end. Therefore we need another case split on the last instruction executed by the hypervisor. Before looking into the two cases we recall that according to the $C\text{-}IL{+}IPI$ compiler correctness theorem writes to $\mathbb{A}_{apic}$ in compiled code correspond to function calls to $sendipi$ on the $C\text{-}IL{+}IPI$ level.

- If β_{l-1} is not a write to $\mathbb{A}_{apic}$, we know that the APIC state of the machines h^k and h^l is equivalent. On the $C\text{-}IL{+}IPI$ level the APIC state stays also unchanged and $hw\text{-}consis$ holds trivially and respectively consistency holds.
- If β_{l-1} is a write to $\mathbb{A}_{apic}$, the current statement in $c^{s(k)}$ can only be a function call to $sendipi$. The execution of the statement changes the APIC state in the $C\text{-}IL{+}IPI$ configuration the same way like the execution of β_{l-1} does this on the processor. After the two steps are executed $hw\text{-}consis$ holds again.

Concerning the ownership safety we need to consider that the safety conjunct in the claim of the Theorem 6.1 is an implication. Thus first we have to prove safety of the *C-IL* steps between $chw2cil(c^{s(k)})$ and $c^{s(l)}$.

From safety of the program $safeprog^{\pi,\theta}_{CC\text{-}IL+IPI}(c^0, o)$, which is part of our hypothesis, we can deduce safety for every step, which implies safety for the execution that we consider. Knowing that the *C-IL* steps are safe we get safety of the hardware steps.

Case 3: We are at a consistency point before an IPI step, i.e. $ipi(\beta_k)$.

From the guard of this transition we know that in h^k exists a processor with set DS bit. Due to the simulation relation the same holds on the *C-IL* level, therefore we can execute in c the IPI step.
Then $l = k + 1$ and $c^{s(l)}$ is obtained from $c^{s(k)}$ according Definition 6.9.
$c^{s(l)} = c^{s(k)}[apic \mapsto send_{ipi}(c.apic, i)]$

The IPI step is safe and changes on both machines the APIC state equivalently letting everything else including the ownership unchanged. Thus $hw\text{-}consis$ is maintained and everything else follows from the induction hypothesis.

Case 4: We are in a consistency point with an external interrupt request. The next consistency point is the one placed by the compiler at the beginning of the execution of the *C-IL* handler.

We define $c^{s(l)}$ by executing a JIPISR step and apply Lemma 6.2 which also guarantees the ownership safety execution.

Case 5: We are in a consistency point before the return statement of the *C-IL* function $ipihandler$. β_k is a core step and executes the first instruction of the return code of the function, i.e. the code of the last statement

$$h.pc_i = info.stmtcode(ipihandler, |\pi.\mathcal{F}(ipihandler).\mathcal{P}| - 1).$$

The next consistency point is the one after the $eret$ step.
During the service routine on both machines the APIC in service bits are set.
On *C-IL* we execute the return statement according to our extended semantic which pops the current frame from the stack and clears the in-service flag. We apply Lemma 6.3 obtaining consistency and safety of the MIPS execution.

✔

Chapter 7

Conclusion

In this thesis we describe a model and theory, how interrupts can be integrated in a pervasive verification approach for system software written in C. In the Verisoft XT project we used semantics, which should be considered incomplete, since they do not contain interrupts. The verification tool VCC is designed and developed for verifying multithreaded C programs. Verification of system software is more complex and requires adjustments of the tool's computational logic. In VCC the program thread and the handler thread are considered as two parallel C threads. Hardware parts are defined as hybrid objects, which are saved as ghost objects but do not follow the restriction, that information cannot flow from ghost to program memory. VCC proofs in this methodology and especially an IPI verification as described in [ACHP10] are incomplete and based on a series of assumptions. In this work we addressed and closed some of the gaps in the underlying theory.

- To the best of our knowledge we present the first C semantics with interrupts occurring during execution of the C-portion of the program[1]. The communication protocol from [ACHP10] and the implementation of AHV and HyperV use a bit vector, which served in our verification as an ad hoc abstraction of the local APIC. With the $CC\text{-}IL{+}IPI$ semantics we are able to replace this bit vector with a representation that fits more precisely to the hardware model. Moreover $CC\text{-}IL{+}IPI$ allows us to talk about interrupts on the C level.

- We have proven a simulation and the transfer of ownership-safety between our new C semantics and a realistic hardware model with reordered executions.

- To justify the reordered schedule and enable the transfer of ownership-safety to arbitrary interleaved executions we have proven an order reduction theorem to reorder handler executions to consistency points.

[1]Interrupts occurring during the execution of assembly portions were already treated in [PBLS16].

Since the VCC logic can interleave threads only at the boundaries between C-statements, any handler verification in VCC has to assume that interrupts occur only at such boundaries. With the reordering theory presented in this thesis we have justified this assumption.

- This thesis gives detailed justification for a VCC based code verification approach of C code with interrupts. The presented semantics as an extension of the computational model of the code-verifier reduces the need of meta arguments in system verification and contributes to the soundness of the verification tool.

- In the appended publication (Chapter 8) we present a generic and modular approach to specifying and verifying parallel interprocess communication, that allows the recovery of procedural abstraction. Furthermore the approach was applied in the verification of real IPI code (i.e. in an industrial and in an academic hypervisor).

As future work VCC has to be extended with a mechanism, which will ensure that the program thread and the handler thread run exclusively. Such an extension is still missing, but we have defined semantics against which it has to be proven sound.

Chapter 8

Appendix: Modular Specification and Verification of Interprocess Communication

In this Chapter we present a join work on a specification and verification method for Interprocess Communication (IPC). The results have been presented at the 2010 Formal Methods in Computer Aided Design conference (FMCAD10) and published in the conference's proceedings [ACHP10]. The paper was written by Eyad Alkassar, Ernie Cohen, Mark Hillebrand and Hristo Pentchev. All authors contributed to the specification of the generic and modular approach described in Section 8.4. The application of the method in the verification of the TLB Flush Example (Section 8.5) and IPIs in the Verisoft XT academic hypervisor and in the Microsoft's Hyper-V^{TM} Hypervisor (Section 8.6) was mostly done by the author of this thesis.

