Store Buffer Reduction Theorem and Application

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

vorgelegt von

Geng Chen

Saarbrücken, Mai 2016

UNIVERSITÄT DES SAARLANDES

Institut für Rechnerarchitektur und Parallelrechner,
Universität des Saarlandes, 66123 Saarbrücken

Tag des Kolloquiums	11. Mai 2016
Dekan	Prof. Dr. Frank-Olaf Schreyer
Prüfungsausschuss	
Vorsitz	Prof. Dr. Antonio Krüger
1. Gutachter	Prof. Dr. Wolfgang J. Paul
2. Gutachter	Prof. Dr. Andreas Podelski
Akademischer Mitarbeiter	Dr. Qanru Sun

Abstract

Short Abstract

The functional correctness of multicore systems can be shown through pervasive formal verification, which proves the simulation between the system software computation and the corresponding hardware computation. In the implementation of the system software, the sequential consistency (SC) of memory is usually assumed by the system programmers. However, most modern processors (x86, Sparc) provide the total store order (TSO) memory model for greater efficiency. A store buffer reduction theorem was presented by Cohen and Schirmer [CS10a] to bridge the gap between the SC and the TSO. Nevertheless, the theorem is not applicable to programs that edit their own page tables. The reason is that the MMU can be treated neither as a part of the program thread nor as a separate thread. This thesis contributes to generalize the Cohen-Schirmer reduction theorem by adding the MMUs.

As the first contribution of this thesis, we present a programming discipline which guarantees sequential consistency for the TSO machine with MMUs. Under this programming discipline, we prove the store buffer reduction theorem with MMUs.

For the second contribution of this thesis, we apply the theorem to the ISA level and the C level. By proving a series of simulation theorems, we apply our store buffer reduction theorem with MMU to the ISA named MIPS-86. After that, we introduce the multicore compiler correctness theorem to map the programming discipline to the parallel C level.

Kurzzusammenfassung

Die funktionale Korrektheit von Mehrkern-Systemen kann durch durchgängige formale Verifikation sichergestellt werden, in welcher die Simulation zwischen Berechnungen der Systemsoftware und der entsprechenden Hardwareberechnungen bewiesen wird. Für die Implementierung der Systemsoftware wird vom Systemprogrammierer im Normalfall das Berechnungsmodell der Sequentiellen Konsistenz (SC) zugrundegelegt. Die meisten modernen Prozessoren (x86, Sparc) bieten jedoch aus Effizienzgründen stattdessen das Berechnungsmodell der Totalen-Schreibzugriff-Ordnung (TSO) an. Cohen und Schirmer [CS10a] präsentieren ein Schreibpufferreduktionstheorem, welches die Lücke zwischen SC und TSO schließt. Dieses Theorem kann allerdings nicht auf Programme angewendet werden, die ihre eigenen Seitentabellen bearbeiten. Der Grund dafür ist, dass die Speicherverwaltungseinkeit (SVE) weder als Teil des Programmfadens noch als separater Faden behandelt werden kann. Diese Dissertation liefert einen Beitrag zur Verallgemeinerung des Cohen-Schirmer Reduktionstheorems, in dem die SVE hinzugenommen wird.

Als ersten Beitrag dieser Dissertation präsentieren wir eine Programmierdisziplin welche Sequentielle Konsistenz auf einer TSO Maschine mit SVE garantiert. Unter dieser Programmierdisziplin beweisen wir das Schreibpufferreduktionstheorem mit SVE.

Als zweiten Beitrag dieser Dissertation wenden wir das Theorem auf der Ebene der Befehlssatzarchitektur und der C Ebene an. Durch eine Reihe von Simulationstheoremen wenden wir unser Schreibpufferreduktionstheorem mit SVE auf die Befehlssatzarchitektur MIPS-86

an. Danach führen wir ein Mehrkern-Compiler Korrektheitstheorem ein, welches die Programmierdisziplin auf die Ebene von parallelem C abbildet.

Acknowledgements

First and foremost, I would like to thank Prof. Wolfgang Paul for offering me an opportunity to study and work with top scientists and patiently advising this thesis. I learned a lot from Prof. Paul's "karate style" of teaching. Working in Prof. Paul's chair is a valuable and unforgettable experience in my life.

I would like to thank all my past and present colleagues. I am especially indebted to Dr. Mikhail Kovalev, as a co-author of the SB reduction theorem who helped me a lot in writing the long paper-and-pencil proof.

I am also very grateful to my parents for supporting me during all these years. Last but not least, thank the Chinese government for offering me the scholarship.

Saarbrücken, December 16th, 2015 Geng Chen

Contents

1 Introduction

Sequential consistency (SC) [Lam79] is an intuitive and widely used memory model in parallel programming. However, processor designers often apply hardware optimizations for higher performance. A common optimization illustrated in Fig 1.1 is to introduce a FIFO store buffer (SB) between the processor and the shared memory system. When the processor executes a store instruction, the store first enters the SB. This store is visible to other processors only after it exits the SB and is applied to the shared memory. For greater efficiency, loads forward from the most recent store of the same address in the SB if possible. This kind of memory model is called total store order (TSO) because each processor sees the same global ordering of stores. In the following example, we will present that the TSO execution violates the sequential consistency. Initially, $a1$ and $a2$ are both 0.

```
T1: a1:=1                          T2: a2:=1
    if(a2==0)                          if(a1==0)
      critical section                   critical section
```

In an SC execution, only one thread is allowed to enter the critical section. However, under TSO, if the updating of `a1` and `a2` both reside in their SBs then both threads are allowed to enter the critical section.

Another complication arises when we consider the programs that modify the page tables. In this case, the memory management units (MMU) are visible and race with processors. The MMU can not be modeled as a processor because:

- it communicates with the processor via the Translation Lookaside Buffer (TLB) which is a local component of the processor. However, processors communicate with each other through shared memory.

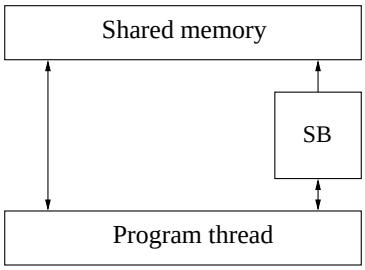


Figure 1.1: Abstract view of TSO.

- it bypasses the SB to access the memory directly.

The following example is from our paper [CCK14]. It shows the presence of MMU violates the SC. Assuming the page table entry (PTE) `pte1` points to the PTE `pte2`, the present bit in both entries is set, and the access bit of `pte1` is `0`. `t0` and `t1` are read temporaries in T1 and MMU1 respectively.

T1:	MMU1:
1: `pte2.p:=0`	3: `pte1.a:=1`
2: `t0:=pte1.a`	4: `t1:=pte2`

Consider a TSO execution where the steps of the T1 are executed before the steps of the MMU1, and the write to `pte2` resides in the SB. After the execution, `t0` is 0 and the MMU reads `pte2` with the present bit set. As a result, the MMU gets an address translation that goes through `pte1` and `pte2`. However, such a TSO execution can not be reproduced under SC. In an SC execution, if we step the MMU1 before T1, the execution ends with `t0` = 1. However, if we step the T1 before MMU1, since the present bit of `pte2` is 0 and can not be used for address translation, the execution ends with a page fault. To find an SC execution which ends with `t0` equals to 0 and `pte2` is 1, the statement 4 needs to be executed before the statement 1 and the statement 2 needs to be executed before the statement 3 while maintaining the programming order. It is impossible to find such an SC execution.

The problems presented above create a gap in multicore system verification. As stated in [CPS13], the correctness theorem of the multicore system includes a simulation theorem between the system implementation and the execution as well as the functional correctness of the system implementation. Most multicore systems are implemented in concurrent C plus assembly code. The multicore compiler correctness theorem in [Bau14] gives the simulation between the implementation and a simplified instruction set architecture (ISA) execution. In the simplified ISA, architectural details like MMUs and SBs should be transparent, and concurrent programs should see sequentially consistent memory. We call this kind of ISA ISA-u (the user's perspective of ISA). During the execution, the complied code runs on an ISA with MMUs and SBs, which we call ISA-sp (the system programmer's perspective of ISA). A TSO memory is provided by ISA-sp. Based on the previous arguments, there exists a gap in the simulation between ISA-u and ISA-sp.

The main goal of this thesis is to find a simulation theorem between ISA-u and ISA-sp. We made the following contributions:

- Propose a programming discipline that guarantees SC for TSO machine with MMU.
- Under the programming discipline, prove a simulation theorem between TSO machine with MMU and SC machine with MMU.
- Apply the SB reduction theorem with MMU to an ISA named MIPS-86.
- Map the programming discipline to parallel C level for user programs.

Chapter 2 of this thesis is a joint work with Ernie Cohen and Mikhail Kovalev. Ernie Cohen and Mikhail Kovalev contributed to building the programming discipline, the machine models

and extending the ownership theorem in [CS10a]. Mikhail Kovalev and the author of this thesis extracted the full paper-and-pencil proof of the SB reduction theorem in [CS10a] from the Isabelle code. The proof in Chapter 2 is largely based on that proof. To adapt to other works in this thesis, the author also changed the notations by adding an explicit ownership generation function to the model and attaching the ownership annotations to volatile operations and read modify writes.

Our initial goal was not only to map the programming discipline for the parallel C user programs but also for the system program that is written in parallel C and modifies the page table. However, currently, we do not have the multicore compiler correctness theorem with MMU. As a consequence, we regard the mapping of programming discipline to system code as our possible future work.

1.1 Related Work

1.1.1 System Software Formal Verification

A survey of operating system verification [Kle09] was published by Klein. The first attempt in pervasive system verification is the CLI stack project [BHMY89]. Since the correct execution of a program depends on the correct translation between high-level language and the machine code, several system components were verified: a code generator for a high-level language, an assembler and linking loader, an operating system kernel, and a microprocessor design.

The Verisoft [Ver07] and VerisoftXT [Ver10] aims at formally verify of an entire computer system from the hardware level up the application software level. [HP08] gives an overview of the verification technology and approach. For the hardware, in [BJK+06] Beyer et al. designed and functionally verified a sequential processor named VAMP. The verification had been carried out in the theorem proving system PVS [ORS92]. In order to bridge the gap between the software and the verified hardware in the Verisoft project, in [LP08] Leinenbach et al. implemented and formally verified a non-optimizing C0 compiler which supports mixing inline assembly with C0 code. In [GHLP05a, APST10], Paul et al. implemented and formally verified a generic operating system kernel called CVM (Communicating Virtual Machines) which formalizes concurrent user processes interacting with an operating system kernel. According to [Kle09], the Verisoft projects demonstrated the most comprehensive and detailed implementation correctness statement of the system software. They made substantial progresses in pervasive verification.

The L4.verified project [KEH+09] focuses on a machine-checked verification of the seL4 microkernel [EKD+07] from an abstract specification down to its C implementation. The goal of the project is to formally guarantee the functional correctness of the C implementation, which means the implementation always fulfills the specification. Instead of the pervasive verification, the correctness of compiler, assembly code, and hardware are assumed.

1.1.2 Weak Memory Model

The sequential consistent (SC) memory model as the most intuitive memory model for a multicore machine was defined in [Lam79] by Lamport. After that, much research has been done in the field of memory models. The survey paper [AG96] focuses on consistency models proposed

for hardware-based shared memory systems. It also describes the models in terms of program behavior. One of the memory model presented in [AG96] is the TSO model. The TSO model was first introduced in [WG94] SPARC V8 processor. In [OSS09], Owens et al. formally described a TSO memory model for x86 named x86-TSO. However, their model did not cover the MMU and the page-table changes.

[CS10b] is the starting point of this thesis. In [CS10b], Cohen et al. presented a programming discipline for concurrent programs and have formally proven that it ensures sequential consistency on TSO machines. Instead of applying lock-based techniques, they classify the memory with the ownership sets (shared, read-only and owned). The store buffer should be flushed between a shared write and a subsequent shared read. Based on the above programming discipline, they proved the simulation theorem between TSO computations and the SC computations in Isabelle [NPW02].

[GMY12] considers a way to reason formally about the interoperability between a data-race free (DRF) client and a library written for the TSO memory model. They provide a simulation relation named TSO-to-SC linearizability to fix the correspondence between the TSO execution and an SC execution of the library. They also proved that the properties of a client are preserved by replacing the TSO library with a TSO-to-SC linearized SC library. In order to get the linearized SC library, the shared variable reads and flushes of the SB are required to be protected by locks [1] which introduce more SB flushes than the programming discipline in [CS10b]. At the end of the paper, they proved a more flexible rule that if a program is quadrangular-race free (QRF) then it is sequentially consistent. The QRF requires fewer SB flushes than our programming discipline, but they did not consider the MMU. Also, the QRF can not be used to simplify establishing TSO-to-SC linearizability, because transforming a QRF TSO trace into an SC one can break the linearizability.

Oberhauser [Obe] improved the programming discipline in [CS10b] to avoid the unnecessary SB flush in the following case: let x be a shared address then

T1: `store` x	T2: `store` x
`load` x	

Oberhauser also gives a short proof (less than 30 pages without dealing the MMU). At the end of [Obe], Oberhauser gives a sketch on how to treat MMUs.

1.2 Outline

Note that in this thesis, we introduce four kinds of ISAs.

- First, to apply the SB reduction theorem with MMU to ISA level, we introduce an ISA named MIPS-86, which is a MIPS core extended with x86/x64 like architecture features (with MMU and SB).
- After applying the SB reduction theorem with MMU to MIPS-86, we get an ISA without SB but with MMU. We call it the SB reduced MIPS-86.

[1] The shared variable reads and flushes occur within a *lock..unlock* block. The *lock* suspends other CPU's execution until the *unlock* command and the *unlock* flushes the SB.

- When we apply the SB reduction theorem to parallel C level for user programs, we first apply the SB reduction theorem to ISA level, then apply the multicore compiler correctness theorem to map the programming discipline to the parallel C level. Since, the MMU and interrupts are invisible to user programs, we need an ISA without MMU and interrupts. We call it SB MIPS, which is MIPS-86 without MMU and interrupts.

- After applying the SB reduction theorem on SB MIPS, we get an ISA without MMU, SB and interrupts. We call it the MIPS ISA.

The remainder of this dissertation is structured as follows.

In Chapter 2, we will first introduce the programming discipline, the ownership policy and formally define the models of store buffer machine and abstract machine as well as the safety conditions for the abstract machine, which makes the reduction theorem to go through. Then, we will introduce the coupling relation, invariants and the SB reduction theorem with MMU. At the last portion of Chapter 2, we will present the full paper-and-pencil proof of the theorem.

In Chapter 3, first, we will introduce the MIPS-86 ISA as well as the SB reduced MIPS-86 ISA. Then, we will instantiate our abstract machine model and SB machine model with an ISA, which is very alike to MIPS-86. The main difference is that in the instantiated machine models, the execution of one instruction is divided up to five phases, however, in MIPS-86, the execution of each instruction is atomic. As a consequence, to apply the SB reduction theorem with MMU to MIPS-86, we need to prove the two simulation theorem: (i) each computation of the MIPS-86 machine can be simulated by a computation of the instantiated SB machine. This simulation theorem is trivial and omitted in this thesis because the instantiated SB machine has more interleavings. (ii) Each computation of the instantiated abstract machine can be simulated by a computation of an SB reduced MIPS-86 machine. This simulation will be proved in the Chapter 4.

In Chapter 4, we will apply the SB reduction theorem with MMU to MIPS-86 level. The main portion of this chapter is proving the second simulation theorem mentioned in the last paragraph. Since the ownership is included in the semantics of the abstract machine and the SB machine, first, we need to provide the semantics with ownership to the SB reduced MIPS-86 machine. We introduce a model named *Cosmos* which gives us the abstract semantics with ownership. Then, we will instantiate the *Cosmos* model with SB reduced MIPS-86. Because the instantiated abstract machine has more interleavings than the *Cosmos* SB reduced MIPS-86 machine, before proving the simulation, we will reduce the number of interleavings of the instantiated abstract machine by reordering each execution phase of the same instruction into one block. At last, we will prove a simulation theorem between the instantiated abstract machine and the SB reduced MIPS-86 *Cosmos* machine. Moreover, we have to maintain the safety conditions in the simulation theorem.

In Chapter 5, we will apply the SB reduction theorem to parallel C level for user programs. Since the interrupts and address translations are invisible for the user program, we will first introduction the two simplified ISAs without MMU and interrupt. The one with SB is called SB MIPS ISA, and the other one without SB is called MIPS ISA. Then, we will simplify the SB reduction theorem to get rid of MMUs. We overload the name SB machine and abstract machine in this chapter. Analogous to Chapter 3, we instantiate the abstract machine and the SB machine model with an ISA very alike to MIPS. Also analogous to Chapter 4, we simplify the *Cosmos*

model to get rid of the MMU steps and instantiate it with MIPS ISA. We will prove a simulation theorem between the instantiated abstract machine and the MIPS *Cosmos* machine. In the last portion of this thesis, we will apply the multicore compiler correctness theorem and map the programming discipline to parallel C level. The multicore compiler theorem is defined base on the *Cosmos* model and consists two part: (i) the order reduction theorem that reorders the arbitrary-interleaved ISA computation into a block-scheduling computation. Each block starts with a compiler consistency point. (ii) the sequential compiler correctness theorem. First, we will introduce the order reduction theorem. Then, we will introduce the C intermediate language (C-IL) and the sequential compiler correctness theorem of C-IL. Moreover, we instantiate the *Cosmos* machine with C-IL and simulate the MIPS *Cosmos* machine with the C-IL *Cosmos* machine, which is the application of the multicore compiler correctness theorem.

1.3 Notation

In the scope of this document we use the following notations from [Sch13] and [Bau14].

1.3.1 Basic Notation

- **Numbers**
  The set of integers is denoted by $\mathbb{Z}$. The set of natural numbers is denoted by $\mathbb{N}$ and the set of boolean values $\{0,1\}$ by $\mathbb{B}$. Given a Boolean value $A \in \mathbb{B}$ and values $x, y \in \mathbb{N} \cup \mathbb{Z}$, the value of the ternary operator is defined as follows:

$$A?x : y = \begin{cases} x & A = 1 \\ y & A = 0 \end{cases}$$

- **Records**
  Let A be a set which is the Cartesian product of sets $A_1, A_2, ..., A_k$ and let $n_1, n_2, ..., n_k$ be names for the individual tuple elements of A. Then, given a tuple

$$c \in A = A_1 \times A_2 \times \ldots \times A_k$$

$$c = (a_1, a_2, \ldots, a_k)$$

  $c.n_i$ is used to refer to a_i – the i-th name refers to the i-th *record field* of the tuple. The term *record* is used to refer to such a named tuple. Records $c \in A$ is also introduced by defining

$$c = (c.n_1, c.n_2, \ldots, c.n_k)$$

  followed by a definition of the types of record fields of c. A record update is denoted as

$$c[n_i := v] = c'$$

  where $\forall j \neq i : c'.n_j = c.n_j$ and $c'.n_i = v$. If $k = 2$, the record $c = (a_1, a_2)$ is also called a *pair*. We define the following functions to get the first and second component of a pair.

$$fst(c) = c.a_1$$

$$snd(c) = c.a_2$$

- **Lists**
  Let $l = [x_0 x_1 x_2 ... x_{n-1}]$ then

$$hd(l) = x_0$$
$$tl(l) = [x_1 x_2 ... x_{n-1}]$$
$$last(l) = x_{n-1}$$

  hd and *last* are used to return the first element and last element of a list respectively. *tl* returns the list without the first element. The i-th element of the list l is identified by

$$l[i] = \begin{cases} x_i & i \in [0 : n-1] \\ \bot & otherwise \end{cases}$$

  $l[i]$ can also be written as $l_{[i]}$ in this thesis. The length of list l is defined as follows:

$$|l| = \begin{cases} n & l = [x_0 x_1 ... x_{n-1}] \\ 0 & l = [] \end{cases}$$

  The concatenation of two lists is defined as follows:

$$a \circ b = l$$

  where:

$$l[i] = \begin{cases} a[i] & i \in [0 : |a| - 1] \\ b[i - |a|] & i \in [|a| : |a| + |b| - 1] \\ \bot & otherwise \end{cases}$$

  Let $l_1 = [x_0 x_1 ... x_{n-1}]$ and $l_2 = [y_0 y_1 ... y_{n-1}]$ then the combination of l_1 and l_2 is defined as:

$$\langle l_1, l_2 \rangle \stackrel{def}{=} l$$

  where:

$$l[i] = \begin{cases} (x_i, y_i) & i \in [0 : n-1] \\ \bot & otherwise \end{cases}$$

  Two lists can be combined only if they have identical length.

- **Sets**
  Given a set A, the Hilbert-choice-operator ϵ chooses an element from A:

$$\epsilon A \in A$$

  This is particularly useful when the set consists of a single element, i.e.

$$\epsilon\{x\} = x$$

  or when a definition does not depend on the specific element chosen. Given a set A,

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

  denotes the power set of A, i.e. the set of all subsets of A.

- **Functions**
  Given two sets A and B,

  $$f \in A \rightharpoonup B$$

  denotes that f is a partial function from set A to set B, i.e. $\exists A' \subseteq A.\ f \in A' \to B$. Given $g \in A \to B$ and $X \subseteq A$, the restriction of g to X is defined as :

  $$g \restriction_X = \lambda x \in X.\ g(x)$$

  The function g at entry $x \in A$ can be updated with a new value $v \in B$ as follows:

  $$g(x \mapsto v) \stackrel{def}{=} \lambda y \in A.\ \begin{cases} v & y = x \\ g(y) & otherwise \end{cases}$$

  The composition of partial functions $f, f' : A \rightharpoonup B$ with disjoint domains is denoted by $f \uplus f'$, where $dom(f) \cap dom(f) = \emptyset$ and $dom(f \uplus f') = dom(f) \cup dom(f')$.

  $$f \uplus f' = \lambda a \in dom(f) \cup dom(f').\ \begin{cases} f(a) & : \quad a \in dom(f) \\ f'(a) & : \quad a \in dom(f') \end{cases}$$

  By adding "$\perp$" to the image set in order to denote undefined results, any partial function $f : A \rightharpoonup B$ can be turned into a total function $f : A \to B \cup \{\perp\}$, given that $\perp \notin B$.

1.3.2 Automaton

Given

- a set Z of *states*,
- a set A of input *alphabet symbols*,
- a *transition function* $\delta \in Z \times A \to Z$,
- a non-empty set $Z_0 \subseteq Z$ of *initial states*,
- a non-empty set $Z_A \subseteq Z$ of *accepting states*, and

we consider the tuple $M = (Z, A, \delta, Z_0, Z_A)$ as an *automaton*. The automaton starts from an arbitrary state $z_0 \in Z_0$. $\delta(z, a) = z'$ means a transition from state z to state z' with input a. The current state can be applied to arbitrarily chosen possible transitions and results in a new result state.

In this thesis the hardware is modeled as an automaton. We define the hardware transition by splitting it into smaller transitions, each of which can happen nondeterministically. For each transition we provide:

- *label*: The name of the transition.
- *input parameters*: Inputs from the external world.

- *precondition*: The guard of the transition i.e. the transition may occur only if it is satisfied. Free variable declarations inside the transition are also contained in the precondition.
- *postcondition*: The effect of the transition.

1.3.3 Binary Arithmetic

When introducing our MIPS-86 ISA we will need to argue about arithmetics on bit strings. Bit strings are finite sequences of bits from set $\mathbb{B}$ and we write down the bits from highest to lowest index. The lowest bit has index zero.

$$\forall a \in \mathbb{B}^n.\ a[n-1:0] = a_{n-1}a_{n-2}\cdots a_0$$

Then any bit string $a \in \mathbb{B}^n$ can be interpreted as a binary number with the following value in $\mathbb{N}$.

$$\langle a[n-1:0]\rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

Similarly we can interpret any bit string as an integer that is encoded in two's-complement representation. The two's-complement value of a bit string is defined as follows.

$$[a[n-1:0]] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0]\rangle$$

It can be shown that in modular arithmetic $\langle a\rangle$ and $[a]$ are congruent modulo 2^n. See Section 2.2.2 in [MP00] for more information on two's complement numbers. For conversion of numbers into bit strings we use the bijections

$$bin_n : [0:2^n) \to \mathbb{B}^n \quad \text{and} \quad twoc_n : [-2^{n-1}:2^{n-1}) \to \mathbb{B}^n$$

with the following properies for all $a \in \mathbb{B}^n$.

$$bin_n(\langle a\rangle) = a \qquad twoc_n([a]) = a$$

As a shorthand we allow to write X_n instead of $bin_n(X)$ for any natural number $X \in \mathbb{N}$. We define binary addition and subtraction modulo 2^n of bit strings $a, b \in \mathbb{B}^n$.

$$a +_n b = (bin_{n+1}(\langle a\rangle + \langle b\rangle))[n-1:0] \qquad a -_n b = (twoc_{n+1}([a] - [b]))[n-1:0]$$

Note above that the representative of a binary or two's complement number modulo 2^n can be obtained by considering only its n least significand bits. Also, since binary and two's complement values of a bit string are congruent modulo 2^n, we could have defined addition using two's complement numbers and subtraction using binary representation of the operands. However we stick to the definitions presented above which look most natural to us.

Besides addition and subtraction we can also apply bitwise logical operations on bit strings. Let $a, b \in \mathbb{B}^n$, then we can extend any binary bitwise operator $\bullet : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ to an n-bit operator $\bullet_n : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$, such that for all $i < n$:

$$(a \bullet_n b)[i] = a_i \bullet b_i$$

In this thesis we will use $\bullet \in \{\wedge, \vee, \oplus, \overline{\vee}\}$, where $\oplus$ stands for exclusive OR (XOR) and $\overline{\vee}$ represents negated OR (NOR). We omit the subscript n where it is unambiguous.

For a bit-string $a \in \mathbb{B}^{8k}, k \in \mathbb{N}$ and $0 \leq i < k$, we define

$$byte(i, a) = a[(i + 1) \cdot 8 - 1 : i \cdot 8]$$

to denote the i-th byte in a. We define for $a \in \mathbb{B}^n$ and $n, k \in \mathbb{N}, k > n$

$$zxt_k = 0^{k-n}a$$

the zero-extended bit-string of length k for a and

$$sxt_k = a_{n-1}^{k-n}a$$

to mean the sign-extended bit-string of length k for a. We use the equivalence relation $\equiv \mod k$ defined as follows for $a, b \in \mathbb{Z}, k \in \mathbb{N} \setminus \{0\}$:

$$a \equiv b \mod k \quad \Leftrightarrow \quad \exists z \in \mathbb{Z} : a - b = z \cdot k$$

The modulo-operator is then defined by

$$a \ \mathbf{mod} \ k = \varepsilon\{x \mid a \equiv x \mod k \wedge x \in \{0, \ldots, k - 1\}\}$$

2 Store Buffer Reduction with MMU – Theorem and Proof

In this chapter we introduce the store buffer (SB) reduction theorem with MMU which generalizes previous work by Cohen and Shirmer [CS10b]. In order to reduce the SB, the memory addresses are partitioned into ownership sets. Our model extends Cohen-Shirmer ownership sets with page table sets. We use the identical program discipline as in [CS10b] based on our extended ownership sets. Memory accesses are governed by the program discipline .

We will first introduce the programming discipline and formally define the models of abstract machine and store buffer machine. Then we will introduce the coupling relation, invariants and the store buffer reduction theorem. In the last portion of this chapter, we will present the full paper-and-pencil proof of the theorem.

This chapter is based on the paper [CCK14] and the technical report [CCK13] by the author of this thesis, Ernie Cohen and Mikhail Kovalev. In order to apply our programming discipline both in instruction set architecture (ISA) level and C level, we have to (i) instantiate the SB reduction theorem with an ISA; (ii) apply the simulation theorem [Bau14] between the ISA level and C level. Therefore, we modify our model to fit to the simulation theorem. One major modification is that the ownership transfers are only performed as side effects of volatile[1] instructions. It is helpful when one acquires a lock and wants to obtain the ownership of the memory protected by the lock. In [CCK14] and [CCK13] we also perform the ownership transfer by a non-blocking ghost instruction. The intuition of the modification is that the ghost instruction is not instantiable in any ISA. Jonas Oberhauser proved that these 2 types of ownership transfer are equal in his on going work. We also perform the ownership annotation generation while the instruction is executing.

2.1 Programming Discipline

The programming discipline introduced here is an extension of the programming discipline from [CS10b] and is based on ownership sets, which have to be maintained explicitly by the user. It contains 2 parts: memory access rules and a flushing rule.

All memory accesses can be either shared (volatile) or local and must be *safe* i.e., obey the rules of the programming discipline. Semantically there is no difference between both types of

[1] We rely on a C-idiom, where shared portions of memory are identified by a `volatile` tag. The `volatile` tag prevents a compiler from applying certain optimizations to shared accesses which could cause undesired behavior, e.g., store intermediate values in registers instead of writing them to the memory. Shared memory accesses are also called volatile.

the accesses, but we enforce different rules on volatile and non-volatile memory operations. The interlocked accesses, i.e. those memory accesses which flush the SB as a side effect, follow the same rules as volatile accesses. We distinguish between the following ownership sets of memory addresses:

- Shared, unowned read-write addresses: used for implementing locks [HL09], lock-free algorithms or shared page tables. Every thread can perform volatile reads and writes to these addresses and MMU of every thread is allowed to read and write this memory.
- Shared, unowned read-only addresses: used for static data. Every thread can perform volatile and non-volatile reads from these addresses.
- Shared, owned read-write addresses: used for single-writer-multiple-readers data structures. Every thread can perform volatile reads, but only the (unique) owner is allowed to do volatile writes to these addresses.
- Unshared, owned read-write addresses: used for thread-local data or for data protected by a lock. The owner is allowed to write and read the data with volatile and non-volatile accesses.
- Owned page table addresses: used for local page tables. The owning thread is allowed to read and write these addresses with volatile accesses. The MMU of the owning thread is allowed to read and write this memory.

Note, that we require the translated physical addresses, rather than the untranslated virtual addresses, to adhere to our programming discipline. Showing that the translated physical addresses of memory accesses are safe can be done if one keeps track of the set of all possible address translations for a given thread.

Note, that the set of addresses which can be accessed by the MMU of a thread is actually defined by the set of reachable PTEs from page table origin (PTO), which is stored in a register. Hence, our discipline requires every reachable PTE address to be either in the set of local page table addresses or to be in the set of shared, unowned read-write addresses. The latter is useful in situations when several concurrent threads are sharing the same set of page tables for address translation. Moreover, a local page table can point to a page table shared by MMUs of several threads, which allows to split the address space of a thread into local and shared parts. The other direction is also possible, i.e. a page table located in the shared memory can contain a link to a local page table. In this case any thread can write the shared page table, but only the MMU of the thread which owns the local page table should be able to access both of them. If another MMU would have an access to the "shared" page table, it would automatically be able to access all page tables linked to it, which violates the safety of the MMU access.

Ownership is transferred as a side effect of a volatile or interlocked write operation. A thread is allowed to acquire ownership of an unowned address and to release the ownership of an owned address. When a thread acquires an unowned address, it can either make it owned unshared, owned shared or an owned page table address. When releasing an owned address a thread decides whether to make it shared read-write or shared read-only. It also can make a shared address which it already owns unshared.

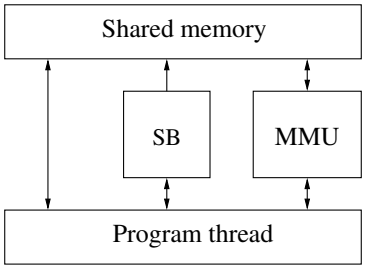


Figure 2.1: Abstract view of x86-TSO with the address translation.

The flushing rule of our programming discipline stays unchanged from [CS10b]: an SB has to be flushed before every volatile read if this read was preceded by a volatile write. This guarantees, that the thread always makes its updates of the shared state visible to others, before it reads a shared variable. In order to implement the flushing rule, we maintain a *dirty bit* in the ghost state. It is set when executing a volatile write and cleared when flushing the SB. Local page tables in this sense are considered as shared state between a thread and its MMU. Safety of a volatile read makes sure that the read is performed only when the dirty flag is not set.

2.2 Formalization

In our computational model the machine contains multiple threads. With the presence of address translation, every thread communicates with a user-visible MMU component (Fig. 3.1). Threads and MMUs also communicate with each other via accessing shared memory. During a computation instructions are issued and appended to the instruction sequence for bookkeeping. Instructions retire from the instruction sequence and enter the store buffer (SB) or apply to shared memory. Instructions emerge from SB and apply to memory. At the same time, MMUs also access the shared memory. When a thread is running in translated mode, a memory access can be executed only when the MMU can provide a suitable address translation for the virtual address of the access.

Before introducing the computational model we define some uninterpreted signatures:

Definition 2.1 (Basic Signatures) The computational model is defined based on the following signatures.

- $\mathbb{A}, \mathbb{V}$ - The set of memory addresses and the set of memory values. The memory is modeled as a function $m : \mathbb{A} \to \mathbb{V}$.
- $\mathbb{P}$ - The set of program states which can be interpreted as the content of a set of registers and some auxiliary flags for denoting the execution phase and page faults.
- $\mathbb{U}$ - The set of MMU states which contains the TLB state and the current value of the page table origin register.
- $\mathbb{T}$ - The set of temporaries which is used to store results of reads.

- $\mathbb{R}$ - The set of all possible access rights for address translation.
- $\mathbb{BW}$ - The set of byte write signals.
- $\mathbb{EEV}$ - The set of external input.

2.2.1 MMU Abstraction

The MMU component can perform non-deterministic steps fetching a page table entry (PTE) from the memory or writing the memory (setting control bits in a PTE). Every read of a PTE can change the state of the MMU, extending the set of translations cached in a TLB. The exact state of the MMU is never known to the user, because MMUs are allowed to perform speculative address translations and to cache them in the TLBs.

A single page table entry occupies a single cell in the memory and has the same type $\mathbb{V}$ as all other memory values. Our MMU model relies on the following (uninterpreted) functions:

- $atran(mmu, va, mode, r) \in 2^{\mathbb{A}}$.
  Given an MMU state $mmu \in \mathbb{U}$, a virtual address $va \in \mathbb{V}$, translation mode $mode \in \mathbb{B}$ (1 - translated mode, 0 - untranslated mode) and the set of access rights $r \in \mathbb{R}$, the function returns the set of translated physical addresses for the specified access. In case there are no available translations the returned set is empty. For the untranslated mode function *atran* should return $\{va\}$. We use this function to obtain an address translation when an instruction is being executed.

- $can\text{-}access(mmu, pa) \in \mathbb{B}$.
  For a physical address $pa \in \mathbb{A}$, the predicate denotes that the MMU can perform an access to a PTE located at address *pa*. This is the case when the MMU has fetched or has found in the TLB a valid PTE, which has the access and the present bits set and which points to the PTE located at address *pa* or when *pa* belongs to the top-level page table. We use this predicate as a guard for MMU read and MMU write steps.

- $\delta_{crtw}(mmu, va) \in \mathbb{U}$.
  This function creates a new walk for virtual address *va* and returns a new MMU state.

- $\delta_{mmur}(mmu, pa, pte) \in 2^{\mathbb{U}}$.
  For page table entry $pte \in \mathbb{V}$ located at address *pa* the function returns the possible set of MMU states after the MMU has processed *pte*. After this step MMU can have complete or incomplete translations through *pte* buffered in the TLB. We use this function to obtain the new state of the MMU after the MMU read step.

- $\delta_{mmuw}(mmu, pa, pte) \in 2^{\mathbb{V}}$.
  This function returns the set of possible PTE values which can be written by the MMU at address *pa*, given that *pte* is the current value of the PTE located at address *pa*. This step models setting of access and dirty bits in a page table entry. We use this function when performing an MMU write step.

- $can\text{-}page\text{-}fault(mmu, va, r, pa, pte) \in \mathbb{B}$.
  This predicate denotes that the MMU can signal a page fault for the virtual address *va* and access rights *r*. The condition for the page fault [2] must be present in the page table entry *pte* located at address *pa* and the MMU must already have an incomplete address translation leading to *pte* buffered in the TLB.

- $\delta_{flush}(mmu, F) \in \mathbb{U}$.
  For the set of (virtual) addresses $F \in 2^{\mathbb{A}}$ the function performs a TLB flush, removing translations for addresses in F from the TLB, and returns the new MMU state after the flush is performed.

- $\delta_{wpto}(mmu, v) \in \mathbb{U}$.
  The function performs a complete SB flush and sets the new value $v \in \mathbb{V}$ for the page table origin (PTO).

We assume *monotonicity* of the MMU i.e., after MMU performs a read of a PTE or walk creation its set of address translations which can be provided by the MMU can only grow. We let $mmu' = \{\delta_{crtw}(mmu, va), \epsilon(\delta_{mmur}(mmu, pa, pte))\}$ then:

$$atran(mmu, va, mode, r) \subseteq atran(mmu', va, mode, r).$$

2.2.2 Instructions

Definition 2.2 (Memory Instruction) The set of memory instructions $\mathbb{I}$ is defined with the following constructors:

$$\begin{aligned}
\mathbb{I} = &\{\textbf{Read } vol\ va\ t\ r\ ext\ bw\ p \mid vol \in \mathbb{B}, va \in \mathbb{A}, t \in \mathbb{T}, r \in \mathbb{R}, bw \in \mathbb{BW}, p \in \mathbb{P}\} \\
&\cup \{\textbf{Write } vol\ va\ (D, f)\ r\ cb\ bw\ p \mid D \in 2^{\mathbb{T}}, f \in (\mathbb{T} \rightharpoonup \mathbb{V}) \to \mathbb{V}\} \\
&\cup \{\textbf{RMW } va\ t\ (D, f)\ cond\ r\ p \mid cond \in (\mathbb{T} \rightharpoonup \mathbb{V}) \to \mathbb{B}\} \\
&\cup \{\textbf{INVLPG } F \mid F \in 2^{\mathbb{A}}\} \\
&\cup \{\textbf{SWITCH } mode \mid mode \in \mathbb{B}\} \\
&\cup \{\textbf{WPTO } v \mid v \in \mathbb{V}\} \\
&\cup \{\textbf{FENCE}\}
\end{aligned}$$

In which:

$$ext \in \mathbb{V} \times \mathbb{BW} \to \mathbb{V} \qquad cb \in \mathbb{V} \times \mathbb{V} \times \mathbb{BW} \to \mathbb{V}$$

Parameter *vol* denotes whether the memory access is volatile.

[2] The page fault can be signalled if the present bit in a PTE is not set, or there is an access rights violation or some of the reserved validity bits in a PTE have inadequate values.

- The read instruction loads the value from virtual address *va* and the translated address *pa* into temporary *t*. *bw* represents the byte write signal and *ext* represents the extension function (zero extend or sign extend). *r* denotes the access permissions, which will be used for the address translation of *va*. If the volatile flag is set then the instruction is allowed to perform an ownership transfer.

  We put some constrains on the uninterpreted parameter *bw* and *ext*. First we introduce an uninterpreted equivalence relation $=_{bw} \in \mathbb{V} \times \mathbb{V} \to \mathbb{B}$ meaning that 2 values are equal with respect to the given byte write signal *bw*. Thus we have:

  $$v_1 = v_2 \to v_1 =_{bw} v_2$$

  We overload the $\leq$ as an uninterpreted relation for byte write signals $\leq \in \mathbb{BW} \times \mathbb{BW} \to \mathbb{B}$. Then we introduce the following properties as constraints on *bw* and *ext*:

  $$bw_2 \leq bw_1 \wedge v_1 =_{bw_1} v_2 \to v_1 =_{bw_2} v_2$$

  It means if 2 values are equal with respect to a given byte write signal then they are also equal with respect to any byte write signals which are not greater than the given one.

  $$v_1 =_{bw} v_2 \to ext(v_1, bw) = ext(v_2, bw)$$

  It means if 2 values v_1 and v_2 are equal with respect to a given byte write signal *bw* then the result of extension function with parameter *bw* does not rely on the choice of first parameter between v_1 and v_2.

- The write instruction stores the value computed by function f at the virtual memory address *va*. Function f takes as a parameter the map from temporaries to pairs of value and physical address and returns a value. D specifies the set of temporaries on which function f operates. Function *cb* combines the old value and new value (computed by f) at virtual address *va* according to the byte write signal *bw*. If the volatile flag is set then the instruction performs the ownership transfer.

  We also put a constraint on the uninterpreted function *cb*.

  $$v_1 =_{bw} v_2 \to cb(v_1, v, bw) =_{bw} v_2$$

  This property means the value combination according to the byte write signal *bw* maintains the relation $=_{bw}$.

- The read modify write instruction (RMW) first loads the value from virtual address *va* as well as the translated address *pa* into temporary *t*. Then it computes the value of the predicate *cond* on the updated set of temporaries and performs a write to *va* if the test succeeds. It also performs the ownership transfer.

- The invlpg instruct removes translations for the virtual addresses in F from the TLB.

- The mode switch instruction changes the translation mode to $mode \in \mathbb{B}$.

- The write to PTO instruction updates the page table origin with value v.
- The fence instruction flushes the SB when executed by the SB machine.

When executed by the SB machine instructions RMW, mode switch, invlpg and write to PTO also flush the SB as a side effect.

To distinguish between different kinds of instructions we introduce predicates $R(I)$, $W(I)$, $RMW(I)$ - for read, write, RMW instructions respectively and $FENCE(I)$, $SWITCH(I)$, $INVLPG(I)$, $WPTO(I)$ - for fence, mode switch, invlpg and write to PTO instructions. Volatile and non-volatile reads and writes are distinguished by predicates $vR(I)$, $vW(I)$ and $nvR(I)$, $nvW(I)$ respectively.

2.2.3 Abstract Machine

The abstract machine with sequentially consistent memory and with address translations does not have SBs. It maintains additional ghost information which allows to enforce the ownership-based programming discipline both for instructions and for MMU memory accesses. We call an execution which maintains this programming discipline *safe*. The abstract machine also contains the ghost *release sets*, which accumulate history information about the addresses released by volatile read instructions. These sets do not influence the execution of the machine and are not used to specify the programming discipline. Hence, one can simply omit them when instantiating the abstract machine. In Sect. 2.3.3 we use these sets to refine the safety criteria.

Configuration

Definition 2.3 (Thread-local Configuration of Abstract Machine) Thread-local configuration $c.ts[i]$ of thread i is defined as a tuple:

$$c.ts[i] = (p, is, \vartheta, mmu, \mathcal{D}, \mathcal{O}, pt, mode, rls_l, rls_s, rls_{pt}) \in \mathbb{K}$$

where:

- $p \in \mathbb{P}$ is the (uninterpreted) program state of the thread,
- $is \in \mathbb{I}^*$ is the instruction list,
- $\vartheta \in \mathbb{T} \rightharpoonup \mathbb{V}$ is the set of read temporaries (a read buffer),
- $mmu \in \mathbb{U}$ is the MMU state,
- $\mathcal{D} \in \mathbb{B}$ is the (ghost) dirty flag,
- $\mathcal{O} \in 2^{\mathbb{A}}$ is the (ghost) thread-local ownership set,
- $pt \in 2^{\mathbb{A}}$ is the (ghost) set of local page table addresses,
- $mode \in \mathbb{B}$ is the translation mode (translated or untranslated),

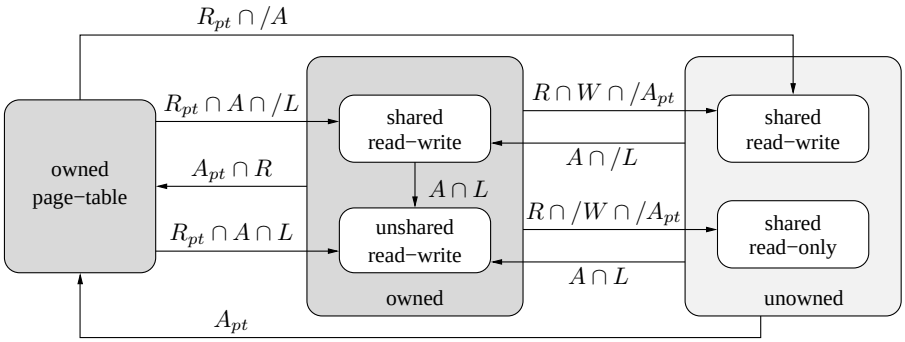


Figure 2.2: Ownership transfer.

- $rls_l, rls_s, rls_{pt} \in 2^{\mathbb{A}}$ are the (ghost) release sets for local, shared and page table addresses respectively.

Definition 2.4 (Abstract Machine Configuration) Configuration of the abstract machine c is defined as a tuple:

$$c = (m, shared, ro, ts) \in \mathbb{M}$$

where $np \in \mathbb{N}$ is the number of threads, $ts \in [0 : np - 1] \to \mathbb{K}$ is the list of thread-local configurations of thread i, $m \in \mathbb{A} \to \mathbb{V}$ is the shared memory of the machine, $shared \in 2^{\mathbb{A}}$ is the (ghost) set of shared addresses and $ro \in 2^{\mathbb{A}}$ is the (ghost) set of read-only addresses.

For components X of thread local configuration $c.ts[i]$ we abbreviate $c.X_{[i]}$. By $c.ghst_{[i]}$ we abbreviate the ghost information of thread i (except the dirty flag) and the shared ghost information:

$$c.ghst_{[i]} = (c.O_{[i]}, c.pt_{[i]}, c.rls_{l[i]}, c.rls_{s[i]}, c.rls_{pt[i]}, c.shared, c.ro)$$

For the union of all release threads of thread i we abbreviate $c.rls_{[i]}$:

$$c.rls_{[i]} = c.rls_{l[i]} \cup c.rls_{s[i]} \cup c.rls_{pt[i]}$$

Ownership Transfer

The ghost ownership annotations *annot* consist of the following sets of addresses: acquired addresses A, the local fraction of acquired addresses L, released addresses R, the writable fraction of released addresses W, acquired page table addresses A_{pt} and released page table addresses R_{pt}. The ownership transfer is performed by volatile write, volatile read and read-modify-write (RMW) instructions. The possible effect of the ownership transfer is given in Fig. 2.2.

Definition 2.5 (Ownership Transfer) Let $ghst = (O, pt, rls_l, rls_s, rls_{pt}, shared, ro)$ be the ghost information of thread i and $annot = (A, L, R, W, A_{pt}, R_{pt})$ is the ownership annotation. Then the ownership transfer performed by volatile read, volatile write or RMW instruction I is defined as:

$$otran(ghst, I, annot) = (O', pt', rls'_l, rls'_s, rls'_{pt}, shared', ro')$$

where the ownership sets change according to Fig. 2.2 and the release sets accumulate released addresses if $vR(I)$ and are cleared otherwise:

$$\begin{aligned} ro' &= ro \cup (R \setminus W) \setminus (A \cup A_{pt}) & \mathcal{O}' &= \mathcal{O} \cup A \setminus R \\ shared' &= shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}) & pt' &= pt \cup A_{pt} \setminus R_{pt} \\ rls'_s &= vR(I)\ ?\ rls_s \cup (R \cap shared) : \emptyset & rls'_{pt} &= vR(I)\ ?\ rls_{pt} \cup R_{pt} : \emptyset \\ rls'_l &= vR(I)\ ?\ rls_l \cup (R \setminus shared) : \emptyset \end{aligned}$$

Semantics

The computation of the abstract machine is a sequence of abstract machine configurations. Each configuration is obtained by applying a non-deterministic transition relation to the previous configuration. Applying the transition relation once is also called one step of computation. Every step is either a program step, a memory step, an MMU step or a page fault step of thread i.

A program step of thread i applies (uninterpreted) function δ_p to the program state, the set of temporaries, the *mode* flag, the MMU state, the instruction sequence and the external inputs of the thread to obtain a new program state and newly generated instructions. These instructions are then appended to the instruction list. For a newly generated read or RMW instruction I we assume the read temporary $I.t$ to be fresh i.e., every read has to be done to a new temporary[3]. We also assume every program state is unique. These assumptions are formalized in Sect. 2.3.5.

Definition 2.6 (Program Step) The semantics of program steps is defined as follows *eev* is the external inputs:

$$\frac{(p', is') = \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev)}{c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c[p_{[i]} := p', is_{[i]} := c.is_{[i]} \circ is']}$$

In which

$$\forall I \in is'.\ W(I) \vee R(I) \vee RMW(I) \rightarrow I.p = c.p_{[i]}$$

A memory step of thread i is defined by a case split on the instruction $I = hd(c.is_{[i]})$ to be executed. In case of a read, write or RMW instruction we first translate the virtual address $I.va$ using the current MMU state and chose a physical address *pa* from the set of available address translations provided by function *atran*. Hence, to execute such an instruction there has to be at least one possible address translation available. For a read instruction we update the value of temporary $I.t$ with the read result v and the translated address *pa*. The value v is computed by function $I.ext$ with the read value $c.m(pa)$ and $I.bw$. In case of a volatile read we perform the ownership transfer according to the result of the (uninterpreted) *og* function. *og* is an ownership annotation generation function which takes the program state and read temporaries and returns the ownership annotation $(A, L, R, W, A_{pt}, R_{pt})$.

[3] When instantiating the model one can easily discharge this assumption by attaching a time stamp to every read destination.

Definition 2.7 (Memory Step for Reads) The semantics of volatile read and non-volatile read are defined as:

$$\frac{nvR(I) \quad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad v = I.ext(c.m(pa), I.bw)}{c \overset{\text{m}}{\Longrightarrow}_i c[\vartheta_{[i]} := c.\vartheta_{[i]}(I.t \mapsto v), is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{vR(I) \quad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad v = I.ext(c.m(pa), I.bw) \quad \vartheta' = c.\vartheta_{[i]}(I.t \mapsto v) \quad ghst' = otran(c.ghst_{[i]}, I, og(I.p, \vartheta'))}{c \overset{\text{m}}{\Longrightarrow}_i c[\vartheta_{[i]} := \vartheta', ghst_{[i]} := ghst', is_{[i]} := tl(c.is_{[i]})]}$$

For a write instruction we obtain the write value by applying $I.cb$ to the value $I.f(\vartheta_{[i]})$, $c.m(pa)$ and $I.bw$. Then we store the write value at memory address pa. In case of a volatile write we also perform the ownership transfer and set the dirty bit. As descried in section 2.1, the dirty bit is a flag used to implement our SB flushing rule.

Definition 2.8 (Memory Step for Writes) The semantics of volatile write and non-volatile read are defined as:

$$\frac{nvW(I) \quad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad v = I.cb(I.f(\vartheta_{[i]}), c.m(pa), I.bw)}{c \overset{\text{m}}{\Longrightarrow}_i c[m := c.m(pa \mapsto v), is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{vW(I) \quad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad v = I.cb(I.f(\vartheta_{[i]}), c.m(pa), I.bw) \quad ghst' = otran(c.ghst_{[i]}, I, og(I.p, c.\vartheta_{[i]}))}{c \overset{\text{m}}{\Longrightarrow}_i c[m := c.m(pa \mapsto v), ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 1, is_{[i]} := tl(c.is_{[i]})]}$$

An RMW instruction first performs a read of memory cell $c.m(pa)$ and the physical address pa into temporary $I.t$ and then checks condition $I.cond$ on the updated set of temporaries. If the test succeeds we obtain the write value by applying $I.f$ to the updated set of temporaries and store this value at address pa. Independent on the test result we reset the dirty bit, clear the release sets and perform the ownership transfer.

Definition 2.9 (Memory Step for RMW) The semantics of RMW is defined as:

$$\frac{RMW(I) \quad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad \vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa)) \quad ghst' = otran(c.ghst_{[i]}, I, og(I.p, \vartheta')) \quad m' = I.cond(\vartheta') \; ? \; c.m(pa \mapsto I.f(\vartheta')) : c.m}{c \overset{\text{m}}{\Longrightarrow}_i c[m := m', \vartheta_{[i]} := \vartheta', ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

Fence instructions do not update the non-ghost part of the state (except reducing the length of the instruction list). For a fence instruction we clear the release sets and reset the dirty bit. Mode switch, INVLPG and write to PTO instructions also clear the release sets and reset the dirty bit as a side effect. In case of a mode switch we change the translation mode to $I.mode$. Invlpg instruction removes the invalidated translation from the MMU using function δ_{flush} and a write to PTO instruction applies function δ_{wpto} to the current MMU state.

Definition 2.10 (Memory Step for Fence, Mode Switch, Invlpg and Write to PTO) The semantics of FENCE, SWITCH, INVLPG and WPTO are defined as:

$$\frac{FENCE(I)}{c \overset{\mathrm{m}}{\Longrightarrow}_i c[\mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]}), rls_{[i]} := \emptyset]}$$

$$\frac{SWITCH(I)}{c \overset{\mathrm{m}}{\Longrightarrow}_i c[mode_{[i]} := I.mode, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]}), rls_{[i]} := \emptyset]}$$

$$\frac{INVLPG(I) \quad mmu' = \delta_{flush}(c.mmu_{[i]}, I.F)}{c \overset{\mathrm{m}}{\Longrightarrow}_i c[mmu_{[i]} := mmu', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]}), rls_{[i]} := \emptyset]}$$

$$\frac{WPTO(I) \quad mmu' = \delta_{wpto}(c.mmu_{[i]}, I.v)}{c \overset{\mathrm{m}}{\Longrightarrow}_i c[mmu_{[i]} := mmu', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]}), rls_{[i]} := \emptyset]}$$

MMU of thread i can either perform a read from the page tables updating the MMU state or a write setting control bits in the page tables. In case of a read the new MMU state is chosen from the set of MMU states provided by function δ_{mmur} and in case of a write we chose the value to be written to the memory from the set of values provided by function δ_{mmuw}. A page fault step is triggered when we are running in translated mode, in the head of the instruction list there is an instruction which requires address translation and the page fault for the address of the instruction can be signalled. As an effect of the page fault we (i) update the program state using (uninterpreted) function δ_{pf} which loads the information about the faulty translation to the program state, (ii) flush all translations for the faulty virtual address from the MMU and (iii) clear the instruction list. As a side effect we also empty the release sets and reset the dirty bit.

Definition 2.11 (MMU Step and Page Fault Step) The semantics of MMU step and page fault

step are defined as:

$$\frac{c.mode_{[i]} \quad mmu' = \delta_{crtw}(c.mmu_{[i]}, va)}{c \overset{\text{muc}}{\Longrightarrow}_i c[mmu_{[i]} := mmu']}$$

$$\frac{c.mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad mmu' \in (\delta_{mmur}(c.mmu_{[i]}, pa, c.m(pa)))}{c \overset{\text{mur}}{\Longrightarrow}_i c[mmu_{[i]} := mmu']}$$

$$\frac{c.mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad v' \in (\delta_{mmuw}(c.mmu_{[i]}, pa, c.m(pa)))}{c \overset{\text{muw}}{\Longrightarrow}_i c[m := c.m(pa \mapsto v')]}$$

$$\frac{\begin{array}{c} c.mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad I = hd(c.is_{[i]}) \quad (R(I) \vee W(I) \vee RMW(I)) \\ can\text{-}page\text{-}fault(c.mmu_{[i]}, I.va, I.r, pa, c.m(pa)) \\ p' = \delta_{pf}(c.p_{[i]}, c.mode_{[i]}, I.va) \quad mmu' = \delta_{flush}(c.mmu_{[i]}, \{I.va\}) \end{array}}{c \overset{\text{pf}}{\Longrightarrow}_i c[is_{[i]} := [], p_{[i]} := p', mmu_{[i]} := mmu', mode_{[i]} := 0, \mathcal{D}_{[i]} := 0, rls_{[i]} := \emptyset]}$$

$$c \overset{\text{mu}}{\Longrightarrow}_i c' \equiv c \overset{\text{muc}}{\Longrightarrow}_i c' \vee c \overset{\text{mur}}{\Longrightarrow}_i c' \vee c \overset{\text{muw}}{\Longrightarrow}_i c'$$

Note, that reading a faulty entry, signalling a page fault, and jump to the interrupt service routine is done in a single atomic transition, i.e. the MMU is not allowed to pre-fetch a faulty PTE first and then use it for signalling a page fault some time later. This forbids to model silent rights granting in page tables i.e., when the user grants more rights in a PTE without a consequent TLB flush, and setting of present bit in a PTE without TLB flushing. In a real TLB of the x86 machine the same behavior can be achieved by performing a fresh re-walk of page tables in case of a page fault [Adv11]

Definition 2.12 (One Step Computation of Abstract Machine) Every step of abstract machine is either a program step, a memory step, an MMU step or a page fault step of thread i:

$$c \underset{\text{eev}}{\Longrightarrow}_i c' \equiv c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \vee c \overset{\text{m}}{\Longrightarrow}_i c' \vee c \overset{\text{mu}}{\Longrightarrow}_i c' \vee c \overset{\text{pf}}{\Longrightarrow}_i c'$$

One step of abstract machine is defined as:

$$c \underset{\text{eev}}{\Longrightarrow} c' \equiv \exists i.\ c \underset{\text{eev}}{\Longrightarrow}_i c'$$

By $c \underset{\text{eev}}{\overset{k}{\Longrightarrow}} c'$ we denote that state c' is reachable from c in exactly k step and by $c \underset{\text{eev}}{\overset{*}{\Longrightarrow}} c'$ we denote the reflexive transitive closure of $\underset{\text{eev}}{\Longrightarrow}$. We also use the same kind of notation when arguing about executions of thread i and executions which consist only of certain kinds of steps (e.g., only memory steps).

Safety Condition

Safety condition for instruction I in thread i restricts the sets of translated physical addresses which can be accessed by read, write and RMW instructions and defines the rules for the ownership transfer. A translated physical address of the volatile read instruction can be either owned by the thread, or shared, or can belong to local page tables. Moreover, we have to make sure that the dirty bit is cleared before we can execute a volatile read. A non-volatile read can only be performed to an owned or read-only address. In case of a volatile write we require the target address to be not present in the ownership and page table sets of other threads and to be excluded from the read only addresses. A non-volatile write can only be performed to owned unshared addresses. For RMW instructions we split cases depending on the result of the RMW test. We treat RMW as a volatile read if the test fails and as a volatile write if the test succeeds. For instructions performing the ownership transfer we require (i) the local fraction L of acquired addresses to be a subset of the acquired addresses A, (ii) acquired addresses $A \cup A_{pt}$ to be disjoint with the ownership and page table sets of other threads, (iii) released addresses R to be a subset of the addresses owned by the thread and released page table addresses R_{pt} to be a subset of the local page table addresses, (iv) acquired addresses A to be a subset of owned, shared and released page table addresses, (v) acquired page table addresses A_{pt} to be a subset of page table, shared and released addresses, (vi) acquired addresses A must be disjoint with released addresses R and acquired page table addresses must be disjoint with released addresses R_{pt} and (vii) acquired addresses A must be disjoint with the acquired page table addresses A_{pt}.

Definition 2.13 (Safety Condition for Ownership Transfer of Volatile Instruction) Let $annot = (A, L, R, W, A_{pt}, R_{pt})$ then the safety condition for ownership transfer of volatile instruction in thread i is defined as:

$$\begin{aligned} &safe\text{-}instr\text{-}otran(c, i, annot) \equiv \forall j \neq i.\ L \subseteq A \wedge (A \cup A_{pt}) \cap (c.O_{[j]} \cup c.pt_{[j]}) = \emptyset \wedge \\ &\quad R \subseteq c.O_{[i]} \wedge R_{pt} \subseteq c.pt_{[i]} \wedge A \subseteq c.O_{[i]} \cup c.shared \cup R_{pt} \wedge A \cap R = \emptyset \wedge \\ &\quad A_{pt} \subseteq c.pt_{[i]} \cup c.shared \cup R \wedge A_{pt} \cap R_{pt} = \emptyset \wedge A_{pt} \cap A = \emptyset \end{aligned}$$

We need some auxiliary definition before defining the safety condition for instruction. For any abstract machine configuration c and instruction I we define

$$\vartheta' = \begin{cases} c.\vartheta_{[i]}(I.t \mapsto c.m(pa)) & RMW(I) \\ c.\vartheta_{[i]}(I.t \mapsto v) & vR(I) \\ c.\vartheta_{[i]} & otherwise \end{cases}$$

In which: $pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \quad v = I.ext(c.m(pa), I.bw)$

Definition 2.14 (Safety Condition for Instructions) Let $annot = og(I.p, \vartheta')$ then the safety

condition for instruction I in thread i is defined as:

$$\begin{aligned}
&safe\text{-}instr(c, i, I, annot) \equiv (\forall pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).\\
&\quad (vR(I) \rightarrow pa \in c.\mathcal{O}_{[i]} \cup c.shared \cup c.pt_{[i]} \wedge \neg c.\mathcal{D}_{[i]}) \wedge\\
&\quad (nvR(I) \rightarrow pa \in c.\mathcal{O}_{[i]} \cup c.ro) \wedge\\
&\quad (vW(I) \rightarrow \forall j \neq i.\ pa \notin c.\mathcal{O}_{[j]} \cup c.pt_{[j]} \wedge pa \notin c.ro) \wedge\\
&\quad (nvW(I) \rightarrow pa \in c.\mathcal{O}_{[i]} \wedge pa \notin c.shared) \wedge\\
&\quad (RMW(I) \wedge \neg I.cond(\vartheta') \rightarrow pa \in c.\mathcal{O}_{[i]} \cup c.shared \cup c.pt_{[i]}) \wedge\\
&\quad (RMW(I) \wedge I.cond(\vartheta') \rightarrow \forall j \neq i.\ pa \notin c.\mathcal{O}_{[j]} \cup c.pt_{[j]} \wedge pa \notin c.ro)) \wedge\\
&\quad (vW(I) \vee vR(I) \vee RMW(I) \rightarrow safe\text{-}instr\text{-}otran(c, i, annot))
\end{aligned}$$

An MMU step reading or writing physical address pa is safe if pa belongs a local page table or to the shared portion of the memory which is not owned by anyone and which does not belong to the read only memory.

Definition 2.15 (Safety Condition for MMU Step) The safety condition for MMU step in thread i is defined as:

$$safe\text{-}mmu\text{-}acc(c, pa, i) \equiv pa \in c.pt_{[i]} \cup c.shared \wedge pa \notin c.ro \wedge \forall j.\ pa \notin c.\mathcal{O}_{[j]}$$

Configuration c of the abstract machine is safe if first instructions in the instruction lists of all threads are safe and all MMU steps as well as the page fault step, which can be performed from c are safe:

Definition 2.16 (Safety Condition for Machine State) Let $I = hd(c.is_{[i]})$ and $annot = og(I.p, \vartheta')$ then the safety condition for machine state c with respect to function og is defined as:

$$\begin{aligned}
&safe\text{-}state(c, og) \equiv \forall i.\ safe\text{-}instr(c, i, I, annot) \wedge\\
&\quad \forall i, pa.\ can\text{-}access(c.mmu_{[i]}, pa) \rightarrow safe\text{-}mmu\text{-}acc(c, pa, i)
\end{aligned}$$

Predicate $safe\text{-}reach(c, n, og)$ denotes that any configuration reachable from configuration c in at most n steps is safe. If we omit the number of steps, then the predicate denotes that any configuration reachable from c is safe.

Definition 2.17 (Safety Condition for Reachable Machine State) The safety condition for every abstract machine states reachable from configuration c within n steps is defined as:

$$\begin{aligned}
&safe\text{-}reach(c, n, og) \equiv\ safe\text{-}state(c, og) \wedge\\
&\quad \forall c'.\ \forall k \leq n.\ c \underset{\text{eev}}{\Rightarrow}^{k} c' \rightarrow safe\text{-}state(c', og)
\end{aligned}$$

The safety condition for every abstract machine state reachable from configuration c is defined as:

$$safe\text{-}reach(c, og) \equiv \forall n.\ safe\text{-}reach(c, n, og)$$

When execution of a abstract machine starts from the initial state and maintains the safety condition, we can be sure that certain relations between various ownership sets are maintained. We gather these properties in the following predicate:

$$\begin{aligned} \mathit{disjoint\text{-}osets}(c) \equiv\ & c.ro \subseteq c.shared \wedge \forall i.\ \forall j \neq i. \\ & c.O_{[i]} \cap c.O_{[j]} = \emptyset \wedge c.O_{[i]} \cap c.ro = \emptyset \wedge \\ & c.O_{[i]} \cap c.pt_{[j]} = \emptyset \wedge c.pt_{[i]} \cap c.pt_{[j]} = \emptyset \wedge \\ & c.O_{[i]} \cap c.pt_{[i]} = \emptyset \wedge c.pt_{[i]} \cap c.shared = \emptyset \end{aligned}$$

$$\mathit{initial}(c) \equiv \mathit{disjoint\text{-}osets}(c) \wedge \forall i.\ c.rls_{[i]} = \emptyset \wedge c.is_{[i]} = []$$

2.2.4 Store Buffer Machine

Our SB machine contains all the components from the abstract machine plus thread-local SBs. The ghost fields carried from the abstract machine configuration are used to simplify the proof, particularly they allow to specify properties of the stores present in the SB without referring to the corresponding configuration of the abstract machine. Store buffers are used not only to buffer memory stores, but also to collect history information about the executed memory and program steps. The ghost fields carried from the abstract machine do not influence the execution of the SB machine. The history information which is recorded in the SB also does not have any influence on the non-ghost components (except of the length of the SB when the history information retires). Hence, proving simulation between an SB machine without the ghost and history components and with them is a trivial task and we omit it here.

The thread-local SB is a FIFO queue of SB instructions $sb \in \mathbb{I}_{sb}^*$.

Definition 2.18 (SB Instruction) The set of SB instruction $\mathbb{I}_{sb}$ is defined as:

$$\begin{aligned} \mathbb{I}_{sb} = & \{\mathbf{Read_{sb}}\ vol\ va\ t\ r\ ext\ bw\ p\ annot\ pa\ v \mid pa \in \mathbb{A}, v \in \mathbb{V}\} \\ & \cup \{\mathbf{Write_{sb}}\ vol\ va\ (D, f)\ r\ cb\ bw\ p\ annot\ pa\ v \mid v \in \mathbb{V}\} \\ & \cup \{\mathbf{Prog_{sb}}\ p_1\ p_2\ is_1\ is_2\ eev \mid p_1, p_2 \in \mathbb{P}, is_1, is_2 \in \mathbb{I}^*, eev \in \mathbb{EEV}\} \end{aligned}$$

The only SB instruction with a non-ghost effect is $\mathbf{Write_{sb}}$, which stores value v to memory address pa when it leaves the SB. The other fields of $\mathbf{Write_{sb}}$ collect the history information and are carried over from the corresponding **Write** instruction, when it is executed and is put to the SB. When read or program steps are executed we also record the ghost information for them in the SB. In case of a read we additionally record physical address pa from where the read was performed and value v that was read. For program steps we record program state p_1 of the thread configuration before the step and program state p_2 after the step together with the instruction sequence is_1 before the step and the newly generated instruction sequence is_2.

We overload predicates $R(I)$, $W(I)$, etc., to work also on SB instructions and introduce predicate $P(I)$ for the recorded program step.

We define some auxiliary functions to convert the format of instructions. Function

$$\mathit{sbins} \in \mathbb{I} \times \mathbb{A} \times \mathbb{V} \times (2^{\mathbb{A}})^6 \to \mathbb{I}_{sb}^*$$

converts a read or write instruction to a corresponding SB instruction:

$$sbins(I, pa, v, annot) =$$
$$\begin{cases} [\mathbf{Read_{sb}}\ I.vol\ I.va\ I.t\ I.r\ I.ext\ I.bw\ I.p\ annot\ pa\ v] & R(I) \\ [\mathbf{Write_{sb}}\ I.vol\ I.va\ I.(D, f)\ I.r\ I.cb\ I.bw\ I.p\ annot\ pa\ v] & W(I) \\ [] & otherwise \end{cases}$$

The function $ins \in \mathbb{I}_{sb} \to \mathbb{I}^*$ performs conversion in the other direction:

$$ins(I) = \begin{cases} [\mathbf{Read}\ I.vol\ I.va\ I.t\ I.r\ I.ext\ I.bw\ I.annot] & R(I) \\ [\mathbf{Write}\ I.vol\ I.va\ I.(D, f)\ I.r\ I.cb\ I.bw\ I.annot] & W(I) \\ [] & otherwise \end{cases}$$

We define an overloaded version of the function ins which operates on a list of SB instructions:

$$ins(sb) = \begin{cases} [] & sb = [] \\ ins(tl(sb)) & P(hd(sb)) \\ ins(hd(sb)) \circ ins(tl(sb)) & \text{otherwise.} \end{cases}$$

The history information for reads and program steps allows to keep track of instructions which have been executed in the store buffer machine after the preceding write instruction is executed, but before it leaves the SB. We use this information in Sect. 2.3.1 to couple the state of SB and abstract machines in the simulation theorem. The history information for writes keeps track of the store values and the physical address chosen for the address translation. This information is coupled with the current state of the SB machine with the help of additional invariants (Sect. 2.3.4). These invariants together with the coupling relation guarantee, for instance, that we can chose the same address translation when executing the corresponding instruction in the abstract machine and that the store value of that instruction in the abstract machine will be the same as in the SB machine.

Configuration

Definition 2.19 (Thread-local Configuration of SB Machine) Thread-local configuration $c_{sbh}.ts_{[i]}$ has all components from the local configuration of the abstract machine plus an SB component:

$$c_{sbh}.ts[i] = (p, is, \vartheta, mmu, pt, mode, \mathcal{D}, \mathcal{O}, rls_l, rls_s, rls_{pt}, sb) \in \mathbb{K}_{sbh}$$

For components X of thread local configuration $c_{sbh}.ts_{[i]}$ we simply write $X_{[i]}$ if configuration c_{sbh} is clear from the context.

Definition 2.20 (SB Machine Configuration) Configuration of the SB machine c_{sbh} has the same components as configurations of the abstract machine:

$$c_{sbh} = (m, shared, ro, ts) \in \mathbb{M}_{sbh}$$

For $X \in \{shared, ro, m\}$ we write X as a shorthand for $c_{sbh}.X$. X' or $X'_{[i]}$ denotes the corresponding component of c'_{sbh}. Note, that we completely omit the configuration identifier **only** for the SB machine, and always write it for the abstract machines in order to avoid confusion. As in the case of the abstract machine, we abbreviate by $rls_{[i]}$ the union of all release sets of thread i and by $ghst_{[i]}$ we denote the ghost information of thread i (excluding the dirty flag) and the shared ghost information.

Semantics

The computation of the SB machine is a sequence of SB configurations. Each configuration is obtained by applying a non-deterministic transition relation to the initial configuration iteratively. Every computation step of SB machine is either a program step, a memory step, an MMU step, a page fault step or SB setp of thread i.

A program step of the SB machine has the same effect as in the abstract machine and is recorded as history information in the SB:

Definition 2.21 (Program Step) The semantics of program step is defined as following in which *eev* is external inputs:

$$\frac{\begin{array}{c}(p', is') = \delta_p(p_{[i]}, \vartheta_{[i]}, mode_{[i]}, mmu_{[i]}, is_{[i]}, eev) \\ I = PROG_{sb}\ p_{[i]}\ p'\ is_{[i]}\ is'\ eev\end{array}}{c_{sbh} \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c_{sbh}[p_{[i]} := p', sb_{[i]} := sb_{[i]} \circ I, is_{[i]} := is_{[i]} \circ is']}$$

In which

$$\forall I \in is'.\ W(I) \vee R(I) \vee RMW(I) \rightarrow I.p = p_{[i]}$$

MMU read, write and walk creation steps of the SB machine have exactly the same semantics as in the abstract machine. The page fault step can occur only when the SB is empty:

Definition 2.22 (Page Fault Step) The semantics of page fault step is defined as:

$$\frac{\begin{array}{c}mode_{[i]} \quad can\text{-}access(mmu_{[i]}, pa) \quad I = hd(is_{[i]}) \quad (R(I) \vee W(I) \vee RMW(I)) \\ can\text{-}page\text{-}fault(mmu_{[i]}, I.va, I.r, pa, m(pa)) \quad sb_{[i]} = [] \\ p' = \delta_{pf}(p_{[i]}, mode_{[i]}, I.va) \quad mmu' = \delta_{flush}(mmu_{[i]}, \{I.va\})\end{array}}{c_{sbh} \overset{\text{pf}}{\Longrightarrow}_i c_{sbh}[is_{[i]} := [], p_{[i]} := p', mmu_{[i]} := mmu', mode_{[i]} := 0, rls_{[i]} := \emptyset]}$$

Predicate $sbehit(I, a)$ denotes whether there is a store buffer hit for the given entry I and address a:

$$sbehit(I, a) = W(I) \wedge I.pa = a.$$

Predicate $sbhit(sb, a)$ denotes whether there is a store buffer hit for the given address a:

$$sbhit(sb, a) = \exists j < |sb|.\ sbehit(sb[j], a).$$

Function $maxhit(sb, a)$ computes the last index of the store buffer entry which hit the given address a or returns $\bot$ if there is no such index:

$$maxhit(sb, a) = \begin{cases} max\{j \mid sbehit(sb[j], a)\} & sbhit(sb, a) \\ \bot & otherwise. \end{cases}$$

A memory step of thread i is defined by a case split on the instruction $I = hd(c.is_{[i]})$ to be executed. The read instruction performs the read and is recorded to the SB as history information. The read value is obtained with the partial function $fwd(sb_{[i]}, m, pa, bw)$, which forwards the last store value to pa in the SB or returns the memory value $m(pa)$ if there are no writes to pa in the SB. If the read access can not be serviced by one store buffer entry (e.g. we have a partial store buffer hit on the last store value to pa) the fwd returns $\bot$. Let $j = maxhit(sb, pa)$ then

$$fwd(sb, m, pa, bw) = \begin{cases} m(pa) & j = \bot \\ sb[j].v & j \neq \bot \wedge bw \leq sb[j].bw \\ \bot & otherwise. \end{cases}$$

The read instruction updates the value of temporary $I.t$ with the read result v, which is computed by function $I.ext$ with the value forwarding from $sb_{[i]}$ and $I.bw$. It is recorded in the SB as the ghost history information. Note that in the semantics we forbid partial forwarding form the SB.

Definition 2.23 (Memory Step for Read) The semantics of read is defined as:

$$\frac{\begin{array}{c} R(I) \quad pa \in atran(mmu_{[i]}, I.va, mode_{[i]}, I.r) \quad v_1 = fwd(sb_{[i]}, m, pa, I.bw) \\ v_1 \neq \bot \quad v = I.ext(v_1, I.bw) \quad \vartheta' = \vartheta_{[i]}(I.t \mapsto v) \quad annot = og(I.p, \vartheta') \end{array}}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[\vartheta_{[i]} := \vartheta', sb_{[i]} := sb_{[i]} \circ sbins(I, pa, v, annot), is_{[i]} := tl(is_{[i]})]}$$

The write instruction is not executed immediately, but is buffered in the SB together with the ghost history information.

Definition 2.24 (Memory Step for Write) The semantics for write is defined as:

$$\frac{\begin{array}{c} W(I) \quad pa \in atran(mmu_{[i]}, I.va, mode_{[i]}, I.r) \\ \mathcal{D}' = vW(I) \vee \mathcal{D}_{[i]} \quad annot = og(I.p, \vartheta_{[i]}) \end{array}}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[sb_{[i]} := sb_{[i]} \circ sbins(I, pa, I.f(\vartheta_{[i]}), annot), is_{[i]} := tl(is_{[i]}), \mathcal{D}_{[i]} := \mathcal{D}']}$$

The read modify write, fence, mode switch, invlpg and write to PTO instructions can be executed only when SB is empty and have the same semantics as defined for the abstract machine.

Definition 2.25 (Memory Step for RMW, FENCE, SWITCH, INVLPG and WPTO)

$$\frac{RMW(I) \quad pa \in atran(mmu_{[i]}, I.va, mode_{[i]}, I.r) \quad \vartheta' = \vartheta_{[i]}(I.t \mapsto m(pa)) \quad ghst' = otran(ghst_{[i]}, I, og(I.p, \vartheta')) \quad sb_{[i]} = [] \quad m' = I.cond(\vartheta')\ ?\ m(pa \mapsto I.f(\vartheta')) : m}{c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c_{sbh}[m := m', \vartheta_{[i]} := \vartheta', ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{FENCE(I) \quad sb_{[i]} = []}{c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c_{sbh}[rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{SWITCH(I) \quad sb_{[i]} = []}{c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c_{sbh}[mode_{[i]} := I.mode, rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{INVLPG(I) \quad sb_{[i]} = [] \quad mmu' = \delta_{flush}(mmu_{[i]}, I.F)}{c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c_{sbh}[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{WPTO(I) \quad sb_{[i]} = [] \quad mmu' = \delta_{wpto}(mmu_{[i]}, I.v)}{c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c_{sbh}[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

The SB collects read, write and program instructions. Among those instructions only writes contain non-ghost data and perform an update of the non-ghost part of the configuration when they leave the SB. An SB step of thread i is defined by a case split on $I = hd(sb_{[i]})$. When a write instruction leaves the SB, then it deliverers a buffered store to the memory and performs an ownership transfer if the write is volatile. A read instruction only performs an ownership transfer. Program instructions are simply skipped. We overload the ownership transfer function as

$$\forall I \in \mathbb{I}_{sb}.\ otran(ghst, I) = otran(ghst, ins(I), I.annot)$$

Definition 2.26 (SB Step) The semantics of an SB step is defined as:

$$\frac{W(I) \quad v = I.cb(I.v, m(I.pa), I.bw) \quad ghst' = (nvW(I)\ ?\ ghst_{[i]} : otran(ghst_{[i]}, I))}{c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c_{sbh}[m := m(I.pa \mapsto v), ghst_{[i]} := ghst', sb_{[i]} := tl(sb_{[i]})]}$$

$$\frac{R(I) \quad ghst' = (nvR(I)\ ?\ ghst_{[i]} : otran(ghst_{[i]}, I))}{c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c_{sbh}[ghst_{[i]} := ghst', sb_{[i]} := tl(sb_{[i]})]} \qquad \frac{P(I)}{c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c_{sbh}[sb_{[i]} := tl(sb_{[i]})]}$$

The computation of the SB machine is defined by a non-deterministic transition relation $c_{sbh} \underset{\text{eev}}{\Longrightarrow} c'_{sbh}$

Definition 2.27 (One Step Computation of SB Machine) Every step of SB machine is either a program step, a memory step, an SB step, an MMU step or a page fault step of thread i:

$$c_{sbh} \underset{\text{eev}}{\Longrightarrow}_i c'_{sbh} \equiv c_{sbh} \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i c'_{sbh} \vee c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{mu}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{pf}}{\Longrightarrow}_i c'_{sbh}$$

One step of SB machine is defined as:

$$c_{sbh} \underset{\text{eev}}{\Longrightarrow} c'_{sbh} \equiv \exists i.\, c_{sbh} \underset{\text{eev}}{\Longrightarrow}_i c'_{sbh}$$

2.3 Store Buffer Reduction

The main property we have to prove is that the reads (including the MMU reads) performed in both machines get the same value. As a result, the crucial role plays the scheduling of the abstract machine. The most straightforward approaches one could think about are (i) executing an instruction on the abstract machine when this instruction is executed on the SB machine and (ii) executing an instruction on the abstract machine when this instruction leaves the SB (i.e., delaying the abstract machine until this point). The history information recorded in the SB in this case helps to reconstruct the instructions which yet have to be executed in the abstract machine. However, both of these approaches do not work. In the first case we get a problem when thread i executes a volatile write and puts it to the store buffer and then thread j executes a volatile read. In the abstract machine the result of the write will already be committed to the memory and thread j will read the new value, while in the SB machine thread j will get the old value, because the write is still present in the store buffer of thread i. The same example also rules out the second approach: if we delay a volatile read of thread j in the abstract machine, then it might be scheduled after the volatile write of thread i leaves the SB, and the abstract machine will again read the new value.

Hence, to guarantee the consistency of read results in both executions we have to schedule the abstract machine in such a way, that

- a volatile write must be delayed in the abstract machine until the volatile write exits the SB in the SB machine,
- a volatile read must be executed simultaneously in both machines. Our programming discipline guarantees that when a volatile read is executed there can be no volatile writes in the SB of the SB machine.

As a result, the shared portion of the memory will be always consistent between the machines. The page tables in that sense are also considered as part of the "shared" memory, even if these page tables are thread-local. Indeed, if the content of local page tables would be inconsistent between the machines, then MMU reads in the abstract machine would either read different values (due to the absence of SB forwarding for MMU reads) or would have to be delayed until the competing volatile writes to local page tables leave the SB. However delaying MMU reads in the virtual machine is also not feasible, because that would force us to delay subsequent MMU writes. These MMU writes might be performed to shared page tables (we do allow a local PTE

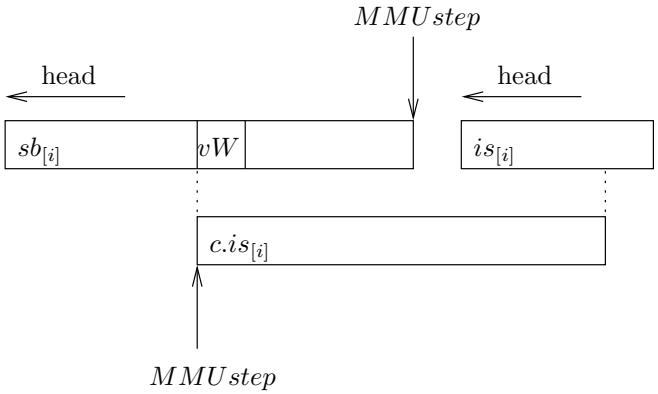


Figure 2.3: Reordering of MMU steps.

to point to a shared PTE), which would lead to inconsistent shared memory. As a result, we have to execute all MMU steps simultaneously in both machines. Together with the possible delay in instruction execution this leads to reordering of MMU steps with respect to executed instructions in a given thread, but this reordering is always done to the left of the instruction sequence (Fig. 2.3). This behaviour is fine, because the monotonicity property of our MMU model guarantees that once added the address translations are never removed from the MMU. In the abstract machine some address translations will be added to the MMU earlier than in the SB machine (if one counts time by the number of executed instructions), but they will still remain there when the instructions which might rely on these address translations are executed. In the following section we define the coupling relation $c_{sbh} \sim c$ which captures the essence of our scheduling policy.

2.3.1 Coupling Relation

The part of the SB after and including the first volatile write is called *suspended*, because these steps are not yet executed on the abstract machine. The part of the SB before the first volatile write (or the whole SB if it does not have any volatile writes) is called *executed*, since the abstract machine has already performed these steps. We introduce the functions $exec(sb)$ and $susp(sb)$, which return the executed and the suspended parts of the SB respectively:

$$exec(sb) = \begin{cases} sb[0 : k-1] & k = min\{j \mid vW(sb[j])\} \\ sb & \text{no vW in } sb. \end{cases}$$

$$susp(sb) = \begin{cases} sb[k : |sb|-1] & k = min\{j \mid vW(sb[j])\} \\ [] & \text{no vW in } sb. \end{cases}$$

In contrast to a non-deterministic memory transition due to non-deterministic address translation, an SB step is always deterministic. To simplify the notation we introduce a function δ_{sb}, which computes the next state of the SB machine after an SB step of thread i or returns the

unmodified machine state in case the SB of thread i is empty:

$$\delta_{sb}(c_{sbh}, i) = \begin{cases} c'_{sbh} & sb_{[i]} \neq [] \wedge c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c'_{sbh} \\ c_{sbh} & sb_{[i]} = []. \end{cases}$$

Configuration of the machine after executing k steps of the SB of thread i is defined inductively as:

$$\delta^k_{sb}(c_{sbh}, i) = \begin{cases} c_{sbh} & k = 0 \\ \delta_{sb}(\delta^{k-1}_{sb}(c_{sbh}, i), i) & \text{otherwise.} \end{cases}$$

Function $\Delta_{sb}(c_{sbh}, i)$ executes all instruction in the SB of thread i and function $\Delta^{exec}_{sb}(c_{sbh}, i)$ executes all instructions before the first volatile write:

$$\Delta_{sb}(c_{sbh}, i) = \delta^{|sb_{[i]}|}_{sb}(c_{sbh}, i)$$
$$\Delta^{exex}_{sb}(c_{sbh}, i) = \delta^{|exec(sb_{[i]})|}_{sb}(c_{sbh}, i).$$

We overload functions $\Delta_{sb}(c_{sbh})$ and $\Delta^{exec}_{sb}(c_{sbh})$ (leaving out the thread id) to compute the machine configuration after consecutive execution of instructions from SBs of all threads, starting with thread id 0:

$$\Delta_{sb}(c_{sbh}) = \Delta_{sb}(...\Delta_{sb}(\Delta_{sb}(c_{sbh}, 0), 1)..., np - 1)$$
$$\Delta^{exec}_{sb}(c_{sbh}) = \Delta^{exec}_{sb}(...\Delta^{exec}_{sb}(\Delta^{exec}_{sb}(c_{sbh}, 0), 1)..., np - 1).$$

We define $\Delta_{sb[\neq i]}(c_{sbh})$ and $\Delta^{exec}_{sb[\neq i]}(c_{sbh})$ to do the same computation but excluding steps of thread i.

With this notation we can now define the coupling relation $c_{sbh} \sim c$ (Definition 2.28).

- To get shared component $X \in \{shared, ro, m\}$ of the abstract machine we take the corresponding component of the SB machine and execute all instructions in the executed portions of SBs of all threads. Note, that since the executed parts of SBs do not contain volatile writes, the content of the shared memory is always consistent between two machines,

- For thread-local components $X \in \{O, pt, rls_l, rls_s, rls_{pt}\}$ of thread i we take the corresponding component of the SB machine and execute all instructions in the executed portion of the SB of thread i.

- To couple the instruction list (Fig. 2.4) we first observe, that the instruction list in the abstract machine should contain all instructions from the suspended part of the SB (with the exception of the history information for program steps) plus the instructions from the instruction list of the SB machine. Note however, that some of the instructions in the SB machine might be generated by the program steps, which are suspended in the virtual machine. Instead of removing these instructions from the instruction list of the SB machine, in the coupling relation we append them to the instruction list of the abstract machine. The function $ins(susp(sb_{[i]}))$ removes the program steps from the suspended

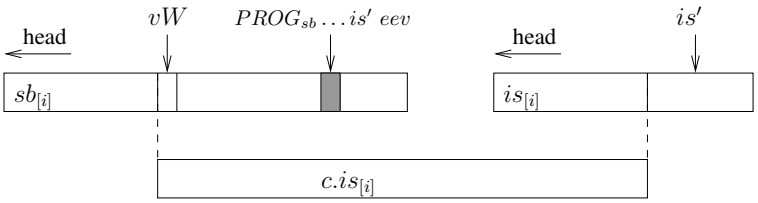


Figure 2.4: Instruction list coupling.

portion of the SB and converts the instructions recorded in the store buffer into regular memory instructions by throwing away the additional history information. The function $p\text{-}ins(susp(sb_{[i]}))$ extracts instructions generated by the program steps recorded in the suspended portion of the SB:

$$p\text{-}ins(sb) = \begin{cases} [] & sb = [] \\ is_2 \circ p\text{-}ins(tl(sb)) & hd(sb) = \mathbf{Prog_{sb}}\ \mathrm{p}_1\ \mathrm{p}_2\ \mathrm{is}_1\ \mathrm{is}_2\ \mathrm{eev} \\ p\text{-}ins(tl(sb)) & \text{otherwise} \end{cases}$$

- The set of temporaries of thread i of the abstract machine is obtained by removing all the temporaries used for reads in the suspended part of the SB, done by the function $del\text{-}t(\vartheta_{[i]}, sb)$

  $$del\text{-}t(\vartheta, sb) = \vartheta \restriction_{dom(\vartheta) \setminus load_t(sb)} .$$

  where:

  $$load_t(sb) = \bigcup \{sb[k].t \mid k < |sb| \wedge R(sb[k])\}$$

  Since we assume all temporaries in the newly generated instructions to be fresh, we can be sure that we do not remove the temporaries which have been used for already executed reads.

- The program state in the abstract machine is obtained by function

  $$hd\text{-}p(p_{[i]}, susp(sb_{[i]}))$$

  which takes the recorded pre-state of the first program instruction in the suspended portion of the SB or simply takes the current program state in the SB machine if there are no suspended program instructions:

  $$hd\text{-}p(p, sb) = \begin{cases} p & sb = [] \\ p_1 & hd(sb) = \mathbf{Prog_{sb}}\ \mathrm{p}_1\ \mathrm{p}_2\ \mathrm{is}_1\ \mathrm{is}_2\ \mathrm{eev} \\ hd\text{-}p(p, tl(sb)) & \text{otherwise.} \end{cases}$$

- The address translation mode and the MMU state are always equal between the machines.
- The dirty bit in the SB machine is set iff it is also set in the abstract machine or if there is a volatile write in the (suspended part of the) SB.

Definition 2.28 (Coupling Relation)

$$
\begin{aligned}
c_{sbh} \sim c \;\equiv\; & \forall X \in \{shared, ro, m\}.\ c.X = \Delta_{sb}^{exec}(c_{sbh}).X \wedge \\
& \forall i.\ \forall X \in \{\mathcal{O}, pt, rls_l, rls_s, rls_{pt}\}.\ c.X_{[i]} = \Delta_{sb}^{exec}(c_{sbh}, i).X_{[i]} \wedge \\
& c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) = ins(susp(sb_{[i]})) \circ is_{[i]} \wedge \\
& c.\vartheta_{[i]} = del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]})) \wedge c.p_{[i]} = hd\text{-}p(p_{[i]}, susp(sb_{[i]})) \wedge \\
& c.mode_{[i]} = mode_{[i]} \wedge c.mmu_{[i]} = mmu_{[i]} \wedge \\
& ((c.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I)) \leftrightarrow \mathcal{D}_{[i]})
\end{aligned}
$$

Note that the coupling relation defined here gives the full consistency between all components only when all the SBs are empty. As a result, one has to require the execution to end with a configuration where SBs are empty in order to use the SB reduction theorem (Theorem 2.29) to transfer all the results of the execution of the SB machine to the abstract machine. However, intermediate configurations, for instance those where only SBs of some threads are empty, can be also used to transfer partial execution results (e.g., for the memory content owned by a thread).

2.3.2 Reduction Theorem

Our main result is a simulation theorem between the SB machine and the abstract machine.

Theorem 2.29 (SB Reduction)

$$
\begin{aligned}
& c_{sbh} \underset{eev}{\Rightarrow}^{*} c'_{sbh} \wedge c_{sbh} \sim c \wedge initial(c) \wedge sbempty(c_{sbh}) \wedge safe\text{-}reach(c) \rightarrow \\
& \quad \exists c'.\ c \underset{eev}{\Rightarrow}^{*} c' \wedge c'_{sbh} \sim c'
\end{aligned}
$$

We consider only executions which start with empty SBs:

$$sbempty(c_{sbh}) = \forall i.\ c_{sbh}.sb_{[i]} = [].$$

We do the proof of Theorem 2.29 on step by step basis i.e., for every step of the SB machine we find a (possibly empty) corresponding sequence of steps of the abstract machine in such a way, that the coupling relation is maintained.

Note, that the scheduling for instructions performing local memory accesses is not so crucial, because our programming discipline guarantees that these accesses never race with memory accesses of other threads and with memory accesses performed by MMUs, including the MMU of the executing thread itself. By a race here we understand two competing accesses where at least one of them is a write.

The following scheduling policy satisfies all the conditions stated above:

- when a volatile write is executed in the SB machine, the abstract machine is delayed and does not make any steps,
- when a volatile read is executed in the SB machine, the abstract machine executes the same step,
- when a non-volatile memory access or a program step of thread i is executed in the SB machine we make a case split on whether the SB of thread i contains a volatile write or not. In case it does, then execution of thread i in the abstract machine is already suspended (it is waiting until the volatile write will leave the SB) and we do not make any steps. In case it does not, then the abstract machine executes the same step of thread i,
- all the other instructions and the page fault step require the SB to be empty before they can be executed. Hence, we execute these steps simultaneously in both machines,
- when a volatile write exits the SB, the abstract machine executes this volatile write and all instructions and program steps recorded in the SB until the next volatile write (or until the end of the SB, if there are no other volatile writes there),
- when a read, non-volatile write, a ghost instruction or the recorded program step exits the SB, the abstract machine does not perform any steps, because it has already performed the corresponding step before,
- MMU steps are always executed simultaneously in both machines.

As a result of the rules stated above, the abstract machine is on-parallel or behind the SB machine in terms of executed memory steps (instructions) and program steps and it is always on-par with the SB machine in terms of executed MMU steps. However, in terms of the stores committed to the memory the abstract machine is either on-parallel or ahead of the SB machine.

2.3.3 Safety of the Delayed Release

Our programming discipline essentially only allows races between volatile accesses of different threads and between MMU accesses. Practically, this means that (i) while the reads are present in the suspended portion of the SB, the read results can not be invalidated by other threads and by MMUs and (ii) when a volatile read or an MMU read is executed in the SB machine, there can be no (non-volatile) writes to the same address in the executed portions of other threads. In the proof this for instance manifests in the following proof obligation: when a volatile write to pa leaves the SB of thread i, there are no (non-volatile) reads to pa in the suspended portions of SBs of other threads. We prove this by contradiction, assuming that such a read exists in the SB of thread j. In the corresponding configuration of the abstract machine this read is not yet executed. Hence, we forward thread j in the virtual machine configuration until the point where this read is at the head of the instruction list. From safety of all reachable traces of the virtual machine, we know that the resulting state is safe. Moreover, we can prove that for all reachable safe states of the abstract machine disjointness of the ownership sets is preserved. This implies a contradiction, because the safety of thread j requires pa to be either owned or read-only and the safety of thread i requires pa to be not owned by other threads and not read-only.

However, for some races the strategy described above does not work. Consider a case when thread i starting with an empty SB performs a non-volatile write to pa and then a volatile read release of pa. Since the SB of thread i does not contain any volatile writes, these steps are immediately executed in the abstract machine. After that, MMU of thread i performs a read from pa. In the current trace of the abstract machine this operation is safe, since the address pa is not owned by any thread at the time of the MMU step. However, the read results in two machines will be inconsistent, because in the abstract machine the store to pa is already committed to the memory and in the SB machine it is still present in the SB of thread i. To rule out this situation, we have to construct another unsafe trace of the abstract machine, which deviated from the current trace somewhere in the past. For the given example this means that we have to consider a trace where the MMU step is performed before the release takes place. Construction of these deviated traces is not feasible in the step-by-step proof of Theorem 2.29, because there we only have safety of reachable traces starting from the current state of the abstract machine. To solve this problem we observe that the complications arise only when the addresses are released by volatile read instructions. Information about these releases is collected into the (ghost) release sets. We use these sets to define *safety of the delayed release*, which can be used to rule out the described situation.

Definition 2.30 (Safety Condition of Delayed Release for Instructions) Safety condition of the delayed release for an instruction I in thread i and for an MMU access to address pa in thread i, where $og(I.p, \vartheta') = (A, L, R, W, A_{pt}, R_{pt})$

$$
\begin{aligned}
&safe\text{-}instr_d(c, i, I, og(I.p, \vartheta')) = safe\text{-}instr(c, i, I, og(I.p, \vartheta')) \wedge \\
&\quad \forall pa.\ \forall j \neq i.\ pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}) \rightarrow \\
&\quad (vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \rightarrow pa \notin c.rls_{l[j]} \cup c.rls_{pt[j]}) \wedge \\
&\quad (nvR(I) \vee vW(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \rightarrow pa \notin c.rls_{[j]}) \wedge \\
&\quad (vW(I) \vee vR(I) \vee RMW(I) \rightarrow (A \cup A_{pt}) \cap c.rls_{[j]} = \emptyset) \\
&safe\text{-}mmu\text{-}acc_d(c, a, i) = safe\text{-}mmu\text{-}acc(c, a, i) \wedge a \notin c.rls_{l[i]} \wedge \forall j \neq i.\ a \notin c.rls_{[j]}
\end{aligned}
$$

Definition 2.31 (Safety Condition of Delayed Release for Machine State) Let $I = hd(c.is_{[i]})$ then

$$
\begin{aligned}
safe\text{-}state_d(c, og) = \forall i.\ &safe\text{-}instr_d(c, i, I, og(I.p, \vartheta')) \wedge \\
&(\forall i, pa.\ can\text{-}access(c.mmu_{[i]}, pa) \rightarrow safe\text{-}mmu\text{-}acc_d(c, pa, i))
\end{aligned}
$$

Definition 2.32 (Safety Condition of Delayed Release for Reachable Machine State)

$$
\begin{aligned}
safe\text{-}reach_d(c, n, og) &= safe\text{-}state_d(c, og) \wedge \forall c'.\forall k \leq n.\ c \underset{\text{eev}}{\Rightarrow}^{k} c' \rightarrow safe\text{-}state_d(c', og) \\
safe\text{-}reach_d(c, og) &= \forall n.\ safe\text{-}reach_d(c, n, og)
\end{aligned}
$$

2.3.4 Invariants

In this section we define invariants $inv(c_{sbh})$ on the SB machine, which we later use in the simulation proof. We start with giving some auxiliary definitions.

