UNIVERSITÄT
DES
SAARLANDES

Provably Sound Semantics Stack for Multi-Core System Programming with Kernel Threads

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von

Artem Alekhin

Saarbrücken, August 2016

Tag des Kolloquiums:	24.02.2017
Dekan:	Univ.-Prof. Dr. Frank-Olaf Schreyer
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Holger Hermanns
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Andreas Podelski
Akademischer Mitarbeiter:	Dr. Hristo Pentchev

Abstract

Operating systems and hypervisors (e.g., Microsoft Hyper-V) for multi-core processor architectures are usually implemented in high-level stack-based programming languages integrated with mechanisms for the multi-threaded task execution as well as the access to low-level hardware features. Guaranteeing the functional correctness of such systems is considered to be a challenge in the field of formal verification because it requires a sound concurrent computational model comprising programming semantics and steps of specific hardware components visible for the system programmer.

In this doctoral thesis we address the aforementioned issue and present a provably sound concurrent model of kernel threads executing C code mixed with assembly, and basic thread operations (i.e., creation, switch, exit, etc.), needed for the thread management in OS and hypervisor kernels running on industrial-like multi-core machines.

For the justification of the model, we establish a semantics stack, where on its bottom the multi-core instruction set architecture performing arbitrarily interleaved steps executes binary code of guests/processes being virtualized and the compiled source code of the kernel linked with a library implementing the threads. After extending an existing general theory for concurrent system simulation and by utilising the order reduction of steps under certain safety conditions, we apply the sequential compiler correctness for the concurrent mixed programming semantics, connect the adjacent layers of the model stack, show the required properties transfer between them, and provide a paper-and-pencil proof of the correctness for the kernel threads implementation with lock protected operations and the efficient thread switch based on the stack substitution.

Kurzzusammenfassung

Betriebssysteme und Hypervisoren (z.B. Microsofts Hyper-V) für Mehrkernprozessor-Architekturen sind üblicherweise in stapelbasierten höheren Programmiersprachen implementiert, welche integrierte Mechanismen für die mehrfadige Aufgabenausführung sowie den Zugriff auf niedrige Hardwarefunktionen hat. Die funktionale Korrektheit solcher Systeme zu garantieren wird im Gebiet der formalen Verifikation als Herausforderung angesehen, da sie eine korrektes nebenläufiges Berechnungsmodell benötigt, welches die Programmiersprachensemantik ebenso beinhaltet wie die für Systemprogrammierer sichtbaren Schritte der einzelnen Hardwarekomponenten.

In dieser Dissertation gehen wir die eben genannten Probleme an und präsentieren ein beweisbar korrektes, nebenläufiges Modell von Betriebssystemkernfaden, die gemischten C- und Assemblerkode ausführen, sowie grundlegenden Fadenoperationen (d.h. Erstellung, Wechsel, Stopp, usw.), die für die Fadenverwaltung in Kernen der Betriebssysteme und Hypervisoren, welche auf praxisnahen Mehrkernprozessoren laufen sollen, nötig sind.

Für die Rechtfertigung des Modells richten wir einen semantischen Stapel ein, dessen niedrigste Ebene die Mehrkernprozessorarchitektur ist, die beliebig verschränkte Schritte des Binärkodes von virtualisierten Gästen bzw. Prozessen und den kompilierten Quelltext des Kerns, verlinkt mit einer Bibliothek, welche die Faden implementiert, ausführt. Nach wir eine bereits vorhandene allgemeine Theorie für die Simulation nebenläufiger Systeme erweitern und indem wir die Reihenfolgen-Reduktion von Schritten unter bestimmten Sicherheitsbedingungen verwenden, verbinden wir die benachbarten Ebenen des Modellstapels, zeigen die nötigen Eigenschaftsübertragungen zwischen ihnen, und geben einen Papier-und-Bleistift-Beweis für die Korrektheit der Implementierung der Kernfäden mit durch Sperren geschützten Operationen und einem effizienten Fadenwechsel, welcher auf Stapelsubstitution basiert.

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Dr. Wolfgang Paul for inviting me to join a leading research team in the field of formal verification and his valuable advice on how to make complicated formal theories simple and clear even for "grandmothers" if, of course, they are not professors in mathematics.

I am indebted also to my former colleagues at the chair of Prof. Paul, particularly to Christoph Baumann, for productive discussions about the research topic, found issues and their solutions.

I am deeply thankful to my friends, especially to David Spieler, Natalia Tsetskhladze-Spieler, Farid Naghizade, Olaf Zeitz, and Guido Döhler, for their understanding, constant encouragement, readiness to help when I was in need, and simply for the great time spent together.

And finally, *above all*, I want to thank my parents and grandparents for believing in me, their love, patience, and support over all years of my studies and despite of thousands of kilometres between us during my work on this PhD thesis.

Contents

List of Theorems

1 Introduction

1.1 Motivation and Overview

Formal verification of operating systems and hypervisors represents an ongoing challenge for the scientific community (e.g., see the survey on projects in [Kle09, CPS13, SF10]). Their safety-critical nature and inherent characteristic of being implemented using low-level hardware features make them a target for pervasive formal verification [Moo03, APST10]. In pervasive formal verification, the proof of correctness is given by establishing a model stack from the lowest level (e.g. gate-level hardware construction) to the highest level of abstraction (e.g. application-level C semantics). Between neighboring models of the model stack, simulation theorems powerful enough for the properties transfer, and revealing required conditions and subtleties are established, resulting in a pervasive theory that allows to show correctness of realistic systems.

In order to achieve the goal of the formal verification in case of operating systems and hypervisors, one has to obtain and use for their implementation a sound formal semantics integrating high-level programming language execution with the effects of invoking exposed hardware features at an adequate level of abstraction. Based on that, the correctness of the system implementation can be established by a simulation with an appropriate abstract model (or specification). However, defining such semantics for industrial-like systems is considered to be an absolutely non-trivial task in the presence of thousands pages (e.g., AMD64 Architecture Programmer's Manual [Adv11a, Adv11b]) of informal specification with non-exhaustive description of system functionality, especially when the parallelism is intensively exploited.

One of the fundamental programming concepts enhancing such a parallelism on single- and multi-core architectures is splitting the execution of a process into *threads*, performing separate tasks within a *process* [BC05, Lov10, Han96, But97]. Since threads share the same address space, other resources, and have only distinct local stacks, they are considered to be light-weight in comparison to processes and require much less overhead during scheduling operations as well as do not need specific inter-process communication. Moreover, threads can take advantages of multi-core hardware by being assigned to different processor cores on which they can be executed.[1] By supporting the thread migration via associating the same thread with multiple processors, an operating system kernel can effectively schedule it with respect to the current workload in the system.

In this doctoral thesis we concentrate mostly on two topics crucial for the aforementioned verification field: sound concurrent programming semantics for the implementation and verification of hypervisor and operating system kernels on an industrial-like multi-core processor architecture, and the extension of this semantics with threads allowing to use the whole power of the multi-threading programming paradigm on the kernel level.

[1] "Processor Affinity is the term used for describing the rules for associating certain threads and certain processors." [TMu03]

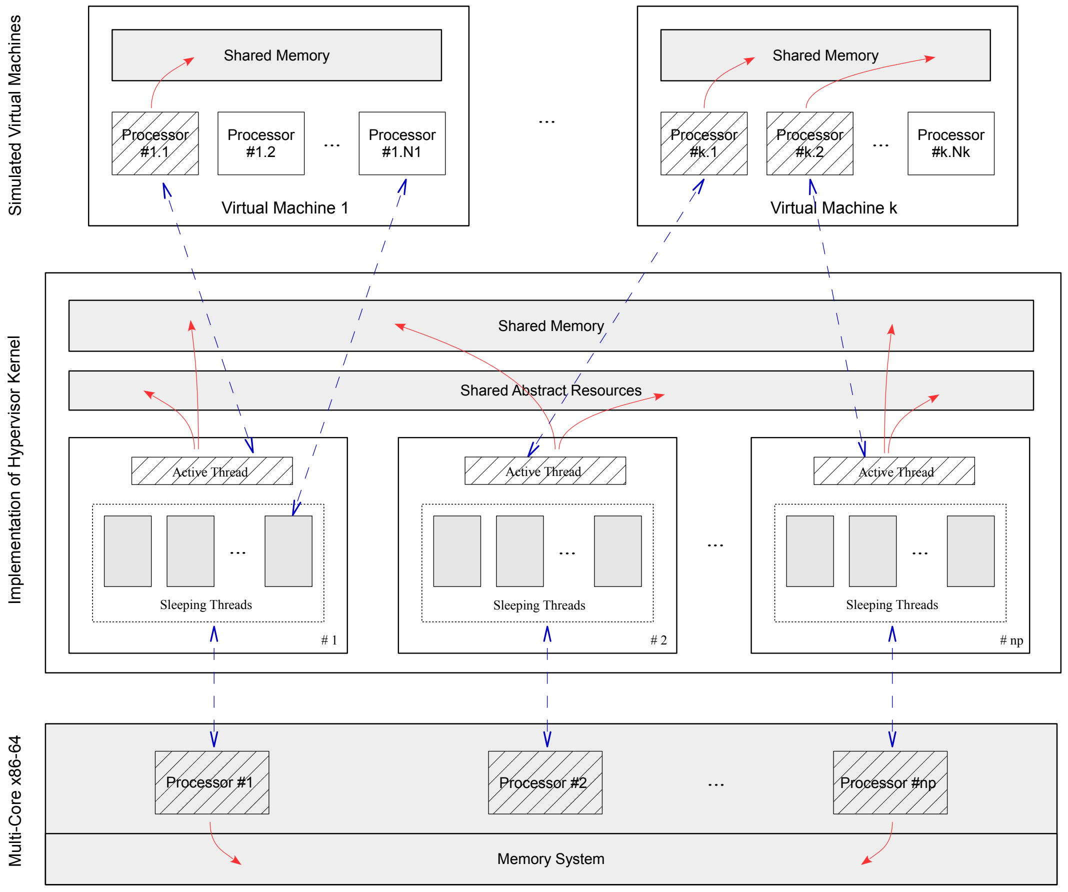

Figure 1.1: Bird's view on hypervisor kernel implemented with threads. Dashed lines show the association between virtual processors and active threads (above), and between the active threads and physical processors executing them (below). The curved arrows depict simultaneous accesses to shared resources.

Contribution and Related Works

The work covered in this thesis was inspired by the Verisoft XT project [The11] aimed at the extension of the pervasive theory developed in the earlier Verisoft project [Ver10] to the industrial multi-core processor system. The project had as one of its goals the formal correctness of Microsoft Hyper-V Hypervisor [LS09]. The kernel layer of this hypervisor consists of threads associated with virtual processors of guests (Figure 1.1). Such virtual processors are scheduled for running on the physical ones by a simple switch of corresponding threads.

The correctness proof of Hyper-V was conducted with the help of VCC [Mic16, CDH^{+}09], a verification tool for concurrent C from Microsoft Research. Though quite a substantial piece of the hypervisor's code was covered, due to the need to redevelop the pervasive theory for the multi-core systems for arguing about the soundness of the VCC application, the research in a similar direction was continued at the chair of Prof. Dr. W. J. Paul after the end of the project.

The work there is basically devoted to the correctness of an academic version of a hypervisor and sound methods for showing it, however, for a simpler MIPS multi-core architecture introduced by Schmaltz [Sch13] with less hardware virtualization features in comparison to the comprehensive x86 model [Deg11] used in Verisoft XT. A survey on the overall theory of multi-core hypervisor verification and a clear road map for the research on this topic were given by Paul, Cohen, and Schmaltz in [CPS13].

Since the x86 and MIPS considered within the Verisoft XT and the research at the chair have rather a relaxed TSO-like (total store ordering) [Owe10] memory model than the sequential consistent memory [Lam79] mostly used by various verification techniques for concurrent programs (e.g., see [App11, CMST10]), Cohern and Schirmer in [CS10] introduced a store buffer flush policy and an ownership discipline guaranteeing the sequential consistency on the TSO. Their results were later extended by Kovalev and Geng Cheng for the case where the memory-management steps are involved [CCK14]. In turn, Oberhauser [Obe16] improved this programming discipline in order to obtain a simpler reduction theorem for x86-TSO.

Since the code of Hyper-V is written in C/assembler, Schmaltz in [Sch13] developed an operational semantics for the C Intermediate Language (C-IL) close to C widely used for system programming. Moreover, she enriched this abstract model with ghost components needed for arguing about VCC soundness. Based on these results, Shadrin in [Sha12, SS12] introduced the integrated high- and low-level programming semantics allowing to perform inter-language calls and returns between C-IL and Macro Assembly. Moreover, he sketched the correctness of an optimizing compiler for such a mixed machine and used it for completing the formal verification of a small hypervisor [AHPP10] on a simplified version of the single-core VAMP architecture [BJK$^+$06]. The semantics and correctness of its implementation was mostly based on the communicating virtual machines (CVM) [GHLP05, IdRT08, Tsy09], a model for a generic operating system kernel communicating with user processes. The CVM for a single-core MIPS architecture and its correctness proof were recently covered in detail in [PBLS15].

In order to apply the compiler correctness on multi-core hardware architecture and be able to specify steps of any more abstract model (e.g., hypervisor and OS kernels) of the pervasive verification stack, Baumann [Bau14b] introduced the ownership-base order reduction and formalized a general simulation theory for concurrent systems with shared memories. Moreover, he adapted separately the C-IL and Macro Assembly semantics, redefined the sequential compiler correctness in both cases for the MIPS architecture, and showed how one can justify their concurrent models. Using these results for C-IL and applying them for a restricted x86 from [Deg11], Kovalev managed to show the TLB virtualization in the context of hypervisor verification, however, by abstracting away the details of the compiler correctness in the concurrent setting. In his work he also assumed the hypervisor kernel construction with threads from Figure 1.1 but did not consider their formal model and correctness.

In the scope of the present doctoral thesis we close this gap and by combining the aforementioned results together and apply the overall theory in detail up to the level of hypervisor/OS kernel implementation on the multi-core MIPS architecture. Along with the detailed revision of the integrated semantics, its compiler correctness and all requirement and software conditions needed for the successful simulation, we extend the model with operations crucial for system programming in the presence of processes/guests steps. Using the obtained sound semantics based on the compiler correctness we formalize a model of kernel threads and answer questions that arose at the beginning of the work covered here: (i) what is the semantics for the kernel implementation with threads, and (ii) how one can implement the stack-based switch of threads using the optimizing C with assembler so that its correctness can be easily guaranteed.

To the best of my knowledge, though there exist quite a few works (e.g., [MPR07, FS05, FSV$^+$06, FSDG08, NYS07, GFSS12]) on the verification of multi-threaded concurrent programs, they either almost exclusively use high-level calculi, or consider threads and operations on them only on the level of assembly programming for simpler processor architectures. Therefore, one can see our results here as the first attempt to argues about the correctness of thread implementation relying on properties of a high-level programming language compiler for an industrial-like multi-core architecture.

Thesis Outline

This doctoral thesis is organized in 9 chapters.

- In the reminder of Chapter 1 we give the basic mathematical notation used throughout the whole work.
- Chapter 2 is devoted to the general model for concurrent system and simulation allowing us to build the overall model stack considered in this thesis. All layers of this stack will be formalized with respect to this theory and, therefore, have a similar structure.
- In Chapter 3 we present the computational model of the multi-core MIPS-86 on which the higher layers of our model stack are implemented.
- In Chapter 4 we argue about a simple store buffer reduction for the MIPS-86 that can be shown with the help of the theory from Chapter 2.
- Chapter 5 provides the definition of the concurrent mixed machine semantics integrating C-IL and Macro Assembly for the MIPS-86 model.
- In Chapter 6 the compiler correctness for the mixed semantics from Chapter 5 is considered in detail so that it can be used in the rest of the work. Moreover, based on the results from Chapter 2 we argue about the soundness of the concurrent mixed machine.
- In Chapter 7 we extend the semantics of the mixed machine with inline assembly execution which also allows to integrate steps of guests and user processes into the model.
- Using the results from the previous chapters, in Chapter 8 we finally introduce the model of kernel threads, give their implementation, and discuss the correctness criteria and proof in the concurrent setting.
- In the last Chapter 9 we shortly summarize the results and discuss directions for future work.

1.2 Basic Notation

In the scope of the work we mostly use the notation and basic mathematical concepts borrowed from the book [KMP14] and the doctoral thesises [Bau14b, Sch13].

Definitions and Shorthands

In order to define new types, predicates, and functions mathematically, we use the identity symbol $\equiv$ marked with def. For instance, one could define the predicate $P(x, y)$ in the following way:

$$P(x,y) \overset{def}{\equiv} Q(x,y) \wedge R(x)$$

Each important named definition is stated in a numbered environment with a name of a newly defined notion.

Moreover, for brevity we also allow to introduce shorthands used locally in a certain context or throughout the whole work. We mark such shorthands with the identity symbol. For example, one could introduce the shorthand

$$p_x \equiv P(x, y)$$

When a formula contains a conjunction we allow to omit the symbol $\wedge$ between the conjuncts and represent them as a numbered list, e.g. we could also define $P(x, y)$ as

$$P(x,y) \overset{def}{\equiv} \begin{array}{ll} \textit{(i)} & Q(x,y) \\ \textit{(ii)} & R(x) \end{array}$$

In case of a definition written in such a form, we then can refer to a certain item by its number in the list of conjuncts, e.g., $P(x, y).(i)$.

Numbers and Sets

In the work we use the following notation for the standard mathematical sets:

- $\mathbb{N} \overset{def}{\equiv} \{1, 2, ...\}$ – the set on natural numbers excluding zero,
- $\mathbb{N}_0 \overset{def}{\equiv} \mathbb{N} \cup \{0\}$ – the set on natural numbers including zero,
- $\mathbb{N}_n \overset{def}{\equiv} \{1, 2, ..., n\}$ – the set of natural numbers in the range from 1 to a non-zero n,
- $\mathbb{Z} \overset{def}{\equiv} \{..., -2, -1, 0, 1, 2, ...\}$ – the set of integers,
- $[i : j] \overset{def}{\equiv} \{i, i + 1, ..., j\}$ – the interval of integers from i to j if $i \leq j$ and empty set otherwise,
- $\mathbb{B} \overset{def}{\equiv} \{0, 1\}$ – the set of Boolean values. In order to refer a Boolean value we also use the well-known term *bit*.

We denote the cardinality of a finite set $\mathbb{A}$ as $\#\mathbb{A}$. In order to select an element from any set $\mathbb{A}$ we use the Hilbert choice operator $\epsilon\mathbb{A}$. Particularly, for a singleton set it returns a unique element $\epsilon\{x\} = x$.

Records

Definition 1.1 (Record Notation for Tuples). Let a set $\mathbb{A}$ be a Cartesian product of sets $\mathbb{A}_1, \mathbb{A}_2, \ldots, \mathbb{A}_k$ that represents a set of *tuples* $c \in \mathbb{A}$ such that

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

Given names (labels) $n_1, n_2, \ldots, n_k$ for all individual elements of tuples in $\mathbb{A}$ such that n_i is associated with an element a_i we talk about $\mathbb{A}$ as a set of *labeled tuples* or *records* and use $c.n_i$ to refer to the i-th *record field* equal to a_i.
Instead of the explicit definition of $\mathbb{A}$ as

$$\mathbb{A} \overset{def}{\equiv} \mathbb{A}_1 \times \mathbb{A}_2 \times \ldots \times \mathbb{A}_k \quad \text{with} \quad c = (c.n_1, c.n_2, \ldots, c.n_k)$$

we intend to use the shorter form

$$\mathbb{A} \overset{def}{\equiv} (n_1 \in \mathbb{A}_1, n_2 \in \mathbb{A}_2, \ldots, n_k \in \mathbb{A}_k)$$

Definition 1.2 (Record Update). For updating a field $c.n_i$ of the record $c \in \mathbb{A}$ from Definition 1.1 with a new value $v \in \mathbb{A}_i$ we introduce the notation $c[n_i := v]$ such that

$$c[n_i := v] \overset{def}{\equiv} c'$$

where

$$\forall j \leq k.\ c'.n_j = \begin{cases} v & : \quad j = i \\ c.n_j & : \quad \text{otherwise} \end{cases}$$

Functions

In the work along with total functions $f : \mathbb{X} \to \mathbb{Y}$, we operate with partial functions $g : \mathbb{X} \rightharpoonup \mathbb{Y}$. Obviously, the type of such a function g can be also defined as total mapping $g : \mathbb{X}' \to \mathbb{Y}$ for some $\mathbb{X}' \subset \mathbb{X}$. To denote the domain of a given function we use the standard mathematical notation $\operatorname{dom}(f) = \mathbb{X}$ and $\operatorname{dom}(g) = \mathbb{X}'$.

Definition 1.3 (Function Domain Restriction). For a given function $f : \mathbb{X} \to \mathbb{Y}$ or $f : \mathbb{X} \rightharpoonup \mathbb{Y}$, and a set $X \subseteq \operatorname{dom}(f)$ we can obtain the *restriction* $f|_X : X \to \mathbb{Y}$ of the function f so that for all $x \in X$ we have

$$f|_X(x) \overset{def}{\equiv} f(x)$$

Definition 1.4 (Function Update). For any such function f we can redefine (or *update*) its mapping from a given element $a \in \operatorname{dom}(f)$ into a new value $v \in \mathbb{Y}$ with the help of the notation $f[a \mapsto v]$ such that for any $x \in \operatorname{dom}(f)$ one obtains

$$f[a \mapsto v](x) \overset{def}{\equiv} \begin{cases} v & : \quad x = a \\ f(x) & : \quad \text{otherwise} \end{cases}$$

Definition 1.5 (Union of Functions). For any given functions $f : \mathbb{X} \to \mathbb{Y}$ and $g : \mathbb{U} \to \mathbb{V}$ with disjoint domains $\mathbb{X} \cap \mathbb{U} = \emptyset$ we define

$$(f \uplus g) : \mathbb{X} \cup \mathbb{U} \to \mathbb{Y} \cup \mathbb{V}$$

such that for all $x \in \mathbb{X} \cup \mathbb{U}$ we have

$$(f \uplus g)(x) \overset{def}{\equiv} \begin{cases} f(x) & : \quad x \in \mathbb{X} \\ g(x) & : \quad \text{otherwise} \end{cases}$$

Note, that the same notation can be easily applied to partial functions if we know their domains.

Sometimes in the work, in order to match formal definitions we will need to transform partial functions into total ones.

Definition 1.6 (Transformation of Partial Function into Total). We can transform a partial function $f : \mathbb{X} \rightharpoonup \mathbb{Y}$ into a total one $\lceil f \rceil : \mathbb{X} \to \mathbb{Y}$ in such a way that for any $x \in \mathbb{X}$ one computes

$$\lceil f \rceil(x) \overset{def}{\equiv} \begin{cases} f(x) & : \quad x \in \operatorname{dom}(f) \\ \epsilon\mathbb{Y} & : \quad \text{otherwise} \end{cases}$$

Analogously for total mappings $f : \mathbb{X}' \to \mathbb{Y}$ such that one has $\mathbb{X}' \subset \mathbb{X}$ we can apply

$$\lceil f \rceil_{\mathbb{X}} \overset{def}{\equiv} \lceil f \rceil$$

Obviously, we will use such transformations only for cases where we are not interested in the added dummy function values.

Sequences and Bit-Strings

Definition 1.7 (Finite Sequences). A *finite sequence* (or *string*) a of many $n \in \mathbb{N}$ elements $a[i]$ or a_i from a set $\mathbb{A}$ is represented as

$$a = (a_1, \ldots, a_n) = a[1:n]$$

and formalized as a mapping from the indices of the elements to the elements

$$a : \mathbb{N}_n \to \mathbb{A}$$

The set $\mathbb{A}^n$ of sequences of length n with elements from $\mathbb{A}$ is then defined as

$$\mathbb{A}^n \overset{def}{\equiv} \{a \mid a : \mathbb{N}_n \to \mathbb{A}\}$$

Definition 1.8 (Concatenation of Sequences). For two finite sequences $a \in \mathbb{A}^n$, $b \in \mathbb{A}^m$ with $n, m \in \mathbb{N}$, the *concatenation* $a \circ b$ results in a sequence of length $n + m$ and is defined as

$$a[1:n] \circ b[1:m] \overset{def}{\equiv} c[1:n+m]$$

such that for all $i \in \mathbb{N}_{n+m}$

$$c[i] = \begin{cases} a[i] & : \quad i \leq n \\ b[i-n] & : \quad \text{otherwise} \end{cases}$$

Definition 1.9 (Empty Sequence). The *empty sequence* ε is a unique sequence of length 0 and is the only element of the set

$$\mathbb{A}^0 \overset{def}{\equiv} \{\varepsilon\}$$

The concatenation of the empty sequence with a sequence $a \in \mathbb{A}^n$ for any $n \in \mathbb{N}_0$ satisfies

$$a \circ \varepsilon = \varepsilon \circ a = a$$

Definition 1.10 (Sets of Finite Sequences). The set $\mathbb{A}^+$ of *non-empty finite sequences* of elements from a set $\mathbb{A}$ is defined as

$$\mathbb{A}^+ \overset{def}{\equiv} \bigcup_{n \in \mathbb{N}} \mathbb{A}^n$$

In turn, the set $\mathbb{A}^*$ of *all finite sequences* of elements from $\mathbb{A}$ is

$$\mathbb{A}^* \overset{def}{\equiv} \mathbb{A}^+ \cup \{\varepsilon\}$$

To denote the length of a sequence $a \in \mathbb{A}^*$ we use the standard notation $|a|$. Moreover, in order to indicate that an element $x \in \mathbb{A}$ belongs to the given sequence a we overload the set membership symbol:

$$x \in a \overset{def}{\equiv} a \neq \varepsilon \wedge \exists i \in \mathbb{N}_{|a|}.\, a[i] = x$$

Definition 1.11 (Subsequences). For a finite sequence $a \in \mathbb{A}^n$, and indices $i, j \in \mathbb{N}_n$, $i \leq j$, we form the subsequnce $a[i:j]$ in the following way:

$$a[i:j] \overset{def}{\equiv} c[1:j-i+1] \quad \text{with} \quad c[k] = a[i+k-1] \quad \text{for all} \quad k \in \mathbb{N}_{j-i+1}$$

Additionally, for indices $i > j$ we simply set $a[i:j] \overset{def}{\equiv} \varepsilon$.

Definition 1.12 (Reverse of a Finite Sequence). In order to reverse a finite sequence $xs \in \mathbb{X}^*$ we use the function $rev(xs) \in \mathbb{X}^*$ computing the result as

$$rev(xs) \overset{def}{\equiv} \begin{cases} x \circ rev(xs') & : \ xs = xs' \circ x \\ \varepsilon & : \ xs = \varepsilon \end{cases}$$

Definition 1.13 (Application of a Function to Elements of a Finite Sequence). Additionally, we introduce the function $map : (\mathbb{X} \to \mathbb{Y}) \times \mathbb{X}^n \to \mathbb{Y}^n$ that applies a given function $f : \mathbb{X} \to \mathbb{Y}$ to each element of a finite sequence $xs \in \mathbb{X}^n$ of length $n \in N$ such that every element with index $i \in \mathbb{N}_n$ of the resulting sequence is computed as

$$map(f, xs)[i] \overset{def}{\equiv} f(xs[i])$$

Along with the finite sequences with elements numbered from left to right and staring from 1 we will also use sequences of bits where the elements have a different numbering used in number representations.

Definition 1.14 (Bit-Strings). A *bit-string* (or *bit-vector*) $a : [0 : n-1] \to \mathbb{B}$ containing elements a_i or $a[i]$ from the set $\mathbb{B}$ has the format

$$a = (a_{n-1}, \ldots, a_0) = a[n-1:0]$$

Moreover, by $\mathbb{B}^n$ we denote the set of all bit-strings of length $n \in \mathbb{N}$:

$$\mathbb{B}^n \overset{def}{\equiv} \{a \mid a : [0 : n-1] \to \mathbb{B}\}$$

Due to the different representation, additionally, we need to overload the notations of the concatenation and the subsequences for the bit-strings.

Definition 1.15 (Concatenation of Bit-Strings). For two bit-strings $a \in \mathbb{B}^n$, $b \in \mathbb{B}^m$ with $n, m \in \mathbb{N}$, the *concatenation* $a \circ b$ is defined as

$$a[n-1:0] \circ b[m-1:0] \overset{def}{\equiv} c[n+m-1:0]$$

such that for all $i \in [0 : n+m-1]$

$$c[i] = \begin{cases} b[i] & : \ i \leq m-1 \\ a[i-m] & : \ \text{otherwise} \end{cases}$$

Definition 1.16 (Bit-Substrings). For given $n \in \mathbb{N}$, $i, j \in [0 : n-1]$, $j \geq i$, and a bit string $a \in \mathbb{B}^n$, we define a *bit substring* $a[j : i]$ of a as

$$a[j:i] \overset{def}{\equiv} c[j-i:0] \quad \text{with} \quad c[k] = a[i+k] \quad \text{for all} \quad k \in [0 : j-i]$$

In the work we will deal with 32-bit processor architecture and refer to a 32-bit string $a \in \mathbb{B}^{32}$ as a *word*. A bit sting of length 8 is called a *byte*. Consequentially, every word consists of 4 bytes.

Definition 1.17 (Byte Extraction). Let $n = 8 \cdot k$ be a multiple of $k \in \mathbb{N}$ and $a \in \mathbb{B}^n$ is a bit-string consisting of k bytes. For $i \in [0 : k-1]$ we define the byte i of the string a as

$$byte(i, a) \overset{def}{\equiv} a[8 \cdot (i+1) - 1 : 8 \cdot i]$$

Definition 1.18 (Repeating Bits). For a bit $x \in \mathbb{B}$ and a number $n \in \mathbb{N}$ we denote a bit-string containing n copies of x by x^n. Formally, it is defined inductively as

$$x^1 \overset{def}{\equiv} x$$
$$x^{n+1} \overset{def}{\equiv} x \circ x^n$$

Note that for brevity in this work we allow to omit the symbol $\circ$ for the concatenation of bit-strings and finite sequences, e.g., $x \circ x^n = xx^n$.

Sometimes, it is more convenient to consider strings of bytes instead of bit-strings.

Definition 1.19 (Byte-Strings). A *byte-string* $a : [0 : n-1] \to \mathbb{B}^8$ has the same format as a bit-string (indexing from right to left), but contains elements a_i or $a[i]$ from the set $\mathbb{B}^8$. Naturally, by $(\mathbb{B}^8)^n$ we denote the set of all byte-strings of length $n \in \mathbb{N}$:

$$(\mathbb{B}^8)^n \overset{def}{\equiv} \{a \mid a : [0 : n-1] \to \mathbb{B}^8\}$$

Obviously, the definitions for the set $(\mathbb{B}^8)^*$, concatenation of byte-strings, and byte-substrings are similar to the ones stated above, and we do not repeat them here for brevity. Additionally, we introduce two functions converting bit-strings into byte-strings and vice versa.

Definition 1.20 (Bit-Strings and Byte-Strings Conversion). For $k \in \mathbb{N}$, a bit-string $x \in \mathbb{B}^{8 \cdot k}$, and a byte string $y \in (\mathbb{B}^8)^k$, the functions

$$\mathit{bits2bytes}(x) \overset{def}{\equiv} y \quad \text{and} \quad \mathit{bytes2bits}(y) \overset{def}{\equiv} x$$

perform the conversion between bit-strings and byte-strings in a way such that for all indices $i \in [0 : k-1]$ and $j \in [0 : 7]$ we have

$$y[i][j] = x[8 \cdot i + j]$$

Binary Arithmetic

Definition 1.21 (Binary Numbers). For bit-strings $a \in \mathbb{B}^n$ of length $n \in \mathbb{N}$ we denote by

$$\langle a \rangle \overset{def}{\equiv} \sum_{i=0}^{n-1} a_i \cdot 2^i$$

the interpretation of a as a *binary number*. We call a the *binary representation* of length n of the natural number $\langle a \rangle \in \mathbb{N}_0$. In turn, the set of numbers with binary representation of length n is defined as

$$B_n \overset{def}{\equiv} \{\langle a \rangle \mid a \in \mathbb{B}^n\}$$

Obviously, it is easy to show (see [KMP14]) that we have

$$B_n = [0 : 2^n - 1]$$

Definition 1.22 (Binary Representation of a Number). For a number $x \in B_n$ we obtain its binary representation $\mathit{bin}_n(x)$ of length $n \in \mathbb{N}$ as

$$\mathit{bin}_n(x) \overset{def}{\equiv} \epsilon\{a \mid a \in \mathbb{B}^n \wedge \langle a \rangle = x\}$$

For a shorter notation we tend to use x_n instead of $\mathit{bin}_n(x)$:

$$x_n \overset{def}{\equiv} \mathit{bin}_n(x)$$

Definition 1.23 (Binary Addition and Subtraction). The operations of the *binary addition* $+_n$ and *subtraction* $-_n$ on two $a, b \in \mathbb{B}^n$ are defined as follows:

$$a +_n b \overset{def}{\equiv} bin_n(\langle a \rangle + \langle b \rangle \bmod 2^n)$$
$$a -_n b \overset{def}{\equiv} bin_n(\langle a \rangle - \langle b \rangle \bmod 2^n)$$

Definition 1.24 (Set of Words Starting from a Given One). For $d \in \mathbb{N}$ words starting from a word $a \in \mathbb{B}^{32}$ we easily define the set

$$\{a\}_d \overset{def}{\equiv} \{a +_{32} i_{32} \mid i \in [0 : d-1]\}$$

Definition 1.25 (Two's Complement Numbers). For bit-strings $a \in \mathbb{B}^n$ of length $n \in \mathbb{N}$ we denote by

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

the interpretation of a as a *two's complement number* and refer to a as the *two's complement representation* of $[a]$. The set of integer numbers with two's complement representation of length n is denoted as

$$T_n \overset{def}{\equiv} \{[a] \mid a \in \mathbb{B}^n\}$$

Obviously, it is equal to the following integer interval:

$$T_n = \left[-2^{n-1} : 2^{n-1} - 1\right]$$

One can easily notice $[a] < 0 \iff a_{n-1}$. Therefore, a_{n-1} is called a *sign bit*.

Definition 1.26 (Two's Complement Representation of a Number). The two's complement representation of an integer number $x \in T_n$ is obtained as

$$twoc_n(x) \overset{def}{\equiv} \epsilon\{a \mid a \in \mathbb{B}^n \wedge [a] = x\}$$

Definition 1.27 (Zero- and Sign-Extensions of Bit-Strings). For $a \in \mathbb{B}^n$, and $n, k \in \mathbb{B}^n$ such that $k > n$, we define *zero-extended* and *sign-extended* bit-strings $zxt_k(a)$ and $sxt_k(a)$ respectively as follows:

$$zxt_k(a) \overset{def}{\equiv} 0^{k-n}a$$
$$sxt_k(a) \overset{def}{\equiv} a_{n-1}^{k-n}a$$

Definition 1.28 (Bit-Operations on Bit-Strings). For bit-string $a, b \in \mathbb{B}^n$ of length $n \in \mathbb{N}$, a bit $c \in \mathbb{B}$, and operations $\bullet \in \{\wedge, \vee, \oplus\}$ we define the corresponding operations on bit-strings as follows:

$$a \bullet_n b \overset{def}{\equiv} (a_{n-1} \bullet b_{n-1}, \ldots, a_0 \bullet b_0)$$
$$c \bullet_n b \overset{def}{\equiv} (c \bullet b_{n-1}, \ldots, c \bullet b_0)$$
$$\neg_n a \overset{def}{\equiv} (\neg a_{n-1}, \ldots, \neg a_0)$$

Computations

All models considered in this work are represented as *deterministic automata* which transition relations δ are described by *transition functions*

$$\delta : \mathbb{C} \times \Sigma \rightharpoonup \mathbb{C}$$

where $\mathbb{C}$ is a set of *states* (or *configurations*), and Σ is a set of *inputs* also called *input alphabet*.