Abstract

The usual goal in implementing IPC is to make a cross-thread procedure call look like a local procedure call. However, formal specifications of IPC typically talk only about data transfer, forcing IPC clients to use additional global invariants to recover the sequential function call semantics. We propose a more powerful specification in which IPC clients exchange knowledge and permissions in addition to data. The resulting specification is polymorphic in the specification of the service provided, yet allows a client to use IPC without additional global invariants. We verify our approach using VCC, an automatic verifier for (suitably annotated) concurrent C code, and demonstrate its expressiveness by applying it to the verification of a multiprocessor flush algorithm.

8.1 Introduction

Procedural abstraction - the ability for the caller of a procedure to abstract a procedure call to a relation between its pre- and poststates - is one of the most important structuring mechanisms in all of programming methodology. The central role of procedural abstraction is reflected in the fact that it is built into not only all modern imperative languages, but also into most program logics and verifiers for such languages. However, in a concurrent or distributed system, procedure calls between threads are provided only indirectly through system calls or libraries for interprocess communication (IPC). This begs the question of how such libraries might be specified so as to provide procedural abstraction to their clients, and how such libraries can be verified to meet these specifications. In this paper, we consider the problem in the context of multithreaded C software, with threads executing in a single shared address space.

To see why this problem is nontrivial, consider a simple implementation where all data is passed through shared memory, and where each ordered pair of caller-callee threads share a mailbox at a fixed address. The caller makes a call by creating a suitable call record in memory (including identification of which procedure to execute, values of the call parameters, and a place to put the return value), writes the address of this record into the mailbox going to the callee, and calls an IPC function to signal the callee. The callee, on receiving the signal, reads the address of the call record from the mailbox, reads the memory to get the call parameters, executes the call, and signals the caller. Note that all memory accesses are sequential; the only synchronization necessary is provided by the IPC layer.

Now, it's not hard to see that the IPC layer is providing functionality similar to a split binary semaphore, with the call records playing the role of the lock-protected data, and the data invariant given by the semantics of the various procedure calls. Thus, a specification for semaphores would provide a natural starting point for a specification for IPC. However, in classical program verification, semaphore operations are specified by their effect on global ghost state; making use of such a specification requires additional global invariants to capture how the clients use each semaphore. Using this kind of specification for IPC would force the client of the remote procedure call to use these global invariants on both call and return. This fails to faithfully capture the local character of procedural abstraction.

A second possibility is to encapsulate these global invariants inside the IPC layer. For example, the IPC specification could be strengthened to include the pre- and post-conditions of the procedure call. This is the sort of specification one would find in a local logic, such as concurrent separation logic (CSL). But such logics typically cannot specify generic semaphores, because the semaphore code has to be polymorphic in both the encapsulated data and the data invariant.[1]

[1] A recent proposal [DDG^{+}10] extends CSL with a facility similar to VCC ghost objects, which should allow to do constructions similar to the one in this paper.

Similarly, taking this approach with the IPC code requires the specification of the code to be polymorphic in the specification of the material being passed between caller and callee.

We propose a different approach to specifying and verifying IPC that allows the recovery of procedural abstraction. The key idea is that IPC routines transfer ghost objects that own the call records, and whose invariants capture the pre- and post-conditions of the procedures. (This is possible because we allow object invariants to mention arbitrary parts of the state, with a semantic consistency check that guarantees the stability of each object invariant while the object exists.) The "contract" between caller and callee is expressed in ghost data as a binary relation between call objects and return objects. The IPC routines can transfer ownership of the ghost objects without knowing their types, making the transport suitably polymorphic. This ghost scaffolding, combined with the (fixed) specification of the IPC routines, yields for the client the sequential procedural abstraction provided by the application function.

We have used this approach to specify and verify an IPC layer, and illustrate its application to a multiprocessor flush algorithm. The implementation was derived from a real verification target, the inter-processor interrupt (IPI) routines of Microsoft's Hyper-V^{TM} hypervisor. All specification and proofs given here have been carried out using VCC, an automatic verifier for (suitably annotated) concurrent C.[2] VCC provides the first-class ghost objects needed to carry out our approach, while allowing the approach to be applied to real implementation code.

8.2 Related Work

Related Work - IPC The correctness of IPC has been tackled in the context of microkernel verification. For example, the IPC implementations of the seL4 [KEH$^+$09] and VAMOS [DDW09] kernels have been formally verified against their respective ABIs. These projects focused on implementation correctness rather than client usability, and specify solely data transfer.

The application of VAMOS IPC provided in [ABP09] shows the shortcomings of this approach: there, correctness statements of the remote procedure calling (RPC) library argue simultaneously on the sender/receiver pair instead of using thread-local reasoning.

A number of formalisms were applied to specification and verification of interprocess communication in the context of the RPC-Memory Specification Case Study [BMS96]. None of the submitted solutions attempted to provide general-purpose sequential procedural abstraction.

In [FSGD09] a verification framework for threads and interrupt handlers based on CSL is described. This work is similar to ours, as both the implementation of (thread-switching) primitives and clients using them, are verified. When threads switch, ownership is transfered and some global invariant on shared data

[2]Sources are available at `http://www.verisoftxt.de/PublicationPage.html`.

is checked. In contrast to our work the client code is interactively verified in two different logics, whereas in our approach both are verified seamlessly and automatically in the same proof context.