The set of all addresses acquired (resp. released) by instructions in store buffer sb is defined as

$$acq(sb) = \bigcup\{sb[k].A \mid k < |sb| \wedge (vR(sb[k]) \vee vW(sb[k]))\}$$
$$rels(sb) = \bigcup\{sb[k].R \mid k < |sb| \wedge (vR(sb[k]) \vee vW(sb[k]))\}$$

The set of all PT addresses acquired (resp. released) by all instructions in store buffer sb is defined as

$$acq_{pt}(sb) = \bigcup\{sb[k].A_{pt} \mid k < |sb| \wedge (vR(sb[k]) \vee vW(sb[k]))\}$$
$$rels_{pt}(sb) = \bigcup\{sb[k].R_{pt} \mid k < |sb| \wedge (vR(sb[k]) \vee vW(sb[k]))\}$$

As counterparts to the safety condition in the abstract machine, we defined the following predicates for the SB machine. The subsequent one checks whether ownership annotations of a given instruction $I \in \mathbb{I}_{sb}$ are safe with respect to a given state of the machine and a given thread ID i. In which, we write

$$\begin{aligned} &safe\text{-}annot(c_{sbh}, i, I) = vR(I) \vee vW(I) \;\rightarrow \\ &\quad I.A \subseteq shared \cup I.R_{pt} \cup O_{[i]} \wedge I.L \subseteq I.A \wedge I.A \cap I.A_{pt} = \emptyset \wedge \\ &\quad I.A \cap I.R = \emptyset \wedge I.R \subseteq O_{[i]} \wedge I.A_{pt} \subseteq shared \cup pt_{[i]} \cup I.R \wedge \\ &\quad I.A_{pt} \cap I.R_{pt} = \emptyset \wedge I.R_{pt} \subseteq pt_{[i]} \end{aligned}$$

Another predicate collects some basic safety properties for the ownership transfer of instruction $I \in \mathbb{I}_{sb}$, which are needed for reordering of this transfer after an SB step of other thread:

$$\begin{aligned} &safe\text{-}otran(c_{sbh}, i, I) = (vW(I) \vee RMW(I) \vee vR(I)) \wedge \\ &\quad I.L \subseteq I.A \wedge I.R \subseteq O_{[i]} \cup acq(sb_{[i]}) \wedge I.R_{pt} \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}) \wedge \\ &\quad (\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset) \wedge \\ &\quad (\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset) \end{aligned}$$

Sets of temporaries used for reading by instructions in instruction list is , store buffer sb or ϑ are defined as

$$load_t(is) = \bigcup\{is[k].t \mid k < |is| \wedge (R(is[k]) \vee RMW(is[k]))\}$$

A store operation (D, f), where the function f maps temporaries to a value and D specifies the subset of temporaries, is valid iff f only depends on the temporaries specified by D:

$$valid\text{-}sop((D, f)) = \forall \vartheta.\ D \subseteq \mathrm{dom}(\vartheta) \rightarrow f(\vartheta) = f(\vartheta\ \restriction_D)$$

Ownership Invariants

oinv1. For every thread non-volatile writes in SB must refer to the owned memory. Reads in the suspended part of the SB have to be owned or refer to read-only memory. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then:

$$oinv1(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ (nvW(I) \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, i).O_{[i]}) \wedge$$
$$(nvR(I) \wedge k \geq |exec(sb_{[i]})| \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, i).O_{[i]} \cup \delta^k_{sb}(c_{sbh}, i).ro)$$

oinv2. Every outstanding volatile write is neither owned by any other thread and nor in other thread's PT set:

$$oinv2(c_{sbh}) = \forall i.\ \forall I \in sb_{[i]}.\ vW(I) \rightarrow$$
$$I.pa \notin \bigcup_{j \neq i} (O_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}))$$

oinv3. In the suspended part of the store buffer outstanding accesses to read-only memory are not in the accumulated ownership sets of others. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then

$$oinv3(c_{sbh}) = \forall i.\ \forall j \neq i.\ \forall k.\ k < |sb_{[i]}| \wedge k \geq |exec(sb_{[i]})| \wedge nvR(I) \wedge$$
$$I.pa \in \delta^k_{sb}(c_{sbh}, i).ro \rightarrow I.pa \notin (O_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}))$$

oinv4. The ownership sets of every two different threads are distinct:

$$oinv4(c_{sbh}) = \forall i.\ \forall j \neq i.\ (O_{[i]} \cup acq(sb_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

Sharing Invariants

sinv1. All outstanding non-volatile writes are unshared. Let $I = sb_{[i]}[k]$, then

$$sinv1(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ nvW(I) \rightarrow I.pa \notin \delta^k_{sb}(c_{sbh}, i).shared$$

sinv2. All unshared addresses are owned or are in PT sets:

$$sinv2(c_{sbh}) = \forall a \notin shared \rightarrow \exists i.\ a \in O_{[i]} \cup pt_{[i]}$$

sinv3. No thread owns read-only memory and read-only memory is shared:

$$sinv3(c_{sbh}) = ro \subseteq shared \wedge \forall i.\ O_{[i]} \cap ro = \emptyset$$

sinv4. The ownership annotations of outstanding ghost and volatile write operations are consistent:

$$sinv4(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ safe\text{-}annot(\delta^k_{sb}(c_{sbh}, i), i, sb_{[i]}[k])$$

sinv5. There are no outstanding writes to read-only memory:

$$sinv5(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ W(sb_{[i]}[k]) \rightarrow sb_{[i]}[k].pa \notin \delta^k_{sb}(c_{sbh}, i).ro$$

Invariants on Temporaries

tinv1. The temporaries used for loads in the instruction list are distinct:

$$tinv1(c_{sbh}) = \forall k < |is_{[i]}|.\ load_t(is_{[i]}[0:k]) \cap load_t(is_{[i]}[k+1 : |is_{[i]}| - 1]) = \emptyset$$

tinv2. The temporaries used for loads in the store buffer are distinct:

$$tinv2(c_{sbh}) = \forall k < |sb_{[i]}|.\ load_t(sb_{[i]}[0:k]) \cap load_t(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) = \emptyset$$

tinv3. The temporaries used for loads in an instruction list are fresh, i.e., are not in the domain of ϑ.

$$tinv3(c_{sbh}) = \forall i.\ load_t(is_{[i]}) \cap \mathrm{dom}(\vartheta_{[i]}) = \emptyset$$

Data Dependency Invariants

dinv1. Every store (D, f) in the instruction list or the store buffer is valid according to *valid-sop*:

$$dinv1(c_{sbh}) = \forall i.\ \forall I \in sb_{[i]} \circ is_{[i]}.\ (W(I) \vee RMW(I)) \rightarrow valid\text{-}sop(I.(D, f))$$

dinv2. Domain D of a store instruction in the instruction list is a subset of previous read temporaries. Let $I = is_{[i]}[k]$, then

$$dinv2(c_{sbh}) = \forall i.\ \forall k < |is_{[i]}|.\ (W(I) \vee RMW(I)) \rightarrow I.D \subseteq dom(\vartheta_{[i]}) \cup load_t(is_{[i]}[0:k])$$

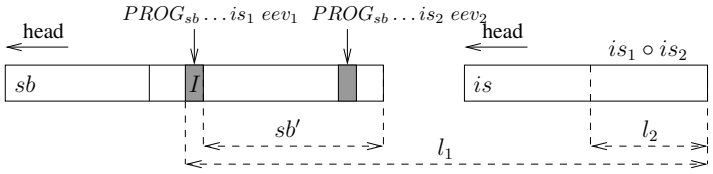


Figure 2.5: Store buffer and instruction list layout in hinv5.

History Invariants

hinv1. In the suspended part of the SB the value stored for a non volatile read is the same as the last write to the same address in the SB or the value in memory, in case there is no hitting write in the buffer. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then

$$hinv1(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ k \geq |exec(sb_{[i]})| \wedge nvR(I) \rightarrow I.v = I.ext(\delta^k_{sb}(c_{sbh}, i).m(I.pa), I.bw).$$

hinv2. There are no volatile reads in the suspended part of the store buffer:

$$hinv2(c_{sbh}) = \forall i.\ \forall I \in susp(sb_{[i]}).\ \neg vR(I)$$

hinv3. For every read the recorded value and physical address coincide with the corresponding value in the temporaries.

$$hinv3(c_{sbh}) = \forall i.\ \forall I \in (sb_{[i]}).\ R(I) \rightarrow (I.v, I.pa) = \vartheta_{[i]}(I.t)$$

hinv4. For every write in a store buffer the recorded value v coincides with $f(\vartheta_{[i]})$ and domain D is a subset of previous read temporaries. Let $I = sb_{[i]}[k]$, then

$$hinv4(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ W(I) \rightarrow I.f(\vartheta_{[i]}) = I.v \wedge I.D \subseteq dom(\vartheta_{[i]}) \setminus load_t(sb[k+1 : |sb_{[i]}| - 1])$$

hinv5. History information for program steps in the store buffer is consistent.
Let $I = sb_{[i]}[k]$, $sb' = sb_{[i]}[k+1 : |sb_{[i]}| - 1]$, $l_1 = ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]}$ and $l_2 = p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1])$ (see Fig. 2.5) then

$$\begin{aligned} hinv5(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ P(I) \rightarrow I.p_2 = hd\text{-}p(p_{[i]}, sb') \wedge \\ \delta_p(I.p_1, del\text{-}t(\vartheta_{[i]}, sb'), mode_{[i]}, mmu_{[i]}, I.is_1, I.eev) = (I.p_2, I.is_2) \wedge \\ I.is_1 = l_1[0 : |l_1| - |l_2| - 1] \end{aligned}$$

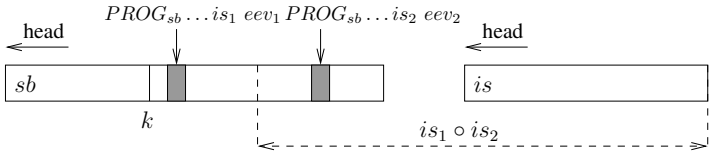


Figure 2.6: Relating generated instructions with instructions in the store buffer and in the instruction list.

hinv6. Any suffix of the store buffer concatenated with the instruction list contains the instructions generated by the program steps in this suffix (see Fig. 2.6). Let $sb' = sb[k : |sb_{[i]}| - 1]$, then

$$hinv6(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ \exists is'.\ ins(sb') \circ is_{[i]} = is' \circ p\text{-}ins(sb')$$

hinv7. Ownership annotations of volatile write instructions in the store buffer are consistent. Let $I = sb_{[i]}[k]$ and $sb' = sb_{[i]}[k : |sb_{[i]}| - 1]$ then

$$hinv7(c_{sbh}) = \forall i, k.\ vW(I) \rightarrow I.annot = og(I.p, del\text{-}t(\vartheta_{[i]}, sb'))$$

MMU Invariant

minv1. In translated mode the physical address of an instruction in the store buffer is present in the current address translation set:

$$minv1(c_{sbh}) = \forall i.\ \forall I \in sb_{[i]}.\ (R(I) \vee W(I)) \rightarrow I.pa \in \text{atran}(mmu_{[i]}, I.va, mode_{[i]}, I.r)$$

Page Table Invariants

pinv1. Page table sets of different threads do not overlap:

$$pinv1(c_{sbh}) = \forall i, j.\ i \neq j \rightarrow (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset$$

pinv2. Page table sets and ownership sets of different threads do not overlap:

$$pinv2(c_{sbh}) = \forall i, j.\ i \neq j \rightarrow (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

pinv3. Page table sets and the shared set do not overlap:

$$pinv3(c_{sbh}) = \forall i.\ pt_{[i]} \cap shared = \emptyset$$

pinv4. Page table set and ownership sets of one thread are disjoint:

$$pinv4(c_{sbh}) = \forall i.\ pt_{[i]} \cap O_{[i]} = \emptyset$$

2.3.5 Assumptions on Program Steps

We introduce a number of assumptions on program steps, which guarantee that the read temporaries are always fresh for every new read instruction. Let $\delta_p(p_{[i]}, \vartheta_{[i]}, is_{[i]}, eev) = (p', is')$, then

1. load temporaries in is' are distinct:

$$\forall k < |is'|.\ load_t(is'[0:k]) \cap load_t(is'[k+1:|is'|-1]) = \emptyset$$

2. load temporaries in is' are distinct from load temporaries in $is_{[i]}$ and $\vartheta_{[i]}$

$$load_t(is') \cap (load_t(is_{[i]}) \cup dom(\vartheta_{[i]})) = \emptyset$$

3. store instructions in is' are valid and their domains only depend on the previously generated load temporaries:

$$\forall k < |is'|.\ I = is'_{[k]} \wedge (W(I) \vee RMW(I)) \rightarrow valid\text{-}sop(I.(D, f)) \wedge$$
$$I.D \subseteq load_t(is'[0:k]) \cup load_t(is_{[i]}) \cup \mathrm{dom}(\vartheta_{[i]})$$

2.3.6 Proof Strategy

We split the proof of Theorem 2.29 into two parts. In the first part we assume safety of the delayed release and in step-by-step fashion show that the coupling invariant is maintained after every step of the SB machine.

Theorem 2.33 (SB Simulation)

$$c_{sbh} \underset{eev}{\Longrightarrow} c'_{sbh} \wedge c_{sbh} \sim c \wedge safe\text{-}reach_d(c, og) \wedge inv(c_{sbh}) \rightarrow$$
$$inv(c'_{sbh}) \wedge (\exists c'.\ c \underset{eev}{\overset{*}{\Longrightarrow}} c' \wedge c'_{sbh} \sim c')$$

The proof of Theorem 2.33 is also split into two parts. In Sect. 2.4 we show that invariants are maintained after every step of the SB machine and in Sect. 2.5 we prove the simulation. When proving that invariants and the coupling relation are maintained, we show only those properties, which can possibly get broken by the step. Those invariants and parts of the coupling relation which we do not consider explicitly are trivially maintained after the step. Note, that all invariants we defined talk about the content of the SB and trivially hold in the initial configuration, i.e. in the case when SBs of all threads are empty.

In the second part of the proof of Theorem 2.29 we show that safety of the delayed release can be derived from regular safety of the abstract machine.

Theorem 2.34 (Safety)

$$initial(c) \wedge safe\text{-}reach(c, og) \rightarrow safe\text{-}reach_d(c, og)$$

This proof is given in Sect. 2.6.

2.4 Maintaining Invariants

In this section we show that the invariants are maintained after every step of the SB machine. We define the accumulated ownership set of thread i as:

$$acc_{ownpt[i]} = O_{[i]} \cup acq(sb_{[i]}) \cup pt_{[i]} \cup acq_{pt}(sb_{[i]})$$
$$acc'_{ownpt[i]} = O'_{[i]} \cup acq(sb'_{[i]}) \cup pt'_{[i]} \cup acq_{pt}(sb'_{[i]})$$

2.4.1 SB Steps

Lemma 2.35 (accumulated ownership sets shrink after δ_{sb})

$$c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c'_{sbh} \wedge inv(c_{sbh}) \rightarrow acc'_{ownpt[i]} \subseteq acc_{ownpt[i]}$$

Proof We do a case split on the SB step. If it does not perform the ownership transfer then the lemma is trivially concluded. Otherwise we let $I = hd(sb_{[i]})$ then from the semantics:

$$O'_{[i]} = O_{[i]} \cup I.A \setminus I.R$$
$$pt'_{[i]} = pt_{[i]} \cup I.A_{pt} \setminus I.R_{pt}$$

From the definition of acq and acq_{pt} we have:

$$acq(sb_{[i]}) = acq(sb'_{[i]}) \cup I.A$$
$$acq_{pt}(sb_{[i]}) = acq_{pt}(sb'_{[i]}) \cup I.A_{pt}$$

We can get:

$$\begin{aligned}
O'_{[i]} \cup acq(sb'_{[i]}) &= (O_{[i]} \cup I.A \setminus I.R) \cup acq(sb'_{[i]}) \\
&= (O_{[i]} \setminus I.R \cup I.A) \cup acq(sb'_{[i]}) && (sinv4(c_{sbh}) \text{ implies } I.A \cap I.R = \emptyset) \\
&= O_{[i]} \setminus I.R \cup (I.A \cup acq(sb'_{[i]})) && \text{(associativity)} \\
&= O_{[i]} \setminus I.R \cup (acq(sb'_{[i]}) \cup I.A) && \text{(commutativity)} \\
&= O_{[i]} \setminus I.R \cup acq(sb_{[i]}) \\
&\subseteq O_{[i]} \cup acq(sb_{[i]})
\end{aligned} \tag{2.36}$$

With identical steps we can also get:

$$pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}) \tag{2.37}$$

The lemma is concluded by (2.36) and (2.37). □

Lemma 2.38 (invariants maintained by δ_{sb})

$$\forall i.\ inv(c_{sbh}) \rightarrow inv(\delta_{sb}(c_{sbh}, i))$$

Proof We let $I = hd(sb_{[i]})$ and $c'_{sbh} = \delta_{sb}(c_{sbh}, i)$. From the semantics of the SB step we have

$$sb'_{[i]} = tl(sb_{[i]}) \quad \text{and} \quad is'_{[i]} = is_{[i]}.$$

We consider only the invariants which are affected by the SB step. For all invariants except *hinv*6, *hinv*1 and *hinv*4 we only consider case $vR(I) \vee vW(I)$. For *hinv*1 we only consider case $W(I)$. For *hinv*4 we consider cases $W(I)$ and $R(I)$.

- *oinv*1. Let $I = sb_{[j]}[k]$, then from $oinv1(c_{sbh})$ we have for all threads j

$$\begin{aligned}&\forall k < |sb_{[j]}|.\ (nvW(I) \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, j).O_{[j]}) \wedge \\ &\quad (nvR(I) \wedge k \geq |exec(sb_{[j]})| \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, j).O_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro).\end{aligned}$$

  For case $j = i$ the property is trivially maintained. For $j \neq i$ the first statement of the invariant also cannot be broken by a step of thread i. For the second statement we have to show for non-volatile reads $I^j = sb_{[j]}[k]$ that

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro.$$

  From $oinv3(c_{sbh})$ that

$$I^j.pa \notin acc_{ownpt[i]}$$

  Hence,

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I^j.pa \notin I.A \cup I.A_{pt}$$

  $I^j.pa$ can not be acquired by thread i and remains in the read only set.

- *oinv*2. From *oinv*2 we have for all threads j

$$\forall I \in sb_{[j]}.\ vW(I) \rightarrow I.pa \notin \bigcup_{k \neq j} acc_{ownpt[k]}$$

  For case $j = i$ the property is trivially maintained. For $j \neq i$ from lemma 2.35 we have

$$acc'_{ownpt[i]} \subseteq acc_{ownpt[i]}$$

  which implies

$$\bigcup_{k \neq j} acc'_{ownpt[k]} \subseteq \bigcup_{k \neq j} acc_{ownpt[k]}$$

  Thus, we have

$$\forall I \in sb_{[j]}.\ vW(I) \rightarrow I.pa \notin \bigcup_{k \neq j} acc'_{ownpt[k]}$$

- *oinv*3. From $oinv3(c_{sbh})$ we have for all threads j:

  $$\forall j' \neq j.\ |exec(sb_{[j]})| \leq k < |sb_{[j]}| \wedge R(I) \wedge I.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I.pa \notin acc_{ownpt[j']}$$

  For case $j = i$ the property is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and

  $$R(I^j) \wedge I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro.$$

  From lemma 2.35 we can conclude of SB steps we know that

  $$\forall j'.\ acc'_{ownpt[j']} \subseteq acc_{ownpt[j']}$$

  Hence, all we have to show is

  $$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro. \tag{2.39}$$

  If $I^j.pa \notin c'_{sbh}.ro$, then it is released by thread j later and (2.39) trivially holds. If $I^j.pa \in c'_{sbh}.ro$ and $I^j.pa \notin c_{sbh}.ro$, then

  $$I^j.pa \in I.R \subseteq O_{[i]}.$$

  From $oinv1(c_{sbh})$ and $hinv2(c_{sbh})$ we know that

  $$I^j.pa \in O_{[j]} \cup acq(sb_j) \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

  If $I^j.pa \in O_{[j]} \cup acq(sb_j)$, we get a contradiction from $oinv4(c_{sbh})$. Hence, we can conclude

  $$I^j.pa \in c_{sbh}.ro \vee I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro.$$

  With $I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro$, (2.39) obviously holds.

- *oinv*4. From $oinv4(c_{sbh})$ we have:

  $$\forall j \neq i.\ (O_{[i]} \cup acq(sb_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

  From the definition of *acq* and semantics of the SB step, we have:

  $$\begin{aligned} O'_{[i]} &\subseteq O_{[i]} \cup I.A \\ &\subseteq O_{[i]} \cup acq(sb_{[i]}) \end{aligned}$$

  $$acq(sb'_{[i]}) \subseteq acq(sb_{[i]}).$$

  Thus, we can conclude:

  $$\forall j \neq i.\ (O'_{[i]} \cup acq(sb'_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

  Since the configuration of other threads is unchanged in c'_{sbh}, we get

  $$oinv4(c'_{sbh}).$$

- *sinv*1. From $sinv1(c_{sbh})$ we have for all threads j

$$\forall k < |sb_{[j]}|.\ I = sb_{[j]}[k] \wedge nvW(I) \rightarrow I.pa \notin \delta^k_{sb}(c_{sbh}, j).shared.$$

For case $j = i$ the property is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $nvW(I^j)$. We have from $oinv1(c_{sbh})$

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \subseteq \mathcal{O}_{[j]} \cup acq(sb_{[j]}).$$

From $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ we can conclude:

$$(\mathcal{O}_{[i]} \cup acq(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset \wedge$$
$$(pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

From $sinv4(c_{sbh})$ and the semantics of SB steps we get

$$\begin{aligned}\delta^k_{sb}(c'_{sbh}, j).shared &\subseteq \delta^k_{sb}(c_{sbh}, j).shared \cup I.R \cup I.R_{pt}\\ &\subseteq \delta^k_{sb}(c_{sbh}, j).shared \cup \mathcal{O}_{[i]} \cup pt_{[i]}.\end{aligned}$$

Hence, the step of thread i can not make address $I^j.pa$ shared and the invariant is maintained.

- *sinv*2. From $sinv2(c_{sbh})$ we have

$$\forall a \notin shared \rightarrow \exists j.\ a \in \mathcal{O}_{[j]} \cup pt_{[j]}.$$

We do a case split:

  - $a \notin shared \wedge a \notin shared'$. Hence,

$$\exists j.\ a \in \mathcal{O}_{[j]} \cup pt_{[j]}.$$

    If $j \neq i$, then the statement is trivially maintained. If $j = i$, we assume

$$a \notin \mathcal{O}'_{[i]} \cup pt'_{[i]}$$

    and prove by contradiction.

    * if $a \in \mathcal{O}_{[i]}$, then we have from the semantics and from $sinv4(c_{sbh})$

$$a \in I.R \wedge a \notin I.A \wedge a \notin I.L \wedge a \notin I.A_{pt},$$

      which implies $a \in shared'$ and gives a contradiction.

    * if $a \in pt_{[i]}$, then

$$a \in I.R_{pt} \wedge a \notin I.A_{pt} \wedge a \notin I.A \wedge a \notin I.L,$$

      which again implies $a \in shared'$ and gives a contradiction.

– $a \in shared \wedge a \notin shared'$. Using $sinv4(c_{sbh})$ we get

$$a \in I.L \cup I.A_{pt} \subseteq I.A \cup I.A_{pt} \subseteq O'_{[j]} \cup pt'_{[j]}.$$

- $sinv3$. From $sinv3(c_{sbh})$, we have

$$\forall j.\, O_{[j]} \cap ro = \emptyset \wedge ro \subseteq shared.$$

From the semantics, we have:

$$\begin{aligned} ro' &= ro \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}) \\ &\subseteq ro \cup (I.R \setminus I.W) \setminus I.A \\ O'_{[i]} &= O_{[i]} \cup I.A \setminus I.R \\ &\subseteq O_{[i]} \cup I.A \setminus (I.R \setminus I.W) \end{aligned}$$

From $sinv4(c_{sbh})$ we have $I.A \cap I.R = \emptyset$. Thus, we can conclude

$$O'_{[i]} \subseteq O_{[i]} \setminus (I.R \setminus I.W) \cup I.A$$

We can also conclude:

$$(ro \cup (I.R \setminus I.W) \setminus I.A) \cap (O_{[i]} \setminus (I.R \setminus I.W) \cup I.A) = \emptyset$$

which gives us

$$ro' \cap O'_{[i]} = \emptyset$$

We instantiate k in $sinv4(c_{sbh})$ with 0 and can get

$$I.R \subseteq O_{[i]} \wedge I.L \subseteq I.A$$

With $oinv4(c_{sbh})$ we get

$$\forall j \neq i.\, O_{[i]} \cap O_{[j]} = \emptyset,$$

which implies

$$I.R \cap O_{[j]} = \emptyset$$

From the semantics we have $O_{[j]} = O'_{[j]}$. Therefore, we can get

$$I.R \cap O'_{[j]} = \emptyset$$

We can conclude

$$\forall j.\, O'_{[j]} \cap ro' = \emptyset.$$

For the shared set we have from the semantics

$$\begin{aligned} shared' &= shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt}) \\ &\supseteq share \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}) \\ &\supseteq ro \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}) \\ &= ro' \end{aligned}$$

- *sinv*4. From $sinv4(c_{sbh})$ we have

$$\forall j.\ \forall k < |sb_{[j]}|.\ safe\text{-}annot(\delta^k_{sb}(c_{sbh}, j), j, sb_{[j]}[k]).$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $vW(I^j) \vee vR(I^j)$. The local ownership sets of thread j remain unchanged. Hence, all we have to show is

$$I^j.A \cap \delta^k_{sb}(c_{sbh}, j).shared = I^j.A \cap \delta^k_{sb}(c'_{sbh}, j).shared, \tag{2.40}$$
$$I^j.A_{pt} \cap \delta^k_{sb}(c_{sbh}, j).shared = I^j.A_{pt} \cap \delta^k_{sb}(c'_{sbh}, j).shared. \tag{2.41}$$

We first show (2.40). From the semantics of SB steps we have

$$shared' = shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt})$$

$\forall j.\ \forall I^j \in sb_{[j]}$ we write X^j as a shorthand to $I^j.X$. From $sinv4(c_{sbh})$ we can conclude

$$\forall I^j.\in sb_{[j]}.\ L^j \subseteq A^j \wedge (A^j \cup A^j_{pt}) \subseteq acc_{ownpt[j]} \wedge (R^j \cup R^j_{pt}) \subseteq acc_{ownpt[j]} \tag{2.42}$$

With $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can get the accumulated ownership set of thread i and thread j are disjoint.

$$acc_{ownpt[i]} \cap acc_{ownpt[j]} = \emptyset \tag{2.43}$$

Thus, the ownership annotation $(A, L, R, W, A_{pt}, R_{pt})$ of every instruction in $sb_{[i]}$ and $sb_{[j]}$ do not overlap. We can reorder the ownership transfer of thread i after the ownership transfer of thread j. That concludes:

$$\delta^k_{sb}(c'_{sbh}, j).shared = \delta^k_{sb}(c_{sbh}, j).shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt})$$

From (2.42) and (2.43) we get

$$I^j.A \cap (I.L \cup I.A_{pt}) = \emptyset$$
$$I^j.A \cap (I.R \cup I.R_{pt}) = \emptyset.$$

Moreover we conclude (2.40). The proof of (2.41) is completely analogous if one takes $pinv1(c_{sbh})$ instead of $oinv4(c_{sbh})$.

- *sinv*5. From $sinv5(c_{sbh})$ we have

$$\forall j.\ \forall k < |sb_{[j]}|.\ W(sb_{[j]}[k]) \rightarrow sb_{[j]}[k].pa \notin \delta^k_{sb}(c_{sbh}, j).ro.$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $W(I^j)$. We have from the semantics of SB steps and $sinv4(c_{sbh})$

$$\begin{aligned}\delta^k_{sb}(c'_{sbh}, j).ro &\subseteq \delta^k_{sb}(c_{sbh}, j).ro \cup I.R \\ &\subseteq \delta^k_{sb}(c_{sbh}, j).ro \cup O_{[i]}.\end{aligned}$$

We consider cases:

– $nvW(I^j)$. With $oinv1(c_{sbh})$ we have

$$I^j.pa \in O_{[j]} \cup acq(sb_{[j]}).$$

From $oinv4(c_{sbh})$ we conclude

$$O_{[i]} \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset \wedge I^j.pa \notin O_{[i]}$$

Hence,

$$I^j.pa \notin \delta^k_{sb}(c'_{sbh}, j).ro.$$

– $vW(I^j)$. The proof as before follows from $oinv2(c'_{sbh})$.

- $hinv1$. From $hinv1(c_{sbh})$ we have:

$$\forall j.\ \forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge I = sb_{[j]}[k] \wedge$$
$$nvR(I) \rightarrow I.v = I.ext(\delta^k_{sb}(c_{sbh}, j).m(I.pa), I.bw).$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $nvR(I^j)$. From $oinv1(c_{sbh})$ we have

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).O_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

The only step of thread i which can change the memory content is $W(I)$.

We now do a case split on $I^j.pa$ and show that $I.pa \neq I^j.pa$.

– $I^j.pa \in \delta^k_{sb}(c_{sbh}, j).O_{[j]}$. Hence,

$$I^j.pa \in O_{[j]} \cup acq(sb_{[j]}).$$

We do case split on I.

* $nvW(I)$. From $oinv1(c_{sbh})$ we get

$$I.pa \in O_{[i]} \cup acq(sb_{[i]})$$

From $oinv4(c_{sbh})$ we get

$$(O_{[i]} \cup acq(sb_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

Thus,

$$I.pa \notin O_{[j]} \cup acq(sb_{[j]})$$

* $vW(I)$. From $oinv2(c_{sbh})$ we can get

$$I.pa \notin O_{[j]} \cup acq(sb_{[j]})$$

In both cases we can get $I.pa \neq I^j.pa$.

– $I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro$. In this case we do a further case split on I.

* $nvW(I)$. From $oinv3(c_{sbh})$ we get

$$I^j.pa \notin (O_{[i]} \cup acq(sb_{[i]}) \cup pt_{[i]} \cup acq_{pt}(sb_{[i]})).$$

From $oinv1(c_{sbh})$ we have

$$I.pa \in O_{[i]} \cup acq(sb_{[i]}).$$

Hence, $I.pa \neq I^j.pa$

* $vW(I)$. If $I^j.pa \in c_{sbh}.ro$, from $sinv5(c_{sbh})$ we can get

$$I.pa \notin c_{sbh}.ro.$$

If $I^j \notin c_{sbh}.ro$, we can conclude

$$I^j.pa \in O_{[j]} \cup acq(sb_{[j]}).$$

From $oinv2(c_{sbh})$, we get $I.pa \neq I^j.pa$.

Finally, we can get

$$\delta^k_{sb}(c_{sbh}, j).m(I^j.pa) = \delta^k_{sb}(c'_{sbh}, j).m(I^j.pa)$$

and concludes the proof.

- $hinv6$. From $hinv6(c_{sbh})$ we have for all $k < |sb_{[i]}|$:

$$\exists is.\ ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

For all $k' < |sb'_{[i]}|$ we get for all prefixes is:

$$ins(sb'_{[i]}[k' : |sb'_{[i]}| - 1]) \circ is'_{[i]} = ins(sb_{[i]}[k' + 1 : |sb_{[i]}| - 1]) \circ is_{[i]}$$
$$is \circ p\text{-}ins(sb'_{[i]}[k' : |sb'_{[i]}| - 1]) = is \circ p\text{-}ins(sb_{[i]}[k' + 1 : |sb_{[i]}| - 1]).$$

Hence, we instantiate k in $hinv6(c_{sbh})$ with $k' + 1$ and get the proof for $hinv6(c'_{sbh})$.

- $pinv1$. From $pinv1(c_{sbh})$, we have:

$$\forall j \neq i.\ (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

From the definition of acq_{pt} and the semantics of the SB step, we have:

$$pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}).$$

Since the configuration of other threads is unchanged in c'_{sbh}, we get

$$pinv1(c'_{sbh})$$

- *pinv*2. From $pinv2(c_{sbh})$, we have:

$$\forall i.\forall j \neq i.(pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

With similar prove steps in lemma 2.35 we can get

$$\begin{aligned} pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) &\subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}) \\ O'_{[i]} \cup acq(sb'_{[i]}) &\subseteq O_{[i]} \cup acq(sb_{[i]}), \end{aligned}$$

which implies $pinv2(c'_{sbh})$.

- *pinv*3. From $pinv3(c_{sbh})$, we have:

$$\forall j.\ pt_{[j]} \cap shared = \emptyset.$$

From the semantics we get

$$\begin{aligned} pt'_{[i]} &= pt_{[i]} \cup I.A_{pt} \setminus I.R_{pt} \\ shared' &\subseteq shared \cup I.R \cup I.R_{pt} \setminus I.A_{pt}. \end{aligned}$$

From $sinv4(c_{sbh})$ and $pinv4(c_{sbh})$ we know that

$$pt_{[i]} \cap I.R = \emptyset.$$

Hence, all new addresses which are added to the shared set are not present in $pt'_{[i]}$. Addresses $I.A_{pt}$ which are added to the pt set, are excluded from the shared set. Therefore, we have

$$pt'_{[i]} \cap shared' = \emptyset.$$

For $j \neq i$ we have from $sinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$

$$pt_{[j]} \cap I.R = pt_{[j]} \cap I.R_{pt} = \emptyset.$$

Thus,

$$pt'_{[j]} \cap shared' = \emptyset.$$

- *pinv*4. From $pinv4(c_{sbh})$, we have:

$$pt_{[i]} \cap O_{[i]} = \emptyset.$$

For $vR(I) \vee vW(I)$ we get

$$\begin{aligned} pt'_{[i]} &= pt_{[i]} \cup I.A_{pt} \setminus I.R_{pt} \\ O'_{[i]} &= O_{[i]} \cup I.A \setminus I.R \end{aligned}$$

By instantiating k in $sinv4(c_{sbh})$ with 0 we can get:

$$I.A_{pt} \cap I.R_{pt} = I.A \cap I.R = \emptyset$$

Thus we can have

$$pt'_{[i]} = pt_{[i]} \setminus I.R_{pt} \cup I.A_{pt}$$
$$O'_{[i]} = O_{[i]} \setminus I.R \cup I.A$$

We let

$$\mathcal{A} = pt_{[i]} \setminus I.R_{pt}$$
$$\mathcal{B} = I.A_{pt}$$
$$\mathcal{C} = O_{[i]} \setminus I.R$$
$$\mathcal{D} = I.A$$

then

$$\begin{aligned} pt'_{[i]} \cap O'_{[i]} &= (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{C} \cup \mathcal{D}) \\ &= ((\mathcal{A} \cup \mathcal{B}) \cap \mathcal{C}) \cup ((\mathcal{A} \cup \mathcal{B}) \cap \mathcal{D}) && \text{(distributivity)} \\ &= ((\mathcal{A} \cap \mathcal{C}) \cup (\mathcal{B} \cap \mathcal{C})) \cup ((\mathcal{A} \cap \mathcal{D}) \cup (\mathcal{B} \cap \mathcal{D})) && \text{(distributivity)} \\ &= (\mathcal{A} \cap \mathcal{C}) \cup (\mathcal{B} \cap \mathcal{C}) \cup (\mathcal{A} \cap \mathcal{D}) \cup (\mathcal{B} \cap \mathcal{D}) && \text{(associativity)} \end{aligned}$$

With $pinv4(c_{sbh})$ we can conclude

$$\mathcal{A} \cap \mathcal{C} = \emptyset$$

With $pinv2(c_{sbh})$ we can conclude

$$\mathcal{B} \cap \mathcal{C} = \mathcal{A} \cap \mathcal{D} = \emptyset$$

By instantiating k in $sinv4(c_{sbh})$ with 0 we get

$$\mathcal{B} \cap \mathcal{D} = \emptyset$$

and conclude the proof.

□

2.4.2 Commutativity of SB Steps

The following function applies the ownership transfer of instruction I in thread i to the provided configuration of the SB machine:

$$otran\text{-}sbh(c_{sbh}, i, I) = c_{sbh}[ghst[i] := otran(c_{sbh}.ghst[i], I)].$$

Lemma 2.44 (ownership transfer commute)

$$inv(c_{sbh}) \wedge safe\text{-}otran(c_{sbh}, i, I) \wedge i \neq j \rightarrow$$
$$\delta_{sb}(otran\text{-}sbh(c_{sbh}, i, I), j) = otran\text{-}sbh(\delta_{sb}(c_{sbh}, j), i, I)$$

Proof The case $|sb_{[j]}| = 0$ is trivial. Otherwise, let I^j denote the first instruction in $sb_{[j]}$:

$$I^j = sb_{[j]}[0].$$

For $I^j.A, I^j.R, \ldots$ we abbreviate $A^j, R^j, \ldots$ and for $I.A, I.R, \ldots$ we write $A, R, \ldots$. We set

$$c'_{sbh} = \delta_{sb}(otran\text{-}sbh(c_{sbh}, i, I), j) \quad \text{and} \quad c''_{sbh} = otran\text{-}sbh(\delta_{sb}(c_{sbh}, j), i, I).$$

- For components $c_{sbh}.X$, where $X \in \{shared, ro\}$ we only have to consider cases when $vR(I^j) \vee vW(I^j)$. From definitions of δ_{sb} and the ownership transfer we have:

$$c'_{sbh}.shared = shared \cup R_{pt} \cup R \setminus (L \cup A_{pt}) \cup R^j_{pt} \cup R^j \setminus (L^j \cup A^j_{pt})$$
$$c'_{sbh}.ro = ro \cup (R \setminus W) \setminus (A \cup A_{pt}) \cup (R^j \setminus W^j) \setminus (A^j \cup A^j_{pt}).$$

  We define:

$$acc_{ownpt[j]} = O_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]})$$

  From $sinv4(c_{sbh})$ and definitions of acq and acq_{pt} we can conclude

$$L^j \subseteq A^j \wedge (A^j \cup A^j_{pt}) \subseteq acc_{ownpt[j]} \wedge (R^j \cup R^j_{pt}) \subseteq acc_{ownpt[j]} \tag{2.45}$$

  From $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we know that the accumulated ownership sets $acc_{ownpt[i]}$ and $acc_{ownpt[j]}$ are disjoint. Predicate $safe\text{-}otran(c_{sbh}, i, I)$ guarantees that release sets of instruction I (i.e. $R \cup R_{pt}$) and acquire sets of instruction I (i.e. $A \cup A_{pt}$) do not overlap with $acc_{ownpt[j]}$. Hence we can conclude that acquire and release sets of instructions I and I^j do not overlap:

$$\begin{aligned}(L \cup A \cup A_{pt}) \cap (R^j \cup R^j_{pt}) &= \emptyset \\ (L^j \cup A^j \cup A^j_{pt}) \cap (R \cup R_{pt}) &= \emptyset \\ (R \cup R_{pt}) \cap (R^j \cup R^j_{pt}) &= \emptyset\end{aligned} \tag{2.46}$$

  Hence,

$$\begin{aligned}c'_{sbh}.shared &= shared \cup R_{pt} \cup R \cup R^j_{pt} \cup R^j \setminus (L \cup A_{pt} \cup L^j \cup A^j_{pt}) \\ &= shared \cup R^j_{pt} \cup R^j \cup R_{pt} \cup R \setminus (L^j \cup A^j_{pt} \cup L \cup A_{pt}) \\ &= c''_{sbh}.shared \\ c'_{sbh}.ro &= ro \cup (R \setminus W) \cup (R^j \setminus W^j) \setminus (A \cup A_{pt}) \setminus (A^j \cup A^j_{pt}) \\ &= ro \cup (R^j \setminus W^j) \cup (R \setminus W) \setminus (A^j \cup A^j_{pt}) \setminus (A \cup A_{pt}) \\ &= c''_{sbh}.ro\end{aligned}$$

- For thread local components of configurations $c_{sbh}.ts$ only the release sets might get affected by the reordering in case $vR(I^j) \vee vW(I^j)$. We first consider case $vR(I) \wedge vR(I^j)$. For the shared release set we have

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
c''_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap (shared \cup R^j \cup R^j_{pt} \setminus (L^j \cup A^j_{pt}))) \\
c'_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap (shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}))) \\
c''_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap shared)
\end{aligned}$$

  Hence, with (2.46) we have

$$\begin{aligned}
c'_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap shared) \\
&= c''_{sbh}.rls_{s[j]} \\
c''_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
&= c'_{sbh}.rls_{s[i]}.
\end{aligned}$$

  For case $(vW(I) \vee RMW(I)) \wedge vR(I^j)$ we conclude

$$\begin{aligned}
c'_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap (shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}))) \\
&= rls_{s[j]} \cup (R^j \cap shared) \\
&= c''_{sbh}.rls_{s[j]} \\
c'_{sbh}.rls_{s[i]} &= c''_{sbh}.rls_{s[i]} = \emptyset.
\end{aligned}$$

  For case $(vW(I) \vee RMW(I)) \wedge vW(I^j)$ we obviously get

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= c''_{sbh}.rls_{s[i]} = \emptyset \\
c'_{sbh}.rls_{s[j]} &= c''_{sbh}.rls_{s[j]} = \emptyset.
\end{aligned}$$

  For case $vR(I) \wedge vW(I^j)$ we conclude

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
&= rls_{s[i]} \cup (R \cap (shared \cup R^j \cup R^j_{pt} \setminus (L^j \cup A^j_{pt}))) \\
&= c''_{sbh}.rls_{s[i]} \\
c'_{sbh}.rls_{s[j]} &= c''_{sbh}.rls_{s[j]} = \emptyset.
\end{aligned}$$

  The proof for the equality of the local and page table release sets is completely analogous.

□

Lemma 2.47 (ownership transfer safe for SB instruction)

$$inv(c_{sbh}) \rightarrow safe\text{-}otran(c_{sbh}, i, hd(sb_{[i]}))$$

Proof The proof immediately follows from $sinv4(c_{sbh})$, $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ as well as definition of acq and acq_{pt}. □

Lemma 2.48 (δ_{sb} commute)

$$inv(c_{sbh}) \wedge \neg vW(hd(sb_{[i]})) \rightarrow \delta_{sb}(\delta_{sb}(c_{sbh}, i), j) = \delta_{sb}(\delta_{sb}(c_{sbh}, j), i)$$

Proof The case when one of the SBs is empty or $i = j$ is trivial. Otherwise, for $k \in \{i, j\}$ let I^k denote the first instruction in $sb_{[k]}$:

$$I^k = sb_{[k]}[0].$$

We set

$$c'_{sbh} = \delta_{sb}(\delta_{sb}(c_{sbh}, i), j) \quad \text{and} \quad c''_{sbh} = \delta_{sb}(\delta_{sb}(c_{sbh}, j), i).$$

With Lemma 2.47 we conclude

$$safe\text{-}otran(c_{sbh}, i, I^i).$$

Applying Lemma 2.44 we get the equality of all ownership and release sets in configurations c'_{sbh} and c''_{sbh}. Hence, the only part which is left to show is the equality of the memory component. The only interesting case here is $nvW(I^i) \wedge W(I^j)$. We show that $I^i.pa \neq I^j.pa$. From $oinv1(c_{sbh})$ we can conclude:

$$I^i.pa \in O_{[i]}.$$

We now do a case distinctions on I^j. In case $nvW(I^j)$ we conclude from $oinv1(c_{sbh})$ and $oinv4(c_{sbh})$

$$I^j.pa \in O_{[j]} \quad \text{and} \quad I^j.pa \notin O_{[i]}.$$

In case $vW(I^j)$ we use $oinv2(c_{sbh})$ to directly conclude:

$$I^j.pa \notin O_{[i]}.$$

□

Lemma 2.49 ($\delta^k_{sb}, \Delta^{exec}_{sb}$ commute)

$$\forall k \leq |sb_{[i]}|.\ inv(c_{sbh}) \rightarrow \delta^k_{sb}(\Delta^{exec}_{sb}(c_{sbh}, j), i) = \Delta^{exec}_{sb}(\delta^k_{sb}(c_{sbh}, i), j)$$

Proof For case $|exec(sb_{[j]})| = 0$ or $k = 0$ the proof is trivial. Otherwise, let

$$c'_{sbh} = \delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}, j), i).$$

Lemma 2.38 guarantees that invariants are maintained by any number of SB steps. Applying Lemma 2.48 we can reorder the last step of thread j after the first step of thread i:

$$c'_{sbh} = \delta_{sb}(\delta_{sb}(\delta^{|exec(sb_{[j]})-1|}_{sb}(c_{sbh}, j), i), j).$$

Performing the same action $|exec(sb_{[j]})|$ times we move the step of thread i before all steps of thread j:

$$c'_{sbh} = \Delta^{exec}_{sb}(\delta_{sb}(c_{sbh}, i), j).$$

To reorder all k steps of thread i we have to repeat this procedure k times, resulting in $k \times |exec(sb_{[j]})|$ applications of lemma 2.48. □

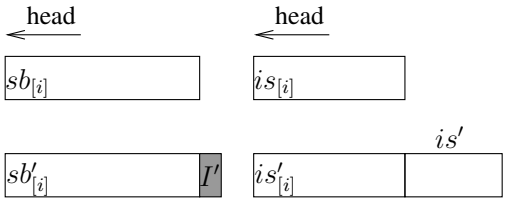


Figure 2.7: Program step

Lemma 2.50 (Δ^{exec}_{sb} commute)

$$\begin{aligned}\forall i.\ inv(c_{sbh}) \rightarrow\ & \Delta^{exec}_{sb}(c_{sbh}) = \Delta^{exec}_{sb}(\Delta^{exec}_{sb}(c_{sbh}, i)) \wedge \\ & \Delta^{exec}_{sb}(c_{sbh}) = \Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i) \wedge \\ & \Delta^{exec}_{sb}(c_{sbh}) = \Delta^{exec}_{sb[\neq i]}(\Delta^{exec}_{sb}(c_{sbh}, i)) \wedge \\ & \Delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}), i) = \Delta^{exec}_{sb}(\Delta_{sb}(c_{sbh}, i)) \wedge \\ & \forall k \leq |sb_{[i]}|.\ \delta^{k}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i) = \Delta^{exec}_{sb[\neq i]}(\delta^{k}_{sb}(c_{sbh}, i))\end{aligned}$$

Proof The proof follows directly from lemmas 2.49 and 2.38. □

2.4.3 Program Step

Lemma 2.51 (invariants maintained by program step)

$$inv(c_{sbh}) \wedge c_{sbh} \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i\ c'_{sbh} \rightarrow inv(c'_{sbh})$$

Proof Let $I' = PROG\ p_{[i]}\ p'_{[i]}\ is_{[i]}\ is'\ eev$ then from the semantics we have $sb'_{[i]} = sb_{[i]} \circ I'$ and $is'_{[i]} = is_{[i]} \circ is'$ (see Fig. 2.7).

- Invariants dealing with temporaries (i.e., *tinv*1, *tinv*3, *dinv*1, *dinv*2) are easily maintained using the assumptions on the program state.
- *hinv*5. From $hinv5(c_{sbh})$ we have

$$\begin{aligned}\forall k < |sb_{[i]}|.\ P(I) \rightarrow\ & I.p_2 = hd\text{-}p(p_{[i]}, tl(sb_1)) \wedge \\ & \delta_p(I.p_1, del\text{-}t(\vartheta_{[i]}, tl(sb_1)), mode_{[i]}, mmu_{[i]}, I.is_1, I.eev) = (I.p_2, I.is_2) \wedge \\ & I.is_1 = l_1[0 : |l_1| - |l_2| - 1]\end{aligned}$$

  where: $I = sb_{[i]}[k]$, $sb_1 = sb_{[i]}[k : |sb_{[i]}| - 1]$, $l_1 = ins(sb_1) \circ is_{[i]}$ and $l_2 = p\text{-}ins(sb_1)$. For the newly recorded program step we have

$$\begin{aligned}& I'.p_2 = p'_{[i]} = hd\text{-}p(p'_{[i]}, []) \wedge \\ & \delta_p(p_{[i]}, \vartheta_{[i]}, mode_{[i]}, mmu_{[i]}, is_{[i]}, eev) = (p'_{[i]}, is') = (I'.p_2, I'.is_2) \wedge \\ & is_{[i]} = is'_{[i]}[0 : |is'_{[i]}| - |is'| - 1]\end{aligned}$$

and the required property holds. Since no new read instruction are added to the store buffer, we have $\vartheta_{[i]} = \vartheta'_{[i]}$ and

$$\forall k < |sb_{[i]}|.\ load_t(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) = load_t(sb'_{[i]}[k+1 : |sb'_{[i]}| - 1]).$$

Hence, the second statement of the invariant is maintained for all program steps, that were in the store buffer before the step. Let $sb_2 = sb'_{[i]}[k : |sb'_{[i]}| - 1]$, $l'_1 = ins(sb_2) \circ is'_{[i]}$ and $l'_2 = p\text{-}ins(sb_2)$ then from the semantics of program step we can conclude:

$$l'_1 = l_1 \circ is' \ \wedge\ l'_2 = l_2 \circ is'$$

Thus, we can conclude

$$\forall k \leq |sb_{[i]}|.\ P(sb_{[i]}[k]) \rightarrow l_1[0 : |l_1| - |l_2| - 1] = l'_1[0 : |l'_1| - |l'_2| - 1]$$

We now consider cases:

- case I is the last program instruction in $sb_{[i]}$. Then we have

$$\begin{aligned} I.p_2 &= hd\text{-}p(p_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \\ &= p_{[i]} \\ &= hd\text{-}p(p'_{[i]}, sb'_{[i]}[k+1 : |sb'_{[i]}| - 1]). \end{aligned}$$

- case I is not the last program instruction in $sb_{[i]}$. Then we have

$$\begin{aligned} I.p_2 &= hd\text{-}p(p_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \\ &= hd\text{-}p(p'_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \\ &= hd\text{-}p(p'_{[i]}, sb'_{[i]}[k+1 : |sb'_{[i]}| - 1]). \end{aligned}$$

This concludes the proof for *hinv*5.

- *hinv*6. From *hinv*6(c_{sbh}) we have for all $k < |sb_{[i]}|$:

$$\exists is.\ ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1])$$

After adding a program step to the store buffer we have for all $k < |sb_{[i]}|$:

$$\begin{aligned} ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) &= ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \\ is'_{[i]} &= is_{[i]} \circ is' \\ p\text{-}ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) &= p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is' \end{aligned}$$

and the invariant holds if we choose the same prefix is, as we had before the step. For $k = |sb_{[i]}|$ we have

$$\begin{aligned} ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) &= [] \\ is'_{[i]} &= is_{[i]} \circ is' \\ p\text{-}ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) &= is' \end{aligned}$$

and the invariant holds if we set $is = is_{[i]}$. This concludes the proof for *hinv*6.

□

2.4.4 Memory Steps

In case of FENCE, INVLPG, mode switch and write to PTO memory steps the store buffer of thread i is empty. Hence, invariant $minv1$ is trivially maintained. Other invariants can not possibly be broken by the step.

Lemma 2.52 (safe execution maintains disjoint sets on abstract machine)

$$disjoint\text{-}osets(c) \wedge c \underset{\text{eev}}{\Rightarrow}^{*} c' \wedge safe\text{-}reach(c, og) \rightarrow disjoint\text{-}osets(c')$$

Proof We prove this lemma by induction on the length of the computation. If $c = c'$ there is nothing to prove. Otherwise, we assume

$$disjoint\text{-}osets(c) \wedge safe\text{-}reach(c, og)$$

as the induction hypothesis and have to prove

$$\forall c', i.\ c \underset{\text{eev}}{\Rightarrow}_i c' \rightarrow disjoint\text{-}osets(c').$$

If we do not perform the ownership transfer, it is trivially true. Otherwise, we assume that $I = hd(c.is_{[i]})$ performs the ownership transfer. Let $og(I.p, c'.\vartheta_{[i]}) = (A, L, R, W, A_{pt}, R_{pt})$ then from $safe\text{-}reach(c, og)$ we conclude $safe\text{-}state(c, og)$ which infers:

$$L \subseteq A.$$

With the definition of the ownership transfer we have:

$$\begin{aligned} c'.ro &= c.ro \cup (R \setminus W) \setminus (A \cup A_{pt}) \\ c'.shared &= c.shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}). \end{aligned}$$

With the induction hypothesis we can conclude:

$$c'.ro \subseteq c'.shared.$$

From $disjoint\text{-}osets(c)$ we trivially get for all $j \neq i$ and $k \neq i$

$$\begin{aligned} j \neq k \rightarrow c'.O_{[k]} \cap c'.O_{[j]} = \emptyset \ \wedge\ c'.O_{[k]} \cap c'.pt_{[j]} = \emptyset \\ c'.pt_{[k]} \cap c'.pt_{[j]} = \emptyset \ \wedge\ c'.O_{[k]} \cap c'.pt_{[k]} = \emptyset. \end{aligned}$$

From $safe\text{-}state(c, og)$ we have:

$$R \subseteq c.O_{[i]} \ \wedge\ R_{pt} \subseteq c.pt_{[i]}.$$

With the induction hypothesis, we can get:

$$\forall k \neq i.\ c'.O_{[k]} \cap c'.ro = \emptyset \ \wedge\ c'.pt_{[k]} \cap c'.shared = \emptyset.$$

From the definition of the ownership transfer we also have:

$$c'.\mathcal{O}_{[i]} = c.\mathcal{O}_{[i]} \cup A \setminus R$$
$$c'.pt_{[i]} = c.pt_{[i]} \cup A_{pt} \setminus R_{pt}.$$

From $safe\text{-}state(c, og)$, we have:

$$\forall j \neq i.\ (A \cup A_{pt}) \cap (c.\mathcal{O}_{[j]} \cup c.pt_{[j]}) = \emptyset.$$

With the induction hypothesis we can conclude:

$$\forall j \neq i.\ c'.\mathcal{O}_{[i]} \cap c'.\mathcal{O}_{[j]} = \emptyset \ \wedge\ c'.\mathcal{O}_{[i]} \cap c'.pt_{[j]} = \emptyset \wedge$$
$$c'.pt_{[i]} \cap c'.pt_{[j]} = \emptyset.$$

From $safe\text{-}state(c, og)$ we also have:

$$A_{pt} \cap A = \emptyset.$$

Thus, with the induction hypothesis we can conclude:

$$c'.\mathcal{O}_{[i]} \cap c'.pt_{[i]} = \emptyset.$$

With the definition of ownership transfer we can also conclude:

$$c'.\mathcal{O}_{[i]} \cap c'.ro = \emptyset \ \wedge\ c'.pt_{[i]} \cap c'.shared = \emptyset.$$

□

Lemma 2.53 (coupling implies disjoint sets)

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \rightarrow disjoint\text{-}osets(c)$$

Proof Lemma 2.38 implies

$$inv(\Delta_{sb}^{exec}(c_{sbh})).$$

The statement of the lemma follows immediately from the coupling relations and from invariants *oinv*4, *pinv*1, *pinv*2, *pinv*3, *pinv*4, and *sinv*3. □

In the following proof, we will use a proof technique that advances the computation of the abstract machine till a certain instruction in the instruction sequence of thread i. During the advancing, the step performed by abstract machine depends on the history information in the $susp(sb_{[i]})$. To show the consistency of ownership annotations between the abstract machine and the SB machine during the advancing, we define the intermediate coupling relation.

Definition 2.54 (Intermediate Coupling Relation)

$$
\begin{aligned}
&sim(c, c_{sbh}, i, k) = \\
&\quad \forall X \in \{shared, ro, m\}.\ n = k + |exec(sb_{[i]})| \wedge c.X = \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X \wedge \\
&\quad \forall j.\ c.mode_{[j]} = mode_{[j]} \wedge c.mmu_{[j]} = mmu_{[j]} \wedge \\
&\qquad ((c.\mathcal{D}_{[j]} \vee \exists I \in sb_{[j]}.\ vW(I)) \leftrightarrow \mathcal{D}_{[j]}) \ \wedge \forall X \in \{O, pt, rls_l, rls_s, rls_{pt}\}. \\
&\qquad (j \neq i \rightarrow c.X_{[j]} = \Delta^{exec}_{sb}(c_{sbh}, j).X_{[j]} \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(susp(sb_{[j]})) = ins(susp(sb_{[j]})) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, susp(sb_{[j]})) \wedge c.p_{[j]} = hd\text{-}p(p_{[j]}, susp(sb_{[j]}))) \wedge \\
&\qquad (j = i \rightarrow c.X_{[j]} = \delta^n_{sb}(c_{sbh}, j).X_{[j]} \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(sb_{[j]}[n : |sb_{[j]}| - 1]) = ins(sb_{[j]}[n : |sb_{[j]}| - 1]) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, sb_{[j]}[n : |sb_{[j]}| - 1]) \wedge \\
&\qquad\qquad c.p_{[j]} = hd\text{-}p(p_{[j]}, sb_{[j]}[n : |sb_{[j]}| - 1]))
\end{aligned}
$$

In case $k = 0$, relation $sim(c, c_{sbh}, i, 0) \equiv c_{sbh} \sim c$. In case $k = 1$, relation $sim(c, c_{sbh}, i, 1)$ couples the states of the SB machine and after the volatile write instruction is executed in the virtual machine. Note that if $n > |sb_{[j]}| - 1$ in case $j = i$, $sb_{[j]}[n : |sb_{[j]}| - 1]$ becomes [].

In Lemma 2.55, Lemma 2.56, Lemma 2.59 and Lemma 2.60 we prove that in the executed portion of $sb_{[j]}$ there is no non volatile write to the target address of read operation in the suspended portion of $sb_{[i]}$. Thus, the read value is consistent in both machines because the generation of ownership annotations only depend on the read values and the instructions. We could prove that the ownership annotations are consistent.

Lemma 2.55 (instruction list not empty)

$$
\begin{aligned}
c.is_{[i]} \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) = ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} \wedge \\
k < |sb_{[i]}| \wedge hinv6(c_{sbh}) \wedge \neg P(sb_{[i]}[k]) \rightarrow c.is_{[i]} \neq []
\end{aligned}
$$

Proof If $k = |sb_{[i]}| - 1$ then we get

$$c.is_{[i]} = ins(sb_{[i]}[k]) \circ is_{[i]}$$

and the lemma trivially holds. Otherwise we prove by contradiction. Let $I = sb_{[i]}[k]$ and $c.is_{[i]} = []$. Then we conclude

$$
\begin{aligned}
p\text{-}ins(sb_{[i]}[k + 1 : |sb_{[i]}| - 1]) &= p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \\
&= ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} \\
&= ins(I) \circ ins(sb_{[i]}[k + 1 : |sb_{[i]}| - 1]) \circ is_{[i]}.
\end{aligned}
$$

From $hinv6(c_{sbh})$ we know that there exists is' such that

$$
\begin{aligned}
&ins(sb_{[i]}[k + 1 : |sb_{[i]}| - 1]) \circ is_{[i]} \\
&= is' \circ p\text{-}ins(sb_{[i]}[k + 1 : |sb_{[i]}| - 1]) \\
&= is' \circ ins(I) \circ ins(sb_{[i]}[k + 1 : |sb_{[i]}| - 1]) \circ is_{[i]},
\end{aligned}
$$

which gives a contradiction. □

Lemma 2.56 (unowned nvW is in local release set)

$$\forall j, k, n, pa.\ k < |exec(sb_{[j]})| \wedge n \leq |susp(sb_{[i]})| \wedge inv(c_{sbh}) \wedge \\ (c_{sbh} \sim c \vee sim(c, c_{sbh}, i, n) \wedge i \neq j) \wedge nvW(sb_{[j]}[k]) \wedge \\ pa = sb_{[j]}[k].pa \wedge (pa \notin c.O_{[j]} \vee pa \in c.shared) \rightarrow pa \in c.rls_{l[j]}$$

Proof From $oinv1(c_{sbh})$ and $sinv1(c_{sbh})$ we have:

$$pa \in \delta^k_{sb}(c_{sbh}, j).O_{[j]} \wedge pa \notin \delta^k_{sb}(c_{sbh}, j).shared.$$

The coupling relations for thread j gives us

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \vee pa \in \Delta^{exec}_{sb}(c_{sbh}).shared. \tag{2.57}$$

or

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \vee pa \in \delta^{n+|exec(sb_{[i]})|}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared \tag{2.58}$$

depending of what kind of simulation relation holds. From $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ it follows for all threads $l \neq j$:

$$pa \notin acc_{ownpt[l]}$$

Hence, with $sinv4(c_{sbh})$ we can conclude that pa is not released by any instruction in $sb_{[l]}$. Thus, from (2.57) we get

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \vee pa \in \Delta^{exec}_{sb}(c_{sbh}, j).shared$$

- Let $pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]}$. In $sb_{[j]}[k : |exec(sb_{[j]})| - 1]$ there must be an instruction, which removes pa from the owns set of thread j. When an address is removed from the ownership set, it is added to one of the release sets. In order to be added to the shared release set, the address has to be shared at the time of the ownership transfer. We have already shown that no thread other than j can make pa shared. The only way for thread j to make an owned unshared address shared, is by releasing it, which puts the address to the local release set.
- Let $pa \in \Delta^{exec}_{sb}(c_{sbh}, j).shared$. In $sb_{[j]}[k : |exec(sb_{[j]})| - 1]$ there has to be an instruction, which releases pa and adds it to the local release set.

Hence, we can conclude

$$pa \in \Delta^{exec}_{sb}(c_{sbh}, j).rls_{l[j]} = c.rls_{l[j]}$$

For case (2.58) we do analogous prove and conclude the lemma. □

Lemma 2.59 (sim implies disjoint sets)

$$\forall i, k.\ k \leq |susp(sb_{[i]})| \wedge sim(c, c_{sbh}, i, k) \wedge inv(c_{sbh}) \rightarrow disjoint\text{-}osets(c)$$

Proof For thread i we have to prove:

$$c.O_{[i]} \cap c.ro = \emptyset$$
$$c.pt_{[i]} \cap c.shared = \emptyset.$$

Let $n = k+|exec(sb_{[i]})|$ then with the help of the intermediate coupling relation this is transformed to

$$\delta^n_{sb}(c_{sbh}, i).O_{[i]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro = \emptyset$$
$$\delta^n_{sb}(c_{sbh}, i).pt_{[i]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = \emptyset.$$

With Lemma 2.50 we have:

$$\delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro = \Delta^{exec}_{sb[\neq i]}(\delta^n_{sb}(c_{sbh}, i)).ro$$
$$\delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = \Delta^{exec}_{sb[\neq i]}(\delta^n_{sb}(c_{sbh}, i)).shared.$$

From the semantics we can conclude:

$$\Delta^{exec}_{sb[\neq i]}(\delta^n_{sb}(c_{sbh}, i)).ro \subseteq \delta^n_{sb}(c_{sbh}, i).ro \bigcup_{\forall j \neq i} rels(exec(sb_{[j]}))$$
$$\Delta^{exec}_{sb[\neq i]}(\delta^n_{sb}(c_{sbh}, i)).shared \subseteq$$
$$\delta^n_{sb}(c_{sbh}, i).shared \bigcup_{\forall j \neq i} rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})).$$

With $sinv3(c_{sbh})$, $pinv3(c_{sbh})$ and Lemma 2.38 we have:

$$\delta^n_{sb}(c_{sbh}, i).O_{[i]} \cap \delta^n_{sb}(c_{sbh}, i).ro = \emptyset$$
$$\delta^n_{sb}(c_{sbh}, i).pt_{[i]} \cap \delta^n_{sb}(c_{sbh}, i).shared = \emptyset.$$

With $oinv4(c_{sbh})$ we can get:

$$\forall j \neq i.\ \delta^n_{sb}(c_{sbh}, i).O_{[i]} \cap (O_{[j]} \cup acq(exec(sb_{[j]}))) = \emptyset.$$

With $sinv4(c_{sbh})$ we can conclude:

$$rels(exec(sb_{[j]})) \subseteq (O_{[j]} \cup acq(exec(sb_{[j]})))$$

We can conclude:

$$\forall j \neq i.\ \delta^n_{sb}(c_{sbh}, i).O_{[i]} \cap rels(exec(sb_{[j]})) = \emptyset$$

With $sinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can also get in an analogous way that:

$$\forall j \neq i.\ \delta^n_{sb}(c_{sbh}, i).pt_{[i]} \cap (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset.$$

Thus, with the intermediate coupling relation we can conclude:

$$c.O_{[i]} \cap c.ro = \emptyset$$
$$c.pt_{[i]} \cap c.shared = \emptyset.$$

For thread $j \neq i$ we have to prove:

$$c.O_{[j]} \cap c.ro = \emptyset,$$
$$c.pt_{[j]} \cap c.shared = \emptyset.$$

With the help of the intermediate coupling relation this is transformed to

$$\Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \cap \delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = \emptyset.$$

We prove this case by contradiction. Assume

$$\exists a, a'.\ a \in \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \cap \delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro \wedge$$
$$a' \in \Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared.$$

With Lemma 2.50 we can get:

$$\Delta^{exec}_{sb}(c_{sbh}).ro \subseteq \Delta^{exec}_{sb}(c_{sbh}, j).ro \bigcup_{\forall k \neq j} rels(exec(sb_{[k]}))$$
$$\Delta^{exec}_{sb}(c_{sbh}).shared \subseteq$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).shared \bigcup_{\forall k \neq j} rels(exec(sb_{[k]})) \cup rels_{pt}(exec(sb_{[k]})).$$

With Lemma 2.38, $sinv3(c_{sbh})$ and $pinv3(c_{sbh})$

$$\Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}, j).ro = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}, j).shared = \emptyset.$$

With analogous step of the previous case we can conclude:

$$\forall k \neq j.\ \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \cap rels(exec(sb_{[k]}) = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap (rels(exec(sb_{[k]})) \cup rels_{pt}(exec(sb_{[k]}))) = \emptyset$$

After that we can conclude:

$$\Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}).ro = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}).shared = \emptyset.$$

With Lemma 2.50 and $sinv4(c_{sbh})$ we can conclude

$$a \in acq(sb[|exec(sb_{[i]})| : n]) \wedge a' \in acq_{pt}(sb[|exec(sb_{[i]})| : n]).$$

This contradicts to $oinv4(c_{sbh})$ and $pinv1(c_{sbh})$. Thus, using the intermediate coupling relation again we can conclude:

$$c.O_{[j]} \cap c.ro = \emptyset$$
$$c.pt_{[j]} \cap c.shared = \emptyset.$$

The remaining properties follow immediately from Lemma 2.38, the intermediate coupling relation and and invariants $oinv4(c_{sbh})$, $pinv1(c_{sbh})$, $pinv2(c_{sbh})$, $pinv4(c_{sbh})$ and $sinv3(c_{sbh})$. □

Lemma 2.60 (no nvW to a read address)

$$\begin{aligned}\forall i, pa.\ & (R(hd(c.is_{[i]})) \vee RMW(hd(c.is_{[i]}))) \wedge n \leq |susp(sb_{[i]})| \wedge \\ & pa \in (\text{atran}(c.mmu_{[i]}, hd(c.is_{[i]}).va, c.mode_{[i]}, hd(c.is_{[i]}).r)) \wedge \\ & (c_{sbh} \sim c \vee sim(c, c_{sbh}, i, n)) \wedge safe\text{-}reach_d(c, og) \wedge \text{inv}(c_{sbh}) \ \rightarrow \\ & \forall j \neq i.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = pa)\end{aligned}$$

Proof By contradiction. Assume

$$\exists j \neq i.\ \exists k < |exec(sb_{[j]})|.\ sb_{[j]}[k] = \textbf{Write}_{\textbf{sb}}\ \text{False va (D, f) r cb bw p annot pa v.}$$

Applying Lemma 2.56 we get

$$pa \notin c.O_{[j]} \vee pa \in c.shared \rightarrow c.rls_{l[j]}.$$

From the safety for reads and RMWs we get

$$pa \in c.O_{[i]} \cup c.shared \cup c.ro \cup c.pt_{[i]} \ \wedge \forall j \neq i.\ pa \notin c.rls_{l[j]}.$$

Hence, we can conclude

$$pa \in c.O_{[j]} \wedge pa \notin c.shared.$$

Applying Lemma 2.53 or Lemma 2.59 we conclude

$$pa \notin c.O_{[i]} \cup c.pt_{[i]} \cup c.ro.$$

and get a contradiction. □

Lemma 2.61 (simulating execution of sb inductive)

$$\begin{aligned}&\forall k.\ 0 \leq k < |susp(sb_{[i]})| \wedge sim(c, c_{sbh}, i, k) \wedge inv(c_{sbh}) \wedge \\ &safe\text{-}reach_d(c, og) \rightarrow \exists c'.\ c \overset{\text{p,m}}{\underset{\text{eev}}{\Longrightarrow_i}} c' \wedge sim(c', c_{sbh}, i, k+1)\end{aligned}$$

Proof For the base case, we have to prove:

$$\begin{aligned}&c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \ \rightarrow \\ &\exists c'.\ c \overset{\text{p,m}}{\underset{\text{eev}}{\Longrightarrow_i}} c' \wedge sim(c', c_{sbh}, i, 1)\end{aligned}$$

From Lemma 2.55 we conclude

$$c.is_{[i]} \neq [].$$

Let $I = hd(susp(sb_{[i]}))$ from the definition of $susp$ we can get $vW(I)$ then from the coupling relation we have

$$hd(c.is_{[i]}) = ins(I).$$

From the coupling relation and from $minv1(c_{sbh})$ we conclude

$$I.pa \in \text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

Let c'' be configuration of the abstract machine after the step:

$$c \overset{\text{m}}{\Longrightarrow}_i c''.$$

For the temporaries and the program state we obviously have

$$\begin{aligned} c''.\vartheta_{[i]} &= c.\vartheta_{[i]} \\ &= del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]})) \\ &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}[|exec(sb_{[i]})| + 1 : |sb_{[i]}| - 1]) \end{aligned}$$

Let $I' = hd(c.is_{[i]})$ then from $hinv7(c_{sbh})$ and the coupling relation we can conclude:

$$\begin{aligned} I.annot &= og(I.p, del\text{-}t(\vartheta_{[i]}, sb_{[i]}[|exec(sb_{[i]})| : |sb_{[i]}| - 1])) \\ &= og(I.p, del\text{-}t(\vartheta_{[i]}, sb_{[i]}[|exec(sb_{[i]})| + 1 : |sb_{[i]}| - 1])) \quad (vW \text{ does not change the temp}) \\ &= og(I'.p, c''.\vartheta) \end{aligned}$$

Hence, we can always execute the volatile write from the head of the instruction list and choose the same translated address as we have previously chosen for the corresponding step of the SB machine. The ownership transfer of abstract machine is also performed according to the ownership annotations recorded in the corresponding instruction of the SB machine.

From the coupling relation and the semantics of abstract and SB machines we get for $X \in \{shared, ro, m\}$:

$$\begin{aligned} c''.X &= \delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).X \\ &= \delta_{sb}^{|exec(sb_{[i]})|+1}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).X \quad \text{(Lemma 2.50)} \end{aligned}$$

For $X \in \{O, pt, rls_{pt}\}$ we get:

$$c''.X_{[i]} = \delta_{sb}^{|exec(sb_{[i]})|+1}(c_{sbh}, i).X$$

The SB machine only accumulates local up-dates on shared set when computing the release local and release shared set. However the abstract machine with delayed release accumulates global updates on shared set while computing the corresponding components. Hence, the coupling relation for release shared and release local set can get broken. We have to prove:

$$c''.rls_{s[i]} = \delta_{sb}^{|exec(sb_{[i]})|+1}(c_{sbh}, i).rls_{s[i]}$$

From the semantics of the abstract machine we have:

$$\begin{aligned} c''.rls_{s[i]} &= c.rls_{s[i]} \cup (I.R \setminus c.shared) \\ &= \Delta_{sb}^{exec}(c_{sbh}, i).rls_{s[i]} \cup (I.R \setminus \Delta_{sb}^{exec}(c_{sbh}).shared) \qquad \text{(coupling relation)} \end{aligned}$$

From the semantics of the store buffer machine we have:

$$\begin{aligned}
&\delta_{sb}^{|exec(sb_{[i]})|+1}(c_{sbh}, i).rls_{s[i]} \\
&= \delta_{sb}^{|exec(sb_{[i]})|}(c_{sbh}, i).rls_{s[i]} \cup (I.R \setminus \delta_{sb}^{|exec(sb_{[i]})|}(c_{sbh}, i).shared) \\
&= \Delta_{sb}^{exec}(c_{sbh}, i).rls_{s[i]} \cup (I.R \setminus \Delta_{sb}^{exec}(c_{sbh}, i).shared) \qquad \text{(defintion of } \Delta)
\end{aligned}$$

Thus, we have to prove the following equation:

$$I.R \cap \Delta_{sb}^{exec}(c_{sbh}).shared = I.R \cap \Delta_{sb}^{exec}(c_{sbh}, i).shared \tag{2.62}$$

From the $sinv4(c_{sbh})$ we can conclude:

$$\begin{aligned}
\Delta_{sb}^{exec}(c_{sbh}).shared \subseteq \Delta_{sb}^{exec}(c_{sbh}, i).shared \cup \\
(\bigcup_{\forall j \neq i}(rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))))
\end{aligned}$$

$$\begin{aligned}
\Delta_{sb}^{exec}(c_{sbh}).shared \supseteq \Delta_{sb}^{exec}(c_{sbh}, i).shared \setminus \\
(\bigcup_{\forall j \neq i}(acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))))
\end{aligned}$$

With $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and the definition of acq and acq_{pt}, we can conclude:

$$\begin{aligned}
I.R \cap \bigcup_{\forall j \neq i}(rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset \\
I.R \cap \bigcup_{\forall j \neq i}(acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset
\end{aligned}$$

Thus, we can conclude:

$$\begin{aligned}
I.R \cup \Delta_{sb}^{exec}(c_{sbh}).shared \subseteq I.R \cap \Delta_{sb}^{exec}(c_{sbh}, i).shared \\
I.R \cup \Delta_{sb}^{exec}(c_{sbh}).shared \supseteq I.R \cap \Delta_{sb}^{exec}(c_{sbh}, i).shared
\end{aligned}$$

which implies (2.62).
For the instruction sequence we conclude with the help of the coupling relation:

$$\begin{aligned}
&c''.is_{[i]} \circ p\text{-}ins(sb_{[i]}[|exec(sb_{[i]})| + 1 : |sb_{[i]}| - 1]) \\
&= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb_{[i]})) \\
&= tl(c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]}))) && \text{(def of } tl) \\
&= tl(ins(susp(sb_{[i]})) \circ is_{[i]}) && \text{(coupling relation)} \\
&= tl(ins(susp(sb_{[i]}))) \circ is_{[i]} && \text{(def of } tl) \\
&= ins(sb_{[i]}[|exec(sb_{[i]})| + 1 : |sb_{[i]}| - 1]) \circ is_{[i]} && \text{(def of } tl)
\end{aligned}$$

$$
\begin{aligned}
& c''.p_{[i]} \\
&= c.p_{[i]} && \text{(semantics)} \\
&= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]})) && \text{(coupling relation)} \\
&= \mathit{hd\text{-}p}(p_{[i]}, sb_{[i]}[|\mathit{exec}(sb_{[i]})| + 1 : |sb_{[i]}| - 1]) && \text{(def of } \mathit{hd\text{-}p} \text{ and } \mathit{vW}(I))
\end{aligned}
$$

From the intermediate coupling relation we get $\mathcal{D}_{[i]}$. From the semantics of the memory step we also get $c''.\mathcal{D}_{[i]}$. This implies

$$(c''.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ \mathit{vW}(I)) \leftrightarrow \mathcal{D}_{[i]}.$$

For induction step we assume following induction hypothesis:

$$\forall k.\ 0 \leq k < |\mathit{susp}(sb_{[i]})| \wedge \mathit{sim}(c, c_{sbh}, i, k) \wedge \mathit{inv}(c_{sbh}) \wedge \mathit{safe\text{-}reach}_d(c, og)$$

Let $n = |\mathit{exec}(sb_{[i]})| + k$ then for induction step, we do a case split on $I = sb_{[i]}[n]$ and execute either a memory or a program step of thread i depending on $sb_{[i]}[n]$:

- case $\neg P(I)$. From Lemma 2.55 we conclude

  $$c.is_{[i]} \neq [].$$

  Hence, from the coupling relation we have

  $$\mathit{hd}(c.is_{[i]}) = \mathit{hd}(\mathit{ins}(sb_{[i]}[n : |sb_{[i]}| - 1])) = \mathit{ins}(I).$$

  If I is a read or a write instruction, then from $\mathit{sim}(c, c_{sbh}, i, k)$ and from $\mathit{minv}1(c_{sbh})$ we get

  $$I.pa \in \mathrm{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

  Hence, as the induction base case we can always execute the instruction from the head of the instruction list by choosing the same translated address and performing the same ownership transfer as we have previously chosen for the corresponding step of the SB machine. Let c' be the configuration of the abstract machine after the step:

  $$c \overset{\mathrm{m}}{\Longrightarrow}_i c'.$$

  For the instruction sequence after the step we obviously get

  $$
  \begin{aligned}
  & c'.is_{[i]} \circ \mathit{p\text{-}ins}(sb_{[i]}[n + 1 : |sb_{[i]}| - 1]) \\
  &= \mathit{tl}(c.is_{[i]}) \circ \mathit{p\text{-}ins}(sb_{[i]}[n : |sb_{[i]}| - 1]) \\
  &= \mathit{tl}(c.is_{[i]} \circ \mathit{p\text{-}ins}(sb_{[i]}[n : |sb_{[i]}| - 1])) \\
  &= \mathit{tl}(\mathit{ins}(sb_{[i]}[n : |sb_{[i]}| - 1]) \circ is_{[i]}) \\
  &= \mathit{tl}(\mathit{ins}(sb_{[i]}[n : |sb_{[i]}| - 1])) \circ is_{[i]} \\
  &= \mathit{ins}(sb_{[i]}[n + 1 : |sb_{[i]}| - 1]) \circ is_{[i]}
  \end{aligned}
  $$

  The coupling for mode, dirty flag, MMU state and program state is trivially maintained, as well as the coupling for threads other than i. For the remaining parts of the coupling relation we do a further case split:

– $W(I)$. From the semantics we get

$$c'.m(I.pa) = I.cb(I.f(c.\vartheta_{[i]}), c.m(I.pa), I.bw).$$

From $hinv4(c_{sbh})$ we know that

$$I.v = I.f(\vartheta_{[i]}).$$

From $hinv4(c_{sbh})$, $dinv1(c_{sbh})$ and the coupling relation we get

$$\begin{aligned} I.f(\vartheta_{[i]}) &= I.f(del\text{-}t(\vartheta_{[i]}, sb_{[i]}[n : |sb_{[i]}| - 1])) \\ &= I.f(c.\vartheta_{[i]}). \end{aligned}$$

From $sim(c, c_{sbh}, i, k)$ we get

$$c.m(I.pa) = \delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m(I.pa)$$

Hence, we can conclude

$$c'.m(I.pa) = \delta^{n+1}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m(I.pa)$$

which implies

$$c'.m = \delta^{n+1}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m$$

and concludes the proof for the memory coupling. For $vW(I)$ we have to prove the coupling of release set and the identity of the ownership annotations. This can be proved by the similar prove technique in the induction base.

– $nvR(I)$. In this case the coupling for the temporaries can get broken. From the semantics we get

$$c'.\vartheta_{[i]} = c.\vartheta_{[i]}(I.t \mapsto (I.ext(c.m(I.pa), I.bw), I.pa)).$$

From $hinv1(c_{sbh})$ we know that

$$I.v = I.ext(\delta^{n}_{sb}(c_{sbh}, i).m(I.pa), I.bw).$$

With Lemma 2.60 we get for all $j \neq i$:

$$\forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = I.pa)$$

Hence, with the intermediate coupling relation we have

$$\begin{aligned} I.ext(c.m(pa), I.bw) &= I.ext(\delta^{n}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m(pa), I.bw) \\ &= I.ext(\delta^{n}_{sb}(c_{sbh}, i).m(I.pa), I.bw) \\ &= I.v \end{aligned}$$

From invariant $hinv3(c_{sbh})$ we have

$$\vartheta_{[i]}(I.t) = (I.v, I.pa)$$

From the semantics and from the coupling relation we now conclude

$$\begin{aligned} c'.\vartheta_{[i]} &= c.\vartheta_{[i]}(I.t \mapsto (I.v, I.pa)) \\ &= \mathit{del\text{-}t}(\vartheta_{[i]}, sb_{[i]}[n : |sb_{[i]}| - 1])(I.t \mapsto (I.v, I.pa)) \\ &= \mathit{del\text{-}t}(\vartheta_{[i]}, sb_{[i]}[n+1 : |sb_{[i]}| - 1]). \end{aligned}$$

which concludes the proof.

From *hinv*2 we can conclude $\neg vR(I)$.

- Case $P(I)$. Let $sb_1 = sb_{[i]}[n : |sb_{[i]}| - 1]$, $l_1 = \mathit{ins}(sb_1) \circ is_{[i]}$ and $l_2 = \mathit{p\text{-}ins}(sb_1)$ then we have from the coupling relation and $\mathit{hinv}5(c_{sbh})$:

$$\begin{aligned} c.\vartheta_{[i]} &= \mathit{del\text{-}t}(\vartheta_{[i]}, sb_1) \\ c.is_{[i]} &= l_1[0 : |l_1| - |l_2| - 1] = I.is_1 \\ c.p_{[i]} &= \mathit{hd\text{-}p}(p_{[i]}, sb_1) \\ c.\mathit{mode}_{[i]} &= \mathit{mode}_{[i]} \\ c.\mathit{mmu}_{[i]} &= \mathit{mmu}_{[i]} \end{aligned}$$

From the intermediate coupling relation we have:

$$\mathit{hd\text{-}p}(p_{[i]}, \mathit{tl}(sb_1)) = I.p_2$$

From the definition of *hd-p* we can conclude

$$c.p_{[i]} = \mathit{hd\text{-}p}(p_{[i]}, sb_1) = I.p_1$$

Observing that the program step does not change the temporaries

$$c.\vartheta_{[i]} = \mathit{del\text{-}t}(\vartheta_{[i]}, sb_1) = \mathit{del\text{-}t}(\vartheta_{[i]}, \mathit{tl}(sb_1))$$

we get from $\mathit{hinv}5(c_{sbh})$:

$$\begin{aligned} &\delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.\mathit{mode}_{[i]}, c.\mathit{mmu}_{[i]}, c.is_{[i]}, I.\mathit{eev}) = (I.p_2, I.is_2) \\ &I.p_2 = \mathit{hd\text{-}p}(p_{[i]}, sb_{[i]}[n+1 : |sb_{[i]}| - 1]) \end{aligned}$$

Hence, we execute the program step from configuration c:

$$c \underset{\text{og,I.eev}}{\overset{\text{p}}{\Longrightarrow}_i} c'.$$

For the program state the coupling is maintained because

$$c'.p_{[i]} = I.p_2 = \mathit{hd\text{-}p}(p_{[i]}, sb_{[i]}[n+1 : |sb_{[i]}| - 1]).$$

From $\mathit{hinv}5(c_{sbh})$ and the coupling relation we can conclude:

$$c'.is_{[i]} = c.is_{[i]} \circ I.is_2.$$

For the instruction sequence we get from the semantics of the abstract machine and from the coupling relation:

$$\begin{aligned}
&c'.is_{[i]} \circ p\text{-}ins(sb_{[i]}[n+1 : |sb_{[i]}| - 1]) \\
&= c.is_{[i]} \circ I.is_2 \circ p\text{-}ins(sb_{[i]}[n+1 : |sb_{[i]}| - 1]) \\
&= c.is_{[i]} \circ p\text{-}ins(sb_{[i]}[n : |sb_{[i]}| - 1]) \\
&= ins(sb_{[i]}[n : |sb_{[i]}| - 1]) \circ is_{[i]} \\
&= ins(sb_{[i]}[n+1 : |sb_{[i]}| - 1]) \circ is_{[i]},
\end{aligned}$$

which concludes the proof for the coupling relation.

□

RMW

Lemma 2.63 (invariants maintained by RMW)

$$\begin{aligned}
&c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh} \wedge \\
&hd(is_{[i]}) = \textbf{RMW} \text{ va t (D, f) r cond p} \rightarrow inv(c'_{sbh})
\end{aligned}$$

Proof Since the store buffer of thread i is empty when we perform the step, invariant *hinv*6 is trivially maintained. Let $(A, L, R, W, A_{pt}, R_{pt}) = og(p, \vartheta'_{[i]})$ then invariants which might get broken by the RMW step are considered below.

- *oinv*1. We proceed the same way as in the proof of *oinv*1 in Lemma 2.38. We have to show for $j \neq i$ for all non-volatile reads $I = sb_{[j]}[k]$ in the suspended part of the SB, that

$$I.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I.pa \notin A \cup A_{pt}. \tag{2.64}$$

  Let c' be the configuration of the abstract machine after we execute the RMW step of thread i:

$$c \overset{\mathrm{m}}{\Longrightarrow}_i c'.$$

  The reasons why this step can be executed with the same ownership transfer and why the result of the RMW test is the same as in the SB machine are given in the proof of lemma 2.94 which does not have $inv(c'_{sbh})$ as hypothesis. After the step we obviously have

$$A \subseteq c'.O_{[i]} \quad \text{and} \quad A_{pt} \subseteq c'.pt_{[i]}.$$

  Let x be the number of instructions up to instruction k in the suspended part of store buffer j:

$$x = k - |exec(sb_{[j]})|.$$

  We execute x steps of thread j in the abstract machine starting from configuration c'. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. From $hinv2(c_{sbh})$ we can get that there is no

volatile read in the suspended portion of $sb_{[j]}$. We refer to the resulting configuration as c'':

$$c' \overset{\substack{\text{p,m } x \\ \text{eev}}}{\Longrightarrow_j} c''.$$

From the proof of Lemma 2.61 we know the ownership transfer in the abstract machine is performed according to the ownership annotations recorded in the corresponding SB instructions. We can also choose the same translated address as the one recorded in the corresponding SB instruction. Since the steps of thread j do not affect the ownership sets of thread i we still have

$$A \subseteq c''.O_{[i]} \quad \text{and} \quad A_{pt} \subseteq c''.pt_{[i]}.$$

Since c'' is a configuration reachable from c it follows

$$safe\text{-}reach_d(c'', og).$$

Hence, instruction I in thread j still has to be safe, which implies

$$I.pa \in c''.O_{[j]} \cup c''.ro.$$

With lemmas 2.53 and 2.52 we conclude (2.64).

- *oinv*2. Let $I = sb_{[j]}[k]$ be a volatile write in store buffer j. To maintain the invariant after the step of thread i we have to show

$$I.pa \notin A \cup A_{pt}.$$

As in the proof of *oinv*1 we execute the step of thread i and steps of thread j up to instruction I and get configurations c' and c'' respectively, where

$$A \subseteq c''.O_{[i]} \quad \text{and} \quad A_{pt} \subseteq c''.pt_{[i]}.$$

Instruction I in thread j still has to be safe, which implies

$$I.pa \notin c''.O_{[i]} \cup c''.pt_{[i]}.$$

and concludes the proof.

- *oinv*3. Let $I = sb_{[j]}[k]$ be a read instruction in the suspended part of store buffer j, such that

$$I.pa \in \delta^k_{sb}(c'_{sbh}, j).ro.$$

Reusing the proof of *oinv*3 from Lemma 2.38 we get

$$I.pa \in \delta^k_{sb}(c_{sbh}, j).ro.$$

To maintain the invariant it is left to show that

$$I.pa \notin A \cup A_{pt}.$$

We have already shown this in the proof for *oinv*1.

- *oinv*4. We need to show:

$$\forall j \neq i.\ A \cap (O_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

From the safety condition for RMW we have:

$$\forall j \neq i.\ A \cap (c.O_{[j]} \cup c.rls_{l[j]} \cup c.rls_{s[j]}) = \emptyset.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$O_{[j]} \cup acq(exec(sb_{[j]})) \subseteq c.O_{[j]} \cup c.rls_{l[j]} \cup c.rls_{s[j]}.$$

Thus, we can conclude:

$$\forall j \neq i.\ A \cap (O_{[j]} \cup acq(exec(sb_{[j]}))) = \emptyset.$$

It is left to show

$$\forall j \neq i.\ A \cap (acq(susp(sb_{[j]}))) = \emptyset.$$

We prove this by contradiction. Assume

$$\exists a \in A.\ \exists j \neq i.\ \exists k \geq |exec(sb_{[j]})|.\ I = sb_{[j]}[k] \wedge vW(I) \wedge a \in I.A.$$

Let c' be configuration of the abstract machine after we execute the RMW step of thread i:

$$c \overset{\mathrm{m}}{\Longrightarrow}_i c'$$

Let x be the number of instructions up to instruction k in the suspended part of store buffer j:

$$x = k - |exec(sb_{[j]})|$$

We execute x steps of thread j in the abstract machine. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. All these instructions can be executed in the abstract machine for the same arguments as in the proof of *oinv*1. We refer to the resulting configuration as c'':

$$c' \underset{\mathrm{eev}}{\overset{\mathrm{p,m}\,x}{\Longrightarrow}_j} c''.$$

All these instructions can be executed in the abstract machine for the same arguments as in the proof of *oinv*1. From the semantics we can get

$$a \in c''.O_{[i]}.$$

Configuration c'' is safe. Hence, the volatile write step of thread j still has to be safe, which implies

$$\forall i \neq j.\ a \notin c''.O_{[i]}$$

and gives a contradiction.

- *sinv*1. The proof is identical to the proof of *sinv*1 from Lemma 2.38 if one uses $safe\text{-}reach_d(c, og)$ instead of $sinv4(c_{sbh})$ to conclude

$$R \subseteq O_{[i]} \quad \text{and} \quad R_{pt} \subseteq pt_{[i]}.$$

- *sinv*2. The proof is identical to the proof of *sinv*2 from Lemma 2.38 if one uses $safe\text{-}reach_d(c, og)$ instead of $sinv4(c_{sbh})$ to conclude the safety properties of the ownership transfer.

- *sinv*3. From the coupling invariant and $safe\text{-}reach_d(c, og)$ we have

$$R \subseteq O_{[i]}.$$

  With $oinv4(c_{sbh})$ we get

$$\forall j \neq i.\, R \cap O_{[j]} = \emptyset,$$

  which implies

$$\forall j.\, O'_{[j]} \cap ro' = \emptyset.$$

  The rest of the proof is identical to the proof of *sinv*3 from Lemma 2.38.

- *sinv*4. Let $I = sb_{[j]}[k]$ be a volatile read or a volatile write in store buffer j. In the proof of *oinv*4 we have already shown that

$$I.A \cap A = \emptyset.$$

  Later in the proof of *pinv*1 we also show

$$I.A \cap A_{pt} = \emptyset.$$

  The rest of the proof is identical to the proof of *sinv*4 from Lemma 2.38 just use $safe\text{-}reach_d(c, og)$ instead of $sinv4(c_{sbh})$.

- *sinv*5. Let $I = sb_{[j]}[k]$ be a write in store buffer j. We can reuse the proof of *sinv*5 from Lemma 2.38 if we show

$$I.pa \notin R.$$

  From safety of the RMW step and from the coupling relation we get $R \subseteq O_{[i]}$. With identical proof of *sinv*5 in Lemma 2.38 we conclude the proof.

- *tinv*2 and *tinv*3. Invariant $hinv3(c_{sbh})$ guarantees that all read temporaries in SBs are present in $\mathrm{dom}(c_{sbh}.\vartheta)$. The invariants are now trivially maintained with $tinv1(c_{sbh})$, $tinv2(c_{sbh})$ and $tinv3(c_{sbh})$.

- *dinv*2. The property is obviously maintained since for all $k \leq |is'_{[i]}|$ it holds

$$dom(c_{sbh}.\vartheta) \cup load_t(is_{[i]}[0:k]) = dom(c'_{sbh}.\vartheta) \cup load_t(is'_{[i]}[0:k-1])$$

- *hinv*1. Let $I = sb_{[j]}[k]$ be a read in the suspended part of store buffer j. We can reuse the proof of *hinv*1 from Lemma 2.38 if we show that addresses of instruction I and of the RMW instruction of thread i are distinct:

  $$I.pa \neq pa.$$

  From $oinv1(c_{sbh})$ we have

  $$I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

  We split cases

  – $I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]}$. Let x be the number of instructions up to instruction k in the suspended part of store buffer j:

    $$x = k - |exec(sb_{[j]})|.$$

    We execute x steps of thread j in the abstract machine starting from configuration c. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. From $hinv2(c_{sbh})$ we can get that there is no volatile read in the suspended portion of $sb_{[j]}$ We refer to the resulting configuration as c'':

    $$c \underset{\text{eev}}{\overset{\text{p,m}\,x}{\Longrightarrow}_j} c''.$$

    Resulting from x applications of Lemma 2.61 we get:

    $$sim(c'', c_{sbh}, j, x)$$

    With the intermediate coupling relation we have:

    $$c''.\mathcal{O}_{[j]} = \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]}.$$

    Configuration c'' is safe. Hence, the RMW step of thread i still has to be safe, which implies

    $$pa \notin c''.\mathcal{O}_{[j]}.$$

  – $I.pa \in \delta^k_{sb}(c_{sbh}, j).ro$. From $oinv3(c_{sbh})$ we know that there are no acquires of $I.pa$ in the executed parts of SBs of other threads. Hence,

    $$I.pa \in \delta^k_{sb}(\Delta^{exec}_{sb[\neq j]}(c_{sbh}), j).ro$$

    If we take $x = k - |exec(sb_{[j]})|$ we can rewrite this as

    $$I.pa \in \delta^x_{sb}(\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq j]}(c_{sbh}), j), j).ro$$

    Applying Lemma 2.50 we get

    $$I.pa \in \delta^x_{sb}(\Delta^{exec}_{sb}(c_{sbh}), j).ro.$$

As in the previous case, we execute x instructions of thread j in the abstract machine starting from configuration c and get configuration c''. From the intermediate coupling relation it follows

$$c''.ro = \delta^{x}_{sb}(\Delta^{exec}_{sb}(c_{sbh}), j).ro.$$

From the safety of the RMW step in configuration c'' we get

$$pa \notin c''.ro,$$

which concludes the proof.

- *pinv*1. To maintain the invariant it is enough to show for all threads $j \neq i$:

$$A_{pt} \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) \subseteq c.pt_{[j]} \cup c.rls_{pt_{[j]}}.$$

Thus, we can conclude from the safety of RMW:

$$A_{pt} \cap (pt_{[j]} \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset.$$

It is left to show

$$A_{pt} \cap (acq_{pt}(susp(sb_{[j]}))) = \emptyset.$$

We prove this by contradiction. Assume

$$\exists a \in A_{pt}.\ \exists k \geq |exec(sb_{[j]})|.\ I = sb_{[j]}[k] \wedge vW(I) \wedge a \in I.A_{pt}.$$

As in the proof of *oinv*4 we execute RMW instruction from configuration c to obtain c' and execute instructions of thread j starting from configuration c' until we executed instruction k. The resulting configuration c'' is safe and we have

$$a \in c''.pt_{[i]}.$$

From the safety of the volatile write step in c'' and the coupling we derive

$$a \notin c''.pt_{[i]}$$

and get a contradiction.

- *pinv*2. To maintain the invariant it is enough to show for all threads $j \neq i$:

$$A \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

The proof of that is completely analogous to the proof of *pinv*1.

- *pinv*3. From the safety of the RMW step, the coupling relation and $pinv4(c_{sbh})$ we have

$$pt_{[i]} \cap R = \emptyset.$$

Hence, we can conclude the proof as we do in the proof of *pinv*3 from Lemma 2.38 by replacing $sinv4(c_{sbh})$ with $safe\text{-}reach_d(c, og)$.

- *pinv*4. The proof trivially follows from the safety of the RMW step.

□

Read and Write

Lemma 2.65 (invariants maintained by vW)

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Write}\ \text{True a (D, f) r cb bw p} \rightarrow inv(c'_{sbh})$$

Proof If the suspended part of SB i is empty, then $c' = c$. Otherwise, to get c' we execute all instructions of thread i from the suspended part of the SB:

$$n = |susp(sb_{[i]})| \quad \text{and} \quad c \overset{\text{p,m}\,n}{\underset{\text{eev}}{\Longrightarrow_i}} c'$$

All these instructions can be executed in the abstract machine, because from the proof of Lemma 2.61 we know the ownership transfer in the abstract machine is performed according to the ownership annotations recorded in the corresponding SB instruction. We can also choose the same translated address as the one recorded in the corresponding SB instruction. Since c' is a configuration reachable from c it follows

$$safe\text{-}reach_d(c', og)$$

From $hinv2(c_{sbh})$ we can get that there is no volatile read in the suspended portion of $sb_{[j]}$. By apply Lemma 2.61 n times we can also get

$$sim(c', c_{sbh}, i, n)$$

which implies

$$c'.is_{[i]} \circ p\text{-}ins([]) = ins([]) \circ is_{[i]}$$
$$c'.\vartheta_{[i]} = del\text{-}t(\vartheta_{[i]}, [])$$

We can conclude

$$hd(c'.is_{[i]}) = hd(is_{[i]}) \wedge c'.\vartheta_{[i]} = \vartheta_{[i]}$$

Since the write operation does not change the temporaries, we can derive that the abstract machine configuration c' must use identical ownership annotations when stepping thread i. We now consider invariants which might get broken by the step:

- $oinv2$. First, we show that this property is maintained for volatile writes of threads $j \neq i$. Let $I = sb_{[j]}[k]$ be a volatile write in store buffer j. We have to show

  $$I.pa \notin A \cup A_{pt}.$$

  The proof of that is identical to the proof of $oinv2$ from Lemma 2.63 if one starts to execute steps of abstract machine from configuration c', where all instructions from the suspended part of the $sb_{[i]}$ are already executed.

Second, we maintain the property for thread i. If $vW(hd(is_{[i]}))$ we also have to show that the property holds for the new volatile write added to the $sb_{[i]}$. For all threads $j \neq i$ it must hold

$$pa \notin \mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}).$$

From the semantics of SB steps and the coupling relation we can conclude:

$$\begin{aligned} pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) &\subseteq c'.pt_{[j]} \cup c'.rls_{pt_{[j]}} \\ \mathcal{O}_{[j]} \cup acq(exec(sb_{[j]})) &\subseteq c'.\mathcal{O}_{[j]} \cup c'.rls_{l_{[j]}} \cup c'.rls_{s[j]}. \end{aligned}$$

From the safety of the volatile write in configuration c' we conclude

$$pa \notin pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) \cup \mathcal{O}_{[j]} \cup acq(exec(sb_{[j]})).$$

It is left to show

$$pa \notin acq_{pt}(susp(sb_{[j]})) \cup acq(susp(sb_{[j]})).$$

We show this by contradiction. Let $I = susp(sb_{[j]})[k]$ be an instruction in the suspended part of store buffer j such that $pa \in I.A \cup I.A_{pt}$. We execute $k+1$ program/memory steps of thread j starting from configuration c'. The choice of the steps to be executed depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c'':

$$c' \overset{\substack{\text{p,m}\, k+1 \\ \text{eev}}}{\Longrightarrow_j} c''.$$

We have

$$I.A \subseteq c''.\mathcal{O}_{[j]} \quad \text{and} \quad I.A_{pt} \subseteq c''.pt_{[j]}.$$

Configuration c'' is safe. Hence, the memory step of thread i still has to be safe, which implies

$$pa \notin c''.\mathcal{O}_{[j]} \cup c''.pt_{[j]}$$

and gives a contradiction.