This means that for modeling non-deterministic systems we formalize automata in such a way that their transition relations are made deterministic by resolving any non-deterministic choice with the help of the input alphabet.

Definition 1.29 (**Multiple Transitions**). Given sequences of configurations $c \in \mathbb{C}^{n+1}$ and inputs $in \in \Sigma^n$ with $n \in \mathbb{N}_0$, a number of steps $k \leq n$, the notation $c_1 \longrightarrow^k_{\delta,in} c_{k+1}$ denotes that $c[1 : k+1]$ is obtained by applying the transition function δ for k steps with corresponding inputs from in. Formally, we define this as

$$c_1 \longrightarrow^k_{\delta,in} c_{k+1} \overset{def}{\equiv} \begin{cases} c_1 \longrightarrow^{k-1}_{\delta,in} c_k \wedge c_{k+1} = \delta(c_k, in_k) & : \quad k > 0 \\ 1 & : \quad k = 0 \end{cases}$$

Generally, for two configurations $s, s' \in \mathbb{C}$, inputs $in \in \Sigma^n$ we denote that s' is reached in $n \in \mathbb{N}_0$ steps from s in the following way:

$$s \longrightarrow^n_{\delta,in} s' \overset{def}{\equiv} \exists a \in \mathbb{C}^{n+1}.\ a_1 \longrightarrow^n_{\delta,in} a_{n+1} \wedge s = a_1 \wedge s' = a_{n+1}$$

As special cases, one can also define the same notation for a one step computation with an input $in \in \Sigma$

$$s \longrightarrow_{\delta,in} s' \overset{def}{\equiv} s \longrightarrow^1_{\delta,in} s'$$

and arbitrarily long computation with inputs $in \in \Sigma^*$

$$s \longrightarrow^*_{\delta,in} s' \overset{def}{\equiv} s \longrightarrow^{|in|}_{\delta,in} s'$$

2 Model for Concurrent Systems and Simulation

The pervasive verification approach of complex computer systems requires building stacks of semantics from the hardware layer up to semantics of abstract models which implementations rely on the correctness of all underlying levels including assemblers, compilers, etc. In such a stack adjacent levels are connected by simulation theorems allowing to discover software conditions under which the simulation of the top most abstract model on the lowest implementation layer can be shown.

However, for models proven to be correct only for their sequential execution we often cannot directly use their correctness in the concurrent setting. As an example, one could consider a compiler of a high-level programming language verified for the sequential execution of a compiled code. Therefore, when this code runs on a multi-core machine, where the processors steps are interleaved arbitrary, one has to justify the concurrent semantics of the same language with the atomic execution of its statements.

For this justification and establishing concurrent semantics stacks Christoph Baumann in his doctoral thesis [Bau14b] based on the earlier developed theory in [LS89, CL98] suggested to employ sequential simulations theorems for blocks of consecutive processor steps. Such blocks starting in configurations where the simulation can hold (so called *consistency points*) are derived via reordering from arbitrarily interleaved transitions under a condition that an applied programming discipline guarantees the absence of memory races. Moreover, the step permutation must preserve the order of shared memory accesses (I/O-steps) as well as local steps for all units present in a model (see Figure 2.1 as an example).

Since in our work we intensively use Baumann's theory of order reduction and concurrent simulation, this chapter is fully devoted to its details. Though most of the definitions, lemmas and theorems are extracted from [Bau14b], we also introduce a few extensions and corrections that were not taken into account in the original work and which will allow us to apply the theory for our topic. Therefore, for additional or substantially modified claims requiring more argumentation we provide our detailed proofs here.

2.1 Abstract Model with Memory and Ownership

In order to model concurrent systems we introduce a generic model including a fixed number of execution units accessing a shared memory. According to [Bau14b] the model got a name *Cosmos* as an acronym for a **Co**ncurrent system with **s**hared **m**emory and **o**wner**s**hip. The *Cosmos* model is based on memory and unit configurations with their transition functions, additional predicates characterizing unit steps, and a ghost ownership state allowing to abstract a programmer's policy guaranteeing safe memory accesses.

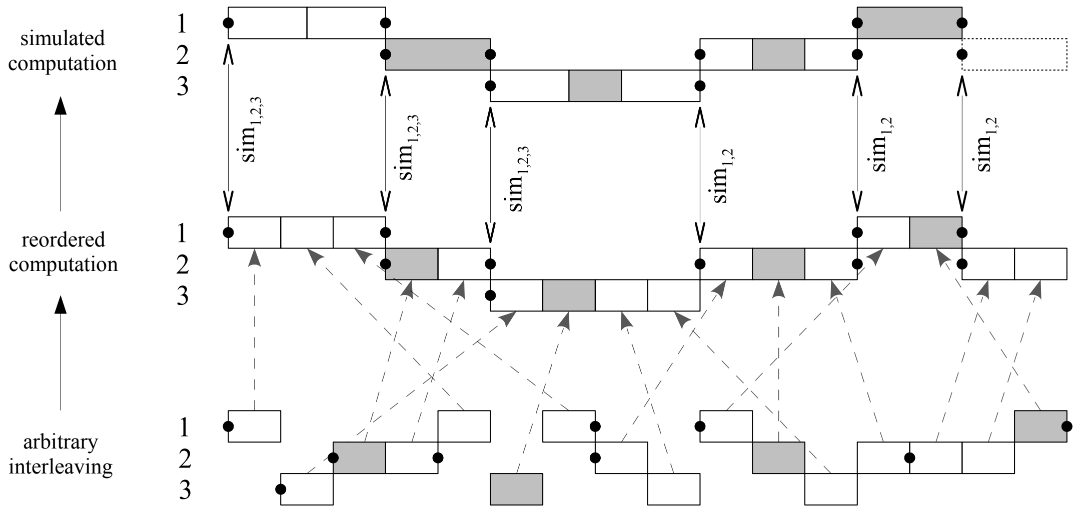

Figure 2.1: Application of order reduction and simulation for a concurrent model with three units. Grey boxes depict I/O-steps whereas empty ones are local. One direction arrows show the permutation of steps for reordering. The reordered interleaving is performed at units' consistency points marked as black dots where the simulation relation sim holds for units present in the subscript. Note, that Unit 3 has not reached its consistency point and cannot be covered by sim, though the changes of the shared memory must be reflected on the abstract level. Oppositely, the steps of Unit 2 have changed only its local configuration and are not considered in the abstraction yet (dotted box).

2.1.1 Signature and Instantiation Parameters

The *Cosmos* model is characterized by a set of instantiable parameters (types and functions) which we call a *Cosmos model signature*.

Definition 2.1 (**Cosmos Model Signature**). A *Cosmos* model $S \in \mathbb{S}$ is given by a tuple

$$\mathbb{S} \overset{def}{\equiv} (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IP}, \mathcal{IO}, \mathcal{OT})$$

with the following named components:

- $\mathcal{A}, \mathcal{V}$ – sets of memory addresses and memory values. In the underlying machine instantiating the *Cosmos* model one usually considers the memory as a mapping $m : \mathcal{A} \to \mathcal{V}$.
- $\mathcal{R} \subseteq \mathcal{A}$ – a fixed set of read-only addresses, that could include, for instance, a non-modified memory region of a compiled code, constants, etc.
- nu – a number of computation units in the machine.
- $\mathcal{U}$ – a set of computation unit states defined in the underlying semantics. For simplicity we assume that all units are instantiated with the same automation.
- $\mathcal{E}$ – a set of external inputs for the units. Though the units communicate via the shared memory, the external inputs are often used to model the communication with an external environment as well as to resolve unit's non-deterministic behaviour.

- $reads : \mathcal{U} \times (\mathcal{A} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow 2^{\mathcal{A}}$ – a set of memory addresses read during a step for given unit and memory configurations, and an external input. In the work we call it a $reads$-set.
- $\delta : \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V}) \times \mathcal{E} \rightharpoonup \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V})$ – a unit transition function taking as inputs a unit state, a partial memory, and external inputs. The result of the defined transition is a unit configuration and a part of the memory after the step.

  The partial memory in the function inputs is restricted to the $reads$-set including all memory addresses needed for the step. The output partial memory represents the updated portion of the memory configuration. In contrast to [Bau14b], the transition function is supposed to be a partial mapping, what is more typical for many semantics and disambiguates the further mathematical formalism in the work.
- $\mathcal{IP} : \mathcal{U} \times (\mathcal{R} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow \mathbb{B}$ – a predicate specifying desired interleaving points ($\mathcal{IP}$-points) on the base of a unit configuration, an external input, and the read-only memory. When the given unit is in the interleaving point and makes a step we allow computations performed by other units to appear (or interleave) before it.
- $\mathcal{IO} : \mathcal{U} \times (\mathcal{R} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow \mathbb{B}$ – a predicate denoting whether a unit step from a given configuration and determined by an external input is suitable for an $\mathcal{IO}$-operation. The $\mathcal{IO}$-operations are used for interactions with the environment and other units and could be, e.g., atomic accesses to the shared portion of memory. The configuration before such a unit step is called an $\mathcal{IO}$-point.
- $\mathcal{OT} : \mathcal{U} \times (\mathcal{R} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow \mathbb{B}$ – indicates whether a given $\mathcal{IO}$-point can be used for the ownership transfer.

The predicate $\mathcal{OT}$ was not introduced in the theory developed by Baumann. However, the application of the theory for different models showed that such a flag allows to restrict the ownership policy for cases where the ownership transfer is undesirable (e.g. non-deterministic processor component steps that cannot be influenced by the programmer; guest steps with its unknown for the hypevisor memory access policy, etc.) and makes the model cleaner without introducing additional restrictions on the safety policy and ownership annotation later on.

In the work we also adopt all definitions with respect to $\mathcal{OT}$. Since Baumann considered the order reduction theory with the ownership transfer at any $\mathcal{IO}$-points, it can be treated as a more general case. The presence of our additional flag does not influence substantially any proofs of the order reduction in [Bau14b] and requires only simple additional bookkeeping not changing the way of argumentation. Therefore, for such theorems and lemmas from [Bau14b] we do not repeat their proofs in this thesis.

For any *Cosmos* model $S \in \mathbb{S}$ we will use in the work the shorthands $\mathcal{A}, \mathcal{V}, \mathcal{R}$, etc. to refer to the corresponding components of its signature when it is clear from the context which machine we deal with.

For a meaningful model instantiation one has to guarantee that the $reads$-set includes all relevant addresses to determine which addresses are read in one unit step.

Definition 2.2 (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, $m, m' \in (\mathcal{A} \rightarrow \mathcal{V})$ memory configurations, and $in \in \mathcal{E}$ be an input for a step of the unit. If the memory contents agree on $reads$-set $R = S.reads(u, m, in)$, then also the $reads$-set wrt. m' agrees with R.

$$insta_r(S) \overset{def}{\equiv} (\, m'|_R = m|_R \implies S.reads(u, m', in) = R\,)$$

Moreover, as states above, not all $\mathcal{IO}$-points can be used for the ownership transfer. The flag $\mathcal{OT}$ must be instantiated only for a subset of $\mathcal{IO}$-operations.

Definition 2.3 (**Instantiation Restriction for $\mathcal{OT}$**). For a *Cosmos* machine S let $u \in \mathcal{U}$ be a computation unit state, $m \in (\mathcal{R} \to \mathcal{V})$ read-only memory, and $in \in \mathcal{E}$ be an input for a step of the unit. Then we require that any $\mathcal{OT}$ is also an $\mathcal{IO}$-point.

$$insta_{ot}(S) \overset{def}{\equiv} (\mathcal{OT}(u, m, in) \implies \mathcal{IO}(u, m, in))$$

Note, that in the thesis we will provide instantiations of the *Cosmos* model with different levels of semantics in the verification stack. However, we will rely on the correctness of the instantiations without proving the aforementioned restriction properties. The detailed proof of the property $insta_r$ for some of the models used in this work can be found in [Bau14b].

2.1.2 Cosmos Machine Configuration and Semantics

Now we define a *Cosmos machine configuration* consisting of a *concurrent machine state* and an *ownership state*. The concurrent machine state represents a machine instantiating the *Cosmos* model in the concurrent setting. The ownership state is a *ghost* state which does not influence the underlying semantics of the concurrent model and is used for modelling the memory access policy for the further justification of the model with a few execution units.

Definition 2.4 (**Concurrent Machine State**). The *concurrent machine state* for a *Cosmos* model $S \in \mathbb{S}$ is a pair

$$\mathbb{M}_S \overset{def}{\equiv} (u : \mathbb{N}_{nu} \to \mathcal{U}, m : \mathcal{A} \to \mathcal{V}),$$

where u maps unit indices to their configurations and m is the state of the memory.

Definition 2.5 (**Ownership State**). The *ownership (ghost) state* for a *Cosmos* model $S \in \mathbb{S}$ is a pair

$$\mathbb{G}_S \overset{def}{\equiv} \left(\mathcal{O} : \mathbb{N}_{nu} \to 2^{\mathcal{A}}, \mathcal{S} \in 2^{\mathcal{A}}\right),$$

where $\mathcal{O}$ maps unit indices to the corresponding units' sets of owned addresses (own-sets) and $\mathcal{S}$ is a set of shared writable addresses.

Definition 2.6 (***Cosmos*** **Machine Configuration**). A *Cosmos machine configuration* for a *Cosmos* model $S \in \mathbb{S}$ is given as a pair

$$\mathbb{C}_S \overset{def}{\equiv} (M \in \mathbb{M}_S, \mathcal{G} \in \mathbb{G}_S)$$

For a configuration $C \in \mathbb{C}_S$ of a *Cosmos* model $S \in \mathbb{S}$ and a unit index $p \in \mathbb{N}_{nu}$ we use the following shorthands:

$$\begin{array}{rcl} C.u_p & \equiv & C.M.u(p) \qquad C.m \;\equiv\; C.M.m \\ C.\mathcal{O}_p & \equiv & C.\mathcal{G}.\mathcal{O}(p) \qquad C.\mathcal{S} \;\equiv\; C.\mathcal{G}.\mathcal{S} \\ reads_p(C, in) & \equiv & reads_p(C.M, in) \;\equiv\; reads(C.M.u(p), C.M.m, in) \\ \mathcal{IO}_p(C, in) & \equiv & \mathcal{IO}_p(C.M, in) \;\equiv\; \mathcal{IO}(C.M.u(p), C.M.m|_{\mathcal{R}}, in) \\ \mathcal{OT}_p(C, in) & \equiv & \mathcal{OT}_p(C.M, in) \;\equiv\; \mathcal{OT}(C.M.u(p), C.M.m|_{\mathcal{R}}, in) \\ \mathcal{IP}_p(C, in) & \equiv & \mathcal{IP}_p(C.M, in) \;\equiv\; \mathcal{IP}(C.M.u(p), C.M.m|_{\mathcal{R}}, in) \end{array}$$

Definition 2.7 (Writes-set for a Machine Step). For a given *Cosmos* model S with a machine configuration $M \in \mathbb{M}_S$ and an input $in \in \mathcal{E}$ we can determine the set of written addresses in the corresponding step of a unit p. This so-called *writes-set* is obtained with the following function.

$$writes_p(M, in) \overset{def}{\equiv} \text{dom}\,(m')\,, \quad \text{where} \quad (u', m') = \delta\left(M.u(p), M.m|_{reads_p(M,in)}, in\right)$$

For a whole *Cosmos* model configuration $C \in \mathbb{C}_S$ we reload the shorthand

$$writes_p(C, in) \equiv writes_p(C.M, in)$$

The transition of the *Cosmos* machine involves transitions on the concurrent machine and the ownership state. Before we define the overall *Cosmos* machine transition function we provide separate semantics of its configuration components.

The *Cosmos* machine units execution is performed by their transition function provided by the model instantiation. A scheduling input shows which unit makes a step.

Definition 2.8 (Concurrent Machine Transition Function). A transition step of the machine for a *Cosmos* model $S \in \mathbb{S}$ is defined by the transition function

$$\Delta_t : \mathbb{M}_S \times \mathbb{N}_{nu} \times \mathcal{E} \rightharpoonup \mathbb{M}_S, \quad s.t. \quad \Delta_t(M, p, in) \overset{def}{\equiv} M'$$

which takes a configuration M, a scheduling input p, an external input in, performs a step of the unit p on its state and the memory by the instantiated δ of the *Cosmos* model, and produces the result configuration

$$M' = \begin{cases} (M.u[p \mapsto u'], m_{unchanged} \uplus m') & : \quad c' = (u', m') \\ undefined & : \quad \text{otherwise} \end{cases}$$

where $c' = \delta\left(M.u(p), M.m|_{reads_p(M,in)}, in\right)$ and $m_{unchanged} = M.m|_{\mathcal{A}\setminus \text{dom}(m')}$.

A few additional inputs serve the ownership changes and are provided by the verification engineer annotating the system execution (e.g. based on a program code).

Definition 2.9 (Ownership Transfer Information). For a transition step of a *Cosmos* machine with a signature $S \in \mathbb{S}$ we define the ownership transfer information

$$\Omega_S \overset{def}{\equiv} \left(Acq \in 2^{\mathcal{A}}, Loc \in 2^{\mathcal{A}}, Rel \in 2^{\mathcal{A}}\right)$$

that includes sets of acquired addresses Acq, acquired local addresses Loc (which should be a subset of Acq), and released addresses Rel.

Then the ownership transfer function for the ownership component of the *Cosmos* model can be defined as follows.

Definition 2.10 (Ownership Transfer Function). A transition step of the ownership component of $S \in \mathbb{S}$ is defined by a function

$$\Delta_o : \mathbb{G}_S \times \mathbb{N}_{nu} \times \Omega_S \to \mathbb{G}_S,$$
$$\Delta_o(\mathcal{G}, p, o) \overset{def}{\equiv} (\mathcal{G}.\mathcal{O}[p \mapsto \mathcal{O}'], \mathcal{S}')\,,$$

where $\mathcal{O}' = (\mathcal{G}.\mathcal{O}(p) \setminus o.Rel) \cup o.Acq$ and $\mathcal{S}' = (\mathcal{G}.\mathcal{S} \setminus o.Loc) \cup o.Rel$

Therefore, the *Cosmos* machine semantics can now be defined on the transitions of the separate machine components.

Definition 2.11 (*Cosmos* Machine Transition Function). For a *Cosmos* model $S \in \mathbb{S}$ we define the transition function

$$\Delta : \mathbb{C}_S \times \mathbb{N}_{nu} \times \mathcal{E} \times \Omega_S \rightharpoonup \mathbb{C}_S,$$

$$\Delta\,(C, p, in, o) \overset{def}{\equiv} \begin{cases} (M', \Delta_o(C.\mathcal{G}, p, o)) & : \quad \Delta_t(C.M, p, in) = M' \\ undefined & : \quad \text{otherwise} \end{cases}$$

2.1.3 Computations and Step Sequence Notation

Recall that we split the *Cosmos* machine configuration into the concurrent machine state and the ghost ownership state. To distinguish transitions of both components we introduced the corresponding transition functions. To argue about reordering of transitions we rely on the execution steps of the concurrent machine (as well as the whole *Cosmos* machine) from a certain alphabet rather than on the resulting system states. In our case the alphabet contains transition information and ownership transfer information. The latter was already defines as Ω_S, and we now switch to the step information needed for the concurrent machine transitions as well as a combination of both to denote the *Cosmos* machine steps.

Definition 2.12 (Concurrent Machine Step Information). The set Θ_S of step information describes a concurrent machine step of *Cosmos* model $S \in \mathbb{S}$

$$\Theta_S \overset{def}{\equiv} (s \in \mathbb{N}_{nu}, in \in \mathcal{E}, io \in \mathbb{B}, ot \in \mathbb{B}, ip \in \mathbb{B})$$

by the scheduling parameter s, the external input in for the step, and the flags io, ot, and ip characterizing the step as an $\mathcal{IO}$-point with the possible ownership transfer if ot is set, and as an interleaving point of the reordered computation correspondingly.

A sequence $\theta \in \Theta_S^*$ of machine steps we call a *machine step (or transition) sequence*. Analogously, $o \in \Omega_S^*$ is an *ownership transfer sequence*.

Definition 2.13 (*Cosmos* Machine Step Information). A step of the *Cosmos* machine contains the concurrent machine step information and the ownership transfer information:

$$\Sigma_S \overset{def}{\equiv} (t \in \Theta_S, o \in \Omega_S)$$

For any step information $\alpha \in \Sigma_S$ we allow to use shorthands such that for $x \in \{s, in, io, ot, ip\}$ and $y \in \{Acq, Loc, Rel\}$ one can write $\alpha.x \equiv \alpha.t.x$ and $\alpha.y \equiv \alpha.o.y$. A sequence $\sigma \in \Sigma_S^*$ is a *Cosmos* machine *step sequence*. We also introduce shorthands mapping sequences of step information σ to transition and ownership transfer sequences.

$$\sigma.t \equiv \sigma_1.t \cdots \sigma_{|\sigma|}.t \qquad \sigma.o \equiv \sigma_1.o \cdots \sigma_{|\sigma|}.o$$

Definition 2.14 (Step Notation). The notation $M \overset{t}{\mapsto} M'$ denotes that the transition $t \in \Theta_S$ is executed from the machine state $M \in \mathbb{M}_S$ and results in $M' \in \mathbb{M}_S$. Additionally $t.io$, $t.ot$, and $t.ip$ correspond to the values of the predicates $\mathcal{IO}$ $\mathcal{OT}$, and $\mathcal{IP}$ respectively.

$$\begin{aligned} M \overset{t}{\mapsto} M' \quad \overset{def}{\equiv} \quad & M' = \Delta_t(M, t.s, t.in) \wedge \mathcal{IP}_{t.s}(M, t.in) = t.ip \wedge \\ & \mathcal{IO}_{t.s}(M, t.in) = t.io \wedge \mathcal{OT}_{t.s}(M, t.in) = t.ot \end{aligned}$$

For steps $\alpha \in \Sigma_S$, which include ownership transfer information, we define a similar notation for the *Cosmos* machine transition from configuration $C \in \mathbb{C}_S$ to $C' \in \mathbb{C}_S$.

$$C \overset{\alpha}{\mapsto} C' \overset{def}{\equiv} C.M \overset{\alpha.t}{\mapsto} C'.M \wedge C'.\mathcal{G} = \Delta_o(C.\mathcal{G}, \alpha.s, \alpha.o)$$

The definitions naturally extend to step sequences by induction. For a sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$, a step $\alpha \in \Sigma_S \cup \Theta_S$, and given configurations $X, X' \in \mathbb{M}_S \cup \mathbb{C}_S$ we have:

$$\begin{aligned} X \overset{\varepsilon}{\longmapsto} X' &\overset{def}{\equiv} X' = X \\ X \overset{\rho\alpha}{\longmapsto} X' &\overset{def}{\equiv} \exists X''.\, X \overset{\rho}{\longmapsto} X'' \wedge X'' \overset{\alpha}{\mapsto} X' \end{aligned}$$

We call a pair $(X, \lambda) \in (\mathbb{C}_S \times \Sigma_S^*) \cup (\mathbb{M}_S \times \Theta_S^*)$ a *Cosmos* machine computation if the following predicate holds:

$$comp(X, \lambda) \overset{def}{\equiv} \exists X' \in \mathbb{C}_S \cup \mathbb{M}_S.\, X \overset{\lambda}{\longmapsto} X'$$

To convert a pair (θ, o) consisting of transition and ownership transfer sequences of the same length $|\theta| = |o|$ into a *Cosmos* machine step sequence σ we use a construct $\langle \theta, o \rangle = \sigma$ such that $\sigma.t = \theta$ and $\sigma.o = o$.

2.1.4 Ownership Policy

To enforce the memory safety guaranteeing the absence of memory races for the concurrent execution of any *Cosmos* machine, we use the memory access policy defined in [Bau14b] and based on the ghost ownership state of the *Cosmos* machine configuration.

Definition 2.15 (Ownership Memory Access Policy). Given a flag $io \in \mathbb{B}$, a $reads$-set R, a $writes$-set W, a set $\mathcal{O}$ of addresses owned by a unit, sets $\mathcal{S}$ and $\mathcal{R}$ of shared and read-only addresses respectively, and addresses $\overline{\mathcal{O}}$ owned by other units, we enforce the following ownership memory access policy given by the following predicate

$$policy_{acc}(io, R, W, \mathcal{O}, \mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}) \overset{def}{\equiv}$$

1. Local steps (i) read only owned or read-only addresses and (ii) write only owned unshared addresses
$$\neg io \implies \begin{array}{ll} \textit{(i)} & R \subseteq \mathcal{O} \cup \mathcal{R} \\ \textit{(ii)} & W \subseteq \mathcal{O} \setminus \mathcal{S} \end{array}$$
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 other units.
$$io \implies \begin{array}{ll} \textit{(i)} & R \subseteq \mathcal{O} \cup \mathcal{S} \cup \mathcal{R} \\ \textit{(ii)} & W \subseteq \mathcal{O} \cup (\mathcal{S} \setminus \overline{\mathcal{O}}) \end{array}$$

Moreover, since we allow the ownership state of the *Cosmos* machine to be changed, we also introduce a safety policy for such transitions.

Definition 2.16 (Ownership Transfer Policy). Given a bit $ot \in \mathbb{B}$, a set of owned addresses $\mathcal{O}$, sets of shared $\mathcal{S}$ and owned by other units $\overline{\mathcal{O}}$ addresses, as well as ownership transfer information $o \in \Omega_S$, we restrict ownership transfer by the predicate

$$policy_{trans}(ot, \mathcal{O}, \mathcal{S}, \overline{\mathcal{O}}, o) \overset{def}{\equiv}$$

1. The ownership-state may not be changed by local steps or $\mathcal{IO}$-steps at which we forbid ownership transfer.

$$\neg ot \quad \Longrightarrow \quad o.Acq = \emptyset \wedge o.Loc = \emptyset \wedge o.Rel = \emptyset$$

2. For $\mathcal{IO}$-steps suitable for ownership transfer, the ownership state is allowed to change as long as the step (i) acquires only addresses which are shared unowned or already owned by the executing unit and (ii) releases only owned addresses. Moreover (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 the same time.

$$\begin{aligned} ot \implies \quad & \textit{(i)} && o.Acq \subseteq \mathcal{S} \setminus \overline{\mathcal{O}} \cup \mathcal{O} \\ & \textit{(ii)} && o.Rel \subseteq \mathcal{O} \\ & \textit{(iii)} && o.Loc \subseteq o.Acq \\ & \textit{(iv)} && o.Acq \cap o.Rel = \emptyset \end{aligned}$$

Note, that in the original work by Christoph Baumann the ownership transfer policy was considered for any $\mathcal{IO}$-point that is in turn enough for the order reduction theorem. However, simulation between two concurrent machines might require a stronger policy with the transfer only at a certain $\mathcal{IO}$-points. Therefore, in our work we are interested only in the safety properties that allow us not only to apply the order reduction but also to argue about the simulation. In fact, since we mark only $\mathcal{IO}$-points to be suitable for the operations on the ownership, one can easily show that our restricted policy does not violate $policy_{trans}(io, \mathcal{O}, \mathcal{S}, \overline{\mathcal{O}}, o)$ from the original theory.

Naturally, one has to restrict how the address space considered for the *Cosmos* model is split into the subsets of addresses wrt. to the ownership state(e.g., different units cannot own the same addresses, etc.). For this purpose we state an invariant that will be preserved if the *Cosmos* machine execution follows the aforementioned policies.

Definition 2.17 (Ownership Invariant). We introduce an *ownership invariant* on an ownership state $\mathcal{G} \in \mathbb{G}_S$ of a *Cosmos* machine which requires (i) the own-sets of different units to be mutually disjoint and (ii) that read-only addresses may not be owned or shared-writable. Moreover (iii) the complete address space consists of all the owned, shared, and read-only addresses.

$$\begin{aligned} oinv(\mathcal{G}) \overset{def}{\equiv} \quad & \textit{(i)} && \forall p, q.\ p \neq q \Longrightarrow \mathcal{G}.\mathcal{O}(p) \cap \mathcal{G}.\mathcal{O}(q) = \emptyset \\ & \textit{(ii)} && \forall p.\ \mathcal{G}.\mathcal{O}(p) \cap \mathcal{R} = \emptyset \ \wedge \ \mathcal{G}.\mathcal{S} \cap \mathcal{R} = \emptyset \\ & \textit{(iii)} && \mathcal{A} = \mathcal{R} \cup \mathcal{G}.\mathcal{S} \cup \bigcup_{p \in \mathbb{N}_{nu}} \mathcal{G}.\mathcal{O}(p) \end{aligned}$$

Moreover we set the shorthand $oinv(C) \equiv oinv(C.\mathcal{G})$ for all $C \in \mathbb{C}_S$.

Now we combine both policies for a step of the *Cosmos* machine and define it inductively for step sequences.

Definition 2.18 (Ownership Safety of a Step). We consider a step of a *Cosmos* model S from configuration $C \in \mathbb{C}_S$ with step information $\alpha \in \Sigma_S$ to be safe with respect to the ownership model when for $R = reads_{\alpha.s}(C, \alpha.in)$, $W = writes_{\alpha.s}(C, \alpha.in)$, and $\overline{\mathcal{O}} = \bigcup_{q \neq \alpha.s} C.\mathcal{O}_q$ the following predicate is fulfilled

$$\begin{aligned} safe_{step}(C, \alpha) \overset{def}{\equiv} \quad & policy_{acc}(\alpha.io, R, W, C.\mathcal{O}_{\alpha.s}, C.\mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}) \wedge \\ & policy_{trans}(\alpha.ot, C.\mathcal{O}_{\alpha.s}, C.\mathcal{S}, \overline{\mathcal{O}}, \alpha.o) \end{aligned}$$

Definition 2.19 (Ownership Safety of a Step Sequence). For a configuration C of a *Cosmos* model S and $\tau \in \Sigma_S^*, \alpha \in \Sigma_S$ we define

$$\begin{aligned} safe(C, \varepsilon) &\overset{def}{\equiv} oinv(C) \\ safe(C, \tau\alpha) &\overset{def}{\equiv} safe(C, \tau) \wedge \exists C', C''.\ C \xmapsto{\tau} C' \overset{\alpha}{\mapsto} C'' \wedge safe_{step}(C', \alpha) \end{aligned}$$

As we have mentioned before, the ownership safe steps of a *Cosmos* machine from a given configuration preserve the ownership invariant. The proof of this fact was shown in [Bau14b].

Lemma 2.1 (Ownership Safe Steps Preserve the Ownership Invariant). *For* Cosmos *machine configurations $C, C' \in \mathbb{C}_S$ and a step sequence $\sigma \in \Sigma_S^*$ we have*

$$C \xmapsto{\sigma} C' \wedge safe(C, \sigma) \implies oinv(C')$$

2.2 Ownership Based Order Reduction

The introduced ownership and safety policy allow us to guarantee not only race-free concurrent memory accesses by units present in the system, but also to reorder all unit steps after their arbitrary interleaved execution. Baumann [Bau14b] showed in this case that arbitrary schedules can be reduced to so called *interleaving point schedules (or $\mathcal{IP}$-schedules)* preserving the result of computations, memory safety and other properties. So, if we prove such properties for all $\mathcal{IP}$-schedules from a given configuration, they also hold for any arbitrarily interleaved computations starting in the same machine configuration.

First, we define the meaning of $\mathcal{IP}$-schedules.

Definition 2.20 (Interleaving Point Schedule). For a step sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$ we define the predicate

$$\mathcal{IP}sched(\rho) \overset{def}{\equiv} (\rho = \rho'\alpha\beta \implies \mathcal{IP}sched(\rho'\alpha) \wedge ((\alpha.s = \beta.s) \vee \beta.ip))$$

that expresses whether the sequence is an interleaving point schedule ($\mathcal{IP}$-schedule).

In other words, the $\mathcal{IP}$-schedule consists of interleaved blocks of units' consecutive steps starting in the interleaving points except the first one.

The idea behind the order reduction is based on the preservation of units' local steps as well as the order of accesses to shared resources. For this purpose we provide a special notation from [Bau14b] extended also for the $\mathcal{OT}$ predicate introduced in this work.

Definition 2.21 (Step Subsequence Notation). For any step sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$ and a unit index p we define the subsequence of steps of unit p as $\rho|_p$, the $\mathcal{IO}$-step subsequence of ρ as $\rho|_{io}$, and the $\mathcal{OT}$-step subsequence of ρ as $\rho|_{ot}$:

$$\rho|_{io} \overset{def}{\equiv} \begin{cases} \alpha\tau|_{io} & : \rho = \alpha\tau \wedge \alpha.io \\ \tau|_{io} & : \rho = \alpha\tau \wedge \neg\alpha.io \\ \varepsilon & : \text{otherwise} \end{cases} \qquad \rho|_{ot} \overset{def}{\equiv} \begin{cases} \alpha\tau|_{ot} & : \rho = \alpha\tau \wedge \alpha.ot \\ \tau|_{ot} & : \rho = \alpha\tau \wedge \neg\alpha.ot \\ \varepsilon & : \text{otherwise} \end{cases}$$

$$\rho|_{p} \overset{def}{\equiv} \begin{cases} \alpha\tau|_{p} & : \rho = \alpha\tau \wedge \alpha.s = p \\ \tau|_{p} & : \rho = \alpha\tau \wedge \alpha.s \neq p \\ \varepsilon & : \text{otherwise} \end{cases}$$

Now, we can define that two step sequences are equivalently reordered, meaning that in both sequences the order of steps mentioned above is preserved. This definition is crucial for arguing about existence of an equivalent interleaving point schedule for a given step sequence.

Definition 2.22 (Equivalent Reordering Relation). Given two step sequences $\rho, \rho' \in \Sigma_S^* \cup \Theta_S^*$, we consider them equivalently reordered when the $\mathcal{IO}$-step and $\mathcal{OT}$-step subsequences as well as the step subsequences of all units are the same:

$$\rho \mathrel{\hat{=}} \rho' \quad \overset{def}{\equiv} \quad \rho|_{io} = \rho'|_{io} \land \rho|_{ot} = \rho'|_{ot} \land \forall p \in \mathbb{N}_{nu}.\ \rho|_p = \rho'|_p$$

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.

As we have seen so far one can instantiate a *Cosmos* model with a definition of interleaving points independent from steps performing $\mathcal{IO}$-operations. However, to make the reordering of steps feasible we have to require that between two $\mathcal{IO}$-points a unit passes through at least one $\mathcal{IP}$-point and all units of the system start their computations in interleaving points.

Definition 2.23 ($\mathcal{IOIP}$ Condition). For any sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$, the 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.

$$\begin{aligned} \mathcal{IOIP}(\rho) \quad \overset{def}{\equiv} \quad & \forall \pi, p.\ \pi = \rho|_p \neq \varepsilon \implies \\ & \qquad \pi_1.ip \land (\forall \tau, \alpha, \varphi, \beta, \omega.\ \pi = \tau\alpha\varphi\beta\omega \land \alpha.io \land \beta.io \implies \exists i.\ \varphi_i.ip) \end{aligned}$$

According to [Bau14b] such an equivalent reordering does not break the $\mathcal{IOIP}$ condition.

Lemma 2.2 (Equivalent Reordering of $\mathcal{IOIP}$ Sequences). *The $\mathcal{IOIP}$ condition is preserved by equivalent reordering. For sequences $\rho, \rho' \in \Sigma_S^* \cup \Theta_S^*$ of* Cosmos *machine S, we have*

$$\rho \mathrel{\hat{=}} \rho' \land \mathcal{IOIP}(\rho) \implies \mathcal{IOIP}(\rho')$$

Now, using this fact and argumentation about sequences with the $\mathcal{IOIP}$ condition one can argue about the existence of $\mathcal{IP}$-schedules.