Overview The paper is structured as follows. In Section 8.3 we outline main VCC concepts. In Section 8.4 we present an IPC algorithm with polymorphic specification, which we use in Section 8.5 to implement and verify a TLB flush protocol. In Section 8.6 we extend these results to multiple senders and receivers as required for the implementation of interprocessor interrupt (IPI) protocols used in real, multiprocessor hypervisors. In Section 8.7 we conclude.

8.3 VCC Overview

In this section, we give a brief overview of VCC. More detailed information and references can be found through the VCC homepage [Mic09]. To understand the VCC view of the world, it is helpful to think of verification in a pure object model, which is used to interpret the C memory state. Thus, we first describe VCC concepts in terms of objects, and then describe how this is applied to C.

Table 8.1 shows a syntax overview of the constructs required for our IPC design presented in the following sections.

Objects In VCC, the state is partitioned into a collection of objects, each with a number of fields. Objects have addresses, so fields can be (typed) object references. Each object has a 2-state invariant, which is expected to hold over any state transition. These invariants can mention arbitrary parts of the state. However, when checking an atomic update to the state, instead of checking the invariants of *all* objects we want to check the invariants of only the updated objects. We justify this by checking, for each object type, that starting from a state in which all object invariants hold, a transition that breaks the invariant of an object of that type must break the invariant of some modified object (not necessarily of that type); such invariants are said to be *admissible*. (In addition, we have to check that stuttering from the poststate of a transition preserves all invariants of all objects.) Both requirements are checked for each object type when the type is defined; this check makes use of type definitions, but not of program code. Details can be found in [CMST10].

Within an object invariant, the (2-state) invariant of other objects can be referred to.[3] A commonly used form of this is *approval*: we say that an object *o* *approves* changes to another object's field *p*→*f*, if *p* has a 2-state invariant stating that *p*→*f* stays unchanged or the invariant of *o* holds. In other words, any change to *p*→*f* requires checking the invariant of *o*. Approval is used to express object dependencies or build object hierarchies, e.g., VCC's ownership model.

[3]This implicitly makes object invariants recursive; to guarantee that all object invariants have a consistent interpretation, we allow such references to occur only with positive polarity.

VCC Keyword	**Description**
Basics	
me	reference to current thread
this	self-reference to object (used in type invariants)
invariant(p)	type invariant with property p
old(e)	evaluates e in prestate (of function, loop, or 2-state invariant)
closed(o)	object o closed; invariants of o guaranteed to hold
inv(o)	evaluates to (2-state) invariant of o
approves$(o, f_1, \ldots, f_n)$	changes of fields $f_1, \ldots, f_n$ require check of o's invariant: $(\bigvee_i$ **old**$(f_i) \neq f_i) \implies$ **inv**(o)
atomic$(o_1, \ldots, o_n)\{s;\}$	marks atomic execution of s; updates only volatile fields of $o_1, \ldots, o_n$
ref_cnt(o)	number of claims that depend on o
claims(c,p)	invariant of claim c implies p
spec$(\ldots)$	wraps ghost code and parameters
$\forall(T\ t; \ldots)$	universal quantification
$\exists(T\ t; \ldots)$	existential quantification
Ownership	
owner(o)	owner of object o
owns(o)	set of objects owned by object o
wrapped(o)	o closed and owned by current thread
mutable(o)	o not closed and owned by current thread
set_closed_owner(o,o')	sets owner of o to o' and extends ownership of o' by o
giveup_closed_owner(o,o')	make o wrapped and remove it from the ownership of o'
Function Contracts	
requires(p)	precondition
ensures(p)	postcondition
writes$(o_1, \ldots, o_n)$	function writes to objects o_i
Spec Types	
mathint	mathematical integers
claim_t	claim pointers
T2 map[T1]	map from *T1* to *T2*
λ*(T1 t1; ...)*	lambda expression over *t1*

Table 8.1: VCC Keywords

Since it is unrealistic to expect objects to satisfy interesting invariants always (e.g., before initialization or during destruction), we add to each object a Boolean ghost field **closed** indicating whether the object is in a "valid" state. Implicitly, the 2-state invariants declared with an object type are meant to hold only across state transitions in which the object is closed in the prestate and/or the poststate. Each object field is classified as either sequential or volatile. Volatile fields can change while the object is closed, while sequential fields cannot. (That is, for each sequential field, there is an implicit object invariant that says that the field does not change while the object is closed.)

Each object has an *owner*, which is itself an object. It is a global system invariant that open objects are owned only by threads, which are regular objects. In the context of a thread *t*, a closed object owned by *t* is said to be *wrapped*, while an open object owned by *t* is said to be *mutable*. Threads themselves have invariants; essentially, the invariant of a thread *t* says that any transition that does not change the state of *t* leaves unchanged 1. the set of objects owned by *t*, 2. the fields of its mutable objects, 3. the sequential fields of its wrapped objects, and 4. the (volatile) fields of closed objects approved by *t* (we call such fields *thread-approved*). Each object *o* implicitly contains an invariant that says that its owner (as well as its owner in the prestate) approves any change to the field *o*→**closed** and to the set of objects owned by *o*.[4]

The *sequential domain* of a closed object is the smallest set of object fields that includes the sequential fields of the object and, if its set of owned objects is declared as nonvolatile, the elements of the sequential domains of the objects that it owns. Intuitively, the values of fields in the sequential domain of *o* are guaranteed not to change as long as *o* remains closed.

Within program code, each memory access is classified as ordinary or atomic. An ordinary write is allowed only to fields of mutable objects; an ordinary read is allowed only to fields of mutable objects, to nonvolatile fields of in the sequential domain of a wrapped object, and to volatile fields of objects that are closed if changes to the field are approved by the reading thread. In an atomic operation, all of the objects accessed have to be known to be mutable or closed (i.e., not open and owned by some other thread), only volatile fields of closed objects may be written, and the update must be shown to preserve the invariants of all updated objects. Before each atomic operation, VCC simulates running other threads by forgetting everything it knows about the state outside of its sequential domain; standard reduction techniques [CL98] can be used to show that we can soundly ignore scheduler boundaries at other locations.