- *oinv*3. The proof is completely analogous to the proof of *oinv*3 from Lemma 2.63 if one starts executing steps from configuration c', where all instructions from the suspended part of $sb_{[i]}$ are already executed.
- *oinv*4. The proof is completely analogous to the proof of *oinv*4 from Lemma 2.63 if one considers c' as the initial configuration of the abstract machine.
- *sinv*4. The property is trivially maintained for old instructions in SBs. For the newly added instruction we conclude from the safety condition of c':

$$\begin{aligned} &A \subseteq c'.shared \cup c'.\mathcal{O}_{[i]} \cup R_{pt} \setminus A_{pt} \wedge L \subseteq A \wedge \\ &A \cap R = \emptyset \wedge R \subseteq c'.\mathcal{O}_{[i]} \wedge A_{pt} \cap R_{pt} = \emptyset \wedge \\ &A_{pt} \subseteq c'.shared \cup c'.pt_{[i]} \cup R \setminus A \wedge R_{pt} \subseteq c'.pt_{[i]}. \end{aligned}$$

From the intermediate coupling relation we have:

$$\begin{aligned} c'.shared &= \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared \\ c'.O_{[i]} &= \Delta_{sb}(c_{sbh}, i).O_{[i]} \\ c'.pt_{[i]} &= \Delta_{sb}(c_{sbh}, i).pt_{[i]}. \end{aligned}$$

To get the desired property it is left to show that:

$$\forall a \in A_{pt} \cup A.\ a \in c'.shared \rightarrow a \in \Delta_{sb}(c_{sbh}, i).shared. \tag{2.66}$$

From Lemma 2.50 we get

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared.$$

From the definition of Δ_{sb}^{exec} and Δ_{sb} we can conclude:

$$\begin{aligned} &\Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared \subseteq \\ &\Delta_{sb}(c_{sbh}, i).shared \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) \end{aligned}$$

From the intermediate coupling relation we have:

$$\begin{aligned} rels(exec(sb_{[j]})) &\subseteq c'.rls_{s[j]} \cup c'.rls_{l[j]} \\ rels_{pt}(exec(sb_{[j]})) &\subseteq c'.ts_{[j]}.rls_{pt}. \end{aligned}$$

From the safety condition of c' we have:

$$(A \cup A_{pt}) \cap \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.ts_{[j]}.rls_{pt}) = \emptyset,$$

which implies (2.66).

- *inv*5. We only consider thread i. We have to show that:

$$pa \notin \Delta_{sb}(c_{sbh}, i).ro. \tag{2.67}$$

With $safe\text{-}reach_d(c', og)$, we can conclude:

$$pa \notin c'.ro.$$

From the construction of c' and the coupling relation, we can get:

$$c'.ro = \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).ro$$

From Lemma 2.50 we have:

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).ro = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).ro.$$

From the semantics, we have:

$$\Delta^{exec}_{sb}(\Delta_{sb}(c_{sbh}, i)).ro \supseteq$$
$$\Delta_{sb}(c_{sbh}, i).ro \setminus (\bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))).$$

With $oinv2(c_{sbh})$, we can get

$$pa \notin \bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))$$

and concludes (2.67).

- *hinv*4. Since the volatile write does not change the temporaries and the result of $load_t$, *hinv*4 is trivially maintained for the outstanding writes already in the SB. For the newly added write in the SB, we let I be the newly added write instruction in the SB then from the semantics

$$I.f(\vartheta'_{[i]}) = I.v$$

With $dinv2(c_{sbh})$ we have

$$I.D \in dom(\vartheta_{[i]}) \cup load_t(is_{[i]}[0]) = dom(\vartheta_{[i]})$$

From the semantics we can conclude

$$I.D \in dom(\vartheta'_{[i]})$$

which implies $hinv4(c'_{sbh})$.

- *hinv*6. From $hinv6(c_{sbh})$ we have for all $k < |sb_{[i]}|$:

$$\exists is.\ ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

From the semantics of the SB machine we get for all prefixes is:

$$ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) \circ is'_{[i]} = ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]}$$
$$is \circ p\text{-}ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

For $k = |sb_{[i]}|$ we get

$$ins(sb'_{[i]}[k]) \circ is'_{[i]} = is_{[i]}$$
$$p\text{-}ins(sb_{[i]}[k]) = [].$$

Hence, the invariant is maintained.

- *hinv*7. In this case since a volatile write does not change the temporaries when executing, we only consider the newly added volatile write store buffer instruction. From the semantics and the definition of *del-t* we can prove that *hinv*7 is trivially maintained.

- *pinv*1 and *pinv*2. The proof of these invariants is completely analagous to the proof of *pinv*1 and *pinv*2 from Lemma 2.63 if one considers c' as the initial configuration of the abstract machine.

□

Lemma 2.68 (invariants maintained by nvW)

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Write}\ \text{False a (D, f) r cb bw p} \rightarrow inv(c'_{sbh})$$

Proof We first obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 2.65. We now consider invariants which might get broken by the step:

- *oinv*1. Following the proof in Lemma 2.65, we can choose the same translated address *pa* of *a* corresponds to that of the store buffer machine. From the safety of configuration c' we have
$$pa \in c'.\mathcal{O}_{[i]}.$$
From the intermediate coupling relation we have:
$$c'.\mathcal{O}_{[i]} = \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]}.$$
Hence, the desired property for the instruction added to the SB holds.

- *sinv*1. Let *pa* be the translated address of *a*. From the safety of configuration c' we have
$$pa \notin c'.shared.$$
From the intermediate coupling relation we have:
$$c'.shared = \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared$$
From Lemma 2.50 we get
$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared.$$
Our goal is to prove
$$pa \notin \delta_{sb}^{|sb_{[i]}|}(c'_{sbh}, i).shared \tag{2.69}$$
The the execution of non-volatile write does not influence the ownership sets. As a consequence, we have
$$\delta_{sb}^{|sb_{[i]}|}(c'_{sbh}, i).shared = \Delta_{sb}(c_{sbh}, i).shared$$

From the semantics and $sinv4(c_{sbh})$ we have

$$\Delta^{exec}_{sb}(\Delta_{sb}(c_{sbh}, i)).shared \supseteq \Delta_{sb}(c_{sbh}, i) \setminus (\bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]})))$$

With $oinv1(c'_{sbh})$, the semantics and the definition of acq we have

$$pa \in \delta^{|sb_{[i]}|}_{sb}(c'_{sbh}, i).\mathcal{O}_{[i]} = \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \subseteq \mathcal{O}_{[i]} \cup acq(sb_{[i]})$$

From $oinv4(c_{sbh})$ and the definition of acq we can conclude

$$\forall j \neq i.\ pa \notin \mathcal{O}_{[j]} \cup acq(exec(sb_{[j]}))$$

From $pinv2(c_{sbh})$ and the definition of acq_{pt} we can conclude

$$\forall j \neq i.\ pa \notin pt_{[j]} \cup acq_{pt}(exec(sb_{[j]}))$$

After that we can conclude

$$pa \notin \bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))$$

which implies

$$pa \notin \Delta_{sb}(c_{sbh}, i)$$

and gives us (2.69).

- $sinv5$. The proof follows immediately from $sinv1(c'_{sbh})$ and Lemma 2.38.
- $hinv4$ is trivially maintained with $dinv2(c_{sbh})$.
- $hinv6$. The proof is identical to the proof of $hinv6$ in Lemma 2.65.

□

Lemma 2.70 (vR implies no vW in SB)

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Read}\ \text{True a t r ext bw p} \rightarrow susp(sb_{[i]}) = []$$

Proof From the coupling relation for the dirty flag we have

$$(c.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I)) = \mathcal{D}_{[i]}.$$

We prove by contradiction. Assume

$$\exists I \in sb_{[i]}.\ vW(I).$$

We obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 2.65. Since there is a volatile write I in the suspended part of the store buffer we can conclude

$$c'.\mathcal{D}_{[i]}.$$

But this contradicts to the safety of the volatile read in configuration c'. □

Lemma 2.71 (invariants maintained by R)

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Read}\ \text{vol a t r ext bw p} \rightarrow inv(c'_{sbh})$$

Proof We first obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 2.65. Let $|susp(sb_{[i]})| = n$ then we can get

$$sim(c', c_{sbh}, i, n) \wedge safe\text{-}reach_d(c', og)$$

From the intermediate coupling relation we can conclude that the identical translated address pa can be used in both machines. From Lemma 2.70 and the coupling relation we can conclude:

$$vR(hd(is_{[i]})) \rightarrow n = 0 \wedge c' = c \wedge hd(is_{[i]}) = hd(c.is_{[i]}).$$

For volatile read we let $og(p, \vartheta'_{[i]}) = (A, L, R, W, A_{pt}, R_{pt})$. First we need to prove the abstract machine c must use identical ownership annotations when stepping thread i in case of a volatile read. In order to prove that we only need to prove the identity of temporaries after reading in both machines. From the coupling relation we have

$$\begin{aligned} c.\vartheta_{[i]} &= \vartheta_{[i]} \\ c.m &= \Delta^{exec}_{sb}(c_{sbh}).m \\ &= \Delta_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m \end{aligned}$$

By Lemma 2.60 we know

$$\forall j \neq i.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = pa)$$

which implies

$$c.m(pa) = \Delta_{sb}(c_{sbh}, i).m(pa)$$

We let

$$v = fwd(sb_{[i]}, m, pa, bw)$$

then from the semantics

$$v \neq \bot$$

What we need to prove becomes

$$ext(v, bw) = ext(c.m(pa), bw) \tag{2.72}$$

We let $l = maxhit(sb_{[i]}, pa)$ and $I' = sb_{[i]}[l]$ then from the definition of fwd we can get:

$$l = \bot \vee l \neq \bot \wedge bw \leq I'.bw$$

We do a case split on l:

- $l = \bot$. From the defintion of *maxhit* we know there are no store buffer hit for address pa. That means:

$$c.m(pa) = \Delta_{sb}(c_{sbh}, i).m(pa) = m(pa) = v$$

  which implies (2.72)

- $l \neq \bot \wedge bw \leq I'.bw$. From the definition of fwd in this case $v = I'.v$. From the semantics of step buffer step, we can get

$$\exists v'.\ c.m(pa) = \Delta_{sb}(c_{sbh}, i).m(pa) = I'.cb(v, v', I'.bw)$$

  From the property of combination function cb we can conclude

$$I'.cb(v, v', I'.bw) =_{I'.bw} v$$

  From the property of bw we can get

$$I'.cb(v, v', I'.bw) =_{bw} v$$

  Finally from the property of ext we can conclude

$$I.ext(v, bw) = I.ext(I'.cb(v, v', I'.bw), bw)$$

  which also implies (2.72).

As a consequence, we must use identical ownership annotations to step volatile read in both machines. We now consider invariants which might get broken by the step:

- $oinv1$. If $vR(hd(is_{[i]}))$ the invariant is trivially maintained. Otherwise we need to show for the newly added non-volatile read:

$$pa \in \Delta_{sb}(c_{sbh}, i).O_{[i]} \cup \Delta_{sb}(c_{sbh}, i).ro.$$

  From the safety of configuration c' we have

$$pa \in c'.O_{[i]} \cup c'.ro.$$

  From the intermediate coupling relation we have:

$$\begin{aligned} c'.O_{[i]} &= \Delta_{sb}(c_{sbh}, i).O_{[i]} \\ c'.ro &= \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).ro. \end{aligned}$$

  To get the desired property it is left to show that:

$$pa \in c'.ro \rightarrow pa \in \Delta_{sb}(c_{sbh}, i).ro \tag{2.73}$$

  From Lemma 2.50 we get

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).ro = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).ro.$$

From the definition of Δ_{sb}^{exec} and Δ_{sb} we can conclude:

$$\Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).ro \subseteq$$
$$\Delta_{sb}(c_{sbh}, i).ro \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})))$$

From the construction of c' and the coupling relation we have:

$$rels(exec(sb_{[j]})) \subseteq c'.rls_{s[j]} \cup c'.rls_{l[j]}$$
$$rels_{pt}(exec(sb_{[j]})) \subseteq c'.rls_{pt[j]}.$$

From the safety condition of c' we have:

$$pa \notin \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.rls_{pt[j]}).$$

which implies (2.73).

- *oinv*2. The proof is identical to the proof of *oinv*2 for Lemma 2.63.
- *oinv*3. For the case of volatile read there is nothing to show because of Lemma 2.70. For a non-volatile read we need to show for all $j \neq i$:

$$pa \in \Delta_{sb}(c'_{sbh}, i).ro \rightarrow pa \notin (O'_{[j]} \cup acq(sb'_{[j]}) \cup pt'_{[j]} \cup acq_{pt}(sb'_{[j]})).$$

From the construction of c' and the coupling relation we have

$$c'.O_{[j]} = \Delta_{sb}^{exec}(c_{sbh}, j).O_{[j]}$$
$$c'.pt_{[j]} = \Delta_{sb}^{exec}(c_{sbh}, j).pt_{[j]}$$
$$c'.ro = \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).ro.$$

From the safety condition of c' we have:

$$pa \in c'.O_{[i]} \cup c'.ro$$
$$pa \notin \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.ts_{[j]}.rls_{pt}).$$

Applying lemmas 2.53 and 2.52 we get

$$pa \notin c'.O_{[j]} \cup c'.pt_{[j]}.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$O'_{[j]} \cup acq(exec(sb'_{[j]})) \subseteq c'.O_{[j]} \cup c'.rls_{l[j]} \cup c'.rls_{s[j]}$$
$$pt'_{[j]} \cup acq_{pt}(exec(sb'_{[j]})) \subseteq c'.pt_{[j]} \cup c'.rls_{pt[j]}.$$

Hence, all it is left to show

$$pa \notin acq(susp(sb'_{[j]})) \cup acq_{pt}(susp(sb'_{[j]})).$$

We have already done this kind of proof for invariant *oinv*2 in Lemma 2.65. For a volatile read the proof is analogous to the proof of *oinv*3 for Lemma 2.63.

- *oinv*4. For volatile read the proof is completely analogous to the proof of *oinv*4 from Lemma 2.63. For non-volatile read there is nothing to prove.

- *sinv*4. For volatile read the proof is completely analogous to the proof of *sinv*4 from Lemma 2.65. For non-volatile read there is nothing to prove.

- *tinv*2 and *tinv*3. Invariant $hinv3(c_{sbh})$ guarantees that all read temporaries in SBs are present in $dom(\vartheta_{[i]})$. The invariants are now trivially maintained with $tinv1(c_{sbh})$, $tinv2(c_{sbh})$ and $tinv3(c_{sbh})$.

- *dinv*2. The property is obviously maintained since for all $k \leq |is'_{[i]}|$ it holds

$$dom(\vartheta_{[i]}) \cup load_t(is_{[i]}[0:k]) = dom(\vartheta'_{[i]}) \cup load_t(is'_{[i]}[0:k-1])$$

- *hinv*1. For the case of volatile read there is nothing to show because of Lemma 2.70. We only need to consider the newly added non-volatile read. For the newly added non-volatile read instruction I, we have to prove:

$$I.ext(fwd(sb'_{[i]}, m', I.pa, I.bw), I.bw) = I.ext(\delta_{sb}^{|sb_{[i]}|}(c'_{sbh}, i).m(I.pa), I.bw) \tag{2.74}$$

  Since the proof of (2.72) can also be adapted to the non-volatile case, we can derive (2.74) by (2.72).

- *hinv*2. Invariant is easily maintained with Lemma 2.70.

- *hinv*3. For the newly added read the property is trivially maintained. For old reads in the SB the property follows from $tinv3(c_{sbh})$.

- *hinv*4. Invariant is trivially maintained with $tinv3(c_{sbh})$.

- *hinv*6. The proof is identical to the proof of *hinv*6 in Lemma 2.65.

- *hinv*7. Let $sb_1 = sb_{[i]}[k : |sb_{[i]}| - 1]$ and $sb_2 = sb'_{[i]}[k : |sb'_{[i]}| - 1]$ then in this case we need to prove the following equation holds:

$$del\text{-}t(\vartheta_{[i]}, sb_1) = del\text{-}t(\vartheta'_{[i]}, sb_2)$$

  which is trivially proved with the semantics and the definition of *del-t*.

- *pinv*1 and *pinv*2. For volatile read the proof of these invariants is identical to the proof of *pinv*1 and *pinv*2 for Lemma 2.63. For non-volatile read these invariants are trivially maintained.

□

2.4.5 MMU and PF Steps

Lemma 2.75 (invariants maintained by MMU)

$$c_{sbh} \sim c \wedge \text{inv}(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\text{mu}}{\Longrightarrow}_i c'_{sbh} \rightarrow \text{inv}(c'_{sbh})$$

Proof The only invariants which might get broken by MMU steps are $hinv1$ and $minv1$.

- $hinv1(c_{sbh})$. This invariants can only get broken by the MMU write step. Let pa be the address written by the MMU. From $hinv1(c_{sbh})$ we have

$$\forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge nvR(I) \rightarrow$$
$$I.v = I.ext(\delta^k_{sb}(c_{sbh}, j).m(I.pa), I.bw),$$

  where $I = sb_{[j]}[k]$. To maintain the invariant we have to show

$$\forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge nvR(I) \rightarrow I.pa \neq pa.$$

  For case $j \neq i$ the proof is analogous to the proof of hinv1 in Lemma 2.63 if we consider the safety condition of mmu step instead of the safety condition of RMW. For case $j = i$ we prove this lemma by contradiction. We assume:

$$\exists k < |sb_{[i]}|.\ k \geq |exec(sb_{[i]})| \wedge I = sb_{[i]}[k] \wedge nvR(I) \wedge I.pa = pa.$$

  Let x be the number of instructions up to instruction k in the suspended part of store buffer i:

$$x = k - |exec(sb_{[i]})|$$

  We execute x program/memory steps of thread i in the abstract machine starting from configuration c. The choice of the steps to be executed depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c':

$$c \overset{\substack{\text{p,m}\,x \\ \text{eev}}}{\Longrightarrow}_i c'.$$

  Since we do not do any MMU steps in this execution, it holds

$$c'.mmu_{[i]} = c.mmu_{[i]} = mmu_{[i]}.$$

  Hence, we can execute the MMU write to address pa in configuration c' and this write has to be safe, because configuration c' is safe. This implies

$$pa \notin c'.ro \cup c'.O_{[i]}.$$

  At the same time, instruction I, which is at the head of the instruction list of thread i in configuration c', also has to be safe, which implies:

$$pa \in c'.O_{[i]} \cup c'.ro$$

  and gives a contradiction.

- *minv*1. We conclude the proof with the monotonicity property for MMU reads and walk creations.

□

Lemma 2.76 (invariants maintained by page fault)

$$c_{sbh} \sim c \wedge \mathrm{inv}(c_{sbh}) \wedge \mathit{safe\text{-}reach}_d(c, og) \wedge c_{sbh} \overset{\mathrm{pf}}{\Longrightarrow} c'_{sbh} \rightarrow \mathrm{inv}(c'_{sbh})$$

Proof The only invariant which might get broken is

- *hinv*6. Because the store buffer and the instruction sequence are both flushed after the page fault step, the invariant is trivially maintained.

□

2.5 Proving Simulation

In this section we prove simulation between the SB and the virtual machines.

2.5.1 SB Steps

Lemma 2.77 (coupling maintained when R, nvW exits SB)

$$\begin{aligned} &c_{sbh} \sim c \wedge \mathrm{inv}(c_{sbh}) \wedge \mathit{hd}(sb_{[i]}) = I \wedge \neg \mathit{vW}(I) \wedge \\ &c'_{sbh} = \delta_{sb}(c_{sbh}, i) \rightarrow c'_{sbh} \sim c \end{aligned}$$

Proof Since the suspended part of SB i is unchanged, the coupling for the instruction sequence, program state and temporaries is trivially maintained. The coupling for the dirty flag, translation mode and MMU state also can not be broken.

For the other parts of the coupling relation we first observe that

$$\Delta^{exec}_{sb}(c_{sbh}, i) = \Delta^{exec}_{sb}(c'_{sbh}, i).$$

Hence, for $X' \in \{O, pt, rls_l, rls_s, rls_{pt}\}$ we trivially get

$$c.X'_{[i]} = \Delta^{exec}_{sb}(c_{sbh}, i).X'_{[i]} = \Delta^{exec}_{sb}(c'_{sbh}, i).X'_{[i]}.$$

With Lemma 2.38 we get $\mathit{inv}(c'_{sbh})$. For $X \in \{\mathit{shared}, ro, m\}$ we conclude

$$\begin{aligned} c.X &= \Delta^{exec}_{sb}(c_{sbh}).X && \text{(coupling)} \\ &= \Delta^{exec}_{sb[\neq i]}(\Delta^{exec}_{sb}(c_{sbh}, i)).X && \text{(lemma 2.50)} \\ &= \Delta^{exec}_{sb[\neq i]}(\Delta^{exec}_{sb}(c'_{sbh}, i)).X \\ &= \Delta^{exec}_{sb}(c'_{sbh}).X. && \text{(lemma 2.50)} \end{aligned}$$

□

When a volatile write exits SB, the virtual machine executes not only this volatile write, but also all local instructions recorded in the SB after the volatile write. To show that the coupling relation is maintained after all these steps, we define another intermediate coupling relation

$$sim'(c, c_{sbh}, i, k),$$

which has to hold after k local instructions of thread i are executed in the virtual machine. In case $k = 0$, relation $sim'(c, c_{sbh}, i, 0)$ couples the states after the volatile write is committed to the memory in the SB machine and after the volatile write instruction is executed in the virtual machine. After we execute all local steps before the next volatile write and have $k = |exec(c_{sbh}.sb_{[i]})|$ then

$$sim'(c, c_{sbh}, i, k) \equiv c_{sbh} \sim c.$$

Definition 2.78 (Intermediate Coupling Relation 2)

$$\begin{aligned}
&sim'(c, c_{sbh}, i, k) = \\
&\quad k \leq |exec(sb_{[i]})| \wedge \forall X \in \{shared, ro, m\}.\ c.X = \delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X \wedge \\
&\quad \forall j.\ c.mode_{[j]} = mode_{[j]} \wedge c.mmu_{[j]} = mmu_{[j]} \wedge \\
&\qquad (c.\mathcal{D}_{[j]} \vee \exists I \in sb_{[j]}.\ vW(I) \leftrightarrow \mathcal{D}_{[j]}) \ \wedge \forall X \in \{O, pt, rls_l, rls_s, rls_{pt}\}. \\
&\qquad (j \neq i \rightarrow c.X_{[j]} = \Delta^{exec}_{sb}(c_{sbh}, j).X_{[j]} \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(susp(sb_{[j]})) = ins(susp(sb_{[j]})) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, susp(sb_{[j]})) \wedge c.p_{[j]} = hd\text{-}p(p_{[j]}, susp(sb_{[j]}))) \wedge \\
&\qquad (j = i \rightarrow c.X_{[j]} = \delta^k_{sb}(c_{sbh}, j).X_{[j]} \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(sb_{[j]}[k : |sb_{[j]}| - 1]) = ins(sb_{[j]}[k : |sb_{[j]}| - 1]) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, sb_{[j]}[k : |sb_{[j]}| - 1]) \wedge \\
&\qquad\qquad c.p_{[j]} = hd\text{-}p(p_{[j]}, sb_{[j]}[k : |sb_{[j]}| - 1]))
\end{aligned}$$

The following lemmas are similar to the corresponding lemmas for intermediate coupling relation *sim*.

Lemma 2.79 (unowned nvW is in local release set for sim')

$$\begin{aligned}
\forall j, k, pa.\ k < |exec(sb_{[j]})| \wedge i \neq j \wedge sim'(c, c_{sbh}, i, n) \wedge inv(c_{sbh}) \wedge nvW(sb_{[j]}[k]) \wedge \\
pa = sb_{[j]}[k].pa \wedge (pa \notin c.O_{[j]} \vee pa \in c.shared) \rightarrow pa \in c.rls_{l[j]}
\end{aligned}$$

Proof The proof is similar to proof of Lemma 2.56. The only difference is from the intermediate coupling relation 2 we get

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).O_{[j]} \vee pa \in \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared$$

The reset of the proof is identical to that of Lemma 2.56. □

Lemma 2.80 (sim' implies disjoint sets)

$$\forall i.\ sim'(c, c_{sbh}, i, n) \wedge inv(c_{sbh}) \rightarrow disjoint\text{-}osets(c)$$

Proof From the intermediate coupling relation 2 we can get:

$$\forall X \in \{share, ro\}.\ c.X = \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X$$
$$\forall Y \in \{O_{[i]}, pt_{[i]}\}.\ c.Y = \delta^n_{sb}(c_{sbh}, i).Y$$

The rest of the proof is identical to that of Lemma 2.59. □

Lemma 2.81 (no nvW to a read address sim')

$$\begin{aligned}\forall i, pa.\ &(R(hd(c.is_{[i]})) \vee RMW(hd(c.is_{[i]}))) \wedge sim'(c, c_{sbh}, i, n) \wedge safe\text{-}reach_d(c, og) \wedge \\ &pa \in (\text{atran}(c.mmu_{[i]}, hd(c.is_{[i]}).va, c.mode_{[i]}, hd(c.is_{[i]}).r)) \wedge \text{inv}(c_{sbh}) \ \rightarrow \\ &\forall j \neq i.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]})[k] \wedge sb_{[j]}[k].pa = pa)\end{aligned}$$

Proof The proof is similar to that of Lemma 2.60. In the proof instead of applying Lemma 2.56 and Lemma 2.59 we apply Lemma 2.79 and Lemma 2.80. he rest of the proof is identical to that of Lemma 2.60. □

Lemma 2.82 (sim' vW exits sb inductive)

$$\begin{aligned}&sim'(c, c_{sbh}, i, k) \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge \\ &k < |exec(sb_{[i]})| \ \wedge \forall k' < |exec(sb_{[i]})|.\ I = sb_{[i]}[k'] \wedge \neg vR(I) \wedge \\ &(R(I) \rightarrow I.v = I.ext(\delta^{k'}_{sb}(c_{sbh}, i).m(I.pa), I.bw)) \ \rightarrow \\ &\exists c'.\ c \underset{\text{eev}}{\Longrightarrow_i} c' \wedge sim'(c', c_{sbh}, i, k+1)\end{aligned}$$

Proof We let $I = sb_{[i]}[k]$ then execute either a memory or a program step of thread i on the abstract machine depending on I.

- $\neg P(I)$. In this case the proof is similar to the induction step of Lemma 2.61. Instead of apply $hinv1(c_{sbh})$ to get the consistency of the read value we use the precondition. From the definition of sim' we do not need to consider the case when $vW(I)$.
- $P(I)$. In this case we let $sb_1 = sb_{[i]}[k : |sb_{[i]}| - 1]$ then the rest of the proof is identical to the corresponding proof of Lemma 2.61. □

Lemma 2.83 (simulating vW exits sb)

$$\begin{aligned}&c_{sbh} \sim c \wedge \text{inv}(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge I = hd(sb_{[i]}) \wedge vW(I) \wedge \\ &c_{sbh} \overset{\text{sb}}{\Longrightarrow_i} c'_{sbh} \rightarrow \exists c'.\ c \underset{\text{eev}}{\Longrightarrow^*_i} c' \wedge c'_{sbh} \sim c'\end{aligned}$$

Proof From Lemma 2.55 we conclude

$$c.is_{[i]} \neq [].$$

Hence, from the coupling relation we have

$$hd(c.is_{[i]}) = ins(I).$$

From the coupling relation and from $minv1(c_{sbh})$ we conclude

$$I.pa \in \text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

Hence, we can always execute the volatile write from the head of the instruction list and choose the same translated address as we have previously chosen for the corresponding step of the SB machine. From the coupling relation we have

$$c.\vartheta_{[i]} = del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]}))$$

From $hinv7(c_{sbh})$ we have

$$\begin{aligned} I.annot &= og(I.p, del\text{-}t(\vartheta'_{[i]}, susp(sb_{[i]}))) \\ &= og(I.p, c.\vartheta_{[i]}) \end{aligned}$$

Thus, we can transfer the ownership on the abstract machine according to the same ownership annotation we previously recorded in I. Let c'' be configuration of the abstract machine after the step:

$$c \overset{\text{m}}{\Longrightarrow}_i c''.$$

From the coupling relation and the semantics of abstract and SB machines we get for $X \in \{shared, ro, m\}$:

$$\begin{aligned} c''.X &= \delta_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X \\ &= \Delta^{exec}_{sb[\neq i]}(c'_{sbh}).X \quad (Lemma 2.50) \end{aligned}$$

For $X \in \{O, pt, rls_{pt}\}$ we get:

$$c''.X_{[i]} = \delta_{sb}(c_{sbh}, i).X = c'_{sbh}.X_{[i]}$$

For $X \in \{rls_s, rls_l\}$ we have to prove:

$$I.R \cap \Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared = I.R \cap c_{sbh}.shared \tag{2.84}$$

From the $sinv4(c_{sbh})$ we can conclude:

$$\Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared \subseteq c_{sbh}.shared \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})))$$

$$\Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared \supseteq c_{sbh}.shared \setminus (\bigcup_{\forall j \neq i} (acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))))$$

With $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and the definition of acq and acq_{pt}, we can conclude:

$$I.R \cap \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset$$
$$I.R \cap \bigcup_{\forall j \neq i} (acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset$$

which implies (2.84).
For the instruction sequence we conclude with the help of the coupling relation:

$$\begin{aligned} c''.is_{[i]} \circ p\text{-}ins(sb'_{[i]}) &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb_{[i]})) \\ &= tl(ins(susp(sb_{[i]}))) \circ is_{[i]} \\ &= ins(sb'_{[i]}) \circ is'_{[i]}. \end{aligned}$$

For the temporaries and the program state it obviously holds

$$\begin{aligned} c''.\vartheta_{[i]} &= c.\vartheta_{[i]} \\ &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}) \\ &= del\text{-}t(\vartheta'_{[i]}, sb'_{[i]}) \\ c''.p_{[i]} = c.p_{[i]} &= hd\text{-}p(p_{[i]}, sb_{[i]}) \\ &= hd\text{-}p(p'_{[i]}, sb'_{[i]}). \end{aligned}$$

From the coupling relation we get $\mathcal{D}_{[i]}$, which implies $\mathcal{D}'_{[i]}$. From the semantics of the memory step we also get $c'.\mathcal{D}_{[i]}$. This implies

$$(c'.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I)) \leftrightarrow \mathcal{D}'_{[i]}.$$

Hence, we have $sim'(c'', c'_{sbh}, i, 0)$. Let n be the length of the executed part of SB i in configuration c'_{sbh}:

$$n = |exec(sb'_{[i]})|.$$

With $hinv1(c_{sbh})$ we can conclude the consistency of the read value the SB machine. From $hinv2(c_{sbh})$ we know that no volatile read instruction exists in $sb_{[i]}$. Then we apply Lemma 2.82 $n-1$ times and execute steps of thread i accordingly. Finally we get configuration c', such that

$$c'' \underset{\text{eev}}{\Rightarrow}{}^{*}_{i} c' \quad \text{and} \quad sim'(c', c'_{sbh}, i, n).$$

To get the coupling $c'_{sbh} \sim c'$ from $sim'(c', c'_{sbh}, i, n)$ we only have to show

$$\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c'_{sbh}), i) = \Delta^{exec}_{sb}(c'_{sbh}),$$

which we easily get with Lemma 2.38 and Lemma 2.50. □

2.5.2 Program Step

For a program step we make a case distinction on whether there is an outstanding volatile write in the SB. When there is a volatile write in the SB, the abstract machine does not perform any steps. Otherwise, both machines make the same step.

Lemma 2.85 (simulating program step with vW)

$$\forall i.\ c_{sbh} \sim c \wedge (\exists k.\ vW(sb_{[i]}[k])) \wedge c_{sbh} \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i\ c'_{sbh} \rightarrow c'_{sbh} \sim c$$

Proof From the coupling relation and the semantics of the program step we have

$$\begin{aligned} c.p_{[i]} &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]})) \\ &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]}) \circ \mathit{PROG}_{sbh}\ p_{[i]}\ p'_{[i]}\ is_{[i]}\ is') \\ &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb'_{[i]})). \end{aligned}$$

In $\mathit{susp}(sb'_{[i]})$ there is now at least one program instruction. Hence, we have

$$\mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb'_{[i]})) = \mathit{hd\text{-}p}(p'_{[i]}, \mathit{susp}(sb'_{[i]})),$$

and the coupling relation for the program state is maintained. For the coupling relation of the instruction sequence we let $I = \mathit{hd}(is_{[i]})$ and is' be the newly generated instructions by the program step.

$$\begin{aligned} & c.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb'_{[i]})) \\ &= c.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb_{[i]})) \circ is' && (\text{def. } \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i) \\ &= \mathit{ins}(\mathit{susp}(sb_{[i]})) \circ is_{[i]} \circ is' && (\text{coupling}) \\ &= \mathit{ins}(\mathit{susp}(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i) \end{aligned}$$

All the other parts of the coupling relation are trivially maintained. □

Lemma 2.86 (simulating program step without vW)

$$\forall i.\ c_{sbh} \sim c \wedge (\forall k.\ \neg vW(sb_{[i]}[k])) \wedge c_{sbh} \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i\ c'_{sbh} \wedge c \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i\ c' \rightarrow c'_{sbh} \sim c'$$

Proof Observing that

$$\mathit{susp}(sb_{[i]}) = \mathit{susp}(sb'_{[i]}) = []$$

we get from the coupling relation

$$\begin{aligned} c.p_{[i]} &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]})) = p_{[i]} \\ c.\vartheta_{[i]} &= \mathit{del\text{-}t}(\vartheta_{[i]}, \mathit{susp}(sb_{[i]})) = \vartheta_{[i]} \\ c.is_{[i]} &= c.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb_{[i]})) = is_{[i]} \\ c.\mathit{mode}_{[i]} &= \mathit{mode}_{[i]} \\ c.\mathit{mmu}_{[i]} &= \mathit{mmu}_{[i]}. \end{aligned}$$

Hence, the resulting program state and the generated instruction sequence in both machines are the same:

$$\begin{aligned}&\delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev)\\ &= \delta_p(p_{[i]}, \vartheta_{[i]}, mode_{[i]}, mmu_{[i]}, is_{[i]}, eev) = (p', is').\end{aligned}$$

For the coupling of the program state we trivially get

$$p' = c'.p_{[i]} = p'_{[i]} = hd\text{-}p(p'_{[i]}, susp(sb'_{[i]})).$$

Coupling for the instruction sequence we have

$$\begin{aligned}&c'.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]}))\\ &= c.is_{[i]} \circ is' \circ p\text{-}ins(susp(sb_{[i]})) && (\text{def. } \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i)\\ &= c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) \circ is' && (\text{no vW})\\ &= ins(susp(sb_{[i]})) \circ is_{[i]} \circ is' && (\text{coupling})\\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i)\end{aligned}$$

All the other parts of the coupling invariant are trivially maintained. □

2.5.3 MMU and PF Steps

In case of any MMU step the same action is performed in both machines.

Lemma 2.87 (no nvW to page tables)

$$\begin{aligned}&\forall i, a.\ c_{sbh} \sim c \wedge \text{inv}(c_{sbh}) \wedge safe\text{-}mmu\text{-}acc_d(c, a, i) \rightarrow\\ &\quad (\forall j, k.\ k < |exec(sb_{[j]})| \wedge nvW(sb_{[j]}[k]) \rightarrow sb_{[j]}[k].pa \neq a)\end{aligned}$$

Proof We prove this lemma by contradiction. Let $I = sb_{[j]}[k]$ and

$$\exists j.\ \exists k < |exec(sb_{[j]})|.\ nvW(I) \wedge I.pa = a$$

Lemma 2.56 gives us

$$(a \notin c.\mathcal{O}_{[j]} \vee a \in c.shared) \rightarrow a \in c.rls_{l[j]}$$

It implies

$$a \in c.\mathcal{O}_{[j]} \cup c.rls_{l[j]}$$

which contradicts to $safe\text{-}mmu\text{-}acc_d(c, a, i)$. □

As an important consequence of Lemma 2.87 we get the equality of the shared and local page table memory contents in SB and abstract machines in case the coupling relation holds.

Lemma 2.88 (simulating MMU and PF steps)

$$\forall i,\, c_{sbh} \sim c \wedge \mathrm{inv}(c_{sbh}) \wedge \mathit{safe\text{-}reach}_d(c, og) \wedge (c_{sbh} \overset{\text{mu}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{pf}}{\Longrightarrow}_i c'_{sbh}) \rightarrow$$
$$\exists c'.\, (c \overset{\text{mu}}{\Longrightarrow}_i c' \vee c \overset{\text{pf}}{\Longrightarrow}_i c') \wedge c'_{sbh} \sim c'$$

Proof We consider cases depending on the type of the step:

- Walk creation for address va. In this case we have

$$\begin{aligned} mmu'_{[i]} &= \delta_{crtw}(mmu_{[i]}, va) && \text{(semantics of SB machine)} \\ &= \delta_{crtw}(c.mmu_{[i]}, va) && \text{(coupling relation)} \\ &= c'.mmu_{[i]} && \text{(semantics of abs machine)} \end{aligned}$$

- MMU read from address a. In this case we have

$$mmu'_{[i]} = \delta_{mmur}(mmu_{[i]}, a, m(a)) \wedge \mathit{can\text{-}access}(mmu_{[i]}, a).$$

  Let c' be the configuration after c performs a MMU read step. From the semantics of the MMU step we have

$$c'.mmu_{[i]} = \delta_{mmur}(c.mmu_{[i]}, a, c.m(a)).$$

  From the coupling relation we have

$$\begin{aligned} mmu_{[i]} &= c.mmu_{[i]} \\ mode_{[i]} &= c.mode_{[i]} \end{aligned}$$

  Hence, we know that

$$\mathit{can\text{-}access}(c.mmu_{[i]}, a)$$

  holds and from the safety of the abstract machine we get

$$\mathit{safe\text{-}mmu\text{-}acc}_d(c, a, i).$$

  With Lemma 2.87 we conclude $m(a) = c.m(a)$. Hence,

$$mmu'_{[i]} = c'.mmu_{[i]}.$$

- MMU write to address a. We have

$$c'_{sbh}.m(a) = x \wedge x \in \delta_{mmuw}(mmu_{[i]}, a, m(a)) \wedge \mathit{can\text{-}access}(mmu_{[i]}, mode_{[i]}, a).$$

  As in the previous case, we conclude

$$\mathit{safe\text{-}mmu\text{-}acc}_d(c, a, i).$$

From the coupling relation we have

$$mmu_{[i]} = c.mmu_{[i]}$$
$$mode_{[i]} = c.mode_{[i]}$$

With Lemma 2.87 we conclude $m(a) = c.m(a)$. After that we know

$$x \in \delta_{mmuw}(c.mmu_{[i]}, a, c.m(a))$$

Hence, we perform the same step in the abstract machine and get

$$c'.m(a) = x = c'_{sbh}.m(a).$$

With Lemma 2.87 we conclude the proof for the memory coupling.

- Page fault at address pa. As in previous case, we can get

$$can\text{-}access(c.mmu_{[i]}, pa) \ \wedge \ safe\text{-}mmu\text{-}acc_d(c, pa, i)$$

With Lemma 2.87, we can conclude $m(pa) = c.m(pa)$. We have $sb_{[i]} = []$. With the coupling relation, we can conclude:

$$c.is_{[i]} = is_{[i]} \ \wedge \ c.p_{[i]} = p_{[i]}$$

Let $I = hd(is_{[i]}) = hd(c.is_{[i]})$, we have:

$$can\text{-}page\text{-}fault(c.mmu_{[i]}, I.va, I.r, pa, c.m(pa))$$

For mmu state of thread i, we have:

$$\begin{aligned} mmu'_{[i]} &= \delta_{flush}(mmu_{[i]}, \{I.va\}) && \text{(semantics)} \\ &= \delta_{flush}(c.mmu_{[i]}, \{I.va\}) && \text{(coupling)} \\ &= c.mmu'_{[i]} && \text{(semantics)} \end{aligned}$$

For program state of thread i, we have:

$$\begin{aligned} p'_{[i]} &= \delta_{pf}(p_{[i]}, \vartheta_{[i]}, is_{[i]}, eev) && \text{(semantics)} \\ &= \delta_{pf}(c.p_{[i]}, c.\vartheta_{[i]}, c.is_{[i]}, eev) && \text{(coupling)} \\ &= c'.p_{[i]} && \text{(semantics)} \end{aligned}$$

With $sb'_{[i]} = []$, we can conclude:

$$c'.p_{[i]} = hd\text{-}p(p'_{[i]}, susp(sb'_{[i]}))$$

From the semantics, we also have:

$$is'_{[i]} = c'.is_{[i]} = [] \ \wedge \ rls'_{[i]} = c'.rls_{[i]} = \emptyset \ \wedge \ \neg\mathcal{D}'_{[i]} \ \wedge \ \neg c'.\mathcal{D}_{[i]}$$

Coupling relation is trivially maintained.

□

2.5.4 Memory Steps

Lemma 2.89 (coupling for instructions maintained with vW)

$$c_{sbh} \sim c \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge susp(sb'_{[i]}) \neq [] \rightarrow$$
$$c.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) = ins(susp(sb'_{[i]})) \circ is'_{[i]}$$

Proof Let $I = hd(is_{[i]})$ then we conclude:

$$\begin{aligned} c.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) &= c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) && (\text{def. } \overset{\text{m}}{\Longrightarrow}_i) \\ &= ins(susp(sb_{[i]})) \circ is_{[i]} && (\text{coupling}) \\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\text{m}}{\Longrightarrow}_i) \end{aligned}$$

□

Lemma 2.90 (coupling for instructions maintained without vW)

$$c_{sbh} \sim c \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge c \overset{\text{m}}{\Longrightarrow}_i c' \wedge susp(sb'_{[i]}) = [] \rightarrow$$
$$c'.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) = ins(susp(sb'_{[i]})) \circ is'_{[i]}$$

Proof Let $I = hd(is_{[i]})$. We conclude

$$\begin{aligned} & c'.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) \\ &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb'_{[i]})) && (\text{def. } \overset{\text{m}}{\Longrightarrow}_i) \\ &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb_{[i]})) \\ &= tl(c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]}))) && (\text{def. } tl) \\ &= tl(ins(susp(sb_{[i]})) \circ is_{[i]}) && (\text{coupling}) \\ &= tl(is_{[i]}) && (\text{no vW}) \\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\text{m}}{\Longrightarrow}_i) \end{aligned}$$

□

FENCE, INVLPG, SWITCH and WritePTO

Lemma 2.91 (simulating FENCE, INVLPG, SWITCH, WPTO)

$$I = hd(is_{[i]}) \wedge (FENCE(I) \vee INVLPG(I) \vee SWITCH(I) \vee WPTO(I)) \wedge$$
$$c_{sbh} \sim c \wedge c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \wedge c \overset{\text{m}}{\Longrightarrow}_i c' \rightarrow c'_{sbh} \sim c'$$

Proof In order for a step to be scheduled, the SB has to be empty:

$$sb'_{[i]} = sb_{[i]} = [].$$

Since the instruction lists are equal in both machines, we know that $hd(c.is_{[i]}) = I$. The coupling for the instruction list is maintained with Lemma 2.90. For the dirty flag and for the release sets rls_X, where $X \in \{l, s, pt\}$ we have

$$c'.\mathcal{D}_{[i]} = \mathcal{D}'_{[i]} = False$$
$$c'.rls_{X[i]} = rls'_{X[i]} = \emptyset.$$

Since the store buffer of thread i in configuration c'_{sbh} is empty, the coupling for the dirty flag and for the release sets obviously holds.

In case $I =$ **INVLPG** F we get for the MMU coupling:

$$c'.mmu_{[i]} = \delta_{flush}(c.mmu_{[i]}, F) = \delta_{flush}(mmu_{[i]}, F) = mmu'_{[i]}.$$

In case $I =$ **WritePTO** v we also get

$$c'.mmu_{[i]} = mmu'_{[i]}$$

with the same argument as in the INVLPG case.

In case $I =$ **Switch** mode for the mode bit coupling we get

$$mode'_{[i]} = c'.mode'_{[i]} = mode$$

All the other parts of the coupling relation can not be possibly broken by a step. □

RMW

Lemma 2.92 (ownership transfer safe after SB step)

$$inv(c_{sbh}) \wedge safe\text{-}otran(c_{sbh}, i, I) \wedge i \neq j \rightarrow safe\text{-}otran(\delta_{sb}(c_{sbh}, j), i, I)$$

Proof Let $c'_{sbh} = \delta_{sb}(c_{sbh}, j)$. From the semantics of the SB step we have

$$pt'_{[j]} \cup acq_{pt}(sb'_{[j]}) \subseteq pt_{[j]} \cup acq_{pt}(sb_{[j]})$$
$$O'_{[j]} \cup acq(sb'_{[j]}) \subseteq O_{[j]} \cup acq(sb_{[j]}),$$

and conclude the proof. □

Lemma 2.93 (ownership transfer, Δ_{sb} commute)

$$inv(c_{sbh}) \wedge safe\text{-}otran(c_{sbh}, i, I) \wedge sb_{[i]} = [] \rightarrow$$
$$\Delta^{exec}_{sb}(otran\text{-}sbh(c_{sbh}, i, I)) = otran\text{-}sbh(\Delta^{exec}_{sb}(c_{sbh}), i, I)$$

Proof We apply Lemma 2.44 as many times as necessary to reorder all executed SB steps of all threads behind the ownership transfer. After every SB step we use lemmas 2.38 and 2.92 to make sure that invariants are maintained after the step and the ownership transfer of instruction I in thread i is still safe. □

Both machines perform the same step when $RMW(hd(is_{[i]}))$.

Lemma 2.94 (simulating RMW)

$$c_{sbh} \sim c \wedge \mathrm{inv}(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh} \wedge c \overset{\mathrm{m}}{\Longrightarrow}_i c' \wedge$$
$$hd(is_{[i]}) = \textbf{RMW}\ \mathrm{va\ t\ (D, f)\ cond\ r\ p} \rightarrow c'_{sbh} \sim c'$$

Proof The coupling for the instruction list, for the dirty flag and for the release sets is maintained with the same arguments as in case of the fence memory step. Since we know that $sb_{[i]}$ is empty, we also get

$$c.is_{[i]} = is_{[i]}$$
$$c.\vartheta_{[i]} = \vartheta_{[i]}.$$

Let $I = hd(is_{[i]})$ and $I' = hd(c.is_{[i]})$ then we have

$$I = I'$$

Invariant $tinv3(c_{sbh})$ guarantees that temporary t is fresh. Hence,

$$c.\vartheta_{[i]}(t) = \vartheta_{[i]}(t) = \bot.$$

From the coupling relation we also have

$$c.mmu_{[i]} = mmu_{[i]}$$
$$c.mode_{[i]} = mode_{[i]}$$

Therefore we can choose identical physical addresses for address translation. Let

$$pa \in (\mathrm{atran}(mmu_{[i]}, va, mode_{[i]}, r))$$

Applying Lemma 2.60 we know that there are no writes to pa in the executed parts of SBs:

$$\forall j.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = pa). \tag{2.95}$$

This implies

$$c.m(pa) = m(pa).$$

Hence, we read the same value and have the same physical address into temporary t on both machines and the coupling for temporaries is maintained.

$$c.\vartheta'_{[i]} = \vartheta'_{[i]}$$

Thus, we have

$$og(I.p, \vartheta'_{[i]}) = og(I'.p, c'.\vartheta_{[i]})$$

which means we can perform the identical ownership transfers in both machines. Moreover, we can conclude:

$$\mathrm{cond}(c.\vartheta_{[i]}(t \mapsto (c.m(pa), pa))) = \mathrm{cond}(\vartheta_{[i]}(t \mapsto (m(pa), pa))).$$

From the coupling relation, the safety of the RMW instruction and invariants $sinv4(c_{sbh})$ $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can conclude

$$\begin{aligned}
&I.L \subseteq I.A \wedge I.R \subseteq \mathcal{O}_{[i]} \ \wedge I.R_{pt} \subseteq pt_{[i]} \wedge \\
&(\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset) \wedge \\
&(\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset).
\end{aligned}$$

Hence, we get

$$safe\text{-}otran(c_{sbh}, i, hd(is_{[i]})).$$

Lemma 2.93 now guarantees that the coupling for the local components of all threads. For the shared and read only sets we need to prove

$$\forall X \in \{shared, ro\}.\ c'.X = \Delta_{sb}^{exec}(c'_{sbh}).X$$

Since the SB steps do not change the temporaries, we have:

$$\begin{aligned}
\Delta_{sb}^{exec}(c'_{sbh}).X &= \Delta_{sb}^{exec}(otran\text{-}sbh(c_{sbh}, i, I)).X && \\
&= otran\text{-}sbh(\Delta_{sb}^{exec}(c_{sbh}), i, I).X && \text{(Lemma 2.93)} \\
&= otran\text{-}sbh(c, i, I).X && \text{(coupling relation)} \\
&= c'.X && \text{(def. of } otran\text{-}sbh\text{)}
\end{aligned}$$

Thus, the coupling is also maintained for shared and read only sets. In case the RMW test fails and no write is performed the memory coupling can not be possibly broken. Otherwise, we have to show that the coupling for memory is maintained. The coupling for memory cells other than *pa* is obviously maintained. For *pa* we get

$$\begin{aligned}
c'.m(pa) &= f(c.\vartheta_{[i]}(t \mapsto (c.m(pa), pa))) \\
&= f(\vartheta_{[i]}(t \mapsto (m(pa), pa))) \\
&= c'_{sbh}.m(pa)
\end{aligned}$$

The store buffer of thread *i* is still empty after the step. (2.95) guarantees that no other SBs have a write to *pa*. Hence, the memory coupling for *pa* is maintained. □

Read and Write

Lemma 2.96 (simulating R,W with vW)

$$\begin{aligned}
&c_{sbh} \sim c \wedge I = hd(is_{[i]}) \wedge (R(I) \vee W(I)) \wedge susp(sb'_{[i]}) \neq [] \wedge \\
&c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \rightarrow c'_{sbh} \sim c.
\end{aligned}$$

Proof The coupling for the instruction list is maintained with Lemma 2.89. If we execute $vW(I)$, then the dirty flag is set and we get

$$\exists k.\ vW(sb'_{[i]}[k]) \leftrightarrow (\mathcal{D}'_{[i]} = True)$$

otherwise the dirty flag is unchanged. The coupling for the dirty flag is maintained. For the other parts of the coupling invariant we consider cases:

- $susp(sb_{[i]}) \neq []$. If $R(I)$ then we from $tinv3(c_{sbh})$ we know that $I.t$ is fresh:

  $$\vartheta_{[i]}(t) = \bot.$$

  Hence, we conclude from the coupling relation and from the semantics of a memory step:

  $$\begin{aligned} c.\vartheta_{[i]} &= del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]})) \\ &= del\text{-}t(\vartheta'_{[i]}, susp(sb_{[i]}) \circ I) \\ &= del\text{-}t(\vartheta'_{[i]}, susp(sb'_{[i]})). \end{aligned}$$

  All the other parts of the coupling relation are trivially maintained because

  $$exec(sb_{[i]}) = exec(sb'_{[i]}).$$

- $susp(sb_{[i]}) = []$. This implies $vW(I)$. Since the volatile write is always added to the suspended part of the SB (even if it was empty before) we have

  $$exec(sb_{[i]}) = exec(sb'_{[i]}).$$

  and all parts of the coupling relation are trivially maintained.

□

A memory step is deterministic for a given translated address pa. We introduce a function δ_m to compute the next state of the SB machine after a memory step of thread i. Let $I = hd(is_{[i]})$ then

$$\delta_m(c_{sbh}, i, pa) \equiv c'_{sbh}$$

In order to guarantee the parameter pa is the physical address used for execution we have following constraints:

$$\begin{aligned} &c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh} \wedge pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \wedge \\ &(RMW(I) \rightarrow \vartheta'_{[i]}(I.t) = m(pa)) \wedge (W(I) \vee R(I) \rightarrow last(sb'_{[i]}).pa = pa) \end{aligned}$$

where:

$$last(l) = l[|l| - 1]$$

With this hypothesis we can prove the following lemma. In the following lemma we let $I = hd(is_{[i]})$, $c'_{sbh} = \Delta^{exec}_{sb}(c_{sbh}, i)$, $c''_{sbh} = \Delta^{exec}_{sb}(c_{sbh})$ and $c'''_{sbh} = \Delta^{exec}_{sb[\neq i]}(c_{sbh})$.

Lemma 2.97 (*vR* result consistent)

$$\begin{aligned} &c_{sbh} \sim c \wedge safe\text{-}reach_d(c, og) \wedge inv(c_{sbh}) \wedge susp(sb_{[i]}) = [] \wedge vR(I) \wedge \\ &pa = \epsilon(atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \wedge v = fwd(sb_{[i]}, m, pa, I.bw) \wedge v \neq \bot \rightarrow \\ &\delta_m(c_{sbh}, i, pa).\vartheta_{[i]} = \delta_m(c'_{sbh}, i, pa).\vartheta_{[i]} = \delta_m(c''_{sbh}, i, pa).\vartheta_{[i]} = \delta_m(c'''_{sbh}, i, pa).\vartheta_{[i]} \end{aligned}$$

Proof From the semantics, we can conclude that the SB steps does not affect the mmu state, address translation mode and the temporaries. Thus, we can use the same pa as the physical address to perform the memory step. We can get

$$\vartheta_{[i]} = \vartheta'_{[i]} = \vartheta''_{[i]} = \vartheta'''_{[i]}$$

Let

$$\begin{aligned} v' &= fwd(sb'_{[i]}, m', pa, I.bw) \\ v'' &= fwd(sb''_{[i]}, m'', pa, I.bw) \\ v''' &= fwd(sb'''_{[i]}, m''', pa, I.bw) \end{aligned}$$

then we need to prove:

$$I.ext(v, I.bw) = I.ext(v', I.bw) = I.ext(v'', I.bw) = I.ext(v''', I.bw)$$

which can be concluded from:

$$\exists R \in \{=, =_{bw}\}, \forall x_1, x_2 \in \{v, v', v'', v'''\}.\ \ x_1\ R\ x_2 \tag{2.98}$$

Applying Lemma 2.60 we can conclude:

$$\forall\, j \neq i.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = pa) \tag{2.99}$$

As a consequence, in configuration c''_{sbh}, c'''_{sbh} the modifications on address pa is issued only from $sb_{[i]}$. Let $l = maxhit(sb_{[i]}, pa)$ and $I' = sb_{[i]}[l]$ then we do a case split on l:

- $l = \bot$. From the definition of fwd we know that pa will not be updated by SB steps of thread i. That implies

$$m(pa) = v = v' = v'' = v''$$

- $l \neq \bot \wedge I.bw \leq I'.bw$. We have

$$\begin{aligned} v''' = v &= I'.v && \text{(def. of } fwd) \\ v'' &= m''(pa) && (susp(sb_{[i]}) = [] \text{ and def. of } fwd) \\ &= m'(pa) && (2.99) \\ &= \Delta_{sb}(c_{sbh}, i).m(pa) && (susp(sb_{[i]}) = [] \text{ and def. of } \Delta) \\ &= I'.cb(I'.v, \delta^l_{sb}(c_{sbh}).m(pa), I'.bw) && \text{(def. of } \Delta) \\ &=_{I'.bw} I'.v \end{aligned}$$

  With the property of $=_{bw}$ we can conclude (2.98).

□

Lemma 2.100 (Δ^{exec}_{sb}, δ_m step commute)

$$
\begin{aligned}
&c_{sbh} \sim c \wedge safe\text{-}reach_d(c, og) \wedge inv(c_{sbh}) \wedge I = hd(is_{[i]}) \wedge (vR(I) \vee nvW(I)) \wedge \\
&pa = \epsilon(atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \wedge susp(sb_{[i]}) = [] \wedge inv(c_{sbh}) \rightarrow \\
&\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa), i) = \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}, i), i, pa), i) \wedge \\
&\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa)) = \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa), i)
\end{aligned}
$$

Proof From the definition of Δ^{exec}_{sb} and $\Delta^{exec}_{sb[\neq i]}$ we can get

$$X \in \{mmu, mode, \vartheta\}.\ X_{[i]} = \Delta^{exec}_{sb[\neq i]}(c_{sbh}).X_{[i]} = \Delta^{exec}_{sb}(c_{sbh}, i).X_{[i]} = \Delta^{exec}_{sb}(c_{sbh}).X_{[i]}$$

Thus, we can use the same pa for memory step of c_{sbh}, $\Delta^{exec}_{sb}(c_{sbh}, i)$ and $\Delta^{exec}_{sb}(c_{sbh})$ in thread i. If $vR(I)$ then by applying Lemma 2.97 we can get the identity of temporaries and the same ownership annotation is recorded as history information for

$$
\begin{aligned}
&\delta_m(c_{sbh}, i, pa),\\
&\delta_m(\Delta^{exec}_{sb}(c_{sbh}, i), i, pa),\\
&\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa),\\
&\delta_m(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i, pa).
\end{aligned}
$$

Adding a volatile read instruction or a non-volatile write to the $sb_{[i]}$ does not affect other instructions in $sb_{[i]}$ and does not change the local state of the thread except the temporaries and the length of $sb_{[i]}$. Hence, we can first execute old instructions in the $sb_{[i]}$, then execute a memory step adding the instruction to the empty SB, and finally perform an SB step executing newly added instruction. This concludes the first statement of the lemma.

Since the invariants are maintained by memory steps, for the second statement we can apply Lemma 2.50 and conclude

$$\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa)) = \Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(\delta_m(c_{sbh}, i, pa)), i).$$

Putting an instruction to the store buffer i only affects the component $ts_{[i]}$. Hence, we can reorder it with the store buffer steps of other threads:

$$\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(\delta_m(c_{sbh}, i, pa)), i) = \Delta^{exec}_{sb}(\delta_m(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i, pa), i).$$

We already proved that the SB steps maintain the invariants and the coupling relation. Then applying the first statement of the lemma we get

$$\Delta^{exec}_{sb}(\delta_m(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i, pa), i) = \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i), i, pa), i).$$

By applying once again Lemma 2.50 we conclude the second statement of the lemma. □

Lemma 2.101 (simulating R,W without vW)

$$
\begin{aligned}
&c_{sbh} \sim c \wedge \mathrm{inv}(c_{sbh}) \wedge safe\text{-}reach_d(c, og) \wedge I = hd(is_{[i]}) \wedge (W(I) \vee R(I)) \wedge \\
&susp(sb'_{[i]}) = [] \wedge c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh} \rightarrow \exists c'.\ c \overset{\mathrm{m}}{\Longrightarrow}_i c' \wedge c'_{sbh} \sim c'
\end{aligned}
$$

Proof Let $I = hd(is_{[i]})$. Since the suspended part of $sb_{[i]}$ is empty we have

$$hd(c.is_{[i]}) = I.$$

If I is a read or a write instruction, then from the coupling relation we have

$$\text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) = \text{atran}(mmu_{[i]}, I.va, mode_{[i]}, I.r).$$

Hence, we can always execute the instruction from the head of the instruction list of the abstract machine and choose the same translated address as we do in the step of the SB machine . Let c' be configuration of the abstract machine after this step:

$$c \overset{\text{m}}{\Longrightarrow}_i c'.$$

The coupling for the instruction list is maintained with Lemma 2.90. We now do a case split on the type of the step and consider other parts of the coupling relation which might get broken by this step:

- $R(I)$. Let pa be the translated address chosen in both the SB and the abstract machine. Lemma 2.60 guarantees that there are no writes to pa in the executed parts of store buffers other than i. With the proof of equation (2.72) in Lemma 2.71 and the coupling for temporaries we can conclude

$$\begin{aligned} c'.\vartheta_{[i]} &= c.\vartheta_{[i]}(t \mapsto (I.ext(c.m(pa), I.bw), I.pa)) \\ &= \vartheta_{[i]}(t \mapsto (I.ext(fwd(sb_{[i]}, m, pa, I.bw), I.bw), I.pa)) \\ &= \vartheta'_{[i]}. \end{aligned}$$

  Thus, we can conclude for $vR(I)$ the ownership transfer is performed in abstract machine according to the ownership annotation recorded in the newly added instruction in $sb'_{[i]}$. We let $og(I.p, \vartheta'_{[i]}) = (A, L, R, W, A_{pt}, R_{pt})$. For $vR(I)$ the coupling relation for ghost components might get broken. Coupling for the ownership sets of threads $j \neq i$ is trivially maintained. Let $X \in \{O, pt, rls_{pt}\}$. From the coupling invariant and the semantics of the memory step of the abstract and the SB machine we get

$$\begin{aligned} c'.shared &= c.shared \cup (R \cup R_{pt}) \setminus (L \cup A_{pt}) \\ &= \Delta^{exec}_{sb}(c_{sbh}).shared \cup (R \cup R_{pt}) \setminus (L \cup A_{pt}) && \text{(coupling relation)} \\ &= \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa), i).shared && \text{(semantics)} \\ c'.ro &= c.ro \cup (R \setminus W) \setminus (A \cup A_{pt}) \\ &= \Delta^{exec}_{sb}(c_{sbh}).ro \cup (R \setminus W) \setminus (A \cup A_{pt}) && \text{(coupling relation)} \\ &= \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa), i).ro && \text{(semantics)} \\ c'.X_{[i]} &= \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}, i), i, pa), i).X_{[i]} && \text{(semantics)} \end{aligned}$$

  With Lemma 2.65 we get $inv(c'_{sbh})$. Applying Lemma 2.100 we get

$$\begin{aligned} \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa), i) &= \Delta^{exec}_{sb}(c'_{sbh}) \\ \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}, i), i, pa), i) &= \Delta^{exec}_{sb}(c'_{sbh}, i) \end{aligned}$$

which concludes the coupling for shared, read-only and thread local ownership sets. For release local and release shared set, we have to prove

$$R \cap c.shared = R \cap c_{sbh}.shared \tag{2.102}$$

From the coupling invariants, we have:

$$c.shared = \Delta_{sb}^{exec}(c_{sbh}).shared$$

With the semantics, we can conclude:

$$\begin{aligned}
&\Delta_{sb}^{exec}(c_{sbh}).shared \subseteq \\
&c_{sbh}.shared \bigcup_{\forall j}(rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))), \\
&\Delta_{sb}^{exec}(c_{sbh}).shared \supseteq \\
&c_{sbh}.shared \setminus (\bigcup_{\forall j}(acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]})))).
\end{aligned}$$

With $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and the semantics we can conclude:

$$\begin{aligned}
I.R \cap c.shared \subseteq R \cap c_{sbh}.shared, \\
I.R \cap c.shared \supseteq R \cap c_{sbh}.shared.
\end{aligned}$$

which concludes (2.102).

- $nvW(I)$. Let $I =$ **Write** False a (D, f) r g bw p and pa be the translated address chosen in both the SB and the abstract machine. From the coupling invariant and the semantics of the memory step of the abstract and the SB machine we get

$$c'.m = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i).m.$$

With Lemma 2.68 we get $inv(c'_{sbh})$. Applying Lemma 2.100 we conclude

$$\delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i) = \Delta_{sb}^{exec}(c'_{sbh}).$$

□

2.6 Proving Safety of the Delayed Release

So far we have used safety of the delayed release of the virtual machine to prove SB reduction theorem. Now we have to show that if all possible executions of the virtual machine satisfy the regular safety, then they also satisfy safety of the delayed release.

thread i	thread j
vR pa $(A, L, \{pa\}, W, A_{pt}, R_{pt})$	—
$\mathbf{MMU_W}$ pa′	—
—	**vW** pa $(\{pa\}, L', R', W', A'_{pt}, R'_{pt})$

(a) Violation of delayed release safety.

thread i	thread j
$\mathbf{MMU_W}$ pa′	—
—	**vW** pa $(\{pa\}, L', R', W', A'_{pt}, R'_{pt})$

(b) Violation of regular safety.

Figure 2.8: Ruling out safety of the delayed release violation: Example 1.

2.6.1 Intuition

If a trace satisfies regular safety, but does not satisfy safety of the delayed release, then the safety violation is due to a clash with release sets of some thread i. This clash happen between (i) an instruction in the head of the instruction list of thread $j \neq i$, which can access or acquire an address from release sets of thread i, or (ii) an MMU of thread $j \neq i$ which can access an address from release sets of thread i, or (iii) an MMU of thread i which can access an address from its own local release set. We do a proof by contradiction: we show that if some execution does not satisfy safety of the delayed release, then there exists another execution, which does not satisfy regular safety. We can obtain such an execution by "undoing" steps of thread i until we reach a point when the conflicting address was released. The steps of thread i which are removed can only be program steps or memory steps executing reads and non-volatile writes (since all the other instructions are clearing the release sets). After removing these steps of thread i we continue our execution until we get the safety violation. Since in the new execution thread i has not put the conflicting address to the release thread yet, it will be either in the owns or in the PT set of thread i. Thus, we will get violation of the regular safety. We also might end up in a situation when we encounter violation of the regular safety earlier in the new execution. This is also fine, since we assume all traces to satisfy regular safety. Note, that we are not removing the MMU steps of thread i, because MMUs are allowed to write the shared memory and can possibly affect the execution flow of other threads. Below we consider a few examples.

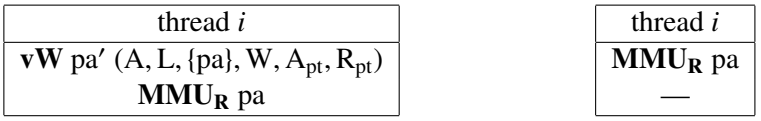
Example 1 Let address pa be in the ownership set of thread i:

$$pa \in c.O_{[i]}.$$

Let thread i execute a volatile read which releases the address pa and an MMU write. After that thread j performs a volatile write acquiring address pa (Fig. 2.8a).

This behaviour satisfies regular safety, because at the time when thread j acquires pa it is not present in the ownership set of thread i. Yet, it is present in the release set of thread i, which means that safety of the delayed release is violated. To rule out this situation we consider another trace, where we "undo" the read operation of thread i (Fig. 2.8b).

The read operation of thread i do not affect execution of thread j (i.e. thread j can execute before the read operation of thread i). Moreover, since we allow only volatile writes to page



(a) Violation of delayed release safety.

(b) Violation of regular safety.

Figure 2.9: Ruling out safety of the delayed release violation: Example 2.

tables, the read operation also cannot affect the MMU steps of thread i. Hence, we can simply postpone execution of the read, execute the MMU write immediately and then perform the step of thread j. In this case address pa is present in the ownership set of thread i, attempt to acquire it by thread j violates the regular safety of the virtual machine.

Example 2 Let address pa again be in the ownership set of thread i:

$$pa \in c.O_{[i]}.$$

Let thread i execute a volatile write and release of address pa. After that MMU of thread i attempts to perform a read from pa. (Fig. 2.9a).

This behaviour again satisfies regular safety, because at the time when MMU performs a read the address pa is shared and is not owned by any thread. Yet, it is present in the release set of thread i, which means that safety of the delayed release is not satisfied. To rule out this situation we "undo" the last memory step of thread i (Fig. 2.9b) and get a trace which violates regular safety.

2.6.2 "Undoing" a Step

We define a simulation relation, which is supposed to hold between the states of the original execution and the states of the execution where a step of thread i has not been performed:

$$\begin{aligned} simd(c,d,i) \equiv \forall\, j \neq i.\ & c.ts_{[j]} = d.ts_{[j]} \wedge \\ & c.mmu_{[i]} = d.mmu_{[i]} \wedge c.mode_{[i]} = d.mode_{[i]} \wedge \\ & c.rls_{l[i]} \subseteq d.rls_{l[i]} \cup (d.O_{[i]} \setminus d.shared) \wedge \\ & c.rls_{s[i]} \subseteq d.rls_{s[i]} \cup d.O_{[i]} \wedge c.rls_{pt[i]} \subseteq d.rls_{pt[i]} \cup d.pt_{[i]} \wedge \\ & \forall a.\ a \notin d.O_{[i]} \vee a \in d.shared.\ c.m(a) = d.m(a). \end{aligned}$$

Lemma 2.103 ensures that we can "undo" a step of thread i. This means that relation $simd$ holds between the states before and after a step of thread i, if this step is a program step or a memory step executing a read or a non-volatile write.

Lemma 2.103 (undoing a step)

$$\begin{aligned} & (c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \vee c \overset{\text{m}}{\Longrightarrow}_i c' \wedge I = hd(c.is_{[i]}) \wedge (nvW(I) \vee R(I))) \wedge \\ & safe\text{-}state(c, og) \rightarrow simd(c', c, i) \end{aligned}$$

Proof Since the step of thread i can not affect the local configuration of threads $j \neq i$ we obviously get

$$c'.ts_{[j]} = c.ts_{[j]}.$$

For $vR(I)$ we let $og(I.p, c'.\vartheta_{[i]}) = (A, L, R, W, A_{pt}, R_{pt})$. If a step of thread i is a memory step performing ownership transfer we have

$$\begin{aligned} c'.rls_{l[i]} &= c.rls_{l[i]} \cup (R \setminus c.shared) \\ c'.rls_{s[i]} &= c.rls_{s[i]} \cup (R \cap c.shared) \\ c'.rls_{pt[i]} &= c.rls_{pt[i]} \cup R_{pt}. \end{aligned}$$

From $safe\text{-}state(c, og)$ we have

$$R \subseteq c.O_{[i]} \quad \text{and} \quad R_{pt} \subseteq c.pt_{[i]}.$$

Hence, we get

$$\begin{aligned} c'.rls_{l[i]} &\subseteq c.rls_{l[i]} \cup (c.O_{[i]} \setminus c.shared) \wedge \\ c'.rls_{s[i]} &\subseteq c.rls_{s[i]} \cup c.O_{[i]} \wedge \\ c'.rls_{pt[i]} &\subseteq c.rls_{pt[i]} \cup c.pt_{[i]}. \end{aligned}$$

If a step of thread i is a memory step executing a non-volatile write instruction I, then from the safety of configuration c we have for all physical addresses pa:

$$pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \rightarrow pa \in c.O_{[i]} \setminus c.shared.$$

This implies

$$\forall a.\ a \notin c.O_{[i]} \vee a \in c.shared.\ c'.m(a) = c.m(a)$$

and concludes $simd(c', c, i)$. □

The following lemma guarantees that if $simd(c, d, i)$ holds and we perform a step from configuration c which is neither a program , memory step nor a page fault step of thread i, then we can also perform a step from configuration d, such that the simulation relation is maintained after the step.

Lemma 2.104 (simd maintained)

$$\begin{aligned} &simd(c, d, i) \wedge disjoint\text{-}osets(d) \wedge safe\text{-}state(d, og) \wedge \\ &(c \overset{\text{mu}}{\Longrightarrow}_i c' \vee c \underset{\text{eev}}{\Longrightarrow}_j c' \wedge j \neq i) \rightarrow \exists d'.\ d \underset{\text{eev}}{\Longrightarrow} d' \wedge simd(c', d', i) \end{aligned}$$

Proof We split cases on the kind of a step from c to c'.

- A step from c to c' is a memory step of thread $j \neq i$. From $simd(c, d, i)$ we get

$$c.ts_{[j]} = d.ts_{[j]}.$$

Hence, we can execute the (same) first instruction I of thread j with the same address translation and the same ownership transfer in both machines. Let d' be the configuration after we execute this instruction from configuration d:

$$d \overset{\mathrm{m}}{\Longrightarrow}_j d'.$$

If this instruction is doing a read (either $R(I)$ or $RMW(I)$) from address pa then from $safe\text{-}state(d, og)$ and $disjoint\text{-}osets(d)$ we conclude:

$$pa \notin d.\mathcal{O}_{[i]} \vee pa \in d.shared.$$

Hence, $simd(c, d, i)$ guarantees that we are reading the same value in both machines. This implies

$$c'.ts_{[j]} = d'.ts_{[j]}.$$

If instruction I is performing ownership transfer or write to memory ($W(I)$ or $vR(I)$ or $RMW(I)$) then we let $(A, L, R, W, A_{pt}, R_{pt})$ be the ownership annotations. From $safe\text{-}state(d, og)$ we get

$$R \subseteq d.\mathcal{O}_{[j]} \wedge R_{pt} \subseteq d.pt_{[j]}.$$

With $disjoint\text{-}osets(d)$ we have

$$R \cap d.\mathcal{O}_{[i]} = R_{pt} \cap d.\mathcal{O}_{[i]} = \emptyset.$$

Configuration of thread i is not changed during a step. Hence,

$$\begin{aligned} d'.\mathcal{O}_{[i]} \setminus d'.shared &= d.\mathcal{O}_{[i]} \setminus d'.shared \\ &= d.\mathcal{O}_{[i]} \setminus (d.shared \cup (R \cup R_{pt}) \setminus (L \cup A_{pt})) \\ &\supseteq d.\mathcal{O}_{[i]} \setminus (d.shared \cup (R \cup R_{pt})) \\ &= d.\mathcal{O}_{[i]} \setminus d.shared \end{aligned}$$

which together with $simd(c, d, i)$ implies

$$c'.rls_{I[i]} \subseteq d'.rls_{I[i]} \cup (d'.\mathcal{O}_{[i]} \setminus d'.shared).$$

If instruction I is writing the memory, then we are writing the same value for both configurations. Moreover, we observe that

$$\forall a.\ a \notin d.\mathcal{O}_{[i]} \vee a \in d.shared \leftrightarrow a \notin d'.\mathcal{O}_{[i]} \vee a \in d'.shared$$

which concludes $simd(c', d', i)$. For other cases the lemma is trivially maintained.

- A step from c to c' is a program step of thread $j \neq i$. From $simd(c, d, i)$ we get

$$c.ts_{[j]} = d.ts_{[j]},$$

which means that we can perform the same program step for both configurations and get $simd(c', d', i)$.

- A step from c to c' is an MMU step of thread i accessing address pa. Thus, we have

$$can\text{-}access(c.mmu_{[i]}, pa).$$

From $simd(c, d, i)$ we know that

$$c.mmu_{[i]} = d.mmu_{[i]} \quad \text{and} \quad c.mode_{[i]} = d.mode_{[i]}.$$

Hence, we have

$$can\text{-}access(d.mmu_{[i]}, pa).$$

Safety of configuration d ensures

$$pa \notin d.\mathcal{O}_{[i]}.$$

Hence, from $simd(c, d, i)$ we get

$$c.m(pa) = d.m(pa).$$

This implies that we can perform the same kind of MMU step from both configurations resulting with the same MMU configuration:

$$c'.mmu_{[i]} = d'.mmu_{[i]}.$$

If MMU step is writing the memory, then we write the same value for both configurations and get

$$c'.m(pa) = d'.m(pa),$$

which concludes $simd(c', d', i)$.

- A step from c to c' is an MMU step of thread $j \neq i$ to address pa. From $simd(c, d, i)$ we get

$$c.ts_{[j]} = d.ts_{[j]}.$$

We easily conclude $simd(c', d', i)$ the same way as in case of the MMU step of thread i.

- A step from c to c' is an page fault step of thread $j \neq i$. Thus, we have

$$can\text{-}access(c.mmu_{[j]}, pa) \wedge can\text{-}page\text{-}fault(c.mmu_{[j]}, I.va, I.r, pa, c.m(pa))$$

As in the case of the MMU step of thread i we get

$$c.m(pa) = d.m(pa) \quad \text{and} \quad can\text{-}access(d.mmu_{[j]}, pa)$$

Therefore we can also have

$$\mathit{can\text{-}page\text{-}fault}(d.mmu_{[j]}, I.va, I.r, pa, d.m(pa))$$

This implies that we can perform the identical page fault step from both configurations resulting with the same program state and MMU configuration:

$$c'.p_{[j]} = d'.p_{[j]} \wedge c'.mmu_{[j]} = d'.mmu_{[j]}$$

which concludes the proof.

□

The following lemma guarantees that we can continue execution of the machine after we "undo" a step of thread i.

Lemma 2.105 (simd computation)

$$\begin{aligned}
&n > 0 \wedge c^0 \underset{\text{eev}}{\Longrightarrow}^{n} c^n \wedge \forall k < n.\neg(c^k \overset{\text{p,m}}{\underset{\text{eev}}{\Longrightarrow}}_i c^{k+1} \vee c^k \overset{\text{pf}}{\Longrightarrow}_i c^{k+1}) \wedge \\
&\mathit{safe\text{-}reach}(d^0, og) \wedge \mathit{disjoint\text{-}osets}(d^0) \wedge \mathit{simd}(c^0, d^0, i) \rightarrow \\
&\exists d^n.\ d^0 \underset{\text{eev}}{\Longrightarrow}^{n} d^n \wedge \mathit{simd}(c^n, d^n, i)
\end{aligned}$$

Proof We will prove an inductive statement, which trivially implies the postcondition of the lemma:

$$\forall l \leq n.\ \exists d^l.\ \mathit{simd}(c^l, d^l, i) \wedge (l > 0 \rightarrow d^0 \underset{\text{eev}}{\Longrightarrow}^{l} d^l).$$

Proof by induction on l. For induction base $l = 0$ we obviously take $n = 0$ and have from the preconditions:

$$\mathit{simd}(c^0, d^0, i).$$

For the induction step $l \rightarrow l + 1$ we have from the induction hypothesis

$$\exists d^l.\ \mathit{simd}(c^l, d^l, i) \wedge (l > 0 \rightarrow d^0 \underset{\text{eev}}{\Longrightarrow}^{l} d^l).$$

Step l in the original computation is either an MMU step of thread i or is an arbitrary step of thread $j \neq i$. With Lemma 2.52 we get

$$\mathit{disjoint\text{-}osets}(d^l).$$

Hence, we can apply Lemma 2.104 to find configuration d^{l+1}, where

$$d^l \underset{\text{eev}}{\Longrightarrow} d^{l+1} \quad \text{and} \quad \mathit{simd}(c^{l+1}, d^{l+1}, i).$$

□

2.6.3 Reconstructing Safety Violation

The following predicate denotes that in configuration c there exists a safety violation due to a clash with release sets of thread j. Let

$$\vartheta' = \begin{cases} c.\vartheta_{[i]}(I.t \mapsto c.m(pa)) & R(I) \vee RMW(I) \\ c.\vartheta_{[i]} & otherwise \end{cases}$$
$$I = hd(c.is_{[i]})$$
$$og(\vartheta'_{[i]}, I.p) = (A, L, R, W, A_{pt}, R_{pt})$$

then

$$\begin{aligned}
&unsafe\text{-}release(c, j, og) \equiv \\
&\quad (\exists i \neq j.\ \exists pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}). \\
&\qquad (vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \wedge pa \in c.rls_{l[j]} \cup c.rls_{pt[j]}) \vee \\
&\qquad (nvR(I) \vee W(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \wedge pa \in c.rls_{[j]}) \vee \\
&\qquad (vW(I) \vee vR(I) \vee RMW(I) \wedge (A \cup A_{pt}) \cap c.rls_{[j]} \neq \emptyset)) \vee \\
&\quad (\exists pa, i.\ can\text{-}access(c.mmu_{[i]}, pa) \wedge \\
&\qquad (pa \in c.rls_{[j]} \wedge i \neq j \vee pa \in c.rls_{l[i]})).
\end{aligned}$$

Lemma 2.106 ensures that we can reconstruct a safety violation after we "undo" a step of thread i and execute all the remaining steps in such a way, that relation *simd* holds between the faulty state of the original computation and the end state of the new computation.

Lemma 2.106 (reconstructing safety violation)

$$simd(c, d, i) \wedge unsafe\text{-}release(c, i, og) \wedge disjoint\text{-}osets(d) \rightarrow \neg safe\text{-}state_d(d)$$

Proof From $simd(c, d, i)$ we know that

$$\begin{aligned}
\forall m \neq i.\ c.ts_{[m]} &= d.ts_{[m]} \qquad (2.107)\\
c.mmu_{[i]} &= d.mmu_{[i]} \\
c.mode_{[i]} &= d.mode_{[i]} \\
c.rls_{l[i]} &\subseteq d.rls_{l[i]} \cup (d.O_{[i]} \setminus d.shared) \\
c.rls_{s[i]} &\subseteq d.rls_{s[i]} \cup d.O_{[i]} \\
c.rls_{pt[i]} &\subseteq d.rls_{pt[i]} \cup d.pt_{[i]}.
\end{aligned}$$

We split cases:

- MMU safety violation for address pa in thread $m \neq i$:

  $$can\text{-}access(c.mmu_{[m]}, pa) \wedge pa \in c.rls_{[i]}$$

  From (2.107) we get

  $$can\text{-}access(d.mmu_{[m]}, pa) \wedge pa \in d.rls_{[i]} \cup d.O_{[i]} \cup d.pt_{[i]}.$$

If $pa \in d.rls_{[i]} \cup d.\mathcal{O}_{[i]}$, then configuration d does not satisfy safety of the delayed release and we are done. If $pa \in d.pt_{[i]}$, then $disjoint\text{-}osets(d)$ ensures

$$pa \notin d.pt_{[m]} \cup d.shared,$$

which also gives safety violation and concludes the proof.

- MMU safety violation for address pa in thread i:

  $$can\text{-}access(c.mmu_{[i]}, pa) \wedge pa \in c.rls_{l[i]}.$$

  As in the previous case we use (2.107) to get

  $$can\text{-}access(d.mmu_{[i]}, pa) \wedge pa \in d.rls_{l[i]} \cup d.\mathcal{O}_{[i]},$$

  which violates safety and concludes the proof.

- Instruction safety violation in thread $m \neq i$. Let $I = hd(c.is_{[m]})$ be the faulty instruction. We prove this by contradiction. Assuming $safe\text{-}state_d(d)$ safety violation can be caused either by a physical address of the instruction being present in the release sets of thread i or by a clash between the acquire sets of instruction I and the release sets of thread i. For the first class of violations let

  $$pa \in atran(c.mmu_{[m]}, I.va, c.mode_{[m]}, I.r)$$

  be the faulty address. From (2.107) we immediately get

  $$pa \in atran(d.mmu_{[m]}, I.va, d.mode_{[m]}, I.r).$$

  If instruction I is a read or an RMW instruction, then from $safe\text{-}state_d(d)$ and $disjoint\text{-}osets(d)$ we conclude:

  $$pa \notin d.\mathcal{O}_{[i]} \vee pa \in d.shared.$$

  Hence, $simd(c, d, i)$ guarantees that the result of a read in configurations c and d is the same:

  $$c.m(pa) = d.m(pa).$$

  We consider sub-cases, where $\vartheta' = c.\vartheta_{[m]}(I.t \mapsto c.m(pa))$ and identical ownership annotations must be used in c and d.

  - $(vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \wedge pa \in c.rls_{l[i]} \cup c.rls_{pt[i]})$. From (2.107) we get

    $$pa \in d.rls_{l[i]} \cup (d.\mathcal{O}_{[i]} \setminus d.shared) \cup d.rls_{pt[i]} \cup d.pt_{[i]}.$$

    If pa is present in one of the release sets:

    $$pa \in d.rls_{l[i]} \cup d.rls_{pt[i]},$$

    then configuration d is unsafe and we are done. If

    $$pa \in (d.\mathcal{O}_{[i]} \setminus d.shared) \cup d.pt_{[i]}$$

then with $disjoint\text{-}osets(d)$ we get

$$pa \notin d.\mathcal{O}_{[m]} \cup d.shared \cup d.pt_{[m]}$$

and conclude the proof.

– $(nvR(I) \vee nvW(I)) \wedge pa \in c.rls_{[i]}$. From (2.107) we get

$$pa \in d.rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]}.$$

With $disjoint\text{-}osets(d)$ we get

$$pa \in d.rls_{[i]} \vee pa \notin d.\mathcal{O}_{[m]} \cup d.ro \cup d.pt_{[m]}$$

and conclude the proof.

– $(vW(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \wedge pa \in c.rls_{[i]})$. From (2.107) we get

$$pa \in d.rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]},$$

which already gives us safety violation.

– $vW(I) \vee vR(I) \vee RMW(I) \wedge (A \cup A_{pt}) \cap c.rls_{[i]} \neq \emptyset$. From (2.107) we get

$$(A \cup A_{pt}) \cap (d.rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]}) \neq \emptyset.$$

which also gives us safety violation and concludes the proof.

□

2.6.4 Simulation Theorem

In the intuitive explanations which we gave in the beginning of this section we are constructing a new trace by undoing all steps of the conflicting thread until we reach a point when the conflicting address is being released. Nevertheless, we do here a simpler proof. We state an induction hypothesis that all traces up to length n satisfy safety of the delayed release. On the induction step we prove by contradiction and assume there exists a trace of length $n + 1$ which does not satisfy safety of the delayed release. We undo only a single step of the conflicting thread i.e., the last memory or program step. We then continue the execution and show that the shorter trace will also violate safety of the delayed release (we assume the regular safety to be always satisfied by all possible traces). Since existence of such a trace contradicts to our induction hypothesis, we conclude that all traces of length $n + 1$ satisfy safety of the delayed release.

Lemma 2.108 (safety ind)

$$initial(c) \wedge safe\text{-}reach(c, og) \rightarrow \forall k \leq n.\ safe\text{-}reach_d(c, k, og)$$

Proof By induction on n. For case $n = 0$ the statement trivially holds since all release sets are empty. For the induction step $n \to n + 1$ we do a proof by contradiction. Assume

$$safe\text{-}reach(c, og) \wedge \neg(\forall k \leq n + 1.\ safe\text{-}reach_d(c, k, og)).$$

Our induction hypothesis guarantees that all configurations up to step n are safe:

$$\forall k \leq n.\ safe\text{-}reach_d(c, k, og).$$

Hence, there must exist a trace with $n + 1$ steps starting from configuration c, where the first n steps are safe and the last step is not safe. We denote the states in this computation by $c^0, \ldots, c^n, c^{n+1}$, where $c^0 = c$, $c^i \underset{\text{eev}}{\Rightarrow} c^{i+1}$ and

$$\neg safe\text{-}state_d(c^{n+1}, og) \wedge \forall k \leq n.\ safe\text{-}state_d(c^k, og).$$

From the precondition of the function we know that state c^{n+1} satisfies regular safety of the virtual machine:

$$safe\text{-}state(c^{n+1}, og).$$

Hence, the safety violation is due to a clash with release sets of some thread i:

$$unsafe\text{-}release(c^{n+1}, i, og).$$

In this case we aim at "undoing" the last program or memory step of thread i and arguing that a (shorter) trace without this steps would still be unsafe, which contradicts to our induction hypothesis. Note, that after the last program or memory step of thread i there can be no page fault steps of thread i, since this step would empty the release sets.

Let k be the state before the last program or memory step of thread i in the computation:

$$c^k \overset{\text{p,m}}{\underset{\text{eev}}{\Longrightarrow_i}} c^{k+1} \wedge \forall m \in [k + 1 : n - 1].\ \neg(c^m \overset{\text{p,m}}{\underset{\text{eev}}{\Longrightarrow_i}} c^{m+1} \vee c^m \overset{\text{pf}}{\Longrightarrow_i} c^{m+1}).$$

If this step is a memory step, then it can execute a read or a non-volatile write, since all other instructions empty the release sets. Hence, we can apply Lemma 2.103 to get

$$simd(c^{k+1}, c^k, i).$$

From $initial(c^0)$ we have

$$disjoint\text{-}osets(c^0).$$

With Lemma 2.52 we get

$$disjoint\text{-}osets(c^k).$$

From $safe\text{-}reach(c^0)$ we have $safe\text{-}reach(c^k)$. We now split cases:

- if $k = n$ then we are removing the last step in our execution sequence and we have

  $$simd(c^{n+1}, c^n, i).$$

  Hence, we apply Lemma 2.106 to reconstruct the safety violation in configuration c^n and get

  $$\neg safe\text{-}state_d(c^n),$$

  which contradict to our induction hypothesis.

- if $k < n$ then we apply Lemma 2.105 (we instantiate $d^0 = c^k$, $c^0 = c^{k+1}$, $n = (n-k)$) and get

$$\exists d^{n-k}.\ d^0 \underset{\text{eev}}{\overset{\text{n-k}}{\Longrightarrow}} d^{n-k} \wedge simd(c^{n+1}, d^{n-k}, i)$$

  Since $k + (n-k) < n + 1$, the constructed sequence is shorter than the original one. With Lemma 2.52 we get $disjoint\text{-}osets(d^{n-k})$. Hence, we apply Lemma 2.106 to reconstruct the safety violation in configuration d^{n-k} and get

$$\neg safe\text{-}state_d(d^{n-k}, og),$$

  which contradicts to our induction hypothesis.

□

The proof of Theorem 2.34 now simply follows from Lemma 2.108.

3 Instantiation of Store Buffer Machine Model

In order to apply our SB reduction theorem with MMU at the ISA level, we will instantiate our abstract machine model in this chapter and prove the simulation between the instantiated machine and an ISA machine named MIPS-86 [Sch13] in next chapter. MIPS-86 is a MIPS processor core extended with x86-64 like architecture features (in particular memory system).

In the first section of this chapter, we will introduce the MIPS-86 ISA. In the second section, we will instantiate the machine models in Chapter 2. During the instantiation, we will discharge all assumptions and constraints.

In our model in Chapter 2, the page table entry *pte* and the memory value *v* have identical type $\mathbb{V}$. From the specification of MIPS-86 [Sch13] we have $pte \in \mathbb{B}^{32}$. Thus, we instantiate the type $\mathbb{V}$ with $\mathbb{B}^{32}$. In order to adapt to the memory in Chapter 2, which is a map $\mathbb{A} \to \mathbb{V}$, we change the byte-addressable memory in [Sch13] to a word addressable memory with byte write signals.

3.1 MIPS ISA

A MIPS-86 machine configuration consists of multiple processors and a shared sequential consistent memory. Each processor has three components: an SB, a TLB, and a processor core. In the processor core there are a general purpose register file (*gpr*), a special purpose register file (*spr*) and a program counter (*pc*). In the remaining portion of this section, we will introduce the formal specifications of all these components. We copy this section from [Sch13] with the following modifications: (i) We use a word addressable memory with byte write signals instead of a byte-addressable memory. (ii) We introduce extra SB flushes, when

* an interrupt happens;

* executing an instruction which is an read modify write instruction, TLB flush instruction, return from exception instruction or an instruction which moves data from the *gpr* to *mode* or *pto* in the *spr*.

We introduce the extra SB flushes to make our SB reduction theorem from Chapter 2 applicable to the MIPS-86 ISA. (iii) In order to keep the instantiated model small, we do not consider caches, devices, and inter-processor interrupts.

Table 3.1: MIPS-86 Special Purpose Registers.

i	synonym	
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 (contains effective address on pfls)
5	pto	page table origin
6	mode	mode register $\in \{0^{31}1, 0^{32}\}$
7	emode	exception mode register (saves mode in case of interrupt)

3.1.1 Processor Core

Definition 3.1 (Processor Core Configuration of MIPS-86) A MIPS-86 processor core configuration $c = (c.pc, c.gpr, c.spr) \in K_{core}$ consists of

- a program counter: $c.pc \in \mathbb{B}^{30}$,
- a general purpose register file: $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$, and
- a special purpose register file: $c.spr : \mathbb{B}^5 \to \mathbb{B}^{32}$. The available special purpose registers of MIPS-86 are listed in table 3.1.

In MIPS-86 ISA, there are three types of instructions: *I*-type instructions, *J*-type instructions and *R*-type instructions. *I*-type instructions are instructions that operate with two registers and a so-called *immediate constant*, *J*-type instructions are absolute jumps and *R*-type instructions rely on three register operands.

The instruction-layout of MIPS-86 depends on the type of instruction. In the subsequent definition of the MIPS-86 instruction layout, *rs*, *rt* and *rd* specify registers of the MIPS-86 machine.

***I*-type instruction layout**

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

***R*-type instruction layout**

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

***J*-type instruction layout**

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

Instruction Layout Overview

A quick overview of available instructions is given in tables 3.2 (for I-type), 3.3 (for J-type) and 3.4 (for R-type). Note that these tables – while giving a general idea what is available and what it approximately does – are not comprehensive. In particular, note that for all instructions whose mnemonic ends with "u", register values are interpreted as binary numbers whereas in all other cases they are interpreted as two's-complement numbers.

Auxiliary Definitions for Instruction Execution

In what follows, we make auxiliary definitions in order to define the processor core transitions that deal with instruction execution. In order to execute an instruction, the processor core needs to read values from the memory. Of relevance to instruction execution is the instruction word $I \in \mathbb{B}^{32}$ and, if the instruction I is a *read* or *rmw* instruction, we need the value $R \in \mathbb{B}^{32}$ read from memory.

Instruction Decoding Formalizing the tables given in subsection 3.1.1, we define the following shorthands for the fields of the MIPS-86 instruction layout:

- instruction opcode
$$opc(I) = I[31 : 26]$$
- instruction type
$$rtype(I) \equiv opc(I) = 0^6 \vee opc(I) = 010^4$$
$$jtype(I) \equiv opc(I) = 0^4 10 \vee opc(I) = 0^4 11$$
$$itype(I) \equiv \neg(rtype(I) \vee jtype(I))$$
- register addresses
$$rs(I) = I[25 : 21]$$
$$rt(I) = I[20 : 16]$$
$$rd(I) = I[15 : 11]$$
- shift amount
$$sa(I) = I[10 : 6]$$
- function code (used only for R-type instructions)
$$fun(I) = I[5 : 0]$$

Table 3.2: *I*-Type Instructions of MIPS-86.

opcode	Mnemonic	Assembler-Syntax	d	Effect
Data Transf(I)er				
100 000	lb	lb *rt rs imm*	1	rt = sxt(m)
100 001	lh	lh *rt rs imm*	2	rt = sxt(m)
100 011	lw	lw *rt rs imm*	4	rt = m
100 100	lbu	lbu *rt rs imm*	1	rt = 0^{24}m
100 101	lhu	lhu *rt rs imm*	2	rt = 0^{16}m
101 000	sb	sb *rt rs imm*	1	m = rt[7:0]
101 001	sh	sh *rt rs imm*	2	m = rt[15:0]
101 011	sw	sw *rt rs imm*	4	m = 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}

opcode	rt	Mnemonic	Assembler-Syntax	Effect
Branch				
000 001	00000	bltz	bltz *rs imm*	pc = pc + (rs < 0 ? imm00 : 4)
000 001	00001	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	00000	blez	blez *rs imm*	pc = pc + (rs $\leq$ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz *rs imm*	pc = pc + (rs > 0 ? imm00 : 4)

Here, $m = m_d(ea(c, I))$.

Table 3.3: *J*-Type Instructions of MIPS-86

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 3.4: *R*-Type Instruction of MIPS-86.

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, System Call				
000000	001 000	jr	jr *rs*	pc = rs
000000	001 001	jalr	jalr *rd rs*	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call
Synchronizing Memory Operations				
000000	111 111	rmw	rmw *rd rs rt*	rd' = m m' = (rd = m ? rt : m)
000000	111 110	mfence	mfence	
TLB Instructions				
000000	111 101	flush	flush	flushes TLB
000000	111 100	invlpg	invlpg *rd*	flushes TLB translations for addr. *rd*

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]

- immediate constants (for I-type and J-type instructions, respectively)

$$imm(I) = I[15:0]$$

$$iindex(I) = I[25:0]$$

For every MIPS-Instruction, we define a predicate on the MIPS-configuration which is true iff the corresponding instruction is to be executed next. The name of such an instruction-decode predicate is always the instruction's mnemonic (see MIPS ISA-tables at the beginning). Formally, the predicates check for the corresponding opcode and function code. E.g.

$$lw(I) \equiv opc(I) = 100011$$

$$\ldots$$

$$add(I) \equiv rtype(I) \wedge fun(I) = 100000$$

The instruction-decode predicates are so trivial to formalize that we do not explicitly list all of them here. Let

$$ill(I) = \neg(lw(I) \vee \ldots \vee add(I))$$

be the predicate that formalizes that the opcode of instruction I is illegal by negating the disjunction of all instruction-decode predicates. Note that, encountering an illegal opcode during instruction execution, an illegal instruction interrupt will be triggered.