Lemma 2.3 (Interleaving Point Schedule Existence). *For every step sequence θ that fulfills the $\mathcal{IOIP}$ condition, we can find an interleaving point schedule θ' which is an equivalent reordering of θ:*

$$\mathcal{IOIP}(\theta) \implies \exists \theta'.\ \theta' \mathrel{\hat{=}} \theta \land \mathcal{IP}sched(\theta')$$

So, every transition sequence can be reordered into an $\mathcal{IP}$-schedule preserving the $\mathcal{IOIP}$ condition. However, one has to show that equivalent reordering also preserves computations for some starting configuration. This fact was proven in [Bau14b] under requirement that computations of *Cosmos* model obey the ownership policy.

Lemma 2.4 (Safety of Reordered Computations). *For* Cosmos *machine configurations $C, C' \in \mathbb{C}_S$ and step sequences $\sigma, \sigma' \in \Sigma_S^*$ we have*

$$C \overset{\sigma}{\longmapsto} C' \wedge \mathit{safe}(C, \sigma) \wedge \sigma \doteq \sigma' \implies \mathit{safe}(C, \sigma') \wedge C \overset{\sigma'}{\longmapsto} C'$$

Summing up Lemmas 2.2– 2.4 we finally conclude that a given execution of a *Cosmos* machine with arbitrarily interleaved unit steps can be simplified to an execution of the machine with an equivalent $\mathcal{IP}$-schedule.

Corollary 2.1. *Any safe* Cosmos *machine computation (C, σ) resulting in a configuration C' and fulfilling the $\mathcal{IOIP}$ condition can be reordered into an equivalent $\mathcal{IP}$-schedule preserving the result of the machine transition, the ownership safety and the $\mathcal{IOIP}$ condition*

$$C \overset{\sigma}{\longmapsto} C' \wedge \mathit{safe}(C, \sigma) \wedge \mathcal{IOIP}(\sigma) \implies$$
$$\exists \sigma'.\ \sigma' \doteq \sigma \wedge \mathcal{IP}\mathit{sched}(\sigma') \wedge \mathcal{IOIP}(\sigma') \wedge \mathit{safe}(C, \sigma') \wedge C \overset{\sigma'}{\longmapsto} C'$$

For the proof of Lemma 2.4 Baumann introduced additional notation and properties for *Cosmos* machine configurations. Here we present only a few of them which we also explicitly rely on in the present work.

Definition 2.24 (*Cosmos* Model Relations). We define the following equivalence relations on *Cosmos* machine configurations $C, C' \in \mathbb{C}_S$ and a unit $p \in \mathbb{N}_{nu}$ to denote (i) the equality of p's unit state, the read-only and owned memory contents, (ii) the equivalence of the ownership configurations for p, (iii) the combination of both that we call a *p-equivalence*, (iv) the complete equality of the ownership states, and (v) the equality of the shared addresses and contents of the read-only and shared memory regions in the system.

$$\begin{array}{rrcl}
\text{(i)} & C \overset{l}{\sim}_p C' & \overset{def}{\equiv} & C.u_p = C'.u_p \wedge C.m|_{C.\mathcal{O}_p \cup \mathcal{R}} = C'.m|_{C.\mathcal{O}_p \cup \mathcal{R}} \\
\text{(ii)} & C \overset{o}{\sim}_p C' & \overset{def}{\equiv} & C.\mathcal{O}_p = C'.\mathcal{O}_p \wedge C.\mathcal{O}_p \cap C.\mathcal{S} = C'.\mathcal{O}_p \cap C'.\mathcal{S} \\
\text{(iii)} & C \approx_p C' & \overset{def}{\equiv} & C \overset{l}{\sim}_p C' \wedge C \overset{o}{\sim}_p C' \\
\text{(iv)} & C \overset{o}{\sim} C' & \overset{def}{\equiv} & \forall p \in \mathbb{N}_{nu}.\ C.\mathcal{O}_p = C'.\mathcal{O}_p \wedge C.\mathcal{S} = C'.\mathcal{S} \\
\text{(v)} & C \overset{s}{\sim} C' & \overset{def}{\equiv} & C.\mathcal{S} = C'.S \wedge C.m|_{C.S \cup \mathcal{R}} = C'.m|_{C.S \cup \mathcal{R}}
\end{array}$$

As an important property concerning the third relation one can claim that an ownership safe step of one unit does not change the configuration of any other unit, its owned and read-only memories as well as its set of owned addresses.

Lemma 2.5 (Environment Steps). *Given a step $\alpha \in \Sigma_S$ and* Cosmos *machine configurations $C, C' \in \mathbb{C}_S$ such that $C \overset{\alpha}{\mapsto} C'$ holds, we have*

$$\forall p \neq \alpha.s.\ \mathit{safe}(C, \alpha) \implies C \approx_p C'$$

Along with the ownership safety it is usually desirable to verify also other properties of the machine execution, e.g. software conditions, well-formedness invariants, etc. Therefore, to denote such a general property that should hold in any traversed stated of a *Cosmos* machine computation, we introduce a predicate $P : \mathbb{C}_S \to \mathbb{B}$ and extend the ownership safety of a step sequence accordingly:

$$\mathit{safe}_P(C, \sigma) \overset{def}{\equiv} \mathit{safe}(C, \sigma) \land \forall C' \in \mathbb{C}_S.\, C \overset{\sigma}{\longmapsto} C' \implies P(C')$$

Now, as a verification engineer dealing with an instantiated *Cosmos* machine, one would like to know whether the verification of safety properties for all arbitrarily interleaved traces originating from a given starting configuration $C \in \mathbb{C}_S$ is covered by verification of these properties for all traces with $\mathcal{IP}$-schedules starting in the same configuration. The order reduction theorem proven in [Bau14b] easily answers this main question.

Definition 2.25 (Verified *Cosmos* Machine). We define a predicate $\mathit{safety}(C, P)$ 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.

$$\mathit{safety}(C, P) \;\equiv\; \forall \theta.\, \mathit{comp}(C.M, \theta) \implies \exists o \in \Omega_S^*.\, \mathit{safe}_P(C, \langle \theta, o \rangle)$$

We also define a predicate $\mathit{safety}_{\mathcal{IP}}(C, P)$ which expresses the same notion of verification for all $\mathcal{IP}$-schedule computations:

$$\mathit{safety}_{\mathcal{IP}}(C, P) \;\equiv\; \forall \theta.\, \mathcal{IP}\mathit{sched}(\theta) \land \mathit{comp}(C.M, \theta) \implies \exists o \in \Omega_S^*.\, \mathit{safe}_P(C, \langle \theta, o \rangle)$$

Additionally, all $\mathcal{IP}$-schedules starting in C need to fulfill the $\mathcal{IOIP}$ condition.

$$\mathcal{IOIP}_{\mathcal{IP}}(C) \;\equiv\; \forall \theta.\, \mathcal{IP}\mathit{sched}(\theta) \land \mathit{comp}(C.M, \theta) \implies \mathcal{IOIP}(\theta)$$

Thus, we consider a *Cosmos* machine ownership safe if any step sequence can be annotated with an ownership transfer sequence such that the ownership policy is obeyed. Using the definitions above we finally state the aforementioned great theorem.

Theorem 2.1 ($\mathcal{IP}$-Schedule Order Reduction). *For a machine configuration C of a* Cosmos *model 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.*

$$\mathit{safety}_{\mathcal{IP}}(C, P) \land \mathcal{IOIP}_{\mathcal{IP}}(C) \implies \mathit{safety}(C, P)$$

To understand why this theorem actually works we need also to show that every *Cosmos* machine computation can be represented by an equivalently reordered $\mathcal{IP}$-schedule computation. This fact is proven in the following lemma:

Lemma 2.6 (Coverage). *From $\mathit{safety}_{\mathcal{IP}}(C, P)$ and $\mathcal{IOIP}_{\mathcal{IP}}(C)$ it follows that for any* Cosmos *machine computations $(C.M, \theta)$ the $\mathcal{IOIP}$ condition is fulfilled and any equivalently reordered $\mathcal{IP}$-schedule can be executed from C.*

$$\begin{gathered}\mathit{safety}_{\mathcal{IP}}(C, P) \land \mathcal{IOIP}_{\mathcal{IP}}(C) \land \mathit{comp}(C.M, \theta) \implies \\ \mathcal{IOIP}(\theta) \land (\forall \theta'.\, \theta \doteq \theta' \land \mathcal{IP}\mathit{sched}(\theta') \implies \mathit{comp}(C.M, \theta'))\end{gathered}$$

From now, given the order reduction theorem we can treat any arbitrary *Cosmos* machine computation at the granularity of interleaving point schedules if the verification conditions are satisfied.

2.3 System Simulation in Concurrency

As we already know, one usually guarantees a simulation for a sequential execution of a unit for which the underlying semantics without environment steps and the ownership is defined. In the context of a concurrent execution of such units, however, only the engineer is responsible to program the machine in a way that the sequential simulation for each unit is preserved and the code is executed as expected. The ownership safety policy considered in this work models exactly such a strategy chosen by the programmer.

The goal of the rest of this chapter is to formulate and prove a general simulation theorem between two *Cosmos* machines that shows how we treat the sequential simulation for each unit in the concurrent context and allows us to transfer the ownership safety and other verified properties to the implementation layer so that we can talk about them only for $\mathcal{IP}$-schedules where every interleaving point is a consistency point.

For this purpose we first introduce a simple notation that enables switching from the $\mathcal{IP}$-schedules to so called *$\mathcal{IP}$-blocks*. Then, we state a *general sequential simulation theorem* which covers the correctness of a sequential machine execution. As a last step, we provide the *general* Cosmos *model simulation theorem* and its proof.

All these parts are described in detail in [Bau14b]. However, our attempts to apply the original theory for different layers of the verification stack considered in our work showed its restrictions and that the theory does not guarantee some desired properties, what, in turn, makes its application here infeasible. One of the main issues not taken into account in [Bau14b] is that the local unit simulation cannot cover the whole system configuration including the local memories of other units because some of them might have not reached their consistency points. Therefore, the unit simulation relation in the concurrent context has to deal with a dynamically changed subset of memory addresses. To make the theory more intuitive and to disambiguate some of ideas behind it, in our version we also adapt many of definitions coming from the original work. This all required to extend the sequential simulation theorem and to provide the reformulated *Cosmos* model simulation theorem as well as its modified proof.

2.3.1 Block Machine Semantics and Reduction to Block Computations

Since we may assume $\mathcal{IP}$-schedules in our theory, we can simplify the concurrent machine semantics to the subsequent atomic execution of blocks starting in interleaving points. We call the machine implementing such semantics an *$\mathcal{IP}$-block machine* or simply a *block machine*.

Definition 2.26 (**$\mathcal{IP}$-Block**). A transition sequence $\lambda \in \Theta_S^*$ is called an *$\mathcal{IP}$-block* of a unit $p \in \mathbb{N}_{nu}$ if it (i) contains only steps by the given unit, (ii) is empty or starts in an interleaving point, and (iii) does not contain any further interleaving points.

$$\begin{aligned} blk(\lambda, p) \overset{def}{\equiv} \quad & (i) \quad \forall \alpha \in \lambda.\ \alpha.s = p \\ & (ii) \quad \lambda \neq \varepsilon \implies \lambda_1.ip \\ & (iii) \quad \lambda \neq \varepsilon \wedge \lambda = \lambda_1 \circ \lambda' \implies \forall \alpha \in \lambda'.\ \neg\alpha.ip \end{aligned}$$

Definition 2.27 (**Block Machine Schedule**). We define the predicate $Bsched$ which denotes that a given block sequence $\kappa \in (\Theta_S^*)^*$ is a block machine schedule:

$$Bsched(\kappa) \overset{def}{\equiv} \forall \lambda \in \kappa.\ \exists p \in \mathbb{N}_{nu}.\ blk(\lambda, p)$$

Obviously, the *flattening concatenation* $\lfloor \kappa \rfloor \overset{def}{\equiv} \kappa_1 \cdots \kappa_{|\kappa|} \in \Theta_S^*$ of all blocks of κ is an $\mathcal{IP}$-schedule.

Instead of defining an additional transition function for the block machine we easily extend our step sequence notation to block sequences.

Definition 2.28 (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 as follows

$$M \xmapsto{\kappa} M' \overset{def}{\equiv} M \xmapsto{\lfloor\kappa\rfloor} M'$$

Obviously, one can argues that any $\mathcal{IP}$-schedule can be transformed into a block schedule.

Lemma 2.7 (Block Machine Schedule Existence). *For any $\mathcal{IP}$-schedule θ we can find a corresponding block schedule κ such that the flattening concatenation of κ's blocks equals θ.*

$$\forall \theta \in \Theta_S^*.\ \mathcal{IP}sched(\theta) \implies \exists \kappa \in (\Theta^*)^*.\ Bsched(\kappa) \wedge \lfloor\kappa\rfloor = \theta$$

Baumann showed that we can reduce the verification of $\mathcal{IP}$-schedules to the verification of block machine computations.

Definition 2.29 (Block Machine Safety). We call a block machine with an initial *Cosmos* machine configuration $C \in \mathbb{C}_S$ safe wrt. the ownership policy and some property P if the following predicate holds

$$\begin{aligned} &safety_B(C, P) \overset{def}{\equiv} \\ &\quad \forall \kappa \in (\Theta_S^*)^*.\ Bsched(\kappa) \wedge comp(C.M, \lfloor\kappa\rfloor) \implies \exists o \in \Omega_S^*.\ safe_P(C, \langle\lfloor\kappa\rfloor, o\rangle) \end{aligned}$$

Then, the following theorem proven in [Bau14b] justifies the desired reduction.

Theorem 2.2 (Block Machine Reduction). *Let C be a configuration of* Cosmos *model S and P be a* Cosmos *model 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.*

$$safety_B(C, P) \implies safety_{\mathcal{IP}}(C, P)$$

2.3.2 Generalized Sequential Simulation Theorem

Consider two *Cosmos* model instantiations $S_c, S_a \in \mathbb{S}$ and machine computations $(d, \sigma) \in \mathbb{M}_{S_c} \times \Theta_{S_c}^*$ and $(e, \tau) \in \mathbb{M}_{S_a} \times \Theta_{S_a}^*$ such that the steps of the *abstract* layer S_a are simulated by the *concrete* S_c. The interleaving points of both machines are instantiated as *consistency points* and unit transitions are performed now on so called *consistency blocks* substituting the term $\mathcal{IP}$-block in the simulation context.

Thanks to the order reduction theorem, it is clear that the simulation theorem proven for the sequential execution without environment steps can now be applied for any consistency block. In out theory of *Cosmos* models we provide its generalized version that need to be instantiated with respect to the sequential simulation between the given concrete and abstract machines. As we mentioned before, in contrast to [Bau14b], the sequential simulation has to take into account the dynamically changed subsets of addresses at which the memory consistency might not hold (*inconsistent memory*). Moreover, we only demand the same number of computation units in both machines $S_a.nu = S_c.nu = nu$ and do not require that the memories of both machines have compatible types (i.e., $S_c.\mathcal{A} \supseteq S_a.\mathcal{A}$ and $S_c.\mathcal{V} = S_a.\mathcal{V}$) because the model S_a might contain an abstraction of the implementation memory.

For this purpose we define a so called *sequential simulation framework* with components usually having counterparts in the sequential simulation between any two machines. Below, for shortness we denote all components $S_a.X$ and $S_c.X$ of the corresponding machines as X_a and X_c respectively.

Definition 2.30 (Sequential Simulation Framework). We introduce a type $\mathbb{R}$ for simulation frameworks $R_{S_c}^{S_a} \in \mathbb{R}$ which contains all the information needed to state the generalized simulation theorem relating sequential computations of units of *Cosmos* models S_c and S_a.

$$\mathbb{R} \overset{def}{\equiv} (\mathcal{P}, sim, \mathcal{CP}a, \mathcal{CP}c, wfa, wfc, suit, sc, wb)$$

- $\mathcal{P}$ – a set of static simulation parameters ($\{\perp\}$ if there are none) common for all units.
- $sim : \mathbb{L}_c \times \mathbb{L}_a \times \mathcal{P} \times 2^{\mathcal{A}_a} \rightarrow \mathbb{B}$ – a simulation relation between given units of S_c and S_a, where $\mathbb{L}_x \equiv (\mathcal{U}_x \times (\mathcal{A}_x \rightarrow \mathcal{V}_x))$ with $x \in \{c, a\}$ is a shorthand for a *sequential machine configuration* containing a state of the unit of the concurrent machine and the whole memory. The parameter of the type $2^{\mathcal{A}_a}$ represents a set of addresses for the possible inconsistent memory.
- $\mathcal{CP}a : \mathcal{U}_a \times \mathcal{P} \rightarrow \mathbb{B}$ – a predicate identifying consistency points of the abstract S_a.
- $\mathcal{CP}c : \mathcal{U}_c \times \mathcal{P} \rightarrow \mathbb{B}$ – a predicate identifying consistency points of the concrete S_c.
- $wfa : \mathbb{L}_a \rightarrow \mathbb{B}$ – a well-formedness condition for a sequential machine configuration for a given unit of the abstract S_a,
- $wfc : \mathbb{L}_c \rightarrow \mathbb{B}$ – well-formedness condition for a sequential machine configuration of the concrete S_c, required for the simulation of sequential computations of S_a.
- $suit : \mathcal{E}_c \rightarrow \mathbb{B}$ – a predicate determining whether a given input of the concrete machine is suitable for the simulation.
- $sc : \mathbb{L}_a \times \mathcal{E}_a \times \mathcal{P} \rightarrow \mathbb{B}$ – software conditions that enable a simulation of sequential computations of S_a. It is defined for a sequential machine configuration of a given unit and a corresponding input for a unit's step.
- $wb : \mathbb{L}_c \times \mathcal{E}_c \times \mathcal{P} \rightarrow \mathbb{B}$ – a predicate that restricts the simulating computations of S_c. We say that a simulating step in a computation of S_c is *well-behaved* iff it fulfills this restriction. The property becomes especially important when we consider a verification stack and need to transfer a part of software conditions to the implementation layer.

To apply the sequential simulation framework in the context of machine step sequences we introduce additional shorthands for $d \in \mathbb{M}_{S_c}$, $e \in \mathbb{M}_{S_a}$, $p \in \mathbb{N}_{nu}$, $par \in \mathcal{P}$ (where $\mathcal{P} \equiv R_{S_c}^{S_a}.\mathcal{P}$),

$icm \in 2^{\mathcal{A}_a}$, steps $\alpha \in \Theta_{S_a}$, $\beta \in \Theta_{S_c}$, and step sequences $\omega \in \Theta^*_{S_c}$, $\tau \in \Theta^*_{S_a}$:

$$\begin{aligned}
sim_p(d, e, par, icm) &\equiv R^{S_a}_{S_c}.sim((d.u(p), d.m), (e.u(p), e.m), par, icm)\\
\mathcal{CP}_p(e, par) &\equiv R^{S_a}_{S_c}.\mathcal{CP}a(e.u(p), par)\\
\mathcal{CP}_p(d, par) &\equiv R^{S_a}_{S_c}.\mathcal{CP}c(d.u(p), par)\\
wf_p(e) &\equiv R^{S_a}_{S_c}.wfa(e.u(p), e.m)\\
wf_p(d) &\equiv R^{S_a}_{S_c}.wfc(d.u(p), d.m)\\
suit(\beta) &\equiv R^{S_a}_{S_c}.suit(\beta.in)\\
suit(\omega) &\equiv \forall \beta \in \omega.\ suit(\beta)\\
wb(d, \beta, par) &\equiv R^{S_a}_{S_c}.wb((d.u(\beta.s), d.m), \beta.in, par)\\
wb(d, \omega, par) &\equiv \forall \theta, \alpha, \theta', d'.\ \omega = \theta\alpha\theta' \wedge d \overset{\theta}{\longmapsto} d' \implies wb(d', \alpha, par)\\
sc(e, \alpha, par) &\equiv R^{S_a}_{S_c}.sc((e.u(\alpha.s), e.m), \alpha.in, par)\\
sc(e, \tau, par) &\equiv \forall \theta, \alpha, \theta', e'.\ \tau = \theta\alpha\theta' \wedge e \overset{\theta}{\longmapsto} e' \implies sc(e', \alpha, par)
\end{aligned}$$

If the given sequence does not match the computation, namely io, ot or/and ip do not correspond to configurations in the computation, or the transition is undefined, the predicates wb and sc can still hold. This issue is resolved when we use the predicates under the requirement of computation existence.

Along with the software conditions, we introduce also a predicate requiring that the inconsistent region of memory is not accessed by a unit during its step.

Definition 2.31 (No Access to Inconsistent Memory). For abstract machine configurations $e, e' \in \mathbb{M}_{S_a}$, $icm \in 2^{\mathcal{A}_a}$ and an existing step $e \overset{\alpha}{\mapsto} e'$ with $\alpha \in \Theta_{S_a}$ we define

$$noacc(e, \alpha, icm) \overset{def}{\equiv} (reads_{\alpha.s}(e, \alpha.in) \cup writes_{\alpha.s}(e, \alpha.in)) \cap icm = \emptyset$$

The definition easily extends to a sequence $\tau \in \Theta^*_{S_a}$:

$$noacc(e, \tau, icm) \equiv \forall \theta, \alpha, \theta', e'.\ \tau = \theta\alpha\theta' \wedge e \overset{\theta}{\longmapsto} e' \implies noacc(e', \alpha, icm)$$

Since our order reduction theory assumes the $\mathcal{IOIP}$ condition, we have to guarantee at least one consistency point between two $\mathcal{IO}$-points. For example, in case of compiler correctness, the compiler is supposed to insert the consistency points into the compiled code in the proper way. Moreover, there must be one-to-one mapping of the $\mathcal{IO}$-points suitable for the ownership transfer and any $\mathcal{IO}$-step performed on the abstract level has to be also implemented by the concrete machines.

In contrast to [Bau14b], where the consistency blocks of both machines must contain the same number of $\mathcal{IO}$-operations, our requirement is weaker and allows to argue about $\mathcal{IO}$-steps of the concrete machine that has no effect on the abstract level. As an example, one could consider an unsuccessful attempt to acquire a lock which implemented as a read of a shared resource without an ownership transfer. In this case, one could reduce the $\mathcal{IO}$-operation to an empty step of the abstract machine.

Definition 2.32 (Requirement on $\mathcal{IO}$-points in Consistency Blocks). We denote the requirements on $\mathcal{IO}$-points in consistency blocks by an overloaded predicate $one\mathcal{IO}$. For a transition sequence $\omega \in \Theta^*_{S_a} \cup \Theta^*_{S_c}$ it demands that the sequence contains at most one $\mathcal{IO}$-step. For two such sequences $\sigma \in \Theta^*_{S_c}$, $\sigma \neq \varepsilon$, and $\tau \in \Theta^*_{S_a}$, however, we require the same number of $\mathcal{OT}$-steps, and if τ contains an $\mathcal{IO}$-operation, σ does it also.

$$\begin{aligned} \mathit{one}\mathcal{IO}(\omega) &\overset{def}{\equiv} \forall i, j \in \mathbb{N}_{|\omega|}.\ \omega_i.io \wedge \omega_j.io \implies i = j \\ \mathit{one}\mathcal{IO}(\sigma,\tau) &\overset{def}{\equiv} \begin{array}{ll} \textit{(i)} & \mathit{one}\mathcal{IO}(\sigma) \wedge \mathit{one}\mathcal{IO}(\tau) \\ \textit{(ii)} & \sigma|_{ot} \neq \varepsilon \Longleftrightarrow \tau|_{ot} \neq \varepsilon \\ \textit{(iii)} & \tau|_{io} \neq \varepsilon \implies \sigma|_{io} \neq \varepsilon \end{array} \end{aligned}$$

Note, that this requirement is still too strong for the simulation where the concrete machine accesses some shared resources invisible on the abstract level and supports the ownership transfer for them. Since we do not deal with such models in the scope of the thesis, we leave this issue for the future work.

The sequential simulation theorem can be applied on any consistency block on the implementation level in order to obtain an abstract consistency block executed by the same unit. However, a given consistency block for a concrete model might be *incomplete*, meaning that the computation unit has not reached the next consistency point yet. Therefore, one has to find an extension of that incomplete block, so that the resulting complete concrete block implements an abstract one. Via this extension we can show exactly the existence of the next consistency point, what must be claimed in the generalized sequential simulation theorem.

Definition 2.33 (Extension of Consistency Block). The extension of a given consistency block $\omega \in \Theta^*_{S_c}$ is denoted by $\omega \rhd^{blk}_p \sigma$ and says that $\sigma \in \Theta^*_{S_c}$ extends ω without adding consistency points to the block. The given ω can be considered as a prefix of the non-empty σ.

$$\omega \rhd^{blk}_p \sigma \overset{def}{\equiv} \exists \tau.\ \sigma = \omega\tau \neq \varepsilon \wedge \mathit{blk}(\sigma, p) \wedge \mathit{blk}(\omega, p)$$

Using the definitions introduced above we can finally state the generalized sequential simulation theorem for two *Cosmos* models $S_a, S_c \in \mathbb{S}$ and a corresponding simulation framework $R^{S_a}_{S_c} \in \mathbb{R}$.

Theorem 2.3 (Generalized Sequential Simulation Theorem). *For any couple of abstract and concrete machine configurations $e \in \mathbb{M}_{S_a}$, $d \in \mathbb{M}_{S_c}$, any unit $p \in \mathbb{N}_{nu}$, and any possible inconsistent portion of memory $icm \in 2^{\mathcal{A}_a}$,*

- *if the unit p in both machines is in a consistency point, well-formed and coupled via the simulation relation wrt. icm and a simulation parameter $par \in \mathcal{P}$, and*
- *if all complete consistency block computations of the unit p in the abstract machine starting from e obey the software conditions, do not access icm and lead to well-formed configurations,*

then for any existing suitable non-empty computations $\omega \in \Theta^*_{S_c}$ from d of the unit p we guarantee that there exist further suitable steps of the same unit leading to the next consistency point such that

- *the steps of the resulting complete block are suitable and well-behaved,*
- *implement an existing complete consistency block of the abstract machine,*
- *and the simulation relation excluding the same icm holds after the steps of both machines in their well-formed configurations for the unit p.*

- *Moreover, the abstract and concrete complete consistency blocks contain at most one $\mathcal{IO}$-step, an $\mathcal{IO}$-step of the abstract machine is always implemented by the concrete machine, and the ownership transfer may be performed in the computations of only both machines.*

$$\forall d \in \mathbb{M}_{S_c}, e \in \mathbb{M}_{S_a}, par \in \mathcal{P}, p \in \mathbb{N}_{nu}, icm \in 2^{\mathcal{A}_a}, \omega \in \Theta^*_{S_c}.$$

$$\begin{aligned}
&\textit{(i)} && wf_p(d) \wedge wf_p(e) \\
&\textit{(ii)} && sim_p(d, e, par, icm) \wedge \mathcal{CP}_p(d, par) \wedge \mathcal{CP}_p(e, par) \\
&\textit{(iii)} && \forall \lambda \in \Theta^*_{S_a}, e' \in \mathbb{M}_{S_a}.\ e \xmapsto{\lambda} e' \wedge blk(\lambda, p) \wedge \mathcal{CP}_p(e', par) \implies \\
& && \qquad noacc(e, \lambda, icm) \wedge sc(e, \lambda, par) \wedge wf_p(e') \\
&\textit{(iv)} && \omega \neq \varepsilon \wedge blk(\omega, p) \wedge suit(\omega) \wedge comp(d, \omega)
\end{aligned}$$

$$\implies$$

$$\exists \sigma \in \Theta^*_{S_c}, d' \in \mathbb{M}_{S_c}, \tau \in \Theta^*_{S_a}, e' \in \mathbb{M}_{S_a}.$$

$$\begin{aligned}
&\textit{(i)} && \omega \rhd^{blk}_p \sigma \wedge suit(\sigma) \\
&\textit{(ii)} && blk(\tau, p) \wedge one\mathcal{IO}(\sigma, \tau) \\
&\textit{(iii)} && d \xmapsto{\sigma} d' \wedge wb(d, \sigma, par) \wedge wf_p(d') \\
&\textit{(iv)} && e \xmapsto{\tau} e' \wedge wf_p(e') \\
&\textit{(v)} && sim_p(d', e', par, icm) \wedge \mathcal{CP}_p(d', par) \wedge \mathcal{CP}_p(e', par)
\end{aligned}$$

Note that for the simulated computation (e, τ) we only demand progress (i.e., $\tau \neq \varepsilon$) in case σ contains $\mathcal{IO}$-steps suitable for the ownership transfer. This easily follows from the definition of $one\mathcal{IO}(\sigma, \tau)$.

Moreover, the claim for all icm allows us to apply the theorem for concurrent systems where environment steps are allowed. As we know already from the order reduction, the interleaving of blocks can appear only in consistency points. Therefore, before the execution of a unit p there might be an incomplete block of another unit $q \neq p$ that destroyed the consistency of some region of memory. So, in contrast to the classical inductive proofs for systems with a single unit done for the static $icm = \emptyset$, our formulation covers all possible cases of environment steps. In the presence of the ownership state and safety policy, we will be able to see which memory portions exactly could be inconsistent.

Now, since the sequential simulation theorem guarantees existence of consistency points, we can consider sequences of consistency block for the abstract and implementation layers and formulate our final correctness statement for concurrent systems.

2.3.3 Cosmos Model Simulation

We have already mentioned the complete and incomplete consistency blocks. First, we need to provide their formal definitions.

Consistency Block Machines

Definition 2.34 (Interleaving Points are Consistency Points). Given a sequential simulation framework $R^{S_a}_{S_c}$ which relates two *Cosmos* models S_c and S_a and a simulation parameter $par \in \mathcal{P}$ we define a predicate denoting that in S_c and S_a the interleaving points are set up to be exactly the consistency points.

$$\mathcal{IPCP}(R_{S_c}^{S_a}, par) \overset{def}{\equiv} \begin{array}{ll} \text{(i)} & \forall d \in \mathbb{M}_{S_c}, \alpha \in \Theta_{S_c}.\, \mathcal{IP}_{\alpha.s}(d, \alpha.in) \iff \mathcal{CP}_{\alpha.s}(d, par) \\ \text{(ii)} & \forall e \in \mathbb{M}_{S_a}, \beta \in \Theta_{S_a}.\, \mathcal{IP}_{\beta.s}(e, \beta.in) \iff \mathcal{CP}_{\beta.s}(e, par) \end{array}$$

Definition 2.35 (Consistency Block Machine Schedule). A block machine schedule $\kappa \in (\Theta_{S_c}^*)^* \cup (\Theta_{S_a}^*)^*$ is a *consistency block machine schedule* iff the following property holds:

$$\mathcal{CP}sched(\kappa, par) \overset{def}{\equiv} Bsched(\kappa) \wedge \mathcal{IPCP}(R_{S_c}^{S_a}, par)$$

In the concurrent model simulation theorem between the concrete and abstract machines we will aim at the verification of safety properties and software conditions for all consistency block machine computations of the abstract machine with all units being in their consistency points after each block execution. Therefore for such a consistency block schedule we add a corresponding requirement.

Definition 2.36 (Complete Consistency Block Machine Computation). A *complete consistency block machine computation* of a block schedule $\nu \in (\Theta_{S_a}^*)^*$ starting from a configuration $e \in \mathbb{M}_{S_a}$ is a block machine computation where all computation units are in consistency points in every configuration

$$\begin{array}{lll} \mathcal{CP}sched_c(e, \nu, par) & \overset{def}{\equiv} & \mathcal{CP}sched(\nu, par) \wedge \forall \nu', \nu'' \in (\Theta_{S_a}^*)^*, e' \in \mathbb{M}_{S_a}. \\ & & \quad \nu = \nu'\nu'' \wedge e \overset{\nu'}{\longmapsto} e' \implies \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(e', par) \end{array}$$

The same definition can be reloaded for the concrete machine with $d \in \mathbb{M}_{S_c}$ and $\kappa \in (\Theta_{S_c}^*)^*$:

$$\begin{array}{lll} \mathcal{CP}sched_c(d, \kappa, par) & \overset{def}{\equiv} & \mathcal{CP}sched(\kappa, par) \wedge \forall \kappa', \kappa'' \in (\Theta_{S_c}^*)^*, d' \in \mathbb{M}_{S_c}. \\ & & \quad \kappa = \kappa'\kappa'' \wedge d \overset{\kappa'}{\longmapsto} d' \implies \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(d', par) \end{array}$$

For block schedules of concrete and abstract machines we introduce a definition of the schedule equivalence that will be later used in the simulation between both machines.

Definition 2.37 (Block Schedules Equivalence). For a block schedule $\kappa \in (\Theta_{S_c}^*)^*$ of the concrete *Cosmos* model with non-empty blocks and a block schedule $\nu \in (\Theta_{S_a}^*)^*$ of the abstract *Cosmos* model with possibly empty blocks or both empty sequences we introduce the equivalence relation denoting that both schedules are of the same length and corresponding blocks in these schedules belong to the same unit. Moreover, each block of both schedules obey the condition $one\mathcal{IO}$.

$$\nu \overset{sch}{\sim} \kappa \overset{def}{\equiv} \begin{array}{llll} \text{(i)} & |\nu| = |\kappa| & & \\ \text{(ii)} & \forall j \in \mathbb{N}.\, j \le |\kappa| \implies & \text{(a)} & one\mathcal{IO}(\kappa_j, \nu_j) \\ & & \text{(b)} & \forall p, q \in \mathbb{N}_{nu}.\, blk(\nu_j, q) \wedge \nu_j \neq \varepsilon \wedge \\ & & & \qquad blk(\kappa_j, p) \implies p = q \end{array}$$

Verification of the Abstract Machine

Analogously to the block machine safety we define the notion of the safety for our complete consistency block machine computations.

Definition 2.38 (**Complete Consistency Block Machine Safety).** The verification of the ownership safety and some property P for all complete block machine computations running out of a configuration $C \in \mathbb{C}_{S_a} \cup \mathbb{C}_{S_c}$ is denoted by a predicate $\mathit{safety}_{cB}(C, P, \mathit{par})$ with $\mathit{par} \in \mathcal{P}$, where, depending on the machine in question, Ω is either Ω_{S_a} or Ω_{S_c}, and Θ is either Θ_{S_a} or Θ_{S_c}:

$$\begin{aligned} \mathit{safety}_{cB}(C, P, \mathit{par}) \overset{\mathit{def}}{\equiv}\ & \forall \kappa \in (\Theta^*)^*. \\ & \quad \mathit{(i)}\quad \mathcal{CP}\mathit{sched}_c(C.M, \kappa, \mathit{par}) \\ & \quad \mathit{(ii)}\quad \mathit{comp}(C.M, \lfloor \kappa \rfloor) \\ & \quad \implies \exists o \in \Omega^*.\ \mathit{safe}_P(C, \langle \lfloor \kappa \rfloor, o \rangle) \end{aligned}$$

As we can see this definition of safety taken from [Bau14b] does not directly correspond to the previously considered statement of the block machine safety but has a similar form. In fact, given that it is always possible to reach a consistency point we could prove the reduction of arbitrary consistency block machine schedules to the complete ones. However, the existence of consistency points and aforementioned reachability can only be justified via the simulation between two machines. Moreover, we do not use the concurrent simulation theorem to show that the abstract machine properties verified for complete consistency block machine computations can be transferred to arbitrary interleaving schedules on the same abstract level. Thus it is useless to treat the reduction of incomplete blocks on a single layer of abstraction.