Ghost Objects VCC verifications make heavy use of *ghost* data and code (surrounded by **spec***()*), used for reasoning about the program but omitted from concrete implementation. VCC provides ghost objects, ghost fields of structured

[4]By default, the set of objects owned by *o* is nonvolatile, and so cannot change while *o* is closed. This can be overridden by declaring **vcc***(volatile_owns)* in the type definition of *o*.

data types, local ghost variables, ghost function parameters, and ghost code. C data types are limited to those that can be implemented with bit strings of fixed length, but ghost data can use additional mathematical data types, e.g., mathematical integers (**mathint**) and maps. VCC checks that information cannot flow from ghost data or code to non-ghost state, and that all ghost code terminates; these checks guarantee that program execution including ghost code simulates the program with the ghost data and ghost code removed.

Claims A ghost object can be used as a first-class chunk of *knowledge* about the state, because the invariant of the object is guaranteed to hold as long as the object is closed. In particular, the owner of the object does not have to worry about the object being opened by the actions of others, so it can make use of the object invariant whenever it needs it. Being a first-class object, the chunk can be stored in data structures, passed in and out of functions, transfered from thread to thread, etc. Because they are so useful, VCC provides syntactic support for these chunks of knowledge, in the form of *claims*. Claims are similar to counting read permissions in separation logic [BCOP05], but are first-class objects; this allows claims to approve changes, be claimed, or even claim things about themselves.

Typically, a claim depends on certain other objects being closed; it is said to "claim" these objects. Since objects are usually designed to be opened up eventually, these "claimed" objects must be prevented from opening up as long as the claim is closed. Concretely, this can be implemented in various ways, the most obvious being for the dependee to track the count **ref_cnt***(o)* of claims that claim *o*, and allowing *o* to be opened only when **ref_cnt***(o)* is zero, cf. [CMST10]. In constructing a claim, the user provides the set of claimed objects and invariant of the claim; VCC checks that this invariant holds and is preserved by transitions under the assumption that the claimed objects are closed (this check corresponds to the admissibility check if the claim was declared with an explicit type). Any predicate implied by this invariant is said to be "claimed" by the claim; this allows a client needing a claim guaranteeing a particular fact to use any claim that claims this fact (without having to know the type of the claim); to make this convenient, VCC gives all claims the same type (**claim_t**); we can think of an additional "subtype" field as indicating the precise invariant.

Function Contracts and Framing Verification in VCC is *function-modular*; when reasoning about a function call, VCC uses the specification of the function, rather than the function body. A function specification consists of preconditions (of the form **requires***(p)*), postconditions (of the form **ensures***(p)*, where *p* is a 2-state predicate, the prestate referring to the state on function entry), and writes clauses (of the form **writes***(o)*, where *o* is an object reference or a set of object references). VCC generates appropriate verification conditions to make sure that the writes clauses are not violated.

Binding to C The discussion above assumed that we are in a world of unaliased objects. To deal with the real C memory state, VCC maintains in ghost state a global variable called the *typestate* that keeps track of where the "real" objects are; these objects correspond to instances of C aggregate types (structs and unions). (Variables of primitive types that are not fields of such objects are put into artificial ghost objects or ghost arrays.) There are system invariants that 1. each memory cell is part of exactly one object in the typestate, 2. if a struct is in the typestate, then each of its subobjects (e.g., fields of aggregate type) are in the typestate, and 3. if a union is in the typestate, then exactly one of its subobjects is in the typestate. These invariants guarantee that if two objects overlap, then they are either identical or one object is a descendant of the other in the object hierarchy. When an object reference is used (other than as the target of an assignment), it is asserted that reference points to an object in the typestate. Thus, the typestate gets rid of all of the "uninteresting" aliasing (like objects of the same type partially overlapping).

8.4 A Polymorphic Specification of IPC

In this section we verify the implementation of a simple communication algorithm between two threads. The threads exchange data over a shared but sequentially accessed message box to which they synchronize access with a Boolean volatile notification flag. To verify the implementation's memory safety, an ownership discipline must be realized in which the ownership of the message box is transferred back and forth between the two threads. We extend this pattern by passing claims between the two threads, which we store in the message box. The properties of these claims can be configured by the clients, thus providing the desired polymorphic procedure call semantics for IPC.

There are various ways to structure annotations and, in particular, the definitions of ghost objects and invariants. At their core, all of these share information via volatile fields, pass on knowledge via claims or object invariants, and make use of thread-approved state for the two communication partners. We chose here a way that is easy to present but also extends cleanly to multiple senders and receivers (cf. Section 8.6).

Scenario We consider the scenario of two threads (0 and 1) exchanging data over a shared message box (of type *MsgBox*). The message box contains two fields (**in** and **out**) which are used for sending a request to the other thread and receiving back a response, respectively. The fields of the message box are nonvolatile and accessed sequentially. The message box is contained in another structure (of type *Mgr*), which additionally holds a volatile Boolean notification flag *n* used to synchronize access to the message box. Given the canonical conversion of Booleans to integers (where *0* and *1* are mapped to **false** and **true**, respectively), this flag identifies the currently acting thread. If set, thread 1 is acting, i.e.,

```
spec(typedef struct vcc(record) InOut {
  unsigned val; mathint gval; claim_t cl; } InOut;)

typedef struct MsgBox {
  unsigned in, out;
  spec(InOut input, output;)
  invariant(input.val≡ in ∧ output.val≡ out)
  invariant(input.cl≠ output.cl ∧
    input.cl ∈ owns(this) ∧ ref_cnt(input.cl)≡ 0 ∧
    output.cl ∈ owns(this) ∧ ref_cnt(output.cl)≡ 0) } MsgBox;
```

Listing 1: Message Box Type with Invariants

preparing a response for thread 0 and posting a new request, and thread 0 may not access the message box. Otherwise, thread 0 is acting and thread 1 may not access the message box. Thread 0 may not clear the flag, and thread 1 may not set it.