Arithmetic and Logic Operations The arithmetic logic unit (ALU) of MIPS-86 behaves according to the following table:

alucon[3:0]	i	alures	ovf
0 000	*	$a +_{32} b$	0
0 001	*	$a +_{32} b$	$[a] + [b] \notin T_{32}$
0 010	*	$a -_{32} b$	0
0 011	*	$a -_{32} b$	$[a] - [b] \notin T_{32}$
0 100	*	$a \wedge_{32} b$	0
0 101	*	$a \vee_{32} b$	0
0 110	*	$a \oplus_{32} b$	0
0 111	0	$\neg_{32}(a \vee_{32} b)$	0
0 111	1	$b[15:0]0^{16}$	0
1 010	*	$0^{31}([a] < [b]?1:0)$	0
1 011	*	$0^{31}(\langle a\rangle < \langle b\rangle?1:0)$	0

Based on inputs $a, b \in \mathbb{B}^{32}$, $alucon \in \mathbb{B}^4$ and $i \in \mathbb{B}$, this table defines $alures(a, b, alucon, i) \in \mathbb{B}^{32}$ and $ovf(a, b, alucon, i) \in \mathbb{B}$. To describe whether a given instruction $I \in \mathbb{B}^{32}$ performs an arithmetic or logic operation, we define the following predicates:

- I-type ALU instruction: $compi(I) \equiv itype(I) \wedge I[31:29] = 001$
- R-type ALU instruction: $compr(I) \equiv rtype(I) \wedge I[5:4] = 10$

- any ALU instruction: $alu(I) \equiv compi(I) \vee compr(I)$

Following the instruction set architecture tables, we formalize the right and left operand of an ALU instruction $I \in \mathbb{B}^{32}$ based on a given processor core configuration $c \in K_{\mathbf{core}}$ as follows:

- left ALU operand: $lop(c, I) = c.gpr(rs(I))$
- right ALU operand: $rop(c, I) = \begin{cases} c.gpr(rt(I)) & rtype(I) \\ sxt_{32}(imm(I)) & /rtype(I) \wedge /I[28] \\ zxt_{32}(imm(I)) & otherwise \end{cases}$

We define the ALU control bits of an instruction $I \in \mathbb{B}^{32}$ as

$$alucon(I)[2:0] = \begin{cases} I[2:0] & rtype(I) \\ I[28:26] & otherwise \end{cases}$$

$$alucon(I)[3] \equiv rtype(I) \wedge I[3] \vee /I[28] \wedge I[27]$$

The ALU result of an instruction I executed in processor core configuration $c \in K_{\mathbf{core}}$ is then given by

$$compres(c, I) = alures(lop(c, I), rop(c, I), alucon(I), itype(I))$$

Jump and Branch Instructions Jump and branch instructions affect the program counter of the machine. The difference between branch instructions and jump instructions is that branch instructions perform conditional jumps based on some condition expressed over general purpose register values. The following table defines the branch condition result $bcres(a, b, bcon) \in \mathbb{B}$, i.e. whether for the given parameters the branch will be performed or not, based on inputs $a, b \in \mathbb{B}^{32}$ and $bcon \in \mathbb{B}^4$:

bcon[3:0]	bcres(a, b, bcon)
001 0	$[a] < 0$
001 1	$[a] \geq 0$
100 *	$a = b$
101 *	$a \neq b$
110 *	$[a] \leq 0$
111 *	$[a] > 0$

We define the following branch instruction predicates that denote whether a given instruction $I \in \mathbb{B}^{32}$ is a jump or successful branch instruction given configuration $c \in K_{\mathbf{core}}$:

- branch instruction: $b(I) \equiv opc(I)[5:3] = 0^3 \wedge itype(I)$
- jump instruction: $jump(I) \equiv j(I) \vee jal(I) \vee jr(I) \vee jalr(I)$
- jump or branch taken:

$$jbtaken(c, I) \equiv jump(I) \vee b(I) \wedge bcres(c.gpr(rs(I)), c.gpr(rt(I)), opc[2:0]rt(I)[0])$$

We define the target address of a jump or successful branch instruction $I \in \mathbb{B}^{32}$ in a given configuration $c \in K_{\mathbf{core}}$ as

$$btarget(c, I) \equiv \begin{cases} c.pc +_{32} sxt_{30}(imm(I))00 & b(I) \\ c.gpr(rs(I)) & jr(I) \vee jalr(I) \\ (c.pc +_{32} 4_{32})[31:28]iindex(I)00 & j(I) \vee jal(I) \end{cases}$$

Shift Operations Shift instructions perform shift operations on general purpose registers. For $a[n-1:0] \in \mathbb{B}^n$ and $i \in \{0, \ldots, n-1\}$ we define the following shift results ($\in \mathbb{B}^n$):

- shift left logical: $sll(a, i) = a[n-i-1:i]0^i$
- shift right logical: $srl(a, i) = 0^i a[n-1:i]$
- shift right arithmetic: $sra(a, i) = a_{n-1}^i a[n-1:i]$

Note that, for MIPS-86, we will use the aforementioned definitions only for $n = 32$. We define the result of a shift operation based on inputs $a \in \mathbb{B}^n$, $i \in \{0, \ldots, n-1\}$, and $sf \in \mathbb{B}^2$ as follows:

$$slures(a, i, sf) = \begin{cases} sll(a, i) & sf = 00 \\ srl(a, i) & sf = 10 \\ sra(a, i) & sf = 11 \end{cases}$$

We define a predicate that, given an instruction $I \in \mathbb{B}^{32}$, expresses whether the instruction is a shift instruction by a simple disjunction of shift instruction predicates:

$$su(I) \equiv sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

Given a shift instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the following shift operands:

- shift distance: $sdist(c, I) = \begin{cases} \langle sa(I) \rangle \bmod 32 & fun(I)[3] = 0 \\ \langle c.gpr(rs(I))[4:0] \rangle \bmod 32 & fun(I)[3] = 1 \end{cases}$
- shift left operand: $slop(c, I) = c.gpr(rt(I))$

The shift function of a shift instruction $I \in \mathbb{B}^{32}$ is given by

$$sf(I) = I[1:0]$$

Memory Accesses We define auxiliary functions that we need in order to define how values are read/written from/to the memory in the overall system's transition function. Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the effective address, access width and byte write signal of a memory access:

- extended effective address: $ea(c, I) = \begin{cases} c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & itype(I) \\ c.gpr(rs(I)) & rtype(I) \end{cases}$

- access width: $d(I) = \begin{cases} 1 & lb(I) \vee lbu(I) \vee sb(I) \\ 2 & lh(I) \vee lhu(I) \vee sh(I) \\ 4 & sw(I) \vee lw(I) \vee rmw(I) \end{cases}$

- byte write signal: $bw(c, I) = \begin{cases} 0001 & lb(I) \vee lbu(I) \vee sb(I) \wedge ea(c, I)[1:0] = 00 \\ 0010 & lb(I) \vee lbu(I) \vee sb(I) \wedge ea(c, I)[1:0] = 01 \\ 0100 & lb(I) \vee lbu(I) \vee sb(I) \wedge ea(c, I)[1:0] = 10 \\ 1000 & lb(I) \vee lbu(I) \vee sb(I) \wedge ea(c, I)[1:0] = 11 \\ 0011 & lh(I) \vee lhu(I) \vee sh(I) \wedge ea(c, I)[1] = 0 \\ 1100 & lh(I) \vee lhu(I) \vee sh(I) \wedge ea(c, I)[1] = 1 \\ 1111 & lw(I) \vee sw(I) \vee rmw(I) \end{cases}$

In $ea(c, I)$ the $ea(c, I)[31 : 2]$ is the effective address and $ea(c, I)[1 : 0]$ is used to compute the byte write signal. The access width is the number of bytes that are read, or, respectively, written. The byte write signal is a flag denoting the location of the target value. We define the misalignment on fetch predicate as follows:

$$malf(c) \equiv c.pc[1:0] \neq 00$$

For an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the misalignment on load/store predicate as follows:

$$\begin{aligned} malls(c, I) \quad \equiv \quad & (lw(I) \vee sw(I) \vee rmw(I)) \wedge ea(c, I)[1:0] \neq 00 \\ & \vee (lhu(I) \vee lh(I) \vee sh(I)) \wedge ea(c, I)[0] \neq 0 \end{aligned}$$

that describes whether the memory access is misaligned. Note that misaligned memory access triggers the corresponding interrupt. In order to denote whether a given instruction $I \in \mathbb{B}^{32}$ is a load or store instruction, we define the following predicates:

- load instruction: $load(I) \equiv lw(I) \vee lhu(I) \vee lh(I) \vee lbu(I) \vee lb(I)$

- store instruction: $store(I) \equiv sw(I) \vee sh(I) \vee sb(I)$

Given a value $v \in \mathbb{B}^{32}$ and an instruction $I \in \mathbb{B}^{32}$, we define the shift for load function

$$s4l(v, I) = \begin{cases} srl(v, 8 \cdot i) & (lb(I) \vee lbu(I)) \wedge bw(c, I)[i] \\ srl(v, 16) & (lh(I) \vee lhu(I)) \wedge bw(c, I)[3] \\ v & otherwise \end{cases}$$

The value read from memory $R \in \mathbb{B}^{32}$ is given as an input to the transition function of the processor core. In order to write this value to a general purpose register, depending on the

memory instruction used, we either need to sign-extend or zero-extend this value:

$$zxt_{32}(v) = 0^{32-|v|} \circ v$$
$$sxt_{32}(v) = v[0]^{32-|v|} \circ v$$
$$lv(R, I) = \begin{cases} zxt_{32}(s4l(R, I)[7:0]) & lbu(I) \\ zxt_{32}(s4l(R, I)[15:0]) & lhu(I) \\ sxt_{32}(s4l(R, I)[7:0]) & lb(I) \\ sxt_{32}(s4l(R, I)[15:0]) & lh(I) \\ R & otherwise \end{cases}$$

Given an instruction $I \in \mathbb{B}^{32}$ and a value $v \in \mathbb{B}^{32}$, we define the shift for store function

$$s4s(v, I) = \begin{cases} sll(v, 8 \cdot i) & sb(I) \wedge bw(c, I)[i] \\ sll(v, 16) & sh(I) \wedge bw(c, I)[3] \\ v & otherwise \end{cases}$$

Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, the store value is given by the last $d(I)$ bytes taken from the general purpose register specified by $rt(I)$:

$$sv(c, I) = s4s(c.gpr(rt(I)), I)$$

General Purpose Register Updates The predicate

$$gprw(I) \equiv alu(I) \vee su(I) \vee lw(I) \vee rmw(I) \vee jal(I) \vee jalr(I) \vee movs2g(I)$$

describes whether a given instruction $I \in \mathbb{B}^{32}$ results in a write to some general purpose register. We define the result destination of an ALU/shift/coprocessor/memory instruction $I \in \mathbb{B}^{32}$ as the following general purpose register address:

$$rdes(I) = \begin{cases} rd(I) & rtype(I) \wedge /movs2g(I) \\ rt(I) & otherwise \end{cases}$$

For an instruction $I \in \mathbb{B}^{32}$, the address of the general purpose register which is actually written to is defined as

$$cad(I) = \begin{cases} 1^5 & jal(I) \vee jalr(I) \\ rdes(I) & alu(I) \vee load(I) \vee rmw(I) \end{cases}$$

We define the value written to the general purpose register specified above based on the instruction $I \in \mathbb{B}^{32}$ and a given processor core configuration $c \in K_{\mathbf{core}}$ as

$$gprdin(c, I, R) = \begin{cases} c.pc +_{32} 4_{32} & jal(I) \vee jalr(I) \\ lv(R, I) & load(I) \vee rmw(I) \\ c.spr(rd(I)) & movs2g(I) \\ alures(lop(c, I), rop(c, I), alucon(I)) & alu(I) \\ sures(slop(c, I), sdist(c, I), sf(I)) & su(I) \end{cases}$$

Definition of Instruction Execution

Based on the auxiliary functions defined in the last subsection, we give the definition of instruction execution in closed form:

Definition 3.2 (Non-Interrupted Instruction Execution) We define the transition function for non-interrupted instruction execution

$$\delta_{instr} : K_{core} \times \Sigma_{instr} \rightharpoonup K_{core}$$

where

$$\Sigma_{instr} = \mathbb{B}^{32} \times (\mathbb{B}^{32} \cup \{\bot\})$$

as

$$\delta_{instr}(c, I, R) = \begin{cases} \bot & (load(I) \vee rmw(I)) \wedge R = \bot \\ c' & otherwise \end{cases}$$

where

- $c'.pc = \begin{cases} btarget(c, I) & jbtaken(c, I) \\ c.spr(epc) & eret(I) \\ c.pc +_{32} 4_{32} & otherwise \end{cases}$
- $c'.gpr(x) = \begin{cases} gprdin(c, I, R) & x = cad(I) \wedge gprw(I) \\ c.gpr(x) & otherwise \end{cases}$
- $c'.spr(x) = \begin{cases} c.gpr(rt(I)) & rd(I) = x \wedge movg2s(I) \\ c.spr(emode) & x = mode \wedge eret(I) \\ c.spr(esr) & x = sr \wedge eret(I) \\ c.spr(x) & otherwise \end{cases}$

Auxiliary Definitions for Triggering of Interrupts

MIPS-86 provides the following interrupt types that are ordered by their priority (interrupt level): Note that the all continue interrupts are either triggered by the execution of ALU operations with overflow or execution of the system call instruction.

Here external event signals are provided as input $eev \in \mathbb{B}^{256}$, in which the $eev[0]$ is the reset signal and $eev[1 : 255]$ is the device interrupt triggered by signals from the environment of the processor, to the processor core transition function. The page-fault signals $pff, pfls \in \mathbb{B}$ are provided by the MMU of the processor to the processor core transition function. In hardware, one interrupt can only happen either in the fetch phase (f) or in the execute phase (x). In the last column of Table 3.5 we show at which phase the corresponding interrupt can take place in the hardware. The priority of interrupts is based on the following rules:

- External interrupts have the highest priority. The reason is that unlike internal interrupts which can be reproduced by repeating computation steps, the external interrupts are inputs from the environment and can not be reproduced. Also, external interrupts have the

Table 3.5: MIPS-86 Interrupt Types and Priority.

level	shorthand	int/ext	type	maskable	description	phase
0	reset	eev	abort	0	reset	f
1	dev	eev	repeat	1	devices	f
2	malf	iev	abort	0	misaligned fetch	f
3	pff	iev	repeat	0	page fault fetch	f
4	ill	iev	abort	0	illegal instruction	x
5	sysc	iev	continue	0	system call	x
6	ovf	iev	continue	1	overflow	x
7	malls	iev	abort	0	misaligned load/store	x
8	pfls	iev	repeat	0	page fault load/store	x

abort type which means the computation is ended or repeat type which can reproduce the internal interrupts after the external one is handled. Since for one step of MIPS, there is only one *eev*, we can assume them are handled in the fetch phase

- Interrupts which can only happen in the fetch phase must have the higher priority than the interrupts which an only happen in the execute phase.
- Misalignment interrupts have higher priority than corresponding page fault interrupts. Because if a misalignment happen, there can not be any memory accesses to the misalignment address.
- Illegal instruction interrupt has the highest priority among all the interrupts which can only happen in the execute phase.

To simplify the proof in Chapter 4, we assume the page fault interrupts have the lowest priority in corresponding phases.

Definition 3.3 (Cause and Masked Cause of an Interrupt on Fetch Phase) We define the cause on fetch $ca_f \in \mathbb{B}^{32}$ of an interrupt and masked cause on fetch $mca_f \in \mathbb{B}^{32}$ of an inerrupt based on the current core configuration $c \in K_{core}$ to be executed, the external event vector $eev \in \mathbb{B}^{256}$ and the page-fault on fetch signals $pff \in \mathbb{B}$ as follows:

- cause of an interrupt on fetch:

$$ca_f(c, eev, pff)[j] \equiv \begin{cases} eev[0] & j = 0 \\ \bigvee_{i=1}^{255} eev[i] & j = 1 \\ c.pc[1:0] \neq 00 & j = 2 \\ pff & j = 3 \\ 0 & otherwise \end{cases}$$

- masked cause of an interrupt on fetch:

$$mca_f(c, eev, pff)[j] \equiv \begin{cases} ca_f(c, eev, pff)[j] \wedge c.spr(sr)[j] & j = 1 \\ ca_f(c, eev, pff)[j] & otherwise \end{cases}$$

  Only interrupt level 1 is maskable on the fetch phase; the corresponding mask can be found in special purpose register *sr* (status register) and is applied to the cause of interrupt to obtain the masked cause.

Definition 3.4 (Cause and Masked Cause of an Interrupt on Execution Phase) We define the cause $ca_x \in \mathbb{B}^{32}$ of an interrupt on execution and masked cause $mca_x \in \mathbb{B}^{32}$ of an interrupt on execution based on the current processor core configuration $c \in K_{core}$, the instruction $I \in \mathbb{B}^{32}$ to be executed, the external event vector $eev \in \mathbb{B}^{256}$ and the page-fault on load/store signals $pfls \in \mathbb{B}$ as follows:

- cause of interrupt:

$$ca_x(c, I, pfls)[j] = \begin{cases} ill(I) \vee c.spr(mode)[0] \wedge (movs2g(I) \vee movg2s(I) \vee eret(I)) & j = 4 \\ sysc(I) & j = 5 \\ ovf(lop(c, I), rop(c, I), alucon(I), itype(I)) & j = 6 \\ ea(c, I)[1:0] \notin \{00, \perp\} & j = 7 \\ pfls & j = 8 \\ 0 & otherwise \end{cases}$$

- masked cause:

$$mca_x(c, I, pfls)[j] = \begin{cases} ca_x(c, I, pfls)[j] \wedge c.spr(sr)[j] & j = 6 \\ ca_x(c, I, pfls)[j] & otherwise \end{cases}$$

  Note that only interrupt levels 1 and 6 are maskable.

To denote that in a given configuration $c \in K_{\mathbf{core}}$, external event signals $eev \in \mathbb{B}^{256}$ and page-fault signals $pff \in \mathbb{B}$ an interrupt is triggered during fetch, we define the predicate

$$jisr_f(c, eev, pff) \equiv \bigvee_j mca_f(c, eev, pff)[j]$$

Given a configuration $c \in K_{\mathbf{core}}$, an instruction $I \in \mathbb{B}^{32}$, external event signals $eev \in \mathbb{B}^{256}$ and page-fault signals $pfls \in \mathbb{B}$ an interrupt is triggered during execution, we define the predicate

$$jisr_x(c, I, pfls) \equiv \bigvee_j mca_x(c, I, pfls)[j]$$

The overall interrupt predicate is defined as

$$jisr(c, I, eev, pff, pfls) \equiv jisr_f(c, eev, pff) \vee jisr_x(c, I, pfls)$$

Note that in a MIPS machine, at most one of the predicates among $jisr_x$ and $jisr_f$ can be true. To determine the interrupt level of the triggered interrupt on fetch phase, we define the function

$$il_f(c, eev, pff) = min\{j \mid mca_f(c, eev, pff)[j] = 1\}$$

To determine the interrupt level of the triggered interrupt on execute phase, we define the function

$$il_x(c, I, pfls) = min\{j \mid mca_x(c, I, pfls)[j] = 1\}$$

The predicate

$$continue(c, I, pfls) \equiv il_x(c, I, pfls) \in \{5, 6\}$$

denotes whether the triggered interrupt is of continue type.

Definition of Interrupt Execution

Definition 3.5 (Interrupt Execution Transition Function) We define $\delta_{jisr_f}(c, eev, pff) = c'$ and $\delta_{jisr_x}(c, I, eev, pfls) = c''$ where $I \in \mathbb{B}^{32}$ is the instruction to be executed, $eev \in \mathbb{B}^{256}$ are the external event signals and $pff, pfls \in \mathbb{B}$ are the page-fault signals provided by the processor's MMU.

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

- $c'.pc = c''.pc = 0^{32}$
- $$c'.spr(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.spr(mode) & x = emode \\ c.spr(sr) & x = esr \\ mca_f(c, eev, pff) & x = eca \\ c.pc & x = epc \\ bin_{32}(k) & x = edata \wedge il_f(c, eev, pff) = 1 \\ c.spr(x) & otherwise \end{cases}$$
- $$c''.spr(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.spr(mode) & x = emode \\ c.spr(sr) & x = esr \\ mca_x(c, I, pfls) & x = eca \\ c.pc & x = epc \wedge \neg continue(c, I, pfls) \\ \delta_{instr}(c, I, \perp).pc & x = epc \wedge continue(c, I, pfls) \\ ea(c, I) & x = edata \wedge il_x(c, I, pfls) = 8 \\ c.spr(x) & otherwise \end{cases}$$
- $c'.gpr = c.gpr$

- $c''.gpr = \begin{cases} c.gpr & \neg continue(c, I, pfls) \\ \delta_{instr}(c, I, \bot).gpr & otherwise \end{cases}$

Processor Core Transition Function

Definition 3.6 (Processor Core Transition Function)

$$\delta_{core} : K_{core} \times \Sigma_{core} \rightarrow K_{core}$$

In which:

$$\Sigma_{core} = \mathbb{B}^{32} \times (\mathbb{B}^{32} \cup \{\bot\}) \times \mathbb{B}^{256} \times \mathbb{B} \times \mathbb{B}$$

$$\delta_{core}(c, I, R, eev, pff, pfls) = \begin{cases} \delta_{jisr_f}(c, eev, pff) & jisr_f(c, eev, pff) \\ \delta_{jisr_x}(c, I, pfls) & jisr_x(c, I, pfls) \\ \delta_{instr}(c, I, R) & otherwise \end{cases}$$

3.1.2 Memory

In MIPS-86 ISA we have a word addressable shared memory with byte write signals:

$$m \in \mathbb{B}^{30} \rightarrow \mathbb{B}^{32} = K_m$$

The function $cb(v_1, v_2, bw) \in \mathbb{B}^{32}$ is used to combine two values $v_1, v_2 \in \mathbb{B}^{32}$ according to the byte write signal $bw \in \mathbb{B}^4$:

$$cb(v_1, v_2, bw) = data$$

where:

$$byte(i, data) = \begin{cases} byte(i, v_2) & bw[i] \\ byte(i, v_1) & otherwise \end{cases}$$

Definition 3.7 (Memory Transition Function) The memory transition function is defined as:

$$\delta_m \in K_m \times \Sigma_m \rightarrow K_m$$

where:

$$\Sigma_m = \mathbb{B}^{30} \times \mathbb{B}^{32} \times \mathbb{B}^4 \cup \mathbb{B}^{32} \times \mathbb{B}^{30} \times \mathbb{B}^{32}$$

Here,

- $(a, v, bw) \in \mathbb{B}^{30} \times \mathbb{B}^{32} \times \mathbb{B}^4$ – describes a write access to address a with value v and the byte write signal bw, and

- $(c, a, v) \in \mathbb{B}^{32} \times \mathbb{B}^{30} \times \mathbb{B}^{32}$ – describes a read-modify-write access to address a with compare-value c and value v to be written in case of success.

We have:

$$\delta_m(m, in)(x) = \begin{cases} cb(m(a), v, bw) & in = (a, v, bw) \wedge x = a \\ v & in = (c, a, v) \wedge m(a) = c \wedge x = a \\ m(x) & otherwise \end{cases}$$

3.1.3 Store Buffer

A store buffer is a FIFO queue between the processor core and the shared memory. It accumulates outgoing processor stores and, if available, forwards requested data on processor loads.

Definition 3.8 (Store Buffer Configuration) The store buffer entry configuration is defined as:

$$K_{sbe} = \{(a, v, bw) \mid a \in \mathbb{B}^{30} \wedge v \in \mathbb{B}^{32} \wedge bw \in \mathbb{B}^4\}$$

The store buffer configuration is defined as follows:

$$K_{sb} = K_{sbe}^*$$

We define some auxiliary function for store buffer forwarding. The transitions of store buffer are formalized in the processor transition relation - we do not provide an individual transition relation for the store buffer. Given a store buffer entry $(a, v, bw) \in K_{sbe}$ and a word address $x \in \mathbb{B}^{30}$, we define the predicate:

$$sbehit((a, v, bw), x) \equiv x = a$$

The function $maxsbhit(sb, x) \in \mathbb{N}$ computes the index of the newest entry of the store buffer $sb \in K_{sb}$ for which there is a hit at address $x \in \mathbb{B}^{30}$.

$$maxsbhit(sb, x) = max\{j \mid sbehit(sb[j], x)\}$$

The following predicate $sbhit(sb, x)$ denotes whether there is a store buffer hit in $sb \in K_{sb}$ at address $x \in \mathbb{B}^{30}$.

$$sbhit(sb, x) \equiv \exists j.\ sbehit(sb[j], x)$$

3.1.4 Translation Lookaside Buffer

In MIPS-86, we proved the virtual memory to implement the memory isolation for each thread. By performing address translation from virtual memory addresses to physical memory addresses (i.e. regular memory addresses of the machine's memory system), the notion of a virtual memory is established-if this translation is injective, virtual memory has regular memory semantics (i.e. writing to an address affects only this single address and values being written can be read again

later). Processors tend to provide a mechanism to activate and deactivate address translation-usually by writing some special control register. In the case of MIPS-86, a special purpose register *mode* is provided which decides whether the processor is running in system mode, i.e. without address translation, or in user mode, i.e. with address translation.

The MIPS-86 applies a 2 level page table hierarchy. One advantage of this is that multi-level page tables tend to require less memory space: only the necessary part of the page table is provided. The disadvantage for the multi-level page table is that more memory lookups are introduced for each memory reference. In order to increase the speed of address translation, we introduce a shared cache called translation lookaside buffer (TLB) between the processor core and the MMU. The purpose of a TLB is to cache address translations done by the MMU and to reuse them later without performing additional memory accesses to page tables. A modern TLB caches not only address translations themselves, which could by considered as complete page table traversals, but also intermediate states of such traversals, which we call walks.

Definition 3.9 (TLB Configuration) The set of TLB configuration is defined as follow:

$$K_{tlb} = 2^{K_{walk}}$$

where the set of walk configuration is given by:

$$K_{walk} = \mathbb{B}^{20} \times \{0, 1, 2\} \times \mathbb{B}^{20} \times \mathbb{B}^3 \times \mathbb{B}$$

A walk $w \in K_{walk}$ consists the following components:

- $w.va \in \mathbb{B}^{20}$. The virtual page address to be translated.
- $w.level \in \{0, 1, 2\}$. The current level of the walk. If it is 0, we call w a complete walk. Otherwise $w.level$ is the number of remaining walk extensions to obtain a complete walk.
- $w.ba \in \mathbb{B}^{20}$. The pointer to the target physical page if $w.level = 0$ otherwise to the next level of page table.
- $w.r \in \mathbb{B}^3$. The accumulated request rights. $w.r = (wr \in \mathbb{B}, us \in \mathbb{B}, ex \in \mathbb{B})$ stands for write permission, user mode access permission and execute permission respectively.
- $w.fault \in \mathbb{B}$. The page fault flag.

Since MIPS-86 is a 32-bit architecture with a word addressable memory, each page consists of 2^{10} consecutive words and a *page address* consists of 20 Bits. The first level page table translates the first 10 Bits of a page address and the second level translates the remaining 10 Bits.

Definition 3.10 (Page Index and Base Address) Given an address $a \in \mathbb{B}^{30}$ we have:

$$a = a.px_2 \circ a.px_1 \circ a.px_0$$

In which

- $a.px_2 \in \mathbb{B}^{10}$. The second-level page index.
- $a.px_1 \in \mathbb{B}^{10}$. The first-level page index.
- $a.px_0 \in \mathbb{B}^{10}$. The offset within a page.

The base address for a is defined as:

$$a.ba = a.px_2 \circ a.px_1$$

Definition 3.11 (Page Table Entry) A page table entry $pte \in \mathbb{B}^{32}$ consists of

- $pte.ba = pte[31 : 12]$. The base address of the next page table or, if the page table is a terminal one, the resulting physical page address for a translation,
- $pte.p = pte[11]$. The present bit,
- $pte.r = pte[10 : 8]$. The access rights for pages accessed via a translation that involves the page table entry,
- $pte.a = pte[7]$. The accessed flag that denotes whether the MMU has already used the page table entry for a translation, and
- $pte.d = pte[6]$. The dirty flag that denotes whether the MMU has already used the page table entry for a translation that had write rights. This particular field is only used for terminal page tables.

For a base address $ba \in \mathbb{B}^{20}$ and an index $i \in \mathbb{B}^{10}$, we define the corresponding *page table entry address* as

$$ptea(ba, i) = ba \circ 0^{10} +_{32} 0^{20}i$$

The page table entry address needed to extend a given walk $w \in K_{walk}$ is then defined as

$$ptea(w) = ptea(w.ba, (w.va \circ 0^{10}).px_{w.level})$$

Given a memory $m \in K_m$ and a walk $w \in K_{walk}$, we define the page table entry needed to extend a walk as

$$pte(m, w) = m(ptea(w))$$

Definition 3.12 (Walk Creation) We define the function

$$winit : \mathbb{B}^{20} \times \mathbb{B}^{20} \to K_{walk}$$

which, given a virtual base address $va \in \mathbb{B}^{20}$, the base address $pto \in \mathbb{B}^{20}$ of the page table origin and returns the initial walk for the translation of va.

$$winit(va, pto) = w$$

is given by

$$w.va = va \quad w.level = 2 \quad w.ba = pto$$
$$w.r = 111 \quad w.fault = 0$$

Note that in our specification of the MMU, the initial walk always has full rights ($w.r = 111$). However, in every translation step, the rights associated with the walk can be restricted as needed by the translation request made by the processor core.

Definition 3.13 (Sufficient Access Rights) For a pair of access rights $r, r' \in \mathbb{B}^3$, we use

$$r \leq r' \equiv \forall j \in [0:2] : r[j] \leq r'[j]$$

to describe that the access rights r are weaker than r', i.e. rights r' are sufficient to perform an access with rights r.

Definition 3.14 (Walk Extension) We define the function

$$wext : K_{walk} \times \mathbb{B}^{32} \rightarrow K_{walk}$$

which extends a given walk $w \in K_{walk}$ using a page table entry $pte \in \mathbb{B}^{32}$ in such a way that

$$wext(w, pte) = w'$$

is given by

$$w'.va = w.va \qquad w'.fault = \neg pte.p \vee \neg w.r \leq pte.r$$
$$w'.level = \begin{cases} w.level - 1 & pte.p \\ w.level & otherwise \end{cases} \qquad w'.ba = \begin{cases} pte.ba & pte.p \\ w.ba & otherwise \end{cases}$$
$$w'.r = \begin{cases} w.r \wedge pte.r & pte.p \\ w.r & otherwise \end{cases}$$

Definition 3.15 (Complete Walk) A walk $w \in K_{walk}$ with $w.level = 0$ is called a complete walk:

$$complete(w) \equiv w.level = 0$$

Definition 3.16 (Setting Accessed/Dirty Flags of a Page Table Entry) Before extending a walk w the MMU sets access and dirty bits in the page table entry used to extend w. Given a page table entry $pte \in \mathbb{B}^{32}$ and a walk $w \in K_{walk}$, we define the function

$$set\text{-}ad(w, pte) = \begin{cases} pte[a := 1, d := 1] & w.r[0] \wedge w.level = 1 \wedge pte.r[0] \\ pte[a := 1] & otherwise \end{cases}$$

which returns an updated page table entry in which the accessed and dirty bits are updated when walk w is extended using pte. Extending a walk with write access right using a terminal page table results in the dirty flag being set for the page table entry. Otherwise, only the accessed flag is set.

Definition 3.17 (Translation Request) A translation request

$$trq = (trq.va, trq.r) \in \mathbb{B}^{30} \times \mathbb{B}^{3}$$

is a pair of

- virtual address $trq.va \in \mathbb{B}^{30}$, and
- access rights $trq.r \in \mathbb{B}^{3}$.

Definition 3.18 (Walk Match) When a walk w only matches a translation request trq in the virtual address, we call this a walk match:

$$match(trq, w) \equiv w.va = trq.va[29 : 10]$$

Definition 3.19 (Walk Hit) When a walk w matches a translation request trq in terms of virtual address and access rights, we call this a walk hit:

$$hit(trq, w) \equiv w.va = trq.va[29 : 10] \wedge trq.r \leq w.r$$

Note that a hit or a match may occur with an incomplete walk.

Definition 3.20 (Faulty Walk) A page fault for a given walk would result if: (i) during a walk extension the page table entry needed to extend is not present or the walk would require more access rights than the page table entry provides, (ii) the matched translation request requires more rights than the walk provides. To denote this, we define the predicate

$$fault(pte, trq, w) \equiv /complete(w) \wedge wext(w, pte).fault \vee match(trq, w) \wedge \neg hit(trq, w)$$

Note that a page fault may occur at any translation level. However, the TLB will only store non-faulty walks (this is an invariant of the TLB) – page faults are triggered by a faulty walk extension or a violation of access rights.

In the top-level transition function of MIPS-86, the transition request hit is introduced as a precondition. The page faults are triggered as follows: the processor core always chooses walks from the TLB non-deterministically to either obtain a translation, or, to get a page-fault when the chosen walk has a faulty walk extension or the access rights are violated. Note that, when a page-fault for a given virtual address occurs, MIPS-86 flushes all faulty walks from the TLB.

Definition 3.21 (Transition Function of the TLB) We define the transition function of the TLB that states the transitions of the TLB

$$\delta_{tlb} : K_{tlb} \times \Sigma_{tlb} \rightarrow K_{tlb}$$

where

$$\Sigma_{tlb} = \{\textbf{flush}\} \times \mathbb{B}^{20} \cup \{\textbf{add-walk}\} \times K_{walk}$$

as a case distinction on the given input:

- flushing a virtual address for a given address space identifier:

$$\delta_{tlb}(tlb, (\textbf{flush}, va)) = \{w \in tlb \mid w.va \neq va\}$$

- adding a walk:

$$\delta_{tlb}(tlb, (\textbf{add-walk}, w)) = tlb \cup \{w\}$$

3.1.5 Sequential MIPS

In order to apply the SB reduction theorem to the MIPS-86 ISA, in addition to the MIPS-86 machine, we also need to define the machine without SB. We call it the SB reduced MIPS-86 machine. In this section, we will give the definition of sequential MIPS-86 machine and sequential SB reduced MIPS-86 machine.

Configuration

Definition 3.22 (Processor Configuration) The set of processor configurations is defined as follows:

$$K_{pro} = K_{core} \times K_{sb} \times K_{tlb}$$

A processor $p = (p.core, p.sb, p.tlb) \in K_{pro}$ contains a processor core, a store buffer and a translation lookaside buffer.

Definition 3.23 (SB Reduced Processor Configuration) The set of SB reduction processor configuration is defines as follows:

$$K_{sbr\text{-}pro} = K_{core} \times K_{tlb}$$

An SB reduced processor $p_{sbr} = (p_{sbr}.core, p_{sbr}.tlb) \in K_{sbr\text{-}pro}$ contains a processor core and a translation lookaside buffer.

Definition 3.24 (Sequential MIPS-86 Machine Configuration)

$$K_{seq} = K_{pro} \times K_m$$

A sequential MIPS-86 machine configuration $c = (c.p, c.m) \in K_{seq}$ consists of a processor and a memory.

Definition 3.25 (SB Reduced Sequential MIPS-86 Machine Configuration)

$$K_{sbr\text{-}seq} = K_{sbr\text{-}pro} \times K_m$$

An SB reduced sequential MIPS-86 machine configuration $c_{sbr} = (c_{sbr}.p_{sbr}, c_{sbr}.m) \in K_{sbr\text{-}seq}$ consists of an SB reduced processor and a memory.

Transition Function

Definition 3.26 (Memory System) The results of read accesses performed by the processor core are described in terms of a memory system that takes into account the store buffer and the memory. We define a function ms that, given these components, returns the merged memory view seen by the processor core:

$$ms(sb, m)(x, bw) = \begin{cases} m(x) & \neg sbhit(sb, x) \\ sb[j].v & maxsbhit(sb, x) = j \wedge bw \leq sb[j].bw \\ \bot & otherwise \end{cases}$$

Definition 3.27 (Input of Processor Transition Function)

$$\begin{aligned}\Sigma_{seq} = &\{\textbf{core}\} \times K_{\textbf{walk}} \times K_{\textbf{walk}} \times \mathbb{B}^{256} \\ &\cup \{\textbf{tlb-create}\} \times \mathbb{B}^{20} \\ &\cup \{\textbf{tlb-extend}\} \times K_{\textbf{walk}} \\ &\cup \{\textbf{tlb-accessed-dirty}\} \times K_{\textbf{walk}} \\ &\cup \{\textbf{sb}\}\end{aligned}$$

- $in = (\textbf{core}, w_I, w_R, eev) \in \Sigma_{seq}$. The processor performs a core step using walks w_I and w_R. w_R is ignored if not necessary.
- $in = (\textbf{tlb-create}, ba) \in \Sigma_{seq}$. The MMU, which is implicitly modeled in the processor, performs a TLB step to create a walk for base address ba.
- $in = (\textbf{tlb-extend}, w) \in \Sigma_{seq}$. The MMU performs a TLB step to extend the walk w.
- $in = (\textbf{tlb-accessed-dirty}, w) \in \Sigma_{seq}$. The MMU sets the access and dirty bit of a page table entry needed to extend w.
- $in = (\textbf{sb}) \in \Sigma_{seq}$. The processor performs an SB step to update the memory.

Definition 3.28 (Input of SB Reduced Processor Transition Function)

$$\begin{aligned}\Sigma_{sbr\text{-}seq} = &\{\textbf{core}\} \times K_{\textbf{walk}} \times K_{\textbf{walk}} \times \mathbb{B}^{256} \\ &\cup \{\textbf{tlb-create}\} \times \mathbb{B}^{20} \\ &\cup \{\textbf{tlb-extend}\} \times K_{\textbf{walk}} \\ &\cup \{\textbf{tlb-accessed-dirty}\} \times K_{\textbf{walk}}\end{aligned}$$

Since the SB reduced processor does not make an SB step, the input of the SB reduced processor transition function do not have **sb** as a parameter.

Definition 3.29 (Sequential MIPS-86 Transition Function)

$$\delta_{seq} : K_{seq} \times \Sigma_{seq} \rightharpoonup K_{seq}$$

$$\delta_{seq}(c, in) = c'$$

We make a case split on in.

- $in = (\textbf{core}, w_I, w_R, eev)$. In this case we define some shorthands:
  - $\forall X \in \{gpr, spr, pc\}.\ p.X = c.p.core.X$.
  - $\forall Y \in \{sb, tlb\}.\ p.Y = c.p.Y$.
  - $mode \equiv p.gpr(mode)[0]$.

- $trqI = (p.pc[31:2], 011)$. The translation request for instruction fetch.
- $pff \equiv mode \wedge fault(pte(c.m, w_I), trqI, w_I)$. Signals whether there is a page-fault-on-fetch for the given walk w_I and the translation request $trqI$.
- $pmaI = \begin{cases} w_I.ba \circ p.pc[11:2] & mode \\ p.pc[31:2] & otherwise \end{cases}$. The physical memory address for instruction fetch of processor core i (which is only meaningful if no page-fault on instruction fetch occurs),
- $I = c.m(pmaI)$. The instruction fetched from memory. Because the self-modifying code is forbidden, we can directly read from memory (in case of a page-fault-on-fetch the value of I has no further relevance).
- $switch(I) \equiv movg2s(I) \wedge \langle rd(I) \rangle = 6$.
- $wpto(I) \equiv movg2s(I) \wedge \langle rd(I) \rangle = 5$.
- $sbf(I) \equiv rmw(I) \vee mfence(I) \vee invlpg(I) \vee flush(I) \vee eret(I) \vee wpto(I) \vee switch(I)$.
- $trqEA = (ea(p.core, I)[31:2], (store(I) \vee rmw(I)) \circ 10)$. The translation request for the effective address.
- $pfls \equiv mode \wedge fault(pte(c.m, w_R), trqEA, w_R) \wedge \neg pff \wedge (store(I) \vee load(I) \vee rmw(I))$. The page-fault-on-load-store signal.
- $pmaEA = \begin{cases} w_R.ba \circ ea(p.core, I)[11:2] & mode \\ ea(p.core, I)[31:2] & otherwise \end{cases}$. the physical memory address for the effective address.
- $R = \begin{cases} \bot & pff \vee pfls \\ ms(p.sb, c.m)(pmaEA, bw(p.core, I)) & otherwise \end{cases}$. The value read from the memory system.

c' is defined iff:

- $mode \rightarrow hit(w_I, trqI)$. The walk w_I must match the translation request for instruction fetch.
- $mode \rightarrow (store(I) \vee load(I) \vee rmw(I)) \wedge \neg pff \rightarrow hit(w_R, trqEA)$. If there is a read or write instruction and no page-fault on fetch has occurred, the walk w_R must match the translation request for the effective address.
- $\neg pff \rightarrow complete(w_I)$. If there is no page-fault on fetch, walk w_I is complete, and thus, provides a translation from virtual to physical address.
- $\neg pfls \rightarrow complete(w_R)$. If there is no page-fault on load/store, walk w_R is complete, and thus, provides a translation from virtual to physical address.
- $sbf(I) \vee jisr(p.core, I, eev, pff, pfls) \rightarrow p.sb = []$. The store buffer is flushed by an interrupt and an instruction can only be executed when the store buffer is empty if it is
  - a read modify write instruction, or

* a fence instruction, or
* a TLB flush instruction (partially or totally), or
* an instruction which updates the special purpose register *mode* or *pto*.

Then the c' is defined as:

$$c'.p.core = \begin{cases} \delta_{core}(p.core, I, R, eev, pff, pfls) & (load(I) \vee rmw(I)) \\ \delta_{core}(p.core, I, \perp, eev, pff, pfls) & (\neg load(I) \wedge \neg rmw(I)) \\ p.core & otherwise \end{cases}$$

$$c'.p.sb = \begin{cases} p.sb \circ (pmaEA, sv(p.core, I), bw(p.core, I)) & store(I) \\ p.sb & otherwise \end{cases}$$

$$c'.p.tlb = \begin{cases} \emptyset & flush(I) \\ \delta_{tlb}(p.tlb, (\textbf{flush}, p.gpr(rd(I)).ba)) & invlpg(I) \\ \delta_{tlb}(p.tlb, (\textbf{flush}, p.pc.ba)) & pff \\ \delta_{tlb}(p.tlb, (\textbf{flush}, ea(p.core, I).ba)) & \neg pff \wedge pfls \\ p.tlb & otherwise \end{cases}$$

$$c'.m = \begin{cases} \delta_m(c.m, (p.gpr(rd(I)), pmaEA, sv(p.core, I))) & rmw(I) \\ c.m & otherwise \end{cases}$$

- in = (**tlb-create**, va). A new walk for virtual address va is created in TLB. c' is defined iff
  - *mode*. The TLB only create walk when the processor is running in user mode.

$$c'.p.tlb = p.tlb \cup winit(va, p.spr(pto).ba)$$

  Creating a new walk in the TLB is a step that affects only the TLB.

- in = (**tlb-set-accessed-dirty**, w). Accessed and dirty bits of the page table entry needed to extend walk w in TLB are set appropriately

  c' is defined iff:

  - *mode*. Page table flags can only be set in translated mode.
  - $w \in p.tlb \wedge \neg complete(w)$. We only set dirty and accessed bits for incomplete walks.
  - $pte(c.m, w).p = 1$. The MMU can only set accessed/dirty flags for page table entries which are actually present.

Then,

$$c'.m = \delta_m(c.m, (ptea(w), set\text{-}ad(w, pte(c.m, w)), 1^4))$$

Setting the page table entry flags only affects the corresponding page table entry in memory. In this model, the MMU non-deterministically sets accessed and dirty flags – enabling walk extension using the given page table entry.

- $in = (\textbf{tlb-extend}, w)$. An existing walk in TLB is extended

  We let $pte = pte(c.m, w)$ then c' is defined iff:

  - $mode$. The MMU can only extend a walk in translated mode.
  - $w \in p.tlb$. The walk to be extended is contained in the TLB.
  - $\neg complete(w)$. The walk is not yet complete.
  - $pte.a \wedge pte.p \wedge (w.level = 1 \wedge w.r[0] \wedge pte.r[0] \rightarrow pte.d)$. The present and accessed/dirty flags are set appropriately.
  - $\neg wext(w, pte).fault$. The walk extension does not result in a faulty walk.

  Then,

$$c'.p.tlb = \delta_{tlb}(p.tlb, (\textbf{add-walk}, wext(w, pte))\}$$

  Walk extension only affects the TLB, note, however, that in order to perform walk extension, the corresponding page-table entry is read from memory.

- $in = (\textbf{sb})$. A memory write exits the store buffer.

  c' is defined iff $p.sb \neq []$. The store buffer can only make a step when it is not empty. Then,

$$c'.p.sb = tl(p.sb)$$
$$c'.m = \delta_m(c.m, p.sb[0])$$

  Store buffer steps never change processor core configurations and TLB configurations. The oldest write in the store buffer is submitted to the memory. Note that here the self-modifying code is forbidden.

Definition 3.30 (SB Reduced Sequential MIPS-86 Transition Function) The definition of SB reduced sequential MIPS-86 transition function is very similar to Definition 3.29. There are two differences: (i) instead of buffering store operations in the SB, all memory updates directly apply to the memory and (ii) instead of SB forwarding, the load operation directly read from the memory.

$$\delta_{sbr\text{-}seq} : K_{sbr\text{-}seq} \times \Sigma_{sbr\text{-}seq} \rightharpoonup K_{sbr\text{-}seq}$$

$$\delta_{sbr\text{-}seq}(c_{sbr}, in_{sbr}) = c'_{sbr}$$

We make a case split on in_{sbr}.

- $in_{sbr} = (\textbf{core}, w_I, w_R, eev)$. In this case we define all the shorthands analogously as in Definition 3.29 by substitute the sequential MIPS machine configuration c with c_{sbr} except the read value R. The read value is defined as:

$$R = \begin{cases} \bot & pff \vee pfls \\ c_{sbr}.m(pmaEA) & otherwise \end{cases}$$

  Under the same guard conditions as the first 4 in the first case of Definition 3.29, we can define c'_{sbr} as:

$$c'_{sbr}.p_{sbr}.core = \begin{cases} \delta_{core}(p_{sbr}.core, I, R, eev, pff, pfls) & (load(I) \vee rmw(I)) \\ \delta_{core}(p_{sbr}.core, I, \bot, eev, pff, pfls) & (\neg load(I) \wedge \neg rmw(I)) \\ p_{sbr}.core & otherwise \end{cases}$$

$$c'_{sbr}.p_{sbr}.tlb = \begin{cases} \emptyset & flush(I) \\ \delta_{tlb}(p_{sbr}.tlb, (\textbf{flush}, p_{sbr}.gpr(rd(I)).ba)) & invlpg(I) \\ \delta_{tlb}(p_{sbr}.tlb, (\textbf{flush}, p_{sbr}.pc.ba)) & pff \\ \delta_{tlb}(p_{sbr}.tlb, (\textbf{flush}, ea(p_{sbr}.core, I).ba)) & \neg pff \wedge pfls \\ p_{sbr}.tlb & otherwise \end{cases}$$

$$c'_{sbr}.m = \begin{cases} \delta_m(c_{sbr}.m, (p_{sbr}.gpr(rd(I)), pmaEA, sv(p_{sbr}.core, I))) & rmw(I) \\ \delta_m(c_{sbr}.m, (pmaEA, sv(p_{sbr}.core, I), bw(p_{sbr}.core, I))) & store(I) \\ c_{sbr}.m & otherwise \end{cases}$$

- In the remaining cases, the definition is completely analogous to Definition 3.29.

3.1.6 Multicore MIPS-86

Definition 3.31 (Multicore MIPS-86 Machine Configuration) The multicore MIPS-86 machine consists of np identical processors and a shared memory

$$K_{MIPS} = ([0 : np - 1] \to K_{pro}) \times K_m$$

$h = (c_{mul}, m) \in K_{MIPS}$ in which c_{mul} is a map from the processor index to the processor configuration.

Definition 3.32 (Multicore MIPS-86 Transition Function)

$$\delta_h : K_{MIPS} \times \Sigma_{MIPS} \rightharpoonup K_{MIPS}$$

In which $\Sigma_{MIPS} = \Sigma_{seq} \times [0 : np - 1]$.

$$\delta_h(h, (in, i)) = h'$$

where:

$$h'.c_{mul}(j) = \begin{cases} \delta_{seq}(h.c_{mul}(i), in).p & i = j \\ h.c_{mul}(j) & otherwise \end{cases}$$
$$h'.m = \delta_{seq}(h.c_{mul}(i), in).m$$

Definition 3.33 (Multicore SB Reduced MIPS-86 Machine Configuration) The multicore SB reduced MIPS-86 machine consists of np identical SB reduced processors and a shared memory

$$K_{sbr\text{-}MIPS} = ([0 : np - 1] \to K_{sbr\text{-}pro}) \times K_m$$

$h_{sbr} = (c_{sbr\text{-}mul}, m)$ in which $c_{sbr\text{-}mul}$ is a map from the processor index to the SB reduced processor configuration.

Definition 3.34 (Multicore SB Reduced MIPS-86 Transition Function)

$$\delta_{h_{sbr}} : K_{sbr\text{-}MIPS} \times \Sigma_{sbr\text{-}MIPS} \rightharpoonup K_{MIPS}$$

In which $\Sigma_{sbr\text{-}MIPS} = \Sigma_{sbr\text{-}seq} \times [0 : np - 1]$.

$$\delta_{h_{sbr}}(h_{sbr}, (in_{sbr}, i)) = h'_{sbr}$$

where:

$$h'_{sbr}.c_{sbr\text{-}mul}(j) = \begin{cases} \delta_{sbr\text{-}seq}(h_{sbr}.c_{sbr\text{-}mul}(i), in_{sbr}).p & i = j \\ h_{sbr}.c_{sbr\text{-}mul}(j) & otherwise \end{cases}$$
$$h'_{sbr}.m = \delta_{sbr\text{-}seq}(h_{sbr}.c_{sbr\text{-}mul}(i), in_{sbr}).m$$

3.2 Instantiation

In this section, we will instantiate our model from Chapter 2. First, we assume the program code resides in the read-only memory, and we will give the formal definition of the assumption at the end of this chapter. To distinguish between the fetch and execution phase, we add a *fetch* flag to the program state. According to the assumptions in Chapter 2, we should maintain the freshness of the temporaries. Thus, we also add a counter n to the program state as the time stamp. Moreover, for the initial configuration c^0, we constraint that

$$\forall i.\ c^0.p_{[i]}.fetch \wedge c^0.p_{[i]}.n = 0 \wedge c^0.is_{[i]} = [] \wedge \forall t \in \mathbb{T}.\ c^0.\vartheta_{[i]}(t) = \bot$$

Note that, in our model in Chapter 2, we make the *mode* as a separate component in the thread-local machine configuration and the page table origin as an implicit part in the MMU state. As a

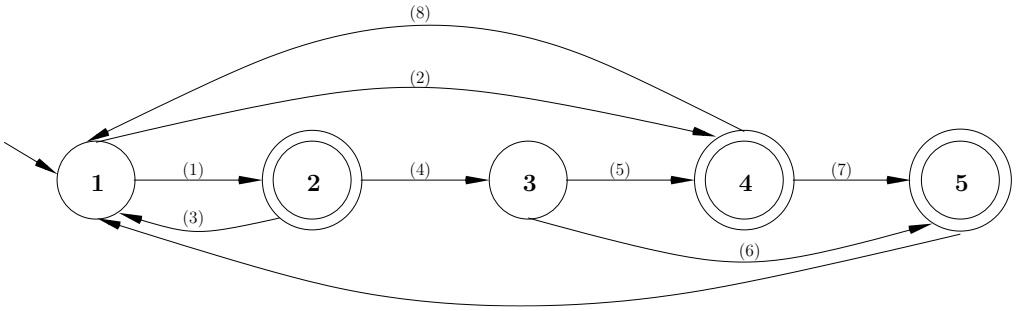


Figure 3.1: The state transitions in an instantated machine

consequence, we need a partial special-purpose-register file that does not contain *mode* and *pto*. We call it spr_p.

Also, note that for a non-page-fault interrupt must be signaled in a program step. However, a page fault interrupt must be signaled in a page fault step. The semantics of *jisr* in a program step and a page fault step are also different. In a page fault step, the *pc*, spr_p and *mode* are updated automatically when an interrupt happens. Nevertheless, in an interrupted program step, only the *pc* and spr_p are updated, and a mode switch memory instruction is generated to update the *mode*.

We need to instantiate the program step like that to maintain our proof in Chapter 2. In the proof, the abstract machine and the SB machine always perform the identical MMU steps simultaneously. To accomplish this, the value of *mode* should always be consistent in both machines. Since the program step does not flush the SB, we can not guarantee the simultaneous execution of a program step in both machines. As a consequence, our semantics in Chapter 2 forbid to change *mode* by program steps. Therefore, in an interrupted program step, a mode switch instruction needs to be generated, which is executed simultaneously on both machines, to update the *mode*.

As depicted in Figure 3.1, the execution of one instruction in the MIPS-86 machine can be divided into the following phases: Initially, we have

$$c.p_{[i]}.fetch$$

1. Program Step. In this step, the machine clears the *fetch* flag and
   - If no interrupts happen, the machine generates a non-vol read (because the code resides in the read-only portion of memory) to fetch from memory and makes the transition (1). The next step of thread i will be a phase 2 memory step.
   - If an interrupt happens, the machine updates *pc* and spr_p, and generate a mode switch memory instruction (transition (2)). The next step of thread i will be a phase 4 memory step. Note that, since the machine has not fetched from memory yet, the only possible interrupts can happen here are non-page-fault interrupts on fetch.

   After the program step we get a new machine configuration c'.

$$\neg c'.p_{[i]}.fetch$$

2. Memory Step or Page Fault Step. We make a further case split:
   - Page Fault Step. If a page fault happens here, the machine performs a page fault step. Note that, for the same reason as the previous phase, it should be a page fault on fetch. The machine updates *pc*, spr_p and *mode*, and sets the *fetch* flag to start the next round of execution (transition (3)). The next step of thread i will be a phase 1 program step. After the phase 2 we have
   $$c''.p_{[i]}.fetch$$
   - Memory Step. If no page fault happen, we let $I' = hd(c'.is_{[i]})$ then $nvR(I')$ should be generated by an uninterrupted phase 1 program step. With the non-vol read memory instruction, the machine fetches an ISA instruction by making the transition (4). In the next step of thread i, it will perform a phase 3 program step. After the phase 2 we have
   $$\neg c''.p_{[i]}.fetch$$
3. Program Step. In this step, to get the new configuration c''', the machine sets the *fetch* flag. We make a further case split here:
   - No interrupt happens. This case consists of two sub-cases depending on the fetched ISA instruction from a previous non-page-fault phase 2 execution:
     - No need to generate a memory instruction (e.g. if an add instruction was fetched in the last phase 2, the program step generates no instruction). The machine executes the fetched ISA instruction and makes the transition (6) as well as updates the corresponding *pc*, spr_p and *gpr*. The next step of thread i will be a phase 5 program step.
     - Otherwise. The machine generates a memory instruction to access the memory or update other components. The machine also updates the *pc* with the next *pc* value and also updates the spr_p. Note that, in this case, the machine does not update the *gpr*. The reason is that *gpr* should be updated with the read value from memory in this case, but the corresponding memory read operation has not been executed yet. The *gpr* will be updated in subsequent steps. The machine makes the transition (5). The next step of thread i will be a phase 4 memory step.
   - A non-page-fault interrupt happens. Based on the previous argument, interrupts on fetch were signaled in the previous phase 1 program step or phase 2 page fault step. The only possible interrupt here is a non-page-fault interrupt on execute. In this case, the machine updates the *pc*, spr_p and *gpr* (for a continue interrupt). The machine generates a mode switch memory instruction to update *mode* and makes the transition (5). The next step of thread i will be a phase 4 memory step.

   After phase 3 we have
   $$c'''.p_{[i]}.fetch$$
4. Memory Step or Page Fault Step.

- If no page fault happens, the machine performs a memory step. Here, we make a further case split:
  - If the instruction in the instruction sequence is a mode switch instruction generated by an interrupted program step, then the machine makes the transition (8). The next step of thread i will be a phase 1 program step.
  - Otherwise, the machine makes the transition (7). The next step of thread i will be a phase 5 program step.
- If a page fault happens, with analogous reasons as in phase 3, the page fault is a page fault on load/store. It updates the pc, spr_p and $mode$, and sets the $fetch$ flag (transition (8)). The machine will perform a phase 1 program step in next step of thread i.

$$c^4.p_{[i]}.fetch$$

5. Program Step. This step increases the counter in program state and updates the corresponding gpr if the last phase 4 memory step is a read or rmw. The next step of thread i will be a phase 1 program step. Note that this phase is only entered for non-interrupted instruction execution.

$$c^5.p_{[i]}.fetch$$

Note that since in one round of abstract machine execution, every phase corresponds to one identical ISA instruction. For each MIPS step, there is only one eev. Therefore, in one round of execution, every program step have identical eev. Also, note that in the phase 5 program step, the interrupts are ignored. For external interrupts which have the highest priority should be handled in the previous program step. For internal interrupts can be handled in the subsequent the program step.

3.2.1 Instantiation of Basic Signatures

- $\mathbb{A}, \mathbb{V}$. The memory is a word-addressable memory with 2^{30} addresses.

$$\mathbb{A} = \mathbb{B}^{30} \qquad \mathbb{V} = \mathbb{B}^{32}$$

- $\mathbb{P}$. The program state is defined as a tuple which contains a counter (or time stamp) $n \in \mathbb{N}$ to maintain the uniqueness of temporaries, a program counter $pc \in \mathbb{B}^{32}$, a previous program counter $ppc \in \mathbb{B}^{32}$ which will be useful in later proofs, a general purpose register file $gpr \in \mathbb{B}^5 \to \mathbb{B}^{32}$, a partial special purpose register file $spr_p \in \mathbb{B}^5 \setminus \{bin_5(5), bin_5(7)\} \to \mathbb{B}^{32}$ which is a special purpose register file without $mode$ and pto, and a flag $fetch \in \mathbb{B}$. Moreover, it contains an auxiliary component $jisr \in \mathbb{B}$ which is useful in the simulation proof in Section 4.3.2. The $jisr$ flag is set when an interrupt happens.

The set of program states is defined as:

$$\mathbb{P} = \mathbb{N} \times \mathbb{B}^{32} \times \mathbb{B}^{32} \times (\mathbb{B}^5 \to \mathbb{B}^{32}) \times (\mathbb{B}^5 \setminus \{bin_5(5), bin_5(7)\} \to \mathbb{B}^{32}) \times \mathbb{B} \times \mathbb{B}$$

For all $p \in \mathbb{P}$. $p = (p.n, p.pc, p.ppc, p.gpr, p.spr_p, p.fetch, p.jisr)$. We define partial special purpose register file as: $spr_p = spr \restriction_{\mathbb{B}^5 \setminus \{bin_5(5), bin_5(7)\}}$. Note that, the initial value of ppc is equal to pc.

- $\mathbb{T}$. The temporary is instantiated as a tuple which contains a name $\epsilon\{I, R\}$ and a counter (or time stamp) $n \in \mathbb{N}$ to make the temporary unique. In the instantiated semantics (section 3.2.3) of the program step, the value of the counter is always increased. Thus each temporary is unique. In the rest of this thesis we write the temporary (I, n) and (R, n) as I_n and R_n for short. The set of temporaries is defined as:

$$\mathbb{T} = \{I, R\} \times \mathbb{N}$$

- $\mathbb{R}$. The set of access rights for address translation is defined as a 3-bit string. For all $r \in \mathbb{R}$ the $r[0]$ stands for write permission, $r[1]$ for user mode access and $r[2]$ for execute permission.

$$\mathbb{R} = \mathbb{B}^3$$

- $\mathbb{BW}$. The set of byte write signals is defined as a subset of $\mathbb{B}^4$.

$$\mathbb{BW} = \{0000, 0001, 0010, 0100, 1000, 0011, 1100, 1111\}$$

- $\mathbb{U}$. The MMU state consists of a TLB $tlb \in 2^{K_{walk}}$ and a value of page table origin $pto \in \mathbb{B}^{32}$.

$$\mathbb{U} = 2^{K_{walk}} \times \mathbb{B}^{32}$$

- $\mathbb{EEV}$. The external input is defined as a 256 bit string. For all $eev \in \mathbb{EEV}$ the component $eev[0]$ is the reset signal, and $eev[1:255]$ is the device interrupt triggered by signals from the external environment of the processor.

$$\mathbb{EEV} = \mathbb{B}^{256}$$

3.2.2 Instantiation of Auxiliary Functions, Predicates, and Relations

Casting Functions In order to reuse the auxiliary functions defined in section 3.1.1 we need the following type cast functions: The function $cast(p, mode, pto) \in K_{core}$ takes a program state $p \in \mathbb{P}$, $mode, pto \in \mathbb{B}^{32}$ and returns a MIPS-86 processor core configuration.

$$cast(p, mode, pto) = c[gpr := p.gpr, pc := p.pc, spr := spr']$$

where:

$$spr' = p.gpr_p(bin_5(5) \mapsto pto)(bin_5(7) \mapsto mode)$$

Some auxiliary functions do not depend on the value of *mode* or *pto*. Thus, we overload the typecast function:

$$cast(p, mode) = cast(p, mode, 0^{32})$$
$$cast(p) = cast(p, 0^{32}, 0^{32})$$

In the remaining part of this chapter, we let $I = \vartheta(I_{p.n})$ and $R = \vartheta(R_{p.n})$ then the auxiliary functions and predicates are defined as follows.

- f. The write value calculation function $f \in (\mathbb{T} \rightharpoonup \mathbb{V}) \to \mathbb{V}$ is defined as:

$$f(\vartheta) = \begin{cases} s4s(p.gpr(rt(I)), I) & store(I) \\ p.gpr(rd(I)) & rmw(I) \end{cases}$$

- D is instantiated as:

$$D = \{I_{p.n}\}$$

- *cond*. The condition predicate $cond \in (\mathbb{T} \rightharpoonup \mathbb{V}) \to \mathbb{B}$ for read modify write is instantiated as:

$$cond(\vartheta) \equiv p.gpr(rd(I)) = R$$

- *cb*. The combination function $cb \in \mathbb{V} \times \mathbb{V} \times \mathbb{BW} \to \mathbb{V}$ for a write operation is used to compute the value to be stored in memory according to the byte write signal. This function is defined identically as the combination function in section 3.1.2.

- $\leq$. The relation $\leq \in \mathbb{BW} \times \mathbb{BW} \to \mathbb{B}$ is defined similarly as the overloaded $\leq$ relation in section 3.1.4.

$$bw_1 \leq bw_2 \equiv \forall i.\ bw_1[i] \leq bw_2[i]$$

- $=_{bw}$. The byte writes equality relation $=_{bw} \in \mathbb{V} \times \mathbb{V} \to \mathbb{B}$ is used to check if 2 data is equal according to a given byte write signal *bw*.

$$v_1 =_{bw} v_2 \equiv bw[i] \to byte(i, v_1) = byte(i, v_2)$$

3.2.3 Instantiation of Transition Functions

MMU Model

- $can\text{-}access(mmu, pa) \in \mathbb{B}$. The predicate denotes weather the MMU in state $mmu \in \mathbb{U}$ can access a page table entry located at $pa \in \mathbb{A}$.

$$can\text{-}access(mmu, pa) \equiv \exists w \in mmu.tlb.\ ptea(w) = pa \wedge \neg complete(w)$$

- $can\text{-}page\text{-}fault(mmu, va, r, pa, pte) \in \mathbb{B}$. The predicate denotes weather the MMU in state $mmu \in \mathbb{U}$ can signal a page fault during the virtual address $va \in \mathbb{A}$ translation. The page fault can be signaled if we can choose a walk w from the $mmu.tlb$ and obtain a faulty walk by extending w with $pte \in \mathbb{V}$ in address $pa \in \mathbb{A}$ or the access rights $r \in \mathbb{R}$ is violated by r. Let $trq = (va, r)$ then

$$can\text{-}page\text{-}fault(mmu, va, r, pa, pte) \equiv \exists w \in mmu.tlb.\ fault(pte, trq, w)$$

- $\delta_{mmur}(mmu, pa, pte) \in 2^{\mathbb{U}}$. The MMU read function fetches a page table entry $pte \in \mathbb{V}$ from the physical address $pa \in \mathbb{A}$ and returns a set of possible MMU states according to the MMU state $mmu \in \mathbb{U}$. We first define the safety condition for walk extensions.

$$\begin{aligned} safe\text{-}wext(w, pte) \equiv & \neg complete(w) \wedge pte.a \wedge pte.p \wedge \\ & (w.level = 1 \wedge w.r[0] \wedge pte.r[0] \rightarrow pte.d) \end{aligned}$$

  Let $\delta_{mmur}(mmu, pa, pte) = A$ then A is defined as:

$$\begin{aligned} A = \{(mmu.pto,\ & mmu.tlb \cup \{wext(w, pte)\}) \mid \\ & w \in mmu.tlb \wedge ptea(w) = pa \wedge safe\text{-}wext(w, pte)\} \end{aligned}$$

- $\delta_{mmuw}(mmu, pa, pte) \in 2^{\mathbb{V}}$. The MMU write function takes a page table entry $pte \in \mathbb{V}$ located at physical address $pa \in \mathbb{A}$ and returns a set of values. The result of the MMU write function is a set of values because whether we should update the dirty bit of pte depends on the walk non-deterministically chosen from the $mmu.tlb$.

$$\begin{aligned} \delta_{mmuw}(mmu, pa, pte) = \{set\text{-}ad(w, pte) \mid & w \in mmu.tlb \wedge \\ & pte.p \wedge ptea(w) = pa \wedge \neg complete(w)\} \end{aligned}$$

- $\delta_{flush}(mmu, F) \in \mathbb{U}$. The TLB flush function takes the MMU state $mmu \in \mathbb{U}$ and a set of addresses $F \in 2^{\mathbb{A}}$ and removes certain walks from the TLB. The Walk $w \in mmu.tlb$ is removed from TLB if there exists $a \in F$ and $a.ba = w.va$.

$$\delta_{flush}(mmu, F) = mmu'$$

  where:

$$mmu' = (mmu.pto, mmu.tlb \setminus \{w \mid \exists a \in F.\ a.ba = w.va\})$$

- $\delta_{wpto}(mmu, v) \in \mathbb{U}$. The page table origin update function takes the MMU state $mmu \in \mathbb{U}$ and a value $v \in \mathbb{B}^{32}$ returns an updated MMU state.

$$\delta_{wpto}(mmu, v) = mmu[pto := v, tlb := mmu.tlb \cup \{winit(va, v) \mid va \in \mathbb{B}^{20}\}]$$

- $\delta_{crtw}(mmu, va) \in \mathbb{U}$. The walk creation function creates a new walk for address va and add it to the TLB.

$$\delta_{crtw}(mmu, va) = mmu[tlb := tlb']$$

in which

$$tlb' = mmu.tlb \cup \{winit(va.ba, mmu.pto[31:2].ba)\}$$

- $\delta_{pf}(p, mode, va) \in \mathbb{P}$. The page fault function jumps to the interrupt service routine when a page fault happens. It takes a program state $p \in \mathbb{P}$, a mode flag $mode$ and a virtual effective address va, and returns an updated program state.

$$\delta_{pf}(p, mode, va) = p'$$

where we let

$$mca_{pff} = 0^3 10^{28} \qquad mca_{pfls} = 0^8 10^{23}$$

then

$$p'.pc = 0^{32} \quad p'.ppc = p.pc \quad p'.n = p.n + 1$$
$$p'.fetch = 1 \quad p'.gpr = p.gpr \quad p'.jisr = 1$$

$$p'.spr_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ mode & x = emode \\ p.spr_p(sr) & x = esr \\ mca_{pff} & x = eca \wedge \neg p.fetch \\ mca_{pfls} & x = eca \wedge p.fetch \\ p.ppc & x = epc \wedge p.fetch \\ p.pc & x = epc \wedge \neg p.fetch \\ va \circ 00 & x = edata \wedge p.fetch \\ p.spr_p(x) & otherwise \end{cases}$$

Note that, according to the argument at the beginning of this section, the page fault can only happen in a phase 2 or phase 4 page fault step. In order to enter the phase 2 page fault step, our semantics guarantee that no other interrupts on fetch in previous phase 1 program step happened (Otherwise, the machine will enter phase 4 to execute the mode switch instruction). Thus, the masked cause for page fault on fetch can be set to $0^3 10^{28}$. Analogously, the masked cause for page fault on load/store can be set to $0^8 10^{23}$.

Also note that, if a page fault happens in phase 2, from the semantics we know that the *fetch* flag was cleared by the previous phase 1 operation. However, since the machine has not performed the corresponding fetch yet, the page fault is a page fault on fetch interrupt. Similarly, a true *fetch* flag indicates that the page fault is a page fault on load/store interrupt.

Moreover, in page fault step, the current pc value should be store in the special purpose register epc. For a phase 4 page fault step, because the next pc value was computed in the previous phase 3 program step, the $p.pc$ value is actually next pc value. Thus, we store the ppc into epc in the phase 4 page fault step.

- $atran(mmu, va, mode, r) \in 2^{\mathbb{A}}$. The address translation function takes an MMU state $mmu \in \mathbb{U}$, a virtual address to be translated $va \in \mathbb{A}$, a $mode \in \mathbb{B}$ denotes whether we are in system mode or user mode, and a $r \in \mathbb{R}$ for the access rights of the translation request. It returns a set of possible translated physical addresses.

$$atran(mmu, va, mode, r) = \begin{cases} \{ba \circ 0^{10} +_{30} 0^{20} \circ va.px_0 \mid ba \in PBA\} & mode = 1 \\ \{va\} & mode = 0 \end{cases}$$

  where:

$$PBA = \{w.ba \mid w \in mmu.tlb \wedge complete(w) \wedge hit((va, r), w)\}$$

Transition Function δ_p in Program Step

The transition function δ_p is used to generate instructions and update the program state. In order to generate the correct instructions, the shared memory access instructions should be distinguished. We can collect the virtual address of these instructions in a set A_{io} which only depends on the compiler. At the ISA level, we treat it as an external parameter from the environment.

Auxiliary Functions

$$ca_f(p, eev) \equiv ca_f(cast(p), eev, 0)$$
$$ca_x(p, mode, I) \equiv ca_x(cast(p, zxt_{32}(mode)), I, 0)$$

Note that the page fault flags are set to 0, because the page faults should be handled later in page fault steps. It is fine since page faults have the lowest priority. We overload the mca_f, mca_x, il_f, il_x, $jisr_f$ and $jisr_x$ as:

$$mca_f(p, eev)[j] \equiv \begin{cases} ca_f(p, eev)[j] \wedge p.spr_p(sr)[j] & j = 1 \\ ca_f(p, eev)[j] & otherwise \end{cases}$$
$$mca_x(p, mode, I)[j] \equiv \begin{cases} ca_x(p, mode, I)[j] \wedge p.spr_p(sr)[j] & j = 6 \\ ca_x(p, mode, I)[j] & otherwise \end{cases}$$

$$il_f(p, eev) = min\{j \mid mca_f(p, eev)[j] = 1\}$$
$$il_x(p, mode, I) = min\{j \mid mca_x(p, mode, I)[j] = 1\}$$

$$continue(p, mode, I) \equiv il_x(p, mode, I) \in \{5, 6\}$$

$$jisr_f(p, eev) \equiv \bigvee_j mca_f(p, eev)[j] \wedge p.fetch$$
$$jisr_x(p, mode, I) \equiv \bigvee_j mca_x(p, mode, I)[j] \wedge \neg p.fetch$$

We define the $jisr$ predicate as follows:

$$jisr(p, mode, \vartheta, eev) \equiv jisr_f(p, eev) \vee jisr_x(p, mode, I)$$

We overload the transition function δ_{jisr_f} as follows:

$$\delta_{jisr_f}(p, mode, \vartheta, eev) = p'$$

in which we let $c' = \delta_{jisr_f}(cast(p, zxt_{32}(mode)), eev, 0)$ then:

$p'.pc = c'.pc$ $\quad p'.ppc = p.pc$ $\quad p'.spr_p = c'.spr_p$ $\quad p'.n = p.n + 1$
$p'.jisr = 1$ $\quad p'.gpr = c'.gpr$ $\quad p'.fetch = 1$

We overload the transition function δ_{jisr_x} as follows:

$$\delta_{jisr_x}(p, mode, \vartheta) = p'$$

in which we let

$$c' = \delta_{jisr_x}(cast(p, zxt_{32}(mode)), I, 0))$$

then:

$p'.pc = c'.pc$ $\quad p'.ppc = p.pc$ $\quad p'.spr_p = c'.spr_p$ $\quad p'.n = p.n + 1$
$p'.gpr = c'.gpr$ $\quad p'.fetch = 1$ $\quad p'.jisr = 1$

We define the following predicate to check whether the machine is in the phase 1 program step.

$$fetch(p, \vartheta, eev) \equiv p.fetch \wedge \neg jisr_f(p, eev) \wedge I = \bot$$

We define the following predicate to check whether the machine is in the phase 3 program step.

$$execute(p, \vartheta, mode) \equiv \neg p.fetch \wedge \neg jisr_x(p, mode, I)$$

We define the following predicate to check whether the machine is in the phase 5 program step.

$$post(p, \vartheta) \equiv p.fetch \wedge I \neq \bot$$

Note that, according to our previous description, the phase 5 program step behaves as an epilogue of an instruction execution, which increases the counter and updates the *gpr* if needed. Thus, this step ignores all the interrupts.

Instruction Generation Function

Definition 3.35 (Instruction Generation) The instruction generation function

$$ins\text{-}gen(p, \vartheta, mode) \in \mathbb{I}^*$$

generates a sequence of instructions according to the value of temporary:

$$ins\text{-}gen(p, \vartheta, mode) = l$$

$$l = \begin{cases} [\textbf{Read}\ vol\ ea(cast(p), I)[31:2]\ R_{p.n}\ 010\ ext\ bw\ p] & load(I) \\ [\textbf{Write}\ vol\ ea(cast(p), I)[31:2]\ (D, f)\ 110\ cb\ bw\ p] & store(I) \\ [\textbf{RMW}\ ea(cast(p), I)[31:2]\ R_{p.n}\ (D, f)\ cond\ 110\ p] & rmw(I) \\ [\textbf{INVLPG}\ F] & invlpg(I) \vee flush(I) \\ [\textbf{SWITCH}\ p.spr_p(emode)[0]] & eret(I) \\ [\textbf{SWITCH}\ p.gpr(rt(I))[0]] & switch(I) \\ [\textbf{WPTO}\ p.gpr(rt(I))] & wpto(I) \\ [\textbf{FENCE}] & mfence(I) \\ [] & otherwise \end{cases}$$

where:

$$vol \leftrightarrow p.pc[31:2] \in A_{io} \qquad F = \begin{cases} p.gpr(rd(I))[31:2] & invlpg(I) \\ \mathbb{B}^{30} & flush(I) \end{cases}$$

$$bw = bw(cast(p), I) \qquad ext(data, bw) = lv(data, I)$$

With the definitions of $ext, cb, \leq$ and $=_{bw}$ the constraints in Section 2.2.2 on bw, ext and cb can be trivially discharged.

Definition of δ_p

Definition 3.36 (Transition Function in Program Step) The transition function

$$\delta_p(p, \vartheta, mode, mmu, is, eev) = (p', is')$$

takes a program state $p \in \mathbb{P}$, temporaries $\vartheta \in \mathbb{T} \rightharpoonup \mathbb{V} \times \mathbb{A}$, a $mode \in \mathbb{B}$, an instruction sequence $is \in \mathbb{I}^*$ and an external input eev and returns an updated program state $p' \in \mathbb{P}$ as well as a sequence of newly generated instructions $is' \in \mathbb{I}^*$. (p', is') is defined iff $is = []$. We let $c' = \delta_{insr}(cast(p, zxt_{32}(mode), mmu.pto), I, 0^{32})$. Note that we use the dummy value 0^{32} to execute the instruction I and get a MIPS core configuration c'. We only use the c' to update the pc, spr_p and gpr for an uninterrupted phase 3 program step if it is no need to access the memory, the new

value computation of these components do not depend on the read results. The updating of gpr with the memory read result is postponed to the phase 5 program step.

$$p' = \begin{cases} p[fetch := 0, jisr := 0] & fetch(p, \vartheta, eev) \\ p_{exec} & execute(p, \vartheta, mode) \\ p_{post} & post(p, \vartheta) \\ \delta_{jisr_f}(p, mode, \vartheta, eev) & jisr_f(p, eev) \wedge I = \bot \\ \delta_{jisr_x}(p, mode, \vartheta) & jisr_x(p, mode, I, eev) \end{cases}$$

$$is' = \begin{cases} [\textbf{Read}\ False\ p.pc[31:2]\ I_{p.n}\ 011\ ext\ 1111\ p] & fetch(p, \vartheta, eev) \\ ins\text{-}gen(p, \vartheta, mode) & execute(p, \vartheta, mode) \\ [] & post(p, \vartheta) \\ [\textbf{SWITCH}\ 0] & otherwise \end{cases}$$

where:
$p_{exec}.pc = c'.pc \quad p_{exec}.spr_p = c'.spr_p \quad p_{exec}.ppc = p.pc \quad p_{exec}.fetch = 1 \quad p_{exec}.n = p.n$
$p_{exec}.jisr = 0$

$$p_{exec}.gpr = \begin{cases} p.gpr & is' \neq [] \\ c'.gpr & otherwise \end{cases}$$

and

$$updategpr(I, x) \equiv (load(I) \wedge rt(I) = x) \vee (rmw(I) \wedge rd(I) = x)$$

$$p_{post}.gpr(x) = \begin{cases} R & updategpr(I, x) \\ p.gpr(x) & otherwise \end{cases} \qquad p_{post}.n = p.n + 1$$

$$\forall X \in \{pc, ppc, spr_p, fetch, jisr\}.\ p_{post}.X = p.X$$

4 Applying Store Buffer Reduction to MIPS-86

In Chapter 3, we instantiate the SB reduction theorem with MMU at the ISA level. In this chapter, we will apply the SB reduction theorem with MMU to MIPS-86 ISA. Thus, we need the model stack in Figure 4.1. The bottom of the stack is the MIPS-86 ISA, which can be trivially simulated by the instantiated SB machine. We omit the trivial simulation theorem in this thesis. The SB machine can be simulated by the abstract machine with the SB reduction theorem in Chapter 2. After that, we need a theorem to simulate the abstract machine with the SB reduced MIPS-86 ISA.

First, we need to define the semantics of SB reduced MIPS-86 ISA with ownership. We introduce a model named Concurrent system with shared memory and ownership (*Cosmos* model) from [Bau14] as the top layer of the model stack in Figure 4.1. To define the safety condition for the MMU access, we extend the ownership in [Bau14] with local page table sets for each threads and the corresponding acquire and release sets of local page tables. Then we instantiate the *Cosmos* model with SB reduced MIPS-86 ISA. Moreover, we will prove a simulation theorem between the instantiated abstract machine and the SB reduced MIPS-86 *Cosmos* machine. At last, we will prove that the ownership and safety is correctly transfered from SB reduced MIPS-86 *Cosmos* machine to the abstract machine.

4.1 *Cosmos* Model

The *Cosmos* model is a generic model for machines that are concurrently accessing a shared memory. The memory accesses of *Cosmos* model is governed by an ownership policy which is a simplified version of the ownership in Chapter 2. The only difference is that, instead of a dynamic read-only set in Chapter 2, a static set for read-only portion of memory is considered. The read-only memory contains the region where the machine code of the system program resides. Therefore, the intuition behind the static read-only set is that we only consider an unswappable code region in our system. In this section, we (i) extend the ownership model with the local page tables (ii) add the acquire and release sets for local page table (iii) extend the safety condition for the new ownership model (iv) reprove Lemma 4.13 and Lemma 4.18. The remaining portion of this section is copied form [Bau14].

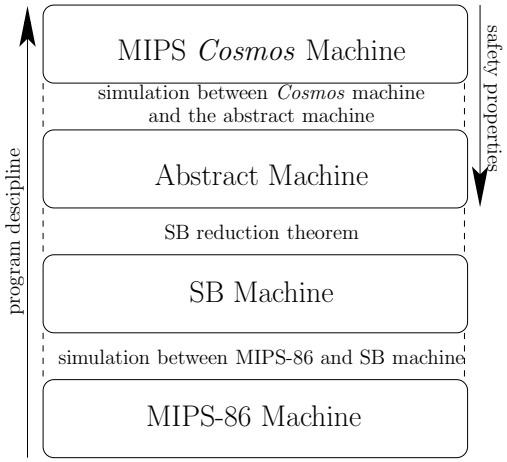


Figure 4.1: Concurrent model stack

4.1.1 Signatures and Instantiation Parameters

We define the *Cosmos* model by introducing a *Cosmos* machine which is a concurrent system of abstract automata operating on a shared memory. We call the different automata *computation units*, or short *units*. They can be instantiated by, e.g., processors, devices, or the semantics of a higher level program. In this work, however, we assume for simplicity that all units are instantiated with the same kind of automaton. Units are only communicating via a shared memory. However we have external input signals to allow for the treatment of external communication and non-determinism.

Definition 4.1 (*Cosmos* model Machine Signature) A *Cosmos* machine S is given by a tuple

$$S = (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP}) \in \mathbb{S}$$

with the following components:

- $\mathcal{A}, \mathcal{V}$ - set of memory addresses and set of memory values, any function $m : \mathcal{A} \to \mathcal{V}$ is called a *memory*, any partial function $m : \mathcal{A} \rightharpoonup \mathcal{V}$ is a *partial memory*.
- $\mathcal{R} \subseteq \mathcal{A}$ - set of read-only addresses (part of the ownership state)
- nu - the number of computation units in the machine
- $\mathcal{U}$ - set of computation unit states
- $\mathcal{E}$ - set of external inputs for the units
- $reads : \mathcal{U} \times (\mathcal{A} \to \mathcal{V}) \times \mathcal{E} \to 2^{\mathcal{A}}$ - the set of memory addresses read by the next step from the given unit configuration, global memory, and external input. This set is called the *reads*-set.

- $\delta : \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V}) \times \mathcal{E} \to \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V})$ - the transition function for the units; takes unit state, a partial memory, and external input; results in a new unit state as well as another partial memory. As the input partial memory, we will provide the shared memory being restricted to the *reads*-set of the step. The output partial memory represents the updated part of memory for the step.
- $\mathcal{IO} : \mathcal{U} \times (\mathcal{R} \to \mathcal{V}) \times \mathcal{E} \to \mathbb{B}$ - denotes whether the next step of the unit is an $\mathcal{IO}$ step. $\mathcal{IO}$ steps represent synchronized interactions with the environment (i.e., all other computation units). Consequently, they include (but are not limited to) all atomic accesses to concurrently accessed data structures in memory, e.g., locks and other synchronization mechanisms. Whether the next step of a unit is an $\mathcal{IO}$ step, may depend on memory but only on the read-only portion.
- $\mathcal{IP} : \mathcal{U} \times (\mathcal{R} \to \mathcal{V}) \times \mathcal{E} \to \mathbb{B}$ - specifies the desired interleaving-points for the units, i.e., states of the computation before which we allow steps of other units to be interleaved. Whether a unit is in an interleaving-point, may depend on memory but only on the read-only portion.

In order to give an intuition for the intended meaning of the components, we consider an instantiation with a 32-bit multiprocessor system running in untranslated mode. For a word-addressable memory, we set $\mathcal{A} = \mathbb{B}^{30}$ and $\mathcal{V} = \mathbb{B}^{32}$. The read-only set $\mathcal{R}$ contains the region where the machine code of the system program resides. The unit state $\mathcal{U}$ contains all processor core registers, and we use $\mathcal{E}$ to model external device interrupt signals. The reads-set always contains the address pointed to by the program counter (or instruction pointer, respectively). Moreover in case of a load instruction, the targeted addresses also contribute to the reads-set. The δ-function then encodes the semantics of the underlying instruction set architecture (ISA). The $\mathcal{IO}$ steps are defined as the shared memory accesses.

Note that in the Cohen-Schirmer theory and Chapter 2 $\mathcal{IO}$ memory instructions are denoted as *volatile* accesses. However to avoid confusion with the notion of volatile accesses on the C level we rename the concept here.

4.1.2 Configurations

Definition 4.2 (Machine State) The machine state M of a *Cosmos* machine S is a pair

$$M = (u, m) \in \mathbb{M}_S$$

where $u : [0 : nu - 1] \to \mathcal{U}$ maps unit indices to their unit states and $m : \mathcal{A} \to \mathcal{V}$ is the state of the memory.

Definition 4.3 (Ownership State) The ownership state $\mathcal{G}$ (ghost state) of a *Cosmos* machine S is a tuple

$$\mathcal{G} = (\mathcal{O}, \mathcal{P}t, \mathcal{S}) \in \mathbb{G}_S$$

where $\mathcal{O} : [0 : nu - 1] \to 2^{\mathcal{A}}$ maps unit indices to the corresponding units' sets of owned addresses (owns-set), $\mathcal{P}t : [0 : nu - 1] \to 2^{\mathcal{A}}$ maps unit indices to the corresponding units' sets of local page table addresses and $\mathcal{S} \subseteq \mathcal{A}$ is the set of shared writable addresses.

Now we can define the configuration of the overall *Cosmos* machine.

Definition 4.4 (*Cosmos* Machine Configuration) A configuration C of *Cosmos* model S is given as a pair

$$C = (M, \mathcal{G}) \in \mathbb{K}_S$$

consists of machine state $M \in \mathbb{M}_S$ and ownership state $\mathcal{G} \in \mathbb{G}_S$.

For $p \in [0 : nu - 1]$ and $unit \in \{core, mu\}$ we use the following shorthands:

$$\begin{aligned}
C.u_p &\equiv C.M.u(p) & C.m &\equiv C.M.m \\
C.\mathcal{O}_p &\equiv C.\mathcal{G}.\mathcal{O}(p) & C.\mathcal{S} &\equiv C.\mathcal{G}.\mathcal{S} \\
C.\mathcal{P}t_p &\equiv C.\mathcal{G}.\mathcal{P}t(p) & & \\
reads_p(C, in) &\equiv reads_p(C.M, in) &\equiv\ & reads(C.M.u(p), C.M.m, in) \\
\mathcal{IO}_p(C, in) &\equiv \mathcal{IO}_p(C.M, in) &\equiv\ & \mathcal{IO}(C.M.u(p), C.M.m|_{\mathcal{R}}, in) \\
\mathcal{IP}_p(C, in) &\equiv \mathcal{IP}_p(C.M, in) &\equiv\ & \mathcal{IP}(C.M.u(p), C.M.m|_{\mathcal{R}}, in)
\end{aligned}$$

Moreover, for defining the semantics, we need to know which addresses are written in a step of the *Cosmos* machine.

Definition 4.5 (Writes-set of a machine step) For a given *Cosmos* model S with configuration $C \in \mathbb{M}_S$ and an input $in \in \mathcal{E}$ we can determine the set of written addresses in the corresponding step of machine p from the result of the delta function. This so-called *writes*-set of machine p is obtained with the following function.

$$writes_p(C, in) = \mathrm{dom}(m') \qquad \text{where} \qquad (u', m') = \delta(C.u_p, C.m|_{reads_p(C,in)}, in)$$

Note that the writes-set only depends on the part of memory that is read in the step. If $reads_p(C, in) = \emptyset$ then $writes_p(C, in) = \emptyset$.

4.1.3 Restrictions on Instantiated Parameters

Not all parameters of a *Cosmos* model can be instantiated arbitrarily. In order to obtain a meaningful model, there is one constraint on the *reads*-set of *Cosmos* model computation units.

Definition 4.6 (Instantiation Restriction for *reads*) By the predicate $insta_r$ we require that the *reads*-set contains all addresses upon whose memory contents it depends. For any *Cosmos* machine S let $u \in \mathcal{U}$ be a computation unit state, $m, m' \in (\mathcal{A} \to \mathcal{V})$ shared memories, and $in \in \mathcal{E}$ be a suitable input for a step of the unit. If the memory contents agree on reads-set $Read = S.reads(u, m, in)$, then also the reads-set wrt. m' agrees with R.

$$insta_r(S) \equiv (m'|_{Read} = m|_{Read} \to S.reads(u, m', in) = Read)$$

This property is needed for instantiations that incorporate a series of read accesses in one unit step. There the first reads can influence which addresses are read in later parts of the step, as in the processor instantiation example above. The reads-set must thus include all relevant

addresses to determine which addresses are read. That means conversely that it only depends on the portion of memory that was read.

In order to be able to deduce that a machine performs the same step after reordering (by exploiting that the content of the memory region given by the *reads*-set is unchanged and thus also the same addresses are read), the property on the *reads*-set is crucial because same steps perform same update to the memory region given by the *writes*-set.

Thus, from now on, when we mention a *Cosmos* model S, we always assume that restriction $insta_r(S)$ holds.

4.1.4 Semantics

Units of the *Cosmos* machine execute according to their transition functions. A scheduling input decides which machine performs the next step. We assume ownership inputs that specify changes to the ownership state. These ownership inputs are given by the verification engineer annotating the program.

Definition 4.7 (*Cosmos* Model Transition Function) For a *Cosmos* machine S, we define transition function

$$\Delta : \mathbb{K}_S \times [0 : nu - 1] \times \mathcal{E} \times (2^{\mathcal{A}})^5 \to \mathbb{M}_S$$

which takes a configuration C, a scheduling input p, an external input $in \in \mathcal{E}$, the set A of acquired addresses, the set L of acquired local addresses (which should be a subset of A), the set R of released addresses, the set A_{pt} of acquired address for local page table and the set R_{pt} of released address from local page table to perform a step of unit p on its state, the common memory, and the ownership state. First, however, we consider the transition on the machine and ownership states separately.

With $(u', m') = \delta(M.u(p), M.m|_{reads(M.u(p),in)}, in)$ and $m_{unchanged} = M.m|_{\mathcal{A}\setminus \mathrm{dom}(m')}$ we define transition function

$$\Delta_t(M, p, in) = (M.u[p \mapsto u'], m'')$$

on the machine state. In which we have

$$m''(x) = \begin{cases} m'(x) & x \in dom(m') \\ m_{unchanged} & otherwise \end{cases}$$

Moreover with

$$\begin{aligned} \mathcal{O}' &= \mathcal{G}.\mathcal{O}_p \cup A \setminus R \\ \mathcal{P}t' &= \mathcal{G}.\mathcal{P}t_p \cup A_{pt} \setminus R_{pt} \\ \mathcal{S}' &= \mathcal{G}.\mathcal{S} \cup R \cup R_{pt} \setminus (L \cup A_{pt}) \end{aligned}$$

we define the ownership transfer function:

$$\Delta_o(\mathcal{G}, p, (A, L, R, A_{pt}, R_{pt})) \equiv (\mathcal{G}.\mathcal{O}[p \mapsto \mathcal{O}'], \mathcal{G}.\mathcal{O}[p \mapsto \mathcal{P}t'], \mathcal{S}')$$

Now the overall transition function for *Cosmos* machine configurations is defined by:

$$\Delta(C, p, in, (A, L, R, A_{pt}, R_{pt})) \equiv (\Delta_t(C.M, p, in), \Delta_o(C.\mathcal{G}, p, (A, L, R, A_{pt}, R_{pt}))$$

The scheduling parameter p determines which unit is going to perform a computation step according to transition function δ consuming external input in, updating the written part of memory accordingly. The ownership transfer inputs $(A, L, R, A_{pt}, R_{pt})$ are used to update the owned addresses and local page table of p and the set of shared-writable addresses.

4.1.5 Computations and Step Sequence Notation

In this section, we describe a computation not by the sequence of states it produces but by the executed sequence σ of steps from a certain alphabet. In our case, the alphabet contains transition information and ownership annotation defined as follows.

Definition 4.8 (Step Information) We define the set Σ_S of step information of a *Cosmos* machine S where

$$\alpha = (s, in, io, ip, A, L, R, A_{pt}, R_{pt}) \in \Sigma_S$$

describes a *Cosmos* machine step, containing the following transition information

- $\alpha.s \in [0 : nu - 1]$ - the scheduling parameter
- $\alpha.in \in \mathcal{E}$ - the external input for the step
- $\alpha.io \in \mathbb{B}$ - marks the step as an $\mathcal{IO}$ operation
- $\alpha.ip \in \mathbb{B}$ - marks the step as interleaving point of the reordered computation

for which we introduce the type:

$$\Theta_S = [0 : nu - 1] \times \mathcal{E} \times \mathbb{B} \times \mathbb{B}$$

Additionally, we have the following ownership transfer information for the step:

- $\alpha.A \subseteq \mathcal{A}$ - the set of acquired addresses
- $\alpha.L \subseteq \mathcal{A}$ - the set of acquired local addresses
- $\alpha.R \subseteq \mathcal{A}$ - the set of released addresses
- $\alpha.A_{pt} \subseteq \mathcal{A}$ - the set of acquired page table addresses
- $\alpha.L_{pt} \subseteq \mathcal{A}$ - the set of released page table addresses

Ownership transfer information is of type:

$$\Omega_S = (2^{\mathcal{A}})^5$$

Below we define projections, mapping step information α to transition information and ownership transfer information.

$$\alpha.t = (\alpha.s, \alpha.in, \alpha.io, \alpha.ip) \qquad \alpha.o = (\alpha.A, \alpha.L, \alpha.R, \alpha.A_{pt}, \alpha.R_{pt})$$

Note that the step information α contains not only the necessary inputs for the *Cosmos* machine step but also the flags $\alpha.ip$ and $\alpha.io$ which will be used as history information for bookkeeping in the order reduction proof. For $t \in \Theta_S$, $M \in \mathbb{M}_S$ and $X \in \{\mathcal{IO}, \mathcal{IP}\}$ we define shorthands $X(M,t) = X(M.u(t.s), M.m|_{\mathcal{R}}, t.in)$.

Definition 4.9 (Step Notation) The notation $M \overset{t}{\mapsto} M'$ denotes that transition $t \in \Theta_S$ is executed from machine state M, resulting in M'. Additionally $t.io$ corresponds to the values of the $\mathcal{IO}$ predicate and $t.ip$ corresponds with the value of the $\mathcal{IP}$ predicate.

$$M \overset{t}{\mapsto} M' \equiv M' = \Delta_t(M, t.s, t.in) \wedge \mathcal{IO}(M,t) = t.io \wedge \mathcal{IP}(M,t) = t.ip$$

For steps $\alpha \in \Sigma_S$ which include ownership transfer information we define a similar notation for the *Cosmos* machine transition from configuration C into C'.

$$C \overset{\alpha}{\mapsto} C' \;\equiv\; C.M \overset{\alpha.t}{\mapsto} C'.M \wedge C'.\mathcal{G} = \Delta_o(C.\mathcal{G}, \alpha.s, \alpha.io)$$

The definitions naturally extend to step sequences $\rho \in \Sigma_S^* \cup \Theta_S^*$ by induction:

$$X \overset{\rho}{\longrightarrow} X' \equiv (\exists X'', \tau, \alpha.\ \rho = \tau\alpha \wedge X \overset{\tau}{\longrightarrow} X'' \overset{\alpha}{\mapsto} X') \vee (\rho = \varepsilon \wedge X = X')$$

We use $\sigma \in \Sigma_S^*$, $\theta \in \Theta_S^*$, and $o \in \Omega_S^*$ to tell step sequences from transition sequences and ownership transfer sequences. A computation of *Cosmos* machine S can be performed with or without the ownership information since this is ghost, or specification state, respectively. A pair $(X, \rho) \in (\mathbb{K}_S \times \Sigma_S^*) \cup (\mathbb{M}_S \times \Theta_S^*)$ is then considered a *Cosmos* machine computation iff the following predicate holds:

$$comp(X, \rho) \;\equiv\; \exists X' \in \mathbb{K}_S \cup \mathbb{M}_S.\ X \overset{\rho}{\longrightarrow} X'$$

We extend our step projection functions to step sequences, by mapping sequences of step information σ to transition and ownership transfer sequences.

$$\sigma.t \equiv \sigma_0.t \cdots \sigma_{|\sigma|-1}.t \qquad\qquad \sigma.o \equiv \sigma_0.o \cdots \sigma_{|\sigma|-1}.o$$

For converting a pair of transition sequence θ and ownership transfer sequence o into a step sequence σ we use the construct $\langle \theta, o \rangle$ which gives us a sequence σ such that $|\sigma| = |\theta| = |o|$ and $\sigma.t = \theta \wedge \sigma.o = o$. In particular then $\sigma = \langle \sigma.t, \sigma.o \rangle$ holds.

4.1.6 Ownership Policy

In this subsection, we present a simplified version, compared to the one in Chapter 2, of ownership model and safety condition. All the access policy are identical to the corresponding one defined in the safety condition in section 2.2.3 except that we do not consider the page table set *pt* or the ownership transfer related to the read-only set.

Definition 4.10 (Ownership Memory Access Policy) Given a bit $io \in \mathbb{B}$, a *reads*-set *Read*, a *writes*-set *Write*, a set of owned addresses $\mathcal{O}$, a set of local page table address $\mathcal{P}t$, the set of shared addresses $\mathcal{S}$, the set of read-only addresses $\mathcal{R}$, and the set of addresses owned by other machines $\overline{\mathcal{O}}$, we enforce the following ownership memory access policy given by the predicate $policy_{acc}(io, Read, Write, \mathcal{O}, \mathcal{P}t, \mathcal{S}, \mathcal{R}, \overline{\mathcal{O}})$:

1. local steps (i) read only owned or read-only addresses and (ii) write only owned unshared addresses

$$/io \rightarrow \quad (i) \quad Read \subseteq \mathcal{O} \cup \mathcal{R}$$
$$(ii) \quad Write \subseteq \mathcal{O} \setminus \mathcal{S}$$

2. $\mathcal{IO}$-steps may (i) read owned, shared and read-only addresses while they (ii) may write owned addresses and shared addresses which are not owned by another machine.

$$io \rightarrow \quad (i) \quad Read \subseteq \mathcal{O} \cup \mathcal{S} \cup \mathcal{P}t \cup \mathcal{R}$$
$$(ii) \quad Write \subseteq \mathcal{O} \cup \mathcal{P}t \cup (\mathcal{S} \setminus \overline{\mathcal{O}})$$

Definition 4.11 (Ownership Transfer Policy) Given a bit $io \in \mathbb{B}$, a set of owned addresses $\mathcal{O}$, the set of shared addresses $\mathcal{S}$, the set of addresses owned by other machines $\overline{\mathcal{O}}$, as well as the updated sets for the owned and shared addresses $\mathcal{O}'$ and $\mathcal{S}'$, we restrict ownership transfer by the predicate $policy_{trans}(io, \mathcal{O}, \mathcal{P}t, \mathcal{S}, \overline{\mathcal{O}}, (A, L, R, A_{pt}, R_{pt}))$.