At it follows from the predicate $\mathit{safety}_{cB}(C, P, \mathit{par})$, we are interested in the concurrent simulation for complete block abstract machine computations for which a safe ownership transfer annotation exists and the property P holds in the configuration C and after each block execution. To denote this for a block machine schedule $\kappa \in (\Theta^*)^*$, an ownership transfer annotation $o \in \Omega^*$, s.t. $|o| = |\lfloor \kappa \rfloor|$ we define the following predicate which we will use in the simulation theorem rather as a shorthand:

$$\begin{aligned} \mathit{safe}_{cB}(C, P, \kappa, o) \overset{\mathit{def}}{\equiv}\ & \forall \kappa', \kappa'' \in (\Theta^*)^*, o', o'' \in \Omega^*. \\ & \kappa = \kappa' \kappa'' \land o = o' o'' \land |o'| = |\lfloor \kappa' \rfloor| \implies \\ & \mathit{safe}_P(C, \langle \lfloor \kappa' \rfloor, o' \rangle) \end{aligned}$$

Obviously, by induction on the length of the block schedule sequence, one can easily show the correspondence between these definitions:

$$\begin{aligned} \mathit{safety}_{cB}(C, P, \mathit{par}) \implies & \forall \kappa \in (\Theta^*)^*. \\ & \quad \mathit{(i)}\quad \mathcal{CP}\mathit{sched}_c(C.M, \kappa, \mathit{par}) \\ & \quad \mathit{(ii)}\quad \mathit{comp}(C.M, \lfloor \kappa \rfloor) \\ & \quad \implies \exists o \in \Omega^*.\ \mathit{safe}_{cB}(C, P, \kappa, o) \end{aligned}$$

Analogously to the safety properties, it is enough to prove the software conditions (SC) only for complete consistency blocks on the abstract level.

Definition 2.39 (**Verification of SC and Well-Formedness for Complete Block Machines).** For all complete block machine schedules running out of a configuration $e \in \mathbb{M}_{S_a}$ we define a predicate denoting the verification of software conditions and the well-formedness invariant for all abstract complete block machine computations:

$$\begin{aligned} scwf_{cB}(e, par) \overset{def}{\equiv}\ & \forall \eta \in \left(\Theta^*_{S_a}\right)^* . \\ & \quad \textit{(i)} \quad \mathcal{CP}sched_c(e, \eta, par) \\ & \quad \textit{(ii)} \quad comp(e, \lfloor \eta \rfloor) \\ & \quad \Longrightarrow \\ & \quad \textit{(i)} \quad sc\,(e, \lfloor \eta \rfloor, par) \\ & \quad \textit{(ii)} \quad \forall e' \in \mathbb{M}_{S_a}.\ e \xmapsto{\eta} e' \implies \forall p \in \mathbb{N}_{nu}.\ wf_p(e') \end{aligned}$$

Definition 2.40 (Verified Abstract *Cosmos* Machine). We call the abstract *Cosmos* machine S_a starting its computations in a configuration $E \in \mathbb{M}_{S_a}$ *verified* wrt. the simulation parameter $par \in \mathcal{P}$ and the property P_{S_a} if and only if the predicates $safety_{cB}(E, P_{S_a}, par)$ and $scwf_{cB}(E.M, par)$ hold.

Invariants and Simulation Relation Between *Cosmos* Machines

First, we introduce a few auxiliary definitions which will be used in the simulation relation between two machines. We already know that in the concrete machine computations not all units might reach the consistency points.

Definition 2.41 (Units in Consistency Points). Given a machine configuration $d \in \mathbb{M}_{S_c}$ and a simulation parameter par as above we can define the set U_{cp} of computation units of d that are in consistency points wrt. the simulation parameter par.

$$U_{cp}(d, par) \overset{def}{\equiv} \{p \in \mathbb{N}_{nu} \mid \mathcal{CP}_p(d, par)\}$$

As mentioned before in the concurrent execution of the consistency blocks the simulation relation of a unit should exclude memory addresses for which the memory might be possibly inconsistent because of units not reached their consistency points. Obviously, as such a memory region for a given unit we can choose other units' local addresses because the unit does not access them anyway.

Definition 2.42 (Other Units' Local Addresses). For a unit $p \in \mathbb{N}_{nu}$ of a model $S \in \mathbb{S}$ we compute a set of local addresses of all other units from the ownership state $\mathcal{G} \in \mathbb{G}_S$ as follows:

$$\overline{\mathcal{SO}}(\mathcal{G}, p) \overset{def}{\equiv} \bigcup_{q \neq p} \mathcal{G}.\mathcal{O}(q) \setminus \mathcal{G}.\mathcal{S}$$

Then, for a given unit one can prove an easy lemma about the whole memory space excluding other units' local addresses.

Lemma 2.8 (Memory Addresses without other Units' Local Addresses). *For any* Cosmos *machine $S \in \mathbb{S}$, its configuration $E \in \mathbb{C}_S$ and a unit $p \in \mathbb{N}_{nu}$, if the ownership invariant $oinv(E)$ holds, then the set all memory addresses without the owned non-shared addresses of all units except p is equal to the union of the read-only, shared and owned by the unit p addresses.*

$$oinv(E) \implies S.\mathcal{A} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) = S.\mathcal{R} \cup E.\mathcal{S} \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})$$

Proof: To prove the claim we open the definition of $\overline{\mathcal{SO}}(E.\mathcal{G},p)$ and use parts (i)-(iii) of the invariant $oinv(E)$ to simplify the computations on the given sets of addresses.

$$\begin{aligned}
S.\mathcal{A} \setminus \overline{\mathcal{SO}}(E.\mathcal{G},p) &= S.\mathcal{A} \setminus \left(\bigcup_{q \neq p} E.\mathcal{O}_q \setminus E.\mathcal{S} \right) \\
&= \left(E.\mathcal{R} \cup E.\mathcal{S} \cup \bigcup_{q} E.\mathcal{O}_q \right) \setminus \left(\bigcup_{q \neq p} E.\mathcal{O}_q \setminus E.\mathcal{S} \right) && \text{by } oinv(E).(iii) \\
&= E.\mathcal{R} \cup E.\mathcal{S} \cup \left(\left(\bigcup_{q} E.\mathcal{O}_q \right) \setminus \left(\bigcup_{q \neq p} E.\mathcal{O}_q \setminus E.\mathcal{S} \right) \right) && \text{by } oinv(E).(ii) \\
&= E.\mathcal{R} \cup E.\mathcal{S} \cup \left(\left(\bigcup_{q} E.\mathcal{O}_q \setminus E.\mathcal{S} \right) \setminus \left(\bigcup_{q \neq p} E.\mathcal{O}_q \setminus E.\mathcal{S} \right) \right) \\
&= S.\mathcal{R} \cup E.\mathcal{S} \cup (E.\mathcal{O}_p \setminus E.\mathcal{S}) && \text{by } oinv(E).(i)
\end{aligned}$$

□

Now, using the sequential simulation relation from the framework $R_{S_c}^{S_a}$, we define the overall concurrent simulation relation for all units in consistency points.

Definition 2.43 (Concurrent Simulation Relation). For a *Cosmos* machine configuration $E \in \mathbb{C}_{S_a}$, a concrete concurrent machine configuration $d \in \mathbb{M}_{S_c}$, the simulation parameter $par \in \mathcal{P}$, and a unit $p \in \mathbb{N}_{nu}$ we define its sequential simulation relation in the context of concurrent execution:

$$csim_p(d,E,par) \overset{def}{\equiv} sim_p\left(d, E.M, par, \overline{\mathcal{SO}}(E.\mathcal{G},p)\right)$$

Then, for all units in consistency points we demand

$$csim(d,E,par) \overset{def}{\equiv} \forall p \in U_{cp}(d,par).\ csim_p(d,E,par)$$

Since the concrete and abstract *Cosmos* machines contain the ownership states, we are also supposed to couple them in our theorem. Additionally, we are interested in the separate simulation relation between shared and read-only memories. We will claim such coupling not only for the units in consistency points, but also for those that have not reached them. For this purpose we provide a so called *shared invariant* introduced in [Bau14b].

Definition 2.44 (Shared Invariant). For concrete and abstract *Cosmos* machines S_c, S_a, and the sequential simulation framework $R_{S_c}^{S_a}$ the shared invariant $sinv_{S_c}^{S_a}$ couples and constrains the ownership states as well as the shared and read-only memories of both machines. For $\mathcal{M}_x = \mathcal{A}_x \rightharpoonup \mathcal{V}_x$ and $\mathbb{O}_x = \mathbb{N}_{nu} \to 2^{\mathcal{A}_x}$ with $x \in \{c,a\}$ we have

$$sinv_{S_c}^{S_a} : (\mathcal{M}_c \times 2^{\mathcal{A}_c} \times 2^{\mathcal{A}_c} \times \mathbb{O}_c) \times (\mathcal{M}_a \times 2^{\mathcal{A}_a} \times 2^{\mathcal{A}_a} \times \mathbb{O}_a) \times \mathcal{P} \to \mathbb{B}$$

Moreover, we introduce a shorthand asserting the shared invariant on two *Cosmos* machine configurations D and E. Let $G(C) \equiv (C.m|_{C.\mathcal{S} \cup S.\mathcal{R}}, C.\mathcal{S}, S.\mathcal{R}, C.\mathcal{O})$, then we define

$$sinv(D,E,par) \equiv sinv_{S_c}^{S_a}(G(D),G(E),par)$$

One can easily prove an important property about the shared invariant preservation in case of local steps by the concrete machine [Bau14b].

Lemma 2.9 (Safe Local Steps Preserve Shared Invariant). *Consider concrete* Cosmos *machine configurations $D, D' \in \mathbb{C}_{S_c}$ such that D' is reached from D by a step sequence ω containing no $\mathcal{IO}$-steps. Moreover, $\hat{E} \in \mathbb{C}_{S_a}$ is an abstract machine configuration corresponding to the beginning or the end of the computation (D, ω). If the given concrete computation is ownership safe, then D and $\hat{E}$ are coupled by the shared invariant if and only if D' and $\hat{E}$ are.*

$$D \overset{\omega}{\longmapsto} D' \wedge \omega|_{io} = \varepsilon \wedge \mathit{safe}(D, \omega) \implies (\mathit{sinv}(D, \hat{E}, \mathit{par}) \Leftrightarrow \mathit{sinv}(D', \hat{E}, \mathit{par}))$$

We will state the concurrent simulation theorem for two machines with initial configurations $D \in \mathbb{C}_{S_c}$ and $E \in \mathbb{C}_{S_a}$ that are coupled via $\mathit{csim}(D.M, E, \mathit{par})$ and $\mathit{sinv}(D, E, \mathit{par})$. However, it is also typical to require some conditions on the initial configuration of the machine implementing the abstract one. We resolve this issue, not considered in [Bau14b], with the help of an additional invariant for the units on the concrete level.

Definition 2.45 (Concrete Machine Unit Invariant). We introduce a *concrete machine unit invariant* that may require some conditions on the unit's configuration and couple it with the ownership configuration if needed:

$$\mathit{uinv}_{S_c}^{S_a} : \mathcal{U}_c \times 2^{\mathcal{A}_c} \times 2^{\mathcal{A}_c} \to \mathbb{B}$$

For a unit $p \in \mathbb{N}_{nu}$ and a *Cosmos* machine configuration $D \in \mathbb{C}_{S_c}$ we give the shorthand

$$\mathit{uinv}_p(D) \equiv \mathit{uinv}_{S_c}^{S_a}(D.u_p, D.\mathcal{O}_p, D.\mathcal{S})$$

Moreover, we demand the given invariant to hold only for units being in consistency points:

$$\mathit{uinv}(D, \mathit{par}) \equiv \forall p \in U_{cp}(D.M, \mathit{par}).\ \mathit{uinv}_p(D)$$

As an example of a simulation needed such an invariant, one could consider a concrete machine with units buffering memory write accesses though this is invisible on the abstract level. A simple solution allowing the simulation might require the buffers of all units to contain only their local owned addresses.

Cosmos Model Simulation Theorem

Now, having the sequential simulation framework $R_{S_c}^{S_a}$, shared invariant $\mathit{sinv}_{S_c}^{S_a}$, unit invariant $\mathit{uinv}_{S_c}^{S_a}$ if needed, and a property $P_{S_a} : \mathbb{C}_{S_a} \to \mathbb{B}$ restricting the ownership safety policy in a desired way, we can state the main *Cosmos* model simulation theorem.

Theorem 2.4 (*Cosmos* Model Simulation Theorem). *For any starting configurations $D \in \mathbb{C}_{S_c}$ and $E \in \mathbb{C}_{S_a}$ of the concrete and abstract* Cosmos *machines respectively, if*

- *there exist units that are in consistency points in D, and all such units have well-formed configurations and obey the unit invariant,*
- *all units in E are in consistency points,*
- *the shared invariant between D and E holds, and the units in consistency point in D are coupled with E via the simulation relation wrt. the simulation parameter par and the ownership state of the abstract machine, and*

- *if the execution of the abstract machine starting in E is verified wrt. the software conditions, well-formedness, ownership safety policy, and the property P_{S_a} restricting it,*

then for any suitable consistency block machine schedule $\kappa \in (\Theta^*_{S_c})^*$ with non-empty blocks executable from $D.M$, there exists a complete block machine schedule $\nu \in (\Theta^*_{S_a})^*$ equivalent to κ according to $\nu \overset{sch}{\sim} \kappa$ and executable from $E.M$ such that

- *the computation $(E.M, \nu)$ leads to well-formed configurations of all units in the abstract machine and is simulated by the well-behaved computation $(D.M, \kappa)$ ending in a configuration with well-formed units in consistency points, and*
- *for any ownership annotation that is safe for $(E.M, \nu)$ wrt. the property $\mathcal{P}_{S_a}$ we guarantee that the resulting configurations of both machines are coupled with the concurrent simulation relation.*
- *Moreover, we can transfer the ownership safety to the implementation layer such that the unit invariant in consistency points as well as the shared invariant between the abstract and concrete* Cosmos *machines are preserved at the end of the computations.*

$$\forall D \in \mathbb{C}_{S_c}, E \in \mathbb{C}_{S_a}, par \in \mathcal{P}, \kappa \in (\Theta^*_{S_c})^*.$$

$$
\begin{array}{ll}
\textit{(i)} & \exists p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(D.M, par)\\
\textit{(ii)} & uinv(D, par) \wedge \forall p \in U_{cp}(D.M, par).\ wf_p(D.M)\\
\textit{(iii)} & \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(E.M, par)\\
\textit{(iv)} & csim\,(D.M, E, par) \wedge sinv(D, E, par)\\
\textit{(v)} & safety_{cB}(E, P_{S_a}, par) \wedge scwf_{cB}(E.M, par)\\
\textit{(vi)} & \mathcal{CP}sched(\kappa, par) \wedge suit\,(\lfloor\kappa\rfloor) \wedge comp(D.M, \lfloor\kappa\rfloor) \wedge \forall\omega \in \kappa.\, \omega \neq \varepsilon
\end{array}
$$

$$\Longrightarrow$$

$$\exists \nu \in (\Theta^*_{S_a})^*, M'_e \in \mathbb{M}_{S_a}, M'_d \in \mathbb{M}_{S_c}.$$

$$
\begin{array}{ll}
\textit{(i)} & \mathcal{CP}sched_c(E.M, \nu, par) \wedge \nu \overset{sch}{\sim} \kappa\\
\textit{(ii)} & E.M \overset{\nu}{\longmapsto} M'_e \wedge \forall p \in \mathbb{N}_{nu}.\ wf_p(M'_e)\\
\textit{(iii)} & D.M \overset{\kappa}{\longmapsto} M'_d \wedge wb(D.M, \lfloor\kappa\rfloor, par) \wedge \forall p \in U_{cp}(M'_d, par).\ wf_p(M'_d)\\
\textit{(iv)} & \forall o_\nu \in (\Omega_{S_a})^*, \mathcal{G}'_e \in \mathbb{G}_{S_a}.
\end{array}
$$

$$E \overset{\langle\lfloor\nu\rfloor, o_\nu\rangle}{\longmapsto} (M'_e, \mathcal{G}'_e) \wedge safe_{cB}(E, P_{S_a}, \nu, o_\nu) \implies$$

$$
\begin{array}{ll}
\textit{(a)} & csim\,(M'_d, (M'_e, \mathcal{G}'_e), par)\\
\textit{(b)} & \exists o_\kappa \in (\Omega_{S_c})^*, \mathcal{G}'_d \in \mathbb{G}_{S_c}.
\end{array}
$$

$$
\begin{array}{ll}
\textit{(b.i)} & D \overset{\langle\lfloor\kappa\rfloor, o_\kappa\rangle}{\longmapsto} (M'_d, \mathcal{G}'_d) \wedge safe(D, \langle\lfloor\kappa\rfloor, o_\kappa\rangle)\\
\textit{(b.ii)} & uinv((M'_d, \mathcal{G}'_d), par)\\
\textit{(b.iii)} & sinv((M'_d, \mathcal{G}'_d), (M'_e, \mathcal{G}'_e), par)
\end{array}
$$

As we see in the theorem, a possibly incomplete consistency block machine execution of S_c implements a complete consistency block computation of S_a. For units that have not reached the consistency points we do not require the simulation to hold. However, for such intermediate states the shared invariant is preserved.

We treat the incomplete blocks depending whether they contain $\mathcal{IO}$-steps or not. If such a block contains only local steps, it cannot change the shared memory and the ownership state. Therefore, we omit it in the abstract computation and represent it there as an empty step. Otherwise, we have to reflect the same changes in the abstract model. In this case, applying the general sequential simulation theorem we find a corresponding complete block of the abstract machine.

Before the proof of the theorem we need to state a few assumption additional to the sequential simulation theorem. The assumptions basically require that we can transfer the ownership safety to the concrete level, and when a particular unit is stepping, it preserves the simulation for other units and does not destroy their properties.

Assumption 2.1 (Safety Transfer with $sinv$ and $uinv$ Preservation for a Stepping Unit). *For any well-behaved complete consistency block computation $(D.M, \sigma)$ of a unit p starting in a* Cosmos *machine configuration D where the unit invariant is obeyed, and implementing a corresponding computation $(E.M, \tau)$ of the abstract machine such that*

- *the shared invariant between E and D holds,*
- *the abstract block computation is safe wrt. a given ownership annotation and the property P_{S_a},*
- *and the unit p in both machines is coupled with the sequential simulation relation before and after the computations,*

we can find an ownership annotation for σ, such that the annotated concrete computation is ownership safe and preserves the shared and unit invariants.

$$\forall D \in \mathbb{C}_{S_c}, d' \in \mathbb{M}_{S_c}, E, E' \in \mathbb{C}_{S_a}, \sigma \in \Theta^*_{S_c}, \tau \in \Theta^*_{S_a}, o_\tau \in \Omega^*_{S_a}, p \in \mathbb{N}_{nu}, par \in \mathcal{P}.$$

$$\begin{aligned}
&\textit{(i)} && D.M \overset{\sigma}{\longmapsto} d' \wedge blk(\sigma, p) \wedge one\mathcal{IO}(\sigma, \tau) \wedge wb(D.M, \sigma, par) \wedge \\
& && wf_p(d') \wedge uinv_p(D) \\
&\textit{(ii)} && E \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E' \wedge blk(\tau, p) \wedge P_{S_a}(E) \wedge safe_{P_{S_a}}(E, \langle \tau, o_\tau \rangle) \wedge \\
& && sc(E.M, \tau, par) \wedge wf_p(E'.M) \\
&\textit{(iii)} && csim_p(D.M, E, par) \wedge sinv(D, E, par) \wedge csim_p(d', E', par) \\
&\Longrightarrow \\
&\exists o_\sigma \in (\Omega_{S_c})^*, \mathcal{G}' \in \mathbb{G}_{S_c}. \\
& &&\textit{(i)} \quad D \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d', \mathcal{G}') \wedge safe(D, \langle \sigma, o_\sigma \rangle) \wedge uinv_p((d', \mathcal{G}')) \\
& &&\textit{(ii)} \quad sinv((d', \mathcal{G}'), E', par)
\end{aligned}$$

Note, that in contrast to the same assumption in [Bau14b], along with the introduction of the unit invariant, we additionally assume that the well-formedness predicates for both machines hold at the end of the consistency block computations. This extension is needed for the proof of the shared invariant that may guarantee well-formedness properties of the shared memories coupled between two machines. Moreover, the steps of the abstract machine have to be save wtr. P_{S_a}.

As we know according to Lemma 2.5 any ownership safe steps of a unit do not change other units' configurations. We use this fact to make the assumptions about the property and simulations preservation for other units present in the considered machines.

Assumption 2.2 (Preservation of Well-Formedness for Other Units of Abstract Machine). *The well-formedness predicate of the abstract machine for a given unit depends only on the local state of this unit and the memory covered by the shared invariant.*

$$\begin{aligned}
&\forall E, E' \in \mathbb{C}_{S_a}, p \in \mathbb{N}_{nu}, par \in \mathcal{P}.\\
&\quad \textit{(i)} \quad wf_p(E.M) \wedge E \approx_p E'\\
&\quad \textit{(ii)} \quad sinv(D', E', par)\\
&\quad \textit{(iii)} \quad oinv(E) \wedge P_{S_a}(E) \wedge oinv(E') \wedge P_{S_a}(E')\\
&\quad \Longrightarrow wf_p(E'.M)
\end{aligned}$$

This assumption differs from the same one stated in [Bau14b] by additional P_{S_a} and $oinv$ allowing to derive needed properties about owned and shared components and the memory in configurations E and E'.

Assumption 2.3 (Preservation of Well-Formedness for Other Units of Concrete Machine). *Analogously to Assumption 2.2, the well-formedness predicate of the concrete machine depends only on the local state of its units and the memory covered by the shared invariant.*

$$\begin{aligned}
&\forall D, D' \in \mathbb{C}_{S_c}, E, E' \in \mathbb{C}_{S_c}, p \in \mathbb{N}_{nu}, par \in \mathcal{P}.\\
&\quad \textit{(i)} \quad wf_p(D.M) \wedge D \approx_p D'\\
&\quad \textit{(ii)} \quad csim_p(D.M, E, par) \wedge sinv(D, E, par) \wedge sinv(D', E', par)\\
&\quad \textit{(iii)} \quad oinv(E) \wedge P_{S_a}(E) \wedge oinv(E') \wedge P_{S_a}(E')\\
&\quad \Longrightarrow wf_p(D'.M)
\end{aligned}$$

In contrast to [Bau14b] this assumption contains in its premises not only P_{S_a} and $oinv$ for the abstract machine, but also the simulation relation for the well-formed sequential machine configuration and the shared invariant.

Assumption 2.4 (Preservation of Simulation Relation and $uinv$ for Other Units). *The sequential simulation relation and the unit invariant for a unit p may depend on the unit's configuration, read-only, shared, and owned by p memories as well as its ownership state and the property P_{S_a} adjusting it.*

$$\begin{aligned}
&\forall D, D' \in \mathbb{C}_{S_c}, E, E' \in \mathbb{C}_{S_a}, p \in \mathbb{N}_{nu}, par \in \mathcal{P}.\\
&\quad \textit{(i)} \quad csim_p(D.M, E, par) \wedge uinv_p(D)\\
&\quad \textit{(ii)} \quad sinv(D, E, par) \wedge P_{S_a}(E) \wedge oinv(E) \wedge oinv(D)\\
&\quad \textit{(iii)} \quad sinv(D', E', par) \wedge P_{S_a}(E') \wedge oinv(E') \wedge oinv(D')\\
&\quad \textit{(iv)} \quad E \approx_p E' \wedge D \approx_p D'\\
&\quad \Longrightarrow\\
&\quad csim_p(D'.M, E', par) \wedge uinv_p(D')
\end{aligned}$$

Note again that this assumption differs from [Bau14b] similarly to Assumption 2.3 considered above and gives more flexibility for the simulation relation and properties to be instantiated.

Though it might be more clear to talk about the well-formedness and the unit invariant preservation for the concrete machine independently from the abstract machine, in this case one would have to transfer P_{S_a} explicitly to some properties on the concrete level in a general way, what is not required by the concurrent simulation theorem and would make it less intuitive. Such explicit property transfer will be shown separately later in this chapter. Moreover, in this thesis we will see how one can use the simulation and these assumptions for different systems.

Proof of Cosmos Model Simulation Theorem: We show the claim by induction on the length of consistency block schedule $n = |\kappa|$.

Base case ($n = 0$): For $|\kappa| = \varepsilon$ we set $\nu = \varepsilon$, $M'_e = E.M$, and $M'_d = D.M$. The claims (i)–(iii) obviously follow from the premises. For the claim (iv) we can only consider $o_\nu = \varepsilon$ and $\mathcal{G}'_e = E.\mathcal{G}$, what gives us the concurrent simulation relation for $D.M$ and E. Applying Assumption 2.1 for $d' = D$, $E' = E$, $\sigma = \tau = o_\tau = \varepsilon$ and the theorem premises we also easily conclude the rest of the theorem statement. Note, that in this case the ownership safety on the concrete level boils down to $\mathit{safe}(D, \varepsilon) = \mathit{oinv}(D.\mathcal{G})$ which follows directly from Assumption 2.1.

Induction hypothesis (IH): We assume the claim holds for any consistency block machine schedule $\bar{\kappa}$ with a fixed length $n = m$. Thus, there exists a simulated complete consistency block machine computation $(E.M, \bar{\nu})$ such that all the desired properties hold. Moreover, $(D.M, \bar{\kappa})$ is safe wrt. ownership.

Induction step ($m \rightarrow m + 1$): Assume that we are given a block machine computation $(D.M, \kappa)$ with $|\kappa| = m+1$. We denote the last block by $\omega = \kappa_{m+1}$ and the previous blocks by $\bar{\kappa} = \kappa[1 : m]$, i.e., $\kappa = \bar{\kappa}\omega$. Using the hypothesis of the theorem for κ for the induction hypothesis and setting up ν for the first m blocks such that $\nu[1 : m] = \bar{\nu}$ we get the m-step consistency block machine computations $(D.M, \bar{\kappa})$ and $(E.M, \bar{\nu})$ leading to the machine states $\bar{M}_d$ and $\bar{M}_e$ respectively and fulfilling the claims of the simulation theorem:

$$\mathcal{CP}sched_c(E.M, \nu[1 : m], par) \wedge \nu[1 : m] \overset{sch}{\sim} \kappa[1 : m] \tag{2.1}$$

$$E.M \overset{\nu[1:m]}{\longmapsto} \bar{M}_e \wedge \forall p \in \mathbb{N}_{nu}.\ wf_p(\bar{M}_e) \tag{2.2}$$

$$D.M \overset{\kappa[1:m]}{\longmapsto} \bar{M}_d \wedge wb(D.M, \lfloor\kappa[1 : m]\rfloor, par) \wedge \forall p \in U_{cp}(\bar{M}_d, par).\ wf_p(\bar{M}_d) \tag{2.3}$$

$$\forall o_{\bar{\nu}} \in (\Omega_{S_a})^*, \bar{\mathcal{G}}_e \in \mathbb{G}_{S_a}. \tag{2.4}$$

$$E \overset{\langle\lfloor\nu[1:m]\rfloor, o_{\bar{\nu}}\rangle}{\longmapsto} (\bar{M}_e, \bar{\mathcal{G}}_e) \wedge \mathit{safe}_{cB}(E, P_{S_a}, \nu[1 : m], o_{\bar{\nu}}) \implies$$

(a) $\mathit{csim}\big(\bar{M}_d, (\bar{M}_e, \bar{\mathcal{G}}_e), par\big)$

(b) $\exists o_{\bar{\kappa}} \in (\Omega_{S_c})^*, \bar{\mathcal{G}}_d \in \mathbb{G}_{S_c}.$

(b.i) $D \overset{\langle\lfloor\kappa[1:m]\rfloor, o_{\bar{\kappa}}\rangle}{\longmapsto} (\bar{M}_d, \bar{\mathcal{G}}_d) \wedge \mathit{safe}(D, \langle\lfloor\kappa[1 : m]\rfloor, o_{\bar{\kappa}}\rangle)$

(b.ii) $\mathit{uinv}((\bar{M}_d, \bar{\mathcal{G}}_d), par)$

(b.iii) $\mathit{sinv}((\bar{M}_d, \bar{\mathcal{G}}_d), (\bar{M}_e, \bar{\mathcal{G}}_e), par)$

Therefore we only need to take care of the probably incomplete last block ω which is non-empty by the hypothesis and being executed by some unit $q = \omega_1.s$. From the theorem hypothesis (vi) we know that the final machine state for the block computation $(D.M, \kappa)$ exists and we denote it d'. Moreover, $D.M \overset{\kappa[1:m]}{\longmapsto} \bar{M}_d$ leads to the intermediate state $\bar{M}_d$, so that we also have $\bar{M}_d \overset{\omega}{\longmapsto} d'$.

To apply the generalized sequential simulation theorem for the computation $(\bar{M}_d, \omega)$ and the abstract machine configuration $\bar{M}_e$ one needs to choose for the unit q the set of memory addresses excluded from its simulation relation and not accessed by the unit during this block execution. The choice of this memory region depends on the simulation relation holding at the beginning of execution of the block ω and addresses not accessed by the unit of the abstract machine for any consistency block starting from $\bar{\mathcal{M}}_e$.

From the theorem hypothesis (v) $safety_{cB}(E, P_{S_a}, par)$ and existing computation $E.M \overset{\nu[1:m]}{\longmapsto} \bar{M}_e$ we know that there exist ownership annotations $o_{\bar{\nu}}$ such that transitions $E \overset{\langle \lfloor \nu[1:m] \rfloor, o_{\bar{\nu}} \rangle}{\longmapsto} (\bar{M}_e, \bar{\mathcal{G}}_e)$ lead to corresponding ownership states $\bar{\mathcal{G}}_e$ and the simulation relation for both machines holds by induction hypothesis (2.4a):

$$\forall p \in U_{cp}(\bar{M}_d, par).\ csim_p(\bar{M}_d, (\bar{M}_e, \bar{\mathcal{G}}_e), par).$$

From the theorem hypothesis (vi) $\mathcal{CP}sched(\kappa[1:m]\omega, par)$ the unit q is obviously at a consistency point in configuration $\bar{M}_d$, therefore

$$sim_q\left(\bar{M}_d, \bar{M}_e, par, \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)\right)$$

So, the sequential simulation theorem applied to the considered case with $icm = \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)$ gets the form:

$$\begin{array}{ll}
(i) & wf_q(\bar{M}_d) \wedge wf_q(\bar{M}_e) \\
(ii) & sim_q(\bar{M}_d, \bar{M}_e, par, icm) \wedge \mathcal{CP}_q(\bar{M}_d, par) \wedge \mathcal{CP}_q(\bar{M}_e, par) \\
(iii) & \forall \lambda \in \Theta^*_{S_a}, \hat{e} \in \mathbb{M}_{S_a}.\ \bar{M}_e \overset{\lambda}{\longmapsto} \hat{e} \wedge blk(\lambda, p) \wedge \mathcal{CP}_p(\hat{e}, par) \implies \\
& \qquad\qquad noacc(\bar{M}_e, \lambda, icm) \wedge sc(\bar{M}_e, \lambda, par) \wedge wf_p(\hat{e}) \\
(iv) & \omega \neq \varepsilon \wedge blk(\omega, p) \wedge suit(\omega) \wedge comp(\bar{M}_d, \omega) \\
\implies &
\end{array}$$

$$\begin{array}{l}
\exists \sigma \in \Theta^*_{S_c}, d'' \in \mathbb{M}_{S_c}, \tau \in \Theta^*_{S_a}, e'' \in \mathbb{M}_{S_a}. \\
\begin{array}{ll}
(i) & \omega \rhd^{blk}_q \sigma \wedge suit(\sigma) \\
(ii) & blk(\tau, q) \wedge one\mathcal{IO}(\sigma, \tau) \\
(iii) & \bar{M}_d \overset{\sigma}{\longmapsto} d'' \wedge wb(\bar{M}_d, \sigma, par) \wedge wf_q(d'') \\
(iv) & \bar{M}_e \overset{\tau}{\longmapsto} e'' \wedge wf_q(e'') \\
(v) & sim_q(d'', e'', par, icm) \wedge \mathcal{CP}_q(d'', par) \wedge \mathcal{CP}_q(e'', par)
\end{array}
\end{array}$$

The premises $(i)-(ii)$ of the applied sequential simulation theorem follow directly from the equations (2.1) – (2.4) and the fact that the unit q is at a consistency point in $\bar{M}_d$ what is known from $\mathcal{CP}sched(\kappa, par)$. Since the simulation relation for q for the chosen icm holds as proven above and by (2.1) the unit is also at a consistency point in $\bar{M}_e$, one trivially gets (ii). We also easily conclude the premise (iv) from the hypothesis (vi) of the *Cosmos* model simulation theorem to be proven for $\kappa = \bar{\kappa}\omega$.

To prove premise (iii) we use the theorem hypothesis (v). So, from $safety_{cB}(E, P_{S_a}, par)$ for all such existing complete consistency block machine computations $(\bar{M}_e, \lambda)$ of unit q leading to $\hat{e}$ there exist ownership safe annotations o_λ such that computations $(\bar{M}_e, \bar{\mathcal{G}}_e) \overset{\langle \lambda, o_\lambda \rangle}{\longmapsto} (\hat{e}, \hat{\mathcal{G}}_e)$ till some $\hat{\mathcal{G}}_e$ are feasible. Due to the safe access and ownership transfer policy during $\langle \lambda, o_\lambda \rangle$ the unit q does not access local owned addresses $\overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)$ of other units and cannot modify them. This fact follows from the inductive application of Lemma 2.5 for the steps of $\langle \lambda, o_\lambda \rangle$. Therefore, for such $(\bar{M}_e, \lambda)$ one guarantees that $noacc(\bar{M}_e, \lambda, \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q))$ holds. From $scwf_{cB}(E.M, par)$ we easily conclude the rest of the premise (v).

Now we can use the claim of the applied sequential simulation theorem. We extend the block ω by steps ω' in order to obtain $\sigma = \omega\omega'$. Thus the sequential simulation guarantees for the unit q the existence of computations $\bar{M}_e \overset{\tau}{\longmapsto} e''$ and $\bar{M}_d \overset{\omega}{\longmapsto} d' \overset{\omega'}{\longmapsto} d''$ such that the simulation relation excluding $\overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)$ holds. Moreover, we choose ownership annotations o_τ such that computations $(\bar{M}_e, \bar{\mathcal{G}}_e) \overset{\langle \tau, o_\tau \rangle}{\longmapsto} (e'', \mathcal{G}''_e)$ for corresponding $\mathcal{G}''_e$ are safe wrt. the ownership and the property P_{S_a}, i.e., $P_{S_a}((\bar{M}_e, \bar{\mathcal{G}}_e)) \wedge \mathit{safe}_{P_{S_a}}((\bar{M}_e, \bar{\mathcal{G}}_e), \langle \tau, o_\tau \rangle)$ holds.

First, we prove our theorem for $\nu[1:m]\tau$ and $\kappa[1:m]\sigma$. As the second step, we will look at ω' because the theorem is originally stated for non-extended ω.

Using equation (2.1) we easily conclude claim (i) of the theorem, namely

$$\mathcal{CP}sched_c(E.M, \nu[1:m]\tau, par) \wedge \nu[1:m]\tau \overset{sch}{\sim} \kappa[1:m]\sigma,$$

because τ is a complete consistency block of the unit q, $one\mathcal{IO}(\sigma, \tau)$ holds, all units were at consistency points in $\bar{M}_e$, and predicate $\mathcal{CP}$ depends only on a unit's machine configuration and read-only memory that were not changed for units $p \neq q$ during safe steps of q in τ.

By now, from equations (2.2) – (2.3) and the sequential simulation theorem we also have

$$\begin{array}{lll} D.M \overset{\kappa[1:m]\sigma}{\longmapsto} d'' & wf_q(d'') & wb(D.M, \lfloor \kappa[1:m]\sigma \rfloor, par) \\ E.M \overset{\nu[1:m]\tau}{\longmapsto} e'' & wf_q(e'') & sim_q(d'', e'', par, \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)) \end{array}$$

Moreover, as mentioned before, by Lemma 2.5 for the safe computation $(\bar{M}_e, \bar{\mathcal{G}}_e) \overset{\langle \tau, o_\tau \rangle}{\longmapsto} (e'', \mathcal{G}''_e)$ we have $\overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q) = \overline{\mathcal{SO}}(\mathcal{G}''_e, q)$ and conclude $sim_q(d'', e'', par, \overline{\mathcal{SO}}(\mathcal{G}''_e, q))$.