The implementation has two functions. Both take a *Mgr* pointer and a thread identifier *a*. The function *snd()* is meant to be called by thread *a* when the notification flag equals *a*. It negates the notification flag, thus sending the response and a new request contained in the message box at that time to the other thread. The function *rcv()* waits in a busy loop until the notification flag equals *a* again, thus receiving the other thread's response (to a preceding *snd()* call) and a new request.

Message Box Listing 1 shows the annotated definition of the message box type. As outlined above, we want to generalize information exchange to beyond the mere transferral of data (the fields **in** and **out** in the message box). We therefore define an abstract I/O type (*InOut*) that carries a ghost value *gval* of unbounded integer type, and a claim pointer **cl** in addition to the implementation data value *val* being transmitted.

An abstract input and output each are an invariant stored in ghost fields of the message box. We maintain an invariant that the input's and output's *val* fields match their implementation counterpart. We also require that the claims pointed to by the input's and output's **cl** fields do not alias and are owned by the message box with a zero reference count.

The latter fact is particularly important. Whoever owns the message box also controls the contained claims, and may make use of the knowledge / property they hold or destroy them. The main functionality of the verified algorithm is thus the transferal of ownership of the message box between the two threads, making sure that the contained data has the desired properties, as instantiated by the client.

Actors The *Actor* type keeps track of the protocol state of a protocol participant. Listing 2 shows the annotated definition of this type. The actor has a nonvolatile pointer *mgr* to the manager, which will hold all protocol invariants.

```
spec(typedef struct vcc(volatile_owns) Actor {
  struct Mgr ∗mgr;
  volatile bool w;
  volatile InOut l_input, r_input;
  invariant(closed(this) ∨ ¬closed(mgr))
  invariant(approves(owner(this), w, l_input, r_input))
  invariant(approves(mgr, owns(this), w, l_input, r_input)) } Actor;)
```

Listing 2: Actor Type and Invariants

```
typedef struct Mgr {
  volatile bool n;
  MsgBox msgBox;
  spec(Actor A[2];
    bool InP[bool][InOut];
    bool OutP[bool][InOut][InOut];)
  invariant(∀(unsigned a; a < 2 ⟹ closed(&A[a]) ∧ A[a].mgr≡ this))
  invariant(A[¬n].w)
  invariant(A[n].w
    ? &msgBox ∈ owns(&A[n]) ∧ OutP[n][A[n].l_input][msgBox.output] ∧
      A[¬n].l_input≡ msgBox.input ∧ InP[n][msgBox.input]
    : A[n].r_input≡ A[¬n].l_input) } Mgr;
```

Listing 3: Manager Type and Invariants

For admissibility reasons, the actor must promise to stay closed longer than *mgr*. All others fields are volatile and may be atomically updated while the actor remains closed. Such updates, however, must be approved by two parties: the manager *mgr*, which checks all the protocol invariants, and the owner of the actor, which is one of the communicating threads and exclusive writer of the fields. The actor is also used as an intermediate owner of the message box during ownership transferral. For this purpose, its owns set is also declared volatile as well as approved by *mgr* but *not* thread-approved, to enable foreign updates by other threads.

The three regular fields of the actor are used as follows. The wait flag *w* is active when the thread owning the actor is waiting for a response from the other thread. The fields *l_input* and *r_input* buffer (abstract) local and remote inputs, i.e., input to the last request sent to or received from the other thread (or, in other words: the evaluation of the call parameters from the caller's and callee's perspective, respectively). In contrast to the *input* fields of the message box itself, which may be opened and updated sequentially by the owning thread, these buffers can be admissibly referred to all the time and used in the protocol invariants.

Manager Listing 3 shows the annotated declaration of the *Mgr* type. In addition to the implementation fields, we also add some ghost components for the verification: the maps *InP* and *OutP* encoding pre- and postconditions for the message exchange, and a two-element array *A* of actors.

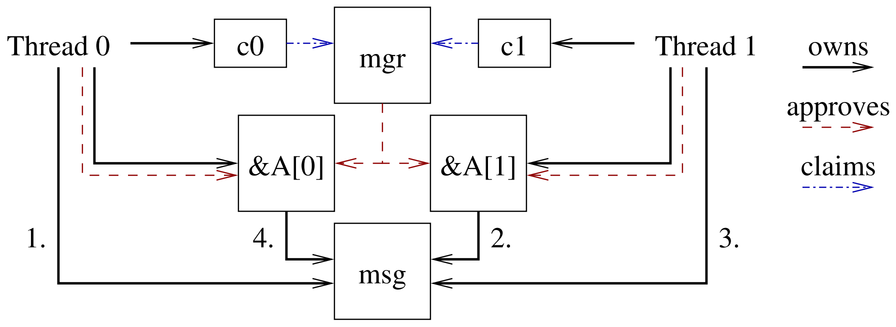

Figure 8.1: Object Structure and Ownership Transfer

The predicates are declared nonvolatile, which allows clients to deduce that they remain unchanged as long as the manager object is closed. They take a Boolean parameter identifying the actor and one resp. two abstract input-output values. The intention is that *InP[a][i]* is true iff *i* is a valid request for thread *a* (i.e., if the request meets the precondition of the service), and *OutP[a][i][o]* is true iff *o* is a valid response to a (valid) request *i* made by thread *a* (i.e., if the response meets the postcondition corresponding to the call). The IPI transport code is polymorphic with respect to these predicates, the concrete definition of which can be provided by the client at initialization.

We now describe the manager's invariants. As described above, each protocol partner *a* owns its corresponding actor *&A[a]*. The first invariant states that both actors remain closed and point back to the manager, which (in combination with the actor's approval invariants) allows us to admissibly talk about the actors in invariants here.