1. The ownership-state may not be changed by local steps.

$$/io \rightarrow A = \emptyset \wedge L = \emptyset \wedge R = \emptyset \wedge A_{pt} = \emptyset \wedge R_{pt} = \emptyset$$

2. For $\mathcal{IO}$-steps, the ownership-state is allowed to change as long as the step (i) acquires addresses which are shared unowned or already owned by the executing unit or released addresses from its local page table set and (ii) releases only owned addresses. And (iii) the acquired local addresses must be a subset of the acquired addresses and (iv) one may not acquire and release the same address at a time. Moreover (v) the page table acquired addresses are shared and unowned or already in the executing unit's local page table set or released from its owned set and (vi) it is disjoint with the acquired set. (vii) The released page table set is a subset of local page table set of the executing unit and (viii) disjoint with the acquired page table set.

$$\begin{aligned} io \rightarrow \quad & (i) \quad A \subseteq \mathcal{S} \setminus \overline{\mathcal{O}} \cup \mathcal{O} \cup R_{pt} \\ & (ii) \quad R \subseteq \mathcal{O} \\ & (iii) \quad L \subseteq A \\ & (iv) \quad A \cap R = \emptyset \\ & (v) \quad A_{pt} \subseteq \mathcal{S} \setminus \overline{\mathcal{O}} \cup \mathcal{P}t \cup R \\ & (vi) \quad A_{pt} \cap A = \emptyset \\ & (vii) \quad R_{pt} \subseteq \mathcal{P}t \\ & (viii) \quad R_{pt} \cap A_{pt} = \emptyset \end{aligned}$$

Definition 4.12 (Ownership Invariant) We state an ownership invariant *inv* on ownership state $\mathcal{G} \in \mathbb{G}_S$ of a *Cosmos* model, requiring (i) the owns-sets and the local page table sets of different units to be mutually disjoint and the owns-set and the local page table set of the same unit also to be disjoint and (ii) that read-only addresses may not be owned or in a local page table set or

shared-writable and the local page table sets are not shared. Moreover (iii) the complete address space is partitioned into the ownership sets as well as shared writable and read-only addresses. Moreover we set $inv(C) \equiv inv(C.\mathcal{G})$ for all $C \in \mathbb{K}_S$.

$$
\begin{aligned}
inv(C) \equiv \quad & (i) \quad \forall p, q.\ p \neq q \rightarrow \mathcal{G}.\mathcal{O}_p \cap \mathcal{G}.\mathcal{O}_q = \emptyset \ \wedge \\
& \qquad\qquad \mathcal{G}.\mathcal{P}t_p \cap \mathcal{G}.\mathcal{P}t_q = \emptyset \ \wedge \\
& \qquad\qquad \mathcal{G}.\mathcal{P}t_p \cap \mathcal{G}.\mathcal{O}_q = \emptyset \ \wedge \\
& \qquad\qquad \mathcal{G}.\mathcal{P}t_p \cap \mathcal{G}.\mathcal{O}_p = \emptyset \\
& (ii) \quad \forall p.\ \mathcal{G}.\mathcal{O}_p \cap \mathcal{R} = \emptyset \wedge \mathcal{G}.\mathcal{P}t_p \cap \mathcal{R} = \emptyset \ \wedge \\
& \qquad\qquad \mathcal{G}.\mathcal{S} \cap \mathcal{R} = \emptyset \wedge \mathcal{G}.\mathcal{S} \cap \mathcal{G}.\mathcal{P}t_p = \emptyset \\
& (iii) \quad \mathcal{A} = \bigcup_{p \in [0:nu-1]} \mathcal{G}.\mathcal{O}_p \cap \mathcal{G}.\mathcal{P}t_p \cup \mathcal{G}.\mathcal{S} \cup \mathcal{R}
\end{aligned}
$$

Lemma 4.13 (Ownership Transfer Properies) Given a configuration $C \in \mathbb{C}_S$ of a *Cosmos* machine S where the ownership invariant holds. Let $C' = \Delta(C, p, in, (A, L, R, A_{pt}, R_{pt}))$ for given step information $(p, in, io, ip, A, L, R, A_{pt}, R_{pt}) \in \Sigma_S$. If the step obeys $policy_{trans}$ and $inv(C)$, we can show (i) that addresses are only transferred between the owned addresses of p and the shared addresses, (ii) the new set of addresses owned by p is disjoint from the set of addresses owned by all other units, (iii) the new set of addresses owned by of p is disjoint from the local page table sets of all other units, (iv) the new local page table set of p is disjoint from the set of addresses owned by all other units, (v) the new local page table set of p is disjoint from the local page table sets of all other units, and (vi) the new local page set of p is disjoint from the new set of addresses owned by p. We let $\overline{\mathcal{O}} = \bigcup_{q \neq p} C.\mathcal{O}_q$ and $\overline{\mathcal{P}t} = \bigcup_{q \neq p} C.\mathcal{P}t_q$ then

$$
\begin{aligned}
& (i) && C'.\mathcal{P}t_p \cup C'.\mathcal{O}_p \cup C'.\mathcal{S} = C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup C.\mathcal{S} \\
& (ii) && \overline{\mathcal{O}} \cap C'.\mathcal{O}_p = \emptyset \\
& (iii) && \overline{\mathcal{P}t} \cap C'.\mathcal{O}_p = \emptyset \\
& (iv) && \overline{\mathcal{O}} \cap C'.\mathcal{P}t_p = \emptyset \\
& (v) && \overline{\mathcal{P}t} \cap C'.\mathcal{P}t_p = \emptyset \\
& (vi) && C'.\mathcal{P}t_p \cap C'.\mathcal{O}_p = \emptyset
\end{aligned}
$$

Proof: By definition of Δ we have:

$$
\begin{aligned}
\forall X \in \{\mathcal{O}, \mathcal{P}t\}, q \neq p.\ C'.X_q &= C.X_q \\
C'.\mathcal{O}_p &= C.\mathcal{O}_p \cup A \setminus R \\
C'.\mathcal{P}t_p &= C.\mathcal{P}t_p \cup A_{pt} \setminus R_{pt} \\
C'.\mathcal{S} &= C.\mathcal{S} \cup R \cup R_{pt} \setminus (L \cup A_{pt})
\end{aligned}
$$

- Claim *(i)*. First, we consider $C'.\mathcal{O}_p \cup C'.\mathcal{S}$.

$$\begin{aligned} C'.\mathcal{O}_p \cup C'.\mathcal{S} &= C.\mathcal{O}_p \cup A \setminus R \cup C.\mathcal{S} \cup R \cup R_{pt} \setminus (L \cup A_{pt}) \\ &= C.\mathcal{O}_p \cup A \setminus R \cup R \cup C.\mathcal{S} \cup R_{pt} \setminus (L \cup A_{pt}) \\ &= C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \cup R_{pt} \setminus (L \cup A_{pt}) \end{aligned}$$

Then, we have:

$$\begin{aligned} C'.\mathcal{P}t_p \cup C'.\mathcal{O}_p \cup C'.\mathcal{S} &= C.\mathcal{P}t_p \cup A_{pt} \setminus R_{pt} \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \cup R_{pt} \setminus (L \cup A_{pt}) \\ &= C.\mathcal{P}t_p \cup A_{pt} \setminus R_{pt} \cup R_{pt} \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus (L \cup A_{pt}) \\ &= C.\mathcal{P}t_p \cup A_{pt} \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus (L \cup A_{pt}) \\ &= C.\mathcal{P}t_p \cup A_{pt} \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L \setminus A_{pt} \end{aligned}$$

With $policy_{trans}$, we have:

$$L \subseteq A \wedge A \cap A_{pt} = \emptyset \tag{4.14}$$

Thus, we can conclude:

$$\begin{aligned} C'.\mathcal{P}t_p \cup C'.\mathcal{O}_p \cup C'.\mathcal{S} &= C.\mathcal{P}t_p \cup A_{pt} \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L \setminus A_{pt} \\ &= C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L \cup A_{pt} \setminus A_{pt} \\ &= C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L \end{aligned}$$

With (4.14), we can get:

$$\begin{aligned} C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L &\subseteq C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \\ C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L &\supseteq C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus A \\ &= C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup C.\mathcal{S} \end{aligned}$$

Also, with $policy_{trans}$, we have:

$$\begin{aligned} A &\subseteq C.\mathcal{S} \setminus \overline{O} \cup C.\mathcal{O}_p \cup R_{pt} \\ &\subseteq C.\mathcal{S} \setminus \overline{O} \cup C.\mathcal{O}_p \cup C.\mathcal{P}t_p \\ &\subseteq C.\mathcal{S} \cup C.\mathcal{O}_p \cup C.\mathcal{P}t_p \end{aligned}$$

We can conclude:

$$\begin{aligned} C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \setminus L &\subseteq C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup A \cup C.\mathcal{S} \\ &\subseteq C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup C.\mathcal{S} \end{aligned}$$

As a consequence, we have:

$$C'.\mathcal{P}t_p \cup C'.\mathcal{O}_p \cup C'.\mathcal{S} = C.\mathcal{P}t_p \cup C.\mathcal{O}_p \cup C.\mathcal{S}$$

- Claim *(ii)*. For the claim (ii), we need to use the invariant about the disjointness of ownership sets in C, in particular we have $\overline{O} \cap C.O_p = \emptyset$. Then it follows:

$$\begin{aligned}\overline{O} \cap C'.O_p &= \overline{O} \cap (C.O_p \cup A \setminus R) \\ &= \overline{O} \cap ((C.O_p \setminus R) \cup A) \\ &= (\overline{O} \cap (C.O_p \setminus R)) \cup (\overline{O} \cap A)\end{aligned}$$

From $policy_{trans}$, we can get:

$$\begin{aligned}A &\subseteq C.S \setminus \overline{O} \cup C.O_p \cup R_{pt} \\ &\subseteq C.S \setminus \overline{O} \cup C.O_p \cup C.\mathcal{P}t_p\end{aligned}$$

With $inv(C)$, we can conclude:

$$A \cap \overline{O} = \emptyset$$

Thus, we have:

$$\begin{aligned}\overline{O} \cap C'.O_p &= \overline{O} \cap (C.O_p \setminus R) \\ &\subseteq \overline{O} \cap C.O_p \\ &= \emptyset\end{aligned}$$

- Claim *(iii)*.

$$\begin{aligned}\overline{\mathcal{P}t} \cap C'.O_p &= \overline{\mathcal{P}t} \cap (C.O_p \cup A \setminus R) \\ &= \overline{\mathcal{P}t} \cap ((C.O_p \setminus R) \cup A) \\ &= (\overline{\mathcal{P}t} \cap (C.O_p \setminus R)) \cup (\overline{\mathcal{P}t} \cap A)\end{aligned}$$

From $policy_{trans}$, we can get:

$$\begin{aligned}A &\subseteq C.S \setminus \overline{O} \cup C.O_p \cup R_{pt} \\ &\subseteq C.S \setminus \overline{O} \cup C.O_p \cup C.\mathcal{P}t_p\end{aligned}$$

With $inv(C)$, we can conclude:

$$A \cap \overline{\mathcal{P}t} = \emptyset$$

Thus, we have:

$$\begin{aligned}\overline{\mathcal{P}t} \cap C'.O_p &= \overline{\mathcal{P}t} \cap (C.O_p \setminus R) \\ &\subseteq \overline{\mathcal{P}t} \cap C.O_p \\ &= \emptyset\end{aligned}$$

- Claim *(iv)*.

$$\begin{aligned}\overline{O} \cap C'.\mathcal{P}t_p &= \overline{O} \cap (C.\mathcal{P}t \cup A_{pt} \setminus R_{pt}) \\ &= \overline{O} \cap ((C.\mathcal{P}t \setminus R_{pt}) \cup A_{pt}) \\ &= (\overline{O} \cap (C.\mathcal{P}t \setminus R_{pt})) \cup (\overline{O} \cap A_{pt})\end{aligned}$$

  From $policy_{trans}$, we can get:

$$A_{pt} \subseteq C.\mathcal{S} \setminus (C.\mathcal{O}_p \cup \overline{O}) \cup C.\mathcal{P}t_p \cup R$$

  Also with $inv(C)$, we can get:

$$\overline{O} \cap A_{pt} = \emptyset$$

  Thus, we have:

$$\begin{aligned}\overline{O} \cap C'.\mathcal{P}t_p &= \overline{O} \cap (C.\mathcal{P}t_p \setminus R_{pt}) \\ &\subseteq \overline{O} \cap C.\mathcal{P}t_p \\ &= \emptyset\end{aligned}$$

- Claim *(v)*. This claim can be proved with analogous steps of previous cases.
- Claim *(vi)*.

$$\begin{aligned}C'.\mathcal{P}t_p \cap C'.\mathcal{O}_p &= (C.\mathcal{P}t_p \cup A_{pt} \setminus R_{pt}) \cap (C.\mathcal{O}_p \cup A \setminus R) \\ &\subseteq (C.\mathcal{P}t_p \cup A_{pt}) \cap (C.\mathcal{O}_p \cup A)\end{aligned}$$

  With $policy_{trans}$ and $inv(C)$, we can get:

$$C.\mathcal{P}t_p \cap C.\mathcal{O}_p = \emptyset \wedge A_{pt} \cap A = \emptyset$$

  We need to prove:

$$A_{pt} \cap C.\mathcal{O}_p = A \cap C.\mathcal{P}t_p = \emptyset \tag{4.15}$$

  From $policy_{trans}$, we can get:

$$\begin{aligned}A_{pt} &\subseteq C.\mathcal{S} \setminus (C.\mathcal{O}_p \cup \overline{O}) \cup C.\mathcal{P}t_p \cup R \\ A &\subseteq C.S \setminus \overline{O} \cup C.\mathcal{O}_p \cup R_{pt}\end{aligned}$$

  With $inv(C)$ we can conclude (4.15).

□

We subsume both the ownership access policy as well as the ownership transfer policy in a single predicate.

Definition 4.16 (Ownership-Safety of a Step) We consider a step of a *Cosmos* machine S from configuration $C \in \mathbb{M}_S$ with step information $\alpha \in \Sigma_S$ to be safe with respect to the ownership model (ownership-safe) when for

$$Read = core\text{-}reads(C.u(\alpha.s), C.m, \alpha.in)$$
$$Write = core\text{-}writes(C.u(\alpha.s), C.m, \alpha.in)$$

and $\overline{O} = \bigcup_{q \neq \alpha.s} C.O_q$ the following predicate is fulfilled.

$$\begin{aligned} safe_{step}(C, \alpha) \equiv\ & policy_{acc}(\alpha.io, Read, Write, C.O_{\alpha.s}, C.\mathcal{P}t_{\alpha.s}, C.\mathcal{S}, \mathcal{R}, \overline{O}) \wedge \\ & policy_{trans}(\alpha.io, C.O_{\alpha.s}, C.\mathcal{S}, \overline{O}, \alpha.o) \end{aligned}$$

Note that we will instantiate the *core-reads* and *core-writes* in the next section. The inductive extension of the notation for step sequences $\sigma \in \Sigma_S^*$ is straight forward.

Definition 4.17 (Ownership-Safety of a Computation) For a configuration C of a *Cosmos* model S, and $\tau \in \Sigma_S^*, \alpha \in \Sigma_S$ we define

$$\begin{aligned} safe(C, \varepsilon) \equiv & inv(C) \\ safe(C, \tau\alpha) \equiv & safe(C, \tau) \wedge \exists C', C''.\ C \xrightarrow{\tau} C' \overset{\alpha}{\mapsto} C'' \wedge safe_{step}(C', \alpha) \end{aligned}$$

Lemma 4.18 (Ownership-Safe Steps Preserve the Ownership Invariant) For configurations $C, C' \in \mathbb{C}_S$ of a *Cosmos* model and step sequence $\sigma \in \Sigma_S^*$, we have:

$$safe(C, \sigma) \wedge C \overset{\sigma}{\longmapsto} C' \rightarrow inv(C')$$

Proof: By induction on $n = |\sigma|$. For $n = 0$ we have $\sigma = \varepsilon$ and $C = C'$. By definition $safe(C, \varepsilon)$ collapses to $inv(C)$ hence $inv(C')$ follows directly.

In the induction step we extend σ from length $n - 1$ to n. We introduce the intermediate configuration C'' as follows.

$$C \overset{\sigma_{[0:n-1)}}{\longrightarrow} C' \overset{\sigma_{n-1}}{\mapsto} C''$$

Induction hypothesis yields $inv(C')$. The ownership invariants can only be broken by an unsafe modification of the ownership state in step σ_{n-1}. In particular we need to consider the set of shared addresses $C'.\mathcal{S}$, the sets of owned addresses $C'.O_p$ and the local page table sets $C'.\mathcal{P}t_p$ for all machine p. Note that by construction a machine can only modify its own ownership set, thus we have:

$$\forall q \neq \sigma_{n-1}.s.\ C'.O_q = C''.O_q \wedge C'.\mathcal{P}t_q = C''.\mathcal{P}t_q$$

Moreover the modification of the ownership configuration is regulated by the $policy_{trans}$ predicate which is part of the definition of $safe_{step}(C', \sigma_{n-1})$. The sets $C'.\mathcal{S}$ and $C'.O_{\sigma_{n-1}.s}$ may not be changed by local steps, then invariants hold by induction hypothesis. For $\mathcal{IO}$ steps of $\sigma_{n-1}.s$

by Lemma 4.13 we obtain the following two necessary requirements for safe ownership transfer.

$$\begin{aligned}
&(i) && C'.\mathcal{P}t_p \cup C'.\mathcal{O}_p \cup C'.\mathcal{S} = C''.\mathcal{P}t_p \cup C''.\mathcal{O}_p \cup C''.\mathcal{S} \\
&(ii) && \overline{\mathcal{O}} \cap C''.\mathcal{O}_p = \emptyset \\
&(iii) && \overline{\mathcal{P}t} \cap C''.\mathcal{O}_p = \emptyset \\
&(iv) && \overline{\mathcal{O}} \cap C''.\mathcal{P}t_p = \emptyset \\
&(v) && \overline{\mathcal{P}t} \cap C''.\mathcal{P}t_p = \emptyset \\
&(vi) && C''.\mathcal{P}t_p \cap C''.\mathcal{O}_p = \emptyset
\end{aligned}$$

Here $\overline{\mathcal{O}} = \bigcup_{q \neq \sigma_{n-1}.s} C'.\mathcal{O}_q$ and $\overline{\mathcal{P}t} = \bigcup_{q \neq \sigma_{n-1}.s} C'.\mathcal{P}t_q$ denotes the set of addresses owned by all other machines and the local page tables of all other machines in configuration C'. As explained above $\overline{\mathcal{O}}$ and $\overline{\mathcal{P}t}$ is not affected by σ_{n-1}. We now prove the parts of ownership invariant $inv(C'')$ one by one.

1. We need to prove:

$$\begin{aligned}
\forall p, q.\ p \neq q \rightarrow & C''.\mathcal{O}_p \cap C''.\mathcal{O}_q = \emptyset \\
& C''.\mathcal{P}t_p \cap C''.\mathcal{P}t_q = \emptyset \\
& C''.\mathcal{P}t_p \cap C''.\mathcal{O}_q = \emptyset \\
& C''.\mathcal{P}t_p \cap C''.\mathcal{O}_p = \emptyset
\end{aligned}$$

   If neither p nor q equals $\sigma_{n-1}.s$ the claim follows immediately from $\forall X \in \{\mathcal{O}, \mathcal{P}t\}, Y \in \{p, q\}$. $C'.X_Y = C''.X_Y$, and $inv(C')$. Otherwise we assume wlog. that $p = \sigma_{n-1}.s$, thus by $C'.X_q = C''.X_q$ and the definition of $\overline{X}$ we get $C''.X_q \subseteq \overline{X}$. From requirement *(ii)* to *(vi)* we can conclude this claim.

2. $\forall p.\ C''.X_p \cap \mathcal{R} = \emptyset$ - If $p \neq \sigma_{n-1}.s$ we have $C''.X_p = C'.X_p$ and by invariant $C'.X_p \cap \mathcal{R} = \emptyset$, hence $C''.X_p \cap \mathcal{R} = \emptyset$ holds. Otherwise, for $p = \sigma_{n-1}.s$, from necessary requirement *(i)* we get $C''.X_{\sigma_{n-1}.s} \subseteq C'.\mathcal{O}_{\sigma_{n-1}.s} \cup C'.\mathcal{S} \cup C'.\mathcal{P}t_{\sigma_{n-1}.s}$, however by ownership invariant $C'.X_{\sigma_{n-1}.s}$ and $C'.\mathcal{S}$ are disjoint from $\mathcal{R}$. Therefore also $C''.X_{\sigma_{n-1}.s}$ is disjoint from $\mathcal{R}$.

3. $C''.\mathcal{S} \cap \mathcal{R} = \emptyset$ - This follows with the same argumentation as in the second part of the case above for $C''.\mathcal{S}$ instead of $C''.X_{\sigma_{n-1}.s}$.

4. $C''.\mathcal{S} \cap C''.\mathcal{P}t_p = \emptyset$ -From the semantics, we have:

$$\begin{aligned}
\forall q \neq \sigma_{n-1}.s.\ C''.\mathcal{P}t_q &= C'.\mathcal{P}t_q \\
C''.\mathcal{P}t_{\sigma_{n-1}.s} &= C'.\mathcal{P}t_{\sigma_{n-1}.s} \cup A_{pt} \setminus R_{pt} \\
C''.\mathcal{S} &= C'.\mathcal{S} \cup R \cup R_{pt} \setminus (L \cup A_{pt})
\end{aligned}$$

   with $inv(C')$ and $policy_{trans}$, we can imply $C''.\mathcal{S} \cap C''.\mathcal{P}t_p = \emptyset$.

5. $\mathcal{A} = \bigcup_{p \in \mathbb{N}_{np}} C''.\mathcal{P}t_p \cup C''.\mathcal{O}_p \cup C''.\mathcal{S} \cup \mathcal{R}$ - The invariant says that all addresses of $\mathcal{A}$ are read-only, shared-writable, owned by some machine or in some machine's local page

table. Using $\overline{X} = \bigcup_{q \neq \sigma_{n-1}.s} C'.X_q = \bigcup_{q \neq \sigma_{n-1}.s} C''.X_q$ this notion can be reformulated as follows:

$$\mathcal{R} \cup C''.\mathcal{S} \cup C''.\mathcal{O}_{\sigma_{n-1}.s} \cup \overline{\mathcal{O}} \cup C''.\mathcal{P}t_{\sigma_{n-1}.s} \cup \overline{\mathcal{P}t} = \mathcal{A}$$

We already have $\mathcal{R} \cup C'.\mathcal{S} \cup C'.\mathcal{O}_{\sigma_{n-1}.s} \cup \overline{\mathcal{O}} \cup C'.\mathcal{P}t_{\sigma_{n-1}.s} \cup \overline{\mathcal{P}t} = \mathcal{A}$ by $inv(C')$. By *(i)* on the ownership transfer we have

$$C'.\mathcal{S} \cup C'.\mathcal{O}_{\sigma_{n-1}.s} \cup C'.\mathcal{P}t_{\sigma_{n-1}.s} = C''.\mathcal{S} \cup C''.\mathcal{O}_{\sigma_{n-1}.s} \cup C''.\mathcal{P}t_{\sigma_{n-1}.s}$$

and the invariant on C'' stated above follows immediately. □

We show not only the ownership-safety but also the arbitrary verified safety properties on the concurrent system. In general, safety properties constrain finite behavior of a *Cosmos* machine and must hold in every traversed state of a *Cosmos* machine computation. Thus we can represent them as an invariant $P : \mathbb{C}_S \to \mathbb{B}$ on the *Cosmos* machine configuration. We extend our safety predicate accordingly:

$$safe_P(C, \sigma) \equiv safe(C, \sigma) \wedge \forall C'.\, C \xrightarrow{\sigma} C' \to P(C')$$

Then we have the following predicates denoting the verification of properties for a particular *Cosmos* model.

Definition 4.19 (Verified *Cosmos* machine) We define the predicate $safety(C, P)$ which states that for all *Cosmos* machine computations starting in C we can find an ownership annotation such that the computation is safe and preserves the given property P.

$$safety(C, P) \equiv \forall \theta.\, comp(C.M, \theta) \to \exists o \in \Omega_S^*.\, safe_P(C, \langle \theta, o \rangle)$$

4.2 SB Reduced MIPS-86 Instantiation

In this section we will instantiate the *Cosmos* model model with the SB reduced MIPS-86 ISA. The instantiation needs to refine the components of a *Cosmos* machine $S \in \mathbb{S}$, which we list again below as a reminder.

$$S = (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP})$$

Moreover for the instantiation we have do discharge instantiation restriction $insta_r(S)$ on the *reads*-function which determines the *reads*-set for a step of a computation unit.

- $S^n_{\text{MIPS-86}}.\mathcal{A} = \mathbb{B}^{30}$ and $S^n_{\text{MIPS-86}}.\mathcal{V} = \mathbb{B}^{32}$ — The memory is word-addressable and contains 2^{30} memory cells.
- $S^n_{\text{MIPS-86}}.\mathcal{R} = A_{phy\text{-}code}$ — We assume that all code to be executed lies in an area $A_{phy\text{-}code} \subseteq \mathcal{A}$ and we set the read-only addresses to be identical with this area. Thus, the self-modifying code can be excluded by the ownership access policy.
- $S^n_{\text{MIPS-86}}.nu = np$ — We have the as many computation units as the number of threads in the abstract machine and the SB machine in Chapter 2.

- $S^n_{\text{MIPS-86}}.\mathcal{U} = K_{sbr\text{-}pro} \times \mathbb{N} \times \mathbb{B} \times (\mathbb{T} \rightharpoonup \mathbb{V})$ — Every computation unit consists a sequential MIPS processor p, a counter n, a dirty bit $\mathcal{D}$ and a temporary ϑ which is a partial function from $\{I, R\} \times \mathbb{N}$ to a 30-bit address a. For all $X \in \{I, R\}$ in (X, n) we write X_n for short. Initially, all X_n map to $\perp$. For all $Y \in \{pc, gpr, spr\}$ we simply write $u.p.Y$ instead of $u.p.core.Y$.

- $S^n_{\text{MIPS-86}}.\mathcal{E} = \Sigma_{sbr\text{-}seq}$ —The input of processor transition function. Recall that the input is defined as:

$$\begin{aligned}\Sigma_{seq} = {} & \{\textbf{core}\} \times K_{\textbf{walk}} \times K_{\textbf{walk}} \times \mathbb{B}^{256} \\ & \cup \{\textbf{tlb-create}\} \times \mathbb{B}^{20} \\ & \cup \{\textbf{tlb-extend}\} \times K_{\textbf{walk}} \\ & \cup \{\textbf{tlb-accessed-dirty}\} \times K_{\textbf{walk}}\end{aligned}$$

  Note that, depending on the input, the computation unit of a *Cosmos* machine can make a:

  – core step to execute an instruction or interrupt.

  – TLB create step to create a new walk.

  – TLB extend step to extend an existing walk.

  – TLB set access-dirty step to set the access and dirty bits of a PTE.

- $S^n_{\text{MIPS-86}}.reads$ — Before defining the reads set, we have to define some auxiliary predicates and shorthands when $in = (\textbf{core}, w_I, w_R, eev)$.

  – $mode \equiv u.spr(mode)[0]$

  – $trqI = (u.pc[31:2], 011)$. Translation request for instruction fetch.

  – $pff \equiv fault(pte(m, w_I), trqI, w_I)$. Signals whether there is a page-fault-on-fetch for the given walk w_I and the translation request $trqI$.

  – $pmaI = \begin{cases} w_I.ba \circ u.pc[11:2] & mode \\ u.pc[31:2] & otherwise \end{cases}$. The physical memory address for instruction fetch of processor core i (which is only meaningful if no page-fault on instruction fetch occurs),

  – $I = m(pmaI)$. The instruction fetched from memory. Because the self-modifying code is forbidden, we can directly read from memory (in case of a page-fault-on-fetch the value of I has no further relevance).

  – $trqEA = (ea(u.p.core, I)[31:2], (store(I) \vee rmw(I)) \circ 10)$. The translation request for the effective address.

- $pfls \equiv mode \wedge fault(pte(m, w_R), trqEA, w_R) \wedge \neg pff \wedge (store(I) \vee load(I) \vee rmw(I))$. The page-fault-on-load-store signal.
- $pmaEA = \begin{cases} w_R.ba \circ ea(u.p.core, I)[11:2] & mode \\ ea(u.p.core, I)[31:2] & otherwise \end{cases}$. The physical memory address for the effective address.

The jump to interrupt service routine predicate is defined as:

$$\begin{aligned} jisr_f(u, eev, pff) &\equiv jisr_f(u.p.core, eev, pff) \\ jisr_x(u, I, eev, pfls) &\equiv jisr_x(u.p.core, I, eev, pfls) \\ jisr(u, I, eev, pff, pfls) &\equiv jisr_f(u, eev, pff) \vee jisr_x(u, I, eev, pfls) \end{aligned}$$

Depending on the executed instructions and the interrupt level different sets of addresses are loaded from memory.

$$\begin{aligned} &core\text{-}reads(u, m, in) \\ &= \begin{cases} \{pmaI, pmaEA\} & in = (\textbf{core}, w_I, w_R, eev) \wedge \\ & \neg jisr(u, I, eev, pff, pfls) \wedge (load(I) \vee rmw(I)) \\ \{pmaI\} & in = (\textbf{core}, w_I, w_R, eev) \wedge \neg jisr(u, I, eev, pff, pfls) \wedge \\ & \neg(load(I) \vee rmw(I)) \\ \{pmaI\} & in = (\textbf{core}, w_I, w_R, eev) \wedge jisr_x(u, I, eev, pfls) \wedge \\ & \neg jisr_f(u, eev, pff) \\ \emptyset & otherwise \end{cases} \\ &S^n_{\text{MIPS-86}}.reads(u, m, in) \\ &= \begin{cases} core\text{-}reads(u, m, in) & in = (\textbf{core}, w_I, w_R, eev) \\ \{ptea(w)\} & in = (\textbf{tlb-extend}, w) \\ \{ptea(w)\} & in = (\textbf{tlb-accessed-dirty}, w) \end{cases} \end{aligned}$$

We need to prove the predicate $insta_r(S^n_{\text{MIPS-86}})$ is for our instantiation. Let

$$Read = S^n_{\text{MIPS-86}}.reads(u, m, in)$$

then

$$m|_{Read} = m'|_{Read} \rightarrow S^n_{\text{MIPS-86}}.reads(u, m', in) = Read$$

Proof Let $Read' = S^n_{\text{MIPS-86}}.reads(u, m', in)$. From the definition of $S^n_{\text{MIPS-86}}.reads$ we can conclude that $Read$ and $Read'$ only depend on the computation unit u and external input in. As a consequence, $Read$ trivially equals to $Read'$. □

- $S^n_{\text{MIPS-86}}.\delta$ — As in Chapter 3, A_{io} is the set of shared memory access instruction virtual addresses. In the *Cosmos* machine the δ-function of the computation units gets only a partial memory as an input, that is determined by the *reads*-set. However the δ_m is defined for a memory that is a total function. Nevertheless we can transform any partial memory function $m : \mathbb{B}^{30} \rightharpoonup \mathbb{B}^{32}$ into a total one by filling in dummy values.

$$\lceil m \rceil = \lambda a \in \mathbb{B}^{30}. \begin{cases} 0^{32} & : \quad m(a) = \perp \\ m(a) & : \quad \text{otherwise} \end{cases}$$

In the definition we let

$$R = \begin{cases} \perp & pff \vee pfls \\ m(pmaEA) & otherwise \end{cases}$$

then define u' and m' as:

$$\begin{aligned}
u'.p &= \delta_{sbr\text{-}seq}((u.p, \lceil m \rceil), in).p \\
u'.n &= u.n + 1 \\
u'.\mathcal{D} &= \begin{cases} True & in = (\textbf{core}, w_I, w_R, eev) \wedge store(I) \wedge u.pc[31:2] \in A_{io} \\ False & in = (\textbf{core}, w_I, w_R, eev) \wedge sbf(I) \vee jisr(u, I, eev, pff, pfls) \\ u.\mathcal{D} & otherwise \end{cases} \\
u'.\vartheta &= \begin{cases} \vartheta' & in = (\textbf{core}, w_I, w_R, eev) \wedge \\ & \neg jisr(u, I, eev, pff, pfls) \wedge (load(I) \vee rmw(I)) \\ u.\vartheta(I_{u.n} \mapsto I) & in = (\textbf{core}, w_I, w_R, eev) \wedge \neg jisr_f(u, eev, pff) \wedge \\ & (\neg jisr_x(u, I, eev, pfls) \rightarrow \neg load(I) \wedge \neg rmw(I)) \\ u.\vartheta & otherwise \end{cases} \\
m' &= \delta_{sbr\text{-}seq}((u.p, \lceil m \rceil), in).m
\end{aligned}$$

where:

$$\begin{aligned}
\vartheta' &= u.\vartheta(I_{u.n} \mapsto I)(R_{u.n} \mapsto lv(R, I)) \\
sbf(I) &\equiv rmw(I) \vee mfence(I) \vee eret(I) \vee invlpg(I) \vee flush(I) \vee switch(I) \vee wpto(I)
\end{aligned}$$

We define the set of written addresses $W(u, m, in)$. A write operation is performed if predicate $wr(u, m, eev)$ holds.

$$\begin{aligned}
wr(u, m, eev) &\equiv (store(I) \vee rmw(I) \wedge m(pmaEA) = u.p.gpr(rd(I))) \wedge \\
&\qquad \neg jisr(u, I, eev, pff, pfls) \\
core\text{-}writes(u, m, in) &= \begin{cases} \{pmaEA\} & in = (\textbf{core}, w_I, w_R, eev) \wedge wr(u, m, eev) \\ \emptyset & otherwise \end{cases} \\
W(u, m, in) &= \begin{cases} core\text{-}writes(u, m, in) & in = (\textbf{core}, w_I, w_R, eev) \\ \{ptea(w)\} & in = (\textbf{set-accessed-dirty}, w) \\ \emptyset & otherwise \end{cases}
\end{aligned}$$

We can define the transition function for MIPS computation units which returns the same new core configuration and the updated part of memory. We define:

$$S^n_{\text{MIPS-86}}.\delta(u, m, in) = (u', m'|_{W(u,\lceil m\rceil,in)})$$

- $S^n_{\text{MIPS-86}}.\mathcal{IO}$ — Unlike the C level, in which the $\mathcal{IO}$ steps are the accesses to volatile variables or calls to synchronization primitives (for example *rmw*), the choice $\mathcal{IO}$ steps on ISA level are made by the verification engineer. We collect the virtual addresses of the $\mathcal{IO}$ instructions in a set A_{io}. Then the definition of the $\mathcal{IO}$ steps on the ISA level is straight forward.

$$\begin{aligned} &S^n_{\text{MIPS-86}}.\mathcal{IO}(u, m, in) \ \equiv \\ &\quad in = (\textbf{core}, w_I, w_R, eev) \wedge \neg jisr(u, I, eev, pff, pfls) \wedge u.pc[31:2] \in A_{io} \end{aligned}$$

- $S^n_{\text{MIPS-86}}.\mathcal{IP}$ — Similarly, what are the $\mathcal{IP}$ steps depends on the compiler and can not be determined in the ISA level. We also collect the virtual address of the $\mathcal{IP}$ instructions in a set A_{cp} which is given by the verification engineer.

$$\begin{aligned} &S^n_{\text{MIPS-86}}.\mathcal{IP}(u, m, in) \ \equiv \\ &\quad in = (\textbf{core}, w_I, w_R, eev) \wedge \neg jisr(u, I, eev, pff, pfls) \wedge u.pc[31:2] \in A_{cp} \end{aligned}$$

Note that we assume an invariant on computations of *Cosmos* machine $S^n_{\text{MIPS-86}}$, stating that $A_{phy\text{-}code}$ has the intended meaning, namely, that we only fetch instructions from this set of addresses.

Definition 4.20 (Initial Configuration of SB reduced MIPS-86 *Cosmos* machine) For the initial configuration C^0, we have

$$\forall t, i \in [0 : np-1].\ C^0.u_i.n = 0 \wedge C^0.u_i.\vartheta(t) = \perp$$

Definition 4.21 (Code Region Invariant) We define the invariant $codeinv(C, A_{phy\text{-}code})$ which states that in all system states reachable from *Cosmos* machine configuration $C \in \mathbb{K}_{S^n_{\text{MIPS-86}}}$ instructions are only fetched from code region $A_{phy\text{-}code} \subseteq \mathbb{B}^{30}$.

$$\begin{aligned} \forall \tau, C'.\ C \xrightarrow{\tau} C' \rightarrow \forall \alpha.\ \ &\alpha.in = (\textbf{core}, w_I, w_R, eev) \wedge \\ &\neg jisr_f(C'.u_{\alpha.s}, eev, pff) \wedge pmaI' \subseteq A_{phy\text{-}code} \end{aligned}$$

in which

$$pmaI' = \begin{cases} w_I.ba \circ C'.u_{\alpha.s}.p.pc[11:2] & C'.u_{\alpha.s}.p.spr(mode)[0] \\ C'.u_{\alpha.s}.pc[31:2] & otherwise \end{cases}$$

4.3 Application of SB Reduction with MMU to MIPS-86

In this section, we will prove the simulation between the instantiated abstract machine and the SB reduced MIPS-86 *Cosmos* machine. First, we reduce the interleaving of the abstract machine. Second, we instantiate the safety property P in the safety condition of the *Cosmos* machine in Definition 4.19. Then, we introduce the coupling relation. Moreover, we prove the simulation theorem.At last, we prove that the safety condition is transferred from the SB reduced MIPS-86 *Cosmos* machine to the abstract machine for the following reason: (i) in the model stack (Fig. 4.1), the ownership safety of low level should follow that of the high level. (ii) the transfer of safety condition enables the application of SB reduction on the abstract level.

Note that, in this section, we only consider the simulation of finite computations because in reality, every computation is finite.

4.3.1 Interleaving Reduction of Abstract Machine Computation

For the same reason as in Section 4.1, we restrict that the read-only set of the abstract machine can not be changed. The restriction can simplify the following reordering proof in this section. Also, we restrict that the read-only memory can not be written.

Definition 4.22 (Read-Only Invariant)

$$
\begin{aligned}
&(c^0 \underset{\text{eev}}{\Longrightarrow}^{*}\ c \rightarrow c^0.ro = c.ro) \wedge \\
&(I = hd(c.is_{[i]}) \wedge (W(I) \vee RMW(I) \wedge I.cond(\vartheta')) \wedge \\
&\qquad pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \rightarrow pa \notin c.ro) \wedge \\
&(c \overset{\text{muw}}{\Longrightarrow}_i c' \rightarrow a \notin c.ro)
\end{aligned}
$$

in which ϑ' is defined in Definition 2.13 and a is the target address of the MMU write step of thread i ($\overset{\text{muw}}{\Longrightarrow}_i$).

We define the following shorthands for the rest of this section: $n^x = c^x.p_{[i]}.n$, $I^x_{isa} = c^x.\vartheta_{[i]}(I_{n^x})$, $R^x_{isa} = c^x.\vartheta_{[i]}(R_{n^x})$ and $I^x = hd(c^x.is_{[i]})$. Recall that according to our instantiation in Section 3.2.3, the execution of one instruction in the abstract machine without interruption can be divided into the following phases: Initially, we have

$$I_{isa} = \bot \wedge c.is_{[i]} = [] \wedge c.p_{[i]}.fetch$$

After the phase 1 program step, we get a new machine configuration c'.

$$I'_{isa} = \bot \wedge \neg c'.p_{[i]}.fetch \wedge |c'.is_{[i]}| = 1 \wedge nvR(I')$$

After the phase 2 memory step we have a new machine configuration c'' which satisfies:

$$I''_{isa} \neq \bot \wedge c''.is_{[i]} = [] \wedge \neg c''.p_{[i]}.fetch$$

After the phase 3 program step, the new machine configuration c''' satisfies:

$$
\begin{aligned}
&I'''_{isa} = I''_{isa} \neq \bot \wedge c'''.p_{[i]}.fetch \wedge \\
&(\neg gen\text{-}ins(I''_{isa}) \rightarrow c'''.is_{[i]} = [])\ \wedge (gen\text{-}ins(I''_{isa}) \rightarrow |c'''.is_{[i]}| = 1)
\end{aligned}
$$

where

$$\begin{aligned} \textit{gen-ins}(I_{isa}) \equiv & \textit{load}(I_{isa}) \vee \textit{store}(I_{isa}) \vee \textit{rmw}(I_{isa}) \vee \textit{flush}(I_{isa}) \vee \textit{mfence}(I_{isa}) \vee \\ & \textit{eret}(I_{isa}) \vee \textit{switch}(I_{isa}) \vee \textit{wpto}(I_{isa}) \vee \textit{invlpg}(I_{isa}) \end{aligned}$$

Depending on weather $c'''.is_{[i]} = []$, the next step of thread i either performs a phase 4 memory step or a phase 5 program step.

- $c'''.is_{[i]} \neq []$. After the phase 4 memory step we have a new machine configuration c^4:

  $$I^4_{isa} = I''_{isa} \neq \bot \wedge c^4.is_{[i]} = [] \wedge c^4.p_{[i]}.\textit{fetch}$$

  Then, after the following phase 5 program step, we get c^5 which satisfies:

  $$I^5_{isa} = \bot \wedge c^5.is_{[i]} = [] \wedge c^5.p_{[i]}.\textit{fetch}$$

- $c'''.is_{[i]} = []$. After the phase 5 program step we reach a machine configuration $c^{5'}$:

  $$I^{5'}_{isa} = \bot \wedge c^{5'}.is_{[i]} = [] \wedge c^{5'}.p_{[i]}.\textit{fetch}$$

We define following auxiliary predicates to check the phase of a configuration.

$$\begin{aligned} \textit{phase}1(c,i) &\equiv c.p_{[i]}.\textit{fetch} \wedge c.is_{[i]} = [] \wedge I_{isa} = \bot \\ \textit{phase}2(c,i) &\equiv \neg c.p_{[i]}.\textit{fetch} \wedge c.is_{[i]} \neq [] \wedge I_{isa} = \bot \\ \textit{phase}3(c,i) &\equiv \neg c.p_{[i]}.\textit{fetch} \wedge c.is_{[i]} = [] \wedge I_{isa} \neq \bot \\ \textit{phase}4(c,i) &\equiv c.p_{[i]}.\textit{fetch} \wedge c.is_{[i]} \neq [] \\ \textit{phase}5(c,i) &\equiv c.p_{[i]}.\textit{fetch} \wedge c.is_{[i]} = [] \wedge I_{isa} \neq \bot \end{aligned}$$

Note that these predicates also hold when interrupts happen. From the semantics, we have that after an interrupted program step, a mode switch instruction is generated and the *fetch* flag is set. The next step of the same thread will be a phase 4 memory step. With the definition of the predicates, we also have *phase*4. After a page fault step, the machine sets the *fetch* flag and increases the counter. It means that the value of the temporary with respect to the current counter is undefined. The page fault step also clears the instruction sequence. With the definition of the predicates, we have *phase*1. According to the semantics, the next step of the same thread will be a phase 1 program step.

We also define a function to check the thread i of an abstract machine configuration c is in which phase:

$$\textit{phase}(c,i) = \begin{cases} 1 & \textit{phase}1(c,i) \\ 2 & \textit{phase}2(c,i) \\ 3 & \textit{phase}3(c,i) \\ 4 & \textit{phase}4(c,i) \\ 5 & \textit{phase}5(c,i) \end{cases}$$

According to the definition of $phase1$,...,$phase5$, for a given c and i only one of the predicates can be true and must be true. Thus, the function $phase$ is well-defined.

Also, we define the code region invariant for the abstract machine. To reduce the overhead, we do not want the instruction fetch operations to flush the SB. Thus, the instruction fetches are performed by non-volatile reads. We assume that the physical address of pc is in the read-only memory to maintain the ownership policy. From the semantics of the abstract machine, we know

Definition 4.23 (Code Region Invariant) We let $I = hd(c.is_{[i]})$ then

$$\forall c.\ phase(c, i) = 2 \rightarrow atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \in c.ro$$

In this section, we will reduce the set of possible interleaving of the abstract machine computation. That means, we want to reorder the steps of abstract machine computation such that the steps belong to the same round (from one phase 1 step until but not including the next phase 1 step. For detail see Section 3.2.) execute consecutively and maintain the thread-local order. We call the consecutive execution of steps belong to the same round an interleaving block.

Every interleaving block can be a complete block or incomplete block. We give the formal definition of complete and incomplete interleaving blocks.

Definition 4.24 (Complete Interleaving Block) $c \underset{eev}{\Rightarrow_i^*} c''$ is a complete interleaving block of thread i iff it satisfies one of the following:

- $c \overset{mu}{\Longrightarrow}_i c''$. The block contains only one MMU step.
- $\neg\exists c^1, c^2.\ \neg(c^1 = c \wedge c^2 = c'') \wedge c \underset{eev}{\Rightarrow_i^*} c^1 \overset{mu}{\Longrightarrow}_i c^2 \underset{eev}{\Rightarrow_i^*} c''$. The block contains no MMU steps at all. In this case, we also require:
  1. $\forall j.\ phase(c, j) = 1$. The block starts with a configuration in phase 1 of every thread.
  2. $\forall j.\ phase(c'', j) = 1$. The block ends with a configuration in phase 1 of every thread.
  3. $\neg\exists c' \notin \{c'', c\}.\ phase(c', i) = 1 \wedge c \underset{eev}{\Rightarrow_i^*} c' \underset{eev}{\Rightarrow_i^*} c''$. Between c and c'' there exists no configuration in phase 1 of thread i.

Definition 4.25 (Incomplete Interleaving Block) $c \underset{eev}{\Rightarrow_i^*} c''$ is a incomplete interleaving block of thread i iff it satisfies all of the following conditions:

- $\forall j.\ phase(c, j) = 1$. The block starts with a configuration in phase 1 of every thread.
- $phase(c'', i) \neq i \wedge \forall j \neq i.\ phase(c'', j) = 1$. The block ends with a configuration other than phase 1 of thread i. Since the steps of thread i does not change the phases of other threads , the phase of thread $j \neq i$ is unchanged (Lemma 4.37).
- $\neg\exists c^1, c^2.\ \neg(c^1 = c \wedge c^2 = c'') \wedge c \underset{eev}{\Rightarrow_i^*} c^1 \overset{mu}{\Longrightarrow}_i c^2 \underset{eev}{\Rightarrow_i^*} c''$. The block contains no MMU steps at all.

- $\neg\exists c' \notin \{c'', c\}.\ phase(c', i) = 1 \wedge c \underset{\text{eev}}{\Rightarrow}_i^* c' \underset{\text{eev}}{\Rightarrow}_i^* c''$. Between c and c'' there exists no configuration in phase 1 of thread i.

To get the interleaving blocks, we reorder the computation in the following way: we do not touch the MMU steps, the page fault steps, the phase 3 program steps which do not generate instructions, and the phase 4 memory steps. We collect the steps which belong to the same round together by moving them towards the untouched step. The semantics guarantee that there exists only one untouched step in each round. After reordering, the order of interleaving blocks is identical to the order of the untouched steps in the original computation.

To reorder the computation, we keep traversing the computation from the initial configuration till the end configuration, for each configuration c if it makes a step of thread i, we do a case split:

1. $phase(c, i) = 1$. In this case, we postpone the step as long as possible until we reach another non-MMU step of thread i. According to the semantics, this step is a phase 2 memory step or a phase 4 memory step. The phase 1 program step can be postponed because it only depends on the thread-local components, which can not be changed by the steps of others or the MMU step of thread i. Also, the program step does not change the global components and thread-local components of other threads as well as the MMU state of thread i. After the postponing, we start to handle the next configuration reachable via the postponed step.

2. $phase(c, i) = 2$. In this case, from the semantics we know that the machine can perform a memory step or a page fault step. Then we do a further case split:

   - If the machine makes a memory step. From the semantics, we can get the machine fetches an instruction in this step. As in the previous case, we also postpone this step as long as possible until we reach another non-MMU step of thread i, which is a phase 3 program step according to the semantics. Because of the TLB, which is also a thread-local component and can not be modified by other threads, we can get the same address translation. By Definition 4.23 and Definition 4.22, the translated address is in the static read-only memory. Thus, it also can not be modified by other threads. After the postponing, the machine can also fetch the same instruction.
   - If the machine makes a page fault step. According to semantics, this step is the last step in the current interleaving block. We do not reorder this step.

   After that, we start to handle the next configuration reachable via the possibly postponed phase 2 step.

3. $phase(c, i) = 3$. In this case, we make a case split on whether the program step generates instructions or not.

   - If no instruction is generated. We do not reorder this step. In this case, we directly start to handle the next configuration.
   - Otherwise. Analogous to the rule 1, the phase 3 program step is postponed until the next non-MMU step of thread i. After that, we start to handle the next configuration reachable via the postponed phase 3 step.

4. $phase(c, i) = 4$. In this case, we do not reorder this step and directly start to handle the next configuration.

5. $phase(c, i) = 5$. In this case, we move the phase 5 program step forward until the previous non-MMU step of thread i. The moving is possible for the same reason as in rule 1. After that, we start to handle the next configuration reachable via the advanced phase 5 step.

6. We do not reorder other steps (i.e. the MMU steps).

We iteratively traverse and reorder the whole finite computation until there is no step to reorder.

To inductively prove the reordering gives us an equivalence computation. We need to prove the following:

- The execution of one instruction in thread i is serialized. For example, the thread i of the abstract machine first performs a phase 1 program step, then a phase 2 memory step for fetching and so on.
- The phase can not be affected by other threads or MMU steps.
- A program step can be move one step forward or backward and does not affect the computation if the corresponding neighboring step is not a non-MMU step of thread i.
- A phase 2 memory step can be postponed one step and do not affect the computation if the next step is not a non-MMU step of thread i.

In the following lemmas, we prove that the execution of each thread is serialized by phases. According to the definition in Section 3.2, for the initial abstract machine configuration we have:

$$\forall i.\ phase(c^0, i) = 1$$

The following series of lemmas can be trivially proved by the semantics and the definition of $phase$.

Lemma 4.26 (Uninterrupted Phase 1 Program Step Leads to Phase 2)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}_i} c' \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \rightarrow phase(c', i) = 2 \wedge \neg c'.p_{[i]}.jisr$$

Lemma 4.27 (Interrupted Phase 1 Program Step Leads to Phase 4)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}_i} c' \wedge phase(c, i) = 1 \wedge jisr_f(c.p_{[i]}, eev) \rightarrow phase(c', i) = 4 \wedge c'.p_{[i]}.jisr$$

Lemma 4.28 (Phase 2 Memory Step Leads to Phase 3)

$$c \overset{\text{m}}{\Longrightarrow}_i c' \wedge phase(c, i) = 2 \rightarrow phase(c', i) = 3 \wedge \neg c'.p_{[i]}.jisr$$

Lemma 4.29 (Phase 2 Page Fault Step Leads to Phase 1)

$$c \overset{\text{pf}}{\Longrightarrow}_i c' \wedge phase(c, i) = 2 \rightarrow phase(c', i) = 1$$

Lemma 4.30 (Uninterrupted Phase 3 Program Step Gen Instr Leads to Phase 4)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \wedge phase(c,i) = 3 \wedge \neg jisr_x(c.p_{[i]}, c.mode_{[i]}, I_{isa}) \wedge gen\text{-}ins(I_{isa}) \rightarrow$$
$$phase(c',i) = 4 \wedge \neg c'.p_{[i]}.jisr$$

Lemma 4.31 (Uninterrupted Phase 3 Program Step No Instr Gen Leads to Phase 5)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \wedge phase(c,i) = 3 \wedge \neg jisr_x(c.p_{[i]}, c.mode_{[i]}, I_{isa}) \wedge \neg gen\text{-}ins(I_{isa}) \rightarrow$$
$$phase(c',i) = 5 \wedge \neg c'.p_{[i]}.jisr$$

Lemma 4.32 (Interrupted Phase 3 Program Step Leads to Phase 4)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \wedge phase(c,i) = 3 \wedge jisr_x(c.p_{[i]}, c.mode_{[i]}, I_{isa} \rightarrow phase(c',i) = 4 \wedge c'.p_{[i]}.jisr$$

Lemma 4.33 (Non Jisr Phase 4 Memory Step Leads to Phase 5)

$$c \overset{\text{m}}{\Longrightarrow}_i c' \wedge phase(c,i) = 4 \wedge \neg c.p_{[i]}.jisr \rightarrow phase(c',i) = 5$$

Lemma 4.34 (Jisr Phase 4 Memory Step Leads to Phase 1)

$$c \overset{\text{m}}{\Longrightarrow}_i c' \wedge phase(c,i) = 4 \wedge c.p_{[i]}.jisr \rightarrow phase(c',i) = 1$$

Lemma 4.35 (Phase 4 Page Fault Step Leads to Phase 1)

$$c \overset{\text{pf}}{\Longrightarrow}_i c' \wedge phase(c,i) = 4 \rightarrow phase(c',i) = 1$$

Lemma 4.36 (Phase 5 Program Step Leads to Phase 1)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \wedge phase(c,i) = 5 \rightarrow phase(c',i) = 1$$

In the following two lemmas, we prove the phase of the thread i can not be changed by other threads and the MMU steps of thread i.

Lemma 4.37 (Phase Maintained by Other's Step)

$$c \underset{\text{eev}}{\Longrightarrow}_j c' \rightarrow \forall i \neq j.\ phase(c,i) = phase(c',i)$$

Proof The phase of thread i in machine configuration c only depends on the program state, instruction sequence and the temporary that are all thread-local components and can not be affected by the execution of other threads. □

Lemma 4.38 (Phase Maintained by MMU Step)

$$c \overset{\text{mu}}{\Longrightarrow}_i c' \rightarrow phase(c,i) = phase(c',i)$$

Proof From the semantics of the MMU step, we know that MMU steps only change the MMU state and the memory (for MMU writes). The phase of each thread only depends on the program state, instruction sequence and the temporary. Thus, the phase can not be changed by the MMU steps of thread i. □

With Lemma 4.26 to Lemma 4.36, we can get the execution of the abstract machine is serialized. With Lemma 4.37 and Lemma 4.38, we know that reordering does not affect the phases. Thus, after reordering, we can still perform the steps of the same phase.

In the following, we prove the one step reordering lemmas.

Lemma 4.39 (Program Steps Switchable with Others) In the first case, in configuration c, the thread i performs a program step and the thread j performs a step, which can be all kinds of possible steps, and get a configuration c''. In the second case, in configuration c, the thread j performs an identical step as in the previous case, then the thread i performs a program step, and get a configuration c^2. We need to prove that $c'' = c^2$

$$\forall i \neq j, x \in \{m, p, muc, muw, mur\}.\ c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \underset{\text{eev'}}{\overset{\text{x}}{\Longrightarrow}}_j c'' \wedge c \underset{\text{eev'}}{\overset{\text{x}}{\Longrightarrow}}_j c^1 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^2 \rightarrow c'' = c^2$$

Proof From the semantics of program step we have:

$$c'.p_{[i]} = \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).p$$
$$c'.is_{[i]} = c.is_{[i]} \circ \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).is$$

For other thread-local components of thread i and the global components, we have

$$\forall Y \in \{\vartheta, mmu, \mathcal{D}, \mathcal{O}, pt, mode, rls_l, rls_s, rls_{pt}\}.\ c.Y_{[i]} = c'.Y_{[i]}$$
$$\forall l \neq i.\ \forall X \in \{m, shared, ro, ts[l]\}.\ c.X = c'.X$$

Thus, we can have the thread-local configuration of thread j $ts[j]$ satisfies:

$$c'.ts[j] = c.ts[j]$$

We can conclude that if the thread j performs a memory step from c' to c'' then thread j also execute the same instruction and can have the same address translation from c to c^1. If the instruction is RMW, we can also get that the condition is equal. Also from the definition of og function, we can get that c' and c can use the same ownership annotations to transfer the ownership in this case. If the thread j performs a program step from c' to c'' then thread j also perform the same program step from c to c^1 and get the same program state and the new instruction sequence. Analogously, for a page fault step and an MMU step, we can get the same

results. After that, we can have

$$\begin{aligned} c''.ts[j] &= c^1.ts[j] \\ c''.m &= c^1.m \\ c''.ro &= c^1.ro \\ c''.shared &= c^1.shared \end{aligned}$$

Since the step of thread j or i does not affect the thread-local components of other threads, we also have

$$\forall k \notin \{j, i\}.\ c.ts[k] = c''.ts[k] = c^2.ts[k]$$

and

$$\begin{aligned} c'.ts[i] &= c''.ts[i] \\ c.ts[i] &= c^1.ts[i] \\ c^1.ts[j] &= c^2.ts[j] = c''.ts[j] \end{aligned}$$

In the following we have to prove:

$$\begin{aligned} c''.ts[i] &= c^2.ts[i] \\ c''.m &= c^2.m \\ c''.ro &= c^2.ro \\ c''.shared &= c^2.shared \end{aligned}$$

From the semantics of the program step of thread i, we have

$$\begin{aligned} c^2.p_{[i]} &= \delta_p(c^1.p_{[i]}, c^1.\vartheta_{[i]}, c^1.mode_{[i]}, c^1.mmu_{[i]}, c^1.is_{[i]}, eev).p \\ &= \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).p \\ &= c'.p_{[i]} \\ &= c''.p_{[i]} \\ c^2.is_{[i]} &= \delta_p(c^1.p_{[i]}, c^1.\vartheta_{[i]}, c^1.mode_{[i]}, c^1.mmu_{[i]}, c^1.is_{[i]}, eev).is \\ &= \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).is \\ &= c'.is_{[i]} \\ &= c''.is_{[i]} \\ c^2.Y_{[i]} &= c^1.Y_{[i]} = c''.Y_{[i]} \\ c^2.m &= c^1.m = c''.m \\ c^2.ro &= c^1.ro = c''.ro \\ c^2.shared &= c^1.shared = c''.shared \end{aligned}$$

This concludes the proof. □

Lemma 4.40 (MMU Step Switchable with Program Step) In configuration c, the thread i first makes a program step then makes an MMU step, which equals the thread i first makes an identical MMU step with identical address then makes an identical program step as before.

$$\forall x \in \{muc, mur, muw\}.\ c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i\ c' \overset{\text{x}}{\Longrightarrow}_i c'' \wedge c \overset{\text{x}}{\Longrightarrow}_i c^1\ \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i\ c^2 \rightarrow c'' = c^2$$

Proof From the semantics of the program step, we have:

$$\begin{aligned} c'.p_{[i]} &= \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).p \\ c'.is_{[i]} &= c.is_{[i]} \circ \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).is \end{aligned}$$

$$\begin{aligned} \forall Y \in \{\vartheta, mmu, \mathcal{D}, \mathcal{O}, pt, mode, rls_l, rls_s, rls_{pt}\}.\ c.Y_{[i]} = c'.Y_{[i]} \\ \forall j \neq i.\ \forall X \in \{m, shared, ro, ts[j]\}.\ c.X = c'.X \end{aligned}$$

Then we make a case split on the type of MMU step:

- Walk creation. From the semantics, we have

  $$\begin{aligned} c''.mmu_{[i]} &= \delta_{crtw}(c'.mmu_{[i]}, va) \\ &= \delta_{crtw}(c.mmu_{[i]}, va) \\ &= c^1.mmu_{[i]} \end{aligned}$$

  From the semantics, we also have:

  $$\begin{aligned} c.mmu_{[i]}.pto &= c'.mmu_{[i]}.pto \\ &= c''.mmu_{[i]}.pto \\ &= c^1.mmu_{[i]}.pto \end{aligned}$$

  For other components of thread i, we have

  $$\begin{aligned} &\forall Z \in \{\vartheta, \mathcal{D}, \mathcal{O}, pt, mode, rls_l, rls_s, rls_{pt}\}. \\ &\qquad c''.Z_{[i]} = c'.Z_{[i]} = c.Z_{[i]} = c^1.Z_{[i]} \\ &c.p_{[i]} = c^1.p_{[i]} \\ &c.is_{[i]} = c^1.is_{[i]} \\ &c.mode_{[i]} = c^1.mode_{[i]} \\ &c'.p_{[i]} = c''.p_{[i]} \\ &c'.is_{[i]} = c''.is_{[i]} \\ &c'.mode_{[i]} = c''.mode_{[i]} \end{aligned}$$

For the global components we have:

$$c''.X = c'.X = c.X = c^1.X$$

From the semantics of the program step, we have:

$$c^2.p_{[i]} = \delta_p(c^1.p_{[i]}, c^1.\vartheta_{[i]}, c^1.mode_{[i]}, c^1.mmu_{[i]}, c^1.is_{[i]}, eev).p$$
$$c^2.is_{[i]} = c^1.is_{[i]} \circ \delta_p(c^1.p_{[i]}, c^1.\vartheta_{[i]}, c^1.mode_{[i]}, c^1.mmu_{[i]}, c^1.is_{[i]}, eev).is$$

With the definition of δ_p in Section 3.2.3, we know that the execution of program step does not depend on the TLB. Thus, we can conclude:

$$\begin{aligned} c^2.p_{[i]} &= \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).p \\ &= c'.p_{[i]} = c''.p_{[i]} \\ c^2.is_{[i]} &= c.is_{[i]} \circ \delta_p(c.p_{[i]}, c.\vartheta_{[i]}, c.mode_{[i]}, c.mmu_{[i]}, c.is_{[i]}, eev).is \\ &= c'.is_{[i]} = c''.is_{[i]} \\ c^2.Z_{[i]} &= c^1.Z_{[i]} = c''.Z_{[i]} \end{aligned}$$

The lemma is concluded in this case.

- MMU read. From the semantics, we have:

$$c''.mmu_{[i]} \in \delta_{mmur}(c'.mmu_{[i]}, pa, c'.m(pa))$$
$$c^1.mmu_{[i]} \in \delta_{mmur}(c.mmu_{[i]}, pa, c.m(pa))$$

We also have:

$$\delta_{mmur}(c'.mmu_{[i]}, pa, c'.m(pa)) = \delta_{mmur}(c.mmu_{[i]}, pa, c.m(pa))$$

Thus, we can choose the proper MMU state such that:

$$c''.mmu_{[i]} = c^1.mmu_{[i]}$$

By analogous steps of the previous case, we can conclude the lemma in this case.

- MMU write. From the semantics, we have

$$v' \in \delta_{mmw}(c'.mmu_{[i]}, pa, c'.m(pa))$$
$$v^1 \in \delta_{mmw}(c.mmu_{[i]}, pa, c.m(pa))$$

We also have

$$\delta_{mmw}(c'.mmu_{[i]}, pa, c'.m(pa)) = \delta_{mmw}(c.mmu_{[i]}, pa, c.m(pa))$$

Thus, we can choose the proper value v such that:

$$v' = v^1$$

Also, we have

$$c'.m = c.m$$

Then we can get

$$\begin{aligned} c''.m &= c'.m(pa \mapsto v') \\ &= c'.m(pa \mapsto v^1) \\ &= c.m(pa \mapsto v^1) \\ &= c^1.m \\ &= c^2.m \qquad \text{(semantics of program step)} \end{aligned}$$

By the semantics of the program step and the MMU write step, for the MMU state of thread i, we have:

$$c.mmu_{[i]} = c'.mmu_{[i]} = c''.mmu_{[i]} = c^1.mmu_{[i]} = c^2.mmu_{[i]}$$

With analogous steps of the previous cases, we can prove the equivalence of other components and concludes the lemma.

□

Lemma 4.41 (Phase 2 Memory Step Switchable with Others) In configuration c, the thread i first performs a phase 2 memory step then the thread j performs a step that equals to the thread j first performs an identical step as before and then the thread i performs a phase 2 memory step.

$$\begin{aligned} &\forall i \neq j, x \in \{m, p, muc, mur, muw\}.\ phase(c, i) = 2 \wedge \\ &\quad c \overset{\text{m}}{\Longrightarrow}_i c' \ \underset{\text{eev'}}{\overset{\text{x}}{\Longrightarrow}}_j \ c'' \wedge c \ \underset{\text{eev'}}{\overset{\text{x}}{\Longrightarrow}}_j \ c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \rightarrow c'' = c^2 \end{aligned}$$

Proof In the proof, the only interesting case is when the thread j performs a memory step to update the memory, or an MMU write step. We make a case split here:

- $c' \overset{\text{m}}{\Longrightarrow}_j c''$. With semantics of the abstract machine, we know that the phase 2 memory step of thread i does not change the thread-local component of thread j and the memory. As a consequence, from c to c^1, the thread j can make the same step with the same physical address and the same condition for RMW. In this case, we let

$$\begin{aligned} I^j &= hd(c'.is_{[j]}) = hd(c.is_{[j]}) \\ I^i &= hd(c.is_{[i]}) = hd(c^1.is_{[i]}) \\ pa^j &\in atran(c'.mmu_{[j]}, I^j.va, c'.mode_{[j]}, I^j.r) \\ &= atran(c.mmu_{[j]}, I^j.va, c.mode_{[j]}, I^j.r) \end{aligned}$$

From the previous argument, we have

$$W(I^j) \vee RMW(I^j) \wedge I^j.cond(c'.\vartheta_{[j]}(I^j.t \mapsto c'.m(pa^j)))$$

and

$$I^j.cond(c.\vartheta_{[j]}(I^j.t \mapsto c.m(pa^j))$$

From Definition 4.22, we have

$$pa^j \notin c'.ro$$

With Definition 4.23, we know

$$\begin{aligned} &atran(c.mmu_{[i]}, I^i.va, c.mode_{[i]}, I^i.r) \\ &= atran(c^1.mmu_{[i]}, I^i.va, c^1.mode_{[i]}, I^i.r) \\ &\in c^1.ro \end{aligned}$$

With Definition 4.22, we have

$$c^1.ro = c'.ro$$

Thus, we can conclude that the phase 2 memory step of thread i and the thread j step are data race free. We can switch them and get the same read result. The proof of the equivalence of other components is trivially proved by the semantics.

- $c' \overset{muw}{\Longrightarrow}_j c''$. We let the target address of MMU write be a. From the Definition 4.22, we have:

$$a \notin c'.ro$$

With analogous prove steps as the previous case, we can conclude this lemma.

Lemma 4.42 (Phase 2 Memory Step Postpone After MMU Step) Each phase 2 memory step can be postponed after MMU steps of the same thread.

$$\forall x \in \{muc, muw, mur\}.\ c \overset{m}{\Longrightarrow}_i c' \overset{x}{\Longrightarrow}_i c'' \wedge phase(c, i) = 2 \rightarrow c \overset{x}{\Longrightarrow}_i c^1 \overset{m}{\Longrightarrow}_i c''$$

Proof Since the phase 2 memory step does not affect the *mode* and MMU state *mmu*. The same MMU step can be advanced. We make a case split on the type of MMU steps.

- $c' \overset{muc}{\Longrightarrow}_i c'' \vee c' \overset{mur}{\Longrightarrow}_i c''$. Because of the monotonicity, the postponed phase 2 memory step can choose the same translated address as the physical address. The equivalence of other components is trivially maintained by the semantics.

- $c' \overset{muw}{\Longrightarrow}_i c''$. Analogous to last case in the proof of Lemma 4.41, we can get that the phase 2 memory step and the MMU write step are data race free. We can postpone the phase 2 memory step after the MMU write step and get the same configuration.

□

According to the argument of reordering, we iteratively apply Lemma 4.39, Lemma 4.40, Lemma 4.41 and Lemma 4.42 to reorder the executions of the abstract machine into interleaving blocks. According to the semantics of the abstract machine (For detail see Section 3.2.), we have the following types of complete interleaving block of thread i. Note that, each block either contains only one MMU step or starts with a phase 1 program step.

1. $c \overset{\text{mu}}{\Longrightarrow}_i c'$. Each MMU step is an individual block.

2. $c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \overset{\text{m}}{\Longrightarrow}_i c''$. The thread i of the abstract machine first performs a phase 1 interrupted program step; then it performs a phase 4 memory step to switch the mode to 0. We have

$$phase(c, i) = 1 \wedge jisr_f(c.p_{[i]}, eev) \wedge phase(c', i) = 4 \wedge phase(c'', i) = 1$$

3. $c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c' \overset{\text{pf}}{\Longrightarrow}_i c''$. The thread i of the abstract machine first performs an uninterrupted phase 1 program step then a page fault on fetch happens and performs a page fault step. Also, we have

$$phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge phase(c', i) = 2 \wedge phase(c'', i) = 1$$

4. $c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{m}}{\Longrightarrow}_i c^4$. The thread i of the abstract machine first performs an uninterrupted phase 1 program step, then perform a phase 2 memory step for fetching. After that, the machine performs an interrupted phase 3 program step of thread i. At last, the machine performs a memory step to switch the mode to 0. According to the semantics, we have:

$$\begin{aligned} &phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge phase(c^1, i) = 2 \wedge \\ &phase(c^2, i) = 3 \wedge jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge \\ &phase(c^3, i) = 4 \wedge phase(c^4, i) = 1 \end{aligned}$$

5. $c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{pf}}{\Longrightarrow}_i c^4$. The thread i of the abstract machine first performs an uninterrupted phase 1 program step, then perform a phase 2 memory step for fetching. After that, the machine performs an uninterrupted phase 3 program step of thread i. At last, the machine performs a phase 4 page fault step. According to the semantics, we have:

$$\begin{aligned} &phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge phase(c^1, i) = 2 \wedge \\ &phase(c^2, i) = 3 \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge \\ &phase(c^3, i) = 4 \wedge phase(c^4, i) = 1 \end{aligned}$$

6. $c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^4$. The thread i of the abstract machine first performs an uninterrupted phase 1 program step and phase 2 memory step as in the previous case, then

performs an uninterrupted phase 3 program step and do not generate memory instructions. At last, the machine performs a phase 5 program step. We have:

$$phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge phase(c^1,i) = 2 \wedge$$
$$phase(c^2,i) = 3 \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge$$
$$phase(c^3,i) = 5 \wedge phase(c^4,i) = 1$$

7. $c \underset{eev}{\overset{p}{\Longrightarrow}}_i c^1 \overset{m}{\Longrightarrow}_i c^2 \underset{eev}{\overset{p}{\Longrightarrow}}_i c^3 \overset{m}{\Longrightarrow}_i c^4 \underset{eev}{\overset{p}{\Longrightarrow}}_i c^5$. As in the previous case, the machine first performs an uninterrupted phase 1 program step, phase 2 memory step, and uninterrupted phase 3 program step. A memory instruction is generated in the phase 3 program step. In the next step, a phase 4 memory step is performed to execute the instruction. At last, the machine performs a phase 5 program step. We have:

$$phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge phase(c^1,i) = 2 \wedge$$
$$phase(c^2,i) = 3 \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge$$
$$phase(c^3,i) = 4 \wedge phase(c^4,i) = 5 \wedge phase(c^5,i) = 1$$

8. Other complete interleaving blocks are forbidden by the semantics.

For simplicity, we assume that in the interleaving-reduced abstract machine computation, only exists complete interleaving blocks. We will discuss the simulation of incomplete blocks at the end of this section.

In the remaining of this thesis, we call these blocks type 1 block,...,type 7 block. In the next section, we will prove that each block can be simulated by one step of SB reduced MIPS-86 *Cosmos* machine.

4.3.2 Simulation Theorem Between Abstract Machine and *Cosmos* Machine

In this subsection, we will prove the simulation between a reordered abstract machine computation and an SB reduced MIPS-86 *Cosmos* machine. First, we instantiate the safety property P in the safety condition of the *Cosmos* machine in Definition 4.19. Then, we will introduce the coupling relation and prove the simulation theorem. At last, we prove that the safety condition is transferred from the SB reduced MIPS-86 *Cosmos* machine to the abstract machine.

Safety Property Instantiation

In order to make the simulation go through, we should prove that the safety properties are transfered from the *Cosmos* machine to the abstract machine. Compare with the safety condition of the *Cosmos* machine, the safety properties of the abstract machine contains two extra properties: (i) the flushing policy: the dirty bit should be cleared before a volatile read. (ii) the safety property for MMU steps: the MMU steps should only access the corresponding local page table and shared-writable memory. In this subsection, we will complement the safety condition of the *Cosmos* machine by instantiate the predicate P in $safety(C, P)$ in Definition 4.19.

Moreover, to prove the safety transferred from the *Cosmos* machine computation to the abstract machine computation, we should obtain the ownership annotations for the abstract machine from the ownership annotations for the *Cosmos* machine. Since the ownership annotations in the abstract machine computation are generated by ownership annotation generation function og, we need to give the definition of og. For a certain annotated program, the ownership annotations are fixed. Thus, the ownership annotation only depends on the program counter and read result in ISA level. The ownership generation functions can be regarded as abstractions of the fixed ownership annotations. The *pc* in the processor core gives us the location of the program, and the temporary gives us the read result. The idea of ownership generation comes from [CS10b], in which the ownership annotation is generated out of program states and temporaries. Hence, it is hard to find out a formula which describes og. Instead, we define the function og constructively, i.e. we define the value of og for every reachable abstract machine configuration. In the instantiation of the predicate P, we introduce a function og_{cos}^{MIPS} which takes a SB reduced MIPS-86 core configuration and a temporary, and returns a tuple of ownership annotation for the SB reduced MIPS-86 *Cosmos* machine. We use the value of og_{cos}^{MIPS} to define the value of og of the abstract machine. The details of the og definition is stated in Lemma 4.64 and the transfer of safety property is proved in Theorem 4.66.

Let $i = \alpha.s$ and $\alpha.annot_{cos} = (\alpha.A, \alpha.L, \alpha.R, \alpha.A_{pt}, \alpha.R_{pt})$ then

$$
\begin{aligned}
&P_{og_{cos}^{\text{MIPS}}}(C) \equiv \\
&\quad (\alpha.io \wedge \alpha.in = (\textbf{core}, w_I, w_R, eev) \ \rightarrow \\
&\qquad (i) \quad (load(I_{isa}) \rightarrow \neg C.u_i.\mathcal{D}) \wedge \\
&\qquad (ii) \quad \alpha.annot_{cos} = og_{cos}^{\text{MIPS}}(C.u_i.core, \vartheta'_{cos})) \wedge \\
&\quad (\exists w \in C.u_i.tlb.\ \neg complete(w) \ \rightarrow \\
&\qquad (iii) \quad ptea(w) \in C.\mathcal{P}t_i \cup C.\mathcal{G}.\mathcal{S} \wedge \forall j.\ ptea(w) \notin C.\mathcal{O}_j)
\end{aligned}
$$

where I_{isa} is the instruction execute in step α.

$$\vartheta'_{cos} = S^n_{\text{MIPS-86}}.\delta(C.u_i, C.m, \alpha.in).u_i.\vartheta$$

$(i), (ii)$ together with *Cosmos* machine machine ownership policy corresponds to $safe\text{-}instr$ in Definition 2.14. (iii) is a counter part to $safe\text{-}mmu\text{-}acc$ in Definition 2.15. Note that, we only cache non-faulty walks in the TLB, therefore, only non-faulty walks are considered in (iii).

Coupling Relation

In this section, we define the coupling relation between the abstract machine configuration and the *Cosmos* machine configuration. In the coupling relation, each component of the abstract machine equals to the corresponding component of the *Cosmos* machine.

Definition 4.43 (Coupling Relation Between Abstract Machine and *Cosmos* machine) We define the coupling relation $c \sim C$ for an abstract machine configuration c and a *Cosmos* machine configuration C.

- For global components, we have:

$$\begin{aligned} c.m &= C.m \\ c.shared \setminus c.ro &= C.\mathcal{S} \\ c.ro &= \mathcal{R} \end{aligned}$$

- For thread-local components, $\forall i \in [0 : np - 1]$ we have:

$$\begin{aligned} \forall X \in \{n, pc, gpr, spr_p\}.\ & c.p_{[i]}.X = C.u_i.X \\ & c.\vartheta_{[i]} = C.u_i.\vartheta \\ & c.mmu_{[i]}.pto = C.u_i.spr(pto) \\ & c.mmu_{[i]}.tlb = C.u_i.tlb \\ & c.mode_{[i]} = C.u_i.spr(mode)[0] \\ & c.\mathcal{D}_{[i]} = C.u_i.\mathcal{D} \\ & c.\mathcal{O}_{[i]} = C.\mathcal{O}_i \\ & c.pt_{[i]} = C.\mathcal{P}_i \end{aligned}$$

Simulation Theorem

To prove the simulation between an interleaving-reduced abstract machine computation and a *Cosmos* machine computation inductively, we have to prove the simulation theorem for each block. Then, we prove that the safety condition is transferred from the *Cosmos* machine computation to the abstract machine computation.

According to the argument in Section 4.3.1, we have 6 kinds of complete blocks. The following series of lemmas give us the simulation between one block of abstract machine execution and one step of *Cosmos* machine execution.

Coupling Maintained by Type 1 Block

Lemma 4.44 (Type 1 Block Simulate By *Cosmos* machine) Each type 1 block, which only contains one MMU step, can be simulated by a step of *Cosmos* machine.

$$c \overset{\text{mu}}{\Longrightarrow}_i c' \wedge c \sim C \rightarrow \exists \alpha.\ C \overset{\alpha}{\mapsto} C' \wedge c' \sim C'$$

Proof Since the abstract machine makes an MMU step, from the semantics, we can conclude that:

$$c.mode_{[i]}$$

From the coupling relation, we can also conclude:

$$C.u_i.spr(mode)[0]$$

We do a case split on the type of the MMU step.

- A walk creation step for address $va \in \mathbb{B}^{30}$. In this step, the MMU creates a walk for address va. We also have

$$C.u_i.spr(mode)[0]$$

Thus, the *Cosmos* machine can perform a **tlb-create** step to creating the same walk. From the coupling relation, we also have

$$c.mmu_{[i]}.pto = C.u_i.spr(pto)$$

We let

$$\alpha = (i, (\textbf{tlb-create}, va.ba), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

in which

$$io = \mathcal{IO}_i(C, \alpha.in) \wedge ip = \mathcal{IP}_i(C, \alpha.in)$$

In the remaining of this chapter, the io and ip flags are always defined as above with respect to the specific α and C. From the semantics of the abstract machine, we have the new walk

$$\begin{aligned} w &= winit(va.ba, c.mmu_{[i]}.pto[31:12]) \\ &= winit(va.ba, C.u_i.spr(pto)[31:12]) \end{aligned}$$

which is also the new walk of the *Cosmos* machine. Thus, coupling for TLB is maintained and for other components are trivially maintained.

- An MMU read step. In this step, the MMU non-deterministically choose a walk w to extend. Let $pte = pte(c.m, w)$ then from the definition of *can-access* and δ_{mmur}, we have

$$\begin{aligned} &\exists w \in c.mmu_{[i]}.tlb.\ \neg complete(w) \wedge pte.p \wedge \\ &\quad pte.a \wedge (w.level = 1 \wedge w.r[0] \wedge pte.r[0] \rightarrow pte.d) \end{aligned}$$

With the coupling relation we have $pte = pte(C.m, w)$ and

$$\begin{aligned} &w \in C.u_i.tlb.\ \neg complete(w) \wedge pte.p \wedge \\ &\quad pte.a \wedge (w.level = 1 \wedge w.r[0] \wedge pte.r[0] \rightarrow pte.d) \end{aligned}$$

Thus, in the *Cosmos* machine, we can choose the same walk to perform the same walk extension. We let

$$\alpha = (i, (\textbf{tlb-extend}, w), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

From the semantics of the abstract machine and the *Cosmos* machine, the coupling for TLB is maintained. For other components, the coupling relation is trivially maintained.

- An MMU write step. In this step, the MMU non-deterministically choose an incomplete walk w and set the access and dirty bit for $c.m(ptea(w))$. From the definition of *can-access* and δ_{mmuw}, we have

$$\exists w \in c.mmu_{[i]}.tlb.\ \neg complete(w) \wedge pte(c.m, w).p$$

With the coupling relation we have:

$$w \in C.u_i.tlb.\ \neg complete(w) \wedge pte(C.m, w).p$$

Thus, we can choose the same walk w from TLB in the *Cosmos* machine to set the access and dirty bit at the address for PTE $c.m(ptea(w))$. With the coupling relation we have

$$c.m(ptea(w)) = C.m(ptea(w))$$

We let

$$\alpha = (i, (\textbf{tlb-set-accessed-dirty}, w), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

From the semantics of the abstract machine and the *Cosmos* machine, the coupling for memory is maintained. For other components, the coupling relation is trivially maintained.

□

Coupling Maintained by Type 2 Block In a type 2 block, the thread i of the abstract machine first performs an interrupted phase 1 program step then makes a phase 4 memory step. This kind of block can be simulated by one step of the *Cosmos* machine. To prove the simulation, we have to prove that the same level of interrupt also can happen in the *Cosmos* machine.

Lemma 4.45 (Ca on Fetch Identical)

$$c \sim C \rightarrow ca_f(c.p_{[i]}, eev) = ca_f(C.u_i.p.core, eev, 0)$$

Proof With the coupling relation we have

$$C.u_i.pc = c.p_{[i]}.pc$$

From the definition of ca_f, we can conclude this lemma. □

Lemma 4.46 (Mca on Fetch Identical)

$$c \sim C \rightarrow mca_f(c.p_{[i]}, eev) = mca_f(C.u_i.core, eev, 0)$$

Proof The coupling relation gives us:

$$c.p_{[i]}.spr_p(sr) = C.u_i.spr_p(sr)$$

This lemma can be concluded by the definition of mca_f and Lemma 4.45. □

Lemma 4.47 (Interrupt on Fetch Occur in Both Machines)

$$c \xRightarrow[\text{eev}]{\text{p}}_i c' \wedge phase1(c, i) \wedge c \sim C \wedge jisr_f(c.p_{[i]}, eev) \rightarrow jisr_f(C.u_i, eev, 0)$$

Proof This lemma can be proved by the definition of $jisr_f$ and Lemma 4.46. □

Lemma 4.48 (Interrupt Level on Fetch Identical)

$$c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c' \wedge phase1(c,i) \wedge c \sim C \wedge jisr_f(c.p_{[i]}, eev) \rightarrow$$
$$il_f(c.p_{[i]}, eev) = il_f(C.u_i.core, eev, pff)$$

Proof From Lemma 4.46, we know that

$$mca_f(c.p_{[i]}, eev) = mca_f(C.u_i.core, eev, 0)$$

With the definition of $jisr_f$, we can conclude:

$$mca_f(c.p_{[i]}, eev) = mca_f(C.u_i.core, eev, 0) \neq 0^{32}$$

Since the page fault on fetch has the lowest priority among the interrupts in fetch phase, according to the definition of il_f, for any flag *pff*, the abstract machine and the *Cosmos* machine handles the same level of interrupt. □

Lemma 4.49 (Type 2 Block Simulate by *Cosmos* machine)

$$c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c' \overset{\text{m}}{\Longrightarrow}_i c'' \wedge phase(c,i) = 1 \wedge jisr_f(c.p_{[i]}, eev) \wedge$$
$$phase(c',i) = 4 \wedge phase(c'',i) = 1 \wedge c \sim C \rightarrow$$
$$\exists \alpha.\ C \overset{\alpha}{\mapsto} C' \wedge c'' \sim C'$$

Proof With Lemma 4.27 and Lemma 4.34, we have

$$phase(c',i) = 4 \wedge phase(c'',i) = 1$$

We can get the above computation of abstract machine is a type 2 interleaving block. We let

$$\alpha = (i, (\textbf{core}, \bot, \bot, eev), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

With Lemma 4.47, we know that the *Cosmos* machine is also interrupted by make a step α. With the definition of the page fault on fetch flag *pff* in Definition 3.29, we know that $pff = 0$. With Lemma 4.48, we have:

$$il_f(c.p_{[i]}, eev) = il_f(C.u_i.core, eev, 0)$$

From the semantics of the abstract machine, we let

$$p' = \delta_{jisr_f}(cast(c.p_{[i]}, zxt_{32}(c.mode_{[i]})), eev, 0)$$

then

$$\begin{aligned}
c'.p_{[i]}.pc &= p'.pc = 0^{32} \\
&= C'.u_i.pc \\
c'.p_{[i]}.n &= c.p_{[i]}.n + 1 \\
&= C.u_i.n + 1 && \text{(coupling relation)} \\
&= C'.u_i.n && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c'.p_{[i]}.spr_p &= p'.spr_p
\end{aligned}$$

in which let $k = min\{j \mid eev[j] = 1\}$ then

$$p'.spr_p(x) = \begin{cases} 0^{32} & x = sr \\ zxt_{32}(c.mode_{[i]}) & x = emode \\ c.p_{[i]}.spr_p(sr) & x = esr \\ mca_f(c.p_{[i]}, eev) & x = eca \\ c.p_{[i]}.pc & x = epc \\ bin_{32}(k) & x = edata \wedge il_f(c.p_{[i]}, eev) = 1 \\ c.p_{[i]}.spr_p(x) & otherwise \end{cases}$$

$$= \begin{cases} 0^{32} & x = sr \\ C.u_i.spr(mode) & x = emode \\ C.u_i.spr(sr) & x = esr \\ mca_f(C.u_i.core, eev, 0) & x = eca \\ C.u_i.pc & x = epc \\ bin_{32}(k) & x = edata \wedge il_f(C.u_i.core, eev, 0) = 1 \\ C.u_i.spr_p(x) & otherwise \end{cases}$$

$$= C'.u_i.spr_p(x)$$

The above equation can be trivially obtained via the semantics of the *Cosmos* machine, the coupling relation, Lemma 4.46 and Lemma 4.48. With the semantics of the abstract machine, we have

$$c'.is_{[i]} = [\textbf{SWITCH}\ 0]$$

and

$$c''.mode_{[i]} = 0 \wedge c''.is_{[i]} = []$$

Other components of c'' equals to the corresponding components of c'. By the semantics of *Cosmos* machine, we have

$$C'.u_i.spr(mode)[0] = 0 = c''.mode_{[i]}$$

The coupling of other components is trivially maintained by the semantics. □

Coupling Maintained by Type 3 Block In a type 3 block, the thread i of the abstract machine first performs an uninterrupted phase 1 program step then makes a phase 2 page fault step. This kind of block can be simulated by one step of the *Cosmos* machine. To prove the simulation, first, we have to prove that when the abstract machine makes a phase 2 page fault step, then the *Cosmos* machine can also have a page fault on fetch.

Lemma 4.50 (Page Fault On Fetch Sync) We let $trqI = (C.u_i.pc[31:2], 011)$ in

$$c \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c' \overset{\text{pf}}{\Longrightarrow_i} c'' \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge c \sim C \rightarrow$$
$$\exists w_I \in C.u_i.tlb.\ C.u_i.spr(mode)[0] \wedge fault(pte(C.m, w_I), trqI, w_I)$$

Proof With the semantics of the page fault step of the abstract machine, we have

$$c'.mode_{[i]}$$

Since the program step does not change the *mode* component, we have

$$\begin{aligned} c'.mode_{[i]} &= c.mode_{[i]} \\ &= C.u_i.spr(mode)[0] \qquad \text{(coupling relation)} \end{aligned}$$

From semantics of the abstract machine, we let $I = I'$ then

$$nvR(I) \wedge I.va = c.p_{[i]}.pc[31:2] \wedge I.r = 011$$

With the definition of *can-page-fault* we have

$$\exists w \in c'.mmu_{[i]}.tlb.\ fault(pte(c'.m, w), (I.va, 011), w)$$

According to the semantics of the abstract machine, the program step does not change the MMU state and the memory. Thus, we have

$$\begin{aligned} c'.mmu_{[i]} &= c.mmu_{[i]} \\ c'.m &= c.m \end{aligned}$$

From the coupling relation we have

$$\begin{aligned} C.u_i.pc &= c.p_{[i]}.pc \\ C.u_i.tlb &= c.mmu_{[i]}.tlb \\ C.m &= c.m \end{aligned}$$

Thus, we can conclude

$$w \in C.u_i.tlb.\ C.u_i.spr(mode)[0] \wedge fault(pte(C.m, w), trqI, w)$$

□

Lemma 4.51 (Type 3 Block Simulate by *Cosmos* machine)

$$c \overset{\substack{\mathrm{p} \\ \mathrm{eev}}}{\Longrightarrow}_i c' \overset{\mathrm{pf}}{\Longrightarrow}_i c'' \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge c \sim C \rightarrow$$
$$\exists \alpha.\ C \overset{\alpha}{\mapsto} C' \wedge c'' \sim C'$$

Proof With Lemma 4.26 and Lemma 4.29, we have:

$$phase(c', i) = 2 \wedge phase(c'', i) = 1$$

Thus, we can get the above abstract machine computation is a type 3 interleaving block. By the definition of $jisr_f$ and $phase(c, i) = 1$, we can conclude:

$$\bigvee_j mca_f(c.p_{[i]}, eev)[j] = 0$$

With Lemma 4.46, we can conclude:

$$mca_f(C.u_i.core, eev, 0) = 0^{32}$$

Which means all the interrupts with higher priority than page fault on fetch can not occur in the *Cosmos* machine. With Lemma 4.50, we can choose the same walk w_I to signal a page fault on fetch in the abstract machine and the *Cosmos* machine. We let

$$\alpha = (i, (\textbf{core}, w_I, \perp, eev), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

From the semantics of the abstract machine, we have

$$c'.mode_{[i]}$$

Also, the phase 1 program step does not change the *mode*. Then, we have

$$\begin{aligned} c'.mode_{[i]} &= 1 \\ &= c.mode_{[i]} \\ &= C.u_i.spr(mode)[0] && \text{(coupling relation)} \end{aligned}$$

Thus, the page fault on fetch flag *pff* in the *Cosmos* machine is 1. By the definition of mca_f, we have

$$mca_f(C.u_i.core, eev, 1) = 0^3 10^{28}$$

We also have

$$jisr_f(C.u_i.core, eev, 1) = 1 \wedge il_f(C.u_i.core, eev, 1) = 3$$

Therefore, both machine can have the same level of interrupt. From the semantics of the abstract machine, we have

$$\begin{aligned} c''.p_{[i]}.pc &= 0^{32} \\ &= C'.u_i.pc && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\ c''.p_{[i]}.n &= c.p_{[i]}.n + 1 \\ &= C.u_i.n + 1 && \text{(coupling relation)} \\ &= C'.u_i.n && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\ c''.p_{[i]}.gpr &= c.p_{[i]}.gpr \\ &= C.u_i.gpr && \text{(coupling relation)} \\ &= C'.u_i.gpr && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\ c''.mode_{[i]} &= 0 \\ &= C'.u_i.spr(mode)[0] && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\ c''.p_{[i]}.spr_p &= spr'_p \end{aligned}$$

in which

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ c'.mode_{[i]} & x = emode \\ c'.p_{[i]}.spr_p(sr) & x = esr \\ 0^3 10^{28} & x = eca \\ c'.p_{[i]}.pc & x = epc \\ c'.p.spr_p(x) & otherwise \end{cases}$$

Since the uninterrupted phase 1 program step does not change the spr_p, pc and $mode$, we can get

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ c.mode_{[i]} & x = emode \\ c.p_{[i]}.spr_p(sr) & x = esr \\ 0^3 10^{28} & x = eca \\ c.p_{[i]}.pc & x = epc \\ c.p.spr_p(x) & otherwise \end{cases}$$

With the coupling relation, we can have

$$\begin{aligned} spr'_p(x) &= \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ C.u_i.spr(mode)[0] & x = emode \\ C.u_i.spr(sr) & x = esr \\ 0^3 10^{28} & x = eca \\ C.u_i.pc & x = epc \\ C.u_i.spr_p(x) & otherwise \end{cases} \\ &= C'.u_i.spr_p(x) \end{aligned}$$

The fault walks are erased from the TLB.

$$\begin{aligned} c''.mmu_{[i]}.tlb &= \delta_{flush}(c'.mmu_{[i]}, \{c'.p_{[i]}.pc\}) && \text{(semantics of abs)} \\ &= c'.mmu_{[i]}.tlb \setminus \{w \mid w.va = c'.p_{[i]}.pc[31:12]\} && \text{(def. of } \delta_{flush}\text{)} \\ &= c.mmu_{[i]}.tlb \setminus \{w \mid w.va = c.p_{[i]}.pc[31:12]\} && \text{(semantics of abs)} \\ &= C.u_i.tlb \setminus \{w \mid w.va = C.u_i.pc[31:12]\} && \text{(coupling relation)} \\ &= \delta_{tlb}(C.u_i.tlb, (\textbf{flush}, C.u_i.pc[31:12])) && \text{(def. of } \delta_{tlb}\text{)} \\ &= C'.u_i.tlb && \text{(semantics of cosmos)} \end{aligned}$$

The coupling of other components is trivially maintained. □

Coupling Maintained by Type 4 Block In a type 4 block, the abstract machine first performs an uninterrupted phase 1 program step, then perform a phase 2 memory step for fetching. After that, the machine performs an interrupted phase 3 program step of thread i. At last, the machine performs a memory step to switch the mode to 0.

To prove the simulation, we first have to prove that the abstract machine and the *Cosmos* machine fetch the same instruction.

Lemma 4.52 (Instruction Identical in Translated Mode) We let

$$pa_f^{cos} = w_I.ba \circ C.u_i.pc[11:2]$$

then

$$c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge c \sim C \wedge c.mode_{[i]} \rightarrow$$
$$\exists w_I \in C.u_i.tlb.\ complete(w_I) \wedge hit((C.u_i.pc[31:2], 011), w_I) \wedge I^2_{isa} = C.m(pa_f^{cos})$$

Proof With Lemma 4.26, we have

$$phase(c^1, i) = 2$$

From the semantics of phase 1 program step of the abstract machine, we have

$$nvR(I^1) \wedge I^1.va = c.p_{[i]}.pc[31:2] \wedge I^1.t = I_n \wedge$$
$$I^1.r = 011 \wedge c^1.mode_{[i]} \wedge c.mmu_{[i]} = c^1.mmu_{[i]}$$

From the semantics of the phase 2 memory step of the abstract machine, we have

$$\exists pmaI^2_{isa} \in atran(c^1.mmu_{[i]}, I^1.va, c^1.mode_{[i]}, I^1.r).\ I^2_{isa} = c^1.m(pmaI^2_{isa})$$

From the definition of *atran* and the coupling relation, we have:

$$\begin{aligned}
&atran(c^1.mmu_{[i]}, I^1.va, c^1.mode_{[i]}, I^1.r)\\
&= \{w.ba \circ I^1.va[9:0] \mid w \in c^1.mmu_{[i]}.tlb \wedge complete(w) \wedge hit((I^1.va, I^1.r), w)\}\\
&= \{w.ba \circ c.p_{[i]}.pc[11:2] \mid w \in c.mmu_{[i]}.tlb \wedge complete(w) \wedge hit((c.p_{[i]}.pc[31:2], 011), w)\}\\
&= \{w.ba \circ C.u_i.pc[11:2] \mid w \in C.u_i.tlb \wedge complete(w) \wedge hit((C.u_i.pc[31:2], 011), w)\}
\end{aligned}$$

Thus, we can choose a complete walk w_I from $C.u_i.tlb$, which satisfies

$$hit((C.u_i.pc[31:2], 011), w_I),$$

such that $pmaI^2_{isa} = pa_f^{cos}$. Since the phase 1 program step does not modify the memory, we have

$$c^1.m = c.m$$

With the coupling relation, we have

$$I^2_{isa} = C.m(pa_f^{cos})$$

□

Lemma 4.53 (Instruction Identical in Untranslated Mode)

$$c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge c \sim C \wedge \neg c.mode_{[i]} \wedge$$
$$\neg jisr_f(c.p_{[i]}, eev) \rightarrow I^2_{isa} = C.m(C.u_i.pc[31:2])$$

Proof With Lemma 4.26, we have

$$phase(c^1, i) = 2$$

From the semantics of phase 1 program step of the abstract machine, we have

$$nvR(I^1) \wedge I^1.va = c.p_{[i]}.pc[31:2] \wedge I^1.t = I_n \wedge I^1.r = 011 \wedge \neg c^1.mode_{[i]}$$

From the semantics of the phase 2 memory step of the abstract machine, we have

$$\exists pmaI^2_{isa} \in atran(c^1.mmu_{[i]}, I^1.va, c^1.mode_{[i]}, I^1.r).\ I^2_{isa} = c^1.m(pmaI^2_{isa})$$

From the definition of *atran* we have:

$$\neg c^1.mode_{[i]} \rightarrow pmaI^2_{isa} = c^1.p_{[i]}.pc[31:2]$$

By the semantics of the abstract machine, the uninterrupted program step does not change the *pc* and the memory. Therefore, we have:

$$pmaI^2_{isa} = c.p_{[i]}.pc[31:2]$$
$$c^1.m = c.m$$

With the coupling relation and the semantics of the abstract machine, we have:

$$\begin{aligned} I^2_{isa} &= c^1.m(pmaI^2_{isa}) \\ &= c.m(c.p_{[i]}.pc[31:2]) \\ &= C.m(C.u_i.pc[31:2]) \end{aligned}$$

□

Then, we need to prove the same level of interrupt happens in both machines.

Lemma 4.54 (Ca on Execute Identical)

$$\forall I'_{isa} \in \mathbb{B}^{32}.\ c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$c \sim C \rightarrow ca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) = ca_x(C.u_i.core, I'_{isa}, 0)$$

Proof With Lemma 4.26, we have:

$$phase(c^1, i) = 2$$

By the definition of ca_x, we have

$$ca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa})[j] =$$
$$\begin{cases} ill(I'_{isa}) \vee c^2.mode_{[i]} \wedge (movs2g(I'_{isa}) \vee movg2s(I'_{isa}) \vee eret(I'_{isa})) & j = 4 \\ sysc(I'_{isa}) & j = 5 \\ ovf(lop(cast(c^2.p_{[i]}), I'_{isa}), rop(cast(c^2.p_{[i]}), I'_{isa}), alucon(I'_{isa}), itype(I'_{isa})) & j = 6 \\ ea(cast(c^2.p_{[i]}), I'_{isa})[1:0] \notin \{00, \bot\} & j = 7 \\ 0 & otherwise \end{cases}$$

Since the result of *lop*, *rop* and *ea* only depend on the *gpr* value and the instruction. The phase 1 program step and the phase 2 memory step do not change the *gpr* value as well as the *mode* value. Thus, we have

$$ca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa})[j] =$$
$$\begin{cases} ill(I'_{isa}) \vee c.mode_{[i]} \wedge (movs2g(I'_{isa}) \vee movg2s(I'_{isa}) \vee eret(I'_{isa})) & j = 4 \\ sysc(I'_{isa}) & j = 5 \\ ovf(lop(cast(c.p_{[i]}), I'_{isa}), rop(cast(c.p_{[i]}), I'_{isa}), alucon(I'_{isa}), itype(I'_{isa})) & j = 6 \\ ea(cast(c.p_{[i]}), I'_{isa})[1:0] \notin \{00, \bot\} & j = 7 \\ 0 & otherwise \end{cases}$$

With the coupling relation, we have:

$$ca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa})[j] =$$
$$\begin{cases} ill(I'_{isa}) \vee C.u_i.spr(mode)[0] \wedge (movs2g(I'_{isa}) \vee movg2s(I'_{isa}) \vee eret(I'_{isa})) & j = 4 \\ sysc(I'_{isa}) & j = 5 \\ ovf(lop(C.u_i.core, I'_{isa}), rop(C.u_i.core, I'_{isa}), alucon(I'_{isa}), itype(I'_{isa})) & j = 6 \\ ea(C.u_i.core, I'_{isa})[1:0] \notin \{00, \bot\} & j = 7 \\ 0 & otherwise \end{cases}$$
$$= ca_x(C.u_i.core, I'_{isa}, 0)$$

□

Lemma 4.55 (Mca on Execute Identical)

$$\forall I'_{isa} \in \mathbb{B}^{32}.\ c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$c \sim C \rightarrow mca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) = mca_x(C.u_i.core, I'_{isa}, 0)$$

Proof With Lemma 4.54, we have:

$$ca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) = ca_x(C.u_i.core, I'_{isa}, 0)$$

From the semantics of the abstract machine we have

$$\begin{aligned} c^2.p_{[i]}.spr_p(sr) &= c.p_{[i]}.spr_p(sr) \\ &= C.u_i.spr(sr) \qquad \text{(coupling relation)} \end{aligned}$$

By the definition of the mca_x, we can conclude this lemma. □

Lemma 4.56 (Interrupt on Execute Occur in Both Machines)

$$\forall I'_{isa} \in \mathbb{B}^{32}.\ c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$c \sim C \rightarrow jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) = jisr_x(C.u_i, I'_{isa}, 0)$$

Proof This lemma can also be proved by the definition of $jisr_x$ and Lemma 4.55. □

Lemma 4.57 (Interrupt Level on Execute Identical)

$$\forall I'_{isa} \in \mathbb{B}^{32}.\ c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$c \sim C \rightarrow il_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) = il_x(C.u_i.core, I'_{isa}, 0)$$

Proof This lemma can be proved by the definition of il_x and Lemma 4.55. □

In the following, we prove a type 4 interleaving block can be simulated by a *Cosmos* machine step.