To prove the rest of the claims we first apply Assumption 2.1 for τ and σ:

(i) $\bar{M}_d \overset{\sigma}{\longmapsto} d'' \wedge blk(\sigma, q) \wedge one\mathcal{IO}(\sigma, \tau) \wedge wb(\bar{M}_d, \sigma, par) \wedge wf_q(d'') \wedge uinv_q((\bar{M}_d, \bar{\mathcal{G}}_d))$

(ii) $(\bar{M}_e, \bar{\mathcal{G}}_e) \overset{\langle \tau, o_\tau \rangle}{\longmapsto} (e'', \mathcal{G}''_e) \wedge blk(\tau, q) \wedge P_{S_a}((\bar{M}_e, \bar{\mathcal{G}}_e)) \wedge$
$\mathit{safe}_{P_{S_a}}((\bar{M}_e, \bar{\mathcal{G}}_e), \langle \tau, o_\tau \rangle) \wedge sc(\bar{M}_e, \tau, par) \wedge wf_q(e'')$

(iii) $sim_q\left(\bar{M}_d, \bar{M}_e, par, \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, q)\right) \wedge sinv((\bar{M}_d, \bar{\mathcal{G}}_d), (\bar{M}_e, \bar{\mathcal{G}}_e), par) \wedge$
$sim_q(d'', e'', par, \overline{\mathcal{SO}}(\mathcal{G}''_e, q))$

$\Longrightarrow$

$\exists o_\sigma \in (\Omega_{S_c})^*, \mathcal{G}''_d \in \mathbb{G}_{S_c}.$

(i) $(\bar{M}_d, \bar{\mathcal{G}}_d) \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d'', \mathcal{G}''_d) \wedge \mathit{safe}((\bar{M}_d, \bar{\mathcal{G}}_d), \langle \sigma, o_\sigma \rangle) \wedge uinv_q((d'', \mathcal{G}''_d))$

(ii) $sinv((d'', \mathcal{G}''_d), (e'', \mathcal{G}''_e), par)$

Obviously, from (2.4) the shared invariant holds and we have $uinv_q((\bar{M}_d, \bar{\mathcal{G}}_d))$ because the unit q is in a consistency point. All the other premises follow directly from the argumentation above. From now we can consider the ownership safe computation $(\bar{M}_d, \bar{\mathcal{G}}_d) \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d'', \mathcal{G}''_d)$ with the ownership annotation o_σ. Moreover, we may concatenate it with $o_{\bar{\kappa}}$ so that we easily get

$$D \overset{\langle \lfloor \bar{\kappa}\sigma \rfloor, o_{\bar{\kappa}} o_\sigma \rangle}{\longmapsto} (d'', \mathcal{G}''_d) \wedge \mathit{safe}(D, \langle \lfloor \bar{\kappa}\sigma \rfloor, o_{\bar{\kappa}} o_\sigma \rangle)$$

The rest to be proven is the preservation of well-formedness, the unit invariant and the sequential simulation relation for other units $p \neq q$.

Applying Lemma 2.5 for the steps of $(\bar{M}_e, \bar{\mathcal{G}}_e) \overset{\langle \tau, o_\tau \rangle}{\longmapsto} (e'', \mathcal{G}''_e)$ and $(\bar{M}_d, \bar{\mathcal{G}}_d) \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d'', \mathcal{G}''_d)$ we get

$$\forall p \neq q.\ (\bar{M}_e, \bar{\mathcal{G}}_e) \approx_p (e'', \mathcal{G}''_e) \wedge (\bar{M}_d, \bar{\mathcal{G}}_d) \approx_p (d'', \mathcal{G}''_d)$$

From (2.4) (a) and $(b.ii)$ we know

$$\forall p \in U_{cp}(\bar{M}_d, par).\ sim_p\left(\bar{M}_d, \bar{M}_e, par, \overline{\mathcal{SO}}(\bar{\mathcal{G}}_e, p)\right) \wedge uinv_p((\bar{M}_d, \bar{\mathcal{G}}_d))$$

Naturally, the ownership states $\mathcal{G}''_d$ and $\mathcal{G}''_e$ support the ownership invariant *oinv* by Lemma 2.1. Moreover, the shared invariant holds at the beginning and the end of the considered computations.

Therefore, since we have $\forall p \in \mathbb{N}_{nu}.\ wf_p(\bar{M}_e)$ and $\forall p \in U_{cp}(\bar{M}_d, par).\ wf_p(\bar{M}_d)$ in equations (2.2) – (2.3), we use Assumptions 2.2 and 2.3 for both machines in order to conclude the well-formedness of units $p \neq q$ in e'' and d''.

Finally, we show that the simulation relations $sim_p\left(d'', e'', par, \overline{\mathcal{SO}}(\mathcal{G}''_e, p)\right)$ and unit invariants $uinv_p((d'', \mathcal{G}''_d))$ for other units $p \in U_{cp}(d'', par)$ with $p \neq q$ are preserved. Applying Assumption 2.4 for each unit p we finish the proof and conclude that the theorem claims hold for $\nu[1:m]\tau$ and $\kappa[1:m]\sigma$.

Now, for the computation $\bar{M}_d \overset{\omega}{\longmapsto} d' \overset{\omega'}{\longmapsto} d''$ we split cases on ω' and prove the theorem for the computation $(D.M, \kappa)$ with $\kappa_{m+1} = \omega$ leading to configuration $M'_d = d'$.

Case 1: $\omega' = \varepsilon$. In this case ω is already complete, $d' = d''$, and we just set $\nu = \bar{\nu}\tau$, $M'_e = e''$ for the theorem to hold.

Case 2: $\omega' \neq \varepsilon \wedge \omega|_{io} \neq \varepsilon$. The consistency block ω contains already an $\mathcal{IO}$-operation and is incomplete. In this case we include τ into the abstract model computation $(E.M, \nu)$ leading to $M'_e = e''$ because one has to maintain the shared invariant.[1]

Since prefixes of sequences cannot contain more $\mathcal{IO}$-points than the original ones, the predicate $one\mathcal{IO}(\omega)$ holds, and we show claim (i) of the theorem as in the previous case.

Obviously, from the sequential simulation theorem we already know $wf_q(e'')$. For the unit q of the concrete machine we do not have to guarantee the well-formedness and the simulation relation because it has not reached a consistency point after the computation $(D.M, k)$. From $wb(D.M, \lfloor\kappa[1:m]\sigma\rfloor, par)$ proven before we also get $wb(D.M, \lfloor\kappa\rfloor, par)$.

Having $safe((\bar{M}_d, \bar{\mathcal{G}}_d), \langle \sigma, o_\sigma \rangle)$ we can split $o_\sigma = o_\omega o'_\omega$ such that $|o_\omega| = |\omega|$, $|o'_\omega| = |\omega'|$. Therefore, it follows that for some intermediate ownership state $\mathcal{G}'_d$ one has

$$safe((\bar{M}_d, \bar{\mathcal{G}}_d), \langle \omega, o_\omega \rangle) \qquad safe((d', \mathcal{G}'_d), \langle \omega', o'_\omega \rangle)$$

We also know $\omega'|_{io} = \varepsilon$ because of $one\mathcal{IO}(\omega\omega')$ and $\omega|_{io} \neq \varepsilon$. Applying Lemma 2.9 for the computation $(d', \mathcal{G}'_d) \overset{\langle \omega', o'_\omega \rangle}{\longmapsto} (d'', \mathcal{G}''_d)$ with $sinv((d'', \mathcal{G}''_d), (e'', \mathcal{G}''_e), par)$ we get

$$sinv((d', \mathcal{G}'_d), (e'', \mathcal{G}''_e), par)$$

Now, similarly to the previous case it is easy to apply Assumptions 2.2–2.4 to show the preservation of the well-formedness, simulation relations, unit invariants for the units $p \neq q$ still being in consistency points in e'' and d'.

Case 3: $\omega' \neq \varepsilon \wedge \omega|_{io} = \varepsilon$. The incomplete consistency block ω contains only local steps of the unit q. In this case we cannot include τ into the simulated computation $(E.M, \nu)$ because it could

[1]As a special case for $\omega|_{io} \neq \varepsilon$ one could consider $\omega|_{ot} \neq \varepsilon$. According to $one\mathcal{IO}(\omega\omega', \tau)$ we allow $\tau|_{io} = \tau|_{ot} = \varepsilon$. Therefore, for $\tau \neq \varepsilon$ we could have set $\nu_{m+1} = \varepsilon$ instead of τ. However, since it would require more argumentation based on technical lemmas from [Bau14b] not covered here, we stick to the shorter proof with $\nu_{m+1} = \tau$.

contain an $\mathcal{IO}$-step introduced only in ω' that is not executed, and change the shared memory state and the ownership so that the shared invariant would not hold after the computations. Thus, we set $\nu_{m+1} = \varepsilon$ and obviously $M'_e = \bar{M}_e$, $\mathcal{G}'_e = \bar{\mathcal{G}}_e$.

Equation (2.2) gives us $\forall p \in \mathbb{N}_{nu}.\ wf_p(\bar{M}_e)$. Moreover, from $safe((\bar{M}_d, \bar{\mathcal{G}}_d), \langle \omega, o_\omega \rangle)$, $\omega|_{io} = \varepsilon$, $sinv((\bar{M}_d, \bar{\mathcal{G}}_d), (\bar{M}_e, \bar{\mathcal{G}}_e), par)$ and Lemma 2.9 we also have

$$sinv((d', \mathcal{G}'_d), (\bar{M}_e, \bar{\mathcal{G}}_e), par)$$

The argumentation about the remaining parts of the claims is analogous to the previous case. This finishes the proof of the theorem.

□

2.4 Order Reduction on the Concrete Level

To apply the *Cosmos* model simulation theorem on a arbitrary interleaved sequence of the concrete model computations starting from D, one has to apply first the order reduction from the arbitrary schedule to the block schedule that will allow us to guarantee that the ownership safety as well as any other properties $P_{S_c} : \mathbb{C}_{S_c} \to \mathbb{B}$ verified on the block schedule hold for the given arbitrary interleaved schedule.

Remember that the order reduction theorem states the safety transfer from safe $\mathcal{IP}$-schedules to arbitrary interleaved *Cosmos* machine schedules, and requires in the hypotheses that all $\mathcal{IP}$-schedule computations leaving D are safe and obey the $\mathcal{IOIP}$ condition. To prove these we would like to use the *Cosmos* model simulation theorem. However, the theorem guarantees such properties only for all block schedules that are suitable for the simulation. This makes the direct application of the order reduction theorem impossible. Instead, we have to state the analogous order reduction theorem but for suitable schedules. We proceed in the way shown by Baumann.

2.4.1 Order Reduction for Suitable Schedules

We redefine the predicates from Section 2.2:

Definition 2.46 (Verified *Cosmos* model for Suitable Schedules). For $\theta \in \Theta^*_{S_c}$, $o \in \Omega^*_{S_c}$, configuration $D \in \mathbb{C}_{S_c}$ and safety property P_{S_c} the predicates

$$\begin{aligned}
safety(D, P_{S_c}, suit) &\overset{def}{\equiv} \forall \theta.\ suit(\theta) \wedge comp(D.M, \theta) \implies \exists o \in \Omega^*_{S_c}.\ safe_{P_{S_c}}(D, \langle \theta, o \rangle) \\
safety_{\mathcal{IP}}(D, P_{S_c}, suit) &\overset{def}{\equiv} \forall \theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \implies \\
&\qquad \exists o \in \Omega^*_{S}.\ safe_{P_{S_c}}(D, \langle \theta, o \rangle) \\
\mathcal{IOIP}_{\mathcal{IP}}(D, suit) &\overset{def}{\equiv} \forall \theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \implies \mathcal{IOIP}(\theta)
\end{aligned}$$

denote the safety and $\mathcal{IOIP}$ condition for all $\mathcal{IP}$-schedules suitable for the simulation.

Being based on the previously shown lemmas from Section 2.2 and a property from [Bau14b] claiming that the equivalent reordering preserves the suitability, Baumann managed to show a stronger reduction theorem that allows to transfer safety properties from a subset of $\mathcal{IP}$-schedules suitable for *Cosmos* model simulation to suitable arbitrarily interleaved schedules.

Theorem 2.5 (**$\mathcal{IP}$-Schedule Order Reduction for Suitable Schedules**). *Given a simulation framework $R_{S_c}^{S_a}$ and a* Cosmos *model configuration $D \in \mathbb{C}_{S_c}$ 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_{S_c}. Moreover, all suitable $\mathcal{IP}$-schedule computations running out of D obey the $\mathcal{IOIP}$ condition. Then ownership safety and P_{S_c} hold on all computations with a schedule suitable for simulation that starts in D.*

$$safety_{\mathcal{IP}}(D, P_{S_c}, suit) \wedge \mathcal{IOIP}_{\mathcal{IP}}(D, suit) \implies safety(D, P_{S_c}, suit)$$

2.4.2 Applying the Cosmos Model Simulation Theorem

Since we can justify verification of ownership safety and other properties of the *Cosmos* machine states for suitable schedules, we would like to see now whether not only the ownership safety but also the well-formedness and well-behaviour of the concrete machine guaranteed by the *Cosmos* model simulation theorem for suitable block schedule computations can be transferred to any arbitrary interleaved suitable step sequences.

In comparison to the well-formedness and unit invarints that are properties of a machine state, the well-behaviour is a property of steps. Baumann resolved this issue by extending the considered concrete *Cosmos* model S_c with some history information recording whether the well-behaviour is violated or not, and showed that using the *Cosmos* model simulation theorem for the original abstract and concrete models as well the aforementioned model extension one guarantees that the proven well-behaviour transfers to arbitrary interleaved computations of the extended concrete machine.

However, such an extension implies some restrictions on the step properties because of the specific semantics of *Cosmos* machines. Recall, that the *Cosmos* machine transition function is based on the unit's transition function that takes as a parameter a portion of memory restricted to the $reads$-set. If in any step of the machine we would recompute the history flag about the well-behaviour, it would be possible only if the well-behaviour does not talk about the memory content at addresses not belonging to the $reads$-set. This fact was not mentioned in the original work [Bau14b] and can be easily noticed if one looks closer how such an extension could be made.

To see this we extend in detail our concrete model S_c to S'_c in the way introduced in Baumann's work:

- For components $X \in \{\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{E}\}$ we have

$$S'_c.X = S_c.X$$

- The unit configuration is extended with the history information corresponding to the well-behaviour.

$$S'_c.\mathcal{U} = (k \in S_c.\mathcal{U}, wb \in \mathbb{B})$$

  The flag wb is computed in every step of the machine and initially will be equal to 1.

- The $reads$-set is equal to the one from the original model

$$S'_c.reads(u, m, in) = S_c.reads(u.k, m, in)$$

- For components $Y \in \{\mathcal{IP}, \mathcal{IO}, \mathcal{OT}\}$ we have

$$S'_c.Y(u, m, in) = S_c.Y(u.k, m, in)$$

- The unit's transition function additionally records the history of well-behaviour

$$S'_c.\delta(u,m,in) = \begin{cases} ((k',wb'),m') & : \; S_c.\delta(u.k,m,in) = (k',m') \\ \mathit{undefined} & : \; \text{otherwise} \end{cases}$$

  where $wb' = u.wb \wedge R^{S_a}_{S_c}.wb\,((u,\lceil m\rceil),in,par)$. Since m is the memory restricted by the $reads$-set and the predicate wb might state properties on values at other addresses, one has to restrict such properties only to the $reads$-set.

Hence, to be able to transfer the well-behaviour to arbitrary interleaving schedules of the extended machine computation we make the following assumption.

Assumption 2.5 (Well-Behaviour Restriction). *For a simulation framework $R^{S_a}_{S_c}$, a unit configuration $u \in \mathcal{U}_c$, memory $m : \mathcal{A}_c \to \mathcal{V}_c$, an input $in \in \mathcal{E}_c$ and a simulation parameter $par \in \mathcal{P}$ we require that the well-behaviour property of a computation step does not depend on memory content outside the $reads$-set $R = S_c.reads(u,m,in)$ of the concrete model S_c:*

$$R^{S_a}_{S_c}.wb\,((u,m),in,par) \iff \Big(\forall m'.\, m'|_R = m|_R \implies R^{S_a}_{S_c}.wb\,((u,m'),in,par)\Big)$$

where $m' : \mathcal{A}_c \to \mathcal{V}_c$.

Given the extended *Cosmos* model S'_d, the transformation of the simulation framework $R^{S_a}_{S_c}$, $sinv^{S_a}_{S_c}$, and $uinv^{S_a}_{S_c}$ into the framework for the abstract and extended concrete machines $R^{S_a}_{S'_c}$ and the corresponding predicates for S'_c is a bookkeeping and we skip the details here.

Now, for a configuration $D \in \mathbb{C}_{S'_c}$ we can define a *Cosmos* machine safety property denoting the well-behaviour in the past before D:

$$W(D) \overset{def}{\equiv} \forall p \in \mathbb{N}_{nu}.\; D.u_p.wb$$

Definition 2.47 (Simulation Hypothesis). Finally, we introduce the predicate $simh$ to denote the *simulation hypotheses* of the concurrent simulation theorem for the framework $R^{S_a}_{S'_c}$, start configurations $D \in \mathbb{C}_{S'_c}$, $E \in \mathbb{C}_{S_a}$, and a simulation parameter $par \in \mathcal{P}$. Moreover, additionally to the safety property P_{S_a} needed for the concurrent simulation in Theorem 2.4, we include any other property $P'_{S_a} : \mathbb{C}_{S_a} \to \mathbb{B}$ involved into the property transfer from the abstract machine to the concrete one (considered below in Section 2.5).

Using the shorthand

$$\widetilde{P}_{S_a}(e) \equiv P_{S_a}(e) \wedge P'_{S_a}(e)$$

for $e \in \mathbb{C}_{S_a}$ we formally define

$$\begin{aligned} simh(D,E,P'_{S_a},par) \overset{def}{\equiv} \quad & \textit{(i)} && uinv(D,par) \wedge \forall p \in U_{cp}(D.M,par).\; wf_p(D.M) \\ & \textit{(ii)} && \exists p \in \mathbb{N}_{nu}.\; \mathcal{CP}_p(D.M,par) \wedge W(D) \\ & \textit{(iii)} && csim(D.M,E,par) \wedge sinv(D,E,par) \\ & \textit{(iv)} && safety_{cB}(E,\widetilde{P}_{S_a},par) \wedge scwf_{cB}(E.M,par) \\ & \textit{(v)} && \mathcal{IPCP}(R^{S_e}_{S'_c},par) \end{aligned}$$

Baumann in his work proved two corollaries of the concurrent simulation theorem that allow us to apply the order reduction on the concrete level.

Corollary 2.2 (Simulating $\mathcal{IP}$-Schedules are Safe and Well-Behaved). *Assuming a pair of simulating* Cosmos *machine configurations $D \in \mathbb{C}_{S'_d}$ and $E \in \mathbb{C}_{S_e}$, if the simulation hypotheses hold for the simulation framework $R^{S_e}_{S'_c}$, the corresponding predicates, and some parameter $par \in \mathcal{P}$, then all suitable $\mathcal{IP}$-schedule computations running out of D are ownership-safe and well-behaved.*

$$\forall D, E, P'_{S_a}, par.\ simh(D, E, P'_{S_a}, par) \implies safety_{\mathcal{IP}}(D, W, suit)$$

Corollary 2.3 ($\mathcal{IOIP}$ Assumption for Simulating $\mathcal{IP}$-Schedules). *Given a* Cosmos *machine simulation constrained as in the previous corollary with start configuration D, then every $\mathcal{IP}$-schedule that is suitable for the simulation and running out of D fulfills the $\mathcal{IOIP}$ condition.*

$$\forall D, E, P'_{S_a}, par.\ simh(D, E, P'_{S_a}, par) \implies \mathcal{IOIP}_{\mathcal{IP}}(D, suit)$$

Therefore, applying these corollaries and Theorem 2.5 we easily reach our goal.

Corollary 2.4 (Ownership Transfer for Simulating Block Machines). *Assuming a pair of simulating* Cosmos *machine configurations $D \in \mathbb{C}_{S'_c}$ and $E \in \mathbb{C}_{S_e}$, if the simulation hypotheses hold for the simulation framework $R^{S_e}_{S'_c}$, corresponding predicates, and some parameter $par \in \mathcal{P}$, then all suitable computations running out of D are ownership-safe and well-behaved.*

$$\forall D, E, P'_{S_a}, par.\ simh(D, E, P'_{S_a}, par) \implies safety(D, W, suit)$$

Note, that among the well-behavior we could also easily transfer the well-formedness and unit invariant in the same manner. Therefore, we have shown that verification of considered properties for block schedules is justified. In other words, the order reduction applied on the concrete level does not influence these properties.

2.5 Property Transfer From Abstract to Concrete Level

For constructing concurrent model stacks for pervasive verification one often needs to transfer the software conditions and safety policy from the abstract level down to the concrete machine computations which can be treated as an abstract level of another underlying simulation layer. For instance, we have already seen that the required well-behavior can follow from the software conditions of the abstract model and guaranteed by the sequential simulation theorem. Now, having the *Cosmos* model simulation theorem between S_e and S'_c with the framework $R^{S_a}_{S'_c}$, invariants $sinv^{S_a}_{S_c}$, $uinv^{S_a}_{S_c}$, the property P_{S_a} as well as any additional property P'_{S_a} of the abstract machine, we can also show how we transfer $\widetilde{P}_{S_a}$ (see Definition 2.47) to some properties on the concrete level. To reach this goal we proceed here with an approach suggested by Baumann in his work and based on ideas from [CL98].

Since generally we are dealing with different machines S_a and S'_c, we cannot translate one-to-one the abstract property $\widetilde{P}_{S_a}$ into some property of the concrete machine. Moreover, we require that $\widetilde{P}_{S_a}$ holds at the beginning and the end of all complete consistency block executions for the abstract machine. To be able to transfer the aforementioned property to the concrete level, one needs to couple the configurations of both machines by the simulation relation. However, it is not possible in the context of the incomplete consistency block execution. For the units not being in consistency points we only require the shared invariant to hold. This has an influence on the sort of properties that could be transferred. Baumann suggested to distinguish between local and global properties and introduced a requirement that a property to be transferred must be *divisible*.

Definition 2.48 (Divisible *Cosmos* machine Safety Property). We say that $P : \mathbb{C}_S \to \mathbb{B}$ is a divisible safety property of a *Cosmos* model S iff for any configuration $C \in \mathbb{C}_S$ it has the following structure

$$P(C) = P_g(C) \wedge \forall p \in \mathbb{N}_{nu}.\ P_l(C, p)$$

where P_g is a *global property* which depends only on shared resources and the ownership state, and P_l constitutes *local properties* for each unit of the system. Consequently they are constrained as shown below for any $C, C' \in \mathbb{C}_S$:

$$C \overset{s}{\sim} C' \wedge C \overset{o}{\sim} C' \implies P_g(C) = P_g(C')$$
$$\forall p.\ C \approx_p C' \implies P_l(C, p) = P_l(C', p)$$

Therefore, we have to translate global properties in all intermediate configurations and for proving them we can use the shared invariant. In turn, local properties can only be used in consistency points where the simulation relation holds.

Definition 2.49 (Simulated *Cosmos* machine Property). Let $\widetilde{P}_{S_a}$ from Definition 2.47 be a divisible *Cosmos* machine safety property on $\mathbb{C}_{S_a}$ in the simulation between two *Cosmos* models S_a and S'_c. Then for a given simulation parameter $par \in \mathcal{P}$ a simulated divisible *Cosmos* machine property $\hat{Q}[\widetilde{P}_{S_a}, par] : \mathbb{C}_{S'_c} \to \mathbb{B}$ can be derived by solving the following formula[2], which states for any configuration $E \in \mathbb{C}_{S_a}$ being completely consistent with $D \in \mathbb{C}_{S'_c}$ that $\hat{Q}[\widetilde{P}_{S_a}, par]$ holds in D iff $\widetilde{P}_{S_a}$ holds in E.

$$\forall D, E.\ (sinv(D, E, par) \wedge \forall p.\ csim_p(D.M, E, par)) \implies \hat{Q}[\widetilde{P}_{S_a}, par](D) = \widetilde{P}_{S_a}(E)$$

Consequently the following constraints must hold for $\hat{Q}[\widetilde{P}_{S_a}, par]$:

$$\forall D, E.\ P_g(E) \wedge sinv(D, E, par) \implies \hat{Q}[\widetilde{P}_{S_a}, par]_g(D)$$
$$\forall D, E, p.\ P_l(E, p) \wedge csim_p(D.M, E, par) \implies \hat{Q}[\widetilde{P}_{S_a}, par]_l(D, p)$$

where P_g, P_l, $\hat{Q}[\widetilde{P}_{S_a}, par]_g$, $\hat{Q}[\widetilde{P}_{S_a}, par]_l$ are global and local properties of $\widetilde{P}_{S_a}$ and $\hat{Q}[\widetilde{P}_{S_a}, par]$ respectively according to the Definition 2.48.

Additionally, we relax the definition of the simulated properties to so called *incompletely simulated* Cosmos *machine properties* because not all units could be in their consistency points.

Definition 2.50 (Incompletely Simulated *Cosmos* machine Property). Given the divisible simulated *Cosmos* machine property $\hat{Q}[\widetilde{P}_{S_a}, par]$ from Definition 2.49 we define an incompletely simulated *Cosmos* machine property $Q[\widetilde{P}_{S_a}, par] : \mathbb{C}_{S'_c} \to \mathbb{B}$ in the following way:

$$Q[\widetilde{P}_{S_a}, par](D) \overset{def}{\equiv} \hat{Q}[\widetilde{P}_{S_a}, par]_g(D) \wedge \forall p \in U_{cp}(D.M, par).\ \hat{Q}[\widetilde{P}_{S_a}, par]_l(D, p)$$

Finally, we can show that $\widetilde{P}_{S_a}$ is translated to the incomplete simulated property maintained on the concrete level. This fact is stated in the following theorem proven in [Bau14b].

[2] i.e., by finding the translated predicate $\hat{Q}$ satisfying the formula

Theorem 2.6 (Simulated Safety Property Transfer). *Given are the simulation framework $R_{S'_c}^{S_e}$ and all the corresponding predicates including the abstract machine safety properties P_{S_a} and P'_{S_a} such that the* Cosmos *model simulation theorem holds for start configurations $D \in \mathbb{C}_{S'_c}$, $E \in \mathbb{C}_{S_a}$, and the simulation parameter $par \in \mathcal{P}$. If the simulation hypothesis are fulfilled and the property $\widetilde{P}_{S_a}$ (given in Definition 2.47 and verified for all complete consistency block machine computations starting in E) translates into the incompletely simulated* Cosmos *machine property $Q[\widetilde{P}_{S_a}, par]$*[3]*, then any suitable* Cosmos *machine schedule leaving D is safe wrt. the ownership, $Q[\widetilde{P}_{S_a}, par]$ holds for all reachable configurations, and all implementing computations are well-behaved.*

Using the shorthand

$$P_{S_c}(d) \equiv W(d) \wedge Q[\widetilde{P}_{S_a}, par](d)$$

with $d \in \mathbb{C}_{S'_c}$, we formally state the theorem as

$$\forall D, E, P'_{S_a}, par.\ simh(D, E, P'_{S_a}, par) \implies safety(D, P_{S_c}, suit)$$

Moreover, it is also practical to know that the simulated properties hold completely for complete consistency block machine computations.

Corollary 2.5 (Complete Simulated Property Transfer). *For complete consistency block machine computations on the concrete level, Theorem 2.6 allows to transfer the property $\widetilde{P}_{S_a}$ into the completely simulated property $\hat{Q}[\widetilde{P}_{S_a}, par], suit)$ if it can be derived. Formally, for $\hat{P}_{S_c}(d) \equiv W(d) \wedge \hat{Q}[\widetilde{P}_{S_a}, par](d)$ with $d \in \mathbb{C}_{S'_c}$, we state*

$$\forall D, E, P'_{S_a}, par.\ simh(D, E, P'_{S_a}, par) \implies safety_{cB}(D, \hat{P}_{S_c}, suit)$$

[3]with the properties of $\hat{Q}$ stated above

3 Formal Model of MIPS-86

In the thesis we consider a multi-core MIPS-86 model defined by Sabine Schmaltz in her doctoral dissertation [Sch13]. The model is based on a sequential MIPS instruction set architecture (ISA) from [KMP14] and extended with components typical for x86-64 architectures: store buffers (SB), memory management units (MMU) with translation lookaside buffers (TLB), devices, local and I/O advanced programmable interrupt controllers (APIC) serving for the interprocessor interrupts (IPI) and device interrupts delivery.

Since in the scope of the thesis we are not interested in the communication between the processors via IPIs as well as device accesses, we stick to a simplified version containing only two of the aforementioned components and supporting the external interrupts. Using the definitions from [Sch13], we first introduce the single core MIPS-86 model, and then provide the semantics for the corresponding multi-core machine. We also adapt the original definitions where it is necessary to match our concurrent simulation theory from the previous chapter. Moreover, we extend the ISA with an instruction from [Kov13] which exists in x86-64 architectures and allows a programmer easier to guarantee the sequential consistent memory in the presence of store buffers.

3.1 Instruction Set Architecture Overview

An overview of the multi-core MIPS-86 model used in this thesis is depicted on Figure 3.1.

The abstraction of the sequential processor cores features atomic fetch-and-execute transitions which can be justified only in the absence of self-modifying code. Namely, if an instruction being fetched from the memory by a core cannot be changed by other processors present in the model, then we can reorder the fetch cycle in a way that it appears exactly before the instruction execution.

The processor core communicates with its TLB performing and caching address translations used by the processor to establish a desired virtual memory abstraction. The SB caches memory write requests of the core and provides data in case of read requests if possible. Both components can also perform their steps independently from the core. For instance, the store buffer can commit its pending stores to the memory. Note, that the memory depicted on Figure 3.1 and considered in our multi-core model is already an abstraction of a cache shared memory system implemented and proven to be sequential consistent in [KMP14].

As suggested by Schmaltz we provide the MIPS-86 model with an order of processor component steps unknown to the programmer. Such a non-deterministic behavior is modeled by a deterministic automaton taking among its inputs some information specifying which component performs a particular transition. We will use this approach not only for a single core model semantics, but also for the multi-core machine where the order of processor executions is determined by their indices.

If a few components of the model perform synchronous steps as a response to an action of a certain component, we call these steps *passive* in comparison to the *active* step triggering them.

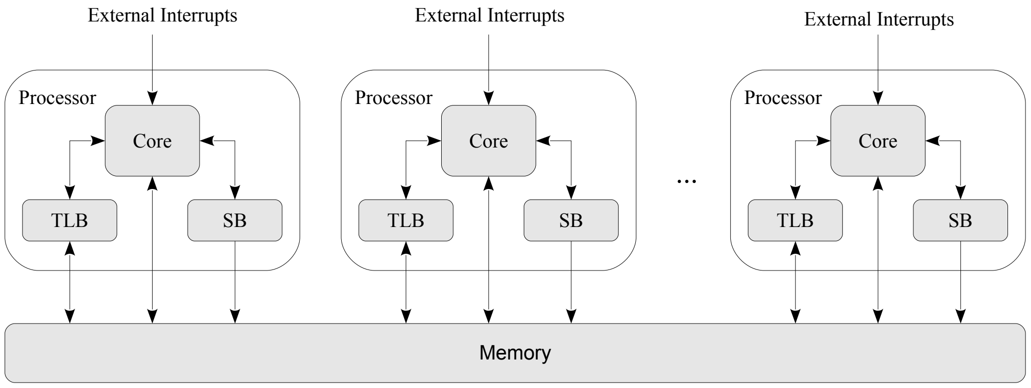

Figure 3.1: Overview of MIPS-86 Components.

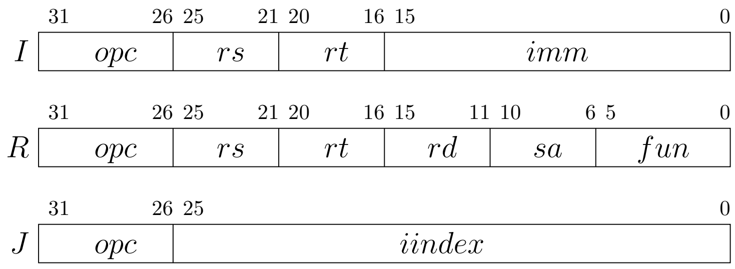

Figure 3.2: Types and Fields of MIPS instructions

A typical passive component of a step is the memory changing its configuration as a result of write accesses from the store buffer.

3.1.1 Instruction Set

The MIPS-86 ISA provides instructions of I-, J, and R-types. The I-type instructions operate with two registers of the core and a so called *immediate constant* whereas the R-type allows to make operations involving three registers. The J-type instructions are used for absolute jumps in the assembly code. Any instruction is represented as a 32-bit binary word and has its own layout depending on the type (see Figure 3.2). The fields rs, rt, and rd stand for the source, target, and destination registers respectively. Another field sa is a shift amount used for shift operations.

An overview of available instructions is given in Tables 3.1– 3.3. Note, that the exact semantics of the instruction execution will be considered later when we define the transition function of the single core MIPS-86. In the tables you find only approximate effects. For example, m denotes the full memory system including also the store buffers, and the table description does not show which component is involved into a word memory access. Moreover, the instructions which mnemonic ends with u perform unsigned arithmetic with binary numbers. A particular case that is worth paying attention to is the instruction $sltiu$ interpreting the sign extended immediate constant as a binary number though in case of $imm[15] = 1$ one obviously has $\langle sxt(imm)\rangle \neq \langle imm\rangle$.

Most of the operations supported in the MIPS-86 ISA deal with the registers from the general

opc	rt	Mnemonic	Assembler-Syntax	Effect
Data Transfer				
100 011		lw	lw $rt\ rs\ imm$	rt = m(rs + sxt(imm))
101 011		sw	sw $rt\ rs\ imm$	m(rs + sxt(imm)) = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000		addi	addi $rt\ rs\ imm$	rt = rs + sxt(imm)
001 001		addiu	addiu $rt\ rs\ imm$	rt = rs + sxt(imm)
001 010		slti	slti $rt\ rs\ imm$	rt = (rs $<$ sxt(imm) ? 1_{32} : 0_{32})
001 011		sltui	sltui $rt\ rs\ imm$	rt = (rs $<$ sxt(imm) ? 1_{32} : 0_{32})
001 100		andi	andi $rt\ rs\ imm$	rt = rs $\wedge$ zxt(imm)
001 101		ori	ori $rt\ rs\ imm$	rt = rs $\vee$ zxt(imm)
001 110		xori	xori $rt\ rs\ imm$	rt = rs $\oplus$ zxt(imm)
001 111		lui	lui $rt\ imm$	rt = imm0^{16}
Branch				
000 001	00000	bltz	bltz $rs\ imm$	pc = pc $+$ (rs $<$ 0 ? imm00 : 4_{32})
000 001	00001	bgez	bgez $rs\ imm$	pc = pc $+$ (rs $\geq$ 0 ? imm00 : 4_{32})
000 100		beq	beq $rs\ rt\ imm$	pc = pc $+$ (rs $=$ rt ? imm00 : 4_{32})
000 101		bne	bne $rs\ rt\ imm$	pc = pc $+$ (rs $\neq$ rt ? imm00 : 4_{32})
000 110	00000	blez	blez $rs\ imm$	pc = pc $+$ (rs $\leq$ 0 ? imm00 : 4_{32})
000 111	00000	bgtz	bgtz $rs\ imm$	pc = pc $+$ (rs $>$ 0 ? imm00 : 4_{32})

Table 3.1: I-Type MIPS-86 Instructions

opc	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j $iindex$	pc = bin_{32}(pc+4_{32})[31:28]iindex00
000 011	jal	jal $iindex$	R31 = pc + 4_{32}, pc = bin_{32}(pc+4_{32})[31:28]iindex00

Table 3.2: J-Type MIPS-86 Instructions

purpose register file (GPR). However, the coprocessor instructions serve for returning from an interrupt handler to the interrupted program and for moving data between GPRs and special purpose registers (SPR) important in the context of system programming. The SPRs available in the considered model are shortly described in Table 3.4. Note, that in contrast to [Sch13] we introduce a register containing a processor identifier. The reason is that the local APIC containing such information is not present in our model. We assume that the processor ID is hard-wired into the corresponding register and writes to this register have no effect.