The remaining invariants define the protocol behavior. For an overview, refer to Figure 8.1 depicting object structure and a protocol run starting from thread 0. In addition to the objects already introduced, each (client) thread i owns a claim ci that guarantees the manager structure to be closed. In phase 1, thread 0 owns the message box and may prepare its response and new request. In phase 2, ownership of the message box has passed from thread 0 to the actor of thread 1, waiting to be processed. Phases 3 and 4 are symmetrical: in phase 3 thread 1 prepares its response, which is then waiting to be processed in phase 4.

In addition to ownership, the protocol invariants restrict values for the actor fields. The second invariant states the non-acting thread, identified by the negated notification flag, must be waiting, i.e., have the wait flag of its actor set.

The third invariant refers to the acting thread, given by the notification flag. The fact that the acting thread is waiting indicates that the message box is still waiting to be processed by the acting thread. It holds a response to the acting thread's last request in the *output* field and a new request in the *input* field. In the corresponding invariant we state that 1. the message box is owned by the current actor, 2. its output is valid with respect to the acting thread's last / locally-stored

```
void snd(struct Mgr *mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(¬mgr→A[a].w)
  requires(wrapped(&mgr→msgBox))
  requires(mgr→OutP[¬a][mgr→A[a].r_input][mgr→msgBox.output])
  requires(mgr→InP[¬a][mgr→msgBox.input])
  writes(&mgr→msgBox,&mgr→A[a])
  ensures(mgr→A[a].l_input≡ old(mgr→msgBox.input))
  ensures(mgr→A[a].w)
{
  atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
    assert(¬mgr→A[a].w ∧ mgr→A[¬a].w ∧ mgr→n≡ a);
    mgr→n = ¬a;
    spec(mgr→A[a].l_input = mgr→msgBox.input;
      mgr→A[a].w = true;
      set_closed_owner(&mgr→msgBox, &mgr→A[¬a]);
      bump_vv(&mgr→A[a]); /* technicality */
    )
  }
}
```

Listing 4: Send function with contract

request, and 3. the new input equals the local input buffer of the other thread and is valid for the acting thread. If the acting thread is not waiting, we require local and remote input buffers of the current and non-current actors, respectively, to match. Note that these input buffers are approved by the acting and non-acting threads, respectively. Thus, this condition states that request inputs may not be changed while the request has not yet been processed.

Operations The (annotated) implementation and the contracts for the send and receive function are given in Listings 4 and 5. Both functions take a manager pointer *mgr*, an actor identifier *a*, and a claim *c* supplied as a ghost parameter stating that the manager is closed. They maintain that the identified actor is wrapped. To send to the other thread, the current thread's actor must be flagged as non-waiting, the message box must be wrapped and hold valid outputs and inputs to the other thread, just as we have seen in the manager invariant for the acting thread. Afterwards, the message box is unknown to be wrapped (the **writes** clause on *&mgr→msgBox* destroys that knowledge), but the input sent to the other thread is buffered in the local input field of the actor (and the current thread's actor is flagged as waiting).

Given a waiting actor, the receive function is guaranteed to return a wrapped message box, that contains a valid response for the old local request and a new valid request.

```
void rcv(struct Mgr ∗mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(mgr→A[a].w)
  writes(&mgr→A[a])
  ensures(¬mgr→A[a].w)
  ensures(wrapped(&mgr→msgBox))
  ensures(mgr→OutP[a][old(mgr→A[a].l_input))][mgr→msgBox.output])
  ensures(mgr→A[a].r_input≡ mgr→msgBox.input)
  ensures(mgr→InP[a][mgr→msgBox.input])
{
  unsigned tmp;
  do
    invariant(mgr→A[a].w)
    invariant(wrapped(&mgr→A[a]))
    invariant(mgr→A[a].l_input≡ old(mgr→A[a].l_input))
    atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
      tmp = mgr→n;
      spec(if (tmp≡ a) {
        mgr→A[a].r_input = mgr→A[¬a].l_input;
        mgr→A[a].w = false;
        giveup_closed_owner(&mgr→msgBox, &mgr→A[a]);
        bump_vv(&mgr→A[a]); /∗ technicality ∗/
      })
    }
  while (tmp≠ a);
}
```

Listing 5: Receive function with contract

As a verification example consider the *snd()* function from Listing 4. VCC automatically verifies that its implementation fulfills the contract. The code consists of a single atomic update on the actors and the manager (where the closedness of the manager and the foreign actor is guaranteed by the claim *c*). The precondition on the wait flag, its thread-approval, and the manager's invariant allow to derive that the current thread is still not waiting, the other thread is waiting, and the notification flag equals *a* just before the atomic operation.[5] Also, the message box, which is in the sequential domain of the thread, must still be wrapped and continues to satisfy the communication preconditions. The notification flag is then flipped (changing the 'acting' thread) and the ghost updates ensure that the atomic update satisfies the manager's invariant (e.g., by transferring ownership of the message box from the current thread to the other thread's actor).

The verification of the *rcv()* function is similar. In addition to the atomic statement, appropriate invariants have to specified for the loop that polls on the notification flag.

[5]The assertion is for illustration only; VCC deduces it automatically.

```
spec(typedef struct Tlb {
  volatile mathint invalid, current;
  invariant(invalid ≤ current ∧
    old(invalid) ≤ invalid ∧ old(current) ≤ current)
} Tlb;)
```

Listing 6: TLB Model

8.5 TLB Flush Example

We implement and verify a protocol for flushing translation look-aside buffers (TLBs) based on the communication algorithm from the previous section, demonstrating the expressiveness of its polymorphic specification.

TLBs are per-processor hardware caches for translations from virtual to physical addresses. These translations are defined by page tables stored in memory, which are asynchronously and non-atomically gathered by the TLBs (requiring multiple reads and writes to traverse the page tables). Since translations are not automatically flushed in response to edits to page tables, operating systems must implement procedures to initiate such flushes on their own.