Lemma 4.58 (Type 4 Block Simulate by *Cosmos* machine)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{m}}{\Longrightarrow}_i c^4 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge c \sim C \rightarrow \exists \alpha.\ C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'$$

Proof With Lemma 4.26, Lemma 4.28, Lemma 4.32and Lemma 4.34, we have

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 4 \wedge phase(c^4, i) = 1$$

Thus, we can conclude that the above abstract machine computation is a type 4 interleaving block. We let

$$\alpha = (i, (\textbf{core}, w_I, \bot, eev), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

in which the w_I is chosen in the following manner: If $c.mode_{[i]}$ then w_I is chosen analogously to Lemma 4.52. Since w_I is complete and no rights violation, we do not have page fault on fetch in the *Cosmos* machine. Otherwise, we let w_I be $\bot$.

Also, because we give the $\perp$ value to the walk used for memory access, the page fault on load/store can not happen in *Cosmos* machine according to the semantics. Thus, the *pfls* flag is 0.

By Lemma 4.52 if $c.mode_{[i]}$ or Lemma 4.53 otherwise, we can conclude that the *Cosmos* machine fetches the same instruction as I^2_{isa}.

With Lemma 4.56, we have

$$jisr_x(C.u_i, I^2_{isa}, 0)$$

With Lemma 4.57, we have

$$il_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) = il_x(C.u_i.core, I^2_{isa}, 0)$$

From the semantics of the abstract machine, we have

$$\begin{aligned}
c^4.p_{[i]}.n &= c^3.p_{[i]}.n \\
&= c^2.p_{[i]}.n + 1 \\
&= c^1.p_{[i]}.n + 1 \\
&= c.p_{[i]}.n + 1 \\
&= C.u_i.n + 1 && \text{(coupling relation)} \\
&= C'.u_i.n && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c^4.p_{[i]}.pc &= 0^{32} \\
&= C'.u_i.pc && \text{(semantics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

Also, from the semantics of the abstract machine, we know an interrupted phase 3 program step generates a mode switch memory instruction to set the *mode* to 0. Thus, we can conclude:

$$\begin{aligned}
c^4.mode_{[i]} &= 0 \\
&= C'.u_i.spr(mode)[0] && \text{(semantics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

For the temporary, according to the semantics of the abstract machine, we have:

$$\begin{aligned}
c^4.\vartheta_{[i]} &= c^3.\vartheta_{[i]} \\
&= c^2.\vartheta_{[i]} \\
&= c^1.\vartheta_{[i]}(I_{n^2} \mapsto I^2_{isa}) \\
&= c.\vartheta_{[i]}(I_n \mapsto I^2_{isa}) \\
&= C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa}) && \text{(coupling relation)} \\
&= C'.u_i.\vartheta && \text{(semantics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

For the spr_p, according to the semantics of the abstract machine, we have:

$$\begin{aligned} c^4.p_{[i]}.spr_p &= c^3.p_{[i]}.spr_p \\ &= spr'_p \end{aligned}$$

in which

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c^2.mode_{[i]} & x = emode \\ c^2.p_{[i]}.spr_p(sr) & x = esr \\ mca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) & x = eca \\ c^2.p_{[i]}.pc & x = epc \wedge \neg continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ c^2.p_{[i]}.pc +_{32} 4_{32} & x = epc \wedge continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ ea(cast(c^2.p_{[i]}), I^2_{isa}) & x = edata \wedge il_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) = 8 \\ c^2.p_{[i]}.spr_p(x) & otherwise \end{cases}$$

From the semantics of the abstract machine, we have that an uninterrupted phase 1 program step and a phase 2 memory step does not change the gpr, spr_p, pc and $mode$. Thus, we have

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.mode_{[i]} & x = emode \\ c.p_{[i]}.spr_p(sr) & x = esr \\ mca_x(c^2.p_{[i]}, c^2.mode_{[i]}, I'_{isa}) & x = eca \\ c.p_{[i]}.pc & x = epc \wedge \neg continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ c.p_{[i]}.pc +_{32} 4_{32} & x = epc \wedge continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ c.p_{[i]}.spr_p(x) & otherwise \end{cases}$$

Since we already proved that both machines have same level of interrupt, then

$$continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \leftrightarrow continue(C.u_i.core, I^2_{isa}, 0)$$

With the coupling relation, we have

$$\begin{aligned} spr'_p(x) &= \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ C.u_i.spr(mode)[0] & x = emode \\ C.u_i.spr_p(sr) & x = esr \\ mca_x(C.u_i.core, I^2_{isa}, 0) & x = eca \\ C.u_i.pc & x = epc \wedge \neg continue(C.u_i.core, I^2_{isa}, 0) \\ C.u_i.pc +_{32} 4_{32} & x = epc \wedge continue(C.u_i.core, I^2_{isa}, 0) \\ C.u_i.spr_p(x) & otherwise \end{cases} \\ &= C'.u_i.spr_p(x) \end{aligned}$$

Let

$$gpr' = \delta_{instr}(cast(c^2.p_{[i]}, zxt_{32}(c^2.mode_{[i]}), c^2.mmu_{[i]}.pto), I^2_{isa}, \bot).gpr$$

then according to the semantics of the abstract machine, we have:

$$\begin{aligned} gpr' &= \delta_{instr}(cast(c.p_{[i]}, zxt_{32}(c.mode_{[i]}), c.mmu_{[i]}.pto), I^2_{isa}, \bot).gpr \\ &= \delta_{instr}(C.u_i.core, I^2_{isa}, \bot).gpr \end{aligned}$$

For the gpr, according to the coupling relation, semantics of the abstract machine and the *Cosmos* machine, we have

$$\begin{aligned} c^4.p_{[i]}.gpr &= c^3.p_{[i]}.gpr \\ &= \begin{cases} c^2.p_{[i]}.gpr & \neg continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ gpr' & otherwise \end{cases} \\ &= \begin{cases} c.p_{[i]}.gpr & \neg continue(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \\ gpr' & otherwise \end{cases} \\ &= \begin{cases} C.u_i.gpr & \neg continue(C.u_i.core, I^2_{isa}, 0) \\ gpr' & otherwise \end{cases} \\ &= C'.u_i.gpr \end{aligned}$$

The coupling for other components is trivially maintained. □

Coupling Maintained by Type 5 Block In a type 5 block, the abstract machine first performs an uninterrupted phase 1 program step, then perform a phase 2 memory step for fetching. After that, the machine performs an uninterrupted phase 3 program step of thread i. At last, the machine performs a page fault step.

To prove the simulation, we first prove that the page fault can also happen in the *Cosmos* machine.

Lemma 4.59 (Page Fault On Execute Sync) We let

$$trqEA = (ea(C.u_i.core, I^2_{isa})[31:2], store(I^2_{isa}) \vee rmw(I^2_{isa}) \circ 10)$$

in

$$\begin{aligned} c \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \overset{\underset{\text{eev}}{\text{p}}}{\Longrightarrow}_i c^3 \overset{\text{pf}}{\Longrightarrow}_i c^4 \wedge phase(c,i) = 1 \wedge \\ \neg jisr_f(c.p_{[i]}, eev) \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge c \sim C \rightarrow \\ \exists w_R \in C.u_i.tlb.\ C.u_i.spr(mode)[0] \wedge fault(pte(C.m, w_R), trqEA, w_R) \end{aligned}$$

Proof With the semantics of the page fault step in the abstract machine, we have

$$R(I^3) \vee W(I^3) \vee RMW(I^3)$$

Thus, we can conclude the program step from c^2 to c^3 generates an instruction. With Lemma 4.26, Lemma 4.28 and Lemma 4.30, we have

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 4$$

From the semantics of the page fault step in the abstract machine, we also have

$$c^3.mode_{[i]} \wedge \exists w \in c^3.mmu_{[i]}.tlb.\ fault(pte(c^3.m, w), (I^3.va, I^3.r), w)$$

By the semantics of the abstract machine and the coupling relation, we have

$$\begin{aligned}
I^3.va &= ea(cast(c^2.p_{[i]}), I^2_{isa})[31:2] \\
&= ea(cast(c.p_{[i]}), I^2_{isa})[31:2] \\
&= ea(C.u_i.core, I^2_{isa})[31:2] \\
I^3.r &= store(I^2_{isa}) \vee rmw(I^2_{isa}) \circ 10 \\
c^3.m &= c.m \\
&= C.m \\
c^3.mode_{[i]} &= 1 \\
&= c.mode_{[i]} \\
&= C.u_i.spr(mode)[0] \\
c^3.mmu_{[i]}.tlb &= c.mmu_{[i]}.tlb \\
&= C.u_i.tlb
\end{aligned}$$

Thus, we have:

$$w \in C.u_i.tlb.\ C.u_i.spr(mode)[0] \wedge fault(pte(C.m, w), trqEA, w)$$

Moreover, concludes the lemma. □

Then, we prove the simulation of type 5 block.

Lemma 4.60 (Type 5 Block Simulate by *Cosmos* machine)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{pf}}{\Longrightarrow}_i c^4 \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge$$
$$\neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge c \sim C \rightarrow \exists \alpha.\ C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'$$

Proof Analogous to the proof of Lemma 4.59, we can get

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 4$$

With Lemma 4.35, we can get

$$phase(c^4, i) = 1$$

Thus, the above computation is a type 5 block. By the semantics of the page fault step in the abstract machine, we have

$$\begin{aligned} c^3.mode_{[i]} &= 1 \\ &= c.mode_{[i]} \end{aligned}$$

We let

$$\alpha = (i, (\textbf{core}, w_I, w_R, eev), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

In α, the w_I is chosen in the same manner as in Lemma 4.52. Also, with w_I, the *Cosmos* machine fetches the same instruction as I^2_{isa}. Since according to Lemma 4.52, the w_I can not cause the page fault on fetch. Thus, the flag *pff* is 0. The w_R is chosen according to the Lemma 4.59, which guarantees that the *Cosmos* machine has a page fault on load/store. That is *pfls* = 1.

With Lemma 4.47 and Lemma 4.56, we can get

$$\neg jisr_f(C.u_i, eev, 0) \wedge \neg jisr_x(C.u_i, I_{isa}, 0)$$

As a consequence, we can conclude that in the *Cosmos* machine only the page fault on load/store interrupt happens. With the definition of mca_x, we have

$$mca_x(C.u_i.core, I^2_{isa}, 1) = 0^8 10^{23}$$

From the semantics of the abstract machine, we have

$$\begin{aligned}
c^4.mode_{[i]} &= 0 \\
&= C'.u_i.spr(mode)[0] && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c^4.p_{[i]}.pc &= 0^{32} \\
&= C'.u_i.pc && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c^4.p_{[i]}.gpr &= c.p_{[i]}.gpr \\
&= C.u_i.gpr && \text{(coupling relation)} \\
&= C'.u_i.gpr && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c^4.p_{[i]}.n &= c^2.p_{[i]}.n + 1 \\
&= c^1.p_{[i]}.n + 1 \\
&= c.p_{[i]}.n + 1 \\
&= C.u_i.n + 1 && \text{(coupling relation)} \\
&= C'.u_i.n && \text{(semantics of } \textit{Cosmos} \text{ machine)} \\
c^4.\vartheta_{[i]} &= c^2.\vartheta_{[i]} \\
&= c^1.\vartheta_{[i]}(I_{n^1} \mapsto I^2_{isa}) \\
&= c.\vartheta_{[i]}(I_n \mapsto I^2_{isa}) \\
&= C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa}) && \text{(coupling relation)} \\
&= C'.u_i.\vartheta && \text{(semantics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

For spr_p, according to the semantics, we have

$$c^4.p_{[i]}.spr_r = spr'_p$$

in which we let $I^3 = I^3$ then

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ c^3.mode_{[i]} & x = emode \\ c^3.spr_p(sr) & x = esr \\ 0^8 10^{23} & x = eca \\ c^3.p_{[i]}.ppc & x = epc \\ I^3.va \circ 00 & x = edata \\ c^3.p_{[i]}.spr_p(x) & otherwise \end{cases}$$

From the semantics, we can get

$$\begin{aligned} I^3.va &= ea(cast(c^2.p_{[i]}), I^2_{isa})[31:2] \\ &= ea(cast(c.p_{[i]}), I^2_{isa})[31:2] \\ &= ea(C.u_i.core, I^2_{isa})[31:2] \qquad \text{(coupling relation)} \end{aligned}$$

Since we do not have misalignments on execute, the last 2 bits of effective address is 00.

$$ea(cast(c.p_{[i]}), I^2_{isa}) = ea(C.u_i.core, I^2_{isa})[1:0] = 00$$

Also, from the semantics of the abstract machine, we can conclude:

$$spr'_p(x) = \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ c.mode_{[i]} & x = emode \\ c.spr_p(sr) & x = esr \\ 0^8 10^{23} & x = eca \\ c.p_{[i]}.pc & x = epc \\ ea(cast(c.p_{[i]}), I^2_{isa}) & x = edata \\ c.p_{[i]}.spr_p(x) & otherwise \end{cases}$$

With the coupling relation and the semantics of the *Cosmos* machine, we can get:

$$\begin{aligned} spr'_p(x) &= \begin{cases} 0^{32} & x = sr \\ 0^{31} \circ C.u_i.spr(mode)[0] & x = emode \\ C.u_i.spr(sr) & x = esr \\ 0^8 10^{23} & x = eca \\ C.u_i.pc & x = epc \\ ea(C.u_i.core, I^2_{isa}) & x = edata \\ C.u_i.spr_p(x) & otherwise \end{cases} \\ &= C'.u_i.spr_p(x) \end{aligned}$$

For the TLB, the fault walks are erased.

$$
\begin{aligned}
c^4.mmu_{[i]}.tlb &= \delta_{flush}(c^3.mmu_{[i]}, \{I^3.va\}).tlb && \text{(semantics of abs)}\\
&= c^3.mmu_{[i]}.tlb \setminus \{w \mid I.va.ba = w.va\} && \text{(def. of } \delta_{flush})\\
&= c.mmu_{[i]}.tlb \setminus \{w \mid I.va.ba = w.va\} && \text{(semantics of abs)}\\
&= C.u_i.tlb \setminus \{w \mid ea(C.u_i.core, I^2_{isa})[31:12] = w.va\} && \text{(coupling relation)}\\
&= \delta_{tlb}(C.u_i.tlb, (\textbf{flush}, ea(C.u_i.core, I^2_{isa})[31:12])) && \text{(def. of } \delta_{tlb})\\
&= C'.u_i.tlb && \text{(semantics of cosmos)}
\end{aligned}
$$

The coupling relation for other components is trivially maintained. □

Coupling Maintained by Type 6 Block The thread i of the abstract machine first performs an uninterrupted phase 1 program step and phase 2 memory step as in the previous case, then performs an uninterrupted phase 3 program step and do not generate memory instructions. At last, the machine performs a phase 5 program step.

Lemma 4.61 (Type 6 Block Simulate by *Cosmos* machine)

$$
\begin{aligned}
c \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i c^3 \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow}}_i c^4 \wedge phase(c,i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge\\
\neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge \neg gen\text{-}ins(I^2_{isa}) \wedge c \sim C \rightarrow\\
\exists \alpha.\, C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'
\end{aligned}
$$

Proof With Lemma 4.26, Lemma 4.28, Lemma 4.31 and Lemma 4.36, we have

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 5 \wedge phase(c^4, i) = 1$$

Thus, the above abstract computation is a type 6 interleaving block. We choose a walk w_I in the same manner as we did in the proof of Lemma 4.58. Thus, the *Cosmos* machine can fetch the same instruction as I^2_{isa}. Then, we let

$$\alpha = (i, (\textbf{core}, w_I, \perp, eev), io, ip, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

Note that, since we have $\neg gen\text{-}ins(I^2_{isa})$, according to the definition of *gen-ins* and the semantics of the *Cosmos* machine, the I^2_{isa} does not require memory accesses in the *Cosmos* machine. As a consequence, we give a $\perp$ value to the w_R which is used for the address translation for the load/store.

From the semantics, we have:

$$\forall X \in \{gpr, spr_p, pc\}.\ c^4.p_{[i]}.X = \delta_{instr}(cast(c.p_{[i]}, zxt_{32}(c.mode_{[i]}), c.mmu_{[i]}.pto), I^2_{isa}, 0^{32}).X$$

The execution of I^2_{isa} does not need the read result from memory to update the gpr. Thus, we use a dummy value 0^{32} as the read result, which does not affect the execution of I^2_{isa}. With the coupling relation, we can conclude:

$$cast(c.p_{[i]}, zxt_{32}(c.mode_{[i]}), c.mmu_{[i]}.pto) = C.u_i.core$$

Thus, we have

$$\begin{aligned} c^4.p_{[i]}.X &= \delta_{instr}(C.u_i.core, I^2_{isa}, 0^{32}) \\ &= C'.u_i.X \end{aligned}$$

For the counter and temporary, we have

$$\begin{aligned} c^4.p_{[i]}.n &= c^3.p_{[i]}.n + 1 && \text{(semantics of the abstract machine)} \\ &= c^1.p_{[i]}.n + 1 && \text{(semantics of the abstract machine)} \\ &= c.p_{[i]}.n + 1 && \text{(semantics of the abstract machine)} \\ &= C.u_i.n + 1 && \text{(coupling relation)} \\ &= C'.u_i.n && \text{(semantics of the } \textit{Cosmos} \text{ machine)} \\ c^4.\vartheta_{[i]} &= c^1.\vartheta_{[i]}(I_{n^1} \mapsto I^2_{isa}) && \text{(semantics of the abstract machine)} \\ &= c.\vartheta_{[i]}(I_n \mapsto I^2_{isa}) && \text{(semantics of the abstract machine)} \\ &= C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa}) && \text{(coupling relation)} \\ &= C'.u_i.\vartheta && \text{(semantics of the } \textit{Cosmos} \text{ machine)} \end{aligned}$$

The coupling of other components is trivially maintained. □

Coupling Maintained by Type 7 Block As in the previous case, the machine first performs an uninterrupted phase 1 program step, phase 2 memory step, and uninterrupted phase 3 program step. A memory instruction is generated in the phase 3 program step. In the next step, a phase 4 memory step is performed to execute the instruction. At last, the machine performs a phase 5 program step.

First, we need to prove that the *Cosmos* machine can use the same target physical address as in the abstract machine.

Lemma 4.62 (Physical Memory Address Identical in Translated Mode) In this lemma, we prove that the *Cosmos* machine can have the same address translation as the abstract machine for memory access instruction.

The abstract machine makes an uninterrupted phase 1, phase 2 memory step to fetch. If the fetched instruction is a load, store or rmw instruction, then another memory access is required. In the next step of the abstract machine, it makes an uninterrupted phase 3 program step to generated a memory instruction. Then, the abstract machine makes a phase 4 memory step that executes the memory instruction.

We let

$$\begin{aligned} ea &= ea(C.u_i.core, I^2_{isa}) \\ rights &= store(I^2_{isa}) \vee rmw(I^2_{isa}) \circ 10 \\ trqEA &= (ea[31:2], rights) \end{aligned}$$

then prove that for every possible translated address pa_{ex} in the phase 4 memory step of the abstract machine, we can find a complete walk w_R in the *Cosmos* machine's TLB such that

- There is a hit for the effective address ea and the walk w_R.
- pa_{ex} is the physical address of ea with respect to w_R.

$$
\begin{aligned}
& c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{m}}{\Longrightarrow}_i c^4 \wedge phase(c, i) = 1 \wedge \\
& \neg jisr_f(c.p_{[i]}, eev) \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode, I^2_{isa}) \wedge c \sim C \wedge \\
& c.mode_{[i]} \wedge (load(I^2_{isa}) \vee store(I^2_{isa}) \vee rmw(I^2_{isa})) \wedge \\
& pa_{ex} \in atran(c^3.mmu_{[i]}, I^3.va, c^3.mode_{[i]}, I^3.r) \rightarrow \\
& \quad \exists w_R \in C.u_i.tlb.\ complete(w_R) \wedge hit((ea[31:2], rights), w_R) \wedge \\
& \quad pa_{ex} = w_R.ba \circ ea[11:2]
\end{aligned}
$$

Proof With $load(I^2_{isa}) \vee store(I^2_{isa}) \vee rmw(I^2_{isa})$, we can conclude:

$$gen\text{-}ins(I^2_{isa})$$

With Lemma 4.26, Lemma 4.28, Lemma 4.30, we have

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 4$$

From the semantics of the abstract machine, we have:

$$
\begin{aligned}
I^3.r &= rights \\
I^3.va &= ea(cast(c^2.p_{[i]}), I^2_{isa})[31:2] \\
&= ea(cast(c.p_{[i]}), I^2_{isa})[31:2] \\
&= ea[31:2] \\
c.mode_{[i]} &= c^3.mode_{[i]} \\
atran(c^3.mmu_{[i]}, I^3.va, c^3.mode_{[i]}, I^3.r) &= atran(c.mmu_{[i]}, I^3.va, c.mode_{[i]}, I^3.r)
\end{aligned}
$$

With the definition of *atran*, we have:

$$
\begin{aligned}
& pa_{ex} \in atran(c.mmu_{[i]}, I^3.va, c.mode_{[i]}, I^3.r) \\
& \Rightarrow pa_{ex} \in \{w.ba \circ I^3.va[9:0] \mid w \in c.mmu_{[i]}.tlb \wedge complete(w) \wedge hit((I^3.va, I^3.r), w)\} \\
& \Rightarrow pa_{ex} \in \{w.ba \circ I^3.va[9:0] \mid w \in C.u_i.tlb \wedge complete(w) \wedge hit((I^3.va, I^3.r), w)\} \\
& \Rightarrow pa_{ex} \in \{w.ba \circ ea[11:2] \mid w \in C.u_i.tlb \wedge complete(w) \wedge hit((ea[31:2], rights), w)\}
\end{aligned}
$$

The lemma is concluded. □

Lemma 4.63 (Physical Memory Address Identical in Untranslated Mode) We prove an analogous lemma for the untranslated case. In this lemma, we reuse all the shorthands in the last

lemma.

$$
\begin{aligned}
& c \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c^1 \overset{\text{m}}{\Longrightarrow_i} c^2 \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c^3 \overset{\text{m}}{\Longrightarrow_i} c^4 \wedge phase(c, i) = 1 \wedge \\
& \neg jisr_f(c.p_{[i]}, eev) \wedge \neg jisr_x(c^2.p_{[i]}, c^2.mode, I^2_{isa}) \wedge c \sim C \wedge \\
& \neg c.mode_{[i]} \wedge (load(I^2_{isa}) \vee store(I^2_{isa}) \vee rmw(I^2_{isa})) \wedge \\
& pa_{ex} \in atran(c^3.mmu_{[i]}, I^3.va, c^3.mode_{[i]}, I^3.r) \rightarrow pa_{ex} = ea[31:2]
\end{aligned}
$$

Proof This lemma can be proved by analogous steps as Lemma 4.62. The only difference is the definition of *atran*. With the definition of *atran* in the untranslated case, we have:

$$
\begin{aligned}
& pa_{ex} \in atran(c.mmu_{[i]}, I^3.va, c.mode_{[i]}, I^3.r) \\
& \Rightarrow pa_{ex} \in \{I^3.va\} \\
& \Rightarrow pa_{ex} \in \{ea[31:2]\}
\end{aligned}
$$

The lemma is concluded. □

Then, we prove that the type 7 interleaving block can be simulated by one *Cosmos* machine step.

Lemma 4.64 (Type 7 Block Simulate by *Cosmos* machine)

$$
\begin{aligned}
& c \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c^1 \overset{\text{m}}{\Longrightarrow_i} c^2 \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c^3 \overset{\text{m}}{\Longrightarrow_i} c^4 \overset{\text{p}}{\underset{\text{eev}}{\Longrightarrow_i}} c^5 \wedge phase(c, i) = 1 \wedge \neg jisr_f(c.p_{[i]}, eev) \wedge \\
& \quad \neg jisr_x(c^2.p_{[i]}, c^2.mode_{[i]}, I^2_{isa}) \wedge gen\text{-}ins(I^2_{isa}) \wedge c \sim C \wedge safety(C, P_{og^{MIPS}_{cos}}) \rightarrow \\
& \quad \exists \alpha.\, C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'
\end{aligned}
$$

Proof With Lemma 4.26, Lemma 4.28, Lemma 4.30 and Lemma 4.33, we have

$$phase(c^1, i) = 2 \wedge phase(c^2, i) = 3 \wedge phase(c^3, i) = 4 \wedge phase(c^4, i) = 5 \wedge phase(c^5, i) = 1$$

Thus, the above abstract machine computation is a type 7 block. We let

$$\alpha = (i, (\textbf{core}, w_I, w_R, eev), io, ip, annot^{MIPS}_{cos})$$

in which

- w_I. If $c.mode_{[i]}$ we apply Lemma 4.52 and choose the walk w_I to perform address translation for fetching. Otherwise, we apply Lemma 4.53 and $w_I = \perp$. The Lemma 4.52 and Lemma 4.53 also guarantee that the *Cosmos* machine fetches identical instruction as I^2_{isa}.
- w_R. If $c.mode_{[i]} \wedge (load(I_{isa}) \vee store(I_{isa}) \vee rmw(I_{isa}))$ we apply Lemma 4.62 and choose the walk w_R such that the *Cosmos* machine accesses the same physical memory address as the abstract machine in the phase 4 memory step. If $\neg c.mode_{[i]} \wedge (load(I_{isa}) \vee store(I_{isa}) \vee rmw(I_{isa}))$ we apply Lemma 4.63 and let $w_R = \perp$. In this case, the *Cosmos* machine also accesses the same physical memory address as the abstract machine in the phase 4 memory step. Otherwise, no memory accesses are required by I^2_{isa}. Thus, we let $w_R = \perp$.

- $annot^{MIPS}_{cos}$. The ownership annotations are defined as:

$$annot^{MIPS}_{cos} = \begin{cases} og^{MIPS}_{cos}(C.u_i.core, \vartheta'_{cos}) & \alpha.io \\ (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) & otherwise \end{cases}$$

  where:

$$\vartheta'_{cos} = S^n_{\text{MIPS-86}}.\delta(C.u_i, C.m, \alpha.in).u_i.\vartheta$$

From the semantics of the abstract machine we know I^3 is the memory instruction generated according to I^2_{isa}. In the following proof, we make a case split on I^2_{isa}.

- $\neg(load(I^2_{isa}) \vee store(I^2_{isa}) \vee rmw(I^2_{isa}))$. With $gen\text{-}ins(I^2_{isa})$, we can conclude

$$mfence(I^2_{isa}) \vee switch(I^2_{isa}) \vee eret(I^2_{isa}) \vee wpto(I^2_{isa}) \vee invlpg(I^2_{isa}) \vee flush(I^2_{isa})$$

  According to the semantics of the abstract machine, I^3 clears the dirty bit. Thus, we have:

$$\begin{aligned} c^5.\mathcal{D}_{[i]} &= c^4.\mathcal{D}_{[i]} \\ &= 0 \\ &= C'.u_i.\mathcal{D} && \text{(semantics of } \textit{Cosmos} \text{ machine)} \end{aligned}$$

  For the temporary, from the semantics and the coupling relation, we have:

$$\begin{aligned} c^5.\vartheta_{[i]} &= c^4.\vartheta_{[i]} \\ &= c^3.\vartheta_{[i]} \\ &= c^2.\vartheta_{[i]} \\ &= c^1.\vartheta_{[i]}(I_{n^1} \mapsto I^2_{isa}) \\ &= c.\vartheta_{[i]}(I_n \mapsto I^2_{isa}) \\ &= C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa}) \\ &= C'.u_i.\vartheta \end{aligned}$$

  Thus, if $mfence(I^2_{isa})$ then the coupling is maintained. We do a further case split on I^2_{isa}.

  - $switch(I^2_{isa})$. From the semantics of the abstract machine, we have:

$$I^3 = \textbf{SWITCH}\ c^2.p_{[i]}.gpr(rt(I^2_{isa}))[0]$$

    With the semantics of the abstract machine, we have:

$$\begin{aligned} c^5.mode_{[i]} &= c^4.mode_{[i]} \\ &= c^2.p_{[i]}.gpr(rt(I^2_{isa}))[0] \\ &= c.p_{[i]}.gpr(rt(I^2_{isa}))[0] \\ &= C.u_i.gpr(rt(I^2_{isa}))[0] && \text{(coupling relation)} \\ &= C'.u_i.spr(mode)[0] && \text{(sematics of } \textit{Cosmos} \text{ machine)} \end{aligned}$$

    The coupling is maintained in this case.

– $eret(I^2_{isa})$. From the semantics of the abstract machine, we have:

$$I^3 = \textbf{SWITCH}\ c^2.p_{[i]}.spr_p(emode)[0]$$

Thus, according to the semantics of the abstract machine, we have:

$$\begin{aligned}
c^5.mode_{[i]} &= c^4.mode_{[i]} \\
&= c^2.p_{[i]}.spr_p(emode)[0] \\
&= c.p_{[i]}.spr_p(emode)[0] \\
&= C.u_i.spr_p(emode)[0] && \text{(coupling relation)} \\
&= C'.u_i.spr(mode)[0] && \text{(sematics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

The coupling is maintained in this case.

– $wpto(I^2_{isa})$. From the semantics of the abstract machine, we have:

$$I^3 = \textbf{WPTO}\ c^2.p_{[i]}.gpr(rt(I^2_{isa}))$$

With the semantics of the abstract machine, we have:

$$\begin{aligned}
c^5.mmu_{[i]}.pto &= c^4.mmu_{[i]}.pto \\
&= c^2.p_{[i]}.gpr(rt(I^2_{isa})) \\
&= c.p_{[i]}.gpr(rt(I^2_{isa})) \\
&= C.u_i.gpr(rt(I^2_{isa})) && \text{(coupling relation)} \\
&= C'.u_i.spr(pto) && \text{(sematics of } \textit{Cosmos} \text{ machine)}
\end{aligned}$$

The coupling is maintained in this case.

– $invlpg(I^2_{isa}) \vee flush(I^2_{isa})$. From the semantics of the abstract machine, we have:

$$I^3 = \textbf{INVLPG}\ F$$

in which

$$F = \begin{cases} c^2.p_{[i]}.gpr(rd(I^2_{isa}))[31:2] & invlpg(I^2_{isa}) \\ \mathbb{B}^3 0 & flush(I^2_{isa}) \end{cases}$$

With the semantics of the abstract machine, the semantics of the *Cosmos* machine

and the coupling relation, we have:

$$
\begin{aligned}
& c^5.mmu_{[i]}.tlb \\
&= c^4.mmu_{[i]}.tlb \\
&= \delta_{flush}(c^3.mmu_{[i]}, F) \\
&= c^2.mmu_{[i]}.tlb \setminus \{w \mid \exists a \in F.\ a[29:10] = w.va\} \\
&= \begin{cases} c^2.mmu_{[i]}.tlb \setminus \{w \mid c^2.p_{[i]}.gpr(rd(I^2_{isa}))[31:12] = w.va\} & invlpg(I^2_{isa}) \\ \emptyset & flush(I^2_{isa}) \end{cases} \\
&= \begin{cases} c.mmu_{[i]}.tlb \setminus \{w \mid c.p_{[i]}.gpr(rd(I^2_{isa}))[31:12] = w.va\} & invlpg(I^2_{isa}) \\ \emptyset & flush(I^2_{isa}) \end{cases} \\
&= \begin{cases} C.u_i.tlb \setminus \{w \mid C.u_i.gpr(rd(I^2_{isa}))[31:12] = w.va\} & invlpg(I^2_{isa}) \\ \emptyset & flush(I^2_{isa}) \end{cases} \\
&= \begin{cases} \delta_{tlb}(C.u_i.tlb, (\textbf{flush}, C.u_i.gpr(rd(I^2_{isa}))[31:12])) & invlpg(I^2_{isa}) \\ \emptyset & flush(I^2_{isa}) \end{cases} \\
&= C'.u_i.tlb
\end{aligned}
$$

The coupling is maintained in this case.

- $(load(I^2_{isa}) \vee store(I^2_{isa}) \vee rmw(I^2_{isa}))$. With the semantics of the abstract machine, we have

$$R(I^3) \vee W(I^3) \vee RMW(I^3)$$

and

$$
\begin{aligned}
I^3.va &= ea(cast(c^2.p_{[i]}), I^2_{isa}) \\
&= ea(cast(c.p_{[i]}), I^2_{isa}) && \text{(semantics of abstract machine)} \\
&= ea(C.u_i.core, I^2_{isa}) && \text{(coupling relation)} \\
I^3.bw &= bw(cast(c^2.p_{[i]}), I^2_{isa}) \\
&= bw(cast(c.p_{[i]}), I^2_{isa}) && \text{(semantics of abstract machine)} \\
&= bw(C.u_i.core, I^2_{isa}) && \text{(coupling relation)}
\end{aligned}
$$

If $c.mode_{[i]}$ we apply Lemma 4.62, otherwise we apply Lemma 4.63. After that, we can get that the abstract machine and the *Cosmos* machine access the same physical address in the memory with the same byte write signals. We let the physical address be pa. Then, for the load and rmw instructions, we have to prove the read value is identical. We let the read value be v.

$$
\begin{aligned}
v &= lv(c^3.m(pa), I^2_{isa}) \\
&= lv(c.m(pa), I^2_{isa}) \\
&= lv(C.m(pa), I^2_{isa})
\end{aligned}
$$

From the semantics of the abstract machine and *Cosmos* machine, and the coupling relation, we also have:

$$
\begin{aligned}
c^5.p_{[i]}.gpr(x) &= \begin{cases} v & updategpr(I^2_{isa}, x) \\ c^4.p_{[i]}.gpr(x) & otherwise \end{cases} \\
&= \begin{cases} v & updategpr(I^2_{isa}, x) \\ c.p_{[i]}.gpr(x) & otherwise \end{cases} \\
&= \begin{cases} v & updategpr(I^2_{isa}, x) \\ C.u_i.gpr(x) & otherwise \end{cases} \\
&= C'.u_i.gpr(x)
\end{aligned}
$$

For the temporary, from the semantics and the coupling relation, we have:

$$
\begin{aligned}
c^5.\vartheta_{[i]} &= c^4.\vartheta_{[i]} \\
&= \begin{cases} c^3.\vartheta_{[i]}(R_{c^3.p_{[i]}.n} \mapsto v) & load(I^2_{isa}) \vee rmw(I^2_{isa}) \\ c^3.\vartheta_{[i]} & otherwise \end{cases} \\
&= \begin{cases} c^2.\vartheta_{[i]}(R_{c^2.p_{[i]}.n} \mapsto v) & load(I^2_{isa}) \vee rmw(I^2_{isa}) \\ c^2.\vartheta_{[i]} & otherwise \end{cases} \\
&= \begin{cases} c^1.\vartheta_{[i]}(I_{c^1.p_{[i]}.n} \mapsto I^2_{isa})(R_{c^1.p_{[i]}.n} \mapsto v) & load(I^2_{isa}) \vee rmw(I^2_{isa}) \\ c^1.\vartheta_{[i]}(I_{c^1.p_{[i]}.n} \mapsto I^2_{isa}) & otherwise \end{cases} \\
&= \begin{cases} c.\vartheta_{[i]}(I_{c.p_{[i]}.n} \mapsto I^2_{isa})(R_{c.p_{[i]}.n} \mapsto v) & load(I^2_{isa}) \vee rmw(I^2_{isa}) \\ c.\vartheta_{[i]}(I_{c.p_{[i]}.n} \mapsto I^2_{isa}) & otherwise \end{cases} \\
&= \begin{cases} C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa})(R_{C.u_i.n} \mapsto v) & load(I^2_{isa}) \vee rmw(I^2_{isa}) \\ C.u_i.\vartheta(I_{C.u_i.n} \mapsto I^2_{isa}) & otherwise \end{cases} \\
&= C'.u_i.\vartheta
\end{aligned}
$$

For the rmw instruction, we have:

$$
\begin{aligned}
&I^3.cond(c^4.\vartheta_{[i]}) \equiv c^3.p_{[i]}.gpr(rd(I^2_{isa})) = c^3.m(pa) \\
&\leftrightarrow c.p_{[i]}.gpr(rd(I^2_{isa})) = c.m(pa) && \text{(semantics of abstract machine)} \\
&\leftrightarrow C.u_i.gpr(rd(I^2_{isa})) = C.m(pa) && \text{(coupling relation)}
\end{aligned}
$$

Thus, the condition value for rmw is also consistent in both machines. For store and rmw instructions, we also have to prove the store value is equal in both machines. In the

abstract machine, we store value is defined as:

$$
\begin{aligned}
I^3.f(c^3.\vartheta_{[i]}) &= \begin{cases} s4s(c^3.p_{[i]}.gpr(rt(I^2_{isa})), I^2_{isa}) & store(I^2_{isa}) \\ c^3.p_{[i]}.gpr(rt(I^2_{isa}), I^2_{isa}) & rmw(I^2_{isa}) \end{cases} \\
&= \begin{cases} s4s(c.p_{[i]}.gpr(rt(I^2_{isa})), I^2_{isa}) & store(I^2_{isa}) \\ c.p_{[i]}.gpr(rt(I^2_{isa}), I^2_{isa}) & rmw(I^2_{isa}) \end{cases} \\
&= \begin{cases} s4s(c.p_{[i]}.gpr(rt(I^2_{isa})), I^2_{isa}) & store(I^2_{isa}) \\ c.p_{[i]}.gpr(rt(I^2_{isa}), I^2_{isa}) & rmw(I^2_{isa}) \end{cases} \\
&= \begin{cases} s4s(C.u_i.gpr(rt(I^2_{isa})), I^2_{isa}) & store(I^2_{isa}) \\ C.u_i.gpr(rt(I^2_{isa}), I^2_{isa}) & rmw(I^2_{isa}) \end{cases}
\end{aligned}
$$

According to the semantics, the store value of the abstract machine is equal to the store value of the *Cosmos* machine. Thus, for the store and rmw instructions, we apply the same update to the memory of the abstract machine and *Cosmos* machine. We can get

$$
\begin{aligned}
C'.m &= c^4.m \\
&= c^5.m \qquad\qquad \text{(semantics of abstract machine)}
\end{aligned}
$$

If $\neg I^3.vol$ then the coupling is maintained. Otherwise, from the semantics we have:

$$
\begin{aligned}
& c^2.p_{[i]}.pc[31:2] \in A_{io} \\
& \Rightarrow c.p_{[i]}.pc[31:2] \in A_{io} \\
& \Rightarrow C.u_i.pc[31:2] \in A_{io}
\end{aligned}
$$

Thus, we can conclude that the *io* flag in step information α is set in this case. For the dirty bit, we have

$$
\begin{aligned}
c^5.\mathcal{D}_{[i]} &= c^4.\mathcal{D}_{[i]} \qquad\qquad \text{(semantics of abstract machine)} \\
&= \begin{cases} 1 & vW(I^3) \\ 0 & RMW(I^3) \\ c.\mathcal{D}_{[i]} & otherwise \end{cases}
\end{aligned}
$$

From the semantics of the abstract machine, we have:

$$
\begin{aligned}
vW(I^3) &\leftrightarrow store(I^2_{isa}) \wedge c^2.p_{[i]}.pc[31:2] \in A_{io} \\
RMW(I^3) &\leftrightarrow rmw(I^2_{isa})
\end{aligned}
$$

Thus, with the coupling relation and the semantics of the *Cosmos* machine, we can con-

clude:

$$c^5.\mathcal{D}_{[i]} = \begin{cases} 1 & store(I^2_{isa}) \wedge c.p_{[i]}.pc[31:2] \in A_{io} \\ 0 & rmw(I^2_{isa}) \\ c.\mathcal{D}_{[i]} & otherwise \end{cases}$$
$$= \begin{cases} 1 & store(I^2_{isa}) \wedge \alpha.io \\ 0 & rmw(I^2_{isa}) \\ C.u_i.\mathcal{D} & otherwise \end{cases}$$
$$= C'.u_i.\mathcal{D}$$

From the definition of $safety(C, P_{og^{MIPS}_{cos}})$, we have

$$(A, L, R, A_{pt}, R_{pt}) = og^{MIPS}_{cos}(C.u_i.core, C'.u_i.\vartheta)$$

We define the value of $og(I^3.p, c^4.\vartheta_{[i]})$ with

$$(A, L, R, R, A_{pt}, R_{pt})$$

To maintain the coupling relation, we do not change the read-only set *ro* in the abstract machine. Thus, all the addresses belong to the released set are writable. Note that the pair $(I^3.p, c^4.\vartheta_{[i]})$ can be computed from the abstract machine configuration c and the pair $(C.u_i.core, C'.u_i.\vartheta)$ also can be computed from the *Cosmos* machine configuration C and α. From the coupling relation, we know that for all c there exists a unique C such that $c \sim C$. Thus, the function *og* in the abstract machine is well defined. From the semantics of both machines and the coupling relation, the abstract machine performs identical ownership transfer on identical ghost information as the *Cosmos* machine, coupling relation for the ghost information is maintained.

□

Theorem 4.65 (Simulation Theorem Between Abstract Machine and *Cosmos* machine) *With the above one block simulation, we can inductively prove the simulation theorem between an interleaving-reduced abstract machine computation, which consists only complete interleaving blocks, and an* Cosmos *machine computation.*

$$c \underset{eev}{\Rightarrow^*} c' \wedge c \sim C \wedge safety(C, P_{og^{MIPS}_{cos}}) \rightarrow \exists \tau.\ C \xrightarrow{\tau} C' \wedge c' \sim C'$$

Incomplete Block Simulation Note that, since the execution of the abstract machine is serialized within each block by the semantics, every incomplete block can be completed. Each step within a block is deterministically executed[1] except the page fault step. The reason is that if there exists a faulty walk in the TLB, the machine can use the walk to signal a page fault or

[1]Here for deterministically we mean that the type of next steps is fixed by the current configuration and thread id. Since for the address translation in memory steps, the non-determinism still exists.

continue execution by choosing another non-faulty walk if possible. We proved the simulation of both cases.

Another interesting case is when we complete a block with a phase 4 memory step, in which the ownership transfer is required. In this case, we need to choose the proper ownership annotations. Since the incomplete block can only be the last block of each thread, without loss of generality, we assume in the computation $c \underset{\text{eev}}{\Rightarrow}^* c'$, only the last interleaving block is incomplete.

$$c \underset{\text{eev}}{\Rightarrow}^* c' \wedge \forall j \neq i.\ phase(c', j) = 1 \wedge phase(c', i) = 4 \wedge (vR(I') \vee vW(I') \vee RMW(I'))$$

We assume that c^1 is the start machine configuration of the last interleaving block. Then c^1 is also the end configuration of the previous interleaving block. Thus, we have:

$$\forall j.\ phase(c^1, j) = 1$$

Applying Theorem 4.65, we can get

$$\exists \tau.\ C \xrightarrow{\tau} C^1 \wedge c^1 \sim C^1$$

We can define the step information α^1 analogously as in Lemma 4.64. Thus, we can also choose the ownership annotations as we did in the proof of Lemma 4.64.

Safety Transfer

At last we need to prove the safety property is correctly transferred.

Theorem 4.66 (Safety Maintenance)

$$c \sim C \wedge safety(C, P_{og^{\text{MIPS}}_{cos}}) \rightarrow safe\text{-}reach(c, og)$$

Proof We prove this theorem by contradiction. We assume

$$\exists c'.\ c \underset{\text{eev}}{\Rightarrow}^* c' \wedge \neg safe\text{-}state(c, og)$$

With the definition of $safe\text{-}state(c, og)$, let

$$\vartheta' = \begin{cases} c'.\vartheta_{[i]}(I'.t \mapsto c'.m(pa)) & RMW(I') \\ c'.\vartheta_{[i]}(I'.t \mapsto I'.ext(c'.m(pa), I'.bw)) & R(I') \\ c'.\vartheta_{[i]} & otherwise \end{cases}$$

$$annot = og(I'.p, \vartheta')$$

then we have

$$\exists i.\ \neg safe\text{-}instr(c', i, I', annot)$$

or

$$\exists i, pa.\ can\text{-}access(c'.mmu_{[i]}, pa) \wedge \neg safe\text{-}mmu\text{-}access(c', pa, i)$$

Thus, we make a case split.

- $\exists i.\ \neg safe\text{-}instr(c', i, I', annot)$. With the definition of $safe\text{-}instr$, we know that only the memory steps can violate the safety condition in this case. Thus, we have

$$phase(c', i) \in \{2, 4\}$$

First, we perform interleaving order reduction in the computation $c \underset{\text{eev}}{\Longrightarrow}^{*} c'$. Since each incomplete blocks can be completed, without loss of generality, we assume the last interleaving block is the only incomplete block in the interleaving-reduced computation from c to c'. We also assume that c^1 is the start machine configuration of the last interleaving block. Then c^1 is also the end configuration of the previous interleaving block. Thus, we have:

$$\forall j.\ phase(c^1, j) = 1$$

Applying Theorem 4.65, we can get

$$\exists \tau.\ C \xrightarrow{\tau} C^1 \wedge c^1 \sim C^1$$

With the definition of $safety(C, P_{og^{\text{MIPS}}_{cos}})$, we can also have:

$$safety(C^1, P_{og^{\text{MIPS}}_{cos}})$$

We let the physical address of memory access be pa then make a further case split here:

  - $phase(c', i) = 2$. Then, from the semantics of the abstract machine, we can have:

$$c^1 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}_i} c'$$

    Also from the semantics, we have

$$pa \in atran(c'.mmu_{[i]}, I.va, c'.mode_{[i]}, I.r)$$

    The code region invariant (Definition 4.23) guarantees that

$$pa \in c'.ro$$

    which can not violate the safety condition.

  - $phase(c', i) = 4$. Then, from the semantics of the abstract machine, we can have:

$$\exists c^2, c^3.\ c^1 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}_i} c^2 \overset{\text{m}}{\Longrightarrow}_i c^3 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}_i} c'$$

    We let the step information for next *Cosmos* machine step from C^1 be α^1

$$\alpha^1 = (i, (\textbf{core}, w_I, w_R, eev), io, ip, annot_{cos})$$

    in which the w_I and w_R are chosen analogously as in Lemma 4.64. The manner of choosing w_I and w_R guarantee that the *Cosmos* machine fetches the same instruction

I'_{isa} and accesses the same memory address as the abstract machine. We can also conclude

$$vR(I') \vee vW(I') \vee RMW(I') \rightarrow \mathcal{IO}_i(C^1, \alpha^1.in)$$

With the definition of *og* in Lemma 4.64, we also know that the same ownership transfer is performed in the *Cosmos* machine as in the abstract machine. With the coupling relation we can conclude:

$$\neg safe\text{-}instr(c', i, I, annot) \rightarrow \neg safety(C^1, P_{og^{\text{MIPS}}_{cos}})$$

which gives us a contradiction.

- $\exists i, pa.\ can\text{-}access(c'.mmu_{[i]}, pa) \wedge \neg safe\text{-}mmu\text{-}access(c', pa, i)$. Since each incomplete interleaving block can be completed, we assume the computation $c \overset{*}{\underset{\text{eev}}{\Longrightarrow}} c'$ can be reordered into a computation consists of complete interleaving blocks. With Theorem 4.65, we can have

$$\exists \tau.\ C \xrightarrow{\tau} C' \wedge c' \sim C'$$

From the definition of *can-access* we have

$$can\text{-}access(c'.mmu_{[i]}, pa) \equiv \exists w \in c'.mmu_{[i]}.tlb.\ ptea(w) = pa \wedge \neg complete(w)$$

From the coupling relation, we can get

$$w \in C'.u_i.p.tlb.\ \neg complete(w)$$

With $P_{og^{MIPS}_{cos}}(C')$ we have

$$ptea(w) \in C'.\mathcal{P}t_i \cup C'.\mathcal{G}.\mathcal{S} \wedge \forall j.\ ptea(w) \notin C'.\mathcal{O}_j$$

which implies

$$safe\text{-}mmu\text{-}access(c', pa, i)$$

Moreover, contradicts the assumption.

□

5 Applying Store Buffer Reduction to C-IL

One goal of this thesis is to provide a programming discipline such that a verifier for sequentially consistent C can verify the user program that is written in C code and running on a TSO machine. Since the MMU is invisible for a user program, in this chapter we only need an SB reduction theorem without MMU. Note that, in the remaining part of this chapter, we call the simplified SB machine and simplified abstract machine the SB machine and the abstract machine respectively. For the simplified SB reduction theorem, we also call it the SB reduction theorem.

In Figure 5.1, we present the model stack. At the bottom layer of the stack is the MIPS machine, which is a simplified MIPS-86 machine without the TLB and MMU. It can be trivially simulated by the SB machine, which resides at the second layer. With the SB reduction theorem, we can use an abstract machine, which is the third layer of model stack, to simulate the SB machine. After that, for every MIPS machine computation, we have a corresponding arbitrary-interleaved (i.e. processors take steps one after another in an arbitrary order) sequentially consistent computation, which obeys our programming discipline. However, our goal is mapping the programming discipline to the C level by applying the multicore compiler correctness theorem, which consists of two theorems: (i) an order reduction theorem to reorder the arbitrarily-interleaved ISA computation into a block-scheduling computation (i.e. processors execute blocks of steps); (ii) a sequential compiler correctness theorem to simulate each ISA block with a C block. As a consequence, we introduce the order reduction theorem in [Bau14]. After the reordering, we can apply the sequential simulation on each execution block. The MIPS *Cosmos* machine is regarded as the fourth layer of the model stack. At last, we apply the multicore compiler correctness theorem from [Bau14] to simulate a MIPS *Cosmos* machine computation with a C Intermediate Language [1] *Cosmos* machine computation. The C-IL *Cosmos* machine is the top layer of our model stack.

In the first section of this chapter, we will introduce the Simplified MIPS ISA, which is a subset of MIPS-86 without the TLB and MMU. Then, we will present the SB reduction theorem without MMU and the instantiation. The first four sections are simplifications of the previous portion of this thesis. Moreover, we will present the order reduction theorem from [Bau14] and the *Cosmos* machine with MIPS instantiation and C-IL instantiation. In the last section, we will present the simulation theorem from [Bau14]. In the last three sections, we make the following technical contributions:

- introducing the dirty bit and temporaries in the MIPS *Cosmos* machine and C-IL *Cosmos*

[1] C-IL, which was developed by S. Schmaltz [SS12] to justify the C verification approach of the verifier VCC [Mic]

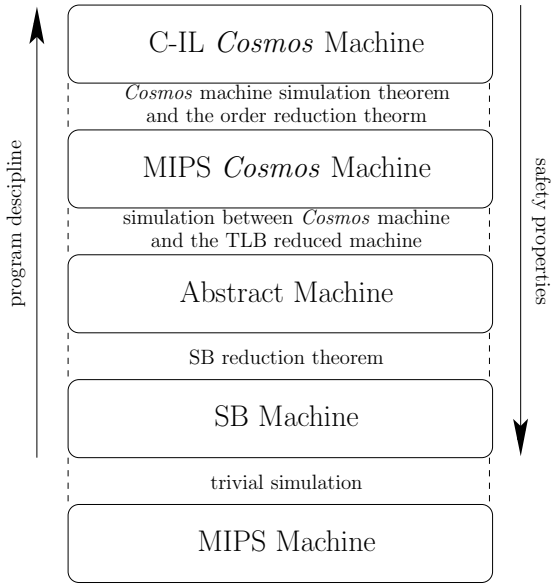


Figure 5.1: Pervasive concurrent model stack

machine.

- extending the simulation relation by adding the simulation relation for the dirty bit.

- overloading the ownership annotation generation function og_{cos}^{MIPS} for MIPS *Cosmos* machine and introducing og_{cos}^{C-IL} C-IL *Cosmos* machine.

- defining og_{cos}^{MIPS} with og_{cos}^{C-IL}.

In this chapter, we assume the user programs run in virtual memory. Further more, we restrict that the page tables are properly set up such that the address translation is always a bijective mapping.

5.1 Simplified ISA

To verify the user program, in which the address translation is invisible, we introduce a simplified MIPS-86 ISA called SB MIPS. The configuration of an SB MIPS processor consists a $core \in K_{core}$ and an $sb \in K_{sb}$. In the semantics, there are three main differences in the transition function of processor core: (i) The $sysc, invlpg$ and $flush$ instructions only update the pc in the core configuration. (ii) Since the SB MIPS is an ISA for the user program, no interrupt occurs in the SB MIPS machine. (iii) The semantics of $movg2s$, $movs2g$ and $eret$ are undefined. We redefine the transition function of processor core δ_{core} defined in Definition 3.6 as a partial

function:

$$\delta_{core}(c, I, R) = \begin{cases} \bot & movg2s(I) \vee movs2g(I) \vee eret(I) \\ \delta_{instr}(c, I, R) & otherwise \end{cases}$$

The rest of the semantics are completely analogous to the semantics of MIPS-86 when $mode = 0$. Note that since the address translation is a bijective mapping, we can use the δ_m, which updates at most one memory cell at each step, as the transition function for memory.

As an ISA for the user program after applying the SB reduction, we obtain an ISA named MIPS, which does not have the SB and TLB. The MIPS processor configuration $p_{mips\text{-}pro} \in K_{mips\text{-}pro}$ only consists of a $core \in K_{core}$. In the semantics, the sequential transition function $\delta_{mips\text{-}seq}$ is defined analogously to $\delta_{sbr\text{-}seq}$ in Definition 3.30 using the overloaded δ_{core} and the semantics when $mode = 0$. The rest of the semantics are completely analogous to the semantics of SB reduced MIPS-86 when $mode = 0$.

5.2 SB Reduction Theorem

In this section, we introduce a simplified version of the SB reduction theorem, which do not consider the MMU and is very similar to the Cohen-Schirmer theorem in [CS10b]. We make an assumption that the user program runs in virtual memory and can not change the special purpose register *mode* and *pto*, and the TLB is invisible, compare with the theorem in Chapter 2, there are the following main differences: In the simplified SB reduction theorem, we do not have

- MMU steps and page fault steps,
- address translation, and
- **INVLPG**, **SWITCH**, **WPTO** instructions.

The machine configurations, safety condition and the invariants are also changed to adapt to the simplification. In the remainder of this chapter, by SB reduction theorem we mean the simplified version.

5.2.1 Instructions, Machine Configurations and Semantics

We define the set of instructions as a subset of Definition 2.2.

Definition 5.1 (User Memory Instruction) The set of user memory instructions $\mathbb{I}_{usr}$ is defined with the following constructors:

$$\mathbb{I}_{usr} = \mathbb{I} \setminus A$$

in which

$$\begin{aligned} A = &\{\textbf{INVLPG } F \mid F \in 2^{\mathbb{A}}\} \\ &\cup \{\textbf{SWITCH } mode \mid mode \in \mathbb{B}\} \\ &\cup \{\textbf{WPTO } v \mid v \in \mathbb{V}\} \end{aligned}$$

The constructors have the same meaning as in Definition 2.2.

Since the page tables are invisible to user programs, in each thread-local abstract machine configuration (Definition 2.3) and thread-local SB machine configuration (Definition 2.19), the local page table set *pt* is set to $\emptyset$.

In the semantics, we define the address translation function *atran* as:

$$atran(mmu, va, mode, r) = \{va\}$$

because the address translation is transparent to the user, the address translation function returns the original address *va*. To make the local page table sets invisible, we also impose the following constraint to the ownership annotation generation function *og* such that the acquired page table addresses A_{pt} and released page table addresses R_{pt} are empty sets:

$$og(p, \vartheta) = (A, L, R, W, \emptyset, \emptyset) = (A, L, R, W)$$

In the semantics, we also get rid of the MMU steps and the page fault steps. We also restrict that in each program step of both the SB machine and the abstract machine, the newly generated instruction sequence $is' \in \mathbb{I}_{usr}$. As a consequence, in each configuration of the abstract machine and the SB machine, the *mode* and the MMU state *mmu* can be ignored. The rest of semantics of the abstract machine and the semantics of the SB machine are analogous to what we defined in Chapter 2.

In the coupling relation, we also ignore the coupling of *pt*, *mode* and *mmu*.

5.2.2 Safety Conditions

In the simplified model, we do not have MMU steps, we simplify the safety condition for machine state *c* as following:

$$safe\text{-}state(c, og) \equiv \forall i.\ safe\text{-}instr(c, i, I, annot)$$

in which

$$\begin{aligned} I &= hd(c.is_{[i]}) \\ v &= I.ext(c.m(I.va), I.bw) \\ \vartheta' &= \begin{cases} c.\vartheta_{[i]}(I.t \mapsto c.m(va)) & RMW(I) \\ c.\vartheta_{[i]}(I.t \mapsto v) & vR(I) \\ c.\vartheta_{[i]} & otherwise \end{cases} \\ annot &= og(I.p, \vartheta') \end{aligned}$$

The remaining safety conditions are analogous to the corresponding definitions in Section 2.2.3. The rest of the SB reduction theorem and the proof are analogous to what we had in Chapter 2.

5.3 Instantiation

In this section, we will instantiate the model in Section 5.2. This section is a simplified version of Chapter 3.

Because we only consider the user program which do not have interrupts, there are only 2 kinds of execution rounds in the instantiated machine. For the execution of each instruction, the machine first performs an uninterrupted phase 1 program step, then a phase 2 memory step for fetch. After that, the machine performs an uninterrupted phase 3 program step. Depending on whether an instruction is generated, the machine either performs a phase 5 program step when no instruction is generated or performs a phase 4 memory step and a phase 5 program step. In this section, we reuse most of the definitions in Chapter 3 to define the new δ_p.

Note that, since interrupts are invisible to the user program, the *eev* is always instantiated to 0^{256}.

5.3.1 Transition Function δ_p in Program Step

We overload the instruction generation function as follows:

Definition 5.2 (Instruction Generation Function) As in Chapter 3, we define $I = \vartheta(I, p.n)$ as the value of temporary I with respect to the counter $p.n$. $switch(I)$ is true if I is a $movg2s$ instruction and updates the SPR *mode*. $wpto(I)$ is satisfied if I is a $movg2s$ instruction and update the SPR *pto*. All the auxiliary definitions can be found in Chapter 3.

$$ins\text{-}gen(p, \vartheta) = \begin{cases} [] & eret(I) \vee invlpg(I) \vee flush(I) \vee switch(I) \vee wpto(I) \\ ins\text{-}gen(p, \vartheta, 1) & otherwise \end{cases}$$

We also overload the auxiliary predicates $fetch$ and *execute*. The parameter *eev* in predicate $fetch$ and the parameter *mode* in predicate *execute* are only used to check if there are interrupts. Since we do not consider interrupts, we give dummy value 0^{256} and 1 to *eev* and *mode* respectively. The full definition of $fetch$ and *execute* also can be found in Chapter 3.

$$fetch(p, \vartheta) \equiv fetch(p, \vartheta, 0^{256})$$
$$execute(p, \vartheta) \equiv execute(p, \vartheta, 1)$$

Definition 5.3 (Transition Function in Program Step) The transition function

$$\delta_p(p, \vartheta, is, 0^{256}) = (p', is')$$

takes a program state $p \in \mathbb{P}$, temporaries $\vartheta \in \mathbb{T} \rightharpoonup \mathbb{V} \times \mathbb{A}$, an instruction sequence $is \in \mathbb{I}^*_{usr}$ and an external input 0^{256}, and returns an updated program state $p' \in \mathbb{P}$ as well as a sequence of newly generated instructions $is' \in \mathbb{I}^*_{usr}$. (p', is') is defined iff $is = []$. We define $c' = \delta_{core}(cast(p), I, 0^{32})$ in which δ_{core} is defined in Section 5.1. Note that we use the dummy value 0^{32} to execute the instruction I and get a MIPS core configuration c'. We only use the c' to update the pc, spr_p and gpr for an uninterrupted phase 3 program step if it is no need to access

the memory, the new value computation of these components do not depend on the read results. The updating of gpr with the memory read result is postponed to the phase 5 program step.

$$p' = \begin{cases} p[fetch := 0, jisr := 0] & fetch(p, \vartheta) \\ p_{exec} & execute(p, \vartheta) \\ p_{post} & post(p, \vartheta) \end{cases}$$

$$is' = \begin{cases} [\textbf{Read}\ False\ p.pc[31:2]\ I_{p.n}\ r\ ext\ 1111\ p] & fetch(p, \vartheta) \\ ins\text{-}gen(p, \vartheta) & execute(p, \vartheta) \\ [] & post(p, \vartheta) \end{cases}$$

where we let $I_{isa} = \vartheta(I_{p.n})$ in

$p_{exec}.pc = c'.pc \quad p_{exec}.spr_p = p.spr_p \quad p_{exec}.ppc = p.pc \quad p_{exec}.fetch = 1 \quad p_{exec}.n = p.n$
$p_{exec}.jisr = 0$

$$p_{exec}.gpr = \begin{cases} p.gpr & is' \neq [] \\ c'.gpr & otherwise \end{cases}$$

and p_{post} are defined in Section 3.2.3. Note that the r field in the **Read** memory instruction is the rights for address translation. It is useless since we do not consider the address translation. We keep it for maintaining the consistency of the memory instruction format.

The rest of instantiation as well as the initial configuration constraint are analogous to what we defined in Chapter 3.

5.4 Apply SB Reudction Theorem on MIPS

This section is a simplified version of Chapter 4. In this section, we first state the simplification of the *Cosmos* model and instantiate the simplified *Cosmos* model with MIPS ISA. Then, we apply the interleaving reduction and the simulation theorem to prove the simulation between the MIPS *Cosmos* machine and the instantiated abstract machine.

5.4.1 Simplified *Cosmos* Model

Since the page tables are invisible to user program, in the machine configuration of the simplified *Cosmos* model, we do not have local page table sets. We also simplify the ownership policy in Section 4.1.6, the definition of the step information and the ownership transfer function.

Configurations

Definition 5.4 (Ownership State) The ownership state $\mathcal{G}$ (ghost state) of a *Cosmos* machine S is a pair

$$\mathcal{G} = (\mathcal{O}, \mathcal{S}) \in \mathbb{G}_S$$

where $\mathcal{O} : [0 : nu - 1] \to 2^{\mathcal{A}}$ maps unit indices to the corresponding units' sets of owned addresses (owns-set) and $\mathcal{S} \subseteq \mathcal{A}$ is the set of shared writable addresses.

Now we can define the configuration of the overall *Cosmos* machine.

Definition 5.5 (*Cosmos* Machine Configuration) A configuration C of *Cosmos* model S is given as a pair

$$C = (M, \mathcal{G}) \in \mathbb{K}_S$$

consists of machine state $M \in \mathbb{M}_S$ in Definition 4.2 and ownership state $\mathcal{G} \in \mathbb{G}_S$. We also reuse all the shorthands defined in Section 4.1.

Semantics and Step Information

Definition 5.6 (*Cosmos* Model Transition Function) For a *Cosmos* machine S, we define transition function

$$\Delta : \mathbb{K}_S \times [0 : nu - 1] \times \mathcal{E} \times (2^{\mathcal{A}})^3 \to \mathbb{M}_S$$

which takes a configuration C, a scheduling input p, an external input $in \in \mathcal{E}$, the set A of acquired addresses, the set L of acquired local addresses (which should be a subset of A) and the set R of released addresses to perform a step of unit p on its state, the common memory, and the ownership state. First, however, we consider the transition on the machine and ownership states separately. The transition Δ_t on the machine sates is defined in Definition 4.7. The transition on the ownership sates is defined as following: let

$$\mathcal{O}' = \mathcal{G}.\mathcal{O}_p \cup A \setminus R$$
$$\mathcal{S}' = \mathcal{G}.\mathcal{S} \cup R \setminus L$$

we define the ownership transfer function:

$$\Delta_o(\mathcal{G}, p, (A, L, R)) \equiv (\mathcal{G}.\mathcal{O}[p \mapsto \mathcal{O}'], \mathcal{S}')$$

Now the overall transition function for *Cosmos* machine configurations is defined by:

$$\Delta(C, p, in, (A, L, R)) \equiv (\Delta_t(C.M, p, in), \Delta_o(C.\mathcal{G}, p, (A, L, R))$$

The follow definition is a counterpart of Definition 4.8 with simplified ownership annotations.

Definition 5.7 (Step Information) We overload the set Σ_S of step information of a *Cosmos* machine S where

$$\alpha = (s, in, io, ip, A, L, R) \in \Sigma_S$$

Each component has the same meaning as in Definition 4.8. Ownership annotation is of type:

$$\Omega_S = (2^{\mathcal{A}})^3$$

Below we define projections, mapping step information α to transition information and ownership transfer information.

$$\alpha.t = (\alpha.s, \alpha.in, \alpha.io, \alpha.ip) \in \Theta_S \qquad \alpha.o = (\alpha.A, \alpha.L, \alpha.R) \in \Omega_S$$

5.4.2 MIPS Instantiation

In a later section, we will present a simulation of the MIPS *Cosmos* machine. Therefore, in this section, we present the full instantiation of the *Cosmos* model S with the MIPS ISA.

$$S = (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP})$$

- $\forall X \in \{\mathcal{A}, \mathcal{V}, nu\}.\ S^n_{\text{MIPS}}.X = S^n_{\text{MIPS-86}}.X$— The memory and the number of computational units are defined as in $S^n_{\text{MIPS-86}}$.
- $S^n_{\text{MIPS}}.\mathcal{R} = A_{code}$ — We assume that all code to be executed lies in an area $A_{code} \in A$ and we set the read-only addresses to be identical with this area.
- $S^n_{\text{MIPS}}.\mathcal{U} = K_{mips\text{-}pro} \times \mathbb{N} \times \mathbb{B} \times (\mathbb{T} \rightharpoonup \mathbb{V})$ — Every computation unit consists a sequential MIPS processor p, a counter n, a dirty bit $\mathcal{D}$ and a temporary ϑ which is a partial function from $\{I, R\} \times \mathbb{N}$ to a 32-bit value v. For all $X \in \{I, R\}$ in (X, n) we write X_n for short. Initially, for all X_n maps to $\perp$. For all $Y \in \{pc, gpr, spr\}$ we simply write $u.p.Y$ instead of $u.p.core.Y$.
- $S^n_{\text{MIPS}}.\mathcal{E} = \epsilon$ —The input of processor transition function.
- $S^n_{\text{MIPS}}.reads$ — The reads set. We let $I_{isa} = m(u.p.pc[31:2])$ then

$$\begin{aligned}
&core\text{-}reads(u, m, in)\\
&= \begin{cases} \{u.p.pc[31:2], ea(u.p.core, I_{isa})\} & load(I_{isa}) \vee rmw(I_{isa}) \\ \{u.p.pc[31:2]\} & otherwise \end{cases}\\
&S^n_{\text{MIPS}}.reads(u, m, in) = core\text{-}reads(u, m, in)
\end{aligned}$$

  The constraint on reads set is discharged analogously as in Section 4.2.

- $S^n_{\text{MIPS}}.\delta$ — As in Chapter 3, A_{io} is the set of shared memory access instruction virtual addresses. In the definition we let

$$R_{isa} = m(ea(u.p.core, I_{isa}))$$

  then we define u' and m' as:

$$\begin{aligned}
u'.p &= \delta_{mips\text{-}seq}((u.p, \lceil m \rceil), in).p\\
u'.n &= u.n + 1\\
u'.\mathcal{D} &= \begin{cases} True & store(I_{isa}) \wedge u.pc[31:2] \in A_{io} \\ False & mfence(I_{isa}) \vee rmw(I_{isa}) \\ u.\mathcal{D} & otherwise \end{cases}\\
u'.\vartheta &= \begin{cases} \vartheta' & load(I_{isa}) \vee rmw(I_{isa}) \\ u.\vartheta(I_{u.n} \mapsto I_{isa}) & otherwise \end{cases}\\
m' &= \delta_{mips\text{-}seq}((u.p, \lceil m \rceil)).m
\end{aligned}$$

where:

$$\vartheta' = u.\vartheta(I_{u.n} \mapsto I_{isa})(R_{u.n} \mapsto lv(R_{isa}, I_{isa}))$$

We define the set of written addresses $W(u, m, in)$. A write operation is performed if predicate $wr(u, m)$ holds.

$$wr(u, m) \equiv (store(I_{isa}) \vee rmw(I_{isa}) \wedge m(ea(u.p.core, I_{isa})) = u.gpr(rd(I_{isa})))$$

$$core\text{-}writes(u, m, in) = \begin{cases} \{ea(u.p.core, I_{isa})\} & wr(u, m) \\ \emptyset & otherwise \end{cases}$$

$$W(u, m, in) = core\text{-}writes(u, m, in)$$

We can define the transition function for MIPS computation units which returns the same new core configuration and the updated part of memory. We define:

$$S^n_{\text{MIPS}}.\delta(u, m, in) = (u', m'|_{W(u,\lceil m \rceil,in)})$$

- $S^n_{\text{MIPS}}.\mathcal{IO}$ — The definition of $\mathcal{IO}$ steps of the *Cosmos* machine are defined as:

$$S^n_{\text{MIPS}}.\mathcal{IO}(u, m, in) \equiv \ u.pc[31:2] \in A_{io}$$

- $S^n_{\text{MIPS}}.\mathcal{IP}$ — Similarly, the definition of $\mathcal{IP}$ steps of the *Cosmos* machine are defined as:

$$S^n_{\text{MIPS}}.\mathcal{IP}(u, m, in) \ \equiv \ u.pc[31:2] \in A_{cp}$$

Analogous to Section 4.2, we define the initial configuration and the code region invariant.

Definition 5.8 (Initial Configuration of SB reduced MIPS-86 *Cosmos* machine) For the initial configuration C^0, we have

$$\forall t, i \in [0 : np - 1].\ C^0.u_i.n = 0 \wedge C^0.u_i.\vartheta(t) = \perp$$

Definition 5.9 (Code Region Invariant 2) We define the invariant $codeinv2(C, A_{code})$, which is a counter part of Definition 4.21. It states that in all system states reachable from initial *Cosmos* machine configuration $C \in \mathbb{K}_{S^n_{\text{MIPS}}}$ instructions are only fetched from code region $A_{code} \subseteq \mathbb{B}^{30}$.

$$\forall \tau, C'.\ C \xrightarrow{\tau} C' \rightarrow \forall \alpha.\ C'.u_{\alpha.s}.p.pc[31:2] \subseteq A_{code}$$

5.4.3 Application on MIPS

In this section, we prove the simulation theorem between the abstract machine and the MIPS *Cosmos* machine. As in Section 4.3, we first perform the interleaving reduction on the abstract machine computation. Then, we prove the simulation theorem.

Interleaving Reduction

As a counter part to Definition 4.22, we define the following invariant on the abstract machine. The abstract machine does not make MMU steps, we simplify the read-only invariant as:

Definition 5.10 (Read-Only Invariant 2) Let $I = hd(c.is_{[i]})$ then

$$(c^0 \underset{\text{eev}}{\Longrightarrow}^* \ c \rightarrow c^0.ro = c.ro) \wedge (W(I) \vee RMW(I) \wedge I.cond(\vartheta')) \rightarrow I.va \notin c.ro)$$

in which ϑ' is defined in Definition 2.13.

In order to prove the interleaving reduction, we use the same strategy and the analogous lemmas as in Section 4.3 with respect to the simplified abstract machine model in Section 5.2. Compare with Section 4.3.1, the reordered abstract machine only can consist type 6 and type 7 complete interleaving blocks because we do not consider interrupts. As in Section 4.3.1, we also assume that in the interleaving-reduced abstract machine computation, only complete interleaving blocks exists. The incomplete blocks are handled in the same manner as in Section 4.3.1.

Simulation Between Abstract Machine and MIPS *Cosmos* Machine

In this subsection, we will prove the simulation between a reordered abstract machine computation and an MIPS *Cosmos* machine. First, we instantiate the safety property P in the safety condition of the *Cosmos* machine in Definition 4.19. Then, we will introduce the coupling relation and prove the simulation theorem. At last, we prove that the safety condition is transferred from the SB reduced MIPS *Cosmos* machine to the abstract machine.

Safety Property Instantiation Analogous to Section 4.3.2, we instantiate the safety property P in $safety(C, P)$ with the overloaded predicate $P_{og_{cos}^{MIPS}}$, which takes a MIPS *Cosmos* machine configuration as parameter. Because the MIPS *Cosmos* machine does not make MMU steps and it does not need external inputs other than the scheduling information, we simplify the definition in Section 4.3.2 as follows: Let

$$\begin{aligned} i &= \alpha.s \\ I_{isa} &= C.m(C.u_i.pc[31:2]) \\ \vartheta'_{cos} &= S^n_{\text{MIPS}}.\delta(C.u_i, C.m, \alpha.in).u_i.\vartheta \\ \alpha.annot_{cos} &= (\alpha.A, \alpha.L, \alpha.R) \end{aligned}$$

then

$$P_{og_{cos}^{\text{MIPS}}}(C) \equiv \alpha.io \rightarrow (load(I_{isa}) \rightarrow \neg C.u_i.\mathcal{D}) \wedge \alpha.annot_{cos} = og_{cos}^{\text{MIPS}}(C.u_i.core, \vartheta'_{cos})$$

By the predicate $P_{og_{cos}^{MIPS}}$, we guarantee that (i) the dirty bit is cleared before an $\mathcal{IO}$ read; (ii) the ownership annotations only depend on local components.

Coupling Relation Since compared with Definition 4.43, we do not consider the interrupts and address translation, the following definition of the coupling relation does not have the coupling of local page table set, *mode*, *pto* and TLB.

Definition 5.11 (Coupling Relation Between Abstract Machine and *Cosmos* machine 2) We define the coupling relation $c \sim C$ for an abstract machine configuration c and a *Cosmos* machine configuration C.

- For global components, we have:

$$\begin{aligned} c.m &= C.m \\ c.shared \setminus c.ro &= C.\mathcal{S} \\ c.ro &= \mathcal{R} \end{aligned}$$

- For thread-local components, $\forall i \in [0 : np - 1]$ we have:

$$\begin{aligned} \forall X \in \{n, pc, gpr, spr_p\}.\ c.p_{[i]}.X &= C.u_i.X \\ c.\vartheta_{[i]} &= C.u_i.\vartheta \\ c.\mathcal{D}_{[i]} &= C.u_i.\mathcal{D} \\ c.\mathcal{O}_{[i]} &= C.\mathcal{O}_i \end{aligned}$$

Simulation Theorem To prove the simulation between an interleaving-reduced abstract machine computation and a *Cosmos* machine computation inductively, we have to prove the simulation theorem for each block. Then, we prove that the safety condition is transferred from the *Cosmos* machine computation to the abstract machine computation.

According to the argument in Section 4.3.1, we have 2 kinds of complete blocks. The following series of lemmas gives us the simulation between one block of abstract machine execution and one step of *Cosmos* machine execution.

We define

$$\begin{aligned} I^2_{isa} &= c^2.\vartheta_{[i]}(I_{c^2.p_{[i]}.n}) \\ io &= S^n_{\text{MIPS}}.\mathcal{IO}(C.u_i, C.m, \alpha.in) \\ ip &= S^n_{\text{MIPS}}.\mathcal{IP}(C.u_i, C.m, \alpha.in) \end{aligned}$$

and *gen-ins* is redefined as:

$$gen\text{-}ins(I_{isa}) \equiv load(I_{isa}) \vee store(I_{isa}) \vee rmw(I_{isa}) \vee mfence(I_{isa})$$

In order to prove the simulation, we only need the following three lemmas because we only have type 6 and type 7 block in the interleaving reduced computation. Lemma 5.12 is a counterpart to Lemma 4.53. Since we do not consider *mode* and interrupts, the corresponding preconditions are dropped in the following lemma. The proof is identical to the proof of Lemma 4.53.

Lemma 5.12 (Instruction Identical 2)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \wedge phase(c,i) = 1 \wedge c \sim C \rightarrow I^2_{isa} = C.m(C.u_i.pc[31:2])$$

The following lemma is a counterpart to Lemma 4.61, the only difference in the proof is the scheduling information α and the application of Lemma 5.12 instead of Lemma 4.53.

Lemma 5.13 (Type 6 Block Simulate by *Cosmos* machine 2)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^4 \wedge phase(c,i) = 1 \wedge$$
$$\neg gen\text{-}ins(I^2_{isa}) \wedge c \sim C \rightarrow \exists \alpha.\, C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'$$

Proof Let

$$\alpha = (i, \epsilon, io, ip, \emptyset, \emptyset, \emptyset)$$

Apply Lemma 5.12, we can conclude the *Cosmos* machine fetches I^2_{isa}. The remaining proof is analogous to the proof of Lemma 4.61. □

The following lemma is a counterpart to Lemma 4.64, in which we need to argue the well-definedness of the function og. Note that compared with the og function in Chapter 2, we imposed a constraint on the og function such that $A_{pt} = R_{pt} = \emptyset$.

Lemma 5.14 (Type 7 Block Simulate by *Cosmos* machine 2)

$$c \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^1 \overset{\text{m}}{\Longrightarrow}_i c^2 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^3 \overset{\text{m}}{\Longrightarrow}_i c^4 \underset{\text{eev}}{\overset{\text{p}}{\Longrightarrow}}_i c^5 \wedge phase(c,i) = 1 \wedge$$
$$gen\text{-}ins(I^2_{isa}) \wedge c \sim C \wedge safety(C, P_{og^{MIPS}_{cos}}) \rightarrow \exists \alpha.\, C \overset{\alpha}{\mapsto} C' \wedge c^4 \sim C'$$

Proof We let

$$\alpha = (i, \epsilon, io, ip, annot_{cos})$$

in which

$$annot_{cos} = \begin{cases} og^{\text{MIPS}}_{cos}(C.u_i.core, S^n_{\text{MIPS}}.\delta(C.u_i, C.m, \alpha.in).u_i.\vartheta) & io \\ (\emptyset, \emptyset, \emptyset) & otherwise \end{cases}$$

Then, we make a case split on I^2_{isa}. From the definition of $safety(C, P_{og^{MIPS}_{cos}})$, we have

$$(A, L, R) = og^{MIPS}_{cos}(C.u_i.core, C'.u_i.\vartheta)$$

We define the value of $og(I^3.p, c^4.\vartheta_{[i]})$ with

$$(A, L, R, R)$$

By the same reason as in the proof of Lemma 4.64, the function og is well-defined. The remainder of the proof is analogous to the proof of Lemma 4.64. □

The simulation and the safety transfer can be proved analogously as in Section 4.3.2.

5.5 Order Reduction

In order to reduce the interleaving of units and justify the block scheduling, we will present the order reduction theorem in this section. We will first formally define the notion of interleaving-point ($\mathcal{IP}$) schedules. We will also state the order reduction theorem which is: arbitrary schedules can be reduced to $\mathcal{IP}$ schedules. Memory *safety* and other properties are preserved, meaning that if we prove them for all interleaving-point schedules and a given start configuration, they hold for all possible computations originating from this state. The only prerequisite is that for any computation, between two $\mathcal{IO}$-points of the same unit, this unit passes at least one interleaving-point and that in the initial state all units are in interleaving-points. This section is copied from [Bau14] and we omit all the proofs and some intermediate lemmas.

Compared with the interleaving reduction in Section 5.4.3, the order reduction theorem is more general. It can reorder an arbitrarily-interleaved *Cosmos* machine steps into a block schedule. The reason why we do not apply the order reduction theorem to reorder our abstract machine computation is that the order reduction theorem can not transfer the *Cosmos* machine safety $safety(C, P)$ to the abstract machine safety $safe\text{-}reach(c, og)$.

5.5.1 Interleaving Point Schedules

We want to consider schedules consisting of interleaved blocks of execution steps, where each block contains only steps of some unit of the *Cosmos* model. At the start of each such block, the executing unit is in an interleaving-point with respect to its $\mathcal{IP}$ predicate. Such blocks we call interleaving-point blocks or $\mathcal{IP}$ blocks. Having a schedule interleaving only such $\mathcal{IP}$ blocks is convenient for Multiprocessor ISA units when we want to apply simulation theorems, e.g., use compiler consistency and go up to the C level, later on. In this case, we would choose the interleaving-points to be exactly the compiler consistency points for the unit under consideration.

Definition 5.15 (Interleaving-Point Schedule) For $\rho \in \Sigma_S^* \cup \Theta_S^* \wedge \alpha, \beta \in \Sigma_S \cup \Theta_S$ we define the predicate

$$\mathcal{IP}sched(\rho) \equiv (\rho = \rho'\alpha\beta \rightarrow \mathcal{IP}sched(\rho'\alpha) \wedge ((\alpha.s = \beta.s) \vee \beta.ip))$$

that expresses whether the sequence exhibits an interleaving-point schedule.

This means an $\mathcal{IP}$ schedule $\rho'\alpha$ can be extended by adding a step β of

1. the same currently running unit $\alpha.s$ or
2. another unit which is currently at an interleaving-point.

Thus, in the steps of the schedule are interleaved in blocks of steps by the same unit and every block starts in an interleaving-point of its executing unit. The only exception is the first block which need not start in an interleaving-point.

We need to introduce the notions of step sub-sequences and equivalent schedule reordering in our step sequence notation.

Definition 5.16 (Step Subsequence Notation) For any step or transition information sequence $\rho, \tau \in \Sigma_S^* \cup \Theta_S^*$, $\alpha \in \Sigma_S \cup \Theta_S$ and unit index p we define the subsequence of steps of unit p as follows:

$$\rho|_p \equiv \begin{cases} \alpha\tau|_p & : \quad \rho = \alpha\tau \wedge \alpha.s = p \\ \tau|_p & : \quad \rho = \alpha\tau \wedge \alpha.s \neq p \\ \varepsilon & : \quad \text{otherwise} \end{cases}$$

In the same way, we introduce the $\mathcal{IO}$ step subsequence of ρ.

$$\rho|_{io} \equiv \begin{cases} \alpha\tau|_{io} & : \quad \rho = \alpha\tau \wedge \alpha.io \\ \tau|_{io} & : \quad \rho = \alpha\tau \wedge /\alpha.io \\ \varepsilon & : \quad \text{otherwise} \end{cases}$$

Based on the subsequence notation, we state what it means that two step sequences are equivalently reordered.

Definition 5.17 (Equivalent Reordering Relation) Given two steps or transition information sequences $\rho, \rho' \in \Sigma_S^* \cup \Theta_S^*$, we consider them equivalently reordered when the $\mathcal{IO}$-step subsequence and the step subsequences of all units are the same:

$$\rho \mathrel{\hat{=}} \rho' \equiv \rho|_{io} = \rho'|_{io} \wedge \forall p \in [0 : nu - 1].\, \rho|_p = \rho'|_p$$

We also say that ρ' is an equivalent reordering of ρ and, for any starting configuration C, that (C, ρ') is an equivalently reordered computation of (C, ρ). Note that $\hat{=}$ is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.

In a given instantiation of a *Cosmos* model, interleaving-points can be defined independently of the definition of $\mathcal{IO}$ operations. However in the reordering theorem we have the requirements that following condition holds.

Definition 5.18 ($\mathcal{IOIP}$ Condition) For any sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$, predicate $\mathcal{IOIP}(\rho)$ denotes that every unit p starts in an interleaving-point, and there is least one interleaving-point between any two $\mathcal{IO}$-points of p.

$$\begin{aligned} \mathcal{IOIP}(\rho) \quad \equiv \quad & \forall \pi, p.\, \pi = \rho|_p \neq \varepsilon \;\rightarrow \\ & \pi_1.ip \wedge (\forall \tau, \alpha, \varphi, \beta, \omega.\, \pi = \tau\alpha\varphi\beta\omega \wedge \alpha.io \wedge \beta.io \rightarrow \exists i.\, \varphi_i.ip) \end{aligned}$$

Interleaving-points must be chosen by the verification engineer instantiating the model so that they fulfill this condition. To understand the necessity of its second part, it is helpful to consider the dual of the statement which says that between two interleaving-points, there is at most one $\mathcal{IO}$ step. This reflects the well-known principle that the steps of a non-atomic operation in a concurrent program can be merged into an atomic step, as long as the operation contains at most one shared variable access [OG76].

In the following sections, we will introduce the order reduction theorem. It contains two portion:

1. for every transition sequence which fulfills the $\mathcal{IOIP}$ condition, we can find an equivalent $\mathcal{IP}$ schedule;
2. the equivalent reordering maintains the *safety* condition.

5.5.2 Reordering into Interleaving-Point Schedules

In this section, we will state the existence of equivalent $\mathcal{IP}$ schedule by proving the following lemma.

Lemma 5.19 (Interleaving-Point Schedule Existence) For every step or transition sequence θ that fulfills the $\mathcal{IO}$-interleaving-point condition, we can find an interleaving-point schedule θ' which is an equivalent reordering of θ:

$$\forall \theta \in \Theta_S^*.\ \mathcal{IOIP}(\theta) \rightarrow \exists \theta'.\ \theta' \mathrel{\hat{=}} \theta \wedge \mathcal{IP}sched(\theta')$$

Since the $\hat{=}$ only depend on the scheduling information (i.e. s and io) of each step, we can extend lemma 5.19 to the case when $\theta \in \Sigma_S^* \cup \Theta_S^*$.

5.5.3 Equivalent Reordering Preserves Safety

In this section, we will state and prove our central reordering theorem which will allow us to reorder arbitrary schedules to interleaving-point schedules preserving the effect of the corresponding computation. For safe computations, we have that all equivalently reordered computations are also safe and lead into the same configuration.

Lemma 5.20 (Safety of Reordered Computations) Let $C, C' \in \mathbb{K}_S$ be *Cosmos* model configurations and let $\sigma, \sigma' \in \Sigma_S^*$ be step sequences with $C \xrightarrow{\sigma} C'$ and $\sigma \mathrel{\hat{=}} \sigma'$ then

$$safe(C, \sigma) \ \rightarrow \ safe(C, \sigma') \wedge C \xrightarrow{\sigma'} C'$$

We have shown that ownership-safety and the effects of safe *Cosmos* machine computations are preserved by equivalent reordering. Before, we already presented that any step sequence can be equivalently reordered into an interleaving-point schedule. Thus, every safe *Cosmos* machine computation is represented by an equivalent interleaving-point schedule computation and the reasoning about systems in verification can be reduced accordingly.

5.5.4 Order Reduction Theorem

In the reduction theorem, the safety of all traces originating from a given starting configuration $C \in \mathbb{K}_S$ must be deduced from the safety of all interleaving-point schedules starting in the same configuration. As a counterpart to Definition 4.19 for the *Cosmos* machine in Section 5.4.1, we have the following predicates:

Definition 5.21 (Verified *Cosmos* machine) We define the predicate $safety_{\mathcal{IP}}(C, P)$ which expresses the same notion of verification for all $\mathcal{IP}$ schedule computations:

$$safety_{\mathcal{IP}}(C, P) \equiv \forall \theta.\ \mathcal{IP}sched(\theta) \wedge comp(C.M, \theta) \rightarrow \exists o \in \Omega_S^*.\ safe_P(C, \langle \theta, o \rangle)$$

Additionally all $\mathcal{IP}$ schedules starting in C need to fulfill the $\mathcal{IOIP}$ condition.

$$\mathcal{IOIP}_{\mathcal{IP}}(C) \equiv \forall \theta.\ \mathcal{IP}sched(\theta) \wedge comp(C.M, \theta) \rightarrow \mathcal{IOIP}(\theta)$$

Using the definitions from above the interleaving-point schedule reduction theorem can then be stated as follows.

Theorem 5.22 ($\mathcal{IP}$ Schedule Order Reduction) *For a configuration C of a* Cosmos *machine S where all $\mathcal{IP}$ schedule computations originating in C fulfill the $\mathcal{IOIP}$ condition, we can deduce safety property P and ownership-safety on all possible computations from the verification of these properties on all $\mathcal{IP}$ schedules.*

$$safety_{\mathcal{IP}}(C, P) \wedge \mathcal{IOIP}_{\mathcal{IP}}(C) \rightarrow \ safety(C, P)$$

Note that we only require safety on the order-reduced *Cosmos* model. The proof of the theorem can be found in [Bau14]. Note that to prove Theorem 5.22, we need the lemma named coverage (Lemma 24) in [Bau14]. In the proof of Lemma 24, Jonas Oberhauser found a gap in page 58 and fixed it in his on going work [PO].

5.6 C-IL Language and *Cosmos* Model Instantiation

In order to instantiate our *Cosmos* model with a higher-level language, in this section we introduce semantics for a simplified version of C. We present the C Intermediate Language (C-IL) that was developed by S. Schmaltz [SS12] in order to justify the C verification approach applied in the Verisoft XT hypervisor verification project [Sch13]. Here, for brevity, we do not give a full definition of the C-IL semantics and omit technical details like type and expression evaluation, that can be looked up in the original research documents. Instead we concentrate on the parts which are relevant for stating a compiler consistency relation and a simulation theorem between a C-IL computation and a MIPS implementation. Such a compiler consistency relation and simulation theorem was already stated by A. Shadrin for the VAMP ISA [Sha12] and we adapt it to the MIPS architecture defined above. Moreover we fix the architecture-dependent environment parameters of C-IL according to our MIPS model.

In what follows we first define the syntax and semantics of C-IL, then we instantiate a *Cosmos* machine with the C-IL semantics obtaining a concurrent C-IL model. The compiler consistency relation and the simulation theorem will be introduced in the subsequent chapter. Applying the Cosmos *model* order reduction theorem then allows to establish a model of structured parallel C-IL, where C program threads are interleaved only at volatile variable accesses. This section is also an almost literally representation of [Bau14] with the following main differences: (i)In order to simplify the compiler consistency relation, instead of a byte-addressable memory as in [Sch13] and [Bau14], the C-IL memory is defined as a word-addressable memory. (ii) To get rid of the non-determinism transition for normal function call (not external function call or compiler intrinsics), we give the default values to the local variables. (iii) We add $mfence$ as a compiler intrinsic.

5.6.1 C-IL Syntax and Semantics

As C-IL is an intermediate representation of C, C-IL programs are not written by a programmer but rather obtained from a C program by parsing and translation. This allows us to focus only on essential features of a high-level programming language and disregard a large portion

of the "syntactic sugar" found in C. In essence, a C-IL program consists of type declarations, variable declarations, and a function table. The body of each function contains C-IL statements which comprise variable assignments (that may make use of pointer arithmetic), label-based control-flow commands, and primitives for function calls and returns. Before we can give a mathematical definition of C-IL programs, we need to introduce C-IL types, values, and expressions. Moreover, there are some environment parameters for C-IL program execution.

Environment Parameters

C-IL is not defined for a certain underlying architecture, nor a given class of programs, nor a particular compiler. Thus, there are many features of the environment, e.g., the memory type, operator semantics, the composite type layout, or global variable placement, that must be seen as a parameter to program execution. In [Sch13] this information is collected in the environment parameter $\theta \in params_{\text{C-IL}}$. Here we do not list the components of θ in their entirety. On the one hand, we fix some of the environment parameters for our MIPS-based version of C-IL. This means, in particular, that:

- the endianness of the underlying architecture is *little endian* ($\theta.endianness = \textbf{little}$),
- pointers consist of 1 word, i.e., they are bit strings of length 32[2] ($\theta.size_{ptr} = 1$),
- we only consider one compiler intrinsic function[3] for executing the MIPS *rmw* and *mfence* instructions ($\theta.intrinsics$ to be defined later), and
- we only use the 32-bit primitive types and the empty type[4]($\theta.\mathbb{T}_P = \{\textbf{i32}, \textbf{u32}, \textbf{void}\}$).

On the other hand, as we do not present the technical details of expression evaluation, a lot of the environment information is irrelevant to us. Still the dependency of certain functions on the environment parameters will be visible by taking θ as an argument. In such cases, we will give explanations on what kind of environment parameters the functions are depending. Nevertheless, we will not disclose all the technical details which can be found in [Sch13].

For defining the C-IL values and transition function, however, we will need to refer to 3 environment parameters in θ specifically:

- a mapping $\theta.\mathcal{F}_{adr}$ from the set of function names to memory addresses. This compiler-dependent function is used to convert function pointers to bit strings and store them in memory, which can be useful for, e.g., setting up interrupt descriptor tables in MIPS-86 systems.
- a mapping $\theta.R_{\textbf{extern}}$ which returns a C-IL state transition relation for external procedures which are declared but not implemented within the C-IL program. A call to such a function then results in a non-deterministic transition according to the transition relation.

[2]The last 2 bits are ignored in the pointer value.

[3]According to [Sch13], compiler intrinsics are pre-defined functions whose bodies are inlined into the program code instead of creating a new stack frame, when compiling function calls. Intrinsics are external functions that are implemented in assembly language.

[4]The empty type **void** is used as a return type for functions that do not return any value.

- a mapping $\theta.alloc_{gv}$ which takes a global variable name and returns its address in the global memory. This is a partial function that needs to be defined for every global variable the program accesses. An important restriction is that the global variable base addresses specified here result in non-overlapping memory ranges for the declared global variables.

A more detailed description of these components shall be given when they are used.

Types

In general C-IL is based on the following sets of names.

- $\mathbb{V}_{\text{C-IL}}$ — names of variables
- $\mathbb{T}_C$ — names of composite (struct) types
- $\mathbb{F}$ — names fields in composite (struct) type variables
- $\mathbb{F}_{name}$ — names of functions

Then we allow the following types in C-IL.

Definition 5.23 (C-IL Types) The set $\mathbb{T}_{\text{C-IL}}$ of all possible C-IL types constructed inductively according to the case split below. For any $t \in \mathbb{T}_{\text{C-IL}}$ one of the following conditions holds.

- $t \in \{\textbf{void}, \textbf{i32}, \textbf{u32}\}$ — t is a primitive type
- $\exists t' \in \mathbb{T}_{\text{C-IL}}.\ t = \textbf{ptr}(t')$ — t is a pointer to a value of type t'
- $\exists t' \in \mathbb{T}_{\text{C-IL}}, n \in \mathbb{N}.\ t = \textbf{array}(t', n)$ — t is an array of values with type t'
- $\exists t' \in \mathbb{T}_{\text{C-IL}}, T \in (\mathbb{T}_{\text{C-IL}} \setminus \{\textbf{void}\})^*.\ t = \textbf{funptr}(t', T)$ — t is a function pointer to a function which takes a list of input parameters with non-empty types according to list T and returns a value with type t'
- $\exists t_C \in \mathbb{T}_C.\ t = \textbf{struct}\ t_C$ — t is a composite type with name t_C

For composite types, we do not store the detailed structure but just the name of the struct. As we will see later, the field definitions for all structs is part of the C-IL program and can thus be looked up there during type evaluation. Moreover the environment information θ contains a parameter to determine the offsets of struct components in memory. However, we did not formally introduce this parameter as we will not use it explicitly in the frame of this thesis.

Besides the types listed above we also have type qualifiers which give hints to the compiler how accesses to variables with a certain qualified type shall be compiled and what kind of optimizations can be applied.

Definition 5.24 (C-IL Type Qualifiers) The set of C-IL type qualifiers is defined as follows:

$$\mathbb{Q} = \{\textbf{volatile}, \textbf{const}\}$$

The type qualifier **volatile** is used in the type declaration of a variable to denote that this variable can change its value independent of the program, therefore we also use the name *volatile* variables. In particular other processes in the system like concurrent threads, interrupt handlers or devices can also access such variables. Consequently the compiler has to take care when compiling accesses to volatile variables in order to make sure that updates are actually visible to the other processes, and that read accesses do not return stale data. In other words, the value of a volatile variable should always be consistent with its implementation in shared memory. This implies that all accesses to volatile variables must be implemented by atomic operations. Thus there are limitations on the kind of optimizations the compiler can possibly apply on volatile variable accesses.

On the other hand, variables that are declared with a **const** type qualifier (*constant variables*) are supposed to keep their value and never be modified. Thus the compiler can perform more aggressive optimizations on accesses to these variables.

We need to extend our type definition to *qualified types* because in non-primitive types we might have different qualifiers on the different levels of nesting.

Definition 5.25 (Qualified C-IL Types) The set $\mathbb{T}_Q$ of all qualified types in C-IL is constructed inductively as follows. For $(q, t) \in \mathbb{T}_Q$ we have the following cases.