3.1.2 Store Buffers

The simplest form of the store buffer we deal with in the MIPS-86 model is a first-in-first-out queue of pending memory write accesses to physical addresses. Any memory read access generated by the processor can than be serviced in the store buffer by finding the most recent SB entry for a given address. If such an entry is not present, the data is read from the memory. Using the store buffers in the hardware implementation allows to perform memory accesses

opcode	fun	rs	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-and-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		cas	cas $rd\ rs\ rt$	rd′ = m(rs) m′(rs) = (rd = m(rs) ? rt : m(rs)) flushes the SB
000000	111 110		mfence	mfence	flushes the SB
000000	111 101		locksw	locksw $rs\ rt$	m(rs) = rt flushes the SB
TLB Instructions					
000000	111 011		flush	flush	flushes TLB
000000	111 010		invlpg	invlpg $rd\ rs$	flushes TLB translations for addr. rd from ASID rs
Coprocessor Instructions					
opcode	fun	rs	Mnemonic	Assembler-Syntax	Effect
010000	011 000	10000	eret	eret	Exception Return
010000		00100	movg2s	movg2s $rd\ rt$	spr(rd)= gpr(rt)
010000		00000	movs2g	movs2g $rd\ rt$	gpr(rt) = spr(rd)

Table 3.3: R-Type MIPS-86 Instructions

index i	shorthand for i and i_5	description
0	sr	status register (masks for interrupts)
1	esr	exception status register
2	eca	exception cause register
3	epc	exception pc (return address after interrupt handling)
4	edata	exception data (effective address on page fault)
5	pto	page table origin
6	asid	address space identifier
7	mode	mode register $\in \{0^{31}1, 0^{32}\}$
8	emode	exception mode register $\in \{0^{31}1, 0^{32}\}$
9	pid	processor identifier

Table 3.4: MIPS-86 Special Purpose Registers.

faster while the memory system is still busy with another operation. However, since there is no communication between the store buffers of processors in a multi-core system, the memory visible for the programmer cannot be treated as sequentially consistent as long as no specific programming discipline is applied.

In order to allow the programmer to obtain the sequentially consistent view of accessed memory in the presence of the store buffers, hardware developers provide serializing processor instructions having an explicit flushing effect on the SB. In our MIPS-86 model we offer three of them. The memory fence instruction $mfence$ is simply used for draining the SB. The compare-and-swap cas performs an atomic conditional memory update whereas the locked write $locksw$ borrowed from [Kov13] acts as the latter one but without a condition on the memory write.

3.1.3 Address Translation

The memory management unit of the MIPS-86 processor being in *user mode* performs a 2-level translation from 32-bit virtual to physical addresses at the granularity of *memory pages* consisting of 2^{12} consecutive bytes. Therefore, a page can be addressed with 20 bits and we treat the corresponding most significant bits of a virtual address as a *virtual page address*.

The first-level page table or a so called *page directory* is accessed at the *page table origin* taken from the SPR register *pto*. This table is used for translating the first 10 bits of the page address to the memory location at which a page table of the second level (or *terminal page table*) resides in the memory. Then a terminal page table entry with an index represented by the second 10 bits of the page address provides a *physical page address*. Note that entries on any level can be marked as not *present*. This means that a corresponding translation is not set up or a memory page is not in the memory. An attempt to access such a page table entry generates a *page fault interrupt* returning the processor core to *system mode* where an operating system kernel or a hypervisor is supposed to resolve the issue.

The result of traversing the graph of the page tables for the address translation is cached in the tagged TLB in order to serve further processor translation requests in a faster way. For such partial or complete address translations in our model we use a term *walks*. The tag used in the TLB and walks is an *address space identifier* (ASID) associating translations with particular users. Since operating systems and hypervisors may modify page tables, the translations present in the TLB could become out of date. In order to allow consistent page table updates in the MIPS-86 ISA we provide two instructions typical for x86-64 architectures. The $flush$ operation just empties the TLB. The instruction *invlpg*, however, removes all walks for a certain virtual page

interrupt level	shorthand	source	type	maskable	description
0	reset	external	abort	no	reset
1	I/O	internal	repeat	yes	devices
2	ill	internal	abort	no	illegal instruction
3	mal	internal	abort	no	misaligned
4	pff	internal	repeat	no	page fault on fetch
5	pfls	internal	repeat	no	page fault on load/store
6	sysc	internal	continue	no	system call
7	ovf	internal	continue	yes	overflow

Table 3.5: MIPS-86 Interrupt Types and Priority.

address and a given ASID. Therefore, using the tagged TLB substantially reduces the number of flushes during a switch between user processes or guests by operating systems and hypervisors respectively.

The specification of the address translation and the TLB used in our MIPS-86 is presented in detail later in Section 3.2.3.

3.1.4 Interrupts

The MIPS-86 model considered in the thesis supports an *interrupt mechanism* for *internal* and *external* interrupts listed in Table 3.5. The former are triggered in the processor during instruction fetch and execution causing a specific event, e.g., an overflow in an arithmetic operation, a system call used to transfer the control from a user process to an operating system for executing a specific task, a page fault occurring when a required address translation is missing, etc. The latter have an external source and are triggered by devices or the reset signal. When an interrupt signal is raised and not *masked*, the processor switches to *system mode* and performs a so called *jump to interrupt service routine* (JISR) where a corresponding interrupt handler is supposed to be called.

For resuming the execution after interrupt handling the MIPS-86 ISA provides the instruction $eret$. Depending on the type of the interrupt, the execution of the interrupted instruction can be *repeated* or *aborted*. Moreover, in case of the type *continue* it is completed before JIRS and we proceed with the next instruction after returning from the interrupt.

The exact semantics of the interrupt mechanism in the MIPS-86 model is given later in this chapter.

3.2 Single Core MIPS-86 ISA

Now, we define the single core MIPS-86 ISA in a top-down manner. We introduce the overall machine configuration, and then describe each component separately in detail. At the end we combine all definitions in the transition function of the MIPS-86 machine.

3.2.1 Configuration Overview

Definition 3.1 (**Processor Configuration of MIPS-86**). A processor configuration

$$\mathbb{C}_{proc} \overset{def}{\equiv} (core \in \mathbb{C}_{core}, sb \in \mathbb{C}_{sb}, tlb \in \mathbb{C}_{tlb})$$

consists of the processor core $core$ executing instructions according to the MIPS-86 ISA, the store buffer sb, and the translation lookaside buffer tlb.

Definition 3.2 (Memory Configuration of MIPS-86). In the model we consider a simple byte-addressable global memory of the type

$$\mathbb{C}_m \overset{def}{\equiv} \mathbb{B}^{32} \to \mathbb{B}^8$$

Definition 3.3 (Configuration of the Single Core MIPS-86). A configuration

$$\mathbb{C}_{\text{MIPS}} \overset{def}{\equiv} (cpu \in \mathbb{C}_{proc}, m \in \mathbb{C}_m)$$

of the *single core MIPS-86 ISA* is represented by the processor configuration cpu and the global memory component m.

3.2.2 Memory and Store Buffer

For the MIPS ISA memory we define effects of read, write, and compare-and-swap operations. The two latter operations are expressed via a memory transition function executed in response to the active steps of the processor core, store buffer, or TLB as we will see in the MIPS single core transition function.

Definition 3.4 (Reading Bit-Strings from Byte Addressable Memory). For a memory $m \in \mathbb{C}_m$, an address $a \in \mathbb{B}^{32}$, and a number $d \in \mathbb{N}$ of bytes, we define

$$m_d(a) \overset{def}{\equiv} \begin{cases} m(a) & : \quad d = 1 \\ m_{d-1}(a +_{32} 1_{32}) \circ m(a) & : \quad \text{otherwise} \end{cases}$$

Since our ISA supports for simplicity only word accesses to the memory, we will use $d = 4$.

Definition 3.5 (Memory Transition Function). We define the memory transition function

$$\delta_m : \mathbb{C}_m \times \Sigma_m \to \mathbb{C}_m$$

with the input alphabet

$$\Sigma_m \overset{def}{\equiv} \left(\mathbb{B}^{32} \times \mathbb{B}^{32}\right) \cup \left(\mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32}\right)$$

such that

- $(a, v) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$ – describes a write access to address a with value v, and
- $(c, a, v) \in \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32}$ – describes a compare-and-swap access to address a with compare-value c and value v to be written in case of success.

$$\delta_m(m, in)(x) \overset{def}{\equiv} \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & : \quad in = (a, v) \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ byte(\langle x \rangle - \langle a \rangle, v) & : \quad in = (c, a, v) \wedge m_4(a) = c \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ m(x) & : \quad \text{otherwise} \end{cases}$$

In turn, the store buffer is modelled as a sequence of entries representing writes of words by the core.

Definition 3.6 (Store Buffer Configuration). The set of *store buffer entries* is given by the type

$$\mathbb{C}_{sbe} \overset{def}{\equiv} \left(a \in \mathbb{B}^{30}, v \in \mathbb{B}^{32}\right)$$

where a is a word address and v is a value to be written to the memory at the address $a \circ 00$. Therefore, the set of store buffer configurations can be defined as follows:

$$\mathbb{C}_{sb} \overset{def}{\equiv} \mathbb{C}_{sbe}^{*}$$

In the MIPS-86 model a non-empty store buffer can make an active step independently from the processor core. During such a step the SB commits a pending memory write specified by the oldest entry present in its configuration.

We do not introduce an individual SB transition function and will formalize the effect of its active and passive steps in the transition of the single core MIPS-86. Here, we provide some auxiliary definitions needed for that.

Definition 3.7 (Store Buffer Hit Predicate). For a given store buffer configuration $sb \in \mathbb{C}_{sb}$ and a memory word address $x \in \mathbb{B}^{30}$, the predicate

$$sbhit(sb, x) \overset{def}{\equiv} \exists j \in |sb| \,.\, sb[j].a = x$$

indicates a *store buffer hit*, meaning that a memory word write at this address is present in sb.

Definition 3.8 (Newest Store Buffer Hit Index). Given a store buffer configuration $sb \in \mathbb{C}_{sb}$ and a word address $x \in \mathbb{B}^{30}$, we define a partial function

$$maxsbhit(sb, x) \overset{def}{\equiv} \max\{j \mid sbhit(sb[j], x)\}$$

which computes the index of the newest SB entry among those for which the hit holds.

Definition 3.9 (Store Buffer Byte Value). For a store buffer configuration $sb \in \mathbb{C}_{sb}$ and a *byte address* $x \in \mathbb{B}^{32}$ we define a partial function $sbv(sb, x)$ that computes a byte value forwarded from the store buffer for the given address in case of the hit.

Let $x_w \equiv x[31:2]$, $x_b \equiv x[1:0]$, and $k \equiv maxsbhit(sb, x_w)$, then if $sbhit(sb, x_w)$ holds, we can compute

$$sbv(sb, x) \overset{def}{\equiv} byte(\langle x_b \rangle, sb[k].v)$$

The results of read accesses performed by the processor core can then be described in terms of a *memory system* including the store buffer and the memory.

Definition 3.10 (Memory System). We define a function ms that, given these components, returns the merged memory view seen by the processor core

$$ms(sb, m)(x) \overset{def}{\equiv} \begin{cases} sbv(sb, x) & : \quad sbhit(sb, x[31:2]) \\ m(x) & : \quad \text{otherwise} \end{cases}$$

3.2.3 Translation Lookaside Buffer

For any memory access by the processor in user mode, the virtual address is translated into the physical one on the base of the page tables residing in the memory and the TLB. In order to perform address translation, the MMU operates on the page tables to create, extend, complete, and drop walks.

Definition 3.11 (TLB Configuration of MIPS-86). We define the set of configurations of the TLB as

$$\mathbb{C}_{tlb} \overset{def}{\equiv} 2^{\mathbb{C}_{walk}}$$

where the set of walks $\mathbb{C}_{walk}$ is given by

$$\mathbb{C}_{walk} \overset{def}{\equiv} \big(vpa \in \mathbb{B}^{20}, asid \in \mathbb{B}^{6}, level \in \{0,1,2\}, pa \in \mathbb{B}^{20}, r \in \mathbb{B}^{3}, fault \in \mathbb{B}\big)$$

with the following components:

- vpa – the virtual page address to be translated,
- $asid$ – the address space identifier the translation belongs to,
- $level$ – the current level of the walk, i.e. the number of remaining walk extensions needed to complete the walk,
- pa – the physical page address of the page table to be accessed next, or, if the walk is complete, the result of the translation,
- r – accumulated access rights, such that $r[0]$ stands for write permission, $r[1]$ for user mode access, and $r[2]$ expresses execute permission,
- $fault$ – a flag indicating a page fault.

Now, we describe the structure of page tables and define the addresses translation for given virtual address, page table origin, and memory configuration.

Definition 3.12 (Page and Byte Index). We split a given virtual address $va \in \mathbb{B}^{32}$ in the following way

$$va \overset{def}{\equiv} va.px_2 \circ va.px_1 \circ va.px_0$$

and define

- the second-level page index $va.px_2 \overset{def}{\equiv} va[31:22]$,
- the first-level page index $va.px.1 \overset{def}{\equiv} va[21:12]$,
- the byte offset $va.px_0 \overset{def}{\equiv} va[11:0]$ of va.

Note, that the concatenation $va.vpa \equiv a.px_2 \circ a.px_1$ constitutes the virtual page address to be translated.

Definition 3.13 (Page Table Entry). A *page table entry* $pte \in \mathbb{B}^{32}$ consists of

- $pte.pa \overset{def}{\equiv} pte[31:12]$ – the physical page address of the next page table or, if the page table is the terminal one, the resulting physical page address for a translation,
- $pte.p \overset{def}{\equiv} pte[11]$ – the present bit,
- $pte.r \overset{def}{\equiv} pte[10:8]$ – the access rights for pages accessed via a translation that involves the page table entry: $pte.r[0]$ indicates the write permission, $pte.r[1]$ is the permission for user accesses, and $pte.r[2]$ allows to fetch and execute instructions,

- $pte.a \overset{def}{\equiv} pte[7]$ – the accessed flag indicating whether the MMU has already used the page table entry for a translation.

Definition 3.14 (Page Table Entry Address). For a physical page address $pa \in \mathbb{B}^{20}$ of a page table and a page index $i \in \mathbb{B}^{10}$, we define the corresponding *page table entry address* as

$$ptea(pa, i) \overset{def}{\equiv} pa \circ 0^{12} +_{32} 0^{20}i00$$

The page table entry address needed to extend a given walk $w \in \mathbb{C}_{walk}$ is then defined as

$$ptea(w) \overset{def}{\equiv} ptea(w.pa, (w.vpa \circ 0^{12}).px_{w.level})$$

Definition 3.15 (Page Table Entry for a Walk). Given a memory $m \in \mathbb{C}_m$ and a walk $w \in \mathbb{C}_{walk}$, we define the page table entry needed to extend this walk as

$$pte(m, w) = m_4(ptea(w))$$

Definition 3.16 (Walk Creation). We define the function

$$winit : \mathbb{B}^{20} \times \mathbb{B}^{32} \times \mathbb{B}^6 \to \mathbb{C}_{walk}$$

which, given a virtual page address $vpa \in \mathbb{B}^{20}$, a page table origin $pto \in \mathbb{B}^{20}$, and an address space identifier $asid \in \mathbb{B}^6$, returns the *initial walk* for the translation of va.

$$winit(vpa, pto, asid) \overset{def}{\equiv} w$$

is given by

$$\begin{aligned} &w.vpa = vpa \qquad && w.level = 2 \qquad && w.r = 111 \\ &w.asid = asid && w.pa = pto[31:12] && w.fault = 0 \end{aligned}$$

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 3.17 (Sufficient Access Rights). For a pair of access rights $r, r' \in \mathbb{B}^3$, we use

$$r \leq r' \overset{def}{\equiv} \forall j \in [0:2].\ r[j] \leq r'[j]$$

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 3.18 (Walk Extension). We define the function

$$wext : \mathbb{C}_{walk} \times \mathbb{B}^{32} \times \mathbb{B}^3 \to \mathbb{C}_{walk}$$

which extends a given walk $w \in \mathbb{C}_{walk}$ using a page table entry $pte \in \mathbb{B}^{32}$ and access rights $r \in \mathbb{B}^3$ in such a way that

$$wext(w, pte, r) \overset{def}{\equiv} w'$$

is given by

$$w'.vpa = w.vpa$$

$$w'.asid = w.asid$$

$$w'.level = \begin{cases} w.level - 1 & : \quad pte.p \\ w.level & : \quad \text{otherwise} \end{cases}$$

$$w'.pa = \begin{cases} pte.pa & : \quad pte.p \\ w.pa & : \quad \text{otherwise} \end{cases}$$

$$w'.r = \begin{cases} w.r \wedge pte.r & : \quad pte.p \\ w.r & : \quad \text{otherwise} \end{cases}$$

$$w'.fault = \neg pte.p \vee \neg r \leq (w.r \wedge pte.r)$$

Definition 3.19 (Complete Walk). A walk $w \in \mathbb{C}_{walk}$ with $w.level = 0$ is called a *complete walk*

$$complete(w) \overset{def}{\equiv} w.level = 0$$

Definition 3.20 (Setting Accessed Flag of a Page Table Entry). Given a page table entry $pte \in \mathbb{B}^{32}$, we define the function

$$seta(pte) \overset{def}{\equiv} pte[a := 1]$$

which returns an updated page table entry in which the accessed bit is set.

Definition 3.21 (Translation Request). A *translation request* from the processor core is defined as a triple

$$TRq \overset{def}{\equiv} \big(asid \in \mathbb{B}^6, va \in \mathbb{B}^{32}, r \in \mathbb{B}^3\big)$$

consisting of the address space identifier $asid$, virtual address va, and access rights r depending on operation performed by the core.

Definition 3.22 (TLB Hit). When a walk $w \in \mathbb{C}_{walk}$ matches a translation request $trq \in TRq$ in terms of virtual page address, address space identifier and access rights, we call this a *TLB hit*:

$$hit(trq, w) \overset{def}{\equiv} \begin{array}{ll} (i) & w.vpa = trq.va.vpa \\ (ii) & w.asid = trq.asid \\ (iii) & trq.r \leq w.r \end{array}$$

Note, that a hit may be to an incomplete walk.

Definition 3.23 (Page-Faulting Walk Extension). A page fault for a given translation request $trq \in TRq$ can occur for a given walk $w \in \mathbb{C}_{walk}$ when its walk extension using a page table entry residing in memory m would result in a fault.

$$fault(m, trq, w) \overset{def}{\equiv} \begin{array}{ll} (i) & hit(trq, w) \\ (ii) & \neg complete(w) \\ (iii) & wext(w, pte(m, w), trq.r).fault \end{array}$$

Though page faults may occur at any translation level, according to [Sch13] the TLB in the MIPS-86 model keeps only non-faulting walks.

Definition 3.24 (Physical Memory Address – Translation result). For a complete walk $w \in \mathbb{C}_{walk}$ and a virtual address $va \in \mathbb{B}^{32}$ we define the result of the translation - the physical memory address as

$$pma(w, va) \overset{def}{\equiv} w.pa \circ va.px_0$$

How exactly the page faults are triggered and how the MMU operations affect the single core MIPS-86 machine configuration will be formalized in the top-level transition function in Section 3.2.5. For now, we define the TLB transition function only for basic operations on its configuration.

Definition 3.25 (Transition Function of the TLB). The transition function of the TLB

$$\delta_{tlb} : \mathbb{C}_{tlb} \times \Sigma_{tlb} \to \mathbb{C}_{tlb}$$

is defined for the input alphabet

$$\Sigma_{tlb} \overset{def}{\equiv} \{\texttt{flush}\} \times \mathbb{B}^6 \times \mathbb{B}^{20} \cup \{\texttt{flush-incomplete}\} \cup \{\texttt{add-walk}\} \times \mathbb{C}_{walk}$$

such that, depending on a given inputs, the TLB performs the following operations:

- flushing a virtual page address for a given address space identifier

$$\delta_{tlb}(tlb, (\texttt{flush}, asid, vpa)) \overset{def}{\equiv} \{w \in tlb \mid \neg(w.asid = asid \wedge w.vpa = vpa)\}$$

- flushing all incomplete walks from the TLB

$$\delta_{tlb}(tlb, \texttt{flush-incomplete}) \overset{def}{\equiv} \{w \in tlb \mid complete(w)\}$$

- adding a walk

$$\delta_{tlb}(tlb, (\texttt{add-walk}, w)) \overset{def}{\equiv} tlb \cup \{w\}$$

3.2.4 Processor Core

3.2.4.1 Configuration and Transition Overview

Definition 3.26 (Processor Core Configuration). A MIPS-86 *processor core configuration*

$$\mathbb{C}_{core} \overset{def}{\equiv} \left(pc \in \mathbb{B}^{32}, gpr : \mathbb{B}^5 \to \mathbb{B}^{32}, spr : \mathbb{B}^5 \to \mathbb{B}^{32}\right)$$

consist of the program counter *pc*, the general purpose register file *gpr*, and the special purpose register file *spr*.

Definition 3.27 (Current Address Space Identifier and Mode). The address space identifier in the core configuration $c \in \mathbb{C}_{core}$ is given by the first 6 bits of the special purpose register $asid$:

$$asid(c) \overset{def}{\equiv} c.spr(asid)[5:0]$$

The mode and exception mode of the processor core are indicated by the first bit of the corresponding SPR:

$$mode(c) \overset{def}{\equiv} c.spr(mode)[0] \qquad emode(c) \overset{def}{\equiv} c.spr(emode)[0]$$

Definition 3.28 (Processor Core Transition Function). We define the processor core transition function

$$\delta_{core} : \mathbb{C}_{core} \times \Sigma_{core} \rightarrow \mathbb{C}_{core}$$

which takes a processor core input from

$$\Sigma_{core} \overset{def}{\equiv} \Sigma_{instr} \times \Sigma_{eev} \times \mathbb{B} \times \mathbb{B}$$

where

$$\Sigma_{instr} \overset{def}{\equiv} \mathbb{B}^{32} \times \mathbb{B}^{32}$$

is a set of inputs required for instruction execution, i.e. a pair of an instruction word $I \in \mathbb{B}^{32}$ and value $R \in \mathbb{B}^{32}$ read from memory in case of lw or cas instructions, and

$$\Sigma_{eev} \overset{def}{\equiv} \mathbb{B}^{256}$$

is used to represent a vector $eev \in \mathbb{B}^{256}$ of external interrupt signals. Moreover, we explicitly pass the page fault on fetch and page fault on load/store signals $pff, pfls \in \mathbb{B}$.

Then, the processor core transition function is defined by a case split on signal $jisr$ (jump to the interrupt service routine) indicating whether an interrupt is triggered in the current step of the machine and the signal $eret$ (return from exception) which is active when the instruction I to be executed is $eret$.

$$\delta_{core}(c, I, R, eev, pff, pfls) \overset{def}{\equiv} \begin{cases} \delta_{jisr}(c, I, R, eev, pff, pfls) & : \quad jisr(c, I, eev, pff, pfls) \\ \delta_{eret}(c) & : \quad \neg jisr(c, I, eev, pff, pfls) \wedge \\ & \quad\; eret(I) \\ \delta_{instr}(c, I, R) & : \quad \text{otherwise} \end{cases}$$

In the definition above, we use the auxiliary transition functions that will be considered in detail in this section:

- execution of an non-interrupted instruction

$$\delta_{instr} : \mathbb{C}_{core} \times \Sigma_{instr} \rightarrow \mathbb{C}_{core}$$

- computation of the next state of the core when an interrupt is triggered

$$\delta_{jisr} : \mathbb{C}_{core} \times \Sigma_{core} \rightarrow \mathbb{C}_{core}$$

- return from exception

$$\delta_{eret} : \mathbb{C}_{core} \rightarrow \mathbb{C}_{core}$$

3.2.4.2 Instruction Execution

First, we introduce auxiliary definitions that will allow us to consider the semantics of the instruction execution by the processor core.

Instruction Decoding

Definition 3.29 (Fields of the Instruction Layout). We define formally the types and fields of instructions according to the MIPS-86 instruction set and the instruction word layout given in Section 3.1.1:

- *instruction opcode*
$$opc(I) \overset{def}{\equiv} I[31:26]$$
- *instruction type*
$$rtype(I) \overset{def}{\equiv} opc(I) = 0^6 \vee opc(I) = 010^4$$
$$jtype(I) \overset{def}{\equiv} opc(I) = 0^4 10 \vee opc(I) = 0^4 11$$
$$itype(I) \overset{def}{\equiv} \neg(rtype(I) \vee jtype(I))$$
- *register addresses*
$$rs(I) \overset{def}{\equiv} I[25:21] \qquad rt(I) \overset{def}{\equiv} I[20:16] \qquad rd(I) \overset{def}{\equiv} I[15:11]$$
- *shift amount*
$$sa(I) \overset{def}{\equiv} I[10:6]$$
- *function code* (used only for R-type instructions)
$$fun(I) \overset{def}{\equiv} I[5:0]$$
- *immediate constant and instruction index* (for I-type and J-type instructions, respectively)
$$imm(I) \overset{def}{\equiv} I[15:0] \qquad iindex(I) \overset{def}{\equiv} I[25:0]$$

Definition 3.30 (Instruction Decoding Predicates). For every mnemonic mn of a MIPS instruction I (see MIPS ISA-tables at the beginning of the chapter) we define a predicate $mn(I)$ which is true, if I is an mn instruction. Formally, the predicates check for the corresponding opcode and function code. E.g.
$$lw(I) \equiv opc(I) = 100011$$
$$add(I) \equiv rtype(I) \wedge fun(I) = 100000$$
The remaining predicates associated with the mnemonics are derived in the same obvious way.

Definition 3.31 (Undefined ISA Instruction). By inspection of the tables we define a predicate $undef(I)$ which is true if an instruction is not defined in the considered MIPS-86 ISA. This predicate is basically a negation of the conjunction of all defined instruction decoding predicates.

Definition 3.32 (Illegal Instruction). We consider an instruction I to be *illegal* for the core in configuration $c \in \mathbb{C}_{core}$ if this instruction does not belong to the instruction set or in user mode there is an attempt to execute one of the coprocessor instructions as well as to flush the TLB:

$$\begin{aligned} ill(c, I) \overset{def}{\equiv}\ & undef(I) \vee \\ & mode(c) \wedge (movg2s(I) \vee movs2g(I) \vee eret(I) \vee flush(I) \vee invlpg(I)) \end{aligned}$$

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

Table 3.6: Specification of ALU Operations

Operations on the translation lookaside buffer in user mode are forbidden for user processes running under an operating system because the OS is responsible for virtualization of user address spaces and the address translation is not visible from the process point of view. In case of a hypervisor, however, a guest operating system running in user mode on the host machine must be able to perform operations on the TLB. However, since the MIPS-86 model does not support hardware virtualization, the hypervisor has to emulate such guest operations by implementing, e.g., the shadow page table mechanism [Kov13].

ALU Operations

The result of arithmetic, logical, and test-and-set operations is computed by the *arithmetic logic unit* (ALU) of the MIPS-86 model.

Definition 3.33 (**ALU Specification**). For two operands $a, b \in \mathbb{B}^{32}$, control bits $alucon \in \mathbb{B}^4$ and a flag $i \in \mathbb{B}$, such that $alucon$ and i represent an ALU operation, we define the result $alures(a, b, alucon, i) \in \mathbb{B}^{32}$ of the operation and the overflow flag $ovf(a, b, alucon) \in \mathbb{B}$ according to Table 3.6.

Definition 3.34 (**ALU Instruction Predicates**). To describe whether a given instruction $I \in \mathbb{B}^{32}$ performs an ALU operation, we define the following predicates:

- I-type ALU instruction: $alui(I) \overset{def}{\equiv} itype(I) \wedge I[31:29] = 001$
- R-type ALU instruction: $alur(I) \overset{def}{\equiv} rtype(I) \wedge I[5:4] = 10$
- any ALU instruction: $alu(I) \overset{def}{\equiv} alui(I) \vee alur(I)$

Definition 3.35 (**ALU Operands of Instruction**). Using the ISA tables, we specify the right and left operands of an ALU instruction $I \in \mathbb{B}^{32}$ for a given processor core configuration $c \in \mathbb{C}_{core}$ as follows:

- left ALU operand: $lop(c, I) \overset{def}{\equiv} c.gpr(rs(I))$

- right ALU operand: $rop(c, I) \overset{def}{\equiv} \begin{cases} c.gpr(rt(I)) & : \quad rtype(I) \\ sxt_{32}(imm(I)) & : \quad \neg rtype(I) \wedge \neg I[28] \\ zxt_{32}(imm(I)) & : \quad \text{otherwise} \end{cases}$

Definition 3.36 (ALU Control Bits of Instruction). We define the ALU control bits of an instruction $I \in \mathbb{B}^{32}$ as

$$alucon(I)[2:0] \overset{def}{\equiv} \begin{cases} I[2:0] & : \quad rtype(I) \\ I[28:26] & : \quad \text{otherwise} \end{cases}$$

$$alucon(I)[3] \overset{def}{\equiv} rtype(I) \wedge I[3] \vee \neg I[28] \wedge I[27]$$

Definition 3.37 (ALU Computation Result). Therefore, the ALU result of an arithmetic, logical, or test-and-set instruction I executed by the processor core in a configuration $c \in \mathbb{C}_{core}$ is defined as

$$compres(c, I) \overset{def}{\equiv} alures(lop(c, I), rop(c, I), alucon(I), itype(I))$$

Jump and Branch Instructions

The jump and branch instruction of the MIPS-86 ISA influence the program control flow by changing the program counter of the processor core to a new address value. In comparison to the ordinary jumps, the branch instructions allow to set the program counter only when a certain condition on a value of a general purpose register holds.

Definition 3.38 (Branch Condition Evaluation). Depending on a comparison operation determined by control bits $bcon \in \mathbb{B}^4$, we define the function $bcres(a, b, bcon) \in \mathbb{B}$ for the brunch condition evaluation comparing a parameter $a \in \mathbb{B}^{32}$ with zero or with a second parameter $b \in \mathbb{B}^{32}$ in the following way:

$$bcres(a, b, bcon) \overset{def}{\equiv} \begin{cases} [a] < 0 & : \quad bcon = 0010 \\ [a] \geq 0 & : \quad bcon = 0011 \\ a = b & : \quad bcon[3:1] = 100 \\ a \neq b & : \quad bcon[3:1] = 101 \\ [a] \leq 0 & : \quad bcon[3:1] = 110 \\ [a] > 0 & : \quad bcon[3:1] = 111 \\ \textit{undefined} & : \quad \text{otherwise} \end{cases}$$

Definition 3.39 (Branch Instruction Predicates). The following branch instruction predicates denote whether a given instruction $I \in \mathbb{B}^{32}$ is a jump or successful branch instruction in a given core configuration $c \in \mathbb{C}_{core}$:

- branch instruction: $b(I) \overset{def}{\equiv} opc(I)[5:3] = 0^3 \wedge itype(I)$
- jump instruction: $jump(I) \overset{def}{\equiv} j(I) \vee jal(I) \vee jr(I) \vee jalr(I)$
- jump or branch taken:

$$jbtaken(c, I) \overset{def}{\equiv} jump(I) \vee b(I) \wedge bcres(c.gpr(rs(I)), c.gpr(rt(I)), opc(I)[2:0]rt(I)[0])$$

Definition 3.40 (Branch Target). Then the target address of a jump or successful branch instruction $I \in \mathbb{B}^{32}$ in a given core configuration $c \in \mathbb{C}_{core}$ is computed as

$$btarget(c, I) \overset{def}{\equiv} \begin{cases} c.pc +_{32} sxt_{30}(imm(I))00 & : \quad b(I) \\ c.gpr(rs(I)) & : \quad jr(I) \vee jalr(I) \\ (c.pc +_{32} 4_{32})[31:28]iindex(c)00 & : \quad j(I) \vee jal(I) \end{cases}$$

Shift Operations

The result of a shift instruction is computed by a *shift unit* performing operations on values from the general purpose registers.

Definition 3.41 (Shift Operations). For bit-string $a \in \mathbb{B}^n$ and a shift distance $i \in \{0, \ldots, n-1\}$ we define the following shift operations available in MIPS-86 ISA:

- shift left logical: $sll(a, i) \overset{def}{\equiv} a[n-i-1:0]0^i$
- shift right logical: $srl(a, i) \overset{def}{\equiv} 0^i a[n-1:i]$
- shift right arithmetic: $sra(a, i) \overset{def}{\equiv} a_{n-1}^i a[n-1:i]$

Note that for MIPS-86 we will use the aforementioned definitions only for $n = 32$.

Definition 3.42 (Shift Unit Specification). For inputs $a \in \mathbb{B}^n, i \in \{0, \ldots, n-1\}$, and a shift operation represented by bits $sf \in \mathbb{B}^2$ (shift function), we define the result of the operation as follows:

$$sures(a, i, sf) \overset{def}{\equiv} \begin{cases} sll(a, i) & : \quad sf = 00 \\ srl(a, i) & : \quad sf = 10 \\ sra(a, i) & : \quad sf = 11 \\ undefined & : \quad \text{otherwise} \end{cases}$$

Definition 3.43 (Shift Instruction Predicate). The predicate $su(I)$ denotes whether an instruction $I \in \mathbb{B}^{32}$ is a shift instruction

$$su(I) \overset{def}{\equiv} sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

Definition 3.44 (Shift Operands). Given a shift instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in \mathbb{C}_{core}$, we define the following shift operands:

- shift distance: $sdist(c, I) \overset{def}{\equiv} \begin{cases} \langle sa(I) \rangle & : \quad fun(I)[3] = 0 \\ \langle c.gpr(rs(I))[4:0] \rangle & : \quad fun(I)[3] = 1 \end{cases}$
- shift left operand: $slop(c, I) \overset{def}{\equiv} c.gpr(rt(I))$

Definition 3.45 (Shift Function). The shift function of a shift instruction $I \in \mathbb{B}^{32}$ is given by

$$sf(I) \overset{def}{\equiv} I[1:0]$$

Definition 3.46 (Shift Unit Computation Result). The result of a shift instruction $I \in \mathbb{B}^{32}$ computed by the shift unit of the processor core in a configuration $c \in \mathbb{C}_{core}$ is defined as

$$shiftres(c, I) \overset{def}{\equiv} sures(slop(c, I), sdist(c, I), sf(I))$$

Memory Access

In order to define how values are read/written from/to the memory system in the single core MIPS-86 transition function we provide a few auxiliary definitions.

Definition 3.47 (Memory Instruction Predicates). For an instruction $I \in \mathbb{B}^{32}$ we define the following predicates:

- memory load instruction: $load(I) \overset{def}{\equiv} lw(I) \vee cas(I)$
- memory store instruction: $store(I) \overset{def}{\equiv} sw(I) \vee locksw(I) \vee cas(I)$
- memory instruction: $mem(I) \overset{def}{\equiv} load(I) \vee store(I)$

Definition 3.48 (Effective Address). The effective address $ea(c, I)$ at which the memory system is accessed by a memory instruction $I \in \mathbb{B}^{32}$ executed by the processor core in configuration $c \in \mathbb{C}_{core}$ is computed as

$$ea(c, I) \overset{def}{\equiv} \begin{cases} c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & : \quad itype(I) \\ c.gpr(rs(I)) & : \quad rtype(I) \end{cases}$$

A byte address of a memory access must be correctly *aligned*, i.e., divisible by the width of the access, what is simply a word (4 bytes) for instruction fetch and data load/store in the considered MIPS-86 ISA. Otherwise, the corresponding interrupt is triggered in the processor core.