We think of page-table reads being marked with unique (increasing) identifiers and model each TLB as an object with two volatile counters,[6] cf. Listing 6. The *current* counter increases as the TLB gathers new translations. The *invalid* counter is a watermark for invalidated translations and is bumped (i.e., copied from the *current* field) when the associated processor issues a TLB flush.

Consider the scenario of two threads, the *caller* (thread 0) requesting the flush and the *callee* (thread 1) performing the flush. We implement this as follows: the caller sends a flush request by invoking the send primitive and subsequently polls for the answer by calling the receive primitive. On callee side, the thread polls via receive for new flush requests. When a flush request has been received, the callee issues a TLB flush operation, and signals back that the flush has been performed using the send primitive. After a completed flush operation, the flush client (e.g., the memory manager) wants to derive that the callee TLB's current *invalid* counter is larger or equal than the callee's *current* counter at the time of the flush operations.

We realize this scenario by embedding the IPC manager (and callee's TLB) into a *flush manager*, as shown in Listing 7. Apart from ownership, the invariants give meaning to the input and output predicates of the communication manager. The ghost value *i.gval* transmitted from the caller to the callee encodes which translations are meant to be flushed. For the callee ($a \equiv$ **true**), the input predicate states that this value is less or equal than the *current* field of its TLB (since the callee could not possibly flush translation 'from the future', i.e., such a request could not be handled by the TLB flush semantics). For the caller ($a \equiv$ **false**), the

[6]While this model is sufficiently detailed to express the semantics of (full) TLB flushes, extensions are needed for applications that go beyond that.

```
typedef struct FlushMgr {
  struct Mgr mgr;
  spec(struct Tlb tlb;)
  invariant(&tlb ∈ owns(&mgr)))
  invariant(mgr.InP≡ λ(bool a; InOut i;
    a ⟹ claims(i.cl, i.gval ≤ tlb.current)))
  invariant(mgr.OutP≡ λ(bool a; InOut i, o;
    a ∨ claims(o.cl, i.gval ≤ tlb.invalid)))
} FlushMgr;
```

Listing 7: Flush Manager Type and Annotations

output predicate then states that the *invalid* field of the callee's TLB is greater or equal than the value, i.e., the requested flush has been performed. For the other cases the input and output predicates are trivially true.

Based on this definition, the correctness of the functions *sendFlush()* and *receiveFlush()* at caller and callee side, respectively, can be proven. The main postcondition that is established by *sendFlush()* for the flush manager *fmgr* then is **old***(fmgr→tlb.current) ≤ fmgr→tlb.invalid.*

8.6 Interprocessor Interrupts

Interprocessor interrupts (IPI) are used in multicore operating systems or hypervisors to implement different synchronization and communication protocols. Via IPIs a thread executing on one processor can trigger the execution of interrupt handlers (here: NMI handlers) on other processors. Using IPIs, a communication protocol can be implemented, in which a caller thread sends work requests to other processors, the callees. Such an IPI protocol is part of the Verisoft XT academic hypervisor, where it may be used for different work types, e.g., for TLB flushing. Thus a polymorphic specification is desirable.

By expanding the simple communication pattern introduced previously, we specified and verified the IPI protocol (and on top of it a TLB flushing protocol) for the academic hypervisor. There are several differences between the previous version of the algorithm and the IPI protocol:

- **More communication partners.** In the simple case we had a single sender and a single receiver. Now we have multiple communication partners, where one sender may invoke an IPC call on many receivers, and where each receiver may be invoked by many senders at the same time.

- **No receiver polling.** The callees in the IPI scenario do not poll for messages. Rather the caller invokes the callee by triggering an IPI. This is done by writing registers of the advanced programmable interrupt controller (APIC), which delivers the interrupts to other processors. In the work at hand we do not yet model this hardware device.

- **More concurrency.** In the new setting we have another source of concurrency, NMI handlers which may interrupt the execution of ordinary threads. Basically, the NMI handler code always acts as receiver or callee and the thread code as sender or caller.

- **Interlocked hardware operations.** Interlocked bit operations are required to atomically access bit vectors which may be written and read concurrently by many threads/handlers.

Implementation Since multiple senders can send requests to multiple receivers, we need a notification bit for each sender/receiver pair. This is implemented by introducing one notification mask per processor. Each bit of such a mask is associated with a specific sending processor. Thus, a sender signals a request by setting its bit in the receiver's notification mask. When finishing the work, the receiver clears that bit. Many senders and receivers can write the same notification mask in parallel, requiring the use of interlocked bit operations.

Similarly, we need one mailbox for each sender/receiver pair. Note, that for each processor pair we need two mailboxes, since both may send messages to each other simultaneously.

In the sending code a while-loop iterates over the set of intended receivers (encoded in a bit mask). In each iteration, first the mailbox is prepared, and then by using an interlocked OR-operation, atomically, the corresponding bit in the receiver mask is set to *1* and the mask is compared with *0*. If this check evaluates to true, an IPI for the receiving processor is triggered via the APIC. Otherwise, nothing has to be done, since some other sender already triggered the interrupt, and the handler has not returned yet.

In the receiving code (implemented as an NMI handler) a while-loop iterates on (possibly multiple) sender requests as long as the receiver's notification mask is not *0*. Once the work for one sender is done, the corresponding bit in the notification mask is cleared by an interlocked AND-operation.

Specification The specification pattern of Section 8.4 can be straightforwardly applied to the IPI protocol. The number of ghost objects scales linearly with the number of processors. The structure and the invariants of message boxes (with their ghost fields encoding input/output claims) and actors introduced in the simple protocol can be reused almost identically in the new setting.