- The empty type is not qualified: $(q, t) = (\emptyset, \textbf{void})$
- Qualified primitive type: $q \subseteq \mathbb{Q} \wedge t \in \{\textbf{i32}, \textbf{u32}\}$
- Qualified pointer type: $q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q.\ t = \textbf{ptr}(t')$
- Qualified array type: $q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q, n \in \mathbb{N}.\ t = \textbf{array}(t', n)$
- Qualified function pointer type:

$$q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q, T \in (\mathbb{T}_Q \setminus \{(\emptyset, \textbf{void})\})^*.\ t = (q, \textbf{funptr}(t', T))$$

- Qualified struct type: $q \subseteq \mathbb{Q} \wedge \exists t_C \in \mathbb{T}_C.\ t = (q, \textbf{struct}\ t_C)$

Thus qualified types are pairs of a set of qualifiers (which may be empty) and a type which may be constructed using other qualified types. For qualified struct types, again, the qualified component declaration will be given elsewhere. Before we can define the C-IL values we need some shorthands to determine the class of a type $t \in \mathbb{T}_{\text{C-IL}}$.

$$\begin{aligned} \mathit{isint}(t) &= t \in \{\textbf{i32}, \textbf{u32}\} \\ \mathit{isptr}(t) &= \exists t'.\ t = \textbf{ptr}(t') \\ \mathit{isarray}(t) &= \exists t', n.\ t = \textbf{array}(t', n) \\ \mathit{isfunptr}(t) &= \exists t', T.\ t = \textbf{funptr}(t', T) \end{aligned}$$

Values

In this section we define sets of values for variables of the different C-IL types defined above. Note that the possible values of a variable do not depend on type qualifiers. A qualified type can be converted into an unqualified type by recursively removing the set of qualifiers leaving only the type information. Let this be done by the following function:

$$qt2t : \mathbb{T}_Q \to \mathbb{T}_{\text{C-IL}}$$

Definition 5.26 (Primitive Values) We define the set $val_{\mathbf{prim}}$ which contains all values for variables of primitive type.

$$val_{\mathbf{prim}} = \bigcup_{b \in \mathbb{B}^{32}} \{\mathbf{val}(b, \mathbf{i32}), \mathbf{val}(b, \mathbf{u32})\}$$

Primitive values consist of the constructor **val** and a 32 bit string as well as the type information whether that bit string should be interpreted as a signed (two's complement) or unsigned binary number. Note that this definition is a simplified version of the corresponding definition in [Sch13] since we only need to consider 32 bit values in our MIPS-based C-IL semantics. Observe also, that we do not define a set of values for the primitive type **void** because this type is used to denote that no value is returned by a function. Consequently in C-IL we cannot evaluate values of type **void**.

Definition 5.27 (Pointer and Array Values) The set $val_{\mathbf{ptr}}$ of values for pointers and arrays is defined as follows.

$$val_{\mathbf{ptr}} = \bigcup_{t \in \mathbb{T}_{\text{C-IL}} \wedge (isptr(t) \vee isarray(t))} \{\mathbf{val}(a, t) \mid a \in \mathbb{B}^{32}\}$$

Here we merge the values for pointers and arrays because we treat array variables as pointers to the first element of an array. Accesses to fields of an array are then resolved via pointer arithmetic in expression evaluation. References to components of local variables of a function are represented by the following values.

Definition 5.28 (Local Variable Reference Values) Let $\mathbb{V}_{\text{C-IL}}$ be the set of all variable names, then the set of values for local variable references is defined as follows.

$$val_{\mathbf{lref}} = \bigcup_{t \in \mathbb{T}_{\text{C-IL}} \wedge (isptr(t) \vee isarray(t))} \{\mathbf{lref}((v, o), i, t) \mid v \in \mathbb{V}_{\text{C-IL}} \wedge o, i \in \mathbb{N}_0\}$$

Local variables are modeled as small separate memories, i.e., lists of words, to allow for pointer arithmetic on them. Therefore in order to refer to a component of a local variable the variable name v and the component's word offset o are saved in the **lref** value. Moreover one needs to know the type t of the referenced component and the index of the function frame i in which the local variable is contained.

Concerning function pointers we distinguish between two kind of values. The first kind $val_{\mathbf{fptr}}$ is used for pointers to those functions of which we know the corresponding memory address according to $\theta.\mathcal{F}_{adr} : \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{32}$. These function pointers can be stored in memory. For function pointers to other functions $f \in \mathcal{F}_{name}$ where $\mathcal{F}_{adr}(f) = \bot$ we use symbolic values from the set $val_{\mathbf{fun}}$. Such pointers cannot be stored in memory but only be dereferenced, resulting in a call of the referenced function.

Definition 5.29 (Function Pointer Values) The two sets of values $val_{\mathbf{fptr}}$ and $val_{\mathbf{fun}}$ for C-IL function pointers are defined as follows.

$$\begin{aligned} val_{\mathbf{fptr}} &= \bigcup_{t \in \mathbb{T}_{\text{C-IL}} \wedge isfunptr(t)} \{\mathbf{val}(a, t) \mid a \in \mathbb{B}^{32}\} \\ val_{\mathbf{fun}} &= \{\mathbf{fun}(f, t) \mid f \in \mathbb{F}_{name} \wedge isfunptr(t)\} \end{aligned}$$

According to the MIPS specification in Section 5.1, the memory is word-addressable with 2^{30} addresses. As a consequence, in an implementation of the C-IL to MIPS compiler can left the least significant 2 bits of the pointer value (including non-symbolic function pointers value and array values) unused. The two extra bits can be used to contain additional data, such as an indirection bit or reference count. The pointer associated with extra data is called tagged pointer. We provide the following function to transform the value of a pointer to an address:

$$ptrv2addr(p) = \begin{cases} a[31:2] & p = \mathbf{val}(a, \mathbf{ptr}(t)) \vee p = \mathbf{val}(a, \mathbf{array}(t, n)) \vee \\ & p = \mathbf{val}(a, \mathbf{funptr}(t)) \\ \bot & otherwise \end{cases}$$

Finally the set *val* of all C-IL values is the union of primitive, pointer, local variable reference, and function pointer types.

$$val = val_{\mathbf{prim}} \cup val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup val_{\mathbf{fptr}} \cup val_{\mathbf{fun}}$$

Note that we do not have values for structs, since we can only evaluate the components of struct variables but not the complete struct.

Expressions

Expressions in C-IL are used on the left and right side of variable assignments, as conditions in control-flow statements, as function identifiers and input parameters, to determine the variable where to store the returned value of a function, as well as the return value itself. A successful evaluation of an expression returns a values from **val**. Thus expressions encode primitive values, pointers, local variable references and function pointers. In C-IL expressions we can use the following unary mathematical operators from the set $\mathbb{O}_1$ for arithmetic, binary, and logical negation.

$$\mathbb{O}_1 = \{-, \sim, !\}$$

The set $\mathbb{O}_2$ comprises all available binary mathematical operators.

$$\mathbb{O}_2 = \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, |, \hat{}, \&\&, ||\}$$

From left to right these symbols represent addition, subtraction[5], multiplication, integer division, modulo, binary left shift, right shift, less, greater, less or equal, greater or equal, equal, unequal, binary AND, OR, and XOR, as well as logical AND, and OR. Other C operators, e.g., for taking the address of a variable or pointer-dereferencing, are not considered mathematical operators. They are treated in the following definition of the structure of C-IL expressions.

[5]Note that the same symbol is used for unary and binary minus, however in the definition of expressions they are used unambiguously.

Definition 5.30 (C-IL Expression) The set $\mathbb{E}$ contains all possible C-IL expressions and is defined inductively. Every $e \in \mathbb{E}$ obeys one of the following rules.

- $e \in \mathit{val}$ — e is a constant C-IL value.
- $e \in \mathbb{V}_{\text{C-IL}}$ — e identifies a C-IL variable by its name. In expression evaluation, local variables take precedence over global variables with the same name.
- $e \in \mathbb{F}_{\mathit{name}}$ — e identifies a C-IL function by its name. Such expressions are used both for calling a function as well as creating function pointers.
- $\exists e' \in \mathbb{E}, \ominus \in \mathbb{O}_1.\ e = \ominus e'$ — e is obtained by applying a unary operator on another expression.
- $\exists e', e'' \in \mathbb{E}, \oplus \in \mathbb{O}_2.\ e = (e' \oplus e'')$ — e is obtained by combining two other expressions with a binary operator.
- $\exists c, e', e'' \in \mathbb{E}.\ e = (c\ ?\ e'\ :\ e'')$ — e consists of three sub-expressions that are combined using the ternary conditional operator. If c evaluates to a value other than zero, then e evaluates to the value of e', otherwise the value of e'' is returned.
- $\exists e' \in \mathbb{E}, t \in \mathbb{T}_{\text{C-IL}}.\ e = (t)e'$ — e represents a type cast of expression e' to type t.
- $\exists e' \in \mathbb{E}.\ e = *(e')$ — e is the value obtained from dereferencing the pointer that is encoded by expression e'
- $\exists e' \in \mathbb{E}.\ e = \&(e')$ — e is the address of the sub-variable denoted by expression e'. Sub-variables are either variables or components of variables.
- $\exists e' \in \mathbb{E}, f \in \mathbb{F}.\ e = (e').f$ — e represents the component with field name f of a struct-type variable described by expression e'
- $\exists t \in \mathbb{T}_Q.\ e = \textbf{sizeof}(t)$ — e evaluates to the size in bytes of a variable with type t.
- $\exists e' \in \mathbb{E}.\ e = \textbf{sizeof}(e')$ — e evaluates to the size in bytes of the type of expression e'.

Note that not all expressions that can be constructed using this scheme are meaningful. For instance, an expression $e \in \mathbb{V}_{\text{C-IL}}$ might reference a variable that does not exist, or an expression e' in $e = \&(e')$ might encode a constant instead of a sub-variable. The well-formedness of expressions is checked during expression evaluation.

Note moreover that $\mathbb{E}$ does not provide a dedicated operation for accessing fields of array variables. This is because the common notation $a[i]$ for accessing field i of an array variable a is just syntactic sugar for the expression $*((a + i))$. Similarly if a is a pointer to a struct-type variable then the common shorthand $a \rightarrow f$ for accessing field f of the referenced struct can be represented by the expression $(*(a)).f$.

Programs

Before we can define the structure of C-IL programs, we need to introduce the statements of the C Intermediate Language.

Definition 5.31 (C-IL Statements) The set $\mathbb{I}_{\text{C-IL}}$ contains all C-IL statements and is defined inductively. For $s \in \mathbb{I}_{\text{C-IL}}$, we have the following cases.

- $\exists e, e' \in \mathbb{E}.\ s = (e = e')$ — s is an assignment of the value encoded by expression e' to the sub-variable or memory location represented by expression e.
- $\exists l \in \mathbb{N}.\ s =$ **goto** l — s is a goto statement which redirects control-flow to label l in the current function.
- $\exists e \in \mathbb{E}, l \in \mathbb{N}.\ s =$ **ifnez** e **goto** l — s is a conditional goto statement which redirects control-flow to label l in the current function if e evaluates to a non-zero value.
- $\exists e, e' \in \mathbb{E}, E \in \mathbb{E}^*.\ s = (e' =$ **call** $e(E))$ — s represents a function call to the function identified by expression e (which must evaluate to a function pointer value), passing the input parameters according to expression list E. The value returned by the function is assigned to the sub-variable of memory location identified by expression e'.
- $\exists e \in \mathbb{E}, E \in \mathbb{E}^*.\ s =$ **call** $e(E)$ — s is a function call without return value.
- $\exists e \in \mathbb{E}.\ s =$ **return** e — s is a return statement. Executing s returns from the current function with the return value denoted by expression e.
- $s =$ **return** — s is a return statement without return value. This variant is used for functions with return type **void**.

Note that above we renamed the set of statements $\mathbb{S}$ from [Sch13] to $\mathbb{I}_{\text{C-IL}}$ in order to avoid collision with our set $\mathbb{S}$ of *Cosmos* machine signatures. The statements listed above make up the body of every C-IL function. All relevant information about the particular functions of a C-IL program is stored in a function table.

Definition 5.32 (C-IL Function Table Entry) The function table entry *fte* of a C-IL function has the following structure of type *FunT*.

$$fte = (rettype, npar, V, P) \in FunT$$

Here the components of *fte* have the following meaning:

- $rettype \in \mathbb{T}_Q$ — the type of the function's return value (return type)
- $npar \in \mathbb{N}$ — the number of input parameters for the function
- $V \in (\mathbb{V}_{\text{C-IL}} \times \mathbb{T}_Q)^*$ — a list of parameter and local variable declarations containing pairs of variable name and type, where the first *npar* entries represent the input parameters

- $P \in \mathbb{I}^*_{\text{C-IL}} \cup \{\textbf{extern}\}$ — either the function body containing a list of C-IL statements that will be executed upon function call, or the keyword **extern** which marks the function as an external function. The effect of external functions is specified by the environment parameter $\theta.R_{\textbf{extern}}$.

Now a C-IL program is defined as follows.

Definition 5.33 (C-IL Program) A C-IL program π has type $prog_{\text{C-IL}}$

$$\pi = (V_G, T_F, \mathcal{F}) \in prog_{\text{C-IL}}$$

with components:

- $V_G \in (\mathbb{V}_{\text{C-IL}} \times \mathbb{T}_Q)^*$ — declaration of global variables
- $T_F : T_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_Q)^*$ — a type table for struct types, containing the type for every field of a given struct type name
- $\mathcal{F} : \mathbb{F}_{name} \rightharpoonup FunT$ — the function table, containing the function table entries for all functions of the program

Because we have the type table $\pi.T_F$ for struct types t we can represent a struct with name t_C simply by the construction $t = \textbf{struct}\ t_C$ without need to save the concrete structure of the struct in the type. This is useful to break the cyclical definitions in many common data structures which may contain pointers to variables of their type. For instance in linked lists, a list item usually contains a pointer to the next list item. Instead of having a cyclical definition like

$$t_{list} = \textbf{struct}((v, \textbf{i32}) \circ (next, \textbf{ptr}(t_{list})))$$

one can then separately define the name and structure of the list item type:

$$t_{list} = \textbf{struct}\ item \qquad \pi.T_F(item) = (v, \textbf{i32}) \circ (next, \textbf{ptr}(t_{list}))$$

With the type table of program, we can also define the default value of each type. The default value is used to initialize the variables of type t that are created without an immediate assignment of their value. The default value for a composite type is the concatenation of the default value of each component. We let $type_i = qt2t(snd(\pi.T_F(t)_i))$ and $n = |\pi.T_F(t)|$ then

$$dft(t) = \begin{cases} \textbf{val}(0^{32}, t) & isint(t) \vee isptr(t) \vee isfunptr(t) \vee isarray(t) \\ dft(type_0) \circ ... \circ dft(type_{n-1}) & otherwise \end{cases}$$

Naturally there are a lot of well-formedness conditions on C-IL programs, for instance, that only declared sub-variables may be used, or that **goto** statements may only target labels within the bounds of the current function. In [Sch13], most of the possible faults in a C-IL program are captured as run-time errors during type evaluation, expression evaluation, and application of the C-IL transition function. However, there are a few conditions missing concerning control-flow

statements. We introduce the following predicate to denote that $s \in \mathbb{I}_{\text{C-IL}}$ is a C-IL control-flow statement which is targeting a label $l \in \mathbb{N}$.

$$ctrl(s, l) \equiv s = \textbf{goto}\ l \vee \exists e \in \mathbb{E}.\ s = \textbf{ifnez}\ e\ \textbf{goto}\ l$$

Now we can define the well-formedness conditions on C-IL programs that are not covered by the run-time error definitions of [Sch13], which will be given later.

Definition 5.34 (C-IL Program Well-formedness) We consider a C-IL program π to be well-formed if it obeys the conditions that (i) every control-flow instruction targets only labels within the corresponding function and that (ii) only functions with return type **void** omit returning a value.

$$\begin{aligned} wfprog_{\text{C-IL}}(\pi) \equiv\ & \forall f \in \mathbb{F}_{name}, s \in \mathbb{I}_{\text{C-IL}}.\ \pi.\mathcal{F}(f) \neq \bot \wedge s \in \pi.\mathcal{F}(f).P \rightarrow \\ & (i)\quad ctrl(s, l) \rightarrow l \in [0 : |\pi.\mathcal{F}(f).P| - 1] \\ & (ii)\quad s = \textbf{return} \rightarrow \pi.\mathcal{F}(f).rettype = \textbf{void} \end{aligned}$$

Note that according to [Sch13] it is allowed to use a statement **return** e for some expression $e \in \mathbb{E}$ to return from a function with return type **void**. The returned value is simply ignored then.

Configurations

Finally, we can define the configurations of the C-IL model. A C-IL configuration consists of a global memory and the current state of the stack. The stack models contain all the local information that is needed for the execution of C-IL functions. For every new function call, a stack frame with the following structure is put on the stack.

Definition 5.35 (C-IL Stack Frame) A C-IL stack frame s is a record

$$s = (\mathcal{M}_{\mathcal{E}}, rds, f, loc) \in frame_{\text{C-IL}}$$

containing the components:

- $f \in \mathbb{F}_{name}$ — the name of the function, to which the stack frame belongs.
- $\mathcal{M}_{\mathcal{E}} : \mathbb{V}_{\text{C-IL}} \rightharpoonup (\mathbb{B}^{32})^*$ — the memory for local variables and parameters. The content of a local variable or parameter is represented as a list of words, thus allowing for pointer arithmetic within the variables.
- $rds \in val_{\textbf{ptr}} \cup val_{\textbf{lref}} \cup \{\bot\}$ — the return destination for function calls from f, which contains a reference to the sub-variable where to store the return value of a called function. If the called function has return type **void** we set rds to $\bot$.
- $loc \in \mathbb{N}$ — the location counter, indexing the next statement in the function body of f to be executed.

Then the definition of a C-IL configuration is straight-forward.

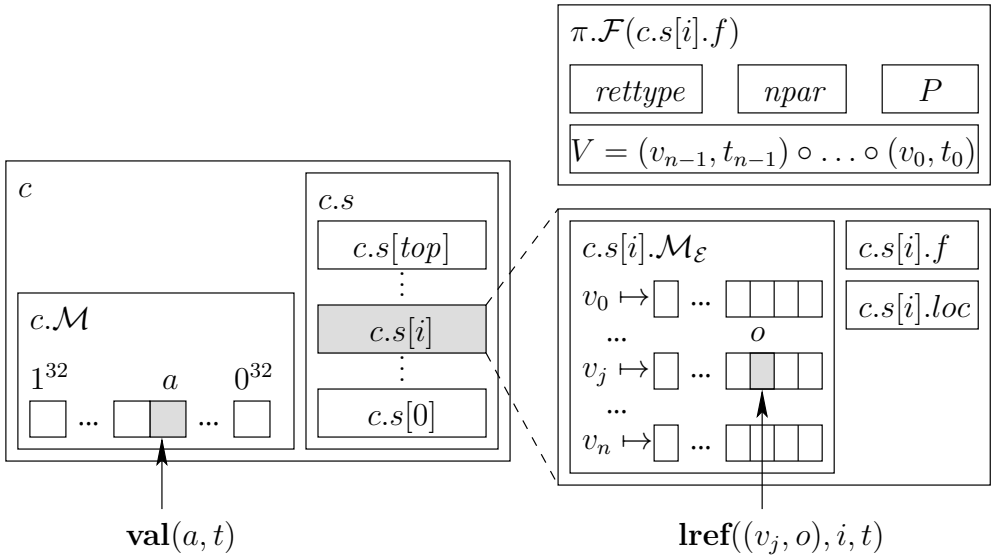


Figure 5.2: Illustration of the C-IL configurations and where pointers and local references are pointing to. This figure is copied from [Sch13] and adapted to our setting and notation. In particular, here $top = |c.s| - 1$.

Definition 5.36 (C-IL Configuration) A C-IL configuration c is a record

$$c = (\mathcal{M}, s) \in conf_{\text{C-IL}}$$

containing the components:

- $\mathcal{M} : \mathbb{B}^{30} \to \mathbb{B}^{32}$ — the global word-adressable memory,
- $s \in frame^*_{\text{C-IL}}$ — the C-IL stack, containing C-IL stack frames, where the top frame is at the end[6] of the list.

See Fig. 5.2 for an illustration. In most cases, the execution of a step of a C-IL program depends only on the top-most frame of the stack and the memory. The location pointer of the stack frame points to the statement in the corresponding function's body that shall be executed next. Global variables are located in the global memory. Moreover, there are the local variables and parameters contained in the local memory of each stack frame. Local variables and parameters obscure global variables with the same name. By using variable identifiers one can only access global memory and the local memory of the top-most frame, however using local references one can also update local memories of the lower frames in the stack. The flat word-addressable global memory can also be accessed by dereferencing plain memory addresses (pointer-arithmetic). Note, however, that, while, in fact, the stack is implemented in a part of the global memory, C-IL does not allow to access local variables or other components of the stack via pointer-arithmetic. Location and layout of stack frames are undisclosed and in the simulation theorem we will have software conditions prohibiting explicit memory accesses to the stack region.

[6]Here we differ from [Sch13] where the top frame is the head of $c.s$.

An important part in the execution of C-IL steps is the evaluation of types and expressions, where the first is useful to define the second. However, the detailed definitions of the evaluation functions are quite technical. They can be found in Section 5.8 of [Sch13]. Here we only declare and use them.

Definition 5.37 (C-IL Type and Expression Evaluation) We introduce the C-IL type evaluation function $\tau_{Q_c}^{\pi,\theta}$ which returns the type for a given C-IL expression wrt. C-IL configuration c, program π, and environment parameters θ.

$$\tau_{Q_{\cdot}^{\cdot}}^{\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup \mathbb{T}_Q$$

Similarly, we introduce the C-IL expression evaluation function $[\![\cdot]\!]_c^{\pi,\theta}$ which returns the for a given C-IL expression.

$$[\![\cdot]\!]_{\cdot}^{\cdot} : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup val$$

Both functions are defined by structural induction on the given expression. They are partial functions because not all expressions are well-formed and can be evaluated properly for a given program and C-IL configuration. In such cases the type and value of an expression e are undefined and we have $\tau_{Q_c}^{\pi,\theta}(e) = \bot$, and $[\![e]\!]_c^{\pi,\theta} = \bot$ respectively.

A typical case of an erroneous expression is a reference to a variable name that is not declared. The type evaluation of a pointer dereferencing $*(e)$ fails if e is not of the type pointer or array. Similarly, the type of an address $\&(e)$ of an expression e is only defined if e describes a subvariable or a dereferenced pointer.

In expression evaluation we have the same restrictions as above, i.e., referencing undeclared variables or function names results in an undefined value. For dereferencing a pointer there is a case distinction on the type of the pointer, thus if the type of some expression $*(e)$ is undefined so is its value. In the evaluation, it is distinguished whether the pointer points to a primitive value or an array. In the first case, one simply reads the referenced memory address, in the latter case the array itself is returned. We cannot reference or dereference complete struct-type variables, but only their primitive or array fields.

The evaluation functions depend on the environment parameters for several reasons. First the type returned by the **sizeof** operator is defined in θ. In expression evaluation, one has to know the offset of the fields in the memory representation of composite variables, which is also a compiler-dependent environment parameter. For evaluating function pointers, we need to check $\theta.\mathcal{F}_{adr}$ in order to determine which of the two function pointer values should be used. Moreover, the effects of mathematical operators and type casts are also compiler-dependent.

Before we can define the C-IL transition function, we need to make a few more definitions. Up to now we have not defined the size of types in memory. It is computed by a function

$$size_\theta : \mathbb{T}_{\text{C-IL}} \to \mathbb{N}$$

which returns the number of words occupied in memory by a value of a given type. Its definition is based on θ because the layout of struct types in memory is depending on the compiler. However for primitive and pointer types t we have $size_\theta(t) = 1$, as expected. Moreover if t is

an array of n elements with type t' then $size_\theta(t) = n \cdot size_\theta(t')$. The complete definition can be found in [Sch13]. Using the type evaluation and type size functions, we can define the following well-formedness conditions for C-IL configurations.

Definition 5.38 (Well-formedness of C-IL Configurations) To be well-formed we require for any configuration $c \in conf_{\text{C-IL}}$, program $\pi \in prog_{\text{C-IL}}$, and environment parameter $\theta \in params_{\text{C-IL}}$ that in every stack frame (i) the function belongs to this frame is defined, (ii) the sizes of local memories correspond to the variable declarations of the corresponding functions, (iii) below the top frame the type of the return destination agrees with the return type of the called function in the frame above (with higher index), and (iv) the current location never leaves the function body. Moreover (v) the program is well-formed. Given a stack $s \in frame^*_{\text{C-IL}}$ we first define:

$$
\begin{aligned}
wfs_{\text{C-IL}}(s,\pi,\theta) \;\equiv\; & \forall i \in [0 : |s| - 1]. \\
& \textit{(i)} \quad \pi.\mathcal{F}(s[i].f) \neq \bot \\
& \textit{(ii)} \quad \forall (v,t) \in \pi.\mathcal{F}(s[i].f).V \rightarrow \\
& \qquad\qquad s[i].\mathcal{M}_{\mathcal{E}}(v) \neq \bot \wedge |s[i].\mathcal{M}_{\mathcal{E}}(v)| = size_\theta(qt2t(t)) \\
& \textit{(iii)} \quad i < |s| - 1 \rightarrow \tau_{Q_c}^{\pi,\theta}(s[i].rds) = \pi.\mathcal{F}(s[i+1].f).rettype \\
& \textit{(iv)} \quad s[i].loc \in [0 : |\pi.\mathcal{F}(s[i].f)| - 1]
\end{aligned}
$$

and set $wf_{\text{C-IL}}(c,\pi,\theta) \equiv wfprog_{\text{C-IL}}(\pi) \wedge wfs_{\text{C-IL}}(c.s,\pi,\theta)$ according to (v).

Thus, the well-formedness of C-IL configurations depends only on the stack but not on the global memory. As we have introduced C-IL configurations, we can also complete the definition of the environment parameter $\theta.R_{\textbf{extern}}$.

Definition 5.39 (External Procedure Transition Relations) We use the environment parameter $\theta.R_{\textbf{extern}}$ to define the effect of external procedures whose implementation is not given by the C-IL programs. It has the following type

$$\theta.R_{\textbf{extern}} : \mathbb{F}_{name} \rightharpoonup 2^{val^* \times conf_{\text{C-IL}} \times conf_{\text{C-IL}}}$$

where for an external procedure x, such that $\pi.\mathcal{F}(x).P = \textbf{extern}$, the set $\theta.R_{\textbf{extern}}(x)$ contains tuples $((i_0,\ldots,i_{n-1}),c,c')$ with the components:

- $i_0,\ldots,i_{n-1}$ — the input parameters to the external procedure
- c, c' — the pre- and post state of the transition

In case an external procedure x is called with a list of input parameters from a C-IL configuration c, the next configuration c' is determined by non-deterministically choosing a fitting transition from $\theta.R_{\textbf{extern}}(x)$.

Closely related to external procedures are the compiler intrinsic functions that are defined by $\theta.intrinsics : \mathbb{F}_{name} \rightharpoonup FunT$. Intrinsics are predefined functions that are provided by the compiler to the programmer, usually to access certain system resources that are not visible in pure C. As anounced before the only intrinsic function considered in our scenario is *rmw* and *mfence*, which are wrapper functions for the *rmw* and $mfence$ assembly instruction. We define

$\theta.intrinsics(rmw) = fte_{rmw}$, $\theta.intrinsics(mfence) = fte_{mfence}$ in the subsequent paragraphs. For all other function names $f \notin \{rmw, mfence\}$ we have $\theta.intrinsics(f) = \bot$.

The function table entry of $mfence$ is defined as:

$$\begin{aligned} fte_{mfence}.rettype &= (\emptyset, \textbf{void}) \\ fte_{mfence}.npar &= 0 \\ fte_{mfence}.V &= \epsilon \\ fte_{mfence}.P &= \textbf{extern} \end{aligned}$$

The intrinsic function *mfence* is implemented by the assembly instruction $mfence$ and thus an external function. It does not take any parameters and only increase the *loc* in the C-IL configuration because after the SB reduction, the SB is not visible in ISA level. We need the *mfence* intrinsic to clear the dirty bit in our *Cosmos* C-IL machine configuration. Before defining the transition relation of *mfence*, we provide a function $inc_{loc} : conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$ to increment the location counter of the top stack frame. It is undefined if the stack is empty, otherwise:

$$inc_{loc}(c) = c[s := c.s[top \mapsto (c.s[top])[loc := c.s[top].loc + 1]]]$$

The transition relation $\theta.R_{\textbf{extern}}(mfence)$ is defined as:

$$\rho_{mfence} = \{((), c, c') \mid c' = inc_{loc}(c)\}$$

The function table entry of *rmw* is defined as:

$$\begin{aligned} fte_{rmw}.rettype &= (\emptyset, \textbf{void}) \\ fte_{rmw}.npar &= 4 \\ fte_{rmw}.V &= (a, (\emptyset, \textbf{ptr}(\{\textbf{volatile}\}, \textbf{i32}))) \circ (u, (\emptyset, \textbf{i32})) \\ &\quad \circ (v, (\emptyset, \textbf{i32})) \circ (r, (\emptyset, \textbf{ptr}(\emptyset, \textbf{i32}))) \\ fte_{rmw}.P &= \textbf{extern} \end{aligned}$$

The intrinsic functions *rmw* is also implemented in assembly and an external function. It takes 4 input arguments a,u,v, and r, where a is a pointer to the volatile memory location that shall be swapped, u is the value with which the memory location referenced by a is compared, and v is the value to be swapped in. The content of the memory location pointed to by a is written to the subvariable referenced by the fourth parameter[7] r. Since the intrinsics are provided by the compiler, they are not part of the program-based function table. We define the combined function table $\mathcal{F}_\pi^\theta$ as follows.

$$\mathcal{F}_\pi^\theta = \pi.\mathcal{F} \uplus \theta.intrinsics$$

Knowing the semantics of the rmw instruction of MIPS, we would also like to define the external procedure transition relation $R_{\textbf{extern}}(rmw)$. However, we first need some more notation for updating a C-IL configuration. When writing C-IL values to the global or some local memory, they have to be broken down into sequences of bytes. First we need a function

[7] Note that the rmw instruction of the MIPS ISA has only three parameters. Thus in the implementation of *rmw* an additional write instruction is needed to update the memory location referenced by r. The parameter r is used as the destination of the *rmw* return result.

$bits2words : \mathbb{B}^{32n} \to (\mathbb{B}^{32})^n$ convert a bit string whose length is a multiple of 32 into a word string.

$$bits2words(x[m:0]) = \begin{cases} bits2words(x[m:32]) \circ (x[31:0]) & : \quad m > 31 \\ (x[m:0]) & : \quad \text{otherwise} \end{cases}$$

Then the conversion from C-IL values to words is done by the following partial function.

$$val2words_\theta(v) = \begin{cases} bits2words(b) & : \quad v = \mathbf{val}(b,t) \\ \bot & : \quad \text{otherwise} \end{cases}$$

Note that this definition excludes local variable references and symbolic function pointers. For these values, the semantics does not provide a binary representation because for local subvariables and functions f where $\theta.\mathcal{F}_{adr}(f) = \bot$, the location in memory is unknown. Also, note that $val2words_\theta$ depends on the environment parameter θ because the conversion to word strings depends on the endianness of the underlying memory system. As our MIPS ISA uses little endian memory representations we simplified the definition of $val2words_\theta$ which contains a case distinction on $\theta.endianness$ in [Sch13]. We still keep the θ though, to keep the notation consistent.

Now we can introduce helper functions to write the global and local memories of a C-IL configuration. We copy the following three definitions from Section 5.7.1 of [Sch13], with the modifications that we fix the pointer size to 1 word, the global memory becomes word-addressable, and the values are 32-bits.

Definition 5.40 (Writing Word-Strings to Global Memory) We define the function

$$write_{\mathcal{M}} : (\mathbb{B}^{30} \to \mathbb{B}^{32}) \times \mathbb{B}^{30} \times (\mathbb{B}^{32})^* \to (\mathbb{B}^{30} \to \mathbb{B}^{32})$$

that writes a word-string B to a global memory $\mathcal{M}$ starting at address a such that

$$\forall x \in \mathbb{B}^{30}.\ write_{\mathcal{M}}(\mathcal{M},a,B)(x) = \begin{cases} \mathcal{M}(x) & \langle x \rangle - \langle a \rangle \notin \{0,\ldots,|B|-1\} \\ B[\langle x \rangle - \langle a \rangle] & \text{otherwise} \end{cases}$$

Definition 5.41 (Writing Word-Strings to Local Memory) We define the function

$$write_{\mathcal{E}} : (\mathbb{V}_{\text{C-IL}} \rightharpoonup (\mathbb{B}^{32})^*) \times \mathbb{V}_{\text{C-IL}} \times \mathbb{N}_0 \times (\mathbb{B}^{32})^* \rightharpoonup (\mathbb{V}_{\text{C-IL}} \rightharpoonup (\mathbb{B}^{32})^*)$$

that writes a word-string B to variable v of a local memory $\mathcal{M}_{\mathcal{E}}$ starting at offset o such that

$$\forall w \in \mathbb{V}_{\text{C-IL}}, i \in [0 : |\mathcal{M}_{\mathcal{E}}(w)| - 1].$$
$$write_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,B)(w)[i] = \begin{cases} \mathcal{M}_{\mathcal{E}}(w)[i] & w \neq v \vee i \notin \{o,\ldots,o+|B|-1\} \\ B[i-o] & \text{otherwise} \end{cases}$$

If, however, $|B| + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin \text{dom}(\mathcal{M}_{\mathcal{E}})$, the function is undefined for the given parameters.

Definition 5.42 (Writing a Value to a C-IL Configuration) We define the function

$$write : params_{\text{C-IL}} \times conf_{\text{C-IL}} \times val \times val \rightharpoonup conf_{\text{C-IL}}$$

that writes a given C-IL value y to a C-IL configuration c at the memory pointed to by pointer x according to environment parameters θ as

$$write(\theta,c,x,y) = \begin{cases} c[\mathcal{M} := write_{\mathcal{M}}(c.\mathcal{M}, a', val2words_\theta(y))] & : x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ & \quad \wedge\, y = \mathbf{val}(b, t) \\ c' & : x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ & \quad \wedge\, y = \mathbf{val}(b, t) \\ write(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & : x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ write(\theta, c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t)), y) & : x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \bot & : \text{otherwise} \end{cases}$$

where $a' = ptrv2addr(x)$, $c'.s[i].\mathcal{M}_\mathcal{E} = write_\mathcal{E}(c.s[i].\mathcal{M}_\mathcal{E}, v, o, val2words_\theta(y))]$ and all other parts of c' are identical to c.

In the first case x contains a pointer to some value in global memory of type t, and we are overwriting it with the primitive or pointer value y. When x is a local variable reference, we update the referenced variable with y in the specified local memory, starting at the given offset. Since arrays in C are treated as pointers to the first element of the array, any write operation to an array is transformed accordingly. Observe that *write* checks for type safety, i.e., that value y and the write target specified by x have the same type. Moreover, we cannot update c using symbolic function pointers for x because these pointers are not associated with any resource in c.

Now we can define the transition relation $\theta.R_{\mathbf{extern}}(rmw)$ for the *rmw* compiler intrinsic function. It consists of two subrelations depending on whether the comparison in the *rmw* instruction was successful or not. In the first case we let $a' = ptrv2addr(a)$ then:

$$\begin{aligned} \rho^{swap}_{rmw} \;=\; \{((a,u,v,r),c,c''') \;\mid\; & \exists b \in \mathbb{B}^{32}.\, u = \mathbf{val}(b, \mathbf{i32}) \wedge c.\mathcal{M}(a') = b \wedge \\ & \exists c', c'' \in conf_{\text{C-IL}}.\, c' = write(\theta, c, a', v) \wedge \\ & c'' = write(\theta, c', r, u) \wedge c''' = inc_{loc}(c'') \,\} \end{aligned}$$

The memory location pointed to by a equals the test value u. Consequently it is updated with v and its old value is stored in r. In addition, the current location in the top frame is incremented. For the failed case, when u does not equal the referenced value of a, the memory location is not updated. The rest of the transition is identical to the case above. We also let $a' = ptrv2addr(a)$ then:

$$\begin{aligned} \rho^{fail}_{rmw} \;=\; \{((a,u,v,r),c,c'') \;\mid\; & \exists b \in \mathbb{B}^{32}.\, u = \mathbf{val}(b, \mathbf{i32}) \wedge c.\mathcal{M}(a') \neq b \wedge \\ & \exists c'' \in conf_{\text{C-IL}}.\, c'' = inc_{loc}(c') \wedge \\ & c' = write(\theta, c, r, \mathbf{val}(c.\mathcal{M}(a'), \mathbf{i32}))\} \end{aligned}$$

Of course, the overall transition relation is the disjunction of both cases.

$$\theta.R_{\textbf{extern}}(rmw) \;=\; \rho^{swap}_{rmw} \cup \rho^{fail}_{rmw}$$

Before we can define the C-IL transition function, there are two more helper functions left to introduce. The first function $\tau : \mathbb{V}_{\text{C-IL}} \to \mathbb{T}$ derives the type of values.

$$\tau(x) = \begin{cases} t & : \quad x = \textbf{val}(y, t) \\ t & : \quad x = \textbf{fun}(f, t) \\ t & : \quad x = \textbf{lref}((v, o), i, t) \end{cases}$$

Last we define function $zero(\theta, x) : params_{\text{C-IL}} \times \mathbb{V}_{\text{C-IL}} \rightharpoonup \mathbb{B}$ which checks whether a given primitive or pointer value equals zero.

$$zero(\theta, x) = \begin{cases} (a = 0^{32 \cdot size_\theta(t)}) & : \quad x = \textbf{val}(a, t) \\ \bot & : \quad \text{otherwise} \end{cases}$$

We finish the introduction of the C-IL semantics with the definition of the C-IL transition function in the next sub-section. It is in great portions a literal copy of Section 5.8.3 in [Sch13].

Transition Function

For given C-IL program π and environment parameters θ, we define a partial transition function

$$\delta^{\pi,\theta}_{\text{C-IL}} : conf_{\text{C-IL}} \times \Sigma_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$$

where $\Sigma_{\text{C-IL}}$ is an input alphabet used to resolve non-deterministic choice occurring in C-IL semantics. Unlike in [Sch13] and [Bau14], there is only one kind of non-deterministic choice: due to the possible non-deterministic nature of external function calls – here, one of the possible transitions specified by relation $\theta.R_{\textbf{extern}}$ is chosen. To resolve these non-deterministic choices, our transition function gets as an input $in = \eta \in \Sigma_{\text{C-IL}}$ containing a mapping η of transition functions for computing the result of external function calls.

$$\Sigma_{\text{C-IL}} \subseteq \mathbb{F}_{name} \rightharpoonup (val^* \times conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}})$$

The inputs are either an updated C-IL configuration c' or a symbolic value $\bot$ to denote deterministic steps. However this opens up the possibility of run-time errors due to nonsensical input sequences which do not provide the right inputs for the current statement, e.g., a $\bot$ symbol instead of the required updated configuration for external function calls. Also, the choice of inputs depends on previous computation results, e.g., in case of external function calls for the previous configuration. In our model, we always provide the necessary inputs to fix *any* non-deterministic choice in the C-IL program execution, and we use an update function instead of an updated configuration for handling external function calls. If we require inputs to contain only transition functions for external function calls that implement state transitions according to $\theta.R_{\textbf{extern}}$, then we can exclude run-time errors due to a bad choice of inputs. The inputs for a computation can thus be chosen independently of the C-IL configuration, and we will only get undefined results due to programming errors. Below we formalize the restriction on $\Sigma_{\text{C-IL}}$.

Definition 5.43 (C-IL Input Constrains) We only consider input alphabets $\Sigma_{\text{C-IL}}$ that fulfill the following restrictions. For any input, we demand that η is defined for all external functions and for all arguments its result reflects the semantics specified by $\theta.R_{\textbf{extern}}$.

$$\begin{aligned}\Sigma_{\text{C-IL}} \quad = \quad \{\, \eta \mid \forall f \in \mathbb{F}_{name}.\, \mathcal{F}^{\theta}_{\pi}(f).P = \textbf{extern} \rightarrow \eta(f) \neq \bot \;\wedge \\ \forall (X, c) \in \text{dom}(\eta[f]).\, (X, c, \eta[f](X, c)) \in \theta.R_{\textbf{extern}} \,\}\end{aligned}$$

In defining the semantics of C-IL, we will use the following shorthand notation to refer to information about the topmost stack frame $top = |c.s| - 1$ in a C-IL-configuration c:

- local memory of the topmost frame: $\mathcal{M}_{\mathcal{E}_{top}}(c) = c.s[top].\mathcal{M}_{\mathcal{E}}$
- return destination of the topmost frame: $rds_{top}(c) = c.s[top].rds$
- function name of the topmost frame: $f_{top}(c) = c.s[top].f$
- location counter of the topmost frame: $loc_{top}(c) = c.s[top].loc$
- function body of the topmost frame: $P_{top}(\pi, c) = \pi.\mathcal{F}(f_{top}(c)).P$
- next statement to be executed: $stmt_{next}(\pi, c) = P_{top}(\pi, c)[loc_{top}(c)]$

Below we define functions that perform specific updates on a C-IL configuration.

Definition 5.44 (Setting the Location Counter) The function

$$set_{loc} : conf_{\text{C-IL}} \times \mathbb{N} \rightharpoonup conf_{\text{C-IL}}$$

defined as

$$set_{loc}(c,l) = c[s := (c.s)[top \mapsto (c.s[top])[loc := l]]]$$

sets the location counter of the top-most stack frame to location l.

Definition 5.45 (Removing the Topmost Frame) The function

$$drop_{frame} : conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$$

which removes the top-most stack frame from a C-IL-configuration is defined as:

$$drop_{frame}(c) = c[s := c.s[0 : top)]$$

Definition 5.46 (Setting Return Destination) We define the function

$$set_{rds} : conf_{\text{C-IL}} \times (val_{\textbf{lref}} \cup val_{\textbf{ptr}} \cup \{\bot\}) \rightharpoonup conf_{\text{C-IL}}$$

that updates the return destination component of the top most stack frame as:

$$set_{rds}(c, v) = c[s := (c.s)[top \mapsto (c.s[top])[rds := v]]]$$

Note that all of the functions defined above are only well-defined when the stack is not empty; this is why they are declared partial functions. In practice, however, executing a C-IL program always requires a non-empty stack.

Definition 5.47 (C-IL Transition Function) We define the transition function

$$\delta^{\pi,\theta}_{\text{C-IL}} : conf_{\text{C-IL}} \times \Sigma_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$$

by a case distinction on the given input:

- Deterministic step, i.e., $stmt_{next}(\pi, c) \neq \textbf{call}\ e(E)$:

$$\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = \begin{cases} inc_{loc}(c') & : stmt_{next}(\pi, c) = (e_0 = e_1) \\ set_{loc}(c, l) & : stmt_{next}(\pi, c) = \textbf{goto}\ l \\ set_{loc}(c, l) & : stmt_{next}(\pi, c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \wedge zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ inc_{loc}(c) & : stmt_{next}(\pi, c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \wedge \neg zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ c'' & : stmt_{next}(\pi, c) = \textbf{return} \\ c'' & : stmt_{next}(\pi, c) = \textbf{return}\ e \wedge rds = \bot \\ write(\theta, c'', rds, [\![e]\!]^{\pi,\theta}_c) & : stmt_{next}(\pi, c) = \textbf{return}\ e \wedge rds \neq \bot \end{cases}$$

  where $c' = write(\theta, c, [\![\&(e_0)]\!]^{\pi,\theta}_c [\![e_1]\!]^{\pi,\theta}_c)$, $c'' = drop_{frame}(c)$ and $rds = rds_{top}(c'')$. Note that for **return** the relevant return destination resides in the caller frame. Also, in case any of the terms used above is undefined due to run-time errors, we set $\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = \bot$.

- Function call:

  $\delta^{\pi,\theta}_{\text{C-IL}}(c, in)$, where all local variables are initialized with corresponding default values in the called function, is defined if and only if all of the following hold:

  - $stmt_{next}(\pi, c) = \textbf{call}\ e(E) \vee stmt_{next}(\pi, c) = (e_0 = \textbf{call}\ e(E))$ — the next statement is a function call (with or without return value),
  - $([\![e]\!]^{\pi,\theta}_c = \textbf{val}(b, \textbf{funptr}(t, T)) \rightarrow \exists f.\ f = \theta.\mathcal{F}^{-1}_{adr}(ptrv2addr(b))) \vee$ $\exists f.\ [\![e]\!]^{\pi,\theta}_c = \textbf{fun}(f, \textbf{funptr}(t, T))$ —the expression e evaluates to some function f or to a function pointer which points to f,
  - $|E| = \mathcal{F}^{\theta}_{\pi}(f).npar \wedge \forall i \in [0 : |E| - 1].\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \rightarrow \tau_{Q^{\pi,\theta}_c}(E[i]) = t$ — the types of all parameters passed match the declaration, and
  - $\mathcal{F}^{\theta}_{\pi}(f).P \neq \textbf{extern}$ — the function is not declared as external in the function table.

  Then, we define

$$\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = c'$$

  such that

$$c'.s = inc_{loc}(set_{rds}(c, rds)).s \circ (\mathcal{M}'_{\mathcal{E}}, \bot, f, 0)$$

$$c'.\mathcal{M} = c.\mathcal{M}$$

where

$$rds = \begin{cases} [\![\&(e_0)]\!]_c^{\pi,\theta} & : \quad stmt_{next}(\pi, c) = (e_0 = \textbf{call}\ e(E)) \\ \bot & : \quad stmt_{next}(\pi, c) = \textbf{call}\ e(E) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes_\theta([\![E[i]]\!]_c^{\pi,\theta}) & : \quad \exists i.\ \mathcal{F}_\pi^\theta(f).V[i] = (v,t) \wedge i < \mathcal{F}_\pi^\theta(f).npar \\ dft(t) & : \quad \exists i.\ \mathcal{F}_\pi^\theta(f).V[i] = (v,t) \wedge i \geq \mathcal{F}_\pi^\theta(f).npar \\ \bot & : \quad \text{otherwise} \end{cases}$$

- External procedure call:

  $\delta^{\pi,\theta}_{\text{C-IL}}(c, in)$ is defined if and only if all of the following hold:

  - $stmt_{next}(\pi, c) = \textbf{call}\ e(E)$ — the next statement is a function call without return value,
  - $([\![e]\!]_c^{\pi,\theta} = \textbf{val}(b, \textbf{funptr}(t, T)) \rightarrow \exists f.\ f = \theta.\mathcal{F}^{-1}_{adr}(b)) \vee$
    $\exists f.\ [\![e]\!]_c^{\pi,\theta} = \textbf{fun}(f, \textbf{funptr}(t, T))$ — expression e evaluates to some function f,
  - $|E| = \mathcal{F}_\pi^\theta(f).npar \wedge \forall i \in [0 : |E| - 1].\ \mathcal{F}_\pi^\theta(f).V[i] = (v,t) \rightarrow \tau_{Q_c}^{\pi,\theta}(E[i]) = t$ — the types of all parameters passed match the declaration,
  - $in.\eta[f]((([\![E_0]\!]_c^{\pi,\theta}, \ldots, [\![E_{|E|-1}]\!]_c^{\pi,\theta}), c) = c'$ — the external transition function for f allows a transition under given parameters E from c to c',
  - $c'.s[0 : top) = c.s[0 : top)$ — the external procedure call does not modify any stack frames other than the topmost frame,
  - $loc_{top}(c').loc = loc_{top}(c) + 1 \wedge f_{top}(c') = f_{top}(c)$ — the location counter of the topmost frame is incremented and the function is not changed,
  - $\mathcal{F}_\pi^\theta(f).P = \textbf{extern}$ — the function is declared as extern in the function table.

  Note that we restrict external function calls in such a way that they cannot be invoked with a return value. However, there is a simple way to allow an external function call to return a result: It is always possible to pass a pointer to some subvariable to which a return value from an external function call can be written.[8]

  Then,

$$\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = c'$$

C-IL Calling Convention

The call procedure in C-IL follows certain conventions that shall be described below. First of all certain general purpose registers of the MIPS processor core have special meaning. The details are depicted in Table 5.1. Note that this is a different setting than in [Sha12]. In particular the stack and base pointers are now stored in registers 29 and 30. The number of registers used for input parameters is limited to four, and the return value of a procedure call is stored in register 2. We omitted the possibility of 64-bit return values that would be stored in two registers.

[8] See the definition of the *rmw* compiler intrinsic for an example.

Alias	⟨Index⟩	Usage
zero	0	always contains 0^{32}
t_1	1	temporary values
rv	2	procedure return value
t_2	3	temporary values
$i_1 \dots i_4$	$4, \dots, 7$	input arguments for procedure calls
$t_3 \dots t_{14}$	$8 \dots 15, 24 \dots 27$	temporary values
$sv_1 \dots sv_8$	$16 \dots 23$	callee-save registers
sp	29	stack pointer
bp	30	stack frame base pointer
ra	31	return address

Table 5.1: Intended usage of the 32 MIPS general purpose registers in function calls of C-IL

Also registers 26, 27 and 28 do not have any special meaning in the framework of this thesis. We explicitly state which registers are the callee-save registers (16-23) that must be preserved across procedure calls by the programmer.

Concerning the procedure call we have four calling conventions CC.1 to CC.4. They read as follows.

CC.1 In a procedure call up to four input parameters may be passed through the general purpose registers i_1 to i_4.

CC.2 Excess parameters must be passed via the caller's *lifo* which is stack component used to store temporary data and procedure input parameters in a last in first out manner. Parameters must be pushed on the stack in reverse order such that in the end the first excess parameter, i.e., the fifth parameter, resides on the top of *lifo*. In the implementation of the stack, there is space reserved for the input parameters passed through the four registers. Thus the size of the memory region in the implementation devoted to storing the parameters of the stack frame i is always equal to $npar_i$. All excess parameters that were pushed on the stack when there were more than four inputs to procedure p are consumed (popped) from the *lifo* by a call of p and become part of the parameter space of the new stack frame.

CC.3 Before the return from a called procedure p all callee-save registers of p must be restored with the contents they had when p was entered. This must also be guaranteed for *sp*, *bp*, and *ra* by every C-IL implementation. The C-IL compiler takes care of saving and restoring these registers.

CC.4 The return value from a procedure call is passed through register *rv*.

Compilation and Stack Layout

We use the stack layout for C-IL from [Sha12], which is depicted in Fig. 5.3 and adhere to the calling conventions CC.1 to CC.4. The C-IL stack layout exhibits the following properties.

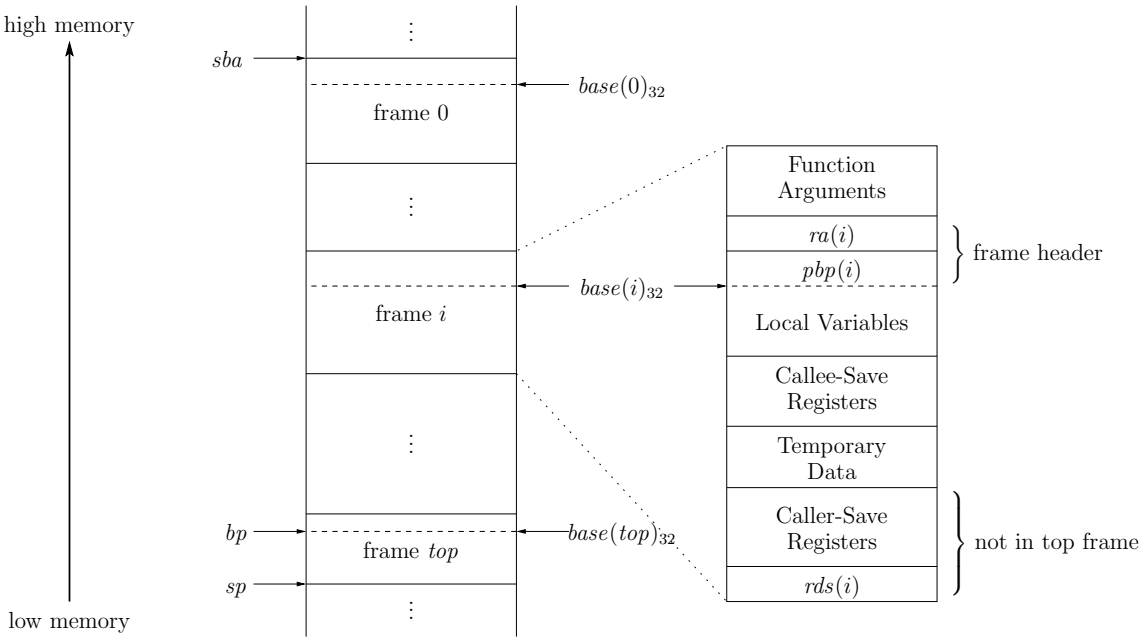


Figure 5.3: The C-IL stack layout.

- A C-IL frame i in memory is identified by a frame base address $base(i)_{32}$. The stack grows downwards starting at a given *stack base address* which is *not* identical with the frame base address of the first frame $base(0)_{32}$.

- The parameters for the function call are stored in the high end of the stack frame. They were stored here by the caller in reverse order. According to CC.1, the first four parameters are passed via registers i_1 to i_4. Nevertheless, we also reserve space for them.

- Between parameter space and the base address of a frame resides the frame header which contains the return address and the previous base pointer.

- The base pointer is stored in register *bp* and always points to the frame base address of the topmost (lowest in memory) stack frame.

- Below the frame base address, we find the region of the stack frame where the local variables are saved.

- Below the local variables the callee save registers are stored. In contrast to [Sha12] we assume for simplicity that the C-IL compiler always stores all eight callee-save registers sv_1 to sv_8 in ascending order (sv_1 is at the highest memory address).

- Below that area the compiler stores temporary values in a last-in-first-out data structure. The size of the temporary area may change dynamically during program execution. This component is the *lifo* region mentioned in the calling convention.

- The stack pointer is stored in register *sp* and always points on the lower end of the temporary data region of the topmost (lowest in memory) stack frame.
- In case a function is called, the compiler first stores the contents of the caller-save registers in the region directly below the temporary values.
- Next, the return destination (where the returned value of a function call should be saved) is stored by the caller. Parameters for the next frame are stored below.
- Caller-save registers, return destination, and the input parameters can be seen as an extension to the *lifo*-like temporary value area. Thus, we are obeying calling convention CC.2. Upon a function call from frame i the parameters become a part of the next stack frame. Thus, they are located above the base address $base(i+1)_{32}$.
- All components of the stack are word-aligned.

Before we can formalize this notion of the stack structure, we need some more information about the compilation process. Therefore, we introduce a C-IL compiler information data structure $info_{IL} \in InfoT_{\text{C-IL}}$ which have the following components for $info_{IL}$.

- $info_{IL}.code : (\mathbb{B}^{32})^*$ — a list of MIPS instructions representing the assembled C-IL program.
- $info_{IL}.cp : \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}$ — identifies the compiler consistency points for a given function and program location.
- $info_{IL}.off : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ - A function calculating the offset in the compiled code of the first instruction which implements a C-IL statement at the specified consistency point in the given function. Note that the offset 0 refers to instruction $info_{IL}.code[0]$.
- $info_{IL}.fceo : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — the offset in the compiled code of the epilogue of a function call in a given function at a given location (see explanation below)
- $info_{IL}.lvr : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{B}^5$ — specifies, if applicable, the *gpr* where a given word-sized local variable of a given function is stored in a given consistency point
- $info_{IL}.lvo : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the offset of local variables (excluding input parameters) in memory relative to the frame base for a given function and consistency point (number of words)
- $info_{IL}.csro : \mathbb{F}_{name} \times \mathbb{N} \times \mathbb{B}^5 \rightharpoonup \mathbb{N}_0$ — specifies the offset within the caller-save area where the specified register is saved by the caller during a function call in the given function and consistency point (number of words, counting relative to upper end with higher address)
- $info_{IL}.size_{CrS} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the size of the caller-save region of the stack for a given caller function and location of function call (number of words)
- $info_{IL}.size_{tmp} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the size of the temporary region of the stack for a given function and consistency point (number of words)

- $info_{IL}.cba : \mathbb{B}^{30}$ — the start address of the code region in memory
- $info_{IL}.sba : \mathbb{B}^{30}$ — the start base address
- $info_{IL}.mss : \mathbb{B}^{30}$ — the maximal size in words of the stack. We define shorthand $msp_{IL} \equiv info_{IL}.sba -_{30} info_{IL}.mss +_{30} 1_{30}$ to denote the minimal allowed value for the stack pointer.

For most of the components, it should be obvious why we need this information in order to define the C-IL compiler consistency relation. The only exception is maybe the *function call epilogue offset fceo*. A function call is not completed after the return statement is executed by the callee, because the caller still has to update the return destination with the return value passed in register *rv*. Also, the stack has to be cleared of the return destination, and caller-save registers need to be restored. The code portion in the compiled code for a function call which is implementing these tasks, we call the *function epilogue*. We need to know the start of the epilogue to define the consistency relation for the return addresses.

In the following, we introduce notation for the frame base addresses and the distances between them. First we introduce some shorthands for the components of the i-th stack frame, implicitly depending on some C-IL configuration $c_{IL} \in conf_{\text{C-IL}}$.

$$\forall x \in \{\mathcal{M}_{\mathcal{E}}, rds, f, loc\}, i \in [0 : top]. \quad x_i = c_{IL}.s[i].x$$

Moreover let $z_i \equiv \pi.\mathcal{F}(f_i).z$ for $z \in \{V, npar\}$ denote the local variable and parameter declaration list, as well as the number of parameters for f_i. The size needed for local variables and parameters on the stack can then be computed as follows.

$$\begin{aligned} size_{par}(i) &= \sum_{j=0}^{npar_i-1} size_{\theta}(qt2t(V_i[j].t)) \\ size_{lv}(i) &= \sum_{j=npar_i}^{|V_i|-1} size_{\theta}(qt2t(V_i[j].t)) \end{aligned}$$

Here for a variable declaration $v \in \mathbb{V} \times \mathbb{T}_Q$, the notation $v.t$ refers to the type component. Then we can define the distance between base addresses, or between the base address of the top stack and the base pointer respectively. It is depending on c_{IL}, π, θ and $info_{IL}$.

$$dist(i) = \begin{cases} size_{lv}(i) + 8 + info_{IL}.size_{tmp}(f_i, loc_i) & : \quad i = top \\ size_{lv}(i) + 8 + info_{IL}.size_{tmp}(f_i, loc_i) \\ + info_{IL}.size_{CrS}(f_i, loc_i) + 1 + size_{par}(f_{i+1}) + 2 & : \quad i < top \end{cases}$$

For the top frame, we only store the local variables, the eight callee-save registers and temporary data in the area bounded by the addresses stored in base pointer and stack pointer. Lower frames (with lower index and higher frame base address) are storing information for the function call associated with the stack frame lying directly above (with higher index and lower frame base address). This includes the caller-save registers, the return destination. The function input parameters and the next frame header are belonging to the callee frame. For simplicity, we do not

make a case distinction whether a function returns a value or not. We reserve the space for the return destination in both cases. Now the frame base addresses are easily defined recursively.

$$base(i) = \begin{cases} \langle info_{IL}.sba \rangle - size_{par}(f_i) - 1 & : \quad i = 0 \\ base(i-1) - dist(i-1) & : \quad i > 0 \end{cases}$$

5.6.2 C-IL Instantiation

In this section, we define a *Cosmos* machine $S^n_{\text{C-IL}} \in \mathbb{S}$ which contains n C-IL computation units working on a shared global memory. All C-IL units share the same program and environment parameters, but they are running on different stacks since each unit can be in a different program state. Hence, we have disjoint stack regions in memory with different stack base addresses but the same length. In the *Cosmos* machine definition we need some information from the compiler. The instantiation is thus based on the parameters $\pi \in Prog_{\text{C-IL}}$, $\theta \in params_{\text{C-IL}}$ and $info^p_{IL} \in InfoT_{\text{C-IL}}$ for $p \in \mathbb{N}_n$. The compiler information is equal for all units except for the stack base address.

$$\forall q, r \in [0 : n-1], c. \quad q \neq r \wedge c \neq sba \rightarrow info^q_{IL}.c = info^r_{IL}.c$$

Thus, we can refer to a common compiler information data structure $info_{IL}$ which is consistent with all $info^p_{IL}$ wrt. all components but *sba*. We define the shorthands for stack and code regions below, adapting them to the C-IL setting.

$$\begin{aligned} CR &= [info_{IL}.cba : info_{IL}.cba +_{30} bin_{30}(|info_{IL}.code|) -_{30} 1_{30}] \\ StR_p &= [info^p_{IL}.sba -_{30} info_{IL}.msp_{IL} +_{30} 1_{30} : info^p_{IL}.sba] \end{aligned}$$

Then required disjointness of stack frames in memory is denoted by:

$$\forall q, r \in [0 : n-1].\ q \neq r \rightarrow StR_q \cap StR_r = \emptyset$$

Before, we already noted that the software conditions on C-IL enforce that no global variables are allocated in the stack or code memory region. However, this is only guaranteed for global variables that are accessed in the program. For the instantiation of our *Cosmos* model, we need to make the requirement explicit. Let $StR = \bigcup_{p=0}^{n-1} StR_p$ be the complete stack region and let

$$A^\theta_{gv}(v,t) = [\theta.alloc_{gv}(v) : \theta.alloc_{gv}(v) +_{30} bin_{30}(size_\theta(qt2t(t))) -_{30} 1_{30}]$$

be the address range allocated for some global variable $v \in \mathbb{V}_{\text{C-IL}}$ of qualified type $t \in \mathbb{T}_Q$. Then we require:

$$\forall (v,t) \in \pi.V_G.\ A^\theta_{gv}(v,t) \cap (CR \cup StR) = \emptyset$$

We now define the components of $S^n_{\text{C-IL}}$ one by one.

- $S^n_{\text{C-IL}}.\mathcal{A} = \{a \in \mathbb{B}^{30} \mid a \notin CR \cup StR\}$ and $S^n_{\text{C-IL}}.\mathcal{V} = \mathbb{B}^{32}$ — we obtain the memory for the C-IL system by cutting out the forbidden address ranges for the stack and code regions from the underlying MIPS memory.

- $S^n_{\text{C-IL}}.\mathcal{R} = \{a \in \mathbb{B}^{30} \mid \exists (v,(q,t)) \in \pi.V_G.\ a \in A^{\theta}_{gv}(v,(q,t)) \wedge \textbf{const} \in q \vee a \in CR\}$ — as constants are supposed to never change their values, we should forbid writing them via the ownership policy by including them in the read-only set. This way, ownership safety guarantees the absence of writes to constant global variables, which cannot be detected by static checks of the compiler. For simplicity, we exclude here constant subvariables of global variables that are not constant. Nevertheless, note that the ownership policy cannot exclude writes to local or dynamically allocated constant variables, because, on the one hand, local variables are not allocated in the global memory and the ownership policy only governs global memory accesses. On the other hand, the set of read-only addresses is fixed in our ownership model. Thus, we cannot add new constant variables to $\mathcal{R}$. We also include the code region into the read-only set, since we neither consider a swappable code region nor self-modifying codes for simplicity.

- $S^n_{\text{C-IL}}.nu = n$ — we have n C-IL computation units.

- $S^n_{\text{C-IL}}.\mathcal{U} = frame^*_{\text{C-IL}} \cup \bot \times \mathbb{N} \times \mathbb{B} \times (\mathbb{T} \rightharpoonup \mathbb{V})$ — each C-IL computation unit consists 4 components: (i) either in a run-time error state $\bot$ or a C-IL stack component s upon which it bases its local computations (ii) similar to the instantiation with MIPS ISA, we also have a counter n (iii) a dirty bit $\mathcal{D}$ to maintain the program discipline of our store buffer reduction theorem (iv) a temporary ϑ which is a partial function from $\{I,R\} \times \mathbb{N}$ to a 32-bit value. For all $X \in \{I,R\}$ in (X,n) we also write X_n for short. Initially, every X_n maps to $\bot$. From the definition of the *Cosmos* machine transition function in later paragraphs, we will only update R_n and $\forall n.\ I_n$ always maps to $\bot$.

- $S^n_{\text{C-IL}}.\mathcal{E} = \Sigma_{\text{C-IL}}$ — The external input alphabet for the C-IL transition function is also suitable for the C-IL *Cosmos* machine.

- $S^n_{\text{C-IL}}.reads$ — We need to specify the explicit read accesses to global memory that are associated with the next C-IL step of a given unit. First we introduce a function to compute the memory region occupied by referenced global subvariables.

  Definition 5.48 (Footprint Function for Global Subvariables) Let $a \in \mathbb{B}^{30}$ be an address that a pointer variable points to and $t \in \{\textbf{ptr}(t'), \textbf{array}(t',n)\}$ the type of that pointer. Then the memory footprint of the referenced subvariable is computed by the following function.

$$fp_\theta(a,t) = \begin{cases} [a : a +_{30} bin_{30}(size_\theta(t'))) & : \quad /isarray(t') \\ \emptyset & : \quad \text{otherwise} \end{cases}$$

  Arrays cannot be accessed as a whole, we only read their elements using pointer arithmetic. Therefore, we define the memory footprint of array variables to be empty.

  Definition 5.49 (Global Memory Footprint of C-IL Expressions) Function

$$A^{\cdot}_{\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup 2^{[0:2^{30})}$$

computes the set of global memory addresses that are accessed when evaluating a given C-IL expression e wrt. some C-IL configuration c, program π, and environment parameters θ as follows. Let $sv(e) \equiv e \in \mathbb{V} \vee \exists e' \in \mathbb{E}, f \in \mathbb{F}.\ e = (e').f$ (i.e., e is a subvariable) in:

$$A_c^{\pi,\theta}(e) = \begin{cases} \emptyset & : \quad e \in val \cup \mathbb{F}_{name} \vee \exists t \in \mathbb{T}_Q.\ e = \textbf{sizeof}(t) \\ & \quad \vee\ sv(e) \wedge [\![\&(e)]\!]_c^{\pi,\theta} \in val_{\textbf{lref}} \\ & \quad \vee\ \exists e' \in \mathbb{E}.\ e \in \{\&(e'), \textbf{sizeof}(e')\} \wedge sv(e') \\ fp_\theta(a,t) & : \quad sv(e) \wedge [\![\&(e)]\!]_c^{\pi,\theta} = p \in val_{\textbf{ptr}} \\ A_c^{\pi,\theta}(e') & : \quad \exists \ominus \in \mathbb{O}_1, t \in \mathbb{T}_Q.\ e \in \{\ominus e', e = (t)e', e = \&(*(e'))\} \\ & \quad \vee\ e = *(e') \wedge [\![e']\!]_c^{\pi,\theta} \in val_{\textbf{lref}} \ \vee\ e = \textbf{sizeof}(*(e')) \\ A_c^{\pi,\theta}(x) \cup A_c^{\pi,\theta}(e') & : \quad \exists e'' \in \mathbb{E} \wedge e = (x\ ?\ e' : e'') \wedge zero(\theta, [\![x]\!]_c^{\pi,\theta}) \\ A_c^{\pi,\theta}(x) \cup A_c^{\pi,\theta}(e'') & : \quad \exists e' \in \mathbb{E} \wedge e = (x\ ?\ e' : e'') \wedge /zero(\theta, [\![x]\!]_c^{\pi,\theta}) \\ A_c^{\pi,\theta}(e') \cup A_c^{\pi,\theta}(e'') & : \quad \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\ A_c^{\pi,\theta}(e') \cup fp_\theta(a',t) & : \quad e = *(e') \wedge [\![e']\!]_c^{\pi,\theta} = p' \in val_{\textbf{ptr}} \\ \bot & : \quad \text{otherwise} \end{cases}$$

where:

$$p = \textbf{val}(x,t) \qquad a = ptrv2addr(p)$$
$$p' = \textbf{val}(x',t) \qquad a' = ptrv2addr(p')$$

The definition is straight-forward for most of the cases. Unlike global subvariables, C-IL values and function names are not associated with any memory address. The same holds for local subvariables. Looking up addresses and type sizes does not touch memory either. In order to dereference a pointer to a global memory location, one must evaluate the address to be read, but also read the memory region referenced by that typed pointer. We need another predicate to detect whether some expression encodes a reference to the *rmw* intrinsic function or the *mfence* intrinsic function. Let the type signature of the *rmw* intrinsic be denoted by $t_{rmw} = \textbf{funptr}(\textbf{void}, \textbf{ptr}(\textbf{i32}) \circ \textbf{i32} \circ \textbf{i32} \circ \textbf{ptr}(\textbf{i32}))$. Then we define:

$$rmw_c^{\pi,\theta}(e) \equiv \exists b \in \mathbb{B}^{32}.\ [\![e]\!]_c^{\pi,\theta} = \textbf{val}(b, t_{rmw}) \wedge \theta.\mathcal{F}_{adr}^{-1}(b) = rmw \vee [\![e]\!]_c^{\pi,\theta} = \textbf{fun}(rmw, t_{rmw})$$

Let the type signature of the *mfence* intrinsic be denoted by $t_{mfence} = \textbf{funptr}(\textbf{void}, \epsilon)$. Then we define:

$$mfence_c^{\pi,\theta}(e) \equiv \exists b \in \mathbb{B}^{32}.\ [\![e]\!]_c^{\pi,\theta} = \textbf{val}(b, t_{mfence}) \ \wedge \theta.\mathcal{F}_{adr}^{-1}(b) = mfence \vee$$
$$[\![e]\!]_c^{\pi,\theta} = \textbf{fun}(mfence, t_{mfence})$$

Now the memory footprint of a C-IL statement is easily defined using the expression footprint notation.

Definition 5.50 (Memory Footprint of C-IL statements) We overload the definition of function $A_c^{\pi,\theta}$ from above to cover also C-IL statements $s \in \mathbb{I}_{\text{C-IL}}$. Let $A_E = \bigcup_{e \in E} A_c^{\pi,\theta}(e)$

for $E \in \mathbb{E}^*$ as well as $A_{rmw} = A_c^{\pi,\theta}(*a) \cup A_c^{\pi,\theta}(u) \cup A_c^{\pi,\theta}(v) \cup A_c^{\pi,\theta}(*r)$ in:

$$A_c^{\pi,\theta}(s) = \begin{cases} \emptyset & : \quad s = \textbf{return} \vee \exists l \in \mathbb{N}.\ s = \textbf{goto}\ l \vee \\ & \quad\ s = \textbf{call}\ e() \wedge mfence_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(e) & : \quad \exists l \in \mathbb{N}.\ s = \textbf{ifnez}\ e\ \textbf{goto}\ l \vee \\ & \quad\ s = \textbf{return}\ e \wedge rds(top{-}1) \in val_{\textbf{lref}} \\ A_c^{\pi,\theta}(e) \cup fp_\theta(a,t) & : \quad s = \textbf{return}\ e \wedge rds(top{-}1) = p \in val_{\textbf{ptr}} \\ A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(e') & : \quad s = (e = e') \\ A_c^{\pi,\theta}(e) \cup A_E & : \quad s = \textbf{call}\ e(E) \wedge /(rmw_c^{\pi,\theta}(e) \vee mfence_c^{\pi,\theta}(e)) \\ A_c^{\pi,\theta}(e) \cup A_{rmw} & : \quad s = \textbf{call}\ e(a,u,v,r) \wedge rmw_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(e') \cup A_E & : \quad s = (e' = \textbf{call}\ e(E)) \\ \bot & : \quad \text{otherwise} \end{cases}$$

where:

$$p = \textbf{val}(x,t) \qquad a = ptrv2addr(p)$$

For most statements, the footprint of the C-IL statements only depends on the expressions they are containing. Only the return statement which returns a value writes additional memory cells. In the special case of *rmw* we know from its semantics that also the memory locations referenced by inputs a and r are accessed.

We introduce the reads-set function for C-IL statements

$$R_{\cdot,\cdot}^{\cdot,\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{I}_{\text{C-IL}} \rightharpoonup 2^{[0:2^{30})}$$

which defined similarly to the memory footprint of C-IL statements but excludes write accesses. Note that the global memory is only updated by variable assignments, return statements with a return value and the *rmw* primitive. For all other statements, we can use the memory footprint function defined earlier.

In case of a *rmw* intrinsic function call of $rmw(a,u,v,r)$, also the memory location referenced by a is read but the target of r is only written.

For the return statements, we simply exclude the memory location referenced by the *rds* component of the previous stack frame to determine the corresponding reads-set.

For assignments, we need to exclude the written memory cells which are specified in the left-hand side of the assignment. However, we cannot simply exclude the left-hand side expression from the computation of the reads-set since there might be read accesses necessary in order to evaluate it. Therefore we perform a case distinction on e in $s = (e = e')$:

1. e is a plain variable identifier — Then no additional global memory cells need to be read in order to obtain the variable's address in memory.

2. e is dereferencing a pointer expression — Then there might be further memory reads necessary in order to evaluate the pointer expression. However, the referenced memory location is not added to the reads-set explicitly. It still might occur in the reads-set though if it contributes to the evaluation of the pointer expression.
3. e contains either a redundant &-$*$-combination or references a field of a subvariable — In the first case expression evaluation simply discards the redundant &-$*$-combination. In the latter case, one first has to evaluate the referenced subvariable before the address of the field can be computed. In both cases, the expression evaluation step does not require memory accesses. Hence, we continue to compute the reads-set recursively with the inner sub-expression which must specify a subvariable.

Using $A'_{rmw} = A_c^{\pi,\theta}(*a) \cup A_c^{\pi,\theta}(u) \cup A_c^{\pi,\theta}(v) \cup A_c^{\pi,\theta}(r)$ we define these ideas formally:

$$R_c^{\pi,\theta}(s) = \begin{cases} A_c^{\pi,\theta}(e') & : \exists e \in \mathbb{V}.\ s = (e = e') \\ A_c^{\pi,\theta}(e') \cup A_c^{\pi,\theta}(e'') & : s = (*(e'') = e') \\ R_c^{\pi,\theta}((e = e')) & : \exists f \in \mathbb{F}.\ s \in \{(\&(*(e)) = e'), ((e).f = e')\} \\ A_c^{\pi,\theta}(e) & : s = \textbf{return}\ e \\ A_c^{\pi,\theta}(e) \cup A'_{rmw} & : s = \textbf{call}\ e(a,u,v,r) \wedge rmw_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(s) & : \text{otherwise} \end{cases}$$

Remember that C-IL configurations $c_{IL} = (\mathcal{M}, s)$ consist of a memory $\mathcal{M} : \mathbb{B}^{30} \to \mathbb{B}^{32}$ and a stack $s \in frame^*_{\text{C-IL}}$, thus a pair $(\lceil m \rceil, u.s)$ consisting of a completed partial *Cosmos* machine memory $m : \mathcal{A} \rightharpoonup \mathcal{V}$ and the stack component in a unit configuration $u.s$ represents a proper C-IL configuration. Now the *reads* function of the *Cosmos* machine can easily be instantiated.

$$S^n_{\text{C-IL}}.reads(u, m, in) = R^{\pi,\theta}_{(\lceil m \rceil, u.s)}(stmt_{next}(\pi, (\lceil m \rceil, u.s)))$$

- $S^n_{\text{C-IL}}.\mathcal{IO}$ — There are two kinds of $\mathcal{IO}$ steps in C-IL. We consider the use of the *rmw* mechanism as an $\mathcal{IO}$ step. Moreover, we include accesses to volatile variables. First we define a predicate to recursively detect if there is a volatile pointer dereference in expression e which is evaluated in the context of a function f of C-IL program π wrt. environment parameter θ

Definition 5.51 (Expression Contains Volatile Pointer Dereference)

$$derefvol_f^{\pi,\theta}(e) \equiv$$
$$\begin{cases} derefvol_f^{\pi,\theta}(e') & : (\exists \ominus \in \mathbb{O}_1.\ e = \ominus e') \vee e = \&(*(e')) \\ & \quad \vee e = \textbf{sizeof}(e') \vee e = (t)e' \\ & \quad \vee \exists f' \in \mathbb{F}.\ e = e'.f' \\ derefvol_f^{\pi,\theta}(e') \vee derefvol_f^{\pi,\theta}(e'') & : \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\ c & : e = (e'\ ?\ e'' : e''') \\ q' = \textbf{volatile} & : e = *(e') \wedge \tau_{Q_f}^{\pi,\theta}(e') = (q', \textbf{ptr}(q,t)) \\ False & : otherwise \end{cases}$$

where

$$c \equiv \mathit{derefvol}_f^{\pi,\theta}(e') \vee \mathit{derefvol}_f^{\pi,\theta}(e'') \vee \mathit{derefvol}_f^{\pi,\theta}(e''')$$

Similarly, we can define another predicate to detect accesses to volatile variables in expression e.

Definition 5.52 (Expression Contains Volatile Variables)

$$\mathit{vol}_f^{\pi,\theta}(e) \equiv$$

$$\begin{cases} \textbf{volatile} = q & : \quad e \in \mathbb{V} \wedge \tau_{\mathcal{Q}f}^{\pi,\theta}(e) = (q,t) \\ \mathit{vol}_f^{\pi,\theta}(e') & : \quad (\exists \ominus \in \mathbb{O}_1.\ e = \ominus e') \vee e = \&(*(e')) \\ & \quad \vee e = \textbf{sizeof}(e') \vee e = (t)e' \\ \mathit{vol}_f^{\pi,\theta}(e') \vee \mathit{vol}_f^{\pi,\theta}(e'') & : \quad \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\ \mathit{vol}_f^{\pi,\theta}(e') \vee \mathit{vol}_f^{\pi,\theta}(e'') \vee \mathit{vol}_f^{\pi,\theta}(e''') & : \quad e = (e'\ ?\ e''\ :\ e''') \\ \textbf{volatile} = q \vee \mathit{vol}_f^{\pi,\theta}(e') & : \quad e = *(e')\ \wedge \tau_{\mathcal{Q}f}^{\pi,\theta}(e') = (q', \textbf{ptr}(q,t)) \\ & \quad \vee e = (e').f' \wedge \tau_{\mathcal{Q}f}^{\pi,\theta}(e) = (q,t) \\ \mathit{False} & : \quad \text{otherwise} \end{cases}$$

Note that the evaluation of constants, function names, addresses of variables, and type casts do not require volatile variable accesses, in general. For most of the other cases, the above definition meets what one would expect intuitively. Nevertheless, there are two cases worth mentioning.

First, as pointer dereferencing may involve two accesses to memory, there are also two possibilities for a volatile access. Both the pointer as well as the referenced subvariable might be volatile. Considering field accesses, we also have several possibilities. On the one hand, the field itself might be declared volatile. On the other hand, the contained struct may be volatile, or the evaluation of the reference to that containing struct variable may involve volatile accesses, respectively.

Note that according to the definition, a C-IL statement may contain more than one volatile variable access. Nevertheless, as updates to volatile variables must be implemented as atomic operations, for simplicity we restrict that there may be at most one volatile variable access per C-IL statement. We set up the following rules which will be enforced by the ownership discipline when defining the $\mathcal{IO}$ predicate accordingly.

- Volatile variables may only be accessed in assignment statements or by the intrinsic function *rmw*.
- Per assignment statement there may be only one access to a volatile variable.
- The right-hand side of assignments with a volatile read is either a volatile variable identifier or it is dereferencing a pointer expression which is *either* volatile *or* pointing to a volatile variable.

– The left-hand side of assignments is a volatile read only when it contains a volatile pointer dereference sub-expression.

– The left-hand side of assignments with a volatile write, is either a volatile variable identifier or it is dereferencing a non-volatile pointer expression which is pointing to a volatile variable.

For an expression e, we define the predicate $no2vol_f^{\pi,\theta}(e)$ to exclude multiple volatile access.

$$
\begin{aligned}
&no2vol_f^{\pi,\theta}(e) \equiv \\
&\begin{cases}
False & : \quad e \in \mathbb{V} \wedge \tau_{Q_f}^{\pi,\theta}(e) = (q,t) \\
no2vol_f^{\pi,\theta}(e') & : \quad (\exists \ominus \in \mathbb{O}_1.\ e = \ominus e') \vee e = \&(*(e')) \\
 & \qquad \vee e = \textbf{sizeof}(e') \vee e = (t)e' \\
 & \qquad \vee \exists f' \in \mathbb{F}.\ e = e.f' \\
c_2 & : \quad \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\
c_3 & : \quad e = (e'\ ?\ e'' : e''') \\
\neg(q = \textbf{volatile} \wedge q' = \textbf{volatile}) & : \quad e = *(e') \wedge \tau_{Q_f}^{\pi,\theta}(e') = (q', \textbf{ptr}(q,t))
\end{cases}
\end{aligned}
$$

where:

$$
\begin{aligned}
c_2 \equiv & \neg(vol_f^{\pi,\theta}(e') \wedge vol_f^{\pi,\theta}(e'')) \wedge no2vol_f^{\pi,\theta}(e') \wedge no2vol_f^{\pi,\theta}(e'') \\
c_3 \equiv & no2vol_f^{\pi,\theta}(e') \wedge no2vol_f^{\pi,\theta}(e'') \wedge no2vol_f^{\pi,\theta}(e''') \wedge \\
& \neg(vol_f^{\pi,\theta}(e') \wedge vol_f^{\pi,\theta}(e'')) \wedge \\
& \neg(vol_f^{\pi,\theta}(e') \wedge vol_f^{\pi,\theta}(e''')) \wedge \\
& \neg(vol_f^{\pi,\theta}(e'') \wedge vol_f^{\pi,\theta}(e'''))
\end{aligned}
$$

We define a similar predicate for evaluating expressions in the top frame of configuration c_{IL}.

$$
no2vol_{c_{IL}}^{\pi,\theta}(e) \ \equiv \ no2vol_{f_{top}(c_{IL})}^{\pi,\theta}(e)
$$

Below we can now define another predicate to statically detect volatile variable read or write accesses in C-IL assignment statements of a given program.

Definition 5.53 (Statement Accesses Volatile Variables) Given a C-IL statement s that is executed in the context of a function f of C-IL program π wrt. environments parameters θ. Then f accesses a volatile variable in case the following predicate is fulfilled.

$$volr_f^{\pi,\theta}(s) \equiv \begin{cases} derefvol_f^{\pi,\theta}(e) \vee vol_f^{\pi,\theta}(e') & : \quad s \equiv (e = e') \\ False & : \quad \text{otherwise} \end{cases}$$

$$volw_f^{\pi,\theta}(s) \equiv \begin{cases} \neg derefvol_f^{\pi,\theta}(e) \wedge vol_f^{\pi,\theta}(e) & : \quad s \equiv (e = e') \\ False & : \quad \text{otherwise} \end{cases}$$

$$vol_f^{\pi,\theta}(s) \equiv volr_f^{\pi,\theta}(s) \vee volw_f^{\pi,\theta}(s)$$

Above, we already introduced the predicate $vol_f^{\pi,\theta}$ which scans expressions of a C-IL function f recursively for volatile variable accesses. We derive a similar predicate for evaluating expressions and statements in the top frame of configuration c_{IL}.

$$volr_{c_{IL}}^{\pi,\theta}(s) \equiv vol_{f_{top}(c_{IL})}^{\pi,\theta}(s)$$
$$volw_{c_{IL}}^{\pi,\theta}(s) \equiv vol_{f_{top}(c_{IL})}^{\pi,\theta}(s)$$
$$vol_{c_{IL}}^{\pi,\theta}(s) \equiv vol_{f_{top}(c_{IL})}^{\pi,\theta}(s)$$
$$vol_{c_{IL}}^{\pi,\theta}(e) \equiv vol_{f_{top}(c_{IL})}^{\pi,\theta}(e)$$

Now we can define the $\mathcal{IO}$ predicate, formalizing the rules stated above.

$$\begin{aligned} &S^n_{\text{C-IL}}.\mathcal{IO}(u, m, in) = 1 \quad \longleftrightarrow \\ &\quad \exists e, e', e'' \in \mathbb{E}, E \in \mathbb{E}^*. \\ &\qquad stmt_{next}(\pi, (\lceil m \rceil, u.s)) = \textbf{call}\ e(E) \wedge rmw_{(\lceil m \rceil, u)}^{\pi,\theta}(e) \\ &\qquad \vee\ stmt_{next}(\pi, (\lceil m \rceil, u.s)) \in \{(e = e'), (e' = e)\} \\ &\qquad\quad \wedge / vol_{c_{IL}}^{\pi,\theta}(e) \wedge vol_{c_{IL}}^{\pi,\theta}(e') \wedge no2vol_{c_{IL}}^{\pi,\theta}(e') \end{aligned}$$

Note that any access to shared memory which does not obey the rules above will not be considered an $\mathcal{IO}$ step and thus be unsafe according to the ownership memory access policy.

- $S^n_{\text{C-IL}}.\delta$ — We simply use the C-IL transition function $\delta^{\pi,\theta}_{\text{C-IL}}$ in the instantiation of the transistion function for the C-IL computation units. Again, we need to fill the partial memory that is given to the $S^n_{\text{C-IL}}.\delta$ as an input with dummy values, so that we can apply $\delta^{\pi,\theta}_{\text{C-IL}}$ on it. Moreover, we need to define the writes-set for a given C-IL statement s because the output memory of the transition function needs to be restricted to this set. As noted above, only assigments, *rmw*, and certain return statements may modify the global memory. Let $X = (\llbracket a \rrbracket_{c_{IL}}^{\pi,\theta}, \llbracket u \rrbracket_{c_{IL}}^{\pi,\theta}, \llbracket v \rrbracket_{c_{IL}}^{\pi,\theta}, \llbracket r \rrbracket_{c_{IL}}^{\pi,\theta})$ in the predicate

$$\begin{aligned} rmw_{c_{IL}}^{\pi,\theta}(s, a, u, v, r, \rho, in) \quad \equiv \quad & s = \textbf{call}\ e(a, u, v, r) \wedge rmw_{c_{IL}}^{\pi,\theta}(e) \\ & \wedge\ (X, c_{IL}, in.\eta[rmw](X, c_{IL})) \in \rho \end{aligned}$$

which denotes that statement s is a call to the *rmw* intrinsic function with the specified input parameters, and that the external function call has an effect according to transition relation $\rho \in \theta.R_{\mathbf{extern}}$. Thus we define the writes-set for a given C-IL statement.

$$
\begin{aligned}
&W^{\pi,\theta}_{c_{IL}}(s, in) \;=\; \\
&\quad\begin{cases}
fp_\theta(a,t) & : \exists e, e' \in \mathbb{E}.\ s = (e = e') \wedge [\![\&(e)]\!]^{\pi,\theta}_{c_{IL}} = p \in val_{\mathbf{ptr}} \\
& \quad \vee\ s = \mathbf{return}\ e \wedge rds_{top-1} = \mathbf{val}(z,t) \in val_{\mathbf{ptr}} \\
fp_\theta(p_1,t) \cup fp_\theta(p_2,t) & : \exists a,u,v,r \in \mathbb{E}.\ rmw^{\pi,\theta}_{c_{IL}}(s,a,u,v,r,\rho^{swap}_{rmw}, in) \\
& \quad \wedge\ [\![a]\!]^{\pi,\theta}_{c_{IL}} = \mathbf{val}(x,t) \wedge [\![r]\!]^{\pi,\theta}_{c_{IL}} = \mathbf{val}(y,t) \wedge t = \mathbf{ptr}(\mathbf{i32}) \\
fp_\theta(p_1, \mathbf{ptr}(\mathbf{i32})) & : \exists a,u,v,r \in \mathbb{E}.\ rmw^{\pi,\theta}_{c_{IL}}(s,a,u,v,r,\rho^{swap}_{rmw}, in) \\
& \quad \wedge\ [\![a]\!]^{\pi,\theta}_{c_{IL}} = \mathbf{val}(x, \mathbf{ptr}(\mathbf{i32})) \wedge [\![r]\!]^{\pi,\theta}_{c_{IL}} \in val_{\mathbf{lref}} \\
fp_\theta(p_2, \mathbf{ptr}(\mathbf{i32})) & : \exists a,u,v,r \in \mathbb{E}.\ rmw^{\pi,\theta}_{c_{IL}}(s,a,u,v,r,\rho^{fail}_{rmw}, in) \\
& \quad \wedge\ [\![r]\!]^{\pi,\theta}_{c_{IL}} = \mathbf{val}(y, \mathbf{ptr}(\mathbf{i32})) \\
\emptyset & : \quad \text{otherwise}
\end{cases}
\end{aligned}
$$

where

$$
\begin{aligned}
a &= ptrv2addr(\mathbf{val}(z,t)) \\
p_1 &= ptrv2addr(\mathbf{val}(x, \mathbf{ptr}(\mathbf{i32}))) \\
p_2 &= ptrv2addr(\mathbf{val}(y, \mathbf{ptr}(\mathbf{i32})))
\end{aligned}
$$

In the first case either execution is returning from a function call with a return value that is written to the memory cells specified by the return destination of the caller function frame, or we have an assignment to a memory location. The remaining cases deal with the various outcomes of a *rmw* intrinsic function call. If the comparison was successful, the targeted shared memory location is written. Also, we must distinguish whether the value read for comparison is returned to a local or global variable. Only in the latter case the variable update contributes to the writes-set.

Now we can define the transition function for C-IL computation units with the following case distinction. Let $stmt = stmt_{next}(\pi, (\lceil m \rceil, u.s)$ and $W = W^{\pi,\theta}_{(\lceil m \rceil, u)}(stmt, in)$ in:

$$
S^n_{\text{C-IL}}.\delta(u, m, in) = (m'|_W, u')
$$

In which the m' and u' are defined as

$$
\begin{aligned}
m' &= \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u.s).\mathcal{M} \\
u'.s &= \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u.s).s \\
u'.n &= u.n + 1 \\
u'.\mathcal{D} &= \begin{cases}
False & : \quad volw^{\pi,\theta}_{c_{IL}}(stmt) \\
True & : \quad stmt = \mathbf{call}\ e(a,u,v,r) \wedge rmw^{\pi,\theta}_{c_{IL}}(e) \\
& \qquad \vee\ stmt = \mathbf{call}\ e() \wedge mfence^{\pi,\theta}_{c_{IL}}(e) \\
u.\mathcal{D} & : \quad otherwise
\end{cases} \\
u'.\vartheta &= u.\vartheta(R_{u'.n} \mapsto v)
\end{aligned}
$$

where we let

$$a' = \begin{cases} base(top) - \Sigma_{j=npar_{top}}^{k-1} size_\theta(qt2t(V_{top}[j].t)) - o & : \quad \exists k.\ V_{top}[k].v = v \\ \bot & : \quad otherwise \end{cases}$$

in

$$v = \begin{cases} m(ptrv2addr(p)) & : \quad stmt = (e = e') \wedge [\![\&e']\!]^{\pi,\theta}_{c_{IL}} = p \in val_{\mathbf{ptr}} \vee \\ & \quad\ rmw^{\pi,\theta}_{c_{IL}}(stmt, a, u, v, r, \rho, in) \wedge [\![a]\!]^{\pi,\theta}_{c_{IL}} = p \\ m(bin_{30}(a')) & : \quad stmt = (e = e') \wedge [\![\&e']\!]^{\pi,\theta}_{c_{IL}} = \mathbf{lref}((v, o), top, t) \\ \bot & : \quad otherwise \end{cases}$$

$$S^n_{\text{C-IL}}.\delta(u, m, in) = \begin{cases} (\mathcal{M}'|_W, s') & : \quad \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u.s), in) = (\mathcal{M}', s') \\ \bot & : \quad \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u.s), in) = \bot \end{cases}$$

Note that in contrast to the C-IL semantics, we do not update the complete memory for external function calls, because doing so would break the ownership memory access policy. Instead, we only update the relevant memory portions according to the semantics of the particular external function, i.e., of *rmw*, in this case. This approach is sound because we have defined the external transition function input η in such a way, that it implements the semantics specified by $\theta.R_{\mathbf{extern}}$.

- $S^n_{\text{C-IL}}.\mathcal{IP}$ — Again we could choose the interleaving-points to be $\mathcal{IO}$ points to allow for an easier verification of concurrent C-IL code. Later, we want to show a simulation between the concurrent MIPS and the concurrent C-IL model, so we have to choose consistency points as interleaving-points such that in the *Cosmos* model we interleave blocks of code that are executed by different C-IL units and each block starts in a consistency point.

$$S^n_{\text{C-IL}}.\mathcal{IP}(u, m, in) = info_{IL}.cp(f_{top}(\lceil m \rceil, u), loc_{top}(f_{top}(\lceil m \rceil, u)))$$

With this definition of $\mathcal{IO}$ steps and interleaving-points we can make sure by the verification of ownership safety, that shared variables are only accessed at a few designated points in the program, which are chosen by the programmer. This allows on the one hand for the efficient verification of concurrent C-IL programs, on the other hand, it enables us to justify the concurrent C-IL model, using our order reduction theorem.

In order to do so we would first need to determine those above set A_{io} of the underlying MIPS *Cosmos* machine (cf. Sect. 5.4.2). Since all assignments contain only one access to a volatile variable, the compiler can ensure the same for the compiled code. We can determine the address of the memory instruction implementing the access with the help of $info_{IL}.cba$, $info_{IL}.off$, and the code compilation function because there is a consistency point before every assignment that includes a volatile variable. We collect all these instruction addresses in A_{io}. The set A_{cp}, which contains the addresses of all consistency points in the machine code, can easily be defined using $info_{IL}.cba$, $info_{IL}.off$, and $info_{IL}.cp$.

Thus, we have instantiated our *Cosmos* machine with the C-IL semantics obtaining a concurrent C-IL model. However, we still need to discharge $insta_r(S^n_{\text{C-IL}})$ which demands the following property of our *reads* function instantiation. We have to prove that if the memories of two C-IL machines $(\lceil m \rceil, u.s)$ and $(\lceil m' \rceil, u.s)$ agree on reads-set $R = S^n_{\text{C-IL}}.reads(u, m, in)$ of the first machine, then both machines are reading the same addresses in the next step.

$$m|_R = m'|_R \rightarrow S^n_{\text{C-IL}}.reads(u, m', in) = R$$

We need the following lemma to discharge $insta_r(S^n_{\text{C-IL}})$.

Lemma 5.54 (C-IL Reads-Set Agreement) Given are a C-IL program π, environment parameters θ and two C-IL configurations c, c' that agree on their stack, i.e., $c.s = c'.s$, and a C-IL statement *stmt*. If the memories of both machines agree on reads-set of *stmt* wrt. configuration c, the reads-sets of *stmt* agree in both configurations. Let $R = \{a \in \mathbb{B}^{32} \mid a \in R^{\pi,\theta}_c(stmt)\}$ in:

$$c.\mathcal{M}|_R = c'.\mathcal{M}|_R \rightarrow R^{\pi,\theta}_c(stmt) = R^{\pi,\theta}_{c'}(stmt)$$

For the proof of lemma 5.54 can be found in [Bau14].

Proof of $insta_r(S^n_{\text{C-IL}})$: Let $s^\pi(u) = (stmt_{next}(\pi, (\lceil m \rceil, u.s)))$. For reads-set

$$R = R^{\pi,\theta}_{(\lceil m \rceil, u.s)}(s^\pi(u))$$

and partial memories m, m' such that $m|_R = m'|_R$ (hence $\lceil m \rceil|_R = \lceil m' \rceil|_R$) we have by definition and Lemma 5.54:

$$\begin{aligned} S^n_{\text{C-IL}}.reads(u, m, in) &= R^{\pi,\theta}_{(\lceil m \rceil, u.s)}(s^\pi(u)) \overset{\text{L5.54}}{=} R^{\pi,\theta}_{(\lceil m' \rceil, u.s)}(s^\pi(u)) \\ &= S^n_{\text{C-IL}}.reads(u, m', in) \qquad \square \end{aligned}$$

5.7 Simulation Theorem for *Cosmos* machine

In this section, we will almost literally represent Chapter 5 in [Bau14]. The main different is that at the end of this section, we argue about the definition of og^{MIPS}_{cos} with og^{C-IL}_{cos}. We omit all the proofs, some corollaries and lemmas which are only used in proofs. The full version of proof can be looked up in [Bau14].

Based on our order reduction theory we now want to explore how to apply local simulation theorems in a concurrent context. Our goal is to state and prove a global *Cosmos* model simulation theorem which argues that the local simulation theorems still hold on computation units. In particular we want the simulation relation to holding for a unit when it reaches a consistency point. Moreover from the verification of the ownership safety on the higher level, memory safety on the lower level should follow. First we introduce a variation of the *Cosmos* model semantics tailored to the formulation of such a simulation theorem. Then we introduce sequential simulation theorems in a generalized manner. Building on the sequential theorems we then formulate and prove a concurrent simulation theorem between *Cosmos* machines, stating the necessary requirements for the sequential simulations to be composable with each other.

In the concurrent simulation, we will profit from the *Cosmos* model order reduction theorem presented before. For every computation unit of the simulating *Cosmos* machine we set up the interleaving-points to be consistency points wrt. the sequential simulation relation. This enables us to conduct a simulation proof between $\mathcal{IP}$ schedules of *Cosmos* machines, applying the sequential simulation theorem separately on each $\mathcal{IP}$ block. In such a scenario, where the interleaving-points are also consistency points wrt. a given simulation relation we speak of *consistency blocks* instead of $\mathcal{IP}$ blocks.

Now a sequential simulation theorem can be applied on any consistency block on the simulated level in order to obtain the simulated abstract consistency block executed by the same unit. However, there is a technicality to be solved, namely that the given concrete block may not be *complete* in the sense that it does not lead to another consistency point. Then one has to find an extension of that incomplete block so that the resulting complete concrete block is simulating an abstract block. We have to formulate the generalized sequential simulation theorem in a way that allows for this kind of extension. Nevertheless, later we will show for the transfer of verified safety properties that it suffices to consider schedules where each consistency block is complete.

5.7.1 Block Machine Semantics

Since we may assume $\mathcal{IP}$ schedules for safe *Cosmos* machine execution, semantics can be simplified. For introducing simulation theorems on *Cosmos* models, it is convenient to define the semantics where we consecutively execute blocks starting in interleaving-points ($\mathcal{IP}$ blocks). Also for now we do not need to consider ownership. Therefore, it is sufficient to model the transitions on the machine state. We call the machine implementing such semantics the $\mathcal{IP}$ *block machine* or short the *block machine*.

We define the block machine semantics for a *Cosmos* machine S. The block machine executes one $\mathcal{IP}$ block in a single step. To this end, it gets a schedule $\kappa \in (\Theta_S^*)^*$ as a parameter which is a sequence of transition sequences representing the individual blocks to be executed. To distinguish blocks and block schedule, we will always use λ for transition sequences and κ for block sequences. Naturally not all block sequences are valid block machine schedules. Each block in the block machine schedule needs to be an $\mathcal{IP}$ block.