Definition 3.49 (Data and Instruction Misalignment Predicates). For an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in \mathbb{C}_{core}$, we define the predicate

$$dmal(c, I) \overset{def}{\equiv} mem(I) \wedge ea(c, I)[1:0] \neq 00$$

indicating whether the data memory access is misaligned. Analogously, the predicate

$$imal(c) \overset{def}{\equiv} c.pc[1:0] \neq 00$$

shows that the program counter for the instruction fetch is not word aligned.

Definition 3.50 (Store Value). Given a memory store instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in \mathbb{C}_{core}$, the store value is a word taken from the general purpose register specified by $rt(I)$

$$sv(c, I) \overset{def}{\equiv} c.gpr(rt(I))$$

GPRs Update

Definition 3.51 (General Purpose Register Write Predicate). The predicate

$$dprw(I) \overset{def}{\equiv} alu(I) \vee su(I) \vee load(I) \vee jal(I) \vee jalr(I) \vee movs2g(I)$$

describes that a given instruction $I \in \mathbb{B}^{32}$ writes to some general purpose register.

Definition 3.52 (General Purpose Register Result Destination). We define the result destination $cad(I)$ of an instruction $I \in \mathbb{B}^{32}$ writing to a general purpose register as an address of this register:

$$cad(I) \overset{def}{\equiv} \begin{cases} 1^5 & : \quad jal(I) \\ rd(I) & : \quad rtype(I) \wedge \neg movs2g(I) \\ rt(I) & : \quad \text{otherwise} \end{cases}$$

Definition 3.53 (General Purpose Register Input). A value written by an instruction $I \in \mathbb{B}^{32}$ to the general purpose register $cad(I)$ in a processor core configuration $c \in \mathbb{C}_{core}$ is computed as

$$gprdin(c, I, R) \overset{def}{\equiv} \begin{cases} c.pc +_{32} 4_{32} & : \quad jal(I) \vee jalr(I) \\ R & : \quad load(I) \\ c.spr(rd(I)) & : \quad movs2g(I) \\ compres(c, I) & : \quad alu(I) \\ shiftres(c, I) & : \quad su(I) \end{cases}$$

where $R \in \mathbb{B}^{32}$ is a word read by the memory instruction from the memory system.

Instruction Execution Transition Function

Using the definitions from above we can now define the transition function for the instruction execution by the processor core.

Definition 3.54 (Non-Interrupted Instruction Execution). For a processor core in a configuration $c \in \mathbb{C}_{core}$, a legal instruction $I \in \mathbb{B}^{32}$, and a data word $R \in \mathbb{B}^{32}$ read from the memory system when needed we define the result $c' \in \mathbb{C}_{core}$ of the non-interrupted instruction execution as

$$\delta_{instr}(c, I, R) \overset{def}{\equiv} c'$$

such that the processor core components are updated in the following way:

- $c'.pc = \begin{cases} btarget(c, I) & : \quad jbtaken(c, I) \\ c.pc +_{32} 4_{32} & : \quad \text{otherwise} \end{cases}$
- $c'.gpr(x) = \begin{cases} gprdin(c, I, R) & : \quad x = cad(I) \wedge gprw(I) \\ c.gpr(x) & : \quad \text{otherwise} \end{cases}$
- $c'.spr(x) = \begin{cases} c.gpr(rt(I)) & : \quad rd(I) = x \wedge movg2s(I) \\ c.spr(x) & : \quad \text{otherwise} \end{cases}$

3.2.4.3 Interrupt Handling

In Table 3.5 we have already shown the interrupt ordered by their priority. Among those the MIPS-86 model supports the external interrupts $eev \in \mathbb{B}^{256}$ such that $eev[0]$ is the signal $reset$ and $eev[255:1]$ are interrupts from devices. This kind of interrupts plays a role in the full model with external devices as well as the local APIC providing these external events to the processor core. Since we are not interested in these hardware components in the scope of the thesis, we do not include them into our model for simplicity. The APIC model for MIPS-86 is described in detail in [Sch13].

Triggering of Interrupts

While external event signals are served as an input to the processor core transition function, the internal event signals, as we know, are triggered in the processor.

Definition 3.55 (Internal Event Vector). Given a processor core configuration $c \in \mathbb{C}_{core}$, an instruction $I \in \mathbb{B}^{32}$ to be executed if possible, and the page fault signals $pff, pfls \in \mathbb{B}$ provided by the MMU to the processor core, we define the computation of the internal even signals $iev(c, I, pff, pfls) \in \mathbb{B}^8$ as follows:

$$iev(c,I,pff,pfls)[i] \overset{def}{\equiv} \begin{cases} \neg imal(c) \wedge \neg pff \wedge ill(c,I) & : \quad i=2 \\ imal(c) \vee (\neg imal(c) \wedge \neg pff \wedge dmal(c,I)) & : \quad i=3 \\ pff \wedge \neg imal(c) & : \quad i=4 \\ pfls \wedge \neg imal(c) \wedge \neg pff \wedge \neg dmal(c,I) & : \quad i=5 \\ \neg imal(c) \wedge \neg pff \wedge sysc(I) & : \quad i=6 \\ \neg imal(c) \wedge \neg pff \wedge alu(I) \wedge & : \quad i=7 \\ ovf(lop(c,I), rop(c,I), alucon(I)) & \\ undefined & : \quad \text{otherwise} \end{cases}$$

Note that though the page fault signals appear as external inputs in the processor core transition function, they originate from the MMU belonging to the processor and treated as internal.

When an interrupt occurs, the information about this interrupt is saved in a special purpose register to allow a programmer to determine the reason (or cause) of the interrupt.

Definition 3.56 (Cause and Masked Cause of Interrupts). Given a processor core configuration $c \in \mathbb{C}_{core}$, an instruction $I \in \mathbb{B}^{32}$ to be executed, an external event vector $eev \in \mathbb{B}^{256}$, and page fault signals $pff, pfls \in \mathbb{B}$, we can define the cause $ca \in \mathbb{B}^8$ and masked cause $mca \in \mathbb{B}^8$ of interrupts in the following way:

$$ca(c,I,eev,pff,pfls)[j] \overset{def}{\equiv} \begin{cases} eev[0] & : \quad j=0 \\ \bigvee_{i=1}^{255} eev[i] & : \quad j=1 \\ iev(c,I,pff,pfls)[j] & : \quad j \in [2:7] \end{cases}$$

$$mca(c,I,eev,pff,pfls)[j] \overset{def}{\equiv} \begin{cases} ca(c,I,eev,pff,pfls)[j] \wedge c.spr(sr)[j] & : \quad j \in \{1,7\} \\ ca(c,I,eev,pff,pfls)[j] & : \quad \text{otherwise} \end{cases}$$

Definition 3.57 (Jump to Interrupt Service Routine Predicate). To denote that in a given configuration $c \in \mathbb{C}_{core}$ for a given instruction $I \in \mathbb{B}^{32}$, external event signals $eev \in \mathbb{B}^{256}$, and page fault signals $pff, pfls \in \mathbb{B}$ an interrupt is triggered, we define the predicate

$$jisr(c,I,eev,pff,pfls) \overset{def}{\equiv} \bigvee_{j} mca(c,I,eev,pff,pfls)[j]$$

Definition 3.58 (Interrupt Level of Triggered Interrupt). To determine the interrupt level of the triggered interrupt, we define the function

$$il(c,I,eev,pff,pfls) \overset{def}{\equiv} min\{j \mid mca(c,I,eev,pff,pfls)[j] = 1\}$$

Definition 3.59 (Continue Type Interrupt Predicate). The predicate

$$cont(c,I,eev,pff,pfls) \overset{def}{\equiv} il(c,I,eev,pff,pfls) \in \{6,7\}$$

shows whether the triggered interrupt is of continue type.

Transition Functions for Exception and Return from Exception

Definition 3.60 (Interrupt Execution Transition Function). For a processor core in a configuration $c \in \mathbb{C}_{core}$, an instruction $I \in \mathbb{B}^{32}$ to be executed, external events $eev \in \mathbb{B}^{256}$, and page fault signals $pff, pfls \in \mathbb{B}$ provided by the processor's MMU we define the computation of the next processor core configuration $c' \in \mathbb{C}_{core}$ in case of an interrupt as

$$\delta_{jisr}(c, I, R, eev, pff, pfls) \overset{def}{\equiv} c'$$

such that

- $c'.pc = 0^{32}$
- $c'.spr(x) = \begin{cases} 0^{32} & : \quad x = sr \\ 0^{32} & : \quad x = mode \\ c.spr(mode) & : \quad x = emode \\ c.spr(sr) & : \quad x = esr \\ zxt_{32}\,(mca(c, I, eev, pff, pfls)) & : \quad x = eca \\ c.pc & : \quad x = epc \wedge \neg cont(c, I, eev, pff, pfls) \\ \delta_{instr}(c, I, R).pc & : \quad x = epc \wedge cont(c, I, eev, pff, pfls) \\ ea(c, I) & : \quad x = edata \wedge il(c, I, eev, pff, pfls) = 5 \\ bin_{32}\,(min\{j \mid eev[j] = 1\}) & : \quad x = edata \wedge il(c, I, eev, pff, pfls) = 1 \\ c.spr(x) & : \quad \text{otherwise} \end{cases}$
- $c'.gpr = \begin{cases} c.gpr & : \quad \neg cont(c, I, eev, pff, pfls) \\ \delta_{instr}(c, I, R).gpr & : \quad \text{otherwise} \end{cases}$

Note that in case of the page fault on fetch the program counter is saved in epc and there is no need to use $edata$ additionally. Moreover, in case of an interrupt of the type $continue$, the core needs to finish the operation and save the computation result to the GPRs.

As we know to restore the execution after interrupt handling the MIPS-86 ISA provides the instruction $eret$.

Definition 3.61 (Return From Exception Transition Function). For a processor core executing the instruction $eret$ in a configuration $c \in \mathbb{C}_{core}$ we compute the core's next configuration $c' \in \mathbb{C}_{core}$ as follows:

$$\delta_{eret}(c) \overset{def}{\equiv} c'$$

- $c'.pc = c.spr(epc)$
- $c'.spr(x) = \begin{cases} c.spr(emode) & : \quad x = mode \\ c.spr(esr) & : \quad x = sr \\ c.spr(x) & : \quad \text{otherwise} \end{cases}$
- $c'.gpr = c.gpr$

Note also that in contrast to [Sch13] our MIPS-86 model allows to return back exactly to the mode in which an interrupt occurred.

3.2.5 Single Core MIPS-86 ISA Transitions

Definition 3.62 (**Single Core MIPS-86 Transition Function**). Transitions of the single core machine are defined by the function

$$\delta_{\text{MIPS}} : \mathbb{C}_{\text{MIPS}} \times \Sigma_{\text{MIPS}} \rightharpoonup \mathbb{C}_{\text{MIPS}}$$

where

$$\begin{aligned}\Sigma_{\text{MIPS}} \overset{def}{\equiv} & \{\texttt{core}\} \times \mathbb{C}_{walk} \times \mathbb{C}_{walk} \times \mathbb{B}^{256} \cup \\ & \{\texttt{tlb-create}\} \times \mathbb{B}^{20} \cup \\ & \{\texttt{tlb-extend}\} \times \mathbb{C}_{walk} \times \mathbb{B}^{3} \cup \\ & \{\texttt{tlb-accessed}\} \times \mathbb{C}_{walk} \cup \\ & \{\texttt{sb}\}\end{aligned}$$

are *processor inputs* specifying which processor component makes a certain kind of a step. Active steps can be done by the core, TLB, and SB.

Now we define the single core MIPS-86 transition function by a case distinction on a given input $in \in \Sigma_{\text{MIPS}}$

$$\delta_{\text{MIPS}}(c, in) \overset{def}{\equiv} c'$$

Any component of the configuration $c' \in \mathbb{C}_{\text{MIPS}}$ not listed explicitly in the following has the same configuration as in $c \in \mathbb{C}_{\text{MIPS}}$.

3.2.5.1 Processor Core Step

The processor core step is indicated by the input

$$in = (\texttt{core}, w_I, w_D, eev)$$

containing the external event vector, TLB walks w_I for the instruction fetch and w_D for the data access (memory or store buffer) in translated mode. These walks are ignored in case of system mode.

For $c \in \mathbb{C}_{\text{MIPS}}$, $core \equiv c.cpu.core$, and $ms(c) \equiv ms(c.cpu.sb, c.m)$ we introduce the following definitions and corresponding shorthands:

- the translation request for instruction fetch in user mode

$$trqI \equiv trqI(core) \overset{def}{\equiv} (asid(core), core.pc, 0 \circ mode(core) \circ 1)$$

- a predicate indicating whether the page fault on fetch occurs for a the given walk and translation request

$$pff \equiv pff(c, w_I) \overset{def}{\equiv} mode(core) \wedge fault(c.m, trqI, w_I)$$

- the physical memory address for instruction fetch, meaningful in user mode only in case of no page fault on fetch

$$pmaI \equiv pmaI(c, w_I) \overset{def}{\equiv} \begin{cases} pma(w_I, core.pc) & : \quad mode(core) \\ core.pc & : \quad \neg mode(core) \end{cases}$$

- the instruction fetched from the memory in the absence of the page fault on fetch

$$I \equiv I(c, w_I) \overset{def}{\equiv} ms_4(c)(pmaI)$$

- the translation request for data access at the effective address

$$trqD \equiv trqD(core, I) \overset{def}{\equiv} (asid(core), ea(core, I), store(I) \circ mode(core) \circ 0)$$

- a predicate indicating the page fault on load/store for the given walks and translation request

$$pfls \equiv pfls(c, w_D, w_I, I) \overset{def}{\equiv} mode(core) \wedge \neg pff \wedge mem(I) \wedge fault(c.m, trqD, w_D)$$

- the physical memory address for the data access relevant if the instruction can be fetched successfully and no page fault on load/store occurs

$$pmaD \equiv pmaD(c, w_D, I) \overset{def}{\equiv} \begin{cases} pma(w_D, ea(core, I)) & : \quad mode(core) \\ ea(core, I) & : \quad \neg mode(core) \end{cases}$$

- data read from the memory system in case of lw or cas instructions

$$R \equiv R(c, w_D, I) \overset{def}{\equiv} ms_4(c)(pmaD)$$

- shorthands for $jirs$ and $reset$

$$jisr \equiv jisr(core, I, eev, pff, pfls) \qquad reset \equiv eev[0]$$

Note that the fetched instruction and data read from the memory system are ignored in certain cases of the processor core step. For example, when the instruction misalignment exception is raised, the instruction is irrelevant in the core transition function.

Transition Conditions:

1. in translated mode the walk w_I must be a walk from TLB and must match the translation request for instruction fetch

$$mode(core) \implies w_I \in c.cpu.tlb \wedge hit(trqI, w_I)$$

2. in translated mode the walk w_I must be complete in case the page fault on fetch is not raised

$$mode(core) \wedge \neg pff \implies complete(w_I)$$

3. if in translated mode the successfully fetched instruction is an instruction accessing the memory system then the matching walk w_D required for data access must be present in the TLB

$$mode(core) \wedge \neg pff \wedge mem(I) \implies$$
$$w_D \in c.cpu.tlb \wedge hit(trqD, w_D)$$

4. this matching walk w_D must be complete in case the page fault on load/store does not appear

$$mode(core) \wedge \neg pff \wedge mem(I) \wedge \neg pfls \implies complete(w_D)$$

5. the compare-and-swap, locked write, or fence instruction can only be executed when the store buffer is empty

$$cas(I) \vee locksw(I) \vee mfence(I) \implies c.cpu.sb = \varepsilon$$

6. the store buffer must be flushed before the core switches from system to user mode

$$\neg mode(core) \wedge emode(core) \wedge eret(I) \implies c.cpu.sb = \varepsilon$$

7. the store buffer must be also empty in case of interrupt in user mode

$$mode(core) \wedge jisr \implies c.cpu.sb = \varepsilon$$

8. we have the same requirement for the store buffer in case of reset in any mode

$$reset \implies c.cpu.sb = \varepsilon$$

Transition Effect:

- processor core
$$c'.cpu.core = \delta_{core}(c, I, R, eev, pff, pfls)$$
- store buffer
$$c'.cpu.sb = \begin{cases} (pmaD[31:2], sv(core, I)) \circ c.cpu.sb & : \neg jisr \wedge sw(I) \\ c.cpu.sb & : \text{otherwise} \end{cases}$$
- translation lookaside buffer

  First, we consider the effect of $invlpg(I)$ which invalidates a single virtual page address and drops all incomplete walks:
$$asid \equiv core.gpr(rs(I))[5:0]$$
$$vpa \equiv core.gpr(rd(I)).vpa$$
$$tlb' \equiv \delta_{tlb}(c.cpu.tlb, (\texttt{flush}, asid, vpa))$$
$$tlb'' \equiv \delta_{tlb}(tlb', \texttt{flush-incomplete})$$
  Then the effect on the core transition on the TLB is defined as follows
$$c'.cpu.tlb = \begin{cases} \emptyset & : \neg jisr \wedge flush(I) \vee \\ & reset \\ tlb'' & : \neg jisr \wedge invlpg(I) \\ \delta_{tlb}(c.cpu.tlb, (\texttt{flush}, asid(core), core.pc.vpa)) & : pff \wedge \neg reset \\ \delta_{tlb}(c.cpu.tlb, (\texttt{flush}, asid(core), ea(core, I).vpa)) & : \neg pff \wedge pfls \wedge \\ & \neg reset \\ c.cpu.tlb & : \text{otherwise} \end{cases}$$

If there is a page fault detected by the TLB, it reacts by flushing all walks for the given virtual page address. This operation allows the TLB to rewalk the page table after the corresponding interrupt handling by the kernel/hypervisor and not to generate the page faults repeatedly for the old walks. Note also that the TLB might make such a flush also when the core handles an interrupt of the higher priority and in fact is not interested in the results of the address translation.

- memory

$$c'.m = \begin{cases} \delta_m(c.m, (pmaD, sv(core, I))) & : \quad \neg jisr \wedge locksw(I) \\ \delta_m(c.m, (core.gpr(rd(I)), pmaD, sv(core, I))) & : \quad \neg jisr \wedge cas(I) \\ c.m & : \quad \text{otherwise} \end{cases}$$

3.2.5.2 Store Buffer Step

During this step the memory write leaves the store buffer. The store buffer step is indicated by the input

$$in = \texttt{sb}$$

Transition Conditions:

The store buffer can only make a step if it is not empty: $c.cpu.sb \neq \varepsilon$.

Transition Effect:

- store buffer

$$sblen \equiv |c.cpu.sb|$$

$$c'.cpu.sb = c.cpu.sb[1 : sblen - 1]$$

- memory

$$sbe \equiv c.cpu.sb[sblen]$$

$$c'.m = \delta_m(c.m, (sbe.a \circ 00, sbe.v))$$

3.2.5.3 Translation Lookaside Buffer Step

Walk Creation This active step of TLB creates a new walk for a given virtual page address vpa and is indicated by the transition function input

$$in = (\texttt{tlb-create}, vpa)$$

Transition Conditions:

The TLB will only create a new walk when the processor core is running in translated mode

$$mode(core)$$

Transition Effect:

$$w' \equiv winit(vpa, core.spr(pto), asid(core))$$

$$c'.cpu.tlb = \delta_{tlb}(c.cpu.tlb, (\texttt{add-walk}, w'))$$

Setting Accessed Bit The accessed bit of the page table entry required for extension of a walk w is set by the active step of the TLB if the transition function input is

$$in = (\texttt{add-accessed}, w)$$

Transition Conditions:

1. the page table entry flag can only be set in translated mode
$$mode(core)$$
2. the TLB sets the accessed bit only for incomplete walks present in the TLB and corresponding to the current address space identifier
$$w \in c.cpu.tlb \land \neg complete(w) \land w.asid = asid(core)$$
3. the accessed bit can only be set for a page table entry marked as present
$$pte(c.m, w).p$$

Transition Effect:
$$c'.m = \delta_m(c.m, (ptea(w), seta(pte(c.m, w))))$$

Walk Extension The TLB extends an existing walk w provided in the input

$$in = (\texttt{add-extend}, w)$$

Transition Conditions:

1. walk extension is performed only in translated mode
$$mode(core)$$
2. the TLB extends only incomplete walks present in the TLB and corresponding to the current address space identifier
$$w \in c.cpu.tlb \land \neg complete(w) \land w.asid = asid(core)$$
3. the access flag of the page table entry used for walk extension is set
$$pte(c.m, w).a$$
4. the extended walk $w' \equiv wext(w, pte(c.m, w), 000)$ is not faulty
$$\neg w'.fault$$

Transition Effect:
$$c'.cpu.tlb = \delta_{tlb}(c.cpu.tlb, (\texttt{add-walk}, w'))$$

3.3 Multi-Core MIPS-86

Definition 3.63 (Configuration of Multi-Core MIPS-86). For a number of processors $np \in \mathbb{N}$ we define the configuration of the multi-core MIPS-86 machine:

$$\mathbb{C}_{\text{MMIPS}} \overset{def}{\equiv} (cpu : \mathbb{N}_{np} \to \mathbb{C}_{proc}, m \in \mathbb{C}_m)$$

where cpu is a function mapping processor index to its configuration, and m is shared memory.

Definition 3.64 (Multi-Core MIPS-86 Transition Function). Transitions of the multi-core machine are described by the function

$$\delta_{\text{MMIPS}} : \mathbb{C}_{\text{MMIPS}} \times \mathbb{N}_{np} \times \Sigma_{\text{MIPS}} \rightharpoonup \mathbb{C}_{\text{MMIPS}}$$

such that for a configuration $mc \in \mathbb{C}_{\text{MMIPS}}$, an index $p \in \mathbb{N}_{np}$ of a processor performing the step, and its proper input $in \in \Sigma_{\text{MIPS}}$, the result of the transition is defined as

$$\delta_{\text{MMIPS}}(mc, p, in) \overset{def}{\equiv} \begin{cases} \big(mc.cpu[p \mapsto c'_p.cpu], c'_p.m\big) & : \quad \delta_{\text{MIPS}}((mc.cpu(p), mc.m), in) = c'_p \\ undefined & : \quad \text{otherwise} \end{cases}$$

4 Store Buffer Reduced MIPS-86 in the Context of Hypervisor/OS Kernels

Hypervisors or operating system kernels are usually written in mixed languages (e.g. C-like language and macro assembly with inline assembly portions). On a multiprocessor system their compiled code runs in system mode in parallel with the compiled code of guest operating systems or processes being executed in user mode, and must guarantee their proper virtualization.

To argue about hypervisor/kernel correctness we have to apply sequential compiler correctness in the context of a multiprocessor system. However, compilers usually guarantee correctness of executed code on the sequentially consistent memory which is not simply given for free in the presence of store buffers. Since in this work we are interested in the verification of the code running in system mode and can argue only about properties of this code, we have to reduce the ISA model introduced in Chapter 3 to a model where the store buffers of the processors executing our kernel or hypervisor are invisible under certain conditions. For the guest/process steps we will provide all components that are needed for further virtualization of guest memory as it is shown by Kovalev for a restricted version of the full x86 architecture [Deg11] in his doctoral work [Kov13].

As we have seen in Section 2.3 for a given unit we can apply the sequential simulation relation for the part of the memory not locally owned by other units. The ownership safety will guarantee that the unit does not access other units' locally owned addresses. Therefore, to be able to apply the compiler correctness on the sequentially consistent memory, we have to perform the store buffer reduction exactly for addresses involved in the simulation relation.

In contrast to the hypervisor/kernel code we are dealing with, the memory access and ownership transfer policies used by the guests or processes are unknown to the hypervisor, and we cannot rely on them. Therefore, we have to consider two distinct sets of memory addresses for the hypervisor/kernel and the guests/processes respectively, and introduce the ownership model only from the hypervisor/kernel point of view.

However, the hypervisor might need to access the memory of the guests to guarantee the desired virtualization. Without safety policy coming from the guests we cannot reduce store buffers for guest steps, and the hypervisor running on one of the processors can see only in the memory the data belonging to a guest being executed on other processors though the actual values might be in their store buffers. This affects the treatment of such addresses in the compiler consistency in a way where only the values from the memory are coupled. A reasonable way suggested in [Kov13] for such hypervisor write accesses is to use the atomic compare-and-swap or locked memory write flushing the store buffer and not changing the view of guests on their memory.

On the other hand, it is clear that the guests should not access the memory part of the hypervisor in order not to influence its execution as well as the treatment of its memory as sequential consistent. The only exception are the MMU steps during guest execution that might use a shadow page table built by the hypervisor and belonging to its address space. If shadow page tables are not shared and are individual for every processor as it is considered in [Kov13], there cannot be any problem since our MIPS-86 ISA model guarantees that the store buffer is flushed when we switch from system to user mode. Any stores in the SB will become visible for the MMU acting during the guest steps on the same processor. A more complicated case is when an SPT is shared for a guest between the processors. The hypervisor needs to make it also shared in its ownership model, and the sequential consistency of a hypervisor portion of memory occupied by this SPT might be desirable for the guest steps.

Now, as follows from the discussion above, for each processor of the multi-core MIPS-86 model, we need to have the sequentially consistent memory for owned (by a processor), read-only and shared hypervisor/kernel addresses. A straightforward way to abstract the store buffers is to use a simple discipline from [Kov13] guaranteeing that at any time there are no pending writes to the same physical memory address in more than one store buffer of the multi-core machine, and writes to the shared data never go into store buffers. Obviously, for the owned non-shared addresses it will easily follow from our ownership safety policy.

However, in contrast to Kovalev's work, in our ownership model the shared addresses are not statically fixed: a unit may acquire the ownership on them, make them local as well as release them. When a processor having in its store buffer a write to some local non-shared addresses releases these addresses without writing them back to the physical memory, it will break the consistency relation for other units because they will need to cover these addresses in their simulation relations. Therefore, we will allow the ownership transfer only at hypervisor/kernel memory accesses flushing the store buffer.

Such a discipline is quite strict in comparison to [CS10] where the authors introduced a more general model and suggested flushing the store buffer only before a shared read in case there were writes to shared data after the last flush. However, application of their theory for our MIPS-86 model and as a consequence for semantics of upper layers of the verification stack is tricky enough and considered in the doctoral thesis [Che16] of Geng Chen where he extended the model with memory management unit steps [CCK14].

In the frame of this work, we will stick to the simpler approach introduced above, which we find enough for the correctness of the model of software threads. We can also easily treat the MMU steps in the store buffer reduction because as one can see from Chapter 3, there are no MMU transitions in system mode what perfectly matches the algorithm with non-shared shadow page tables. Moreover, we aim at the application of the general theory from Chapter 2 for all layers of verification stack in this work. The simpler store buffer reduction can be done with help of the *Cosmos* model simulation theorem, which in turn will show its first full power for correctness proofs.

Note that this chapter does not pretend to introduce a new store buffer reduction strategy, but rather serves a few purposes needed for the pervasive verification of the software threads considered in this work: (i) the application of the store buffer reduction from [Kov13] for our multi-core MIPS-86 machine and the ownership model from Chapter 2, (ii) determining and checking the software conditions and safety policy enabling the SB reduction for MIPS-86 in the presence of guest/user steps, (iii) working out definitions for the instantiation of the *Cosmos* model with the multi-core MIPS-86 machine, and (iv) the detailed application of the general concurrent simulation theorem for the bottom layer of the verification stack.

We proceed as follows. First, we introduce the store buffer reduced machine simulating our reference MIPS-86 machine. In fact, we will use the same type of MIPS-86 configuration defined before, in which we will require the store buffer to be empty when the processor is stepping in system mode. The only difference is the transition function that will not write to the store buffer in this case. Next, we will prove that such a reduced machine is encoded by the steps of the reference (non-reduced) machine under the ownership safety and software conditions. For this purpose we will apply directly the theory of *Cosmos* model and simulation in concurrency by instantiating it with our reduced and reference MIPS-86 machines, and then formulating the sequential simulation relation wrt. the ideas described above and proving the corresponding assumptions.

4.1 Store Buffer Reduced MIPS-86

As we have mentioned above for the store buffer reduced single core and multi-core MIPS-86 models we use the same configurations defined before, namely $\mathbb{C}_{\text{MIPS}}$ and $\mathbb{C}_{\text{MMIPS}}$. The definitions of the corresponding transition functions, however, need to be extended.

Definition 4.1 (Reduced Single Core MIPS-86 Transition Function). Transitions of the reduced single core machine are defined by the function

$$\delta_{r\text{MIPS}} : \mathbb{C}_{\text{MIPS}} \times \Sigma_{\text{MIPS}} \rightharpoonup \mathbb{C}_{\text{MIPS}}$$

such that

$$\delta_{r\text{MIPS}}(c, in) \overset{def}{\equiv} c'$$

differs from the transition function $\delta_{\text{MIPS}}(c, in)$ only for the processor core steps determined by $in = (\texttt{core}, w_I, w_D, eev)$. The processor core makes a step under the same conditions, however, the effect of the transition on its store buffer and the memory is defined as:

$$c'.cpu.sb = \begin{cases} (pmaD[31:2], sv(core, I)) \circ c.cpu.sb & : \neg jisr \wedge sw(I) \wedge mode(core) \\ c.cpu.sb & : \text{otherwise} \end{cases}$$

$$c'.m = \begin{cases} \delta_m(c.m, (pmaD, sv(core, I))) & : \neg jisr \wedge (locksw(I) \vee \\ & \quad sw(I) \wedge \neg mode(core)) \\ \delta_m(c.m, (core.gpr(rd(I)), pmaD, sv(core, I))) & : \neg jisr \wedge cas(I) \\ c.m & : \text{otherwise} \end{cases}$$

The definition for all other components of c' is the same as in Section 3.2.5.

Note, that since we do not write to the store buffer in system mode, and it is flushed if the mode is changed (see Section 3.2.5.1), the initially empty store buffer will remain empty in system mode and will not be able to make active steps in this case.

Definition 4.2 (Reduced Multi-Core MIPS-86 Transition Function). Transitions of the reduced multicore machine are described by the function

$$\delta_{r\text{MMIPS}} : \mathbb{C}_{\text{MMIPS}} \times \mathbb{N}_{np} \times \Sigma_{\text{MIPS}} \rightharpoonup \mathbb{C}_{\text{MMIPS}}$$

such that for a configuration $mc \in \mathbb{C}_{\text{MMIPS}}$, an index of a processor $p \in \mathbb{N}_{np}$ performing the step, and its proper input $in \in \Sigma_{\text{MIPS}}$, the result of the transition is defined as

$$\delta_{r\text{MMIPS}}(mc, p, in) \overset{def}{\equiv} \begin{cases} \big(mc.cpu[p \mapsto c'_p.cpu], c'_p.m\big) & : \delta_{r\text{MIPS}}((mc.cpu(p), mc.m), in) = c'_p \\ undefined & : \text{otherwise} \end{cases}$$

4.2 Memory Address Space in Hypervisor Context

Since in the scope of this work we consider the hypervisor/OS kernel written in the aforementioned programming languages, we can define what exactly the memory address space of the running system includes wrt. the system in question. We split the memory into portions provided as instantiation parameter of the model we consider. Later on, when we talk about exact models determined by a code to be executed, we will instantiate these parameters correspondingly.

We distinguish the following fixed sets of addresses partitioning the MIPS-86 memory space in the context of the hypervisor/kernel running atop of the MIPS-86 multicore machine:

- a set of hypervisor or OS kernel addresses $A_{hyp} \subseteq \mathbb{B}^{32}$,
- a set of addresses of all guests/processes $A_{guest} \subset \mathbb{B}^{32}$ such that

$$A_{hyp} \cup A_{guest} = \mathbb{B}^{32}, \quad A_{hyp} \cap A_{guest} = \emptyset,$$

- among hypervisor/kernel addresses we differentiate a code region A_{code}, sets of addresses occupied by constants A_{const} and other data addresses A_{data} that can be accessed by read and write operations:

$$A_{hyp} = A_{code} \cup A_{const} \cup A_{data}$$

  Note, that all these sets are fixed and pairwise disjoint:

$$\forall X, Y \in \{A_{code}, A_{const}, A_{data}\} \, . \, X \neq Y \implies X \cap Y = \emptyset$$

4.3 MIPS-86 Cosmos Model Instantiations

4.3.1 Reference Machine Instantiation

For the reference non-reduced multi-core MIPS-86 machine we define an instantiation $S_{\text{MIPS}} \in \mathbb{S}$ of the *Cosmos* model containing $np \in \mathbb{N}$ processors.

- $S_{\text{MIPS}}.\mathcal{A} = \mathbb{B}^{32}$, $S_{\text{MIPS}}.\mathcal{V} = \mathbb{B}^{8}$ – The sets of addresses and values correspond to the type of the global byte-addressable memory.
- $S_{\text{MIPS}}.\mathcal{R} = A_{code} \cup A_{const}$ – The read-only addresses are defined by the corresponding areas occupied by the translated hypervisor/kernel code and its variables annotated as constants.
- $S_{\text{MIPS}}.nu = np$ – As the number of computational units we consider the number of processors $np \in \mathbb{N}$ in the multicore MIPS-86 model.
- $S_{\text{MIPS}}.\mathcal{U} = \mathbb{C}_{proc}$ – The configuration of the computation unit is the MIPS-86 processor configuration which includes the processor core, store buffer and TLB.
- $S_{\text{MIPS}}.\mathcal{E} = \Sigma_{\text{MIPS}}$ – We treat the MIPS-86 processor inputs in external inputs of the *Cosmos* model unit. The input as shown before determines which component of the processor makes a step and provides the corresponding component inputs.
- $S_{\text{MIPS}}.reads : \mathcal{U} \times (\mathcal{A} \to \mathcal{V}) \times \mathcal{E} \to 2^{\mathcal{A}}$ – The set of memory addresses read during a step depends on which processor component is involved into the transition, and in case of the core step it is determined by an instruction to be executed.

Since we consider the byte addressable memory as well as byte addresses for the safety policy, and in the MIPS-86 model support only word accesses to the memory, we will use a shorthand computing a set of 4 byte addresses from a byte address for a word access. Generally, for a byte address $a \in \mathbb{B}^{32}$ and a number $d \in \mathbb{N}$ of consecutive bytes to be accessed we refer to $\{a\}_d$ from Definition 1.24 for this purpose.

For the property $insta_r(S_{\text{MIPS}})$ (given in Definition 2.2) to hold one has to include into the $reads$-set all addresses to be read during one step transition of the MIPS-86 model.

For a MIPS-86 configuration $c \in \mathbb{C}_{\text{MIPS}}$, $core \equiv c.cpu.core$, and an input $in \in \Sigma_{\text{MIPS}}$ such that we have $in = (\texttt{core}, w_I, w_D, eev)$ and the transition function $\delta_{\text{MIPS}}(c, in)$ is defined we first introduce separately sets of addresses $AF(c, in)$ and $AR(c, in)$ needed for the instruction fetch and data read from the memory respectively in the absence of interrupts.

$$AF(c, in) \overset{def}{\equiv} \{pmaI(c, w_I)\}_4$$

For the core step the set of data addresses to be read from the memory can only be defined if the instruction $I \equiv I(c, w_I)$ is fetched successfully, meaning there were no exceptions raised before we get interested in decoding of the fetched instruction. This set is not empty in case of the compare-and-swap or load word instructions.

$$AR(c, in) \overset{def}{\equiv} \begin{cases} \{pmaD(c, w_D, I)\}_4 & : \ load(I) \\ \emptyset & : \ \text{otherwise} \end{cases}$$

Now, depending on the interrupt handled by the core we define a set $reads_{core\text{-}mem}(c, in)$ computing instruction and data memory addresses needed for the core step without taking into account accesses to the page table for the address translation. For example, in case of an external interrupt, instruction address misalignment, or a page fault on fetch according to the MIPS-86 ISA model semantics we are not interested in any data fetched/read from the memory. In the absence of any interrupts and depending whether the instruction reads from the memory, the $reads$-set includes the fetch and read addresses defined above. Using the shorthand $il \equiv il(core, I, eev, pff(c, w_I), pfls(c, w_D, I))$, we compute $reads_{core\text{-}mem}(c, in)$ in the following way

$$reads_{core\text{-}mem}(c, in) \overset{def}{\equiv} \begin{cases} \emptyset & : \ il \in \{reset, I/O, pff\} \vee \\ & \quad\ il = mal \wedge imal(core) \\ AF(c, in) & : \ il = mal \wedge dmal(core, I) \vee il = pfls \\ AF(c, in) \cup AR(c, in) & : \ \text{otherwise} \end{cases}$$

Note here, that there is a difference between pff and $pff(c, w_I)$. The former denotes the synonym $pff \equiv 4$ earlier introduced in Table 3.5 for the corresponding interrupt level and indicates the internal interrupt event signal mutual exclusive in relation to the other iev. Its computation uses $pff(c, w_I)$ coming from the MMU.