If n is the number of processors, $2 \cdot n$ actor objects are required, since each processor may act both as sender—when running thread-code—or as receiver—when running NMI handler code. Though executed on the same processor, both code portions are two logically different entities, possibly residing in different protocol states, and owning different sets of mailboxes. That is also how we deal with thread and NMI handler concurrency: each of the NMI handler and the thread code own (and thus approve) separate actors. Note that in the IPI case, a single actor may communicate with many other partners, requiring it to maintain protocol state (the wait flag, and the remote and local input fields) per processor. The invariants of the manager are similar to those from the simple protocol.

Multiprocessor TLB Flush The TLB abstraction and specification is similar to the previous section, but with a separate TLB for each processor.

IPIs in Microsoft's Hyper-V$^{\text{TM}}$ Hypervisor In the context of the Verisoft project we also studied the correctness of the IPI mechanism implemented in Microsoft's Hyper-V hypervisor. Though comparable in complexity to the IPI routine of the academic hypervisor, there are several differences:

- **Efficiency.** By introducing additional protocol variables sequential access to some of the shared data can be ensured, and thus fewer (costly) interlocked operations are required.
- **Lazy work.** The interrupt handler signals the receipt of the request and the accomplishing of the work separately. This allows for implementing less blocking caller code.

We have verified the implementation against a non-generic specification in VCC and are confident that this effort can be easily adapted to the generic specification used here.

8.7 Conclusion

The verification presented here achieves the desired goal— it allows IPC clients to reason about IPCs like local procedure calls. As future work, the structure presented in Section 8.4 can be made modular even with respect to the set of functions provided via IPC. We can improve the structure slightly by changing the *Mgr* type; instead of the maps *InP* and *OutP*, the *Mgr* could hold a mapping of function tags to function objects, where each function object has its own *InP* and *OutP* maps. This would allow function objects to be reused in different managers, or even dynamically registered for IPC.

In principle, the technique presented here could also be applied to RPC, where the caller and callee execute in different address spaces. This requires translating the claims representing the pre- and post-conditions from one address space to the other. One possible way to achieve this effect would be to take the claim in the caller space, couple this to a second state in a way that captures the guarantees of the RPC, and existentially quantify away the caller space.

Index

Bibliography

[ABP09] E. Alkassar, S. Bogan, and W. Paul. Proving the correctness of client/server software. In *Sadhana Journal*, volume 34, pages 145–192. Springer, 2009.

[ACHP10] Eyad Alkassar, Ernie Cohen, Mark A. Hillebrand, and Hristo Pentchev. Modular specification and verification of interprocess communication. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 167–174, 2010.

[Alk09] Eyad Alkassar. *OS Verication Extended - On the Formal Verication of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, University of Saarland, 2009.

[App12] Andrew W. Appel. Verified software toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, page 2, 2012.

[Bau14] Christoph Baumann. *Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems*. PhD thesis, Saarland University, Saarbrücken, 2014.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270, 2005.

[BMS96] Manfred Broy, Stephan Merz, and Katharina Spies, editors. *Formal Systems Specification, The RPC-Memory Specification Case Study (the book grow out of a Dagstuhl Seminar, September 1994)*, volume 1169 of *Lecture Notes in Computer Science*. Springer, 1996.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, pages 317–331, 1998.

[CMST10] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 480–494, 2010.

[DDG$^+$10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 504–528, 2010.

[DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *J. Autom. Reasoning*, 42(2-4):349–388, 2009.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Saarbrücken, 2011.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms – Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, Saarbrücken, 2009.

[FSGD09] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[Heb10] Erik Hebisch. *A survey of multicore in the german software developers community*. Fraunhofer IAO, http://www.ipd.uni-karlsruhe.de/multicore/separs/downloads/2010-12-02-SEPARS-Stuttgart-IAOMWare-Surveys.pdf, December 2010.

[Inc06] The Santa Cruz Operation Inc. System v application binary interface - mips risc processor supplement 3rd edition. Technical report, February 2006.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220, 2009.

[KLM$^+$15] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 229–234, San Jose, CA, USA, 2015. EDA Consortium.

[KMP14] Mikhail Kovalev, Silvia M. Mueller, and Wolfgang J. Paul. *A Pipelined Multi-core MIPS Machine - Hardware Implementation and Correctness Proof*. Springer, 2014.

[Kov13] Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, 2013.

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardware Design: Formaler Entwurf digitaler Schaltungen*. Teubner, 1995.

[Lei08] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., 1st edition, 1999.

[LP08] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08).*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 23–40. Elsevier Science B. V., 2008.

[LPP05a] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 2005.

[LPP05b] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005.

[LT93] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[Mic09] Microsoft Corporation., http://vcc.codeplex.com/. *VCC: A C Verifier.*, 2009.

[MP00] Silvia M.Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[Pau07] Wolfgang Paul. System Architecture (Lecture Notes). URL: `http://busserver.cs.uni-sb.de/lehre/vorlesung/info2/ss08/material/mitschrift07.pdf`, 2007.

[PBLS16] Wolfgang Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. System architecture as an ordinary engineering discipline. URL: `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur/ws15/books/sysbook.pdf`, 2016.

[Pra95] Vaughan R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 97–107, 1995.

[PSS12] Wolfgang Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5 (Software Engineering and Formal Methods)*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer Berlin Heidelberg, 2012.

[Sch13] Sabine Schmaltz. *Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C*. PhD thesis, Saarland University, Saarbrücken, 2013.

[Sha12] Andrey Shadrin. *Mixed Low- and High Level Programming Languages Semantics. Automated Verification of a Small Hypervisor: Putting It All Together.* PhD thesis, Saarland University, Saarbrücken, 2012.

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *4th International Conference on Veried Software: Theories, Tools, and Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, Philadelphia, USA, 2012. Springer Berlin / Heidelberg.

[Sta] Statista GmbH, http://www.statista.com.

[Sta10] Artem Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, SaarbrÃ$\frac{1}{4}$cken, 2010.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken, 2009.

[Ver] Verisoft Consortium., http://www.verisoft.de. *Verisoft Project*.