Definition 5.55 ($\mathcal{IP}$ Block) A transition sequence $\lambda \in \Theta_S^*$ is called an $\mathcal{IP}$ block of machine $p \in S.nu$ if it (i) contains only steps by that machine, (ii) is empty or starts in an interleaving-point, and (iii) does not contain any further interleaving-points.

$$\begin{aligned} blk(\lambda, p) \equiv \quad & (i) \quad \forall \alpha \in \lambda.\, \alpha.s = p \\ & (ii) \quad \lambda \neq \varepsilon \rightarrow \lambda_1.ip \\ & (iii) \quad \forall \alpha \in tl(\lambda).\, /\alpha.ip \end{aligned}$$

Thus, we require the $\mathcal{IP}$ blocks to be *minimal* in the sense that they contain at most one interleaving-point. For technical reasons, empty blocks are also considered $\mathcal{IP}$ blocks. We define the appropriate predicate *Bsched* which denotes that a given a block sequence $\kappa \in (\Theta_S^*)^*$ is a block machine schedule.

$$Bsched(\kappa) \equiv \forall \lambda \in \kappa. \exists p \in \mathbb{N}_{nu}.\, blk(\lambda, p)$$

Note that this implies that the *flattening concatenation* $\lfloor \kappa \rfloor = \kappa_1 \cdots \kappa_{|\kappa|}$ of all blocks of κ form an $\mathcal{IP}$ schedule.

Lemma 5.56 The flattening concatenation of all blocks of any block machine schedule $\kappa \in (\Theta_S^*)^*$ is an $\mathcal{IP}$ schedule.

$$Bsched(\kappa) \rightarrow \mathcal{IP}sched(\lfloor \kappa \rfloor)$$

Instead of defining a transition function for the block machine we extend our step sequence notation to block sequences as follows.

Definition 5.57 (Step Notation for Block Sequences) Given two machine states $M, M' \in \mathbb{M}_S$ and a block machine schedule $\kappa \in (\Theta_S^*)^*$, we denote that M' is reached by executing the block machine from state M wrt. schedule κ by the following notation.

$$M \overset{\kappa}{\longmapsto} M' \;=\; M \overset{\lfloor \kappa \rfloor}{\longmapsto} M'$$

Then a pair (M, κ) is a computation of the block machine, if there exists a machine state M' that can be reached via schedule κ from M, i.e., $M \overset{\kappa}{\longmapsto} M'$. Furthermore we need to introduce safety for the block machine wrt. the ownership policy and some safety property P. Similar to *safety* and $safety_{\mathcal{IP}}$ defined earlier, the verification of all block machine computations running out of configuration C wrt. ownership and some *Cosmos* machine safety property P is denoted by the following predicate.

$$\begin{aligned} &safety_B(C, P) \equiv \\ &\quad \forall \kappa \in (\Theta_S^*)^*.\ Bsched(\kappa) \wedge comp(C.M, \lfloor \kappa \rfloor) \;\rightarrow\; \exists o \in \Omega_S^*.\ safe_P(C, \langle \lfloor \kappa \rfloor, o \rangle) \end{aligned}$$

In order to justify the verification of systems using block machine schedules instead of $\mathcal{IP}$ schedules, we need to introduce another reduction theorem. However, since the two concepts are so closely related this is a rather easy task.

Theorem 5.58 (Block Machine Reduction) *Let C be a configuration of* Cosmos *machine S and P be a* Cosmos *machine safety property. Then if all block machine computations running out of C are ownership-safe and preserve P, the same holds for all $\mathcal{IP}$ schedules starting in C.*

$$safety_B(C, P) \rightarrow safety_{\mathcal{IP}}(C, P)$$

The complete proof can be found in [Bau14].

5.7.2 Generalized Sequential Simulation Theorems

The computer systems can be described in several layers of abstractions, e.g. on the ISA level and the level of C-IL or even higher levels of abstraction [GHLP05b]. Between different levels, there are sequential simulation theorems. Such simulation theorems are proven for sequential execution traces where no environment steps are interleaved. However, it is desirable to have the simulation relation hold also in the context of the concurrent system. Thus we need to be able to apply sequential simulation theorems in a system wide simulation proof between two *Cosmos* model instantiations $S_d, S_e \in \mathbb{S}$ where the interleaving-points are instantiated to

be the consistency points wrt. the corresponding simulation relation. Recall that we speak of *consistency blocks* instead of $\mathcal{IP}$ blocks then.

In the sequel we develop a generalized theory of sequential simulation theorems. We consider the simulation between computations $(d, \sigma) \in \mathbb{M}_{S_d} \times \Theta^*_{S_d}$ and $(e, \tau) \in \mathbb{M}_{S_e} \times \Theta^*_{S_d}$ considering only the machine state of these *Cosmos* machines. We also speak of S_d as the *concrete* and of S_e as the *abstract* simulation layer, where computations of S_d are simulated by S_e. For simplicity we assume that the two systems have compatible memory types. Also both systems have the same number of computation units. Using the shorthands x_d and x_e for components $S_d.x$ and $S_e.x$ we demand:

$$\mathcal{A}_d \supseteq \mathcal{A}_e \qquad \mathcal{V}_d = \mathcal{V}_e \qquad nu_d = nu_e = nu$$

Observe that the memory address range of S_d might be larger than that of S_e. This means that the latter may abstract from certain memory regions in the former. For example, this is useful when we abstract a stack of local memories from a stack memory region when we consider compilation of C-IL programs as we have seen before. The stack region is then excluded from the shared memory. As we aim for a generalized theory about concurrent simulation theorems, we first define a framework for specifying sequential simulation theorems in a uniform way.

Definition 5.59 (Sequential Simulation Framework) We introduce a type $\mathcal{R}bb$ for simulation frameworks R_{S_d,S_e} which contain all the information needed to state a generalized simulation theorem relating sequential computations of units of *Cosmos* machines S_d and S_e.

$$R_{S_d,S_e} = (\mathcal{P}, sim, \mathcal{CP}a, \mathcal{CP}c, wfa, sc, wfc, suit, wb) \in \mathcal{R}bb$$

In particular we have the following components where $\mathbb{L}_x \equiv (\mathcal{U}_x \times (\mathcal{A}_x \to \mathcal{V}_x))$ with $x \in \{d, e\}$ is a shorthand for the type of a *local configuration* of *Cosmos* machine S_x containing the state of one computation unit and shared memory:

- $\mathcal{P}$ — the set of simulation parameters, which is $\{\bot\}$ if there are none,
- $sim : \mathbb{L}_d \times \mathcal{P} \times \mathbb{L}_e \to \mathbb{B}$ — a simulation relation between local configurations of computation units of S_d and S_e, depending on a simulation parameter from $\mathcal{P}$,
- $\mathcal{CP}a : \mathcal{U}_e \times \mathcal{P} \to \mathbb{B}$ — a predicate to identify consistency points of the abstract *Cosmos* machine S_e,
- $\mathcal{CP}c : \mathcal{U}_d \times \mathcal{P} \to \mathbb{B}$ — a predicate to identify consistency points of the concrete *Cosmos* machine S_d,
- $wfa : \mathbb{L}_e \to \mathbb{B}$ — a well-formedness condition for a local configuration of the abstract *Cosmos* machine S_e,
- $sc : \mathbb{M}_{S_e} \times \Theta_{S_e} \times \mathcal{P} \to \mathbb{B}$ — software conditions that enable a simulation of sequential computations of *Cosmos* machine S_e, here defined for a given step,
- $wfc : \mathbb{L}_d \to \mathbb{B}$ — well-formedness condition for a local configuration of the concrete *Cosmos* machine S_d, required for the simulation of sequential computations of S_e,

- $suit : \Theta_{S_d} \to \mathbb{B}$ — a predicate to determine whether a given step by the concrete *Cosmos* machine is suitable for simulation.
- $wb : \mathbb{M}_{S_d} \times \Theta_{S_d} \times \mathcal{P} \to \mathbb{B}$ — a predicate that restricts the simulating computations of S_d. We say that a simulating step in a computation of S_d is *well-behaved* iff it fulfills this restriction.

We give some intuitions on how to instantiate the components in the simulation framework. Formal instantiations will be introduced in the subsequent section. Here we interpret S_e as the C-IL *Cosmos* machine and S_d as the MIPS *Cosmos* machine. The simulation relation can be instantiated as the C-IL compiler consistency relation. The consistency points in C-IL level are program locations (i) at function entry, (ii) directly before and after function calls (including external functions), (iii) at volatile variable accesses, (iv) directly before return statements. In MIPS level, the consistency points are defined as program locations correspond to C-IL consistency points. The well-formedness condition for C-IL local configuration consists 2 portion: the C-IL program well-formedness (Definition 5.34) and the C-IL configuration well-formedness (Definition 5.38). The C-IL software condition can be interpreted as the execution next step may (i) not produce a run-time error, (ii) not result in a stack overflow, (iii) not explicitly access the stack or code region and that (iv) the code region fits into memory and is disjoint from the stack region. The suitability and good behavior (i.e., being well-behaved) were somewhat indiscriminate, and we want to highlight the difference between the two concepts. While the suitability is a necessary condition on the schedule of the concrete *Cosmos* machine for the simulation to work, good behavior is a property that is guaranteed for simulating computations by the simulation theorem. These properties become important in a stack of simulation properties where they should imply the software conditions on the abstract layer of the underlying simulation theorem. In our instantiation, *wfc*, *suit* and *wb* ensure that no interrupts are triggered and that instructions are fetched from the code region.

The consistency point predicates $\mathcal{CP}a$ and $\mathcal{CP}c$ are used later to define the interleaving-points in the concurrent *Cosmos* machine computations.

As mentioned before we need to be able to apply the sequential simulation theorem on incomplete consistency blocks. Thus, we consider a given consistency block $\omega \in \Theta^*_{S_d}$ as the basis for the simulating concrete computation. We have to extend ω into a complete non-empty consistency block σ which is simulating some abstract consistency block. Formally the extension of some transition sequence is denoted by the relation $\omega \rhd^{blk}_p \sigma$ which is saying that σ extends ω without adding consistency points to the block. Alternatively we can say that ω is a prefix of the consistency block σ

$$\omega \rhd^{blk}_p \sigma = \exists \tau.\ \sigma = \omega\tau \neq \varepsilon \wedge blk(\sigma, p) \wedge blk(\omega, p)$$

In order to be able to integrate the sequential simulation theorems into the concurrent system later on, there is an additional proof obligation in the sequential simulation below. It is there to justify the $\mathcal{IOIP}$ condition of the underlying order reduction theorem which demands that there is at most one $\mathcal{IO}$ step between two subsequent interleaving-points of the same computation unit. This property has to be preserved by the concrete implementation of the abstract specification level. Moreover, there should be a one-to-one mapping of $\mathcal{IO}$ steps on the abstract level to

the $\mathcal{IO}$ steps on the concrete level. That means that in corresponding blocks *Cosmos* machine S_d may only perform an $\mathcal{IO}$ step when S_e does and vice versa. If this would not be the case we could not couple the ownership state of S_d and S_e later, because at $\mathcal{IO}$ steps we allow for ownership transfer. Transferring ownership on one level but not on the other then may lead to inconsistent ownership configurations. We denote the requirements on $\mathcal{IO}$ points in consistency blocks by the overloaded predicate *one*$\mathcal{IO}$. For a single transition sequence σ it demands that σ contains only one $\mathcal{IO}$ step. For a pair (σ, τ) it demands that they contain the same number of $\mathcal{IO}$ steps but at most one.

$$\begin{aligned} \mathit{one}\mathcal{IO}(\sigma) &\equiv \forall i, j \in \mathbb{N}_{|\sigma|}.\ \sigma_i.\mathit{io} \wedge \sigma_j.\mathit{io} \rightarrow i = j \\ \mathit{one}\mathcal{IO}(\sigma, \tau) &\equiv (\tau|_{\mathit{io}} = \varepsilon \leftrightarrow \sigma|_{\mathit{io}} = \varepsilon) \wedge \mathit{one}\mathcal{IO}(\sigma) \wedge \mathit{one}\mathcal{IO}(\tau) \end{aligned}$$

Moreover we introduce the following shorthands for $d \in \mathbb{M}_{S_d}$, $e \in \mathbb{M}_{S_e}$, $p \in \mathbb{N}_{nu}$, $\mathit{par} \in R_{S_d,S_e}.\mathcal{P}$, $\omega \in \Theta_{S_d}$, and $\tau \in \Theta_{S_e}$.

$$\begin{aligned} \mathcal{P} &\equiv R_{S_d,S_e}.\mathcal{P} \\ \mathit{sim}_p(d, \mathit{par}, e) &\equiv R_{S_d,S_e}.\mathit{sim}((d.u(p), d.m), \mathit{par}, (e.u(p), e.m)) \\ \mathcal{CP}_p(e, \mathit{par}) &\equiv R_{S_d,S_e}.\mathcal{CP}a(e.u(p), \mathit{par}) \\ \mathcal{CP}_p(d, \mathit{par}) &\equiv R_{S_d,S_e}.\mathcal{CP}c(d.u(p), \mathit{par}) \\ \mathit{wf}_p(e) &\equiv R_{S_d,S_e}.\mathit{wfa}(e.u(p), e.m) \\ \mathit{sc}(e, \tau, \mathit{par}) &\equiv \forall \theta, \alpha, \theta', e'.\ \tau = \theta\alpha\theta' \wedge e \overset{\theta}{\longmapsto} e' \rightarrow R_{S_d,S_e}.\mathit{sc}(e', \alpha, \mathit{par}) \\ \mathit{wf}_p(d) &\equiv R_{S_d,S_e}.\mathit{wfc}(d.u(p), d.m) \\ \mathit{suit}(\omega) &\equiv \forall \alpha \in \omega.\ R_{S_d,S_e}.\mathit{suit}(\alpha) \\ \mathit{wb}(d, \omega, \mathit{par}) &\equiv \forall \theta, \alpha, \theta, d'.\ \omega = \theta\alpha\theta' \wedge d \overset{\theta}{\longmapsto} d' \rightarrow R_{S_d,S_e}.\mathit{wb}(d', \alpha, \mathit{par}) \end{aligned}$$

Note that we overload $\mathcal{CP}_p$ and use both for machine states of type $\mathbb{M}_{S_d}$ and $\mathbb{M}_{S_e}$. In the same way, we have overloaded wf_p. In what follows we will always use letter d to represent concrete machine states and letter e for abstract ones.

The generalized sequential simulation theorem is stated such that it allows for completing incomplete consistency blocks on the concrete abstraction layer. Given a concrete machine computation (d, ω), where the simulation relation sim_p holds between initial machine state d and an abstract state e for some computation unit p and ω is an incomplete consistency block executed by p. We need to be able to extend ω into a transition sequence σ that leads into a consistency point, obtaining a complete consistency block for which there is a simulated computation (e, τ) on the abstract level (cf. Fig. 5.4).

The ability to extend incomplete blocks into complete ones is important in the proof of the concurrent simulation theorem where we need to find a simulated abstract computation for a concurrent concrete block machine computation, where most of the consistency blocks are probably incomplete. In this situation, we can use the generalized sequential simulation theorem for completing the concrete blocks and finding the simulated abstract consistency blocks. Formally the theorem reads as follows.

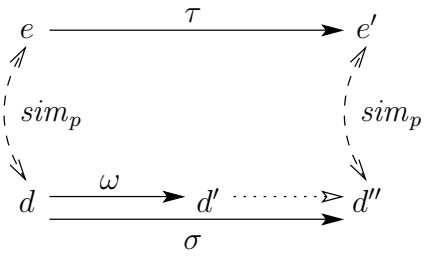


Figure 5.4: Illustration of the generalized sequential simulation theorem. Here σ extends consistency block ω of unit p, i.e., $\omega \rhd_p^{blk} \sigma$, such that the computation reaches another consistency point and simulates abstract computation (e, τ).

Theorem 5.60 (Generalized Sequential Simulation Theorem) *Given are two starting machine states $d \in \mathbb{K}_{S_d}$, $e \in \mathbb{K}_{S_e}$, a simulation parameter $par \in R_{S_d,S_e}.\mathcal{P}$ and a transition sequence $\omega \in \Theta^*_{S_d}$. If for any computation unit $p \in \mathbb{N}_{nu}$ (i) d and e are well-formed and (ii) consistent wrt. par, (iii) ω is a possibly incomplete consistency block of unit p that is suitable for simulation and executable from d, and (iv) all complete consistency blocks of unit p which are starting in e are obeying the software conditions for S_e and lead into well-formed configurations,*

$$\begin{array}{lll} \forall d, e, par, \omega, p. & (i) & wf_p(d) \wedge wf_p(e) \\ & (ii) & sim_p(d, par, e) \wedge \mathcal{CP}_p(d, par) \wedge \mathcal{CP}_p(e, par) \\ & (iii) & blk(\omega, p) \wedge suit(\omega) \wedge \exists d'.\ d \overset{\omega}{\longmapsto} d' \\ & (iv) & \forall \pi, e'.\ e \overset{\pi}{\longmapsto} e' \wedge blk(\pi, p) \wedge \mathcal{CP}_p(e', par) \rightarrow sc(e, \pi, par) \wedge wf_p(e') \end{array}$$

then we can find sequences $\sigma \in \Theta^*_{S_d}$, $\tau \in \Theta^*_{S_e}$ and configurations $d'' \in \mathbb{K}_{S_d}$, $e'' \in \mathbb{K}_{S_e}$ such that (i) σ is a suitable schedule and a consistency block of unit p extending the given block ω, τ is a consistency block of unit p, and σ and τ contain the same amount of $\mathcal{IO}$ steps but at most one. Moreover (ii) (d, σ) is a well-behaved computation with leading into well-formed state d'' and (iii) executing τ from e leads into well-formed configuration e''. Finally (iv) d'' and e'' are consistency points of unit p and consistent wrt. simulation parameter par:

$$\begin{array}{lll} \rightarrow \quad \exists \sigma, \tau, d'', e''. & (i) & \omega \rhd_p^{blk} \sigma \wedge suit(\sigma) \wedge blk(\tau, p) \wedge one\mathcal{IO}(\sigma, \tau) \\ & (ii) & d \overset{\sigma}{\longmapsto} d'' \wedge wb(d, \sigma, par) \wedge wf_p(d'') \\ & (iii) & e \overset{\tau}{\longmapsto} e'' \wedge wf_p(e'') \\ & (iv) & sim_p(d'', par, e'') \wedge \mathcal{CP}_p(d'', par) \wedge \mathcal{CP}_p(e'', par) \end{array}$$

Note that for the simulated computation (e, τ) we only demand progress (i.e., $\tau \neq \varepsilon$) in case σ contains $\mathcal{IO}$ steps. Then $\tau \neq \varepsilon$ follows from $one\mathcal{IO}(\sigma, \tau)$. In contrast, by $\omega \rhd_p^{blk} \sigma$ we only consider such computations (d, σ) that are progressing in every simulation step, i.e., $\sigma \neq \varepsilon$. This setting rules out trivial simulations with empty transition sequences σ and τ in case $\omega = \varepsilon$.

For proving the theorem for C-IL and MIPS *Cosmos* machine one needs to know the code generation function of a given optimizing C-IL compiler and prove the correctness of the gen-

erated code for statements between consistency points. Then using code consistency one argues that only the correct generated code is executed, eventually leading to another consistency point.

Additionally, note that the sequential simulation theorem does not restrict the ownership state in any way. All predicates depend only on the machine state of a *Cosmos* machine. However for proving our concurrent simulation theorem, we will need an assumption on the ownership-safety of the simulated computation.

5.7.3 Instantiation of Sequential Simulation Framework

In this section, we will instantiate the sequential simulation framework as $R_{S^n_{\text{MIPS}},S^n_{\text{C-IL}}}$. First we will define the compiler consistency points and the compiler consistency relation. Note that we will not provide a compiler for C-IL but just state a compiler consistency relation that couples a MIPS implementation with the C-IL language level and use the consistency relation to establish a simulation theorem between a C-IL *Cosmos* machine and a MIPS *Cosmos* machine, thus justifying the notion of structured parallel C, which is assumed by C code verification tools. Then we will give the definition of C-IL software condition. Moreover, we will state the well-formedness of MIPS configuration, the suitability and the well-behaving of MIPS computations.

Compiler Consistency Points and Compiler Consistency Relation

We aim for a theory that is also applicable for optimizing compilers. In non-optimizing compilers, the compilation is a function mapping one C statement to a number of implementing assembly statements. An optimizing compiler applies optimizing transformations to the compiled code of a sequence of C-IL statements, typically with the aim of reducing redundancy and the overall code execution time. Typical optimizations are, e.g., saving intermediate results of expression evaluation to reuse them for the implementation of subsequent statements, or avoiding to store frequently used data in main memory, because accesses to registers are much faster. This means however that variables are not consistent with their memory representations for most of the time. There are only a few points in a C program where the consistency relation holds with the optimized implementation, and we call these points *compiler consistency points* or short *consistency points*. For C-IL, we assume that certain locations in a function are always consistency points. These consistency points are, in particular:

- at function entry
- directly before and after function calls (including external functions)
- between two consecutive volatile variable accesses
- directly before return statements

Definition 5.61 (Required C-IL Compiler Consistency Points) Given a compiler information $info_{IL}$ for a C-IL program π and environment parameter θ, the following predicate holds, iff there are compiler consistency points (i) at the entry of every function, (ii) before and after function calls, (iii) between any two consecutive volatile variable accesses, and (iv) before return

statements. Let $s_{f,i} = \pi.\mathcal{F}(f).P[i]$, $call(s) = \exists e, e', E.\ s \in \{\textbf{call}\ e(E), (e'' = \textbf{call}\ e(E))\}$, and $ret(s) = \exists e.\ s \in \{\textbf{return}, \textbf{return}\ e\}$ in:

$$\begin{aligned} reqCP(\pi, \theta, info_{IL}) \equiv\ & \forall f \in \mathbb{F}_{name}, i \in \mathbb{N}.\ \pi.\mathcal{F}(f).P \neq \textbf{extern} \wedge i \leq |\pi.\mathcal{F}(f).P| \rightarrow \\ & (i)\quad info_{IL}.cp(f, 0) \\ & (ii)\quad call(s_{f,i}) \rightarrow info_{IL}.cp(f, i) \wedge info_{IL}.cp(f, i+1) \\ & (iii)\quad vol_f^{\pi,\theta}(s_{f,i}) \wedge (\exists j < i.\ vol^{\pi,\theta}(s_{f,j})) \rightarrow \exists k \in (j : i).\ info_{IL}.cp(f, k) \\ & (iv)\quad ret(s_{f,i}) \rightarrow info_{IL}.cp(f, i) \end{aligned}$$

The C-IL compilation function now is mapping a block of C-IL statements between consistency points to blocks of assembly instructions. However as the optimizations depend on the program context we rather model the code generation as a function depending on the C-IL program, the function, and the location of the consistency point starting the block which should be compiled.

$$cpl : prog_{\text{C-IL}} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup (\mathbb{B}^{32})^*$$

This means that a C-IL program is compiled by applying the compilation function *cpl* subsequently on every consistency block of the program. The function *cpl* compiles each volatile access instruction into a sequence of instructions which contains exactly one shared memory access. The compiled code for the program contains the compiled code for every function positioned in a way so that jumps between functions are linked correctly.

Now we will define the compiler consistency relation that links a C-IL computation to its implementation on the MIPS ISA level. We want to relate a C-IL configuration $c_{IL} = (s, \mathcal{M})$ to an ISA state $c_{\text{MIPS}} = (p, m)$ that implements the program π using the environment parameters θ and compiler information $info_{IL}$. Note that for all $X \in \{pc, gpr, spr\}$. $c_{\text{MIPS}}.p.X$ we write $c_{\text{MIPS}}.X$ for short. Formally we thus define a simulation relation

$$consis_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}})$$

stating the consistency between these entities. The relation is supposed to hold only in compiler consistency points, which are identified by a function name and a location according to the $info_{IL}.cp$ predicate. We define the following predicate which holds iff c_{IL} is currently in a consistency point.

$$cp(c_{IL}, info_{IL}) \equiv info_{IL}.cp(f_{top}(c_{IL}), loc_{top}(c_{IL}))$$

The compiler consistency relation is split in two sub-relations covering control and data consistency. The first part talks about control-flow and is thus concerned with the program counter and the return address. Let the following function compute the start address of the compiled code for the C-IL statements starting from a consistency point *loc* in function f.

$$adr(info_{IL}, f, loc) \equiv info_{IL}.cba +_{30} info_{IL}.off(f, loc)_{30}$$

Again we define shorthands for return address, previous base pointer, and also the return destination, depending on some ISA configuration c_{MIPS}.

$$\begin{aligned} \forall i \in [0 : |c_{IL}.s| - 1].\ ra(i) &\equiv c_{\text{MIPS}}.m((base(i) + 1)_{30}) \\ \forall i \in [0 : |c_{IL}.s| - 1).\ rds(i) &\equiv c_{\text{MIPS}}.m((base(i) + 2 + size_{par}(i))_{30}) \\ \forall i \in [0 : |c_{IL}.s| - 1].\ pbp(i) &\equiv c_{\text{MIPS}}.m(base(i)_{30}) \end{aligned}$$

Note that $rds(i)$ is the return destination of function belongs to frame $i + 1$.

Definition 5.62 (C-IL Control Consistency) We define control consistency sub-relation for C-IL $consis^{control}_{\text{C-IL}}$, which states that (i) the program counter of the MIPS machine must point to the start of the compiled code for the current statement in the C-IL machine which is at a compiler consistency point. In addition (ii) the return address of any stack frame is pointing to the beginning of the function call epilogue for the function call statement in the previous frame (with lower index).

$$
\begin{aligned}
&consis^{control}_{\text{C-IL}}(c_{IL}, info_{IL}, c_{\text{MIPS}}) \equiv \\
&\quad (i) \quad cp(c_{IL}, info_{IL}) \rightarrow c_{\text{MIPS}}.pc = adr(info_{IL}, f_{top}, loc_{top}) \\
&\quad (ii) \quad \forall i \in [1, |c_{IL}.s| - 1].\ ra(i) = info_{IL}.cba +_{30} info_{IL}.fceo(f_{i-1}, loc_{i-1} - 1)_{30}
\end{aligned}
$$

According to the C-IL semantics, the current location of a caller frame already points to the statement after the function call (which is a consistency point). To obtain the location of the function call we, therefore, have to subtract one from that location. When control returns to the caller frame, on the ISA level first the function call epilogue is executed before the consistency point is reached.

Data consistency is split into several parts covering registers, the global memory, local variables, the code region as well as the stack structure. The register consistency relation covers only the stack and base pointers.

Definition 5.63 (C-IL Register Consistency) The C-IL register consistency relation demands, that (i) the base pointer points to the base address of the top frame, while (ii) the stack pointer points to the top-most element of the temporary values (growing downwards) in the top frame.

$$
\begin{aligned}
consis^{regs}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \equiv \quad &(i) \quad c_{\text{MIPS}}.gpr(bp) = bin_{30}(base(top)) \circ 00 \\
&(ii) \quad c_{\text{MIPS}}.gpr(sp) = bin_{30}(base(top) - dist(top)) \circ 00
\end{aligned}
$$

In the code consistency relation we also need to couple π with the compiled code.

Definition 5.64 (C-IL Code Consistency) For C-IL code consistency we require that (i) the compiler consistency points were selected by the compiler according to our requirements, (ii)the compiled code in the compiler information is actually corresponding to the C-IL program, and that (iii) the compiled code is converted to binary format and resides in a contiguous region in the memory of the MIPS machine starting at the code base address.

$$
\begin{aligned}
&consis^{code}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \equiv \\
&\quad (i) \quad reqCP(\pi, \theta, info_{IL}) \\
&\quad (ii) \quad \forall f \in \text{dom}(\pi.\mathcal{F}), l.\ info_{IL}.cp(f, l) \rightarrow \\
&\qquad\qquad \forall i \in [0 : |cpl(\pi, f, l)| - 1].\ info_{IL}.code[info_{IL}.off(p, l) + i] = cpl(\pi, f, l)[i] \\
&\quad (iii) \quad \forall j \in [0 : |info_{IL}.code| - 1]. \\
&\qquad\qquad info_{IL}.code[j] = c_{\text{MIPS}}.m(info_{IL}.cba +_{30} bin_{30}(j))
\end{aligned}
$$

Now we demand memory consistency for all addresses but the code region and the stack region, because these addresses may not be accessed directly in C-IL programs.

$$consis^{mem}_{\text{C-IL}}(c_{IL}, info_{IL}, c_{\text{MIPS}}) \equiv \forall ad \in \mathbb{B}^{30}.\ \langle ad \rangle \notin CR \cup StR \rightarrow c_{\text{MIPS}}.m(ad) = c_{IL}.\mathcal{M}(ad)$$

Note that this definition includes the consistency for global variables since they are always allocated in the global memory $c.\mathcal{M}$. The allocated address for a given global variable is determined by a global variable allocation function $\theta.alloc_{gv} : \mathbb{V} \rightharpoonup \mathbb{B}^{30}$. We did not introduce it in the C-IL semantics because it is only relevant for the definition of expression evaluation, which we excluded from our presentation.

In contrast to global variables, local variables are allocated on the stack using offsets from $info_{IL}.lvo$. Moreover top frame local variables and parameters may be kept in registers according to the compiler information $info_{IL}.lvr$. In [Sha12] the local variable consistency relation did not talk about the frames below the top frame (*caller frames*), however, such a compiler consistency relation is not inductive in the sense that it cannot be used in an inductive compiler correctness proof. When treating **return** instructions one cannot establish the local variable consistency for the new top frame without knowing where the values of the local variables of that frame were stored before returning.

In fact for the local variables and parameters of caller stack frames there are three possibilities depending on where they are expected to be stored upon return from the called function. If they are supposed to be allocated on the stack upon function return, then we demand that they already reside in their dedicated stack location during the execution of the callee. If they are to be allocated in caller-save registers, we require the caller to store them in its caller-save area during the function call. Similarly, we demand the callee to store them in the callee-save area if we expect their value to reside in callee-save registers after returning from the function call. Below we give a correct definition of the C-IL local variable consistency relation.

Definition 5.65 (C-IL Local Variable Consistency) Compiler consistency relation $consis^{lv}_{\text{C-IL}}$ couples the values of local variables (including parameters) of stack frames with the MIPS ISA implementation. Let

$$\begin{aligned}
(v_{i,j}, t_{i,j}) &\equiv V_i[j] \\
r_{i,j} &\equiv info_{IL}.lvr(v_{i,j}, f_i, loc_i) \\
lva_{i,j} &\equiv bin_{30}(base(i) - info_{IL}.lvo(v_{i,j}, f_i, loc_i)) \\
para_{i,j} &\equiv bin_{30}\left(base(i) + 2 + \sum\nolimits_{k=0}^{j-2} size_\theta(qt2t(t_{i,k}))\right) \\
crsbase_i &\equiv base(i) - (|V_i| - npar_i) - 8 - info_{IL}.size_{tmp}(f_i, loc_i) - 1 \\
crsa_{i,j} &\equiv bin_{30}\left(csrbase_i - info_{IL}.crso(f_i, loc_i, r_{i,j})\right) \\
csa_{i,j} &\equiv bin_{30}\left(base(i) - (|V_{i+1}| - npar_{i+1}) - \epsilon\{k \in \mathbb{N}_{32} \mid r_{i,j} = sv_k\}\right)
\end{aligned}$$

where $v_{i,j}$ is the j-th local variable in frame i with type $t_{i,j}$, that is allocated on the stack if $r_{i,j}$ is undefined. Then it is stored at local variable address $lva_{i,j}$ or parameter address $para_{i,j}$. In the other case that $r_{i,j}$ is defined, variables of the top frame are stored in the corresponding registers. Variables of other stack frames that are allocated in registers are stored either in the caller-save

area starting from (upper) base address $crsbase_i$ at address $crsa_{i,j}$, or in the callee-save area of the callee frame at address $csa_{i,j}$. Formally, with $CS = \{sv_1, \ldots, sv_8\}$:

$$consis^{lv}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \equiv \forall i \in \mathbb{N}_{top}, j \in \mathbb{N}_{|V_i|}.$$
$$\rightarrow \mathcal{M}_{\mathcal{E}i}(v_{i,j}) = \begin{cases} c_{\text{MIPS}}.gpr(r_{i,j}) & : \quad r_{i,j} \neq \bot \wedge i = top \\ c_{\text{MIPS}}.m(csa_{i,j}) & : \quad r_{i,j} \in CS \wedge i < top \\ c_{\text{MIPS}}.m(crsa_{i,j}) & : \quad r_{i,j} \in \mathbb{B}^5 \setminus CS \wedge i < top \\ c_{\text{MIPS}}.m_{size_\theta(qt2t(t_{i,j}))}(lva_{i,j}) & : \quad r_{i,j} = \bot \wedge j > npar_i \\ c_{\text{MIPS}}.m_{size_\theta(qt2t(t_{i,j}))}(para_{i,j}) & : \quad \text{otherwise} \end{cases}$$

Note that we restricted the optimizing compiler by demanding that it always saves all eight callee-save registers in the callee-save area. A lazier implementation might just keep them in the registers if they are not modified. In the case of further function calls their values would be preserved by the calling convention. Such a setting would lead to a much more complex situation where local variables of caller frames on the bottom of the stack may be stored in much higher stack frames or even the registers of the top frame. In order to keep the definitions simple, we did not allow such optimizations here. The consistency relation for the remaining stack components is stated below.

Definition 5.66 (C-IL Stack Consistency) The C-IL stack component is implemented correctly in memory, if in every stack frame except the lowest one (i) the previous base pointer field contains the address of the base of the previous frame (with higher index), and if (ii) the return destination points to the correct address, according to the *rds* component of the C-IL function frame i, in case it is defined. Let $alv = bin_{32}(base(j) - info_{IL}.lvo(v, f_j, loc_j) + o)$ in:

$$consis^{stack}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \equiv \forall i \in [0 : |c_{IL}.s| - 1).$$
$$(i) \quad pbp(i+1) = base(i)$$
$$(ii) \quad rds_i \neq \bot \rightarrow rds(i) = \begin{cases} a & : \quad rds_i = \mathbf{val}(a, t) \in val_{\mathbf{ptr}} \\ alv & : \quad rds_i = \mathbf{lref}((v, o), j, t) \in val_{\mathbf{lref}} \end{cases}$$

Now we can collect all sub-relations and define the overall compiler consistency relation between C-IL and MIPS configurations.

Definition 5.67 (C-IL Compiler Consistency Relation) The C-IL consistency relation comprises the consistency between MIPS and C-IL machine wrt. (i) program counter and return addresses, (ii) the code region, (iii) stack and base pointer registers, (iv) the global memory region, (v) the local variables and parameters, as well as (vi) return destinations and the chain of previous base pointers.

$$consis_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \equiv$$
$$(i) \quad consis^{control}_{IL}(c_{IL}, info_{IL}, c_{\text{MIPS}}) \qquad (iv) \quad consis^{mem}_{IL}(c_{IL}, info_{IL}, c_{\text{MIPS}})$$
$$(ii) \quad consis^{regs}_{IL}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \qquad (v) \quad consis^{lv}_{IL}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}})$$
$$(iii) \quad consis^{code}_{IL}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \qquad (vi) \quad consis^{stack}_{IL}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}})$$

Software Condition, Well-formedness, and Well-behaving

Definition 5.68 (C-IL Software Conditions) A C-IL program can be implemented if all reachable configurations obey the software conditions denoted by the following predicate. Given a C-IL configuration c_{IL}, programme π, environment parameters θ, and assembler information $info_{IL}$, then the next step according to input $in \in \Sigma_{\text{C-IL}}$ may (i) not produce a run-time error, (ii) not result in a stack overflow, and (iii) not explicitly access the stack or code region. Additionally, in (ii) we demand the minimal stack pointer value to be positive, and that (iv) the code region fits into memory and is disjoint from the stack region.

$$\begin{aligned} sc_{\text{C-IL}}(c_{IL}, in, \pi, \theta, info_{IL}) \equiv \quad & (i) \quad \delta^{\pi,\theta}_{\text{C-IL}}(c_{IL}, in) \neq \bot \\ & (ii) \quad /stackovf(c_{IL}, \pi, \theta, info_{IL}) \wedge msp_{IL} \geq 0 \\ & (iii) \quad A^{\pi,\theta}_{c_{IL}}(stmt_{next}(\pi, c_{IL})) \cap (CR \cup StR) = \emptyset \\ & (iv) \quad CR \subseteq [0 : \langle 2^{32} \rangle) \wedge CR \cap StR = \emptyset \end{aligned}$$

Note that these restrictions imply that accessed global variables are not allocated in the stack or code region by the compiler. Also, by (i) the software conditions exclude common programming errors like out-of-bounds array accesses or dereferencing dangling pointers to local variables.

Another software condition one could think of is to limit the number of global variables so that all fit in global memory. However, this is already covered here because of two facts. First, in C-IL semantics there is an explicit allocation function $\theta.alloc_{gv}$ for global variables which determine their addresses in global memory. Secondly, the absence of run-time errors ensures that every global variable that is ever accessed is allocated. Thus, we cannot have too many global variables in a program that is fulfilling the software conditions stated above.

Concerning the well-formedness of MIPS configurations and well-behaving of MIPS computations, they ensure that no external or internal interrupts are triggered and that instructions are fetched from the code region. We let $I = c_{\text{MIPS}}.m(c_{\text{MIPS}}.pc)$ then

$$\begin{aligned} suit^{\text{C-IL}}_{\text{MIPS}}(eev) &\equiv /eev[0] \\ wb^{\text{C-IL}}_{\text{MIPS}}(c_{\text{MIPS}}, eev) &\equiv /jisr(c_{\text{MIPS}}.p, I, eev, 0, 0) \wedge \langle c_{\text{MIPS}}.pc \rangle \subseteq CR \\ wf^{\text{C-IL}}_{\text{MIPS}}(c_{\text{MIPS}}) &\equiv c_{\text{MIPS}}.spr(sr)[\text{dev}] = 0 \end{aligned}$$

Instantiation

We define the sequential simulation framework $R_{S^n_{\text{MIPS}}, S^n_{\text{C-IL}}}$. Let

$$\begin{aligned} c_{\text{MIPS}} &= (p, m) \qquad u_{\text{MIPS}} = (c_{\text{MIPS}}, \mathcal{D}_{\text{MIPS}}, \vartheta_{\text{MIPS}}) \in \mathbb{L}_{S^n_{\text{MIPS}}} \\ c_{IL} &= (s, \mathcal{M}) \qquad u_{\text{C-IL}} = (c_{IL}, \mathcal{D}_{IL}, \vartheta_{IL}) \in \mathbb{L}_{S^n_{\text{C-IL}}} \end{aligned}$$

then

$$R_{S^n_{\text{MIPS}},S^n_{\text{C-IL}}}.\begin{cases}\mathcal{P} & = & InfoT_{\text{C-IL}}\\ sim(u_{\text{MIPS}}, info_{IL}, u_{IL}) & = & consis_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, c_{\text{MIPS}}) \wedge \\ & & \mathcal{D}_{\text{MIPS}} = \mathcal{D}_{IL}\\ \mathcal{CP}a(s, info_{IL}) & = & info_{IL}.cp(s[|s|-1].f, s[|s|-1].loc)\\ \mathcal{CP}c(p, info_{IL}) & = & p.pc \in A^{\text{C-IL}}_{cp}\\ wfa(c_{IL}) & = & wf_{\text{C-IL}}(c_{IL}, \pi, \theta)\\ sc(M_{IL}, t, info_{IL}) & = & sc_{\text{C-IL}}((M_{IL}.m, M_{IL}.u(t.s)), t.in, \pi, \theta, info_{\mu})\\ wfc(h) & = & wf^{\text{C-IL}}_{\text{MIPS}}(h)\\ suit(\alpha) & = & suit^{\text{C-IL}}_{\text{MIPS}}(\alpha.in)\\ wb(M_{\text{MIPS}}, t, info_{IL}) & = & wb^{\text{C-IL}}_{\text{MIPS}}((M_{\text{MIPS}}.m, M_{\text{MIPS}}.u(t.s)), t.in)\end{cases}$$

Here $A^{\text{C-IL}}_{cp}$ represents the instruction addresses of all consistency-points on the MIPS level and was defined as follows.

$$A^{\text{C-IL}}_{cp} \equiv \{adr(info_{IL}, f, loc) \mid f \in \text{dom}(\mathcal{F}^{\theta}_{\pi}) \wedge loc \leq |\mathcal{F}^{\theta}_{\pi}.P| - 1 \wedge info_{IL}.cp(f, loc)\}$$

For the definition of $\mathcal{IO}$ steps at the MIPS level, we again have to define the set A_{io}. We need to collect all addresses of memory instructions which implement volatile variable updates. However without the code generation function we do not know where the implementing memory instruction is placed in the code memory region. To this end, we introduce the uninstantiated function *volma* which returns the instruction address for a volatile memory access at a given location *loc* of a C-IL function f in program π.

$$volma : Prog_{\text{C-IL}} \times params_{\text{C-IL}} \times InfoT_{\text{C-IL}} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{B}^{32}$$

To compute the function we naturally also need to know the code base address from the compiler information and information on compiler intrinsics from environment parameter θ. We assume that *volma* is defined for program locations where we expect volatile variable accesses and for external functions in case they are supposed to update the shared memory.[9] Then A_{io} is defined as follows.

$$A_{io} = \{volma(\pi, \theta, info_{IL}, f, loc) \mid f \in \text{dom}(\pi.\mathcal{F}^{\theta}_{\pi}), loc < |\pi.\mathcal{F}^{\theta}_{\pi}(f).P|\} \setminus \{\bot\}$$

Again the theorem allows us to couple uninterrupted sequential MIPS ISA computations with a corresponding C-IL computation. Any uninterrupted ISA computation of a big enough length, that is running out of a consistency point, contains the simulating ISA computation from the theorem as a prefix because without external inputs any ISA computation is only depending on the initial configuration. Thus by induction on the number of consistency points passed one can repeat this argument and find the C-IL computation that is simulated by the original ISA computation.

Note that since we assumed an optimized compiler, the number of memory accesses might be different before and after the compilation. As a consequence, we can not build the simulation relations for temporaries.

[9] In case of external function *rmw*, *volma* returns for location *loc* = 0 the address of the rmw instruction implementing the shared memory access. Note that there must exist only one such instruction.

5.7.4 *Cosmos* Model Simulation

Using the sequential simulation theorems in an interleaved execution trace, we now aim to establish a system-wide simulation between two block machine computations (d, κ) and (e, ν). The simulated (concrete) computation (d, κ) need not be complete. However (e, ν) is a complete block machine computation. In Section 5.7.7 we will reduce reasoning to simulation between *complete block machine computations*.

Consistency Blocks and Complete Block Machine Computations

We already introduced the notions of complete and incomplete consistency blocks informally. Now we want to give a formal definition. Consistency blocks start in consistency points, i.e. configurations of S_d in which the sequential simulation relation holds wrt. some configuration of S_e and vice versa. Our concurrent simulation theorem is based on the application of our order reduction theorem on S_d where we choose the interleaving-points to be exactly the consistency points as mentioned before. Similarly interleaving-points and consistency points of S_e are identical. These requirements on the instantiation of S_d and S_e are formalized in the following predicate.

Definition 5.69 (Interleaving-Points are Consistency Points) Given a sequential simulation framework R_{S_d,S_e} which relates two *Cosmos* machines S_d and S_e and a simulation parameter $par \in \mathcal{P}$ we define a predicate denoting that in S_d and S_e the interleaving-points are set up to be exactly the consistency points.

$$\begin{aligned} \mathcal{IPCP}(R_{S_d,S_e}, par) \quad &\equiv \quad \forall d \in \mathbb{M}_{S_d}, \alpha \in \Theta_{S_d}. \quad \mathcal{IP}_{\alpha.s}(d, \alpha.in) \leftrightarrow \mathcal{CP}_{\alpha.s}(d, par) \\ &\wedge \quad \forall e \in \mathbb{M}_{S_e}, \beta \in \Theta_{S_e}. \quad \mathcal{IP}_{\beta.s}(e, \beta.in) \leftrightarrow \mathcal{CP}_{\beta.s}(e, par) \end{aligned}$$

If these properties holds we speak of consistency blocks instead of $\mathcal{IP}$ blocks. This is reflected in the definition of consistency block machine schedules $\kappa \in (\Theta^*_{S_d})^* \cup (\Theta^*_{S_e})^*$.

$$\mathcal{CP}sched(\kappa, par) \equiv Bsched(\kappa) \wedge \mathcal{IPCP}(R_{S_d,S_e}, par)$$

Given a *Cosmos* machine state $d \in \mathbb{K}_{S_d}$ and a simulation parameter *par* as above we can define the set U_c of computation units of d that are currently in consistency points wrt. the simulation parameter *par*.

$$U_c(d, par) \equiv \{p \in \mathbb{N}_{S_d.nu} \mid \mathcal{CP}_p(d, par)\}$$

With the above setting of interleaving-points for *par* thus for any computation (d, α) with $\alpha.ip$ we have $\alpha.s \in U_c(d, par)$. Now a complete block machine computation is a block machine computation where all computation units are in consistency points in every configuration. This is encoded in the following overloaded predicate.

$$\begin{aligned} \mathcal{CP}sched_c(d, \kappa, par) \quad &\equiv \quad \mathcal{CP}sched(\kappa, par) \wedge \forall \kappa', \kappa'', d'. \\ &\qquad \kappa = \kappa'\kappa'' \wedge d \overset{\kappa'}{\longmapsto} d' \rightarrow \forall p \in \mathbb{N}_{nu}. \mathcal{CP}_p(d', par) \\ \mathcal{CP}sched_c(e, \nu, par) \quad &\equiv \quad \mathcal{CP}sched(\nu, par) \wedge \forall \nu', \nu'', e'. \\ &\qquad \nu = \nu'\nu'' \wedge e \overset{\nu'}{\longmapsto} e' \rightarrow \forall p \in \mathbb{N}_{nu}. \mathcal{CP}_p(e', par) \end{aligned}$$

Note that we could prove the reduction of arbitrary consistency block machine schedules to complete ones given that for every machine it is always possible to reach a consistency point again (completability). However, the completability assumption needs to be justified by the simulation running on the machine. In addition, the consistency points are only meaningful in connection with a simulation theorem. Thus, it is useless to treat the reduction of incomplete blocks on a single layer of abstraction. The safety transfer theorem for complete block schedules along with our *Cosmos* model simulation theory will be presented in the subsequent sections. There the verification of ownership-safety and a *Cosmos* model safety property P for all complete block machine schedules running out of a configuration $C \in \mathbb{K}_{S_d} \cup \mathbb{K}_{S_e}$ is defined below with $\Omega = \Omega_{S_d} = \Omega_{S_e}$ and $\Theta = \Theta_{S_d} \cup \Theta_{S_e}$.

$$\begin{aligned} \mathit{safety}_{cB}(C, P, \mathit{par}) \;\equiv\; & \forall \kappa \in (\Theta^*)^*.\ \mathcal{CP}\mathit{sched}_c(C, \kappa, \mathit{par}) \wedge \mathit{comp}(C.M, \lfloor \kappa \rfloor) \\ & \rightarrow\; \exists o \in \Omega^*.\ \mathit{safety}_P(C, \langle \lfloor \kappa \rfloor, o \rangle) \end{aligned}$$

Requirements on Sequential Simulation Relations

Now we define the overall simulation relation between two machine states $d \in \mathbb{K}_{S_d}$ and $e \in \mathbb{K}_{S_e}$. We demand that the local simulation relations hold for all machines in consistency points.

$$\mathit{sim}(d, \mathit{par}, e) \;\equiv\; \forall p \in U_c(d, \mathit{par}).\ \mathit{sim}_p(d, \mathit{par}, e)$$

We will later on require that the simulation relation holds between the corresponding machine states of the consistent *Cosmos* machine computations. This means that there are units in the concrete computation which are at times not coupled with the computation on the abstract simulation layer. More precisely, this is the case for units which have not reached a consistency point again at the end of the computation, i.e., their last block in the block machine schedule is incomplete. Only for complete block machine computations we have that units are coupled in all intermediate machine states. In order to compose the simulations, we assume a certain structure and properties of the simulation relations which enable the composition in the first place. We introduce the following framework for concurrent simulation between S_d and S_e.

Definition 5.70 (Concurrent Simulation Framework) A concurrent simulation framework for *Cosmos* machines S_d and S_e is a pair containing the sequential simulation framework R_{S_d,S_e} as well as a *shared memory and ownership invariant shared-inv* (short: shared invariant) that is coupling and constraining the shared memory and the ownership states of both systems. Let $\mathcal{M}_x = \mathcal{A}_x \rightharpoonup \mathcal{V}_x$ and $\mathbb{O}_x = \mathbb{N}_{nu} \rightarrow 2^{\mathcal{A}_x}$ in:

$$\mathit{shared\text{-}inv} : (\mathcal{M}_d \times 2^{\mathcal{A}_d} \times 2^{\mathcal{A}_d} \times \mathbb{O}_d) \times \mathcal{P} \times (\mathcal{M}_e \times 2^{\mathcal{A}_e} \times 2^{\mathcal{A}_e} \times \mathbb{O}_e) \rightarrow \mathbb{B}$$

We introduce a shorthand that is asserting the shared invariant on two *Cosmos* machine configurations D and E. Let $G_x(C) = (C.m|_{C.\mathcal{S} \cup S_x.\mathcal{R}}, C.\mathcal{S}, S_x.\mathcal{R}, C.\mathcal{G}.\mathcal{O})$ in:

$$\mathit{shared\text{-}inv}(D, \mathit{par}, E) \equiv \mathit{shared\text{-}inv}(G_d(D), \mathit{par}, G_e(E))$$

Recall here that $C.\mathcal{G}.\mathcal{O}$ is the mapping of units to ownership sets that is a part of the *Cosmos* machine ghost state. Also, note that the $\mathit{shared\text{-}inv}(D, \mathit{par}, E)$ depends only on the ownership

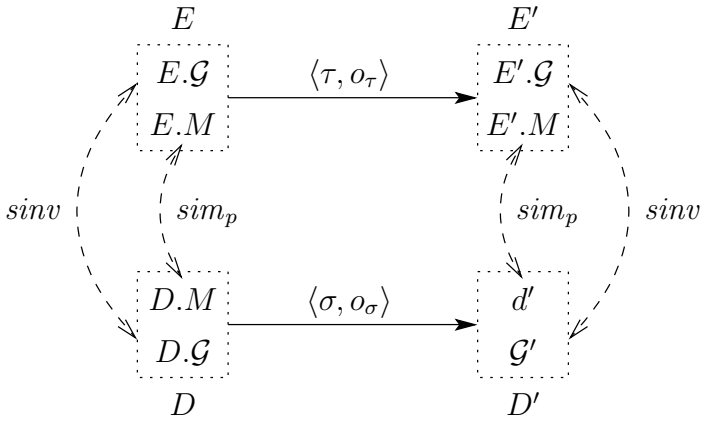


Figure 5.5: Illustration of Assumption 1. The simulating computation $\langle \sigma, o_\sigma \rangle$ must be ownership-safe and preserve the shared invariant *shared-inv*.

state and the portion of memory covered by the shared addresses. Thus, ownership-safe local steps are preserving the shared invariant since by the ownership-policy they do not modify the ownership state nor shared memory.

The shared invariant is introduced as a common abstraction relation to the shared memory and ownership model of S_d and S_e. If $\mathcal{A}_d = \mathcal{A}_e$, then *shared-inv* should be just an identity mapping between the corresponding components of the concrete and abstract simulation levels. However, as we allow to abstract from portions of the abstract memory, the shared invariant may be more complex.

For instance in the C-IL scenario we abstract the function frames from the stack region in memory. While these memory regions are invisible on the abstract level, we would like to protect them via the ownership model from modification by other threads on the concrete level.

The shared invariant is then used to cover such resource abstraction relations and formulate instantiation-specific ownership invariants. We will give examples for the shared invariant later when we instantiate the concurrent simulation framework. Below we formulate constraints on the predicates and the simulation relation introduced above, needed for an integration of the sequential simulation theorems into a concurrent one. These assumptions must be discharged by any instantiation of the concurrent simulation framework.

The most important assumption is stated first. On the one hand we require computation units of S_d and S_e to maintain *shared-inv* according to the software conditions on computations of S_e and the definition of good behaviour for computations of S_d.

Moreover, we need to assume an ownership-safety transfer theorem about the simulation which is essential in the construction of a pervasive concurrent model stack using ownership-based order reduction.

Assumption 1 (Safety Transfer and *shared-inv* Preservation) Consider a concurrent simulation framework $(R_{S_d,S_e}, \textit{shared-inv})$ and a complete consistency block computation $(D.M, \sigma)$ that is implementing an abstract consistency block $(E.M, \tau)$. We assume that (i) the concrete computation is well-behaved, leading into state $d' \in \mathbb{M}_{S_d}$, σ is a consistency block of p, and

both schedules contain the same number of $\mathcal{IO}$ steps but at most one. Moreover (ii) τ is also a consistency block of p, the computation is safe according to ownership annotation o_τ, and leads into $E' \in \mathbb{M}_{S_e}$ obeying the software conditions on S_e. Finally (iii) the simulation relation for p and the shared invariant holds between $D.M$ and $E.M$, and the simulation relation holds also for the resulting configurations.

$$\begin{aligned}
&\forall D, d', E, E', \sigma, \tau, o_\tau, p, par.\\
&\quad (i) \quad D.M \overset{\sigma}{\longmapsto} d' \wedge blk(\sigma, p) \wedge one\mathcal{IO}(\sigma, \tau) \wedge wb(D.M, \sigma, par)\\
&\quad (ii) \quad E \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E' \wedge blk(\tau, p) \wedge safe(E, \langle \tau, o_\tau \rangle) \wedge sc(E.M, \tau, par)\\
&\quad (iii) \quad sim_p(D.M, par, E.M) \wedge shared\text{-}inv(D, par, E) \wedge sim_p(d', par, E'.M)
\end{aligned}$$

Then there exists an ownership annotation o_σ for σ, such that the annotated concrete computation (i) results in d' and a ghost state $\mathcal{G}'$, (ii) it is ownership-safe, and (iii) preserves *shared-inv*.

$$\begin{aligned}
\rightarrow \quad \exists o_\sigma \in \Omega^*_{S_d}, \mathcal{G}'. \quad &(i) \quad D \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d', \mathcal{G}')\\
&(ii) \quad safe(D, \langle \sigma, o_\sigma \rangle)\\
&(iii) \quad shared\text{-}inv((d', \mathcal{G}'), par, E')
\end{aligned}$$

See Fig. 5.5 for an illustration. For C-IL in order to discharge the assumption we would need to show, e.g., that volatile accesses are compiled correctly such that the correct addresses are accessed. Additionally we would need to prove that the memory accesses implementing stack operations are only targeting the stack region and that ownership on the concrete level can be set up such that these memory accesses are safe.

Note that above we do not restrict in any way the ownership transfer on S_e. This means conversely that *shared-inv* can in fact only restrict the ownership state of S_d that is not covered by $\mathcal{A}_e$. Moreover, assumption $safe(E, \langle \tau, o_\tau \rangle)$ and the shared invariant between D and E imply $inv(D)$. The sequential simulation relation does not cover the ownership state but is needed for technical reasons, too. We show this as a corollary.

Corollary 1 *If two ghost configurations $\mathcal{G}_d$ and $\mathcal{G}_e$ are coupled by the shared invariant and the simulation relation for any p, then the ownership invariant is transfered from $\mathcal{G}_e$ to $\mathcal{G}_d$.*

$$(\exists M_d, M_e.\ shared\text{-}inv((M_d, \mathcal{G}_d), par, (M_d, \mathcal{G}_e)) \wedge sim_p(M_d, par, M_e)) \wedge inv(\mathcal{G}_e) \rightarrow inv(\mathcal{G}_d)$$

Proof: By $\sigma = \tau = \varepsilon$ the hypotheses of Assumption 1 applied for $D = (M_d, \mathcal{G}_d)$ and $E = (M_e, \mathcal{G}_e)$ collapse to $sim_p(D.M, par, E.M)$, $shared\text{-}inv(D, par, E)$ and $inv(E)$ which hold by our hypothesis. Thus we have $safe(D, \varepsilon)$ which in turn implies $inv(D)$. □

Below we introduce another property which is needed to establish the sequential consistency relations in a concurrent setting.

Assumption 2 (Preservation of sim_p) The sequential simulation relation for unit p only depends on p's local state and the memory covered by the shared invariant.

$$\begin{aligned}
&\forall D, D' \in \mathbb{K}_{S_d}, E, E' \in \mathbb{K}_{S_e}, par \in \mathcal{P}, p \in \mathbb{N}_{nu}.\\
&\qquad sim_p(D.M, par, E.M) \wedge D \approx_p D' \wedge E \approx_p E' \wedge shared\text{-}inv(D', par, E')\\
&\qquad\qquad \rightarrow sim_p(D'.M, par, E'.M)
\end{aligned}$$

This assumption allows us to maintain the simulation during environment steps. Furthermore, the well-formedness of machine states cannot be broken by safe steps of other participants in the system if they were maintaining the shared invariant.

Assumption 3 (Preservation of Well-formedness) The well-formedness predicates only depend on the local state of their respective units and the memory covered by the shared invariant. For all $D, D' \in \mathbb{K}_{S_d}$, $E, E' \in \mathbb{K}_{S_e}$, $par \in \mathcal{P}$, and $p \in \mathbb{N}_{nu}$ we have:

$$\begin{aligned} wf_p(D.M) \wedge D \approx_p D' \wedge \textit{shared-inv}(D', par, E') &\rightarrow wf_p(D'.M) \\ wf_p(E.M) \wedge E \approx_p E' \wedge \textit{shared-inv}(D', par, E') &\rightarrow wf_p(E'.M) \end{aligned}$$

5.7.5 Simulation Theorem

With the assumptions stated above we can show a global *Cosmos* model simulation theorem, given computations on the abstract level are proven to be safe wrt. ownership and a *Cosmos* machine safety property P. We claim that it is enough to verify all complete block computations leaving starting state E. This is the crucial prerequisite to enable a safe composition of computations. From a given consistency point, a sequential computation of some unit p into the next consistency point must be safe. We do not treat property transfer for other safety properties than ownership-safety for now. However, we instantiate the *Cosmos* machine safety property P so that it implies the well-formedness of machine states of S_e and that computations obey the software conditions of the abstract simulation layer.

Since obeying the software conditions is a property of steps rather than of states, we extend the unit states of S_e with some history information, recording the occurrence of software condition violations. Thus we use a modified *Cosmos* machine S'_e where each unit gets an additional boolean flag *sc* which is initially 1 and becomes 0 as soon as a step violates the software conditions, i.e., for all $\alpha \in \Theta_{S'_e}$, $e, e' \in \mathbb{M}_{S'_e}$, $par \in R_{S_d,S_e}.\mathcal{P}$ and $p \in \mathbb{N}_{S_e.nu}$ we have:

$$e \overset{\alpha}{\mapsto} e' \rightarrow e'.u(p).sc = e.u(p).sc \wedge sc(e, \alpha, par)$$

Assuming the generalized sequential simulation theorem to be proven and the simulation relations and predicates to be constrained as presented above, we can now show the desired concurrent simulation theorem.

Theorem 5.71 (*Cosmos* Model Simulation Theorem) *Given are two* Cosmos *machine start configurations $D \in \mathbb{K}_{S_d}$ and $E \in \mathbb{K}_{S_e}$ as well as block machine schedule κ and concurrent simulation framework $(R_{S_d,S'_e}, \textit{shared-inv})$. We assume that (i) κ is a suitable consistency block schedule without empty blocks, (ii) that κ is executable from $D.M$ and at least one machine in D is in a consistency point, that (iii) all complete block machine computations running out of E are proven to obey ownership-safety and maintain* Cosmos *machine property P, and that (iv) P implies that every computation unit of S'_e is well-formed and does not violate software conditions. Moreover (v) units of D in consistency-points are well-formed. Finally we require that*

(vi) D and E are consistent wrt. simulation parameter par ∈ $\mathcal{P}$ and the shared invariant holds.

$$
\begin{array}{lll}
\forall D, \kappa, E, par, P. & (i) & \mathcal{CP}sched(\kappa, par) \wedge \forall \lambda \in \kappa.\ \lambda \neq \varepsilon \wedge suit(\lambda) \\
& (ii) & comp(D.M, \lfloor \kappa \rfloor) \wedge \exists p \in \mathbb{N}_{nu}.\ \mathcal{CP}_p(D.M, par) \\
& (iii) & safe_{cB}(E, P, par) \\
& (iv) & \forall E' \in \mathbb{K}_{S'_e}, p \in \mathbb{N}_{nu}.\ P(E') \rightarrow wf_p(E'.M) \wedge E'.u_p.sc \\
& (v) & \forall p \in U_c(D.M, par).\ wf_p(D.M) \\
& (vi) & sim(D.M, par, E.M) \wedge shared\text{-}inv(D, par, E)
\end{array}
$$

If these hypotheses hold we can show that there exists a block machine schedule ν such that (i) ν is complete, has the same length as κ, and describes a Cosmos *machine computation starting in E.M. This computation is simulated by (D.M, κ) and for the resulting machine states M'_d and M'_e we know that (ii) they are well-formed for all units of M'_e and all units M'_d in consistency points, and (iii) the simulation relation. Moreover (iv) the simulating computation and well-behaved and each corresponding pair of consistency blocks contains the same number of $\mathcal{IO}$ steps but at most one. Finally (v) for any ownership annotation $o_\nu \in \Omega^*_{S_e}$ to computation (E.M, ν) that is safe and producing a ghost state $\mathcal{G}'_e$, we can find a corresponding annotation $o_\kappa \in \Omega^*_{S_d}$ for (D.M, κ) resulting (v.a) in ghost state $\mathcal{G}'_d$ such that (v.b) the computation is ownership-safe and (v.c) the shared invariant holds between the resulting* Cosmos *machine configurations.*

$$
\begin{array}{lll}
\exists \nu, M'_e. & (i) & \mathcal{CP}sched_c(E.M, \nu, par) \wedge |\nu| = |\kappa| \wedge D.M \overset{\kappa}{\longmapsto} M'_d \wedge E.M \overset{\nu}{\longmapsto} M'_e \\
& (ii) & \forall p \in \mathbb{N}_{nu}.\ wf_p(M'_e) \wedge \forall p \in U_c(M'_d, par).\ wf_p(M'_d) \\
& (iii) & sim(M'_d, par, M'_e) \\
& (iv) & wb(D.M, \lfloor \kappa \rfloor, par) \wedge \forall j < |\kappa|.\ one\mathcal{IO}(\kappa_j, \nu_j) \\
& (v) & \forall o_\nu, \mathcal{G}'_e.\ E \overset{\langle \lfloor \nu \rfloor, o_\nu \rangle}{\longmapsto} (M'_e, \mathcal{G}'_e) \wedge safe(E, \langle \lfloor \nu \rfloor, o_\nu \rangle) \rightarrow \\
& & \quad \exists o_\kappa, \mathcal{G}'_d. \quad (v.a) \quad D \overset{\langle \lfloor \kappa \rfloor, o_\kappa \rangle}{\longmapsto} (M'_d, \mathcal{G}'_d) \\
& & \qquad\qquad\ \ (v.b) \quad safe(D, \langle \lfloor \kappa \rfloor, o_\kappa \rangle) \\
& & \qquad\qquad\ \ (v.c) \quad shared\text{-}inv((M'_d, \mathcal{G}'_d), par, (M'_e, \mathcal{G}'_e))
\end{array}
$$

The simulation theorem is illustrated in Fig. 5.6. Note that we do not require that all units start with consistency blocks. However, this is implicitly guaranteed for all units running in κ by the definition of block machine schedules and the $\mathcal{IPCP}$ condition. If $\kappa = \varepsilon$ then hypothesis (ii) ensures that *sim* does not hold vacuously between *D* and *E*.

Furthermore, in the simulation theorem a possibly incomplete consistency block machine computation of S_d is simulating a complete consistency block machine computation by S'_e. For the computation units whose final blocks are incomplete, i.e., who have not yet reached another consistency point, the simulation relation is not holding. However in all intermediate states of the block machine computation the shared invariant must hold. For the treatment of incomplete blocks, we thus distinguish two cases.

On one hand, if the incomplete block contains only local steps we can simply omit it and represent it by a stuttering step (i.e., an empty block) on the abstract simulation level, because it does not affect the shared memory or ownership state.

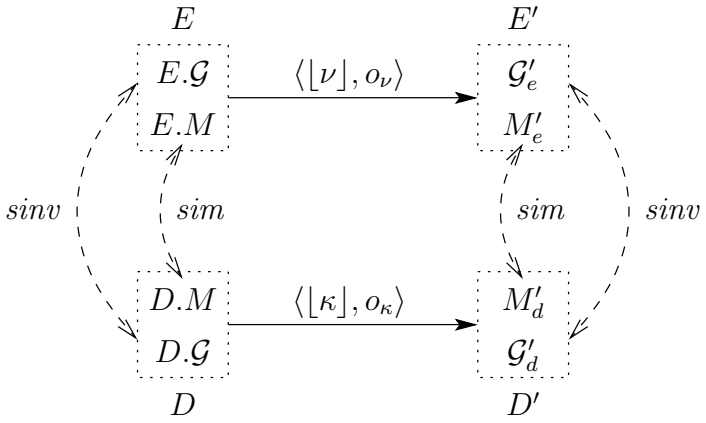


Figure 5.6: The *Cosmos* model simulation theorem. Computation $(D, \langle\lfloor\kappa\rfloor, o_\kappa\rangle)$ is ownership-safe and simulates an abstract computation $(E, \langle\lfloor\nu\rfloor, o_\nu\rangle)$.

On the other hand, if the incomplete block contains an $\mathcal{IO}$ step it may affect the shared memory or ownership state and in order to maintain the shared invariant the incomplete block must be represented properly on the abstract level. To this end, we use the sequential simulation relation completing the block and obtaining the simulated consistency block of the abstract *Cosmos* machine computation. These are the core ideas of the proof of the concurrent simulation.

Note that we need to find a safe annotation for $(D.M, \kappa)$ for any given safe annotation on the abstract level. It does not suffice to simply find one pair of safe annotations for the simulating computations because such a formulation is not applicable in the inductive proof of ownership-safety transfer. The full version of the proof can be looked up in [Bau14].

5.7.6 Applying the Order Reduction Theorem

Our order reduction theorem allows to transfer safety from safe $\mathcal{IP}$ schedules to arbitrarily interleaved *Cosmos* machine schedules. Remember that this safety transfer theorem has two hypotheses, namely that all $\mathcal{IP}$ schedule computations leaving configuration D are safe and fulfil the $\mathcal{IOIP}$ condition, saying that all units start in interleaving-points and that a unit always passes an interleaving-point between two $\mathcal{IO}$ steps. Now it would be desirable if we could use the *Cosmos* model simulation theorem proven above in order to obtain $safety_{\mathcal{IP}}(D, P)$ and $\mathcal{IOIP}_{\mathcal{IP}}(D)$. However, we cannot prove these hypotheses of the order reduction theorem directly. Instead, we can derive two weaker properties from the simulation theorem. With $\theta \in \Theta^*_{S_d}$, $o \in \Omega^*_{S_d}$, the predicates

$$
\begin{aligned}
safety(D, P, suit) &\equiv \forall\theta.\ suit(\theta) \wedge comp(D.M, \theta) \rightarrow \exists o.\ safe_P(D, \langle\theta, o\rangle) \\
safety_{\mathcal{IP}}(D, P, suit) &\equiv \forall\theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \\
&\qquad\qquad \rightarrow \exists o.\ safe_P(D, \langle\theta, o\rangle) \\
\mathcal{IOIP}_{\mathcal{IP}}(D, suit) &\equiv \forall\theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \rightarrow \mathcal{IOIP}(\theta)
\end{aligned}
$$

denote the safety and $\mathcal{IOIP}$ condition for all ($\mathcal{IP}$) schedules that are suitable for simulation. After that, we can show a stronger order reduction theorem that allows to transfer safety properties from the subset of suitable $\mathcal{IP}$ schedules down to suitable arbitrarily interleaved schedules. We furthermore augment the machine states of S_d with a history variable *wb* similar to the *sc* flag of S'_e. The additional semantics for the extended machine S'_d with $d, d' \in \mathbb{M}_{S'_d}$, step $\alpha \in \Theta_{S_d}$, and parameter $par \in \mathcal{P}$ is given by:

$$d \overset{\alpha}{\mapsto} d' \rightarrow d'.u(p).wb = d.u(p).wb \wedge wb(d, \alpha, par)$$

Now we define *Cosmos* machine safety property for a given parameter *par* that denotes good behavior in the past (before *D*) for all computation units.

$$W : \mathbb{K}_{S_d} \rightarrow \mathbb{B} \qquad W(D) \equiv \forall p \in \mathbb{N}_{nu}.\ D.u_p.wb$$

Theorem 5.72 ($\mathcal{IP}$ Order Reduction for Suitable Schedules) *Given a simulation framework $R_{S'_d,S'_e}$ and a* Cosmos *model configuration $D \in \mathbb{K}_{S'_d}$ for which it has been verified that all suitable $\mathcal{IP}$ schedules originating in D are safe wrt. ownership and a* Cosmos *machine property P. Moreover, all suitable $\mathcal{IP}$ schedule computations running out of D obey the $\mathcal{IOIP}$ condition. Then the ownership safety and P hold on all computations with a schedule suitable for simulation that starts in D.*

$$safety_{\mathcal{IP}}(D, P, suit) \wedge \mathcal{IOIP}_{\mathcal{IP}}(D, suit) \rightarrow safety(D, P, suit)$$

Note that for a trivial instantiation of $suit(\alpha) \equiv 1$, the new order reduction theorem implies the old one. The complete proof can be found in [Bau14].

5.7.7 Property Transfer and Complete Block Simulation

Above we have shown the existence of a simulation between any concrete consistency block machine computation and a complete abstract block machine computation. Moreover, we have proven property transfer for memory safety. For the transfer of other safety properties, it is important to remember how the simulation proof was conducted.

The sequential simulation was proven to hold only for units that are in consistency points in the computation of the concrete *Cosmos* machine. For all other units, no statement could be made about their states and locally owned memory regions. However the shared invariant on shared memory and the ownership state was proven to hold in all configurations of a simulating computation.

This has an influence on the kind of properties we can transfer from the abstract down to the concrete simulation level. We will have to distinguish between global and local properties. Moreover, safety properties proven on the abstract level do not translate one-to-one to the concrete level because we are dealing with different *Cosmos* machine instantiations. The "translation" of the verified abstract safety properties to properties of the concrete machine is achieved via the coupling relations between configurations of S'_d and S'_e, i.e., by the shared invariant for global properties, and by the sequential simulation relation for local properties of units in consistency points.

This notion of *simulated Cosmos machine properties* is formalized below. We finish the section by proving transfer of *Cosmos* machine safety properties for complete and incomplete block machine schedules.

Simulated *Cosmos* machine Properties

As explained above we cannot directly transfer a verified *Cosmos* machine property P from the abstract to the concrete simulation level. Naturally P is formulated in terms of S'_e and we cannot apply it to configurations of S'_d. However we can translate P into a *simulated Cosmos machine property* $\hat{Q}$ which holds for $D \in \mathbb{K}_{S'_d}$ iff P holds in a completely consistent state $E \in \mathbb{K}_{S'_e}$. Here we follow the approach of Cohen and Lamport for property transfer [CL98]. Nevertheless, we cannot translate arbitrary properties. First, they must be *divisible* in global and local subproperties.

Definition 5.73 (Divisible *Cosmos* machine Safety Property) We say that P is a divisible *Cosmos* machine safety property on the abstract machine S'_e iff it has the following structure

$$\forall E \in \mathbb{K}_{S'_e}.\ P(E) \equiv P_g(E) \wedge \forall p.\ P_l(E, p)$$

where P_g is a global property which depends only on shared resources and the ownership model and P_l constitutes local properties for each unit of the system. Consequently they are constrained as shown below for any $E, E' \in \mathbb{K}_{S'_e}$.

$$E \overset{s}{\sim} E' \wedge E \overset{o}{\sim} E' \ \rightarrow\ P_g(E) = P_g(E')$$

$$\forall p.\ E \approx_p E' \ \rightarrow\ P_l(E, p) = P_l(E', p)$$

The distinction between global and local properties is motivated by the simulation proof. Global properties are only restricting the shared memory and ownership state, the part of the configuration that is covered by the shared invariant which is holding at all times between simulating computations. Conversely, local properties depend on the local configuration of a single unit, which are only coupled with the implementation at consistency points. Thus, we can translate global properties in all intermediate configurations using the shared invariant and translate local properties in consistency-points using the simulation relation.

Arbitrary safety properties that couple shared memory with local data, or couple the local data of several units, can in general not be translated because the involved computation units might never be in consistency-points at the same time. Technically we forbid safety properties that are stated as a disjunction of global and local properties. However, this is not a crucial restriction, and we could without problems allow properties of the form $P_g(C) \vee P_l(C)$ if needed. The notion of the property translation is formalized as follows.

Definition 5.74 (Simulated *Cosmos* machine Property) Let P be a divisible *Cosmos* machine safety property on $\mathbb{K}_{S'_e}$ and $(R_{S'_d,S'_e}, \textit{shared-inv})$ be a concurrent simulation framework between machines S'_e and S'_d. Then for a given simulation parameter $par \in \mathcal{P}$ the simulated *Cosmos* machine property $\hat{Q}[P, par] : \mathbb{K}_{S'_d} \rightarrow \mathbb{B}$ can be derived by solving the following formula, which

states for any configuration $E \in \mathbb{K}_{S'_e}$ being completely consistent with $D \in \mathbb{K}_{S'_d}$ that $\hat{Q}[P, par]$ holds in D iff P holds in E.

$$\forall D, E.\ \textit{shared-inv}(D, par, E) \wedge \forall p.\ \textit{sim}_p(D.M, par, E.M) \ \rightarrow (\hat{Q}[P, par](D) = P(E))$$

Note that $\hat{Q}[P, par]$ may be undefined for certain properties P.[10] Moreover, as $\hat{Q}[P, par]$ should be a divisible *Cosmos* machine property, we must be able to split it into global part $\hat{Q}[P, par]_g$ and local parts $\hat{Q}[P, par]_l$ such that:

$$\hat{Q}[P, par](D) = \hat{Q}[P, par]_g(D) \wedge \forall p.\ \hat{Q}[P, par]_l(D, p)$$

Consequently the following constraints must hold for $\hat{Q}[P, par]$.

$$\begin{aligned} \forall D, E. \quad & P_g(E) \wedge \textit{shared-inv}(D, par, E) \rightarrow \hat{Q}[P, par]_g(D) \\ \forall D, E, p. \quad & P_l(E, p) \wedge \textit{sim}_p(D.M, par, E.M) \rightarrow \hat{Q}[P, par]_l(D, p) \end{aligned}$$

While it is desirable to have local properties hold for all units, we have seen that for configurations in incomplete consistency block machine computations there are units for which the sequential simulation and thus local simulated properties do not hold. Therefore, we have to relax the definition of simulated properties and introduce *incompletely simulated Cosmos machine properties*.

Definition 5.75 (Incompletely Simulated Cosmos Machine Property) For a given *Cosmos* machine property P, concurrent simulation framework $(R_{S'_d,S'_e}, \textit{shared-inv})$, simulation parameter $par \in \mathcal{P}$, and configurations $D \in \mathbb{K}_{S'_d}, E \in \mathbb{K}_{S'_e}$ we define an incompletely simulated *Cosmos* machine property $Q[P, par] : \mathbb{K}_d \rightarrow \mathbb{B}$ below.

$$Q[P, par](D) \ \equiv \ \hat{Q}[P, par]_g(D) \wedge \forall p \in U_c(D.M, par).\ \hat{Q}[P, par]_l(D, p)$$

Its definition uses the global and local parts of the simulated *Cosmos* machine property $\hat{Q}[P, par]$. The global part should hold for all configurations in a block schedule D and the local properties only if the corresponding machine is in a consistency point.

Property Transfer

In order to prove the transfer of safety properties from the abstraction simulation level down to arbitrary consistency block schedules on the concrete level, we first define a shorthand for the simulation hypotheses.

Definition 5.76 (Simulation Hypotheses) We define a predicate *simh* to denote the hypotheses of the concurrent simulation theorem for a framework $(R_{S'_d,S'_e}, \textit{shared-inv})$, start configurations $D \in \mathbb{K}_{S'_d}$, $E \in \mathbb{K}_{S'_e}$, and a simulation parameter $par \in \mathcal{P}$. We demand that (i) all units D in consistency points are well-formed for all units, (ii) at least one unit is in a consistency point and

[10] This can be the case when P argues about components of S'_e that are not coupled with the concrete level S'_d via the simulation relation and shared invariant. Typically ghost state components fall into this category if they do not have counterparts in the ghost state of the implementation.

for all units the *wb* flag is set to true, and (iii) consistent wrt. the simulation relation and shared invariant. We assume to have proven the sequential simulation theorem according to $R_{S'_d,S'_e}$ fulfilling the $\mathcal{IPCP}$ condition and Assumptions 1-3. Moreover (iv) memory safety is verified for all complete block computations starting in E along with a property P that (v) implies that computations running out of E obey the software conditions and preserve well-formedness.

$$
\begin{array}{lll}
simh(D,E,P,par) & \equiv & (i) \quad \forall p \in U_c(D.M, par).\ wf_p(D.M) \\
& & (ii) \quad \exists p \in \mathbb{N}_{nu}.\ \mathcal{CP}_p(D.M, par) \wedge W(D) \\
& & (iii) \quad sim(D.M, par, E.M) \wedge shared\text{-}inv(D, par, E) \\
& & (iv) \quad safety_{cB}(E, par, P) \wedge \mathcal{IPCP}(R_{S'_d,S'_e}, par) \\
& & (v) \quad \forall E' \in \mathbb{K}_{S'_e}, p \in \mathbb{N}_{nu}.\ P(E') \rightarrow wf_p(E'.M) \wedge E'.u_p.sc
\end{array}
$$

Finally, we prove the transfer of safety properties from the abstract simulation level down to arbitrary consistency block schedules on the concrete level.

Theorem 5.77 (Simulated Safety Property Transfer) *Given are a concurrent simulation framework consistent* $(R_{S'_d,S'_e}, shared\text{-}inv)$ *with* $par \in \mathcal{P}$ *and start configurations* $D \in \mathbb{K}_{S'_d}$, $E \in \mathbb{K}_{S'_e}$ *such that the simulation hypotheses are fulfilled. In particular if we have verified ownership-safety and a* Cosmos *machine property P for all complete block machine computations starting in E and P translates into the incompletely simulated* Cosmos *machine property* $Q[P, par]$*, then any suitable* Cosmos *machine schedule leaving D is safe wrt. ownership,* $Q[P, par]$ *holds for all reachable configurations, and all implementing computations are well-behaved.*

$$simh(D,E,P,par) \ \rightarrow \ safety(D, Q[P,par] \wedge W, suit)$$

Thus, the incompletely simulated *Cosmos* machine property for any P is maintained on the concrete level by the concurrent simulation. However in order the illustrate the usability of our framework for reordering and simulation we will return to our *Cosmos* model instantiations below and establish the concurrent simulation theorems between MIPS and C-IL.

5.7.8 Instantiations

In the previous chapters, we have introduced the *Cosmos* machines S^n_{MIPS} and $S^n_{\text{C-IL}}$ which where instantiated according to the MIPS and C-IL semantics presented earlier. We also defined sequential simulation relations for the C-IL resulting in programs running on the MIPS ISA level. In the remainder of this chapter, we will instantiate our concurrent simulation framework accordingly. Since we already instantiated the sequential simulation framework, we only have to instantiate the shared invariants and prove the safety transfer. Note that for the simulation we set parameter A_{code} of the MIPS machine equal to $\{a \in \mathcal{A} \mid \langle a \rangle \in CR\}$.

Shared Invariant and Concurrent Simulation Assumptions

For establishing a concurrent simulation between MIPS and C-IL, we first of all need to define the invariant about the shared memory and the ownership state. In general we demand that the shared memory, as well as the ownership configuration, is identical. Nevertheless, we need

to take into account that for C-IL the code and stack region is excluded from the memory address range. On the ISA level the stack region dedicated to unit p is always owned by p. By construction, the code region lies in the set of read-only addresses.

Definition 5.78 (Shared Invariant for Concurrent MIPS–C-IL Simulation) Given memories m_h, m_{IL}, read-only sets $\mathcal{R}_h$, $\mathcal{R}_{IL}$, sets of shared addresses $\mathcal{S}_h$ and $\mathcal{S}_{IL}$, as well as ownership mappings O_h and O_{IL}, we define the shared invariant for concurrent simulation of $S^n_{\text{C-IL}}$ by S^n_{MIPS} wrt. assembler information $info_{IL}$ as follows. We demand (i) that memory contents are equal for all but the stack and code regions, that (ii) the shared addresses are equal, that (iii) the read-only addresses on the MIPS level contain all read-only addresses from C-IL plus the code region, and (iv) that all units own the same addresses on the MIPS level as on the C-IL level plus the individual stack region.

$$
\begin{aligned}
&\textit{shared-inv}^{\text{C-IL}}_{\text{MIPS}}((m_h, \mathcal{S}_h, \mathcal{R}_h, O_h), \textit{info}_{IL}, (m_{IL}, \mathcal{S}_{IL}, \mathcal{R}_{IL}, O_{IL})) \;\equiv \\
&\quad \textit{(i)} \quad m_h|_{S^n_{\text{C-IL}}.\mathcal{A}} = m_{IL} \\
&\quad \textit{(ii)} \quad \mathcal{S}_h = \mathcal{S}_{IL} \\
&\quad \textit{(iii)} \quad \mathcal{R}_h = \mathcal{R}_{IL} \cup CR \\
&\quad \textit{(vi)} \quad \forall p \in \mathbb{N}_{nu}.\, O_h(p) = O_{IL}(p) \cup StR_p
\end{aligned}
$$

Thus we obtain concurrent simulation framework $(R_{S^n_{\text{MIPS}}, S^n_{\text{C-IL}}}, \textit{shared-inv}^{\text{C-IL}}_{\text{MIPS}})$ for which Assumptions 1-3 are to be proven. Again, as we do not know the C-IL compilation function, the part of Assumption 1 demanding that the compilation preserves safety wrt. the memory access ownership policy cannot be discharged here. The proof of the preservation of $\textit{shared-inv}^{\text{C-IL}}_{\text{MIPS}}$, Assumptions 2 and 3 can be found in [Bau14].

Proving Safety Transfer

It remains to prove that for the MIPS simulation of an ownership-safe C-IL computation we can also find a safe ownership annotation. In order to apply our programming discipline, we need to instantiate the safety property P and Q. For all configuration $c_{IL} \in \mathbb{K}_{S^n_{\text{C-IL}}}$, $c_{\text{MIPS}} \in \mathbb{K}_{S^n_{\text{MIPS}}}$ and the step α we have the following axillary definitions. The corresponding local C-IL and MIPS configuration is defined as

$$
\begin{aligned}
p &= \alpha.s \\
c_{IL_p} &= (\lceil c_{IL}.M.m \rceil, c_{IL}.M.u_p.s) = (\mathcal{M}, s) \\
c_{\text{MIPS}_p} &= (\lceil c_{IL}.M.m \rceil, c_{\text{MIPS}}.u_p.c) = (m, c)
\end{aligned}
$$

The next instruction in the local MIPS configuration is defined as

$$
I = c_{\text{MIPS}_p}.m(c_{\text{MIPS}_p}.pc)
$$

The next statement in the local C-IL configuration is defined as

$$
\textit{stmt} = \textit{stmt}_{next}(\pi, c_{IL_p})
$$

The value of the corresponding counter is defined as

$$n_{IL} = c_{IL}.M.u_p.n$$
$$n_{\text{MIPS}} = c_{\text{MIPS}}.M.u_p.n$$

Then we define the updated temporaries by read or rmw in both machines

$$v_{IL} = \begin{cases} a & : \quad stmt = (e = e') \wedge [\![e']\!]^{\pi,\theta}_{c_{IL}} = \mathbf{val}(a,t) \wedge a \in \mathbb{B}^{32} \vee \\ & \quad\; rmw^{\pi,\theta}_{c_{IL_p}}(stmt, a, u, v, r, p, in) \\ \bot & : \quad otherwise \end{cases}$$
$$\vartheta'_{IL}(c_{IL}, p) = c_{IL}.M.u_p.\vartheta(R_{n_{IL}+1} \mapsto v_{IL})$$
$$\vartheta'_{\text{MIPS}}(c_{\text{MIPS}}, p) = c_{\text{MIPS}}.M.u_p.\vartheta(I_{n_{\text{MIPS}}+1} \mapsto I)(R_{n_{\text{MIPS}}+1} \mapsto lv(c_{\text{MIPS}_p}.m(ea(c_{\text{MIPS}_p}.c, I))))$$

Definition 5.79 (Safety Property at C-IL Level) If the next step is an $\mathcal{IO}$ step then (i) the dirty bit should be cleared for volatile reads. (ii) the ownership annotations are generated by a function $og^{\text{C-IL}}_{cos}$[11] with local components. The safety property $P_{og^{\text{C-IL}}_{cos}}$ is defined as

$$\begin{aligned} \forall c_{IL} \in \mathbb{K}_{S^n_{\text{C-IL}}}.\ & P_{og^{\text{C-IL}}_{cos}}(c_{IL}) \equiv \alpha.io \rightarrow \\ & (volr^{\pi,\theta}_f(stmt) \rightarrow \neg c_{IL}.u_p.\mathcal{D}) \wedge \\ & (\alpha.Acq, \alpha.Loc, \alpha.Rel) = og^{\text{C-IL}}_{cos}(c_{IL_p}.s, \vartheta'_{IL}(c_{IL}, p)) \end{aligned}$$

We instantiate predicate P as following to fulfill the simulation hypothesis:

$$P(c_{IL}) \equiv P_{og^{\text{C-IL}}_{cos}}(c_{IL}) \wedge wf_p(c_{IL}.M) \wedge c_{IL}.u_p.sc$$

Definition 5.80 (Safety Property at MIPS Level) Since at the MIPS level we already defined the safety property in the last section, we use it to instantiate the predicate Q.

$$Q(c_{\text{MIPS}}) \equiv P_{og^{\text{MIPS}}_{cos}}(c_{\text{MIPS}})$$

The instantiated safety property P and Q only depend on local components of the corresponding configuration. Thus, we can transfer the property by theorem 5.77. Next, we want to identically transfer the ownership annotations from C-IL to MIPS.

Ownership Identity Mapping Assuming we have one C-IL configuration c_{IL} and one MIPS configuration c_{MIPS}. The following compiler consistency relation is satisfied.

$$consis_{\text{C-IL}}(c_{IL_p}, \pi, \theta, info_{IL}, c_{\text{MIPS}_p})$$

[11] The ownership annotation generation function $og^{\text{C-IL}}_{cos}$ is an abstraction of specification codes (or ghost codes) which are used to update the specification states (or ghost codes). In a C verifier like VCC, the code to be verified is annotated with specification codes and specification states. The definition of $og^{\text{C-IL}}_{cos}$ depends on the specific annotated program and the specific C verifier.

From the definition of C-IL control consistency $consis^{control}_{\text{C-IL}}(c_{IL_p}, info_{IL}, c_{\text{MIPS}_p})$ we know that $c_{\text{MIPS}_p}.pc$ points to the start of the compiled code for the current statement in the C-IL machine. If we keep stepping the unit p of both machines then from the property of cpl we know that each $\mathcal{IO}$ point of C-IL machine has a counterpart in the MIPS machine. We let the corresponding configurations be c'_{MIPS} and c'_{IL}. As a consequence, we have:

$$c'_{\text{MIPS}_p}.pc = volma(\pi, \theta, info_{IL}, f_{top}(c'_{IL_p}), loc_{top}(c'_{IL_p}))$$

We define the value of $og^{\text{MIPS}}_{cos}(c'_{\text{MIPS}_p}.c, \vartheta'_{\text{MIPS}}(c'_{IL}, p))$ as

$$og^{\text{C-IL}}_{cos}(c'_{IL_p}.s, \vartheta'_{IL}(c'_{\text{MIPS}}, p))$$

In the remaining portion of this paragraph, we will state the function og^{MIPS}_{cos} is well defined. What we need to prove is for

- an initial MIPS *Cosmos* machine $c^0_{\text{MIPS}} \in \mathbb{K}_{S^n_{\text{MIPS}}}$
- an initial C-IL *Cosmos* machine $c^0_{\text{C-IL}} \in \mathbb{K}_{S^n_{\text{C-IL}}}$
- a MIPS *Cosmos* machine computation κ
- a C-IL program π and a parameter $info_{\text{C-IL}}$
- an instantiated concurrent simulation framework and property

which fulfill the *Cosmos* model simulation theorem. We need to prove there exists a unique C-IL *Cosmos* machine computation which simulations κ. The only non-determinism during the execution might come from the intrinsic *rmw*. The input depends on the result of the comparing between the read value and the compare value. Thus, the *rmw* does not introduce any non-determinism, and the uniqueness of the execution trace is guaranteed.

Safety Transfer Since ownership state is extended identically going from C-IL to MIPS, using the following facts, we can justify that compiled safe code does not break the memory access policy.

- The compiled code is placed in the code region, which is the only target of instruction fetch for well-behaved code. Moreover, we do not have self-modifying code. Therefore, only instructions generated by the C-IL compiler are executed.
- For any implementation of a C-IL statement only the stack and the memory footprints of the involved expressions may be read or written. Note that this does not follow from compiler consistency for intermediate computation steps.
- Local variables are allocated in the stack region which is owned by the executing computation unit. Therefore, if local variable accesses are compiled, so that they access only the stack, the operation is ownership-safe.

- Since there is at most one $\mathcal{IO}$ step per-consistency block where ownership can change, the ownership state for the $\mathcal{IO}$ step and all previous local steps is the same and by the shared invariant consistent with the ownership state on the C-IL level before the $\mathcal{IO}$ step. Similarly, all successor steps of the $\mathcal{IO}$ operation are computed with the same ownership state that is consistent by the shared invariant with the ownership state after the $\mathcal{IO}$ step on the C-IL level.
- As the same shared memory addresses are accessed wrt. the same ownership state, the ownership-safety of memory access can be transferred.

This finishes our instantiation of the concurrent simulation framework between MIPS and C-IL as well as the chapter on concurrent simulation.

6 Conclusion and Future Work

6.1 Conclusion

To the best of our knowledge, this thesis presents the first store buffer reduction theorem with MMU, as well as the first application of the theorem on ISA level and C level. In this thesis, we first proposed a programming discipline that guarantees the SC execution on a TSO machine with MMU. We formally defined an SB machine model and an abstract machine model. Under the programming discipline, we proved the store buffer reduction theorem with MMU, which is a simulation theorem between the abstract machine and the SB machine. We also introduced the ownership theory to make our proof go through. In the remainder of this thesis, we introduced four kinds of ISAs.

- MIPS-86. To apply the SB reduction theorem to ISA level, An ISA named MIPS-86 is introduced, which is a MIPS core with the x86/64 like memory system (including MMU and SB).
- SB reduced MIPS-86. After applied the SB reduction theorem with MMU to MIPS-86, we obtained the SB reduced MIPS-86, which is MIPS-86 without the SB.
- SB MIPS. This kind of ISA is introduced for user program in which the MMU and interrupts are invisible. SB MIPS is MIPS-86 without the MMU and interrupts.
- MIPS. After applied the SB reduction theorem to SB MIPS, we got the MIPS ISA, which is MIPS-86 without the SB, the MMU and interrupts.

In the next portion of this thesis, we applied the SB reduction theorem with MMU to the ISA level. We introduced MIPS-86 as well as the SB reduced MIPS-86 ISA. We also instantiated the abstract machine and the SB machine with an ISA very alike to MIPS-86. The main difference is that in the instantiated machine, the execution of one instruction is divided up to five phases, while in the MIPS-86 machine, each instruction is atomic. As a consequence, by applying we mean that

1. simulate the MIPS-86 machine execution with an SB machine execution, which is trivial and omitted in this thesis.
2. simulate the abstract machine execution with an SB reduced MIPS-86 machine execution. First, we need to provide the ownership semantics to the MIPS-86 machine. As a consequence, we introduced the *Cosmos* model and instantiated it with the SB reduced MIPS-86 machine (we called it the SB reduced MIPS-86 *Cosmos* machine) which gives

us the semantics with ownership. Then, we proved a simulation theorem between the SB reduced MIPS-86 *Cosmos* machine and the instantiated abstract machine.

In the last portion of the thesis, we applied the SB reduction theorem to the parallel C level for user programs. Since the MMU is not visible for a user program, we tailored our previous results for the machines without MMUs. We also introduced the C-IL and instantiated the *Cosmos* machine with a C-IL machine. At last we presented a simulation between a MIPS *Cosmos* machine[1] and a C-IL *Cosmos* machine. With the series of the simulation theorems, we mapped the programming discipline to the parallel C level.

The conclusion section is ended by revisiting the first two examples mentioned in Chapter 1. We apply the programming discipline to these examples with identical initial conditions and argue that the SC is maintained.

```
T1: a1:=1                    T2: a2:=1
    FENCE                        FENCE
    if(a2==0)                    if(a1==0)
      critical section             critical section
```

In the first example, a `FENCE` instruction is inserted between the shared variable write and read according to the programming discipline. In this case, when the program reaches the `if` statement, we can be sure that at least one of the stores already emerged from its SB which only allow one thread to enter the critical section. The SC is maintained in this example.

```
T1: pte2.p:=0          MMU1: pte1.a:=1
    FENCE                    t1:= pte2
    t0:=pte1.a
```

Consider the same TSO execution as in Chapter 1 where the steps of the thread are executed before the steps of the MMU. The inserted `FENCE` guarantees the updating of `pte2.p` is visible to the MMU and results a page fault both in the SC execution and the TSO execution.

6.2 Future Work

As it is mentioned in Chapter 1, one initial goal of our thesis was to apply the SB reduction theorem with MMU to the parallel C program that modifies the page table. We switched the goal due to the lack of the multicore compiler correctness theorem with MMU. Thus, our possible future work is to present and prove that theorem which includes:

- a reordering theorem with MMU. Unlike the local processor steps, the MMU steps can be advanced but not postponed because of the monotonicity of TLBs. Also, both the global order of MMU steps from different processors and the local order of the MMU steps from one processor must be maintained during the reordering. We can use the same technology as in Chapter 4 by postponing all the local processor steps as far as possible for finite computations.

[1] A *Cosmos* machine instantiated with MIPS.

- a sequential compiler correctness theorem with MMU. After reordering, we can obtain this theorem by inserting new compiler consistency points before and after each MMU step.

Bibliography

[Adv11] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.19 edition, September 2011.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[APST10] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an os microkernel: Inline assembly, memory consumption, concurrent devices. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, pages 71–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[Bau14] Christoph Baumann. *Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems*. PhD thesis, Saarland University, Saarbrücken, 2014.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reason.*, 5(4):411–428, November 1989.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together–formal verification of the vamp. *International Journal on Software Tools for Technology Transfer*, 8(4-5):411–430, 2006.

[CCK13] G. Chen, E. Cohen, and M. Kovalev. Store buffer reduction with mmus: Complete paper-and-pencil proof. Technical report, Saarland University, Saarbrüken, 2013.

[CCK14] Geng Chen, Ernie Cohen, and Mikhail Kovalev. Store buffer reduction with mmus. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, Lecture Notes in Computer Science, pages 117–132. Springer International Publishing, 2014.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331, 1998.

[CPS13] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of Multi Core Hypervisor Verification. In *Proceedings of the 39th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '13, Berlin, Heidelberg, 2013. Springer-Verlag.

[CS10a] Ernie Cohen and Bert Schirmer. From Total Store Order to Sequential Consistency: A Practical Reduction Theorem. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418. Springer Berlin / Heidelberg, 2010.

[CS10b] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *ITP*, pages 403–418, 2010.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association.

[GHLP05a] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In *Theorem Proving in Higher Order Logics*, pages 1–16. Springer, 2005.

[GHLP05b] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the Correctness of Operating System Kernels. In J. Hurd and T. Melham, editors, *Theorem Proving in High Order Logics (TPHOLs) 2005*, Lecture Notes in Computer Science, Oxford, U.K., 2005. Springer.

[GMY12] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: Sequentially consistent specifications of tso libraries. In *Distributed Computing*, pages 31–45. Springer, 2012.

[HL09] Mark Hillebrand and Dirk Leinenbach. Formal Verification of a Reader-Writer Lock Implementation in C. In *4th International Workshop on Systems Software Verification (SSV09)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 123–141. Elsevier Science B. V., 2009.

[HP08] MarkA. Hillebrand and WolfgangJ. Paul. On the architecture of system verification environments. In Karen Yorav, editor, *Hardware and Software: Verification and Testing*, volume 4899 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin Heidelberg, 2008.

[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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kle09] Gerwin Klein. Operating system verification–An overview. *Sadhana*, 34(1):27–69, 2009.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification–from verified programs to verified systems. *Electronic Notes in Theoretical Computer Science*, 217:23–40, 2008.

[Mic] Microsoft Research. The VCC Manual. URL: `http://vcc.codeplex.com`.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[Obe] Jonas Oberhauser. A Simpler Reduction Theorem for x86-TSO. In *Accepted by VSTTE 2015*.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009.

[PO] Lutsyk Petro Paul, Wolfgang J. and Jonas Oberhauser. *Multi Core System Architecture*.

[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 language semantics and automated verification of a small hypervisor*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 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 Verified Software: Theories, Tools, and Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, Philadelphia, USA, 2012. Springer Berlin / Heidelberg.

[Ver07] Verisoft Consortium. The Verisoft Project. URL: `http://www.verisoft.de/`, 2003-2007.

[Ver10] Verisoft Consortium. The Verisoft-XT Project. URL: `http://www.verisoftxt.de/`, 2007-2010.

[WG94] David L Weaver and Tom Gremond. *The SPARC architecture manual*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

Index