As we know from the semantics of the MIPS-86 model, during a core step in user mode we always try to extend a matching walk if it is incomplete. Therefore, we also define sets of byte addresses $reads_{core\text{-}fpte}(c, in)$ and $reads_{core\text{-}dpte}(c, in)$ for reading required page table entries for instruction fetch and data access respectively. For shortness we make the definitions under conditions following from the MIPS-86 semantics. For any other cases not satisfying them the sets are obviously empty.

$$mode(core) \wedge w_I \in c.cpu.tlb \wedge hit\,(trqI(core), w_I) \wedge \neg complete(w_I) \implies$$
$$reads_{core\text{-}fpte}(c, in) \overset{def}{\equiv} \{ptea(w_I)\}_4$$

$$\begin{gathered} mode(core) \wedge w_I \in c.cpu.tlb \wedge hit\,(trqI(core), w_I) \wedge complete(w_I) \wedge \\ mem(I) \wedge w_D \in c.cpu.tlb \wedge hit\,(trqD(core, I), w_D) \wedge \neg complete(w_I) \implies \\ reads_{core\text{-}dpte}(c, in) \overset{def}{\equiv} \{ptea(w_D)\}_4 \end{gathered}$$

Putting it all together we easily get the set of all byte addresses to be read during the processor core step with the input in:

$$reads_{core}(c, in) \overset{def}{\equiv} reads_{core\text{-}fpte}(c, in) \cup reads_{core\text{-}dpte}(c, in) \cup reads_{core\text{-}mem}(c, in)$$

For a TLB step indicated by $in = (\texttt{add-accessed}, w)$ or $in = (\texttt{add-extend}, w)$ we introduce

$$reads_{tlb}(in) \overset{def}{\equiv} \{ptea(w)\}_4$$

Now, depending on the input in showing which processor component makes a step, we define the *reads*-set for the MIPS-86 model.

$$reads_{\mathrm{MIPS}}(c, in) \overset{def}{\equiv} \begin{cases} reads_{core}(c, in) & : in = (\texttt{core}, w_I, w_D, eev) \\ reads_{tlb}(in) & : in = (\texttt{add-accessed}, w) \vee \\ & in = (\texttt{add-extend}, w) \\ \emptyset & : \text{otherwise} \end{cases}$$

As the instantiation for the *Cosmos* model with MIPS-86 we set

$$S_{\mathrm{MIPS}}.reads(u, m, in) = reads_{\mathrm{MIPS}}((u, m), in)$$

- $S_{\mathrm{MIPS}}.\delta : \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V}) \times \mathcal{E} \rightharpoonup \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V})$ – The instantiation of the transition function is based on the single core MIPS-86 transition function. As follows from the definition, the *Cosmos* model transitions affects the unit's configuration and the portion of memory to be written.

  We define the set of addresses to be written from the point of view of the processor. Since the stores with sw go to the processor's store buffer, they do not appear in the memory component. However, to be able to apply the safety policy explicitly for addresses written to the store buffer we include these addresses into the aforementioned set. This will be also the case for the reduced machine where any writes of the hypervisor/kernel are applied for the abstract memory without the store buffer.

  For $in = (\texttt{core}, w_I, w_D, eev)$, $c \in \mathbb{C}_{\mathrm{MIPS}}$, as well as shorthands $I \equiv I(c, w_I)$, $core \equiv c.cpu.core$, and $jisr \equiv jisr(core, I, eev, pff(c, w_I), pfls(c, w_D, I))$ we introduce predicates $swap(c, in)$ indicating if the cas-instruction writes to the memory, and $wr(c, in)$ showing that the core performs a write operation to the memory or store buffer:

$$swap(c, in) \overset{def}{\equiv} cas(I) \wedge core.gpr(rd(I)) = c.m_4(pmaD(c, w_D, I))$$

$$wr(c, in) \overset{def}{\equiv} \neg jisr \wedge (sw(I) \vee locksw(I) \vee swap(c, in))$$

$$writes_{core}(c, in) \overset{def}{\equiv} \begin{cases} \{pmaD(c, w_D, I)\}_4 & : wr(c, in) \\ \emptyset & : \text{otherwise} \end{cases}$$

For a TLB step indicated by $in = (\texttt{add-accessed}, w)$ we introduce

$$writes_{tlb}(in) \overset{def}{\equiv} \{ptea(w))\}_4$$

In case of a store buffer step we have

$$writes_{sb}(c) \overset{def}{\equiv} \{c.cpu.sb[|c.cpu.sb|].a \circ 00\}_4$$

So, the $writes$-set for the MIPS-86 model is then defined in the following way:

$$writes_{\text{MIPS}}(c, in) \overset{def}{\equiv} \begin{cases} writes_{core}(c, in) & : \ in = (\texttt{core}, w_I, w_D, eev) \\ writes_{tlb}(in) & : \ in = (\texttt{add-accessed}, w) \\ writes_{sb}(c) & : \ in = \texttt{sb} \\ \emptyset & : \ \text{otherwise} \end{cases}$$

Note that the definition will be used only when the transition function is defined.

To instantiate the unit transition function for the *Cosmos* model, we need to transform the partial memory $m : \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$ defined by the *reads*-set to the type $\mathbb{C}_m$ that is a total mapping. Such a memory $\lceil m \rceil$ maps addresses outside of the *reads*-set to some dummy values, which are not accessed if the *reads*-set is instantiated properly and are introduced only to match the formal definitions.

Now, depending whether the single core MIPS-86 transition is defined or not, we instantiate the *Cosmos* model unit transition function as follows

$$S_{\text{MIPS}}.\delta(u, m, in) = \begin{cases} \left(u', m'|_{writes_{\text{MIPS}}((u, \lceil m \rceil), in)}\right) & : \ \delta_{\text{MIPS}}((u, \lceil m \rceil), in) = (u', m') \\ undefined & : \ \text{otherwise} \end{cases}$$

- $S_{\text{MIPS}}.\mathcal{IP} : \mathcal{U} \times (\mathcal{R} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow \mathbb{B}$ – Since for the store buffer reduction we will provide the simulation relation for every step and do not apply the order reduction we will consider as an interleaving point any MIPS-86 configuration from which the transition exists for a given input. For simplicity, we set the predicate to true for any input parameters of the transition because we are interested in the interleaving points only in cases of the transition step existence.
$$S_{\text{MIPS}}.\mathcal{IP}(u, m, in) = 1$$
- $S_{\text{MIPS}}.\mathcal{IO} : \mathcal{U} \times (\mathcal{R} \rightarrow \mathcal{V}) \times \mathcal{E} \rightarrow \mathbb{B}$ – The instantiation of $\mathcal{IO}$-points depends on whether a guest or the hypervisor/kernel makes a step.

  For our simple store buffer reduction discipline we do not allow a shared write of the hypervisor/kernel to be in the SB. Therefore, we treat the instruction sw in system mode as a local step.

  As for the guest/user steps we will consider any processor step in user mode as suitable for $\mathcal{IO}$-operation without looking at the exact operation performed by the processor. In fact, definition of the $\mathcal{IO}$-points must depend on the read-only memory, what is not true from the hypervisor point of view because of the address translation. Therefore, technically in the *Cosmos* model we cannot identify the memory instruction performed by the core in user mode.

  Moreover, in the work later when we will apply the order reduction for the hypervisor/kernel, we will treat all guest/user steps in the same manner and preserve their order in the global schedule. The reason is that safety policies of guest/users are in general unknown for us.

Sticking to the previously used shorthands we define

$$\begin{aligned}\mathcal{IO}_{\mathrm{MIPS}}(c, in) \overset{def}{\equiv}\; & \neg mode(core) \wedge in = (\texttt{core}, w_I, w_D, eev) \wedge \\ & \neg jisr \wedge mem(I) \wedge \neg sw(I) \vee \\ & mode(core)\end{aligned}$$

$$S_{\mathrm{MIPS}}.\mathcal{IO}(u, m, in) = \mathcal{IO}_{\mathrm{MIPS}}((u, \lceil m \rceil_{\mathbb{B}^{32}}), in)$$

Note here that extending the memory with dummy values does not influence the predicate.

- $S_{\mathrm{MIPS}}.\mathcal{OT} : \mathcal{U} \times (\mathcal{R} \to \mathcal{V}) \times \mathcal{E} \to \mathbb{B}$ – We allow to transfer the ownership only at hypervisor/kernel operations flushing the store buffer so that data at released addresses becomes available for other processors.

$$\mathcal{OT}_{\mathrm{MIPS}}(c, in) \overset{def}{\equiv} \neg mode(core) \wedge in = (\texttt{core}, w_I, w_D, eev) \wedge \neg jisr \wedge (cas(I) \vee locksw(I))$$

$$S_{\mathrm{MIPS}}.\mathcal{OT}(u, m, in) = \mathcal{OT}_{\mathrm{MIPS}}((u, \lceil m \rceil_{\mathbb{B}^{32}}), in)$$

This instantiation trivially guarantees that the property $insta_{ot}(S_{\mathrm{MIPS}})$ is not violated.

4.3.2 Reduced Machine Instantiation

For the store buffer reduced MIPS-86 machine the instantiation of the *Cosmos* model is similar except those parameters that are based on the transition function. We denote this *Cosmos* model as $S_{r\mathrm{MIPS}} \in \mathbb{S}$ and consider its signature.

- For components $X \in \{\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \mathcal{IP}, \mathcal{IO}, \mathcal{OT}\}$ the instantiation is equal to the one for the reference MIPS-86

$$S_{r\mathrm{MIPS}}.X = S_{\mathrm{MIPS}}.X$$

- The instantiation of δ is now based on the transition function of the store buffer reduced single core MIPS-86 machine:

$$S_{r\mathrm{MIPS}}.\delta(u, m, in) = \begin{cases} (u', m'|_{writes_{\mathrm{MIPS}}((u,\lceil m \rceil),in)}) & : \delta_{r\mathrm{MIPS}}((u, \lceil m \rceil), in) = (u', m') \\ undefined & : \text{otherwise} \end{cases}$$

As we have mentioned above, we use here the same definitions of the $reads$- and $writes$ sets as for the reference MIPS-86 machine.

4.4 Store Buffer Reduction Correctness

Now we can justify the SB reduced multi-core MIPS-86 model by showing the existence of its steps encoded by any steps of the reference machine under certain conditions. We proceed exactly in the way introduced for the concurrent simulation in Chapter 2.

4.4.1 Simulation for the Single Core Machine

First, we define the simulation relation for the store buffer reduction from the point of view of a singe processor in the model and trivially prove the simulation for any step of the single core machine. This case corresponds to the sequential simulation theorem for the *Cosmos* machine where the ownership is not considered yet.

Definition 4.3 (Single Core MIPS-86 Simulation Relation for SB Reduction). For $c, c_r \in \mathbb{C}_{\text{MIPS}}$ such that c is a configuration of the reference single core MIPS-86 machine and c_r - the store buffer reduced machine, and a subset set of byte addresses $icm \subset A_{hyp}$ representing a possible inconsistent memory region of the hypervisor/kernel, we define the simulation relation $sim_{r\text{MIPS}}(c, c_r, bm)$ between two machines denoting that

- the configurations of the cores and TLBs of both machines are equal,
- the SB of the reduced machine is empty in system mode and is equal to the SB of the reference machine in user mode,
- the guests/users memory region of the reduced machine is equal to this region in the physical memory of the reference machine,
- the content of the hypervisor/kernel memory excluding a possible inconsistent region resides in the SB or the physical memory of the reference machine in system mode, and only in the physical memory in user mode.

$$sim_{r\text{MIPS}}(c, c_r, icm) \overset{def}{\equiv}$$

$$\begin{array}{ll}
\textit{(i)} & c_r.cpu.core = c.cpu.core \\
\textit{(ii)} & c_r.cpu.tlb = c.cpu.tlb \\
\textit{(iii)} & c_r.cpu.sb = \begin{cases} \epsilon & : \neg mode(c_r.cpu.core) \\ c.cpu.sb & : \text{otherwise} \end{cases} \\
\textit{(iv)} & \forall a \in A_{guest}.\ c_r.m(a) = c.m(a) \\
\textit{(v)} & \forall a \in A_{hyp} \setminus icm.\ c_r.m(a) = \begin{cases} ms(c.cpu.sb, c.m)(a) & : \neg mode(c_r.cpu.core) \\ c.m(a) & : \text{otherwise} \end{cases}
\end{array}$$

Note, that the last is important for the MMU steps when after interrupt handling the processor switches back to user mode. Moreover, when we will use this simulation relation in the concurrent setting, we will set icm to the locally owned addresses of other processors since the memory values at such addresses might be in their store buffers.

We introduce also a technical invariant stating that there are no pending writes to icm and guest/user memory in the store buffer of the processor in system mode.

Definition 4.4 (SB Invariant). For a processor configuration $cpu \in \mathbb{C}_{proc}$ and a set $icm \subset A_{hyp}$ we define

$$inv_{sb}(cpu, icm) \overset{def}{\equiv} \neg mode(cpu.core) \implies \forall a \in icm \cup A_{guest}.\ \neg sbhit(cpu.sb, a[31:2])$$

Obviously, as one of conditions in the sequential simulation theorem we require that icm is not accessed by the reduced machine.

Definition 4.5 (No Access to icm by Single Core Reduced Machine). For a step of the single core reduced machine from a configuration $c_r \in \mathbb{C}_{\mathrm{MIPS}}$ and determined by an input $in_r \in \Sigma_{\mathrm{MIPS}}$ we define

$$noacc_{r\mathrm{MIPS}}(c_r, in_r, icm) \overset{def}{\equiv} (reads_{\mathrm{MIPS}}(c_r, in_r) \cup writes_{\mathrm{MIPS}}(c_r, in_r)) \cap icm = \emptyset$$

Moreover, following the ideas from the discussion at the beginning of the chapter we formulate the software conditions that are enough for proving our simulation relation in the sequential context. For the concurrent context later we will additionally rely on the ownership policy.

Definition 4.6 (Software Conditions for SB Reduction). For a step of the single core reduced machine from a configuration $c_r \in \mathbb{C}_{\mathrm{MIPS}}$ and determined by an input $in_r \in \Sigma_{\mathrm{MIPS}}$ we require that in user mode the processor core and SB do not access the hypervisor/kernel addresses, and the hypervisor/kernel is not allowed to write to the guest/user memory by sw instruction.

$$\begin{array}{lll} sc_{r\mathrm{MIPS}}(c_r, in_r) \overset{def}{\equiv} & (i) & mode(c_r.cpu.core) \wedge in_r = (\mathtt{core}, w_I, w_D, eev) \implies \\ & & reads_{core\text{-}mem}(c_r, in_r) \cup writes_{core}(c_r, in_r) \subset A_{guest} \\ & (ii) & mode(c_r.cpu.core) \wedge in_r = \mathtt{sb} \implies writes_{sb}(c_r) \subset A_{guest} \\ & (iii) & \neg mode(c_r.cpu.core) \wedge in_r = (\mathtt{core}, w_I, w_D, eev) \wedge \\ & & writes_{core}(c_r, in_r) \cap A_{guest} \neq \emptyset \implies \neg sw(I(c_r, w_I)) \end{array}$$

The following theorem justifies our store buffer reduction discipline for any step of the single core MIPS-86 and is needed for establishing the concurrent simulation. We also guarantee the correspondence of the $\mathcal{IO}$- and $\mathcal{OT}$-steps between the reduced and reference machines according to the requirement $one\mathcal{IO}$ from Chapter 2.

Theorem 4.1 (SB Reduction for a Step of the Single Core MIPS-86).

$$\begin{array}{l} \forall c, c', c_r \in \mathbb{C}_{\mathrm{MIPS}}, in \in \Sigma_{\mathrm{MIPS}}, icm \in 2^{\mathbb{B}^{32}}. \\ \quad (i) \quad icm \subset A_{hyp} \wedge sim_{r\mathrm{MIPS}}(c, c_r, icm) \wedge inv_{sb}(c.cpu, icm) \\ \quad (ii) \quad c' = \delta_{\mathrm{MIPS}}(c, in) \\ \quad (iii) \quad \forall in_r \in \Sigma_{\mathrm{MIPS}}, c'_r \in \mathbb{C}_{\mathrm{MIPS}}.\ c'_r = \delta_{r\mathrm{MIPS}}(c_r, in_r) \implies \\ \qquad\qquad noacc_{r\mathrm{MIPS}}(c_r, in_r, icm) \wedge sc_{r\mathrm{MIPS}}(c_r, in_r) \\ \quad \implies \\ \quad \exists c'_r \in \mathbb{C}_{\mathrm{MIPS}}. \\ \qquad (i) \quad c'_r = c_r \wedge \neg\mathcal{OT}_{\mathrm{MIPS}}(c, in) \vee \\ \qquad\qquad \exists in_r \in \Sigma_{\mathrm{MIPS}}.\ c'_r = \delta_{r\mathrm{MIPS}}(c_r, in_r) \wedge \\ \qquad\qquad\qquad \mathcal{IO}_{\mathrm{MIPS}}(c_r, in_r) \implies \mathcal{IO}_{\mathrm{MIPS}}(c, in) \wedge \\ \qquad\qquad\qquad \mathcal{OT}_{\mathrm{MIPS}}(c, in) \iff \mathcal{OT}_{\mathrm{MIPS}}(c_r, in_r) \\ \qquad (ii) \quad sim_{r\mathrm{MIPS}}(c', c'_r, icm) \wedge inv_{sb}(c'.cpu, icm) \end{array}$$

Proof: The theorem is proven by a case split on the step $c' = \delta_{\mathrm{MIPS}}(c, in)$ performed by the processor of the reference machine in translated and untranslated modes. As icm we choose any set $icm \subset A_{hyp}$ of addresses that cannot be covered by the simulation relation at the beginning of the step and are not accessed during the step in the reduced machine.

For the proof we introduce a few shorthands. Namely, for the configurations c, c_r, c', c'_r and processor components $X \in \{core, sb, tlb\}$ we denote $X \equiv c.cpu.X$, $X_r \equiv c_r.cpu.X$, $X' \equiv c'.cpu.X$, $X'_r \equiv c'_r.cpu.X$.

Since in the starting configurations the cores of both machines are coupled, the modes are also trivially equivalent $mode(core_r) = mode(core)$.

First, we consider a **hypervisor/kernel** step performed in **untranslated mode** on the reference machine. From the semantics of the single core MIPS-86 model we know that only the SB and core can make a step and in case of a core step the input walks are ignored.

1. **Store buffer step**: $in = \texttt{sb}$
   The SB writes a word to the memory at the address a such that $sbhit(c.cpu.sb, a[31:2])$ holds. All other processor components are not changed. From $inv_{sb}(c.cpu, icm)$ we know $a \notin icm \cup A_{guest}$. Therefore, the guest memory in $c'.m$ is not changed and the buffer commits the store to the portion of the hypervisor/kernel memory covered by the simulation relation, namely $c_r.m_4(a) = ms_4(sb, c.m)$. An empty step of the reduced machine $c'_r = c_r$ preserves the simulation relation and we have $c'_r.m_4(a) = ms_4(sb', c'.m)$. Moreover, $inv_{sb}(c'.cpu, icm)$ still holds because the SB contains now one pending store less.

2. **Core step**: $in = (\texttt{core}, w_I, w_D, eev)$
   Along with the step of the reference machine we consider the core step of the reduced machine with the same input $in_r = in$ and $c'_r = \delta_{r\text{MIPS}}(c_r, in_r)$.

   In case of an external interrupt or the instruction misalignment no memory access is performed and the cores of both machines make the same non-empty step (which is also not an $\mathcal{IO}$-step) preserving the simulation relation. Note that by $reset$ the TLBs are flushed and become empty. Moreover, in the reference machine for the $reset$ to be handled by the core we require the SB to be already empty, meaning that the SB must have made its steps if it had pending stores.

   Otherwise, the processors read the same instruction $I = ms_4(c)(core.pc) = m_4(c_r)(core_r.pc)$ from the consistent portion of memory what follows from the simulation relation and the predicate $noacc_{r\text{MIPS}}(c_r, in_r, icm)$.

   If we have $\neg mem(I)$ or $mem(I) \wedge dmal(core, I)$, the instruction does not read/write from/to the memory or the store buffer, and the cores perform the equivalent step. The TLBs are flushed in the same manner for $flush$ and $invlpg$ instructions. In case of the $mfence$ instruction the core of the reference machine makes the step only if the SB has written back its pending stores. In the reduced machine the execution of $mfence$ affects only the core. Therefore for the aforementioned cases the theorem trivially holds.

   If we have $\neg mode(core) \wedge emode(core) \wedge eret(I)$, then the requirement on the store buffer of the reference machine is the same as for $mfence$, $sb = \varepsilon$, and one obviously has $\forall a \in A_{hyp} \setminus icm.\ m_r(a) = m(a)$. On the other hand, in the reduced machine from the simulation relation we also have $sb_r = \varepsilon$, concluding $sb'_r = sb' = \varepsilon$. Both of the cores are in user mode in c' and c'_r, the coupling relation as well as the theorem hold.

   For other instructions I we have to distinguish between lw, sw, cas, and $locksw$:

   - **data word read** $lw(I)$
     The reference machine reads a word $R = ms_4(c)(ea(core, I))$ from the memory or the store buffer. The reduced machine gets also the data $R_r = c_r.m(ea(core_r, I))$ at the same address $ea(core_r, I) = ea(core, I)$ because the cores are equivalent. From $noacc_{r\text{MIPS}}(c_r, in_r, icm)$ and the simulation relation we conclude that both machines read the same data $R = R_r$ from the consistent memory. Therefore, the cores perform the same step leading to the consistent configurations, s.t $sim_{r\text{MIPS}}(c', c'_r, icm)$ holds.

The configuration of the store buffer does not change and support the invariant $inv_{sb}(c'.cpu, icm)$. Moreover, for $lw(I)$ we have $\mathcal{IO}_{\text{MIPS}}(c, in) = \mathcal{IO}_{\text{MIPS}}(c_r, in_r) = 1$ and $\mathcal{OT}_{\text{MIPS}}(c, in) = \mathcal{OT}_{\text{MIPS}}(c_r, in_r) = 0$, what concludes the claim.

- **data word write** $sw(I)$
  As argued above in both machines the computed effective addresses are equal and out of icm. From the software condition (iii) we know that the hypervisor/kernel does not write to the guest memory by $sw(I)$. Therefore, both machines write the data word at the byte addresses belonging to the consistent portion $A_{hyp} \setminus icm$. In the reference machine the write goes to the store buffer preserving $inv_{sb}(c'.cpu, icm)$. The reduced machine writes the same word to its memory. Therefore, one easily concludes the memory consistency $\forall a \in A_{hyp} \setminus icm.\ c'_r.m(a) = ms(sb', c'.m)(a)$. Moreover, one has $\mathcal{IO}_{\text{MIPS}}(c, in) = \mathcal{IO}_{\text{MIPS}}(c_r, in_r) = 0$. The theorem holds for this case.
- **locked write** $locksw(I)$
  In comparison to sw, the execution of $locksw$ requires the store buffer of the reference machine to be empty and writes directly to the memory in both machines. As in the previous case the machines store the same word to the consistent portion of the memory, however, including in this case not only the hypervisor addresses but also the guest region. Since the SB of the reference machine does not receive a new pending store, the theorem obviously holds.
- **compare-and-swap** $cas(I)$
  Analogously to the previous step, the core step for $cas(I)$ in the reference machine is made when its SB is empty. After the transition one still has $sb'_r = sb' = \varepsilon$. In contrast to the previously considered lw the read result R is ignored by the core, and the operation is performed atomically by the memory, namely

  $$c'.m = \delta_m(c.m, (core.gpr(rd(I)), ea(core, I), sv(core, I))).$$

  From the software conditions and $noacc_{r\text{MIPS}}(c_r, in_r, icm)$ we already know that the byte addresses to be read and written belong to the guest or hypervisor memory consistent for the machines and excluding icm. Therefore, it is easy to conclude that the memory transition in both machines is the same, and the relation $sim_{r\text{MIPS}}(c', c'_r, icm)$ holds. Since the SB is not changed, its invariant is preserved. Additionally, the cas operations is considered to be an $\mathcal{IO}$-step suitable for the ownership transfer.

  This last case finishes the proof of the theorem for a hypervisor/kernel step.

Now, we consider a **guest/process step** in **translated mode**. In this case any processor component can perform a transition depending on the input in and the machine's semantics. Since we mark any guest/process step as an $\mathcal{IO}$-point even in the case of a non-memory operation, and no such steps are suitable for the ownership transfer, we have only to prove that the reduced machine makes also a corresponding non-empty step in translated mode, namely $\exists in_r \in \Sigma_{\text{MIPS}}.\ c'_r = \delta_{r\text{MIPS}}(c_r, in_r)$ such that $sim_{r\text{MIPS}}(c', c'_r, icm)$ holds.

From the first look, both machines are equal and should preserve the simulation relation if $in_r = in$. However, one has to argue about the equivalence of data read and the instruction fetched from the memory and show that the memory consistency is not violated. Not to miss the details we consider the following cases:

1. **Walk creation**: $in = (\texttt{tlb-create}, va)$.
   The TLB of the reference machine creates an initial walk for a chosen virtual address va

and no memory operation is performed. Since the TLBs of both machines are equal before and after the step, the theorem holds.

2. **Setting the accessed bit**: $in = (\texttt{add-accessed}, w)$.
   From the reference machine transition function this step is defined when the conditions $w \in c.cpu.tlb \wedge \neg complete(w) \wedge w.asid = asid(c.cpu.core)$ hold and the page table entry is present: $pte(c.m, w).p$. For the reduced machine to perform the same step one has to show $pte(c_r.m, w) = pte(c.m, w)$. This is true indeed, because from $noacc_{r\text{MIPS}}(c_r, in_r, icm)$ we know that the accessed pte resides in the guest or hypervisor memory covered by the consistency relation. So, both machines perform the same transition setting the bit in the memory and the claims hold.

3. **Walk extension**: $in = (\texttt{add-extend}, w)$
   Additionally to the conditions of the previous case, the accessed flag of the page table entry must be set $pte(c.m).a$ and an extended walk $w' = wext(w, pte(c.m, w), 000)$ must be not faulty $\neg w'.fault$. As shown above, the read pte is the same for the machines, therefore, they make the same step preserving the relation.

4. **Store buffer step**: $in = \texttt{sb}$
   In case of a store buffer transition of the non-reduced machine the reduced machine makes the same step. From the software condition (ii) the write of the memory is made at an address belonging to the guest. Therefore, the simulation holds also after the step.

5. **Core step**: $in = (\texttt{core}, w_I, w_D, eev)$
   For a core step in translated mode one has to consider the address translation for the instruction address, and for the effective address if the instruction address translation succeeds and the fetched instruction accesses the memory.

   If the walk w_I present in the TLB and matching the translation request is incomplete, similarly to the case of the TLB setting accessed bit one can show that in both machines the TLB reads the same page table entry $pte(c_r.m, w_I) = pte(c.m, w_I)$.

   Therefore, for the complete or incomplete walk w_I one easily concludes

   $$pff = pff(c_r, w_I) = pff(c, w_I).$$

   Now, for the complete walk w_I we calculate the same translated address $pmaI(c_r, w_I) = pmaI(c, w_I)$ which according to the software condition (i) refers to the instruction in the guest memory. From the simulation relation we then get $I = I(c_r, w_I) = I(c, w_I)$.

   Again, as above, for the complete w_I and incomplete w_D present in the TLB and matching the data translation request we get $pte(c_r.m, w_D) = pte(c.m, w_D)$ and

   $$pfls = pfls(c_r, w_D, w_I, I) = pfls(c, w_D, w_I, I).$$

   The physical address $pmaD(c_r, w_D, I) = pmaD(c, w_D, I)$ computed in case of $complete(w_D)$ analogously points to a word residing in the guest memory. Hence, the word accessed at this address is equal in both of the machines.

   Now, we consider how according to the semantics the components of the single core MIPS-86 model are changed during the core step. First, we pay attention on the TLB.

   For the incomplete walk w_I with the faulty extension the TLBs of the both machines are flushed in the same way independently from the reaction of the core. In turn, the walk w_D is ignored by the TLB. Therefore, we get $tlb'_r = tlb'$.

If w_I is complete and $mem(I)$ holds, but the incomplete w_D causes a faulty extension, then the TLBs are flushed similarly for the effective address and are coupled.

The transition of the processor core and passive changes on other components but the TLB depend on whether an interrupt is raised in the core or not.

- $jisr(core, I, eev, pff, pfls)$:
  In this case the core switches to system mode as a result of the step. Since in both machines the cores are equal and we have proven that the instruction as well as the page fault signals from the TLBs are the same, then

  $$il(core_r, I, eev, pff, pfls) = il(core, I, eev, pff, pfls).$$

  Hence, the cores handle the same interrupt and are coupled after the step. The memory is not changed. As for the store buffers, in case of the interrupt the transition function of MIPS-86 is only defined when $sb_r = sb = \varepsilon$. This obviously concludes the claim for $sim_{r\text{MIPS}}(c', c'_r, icm)$.
- $\neg jisr(core, I, eev, pff, pfls)$:
  In the absence of interrupts in the reference machine, the reduced machine performs the same step. If the instruction I does not interfere with the memory or store buffer, only the core configurations are changed and equal after the transition. The instructions $flush$ and $invlpg$ were covered in the previous case because they cause the illegal instruction interrupt in user mode. The instruction $mfence$ can be performed if the store buffers are empty, and, therefore, does not change anything except the program counter. For any instruction I with $mem(I)$ we have already proven above that it accesses a word in the coupled guest memory, and, as a result, doest not destroy the simulation relation for the hypervisor portion of memory. We conclude that all components in both MIPS-86 machines are changed in the same way, what, in turn, preserves $sim_{r\text{MIPS}}(c', c'_r, icm)$.

$\square$

Now, to establish the concurrent simulation for any unit of the *Cosmos* models instantiated with the reduced and reference MIPS-86, we properly instantiate the sequential simulation framework $R^{S_{r\text{MIPS}}}_{S_{\text{MIPS}}} \in \mathbb{R}$. Since there is no specific simulation parameter, for simplicity we drop the parameter equal to $\perp$ in all predicates. So, for $c, c_r \in \mathbb{C}_{proc} \times \mathbb{C}_m$, $cpu, cpu_r \in \mathbb{C}_{proc}$, $in, in_r \in \Sigma_{\text{MIPS}}$ where the subscript r indicates the reduced machine, and $icm \in 2^{\mathbb{B}^{32}}$, we have

$$R^{S_{r\text{MIPS}}}_{S_{\text{MIPS}}}. \begin{cases} \mathcal{P} & = \perp \\ sim(c, c_r, icm) & = icm \subset A_{hyp} \wedge sim_{r\text{MIPS}}(c, c_r, icm) \wedge inv_{sb}(c.cpu, icm) \\ sc(c_r, in_r) & = sc_{r\text{MIPS}}(c_r, in_r) \\ X & = 1 \text{ for } X \in \{\mathcal{CP}a(cpu_r), \mathcal{CP}c(cpu),\ wfa(c_r), \\ & \qquad\qquad wfc(c), suit(in), wb(c, in)\} \end{cases}$$

Since the theorem just proven above fully corresponds to the statement of Theorem 2.3 for $R^{S_{r\text{MIPS}}}_{S_{\text{MIPS}}}$, and we consider the simulation for every step of the reference machine such that an empty step of the reduced machine is treated as an empty consistency block for the *Cosmos* machine transition, it is easy to show that using Theorem 4.1 one also proves Theorem 2.3 for our case. For brevity, we leave this bookkeeping here as a trivial exercise.

4.4.2 SB Reduction for the Multi-Core MIPS-86

Finally, to prove store buffer reduction for the multi-core MIPS-86 machine in the presence of the ownership model, we define the invariants needed according to Chapter 2 and show that the required assumptions hold.

Definition 4.7 (Shared Invariant for Multi-Core MIPS-86 SB Reduction). For the *Cosmos* models instantiated with the reference and reduced MIPS-86 machines we demand the equality of their sets of shared $\mathcal{S}$, $\mathcal{S}_r$, read-only $\mathcal{R}$, $\mathcal{R}_r$, and owned $\mathcal{O}(p)$, $\mathcal{O}_r(p)$ by each unit p addresses, as well as the contents of shared and read-only memories m, m_r.

$$sinv_{S_{\text{MIPS}}}^{S_{r\text{MIPS}}}((m, \mathcal{S}, \mathcal{R}, \mathcal{O}), (m_r, \mathcal{S}_r, \mathcal{R}_r, \mathcal{O}_r)) \overset{def}{\equiv}$$

$$\begin{array}{ll} \textit{(i)} & \mathcal{S} = \mathcal{S}_r \\ \textit{(ii)} & \mathcal{R} = \mathcal{R}_r \\ \textit{(iii)} & \forall p \in \mathbb{N}_{nu}.\ \mathcal{O}(p) = \mathcal{O}_r(p) \\ \textit{(iv)} & m = m_r \end{array}$$

As we have already seen, we forbid the guest/process to modify the hypervisor/kernels's ownership. However, so far the hypervisor is able to acquire the guest/user addresses, what, in term, might restrict guest/user accesses because we use the same safety policy for both kinds of steps. Since the guest/process steps should be also safe and we would like to allow guests and processes always to access their memory, we need to guarantee that the hypervisor/kernel never acquires anything from A_{guest}.

Definition 4.8 (Restriction on the Ownership Transfer in SB Reduced Machine). For any configuration $E \in \mathbb{C}_{S_{r\text{MIPS}}}$ of the SB reduced *Cosmos* machine we require that A_{guest} is always treated as shared and no addresses from A_{guest} can be owned by any execution unit.

$$P_{S_{r\text{MIPS}}}(E) \overset{def}{\equiv} A_{guest} \subset E.\mathcal{S} \wedge \forall p \in \mathbb{N}_{nu}.\ E.\mathcal{O}_p \cap A_{guest} = \emptyset$$

Note, that with this setting the software condition (iii) not allowing the hypervisor/kernel to access A_{guest} by the instruction sw becomes redundant because the memory access policy for our instantiated *Cosmos* model already covers this case. To avoid additional bookkeeping, however, we leave the software conditions unmodified.

From Definition 4.4 of the SB invariant being applied in the concurrent simulation we know that the SB of a processor in system mode does not contain stores to A_{guest} and other processors' local addresses. Nevertheless, we need to extend it also for the *Cosmos* model. The SB invariant together with its extension must guarantee that in system mode the store buffer contains only writes local for the processor.

Definition 4.9 (Extention of the SB Invariant for the *Cosmos* machine). For any processor being in system mode in a configuration $cpu \in \mathbb{C}_{proc}$ in the reference *Cosmos* machine with shared addresses $\mathcal{S}$ we additionally demand that its SB does not contain pending writes to the hypervisor/kernel shared and read-only memory.

$$\begin{aligned} extinv_{sb}(cpu, \mathcal{S}) \overset{def}{\equiv} & \neg mode(cpu.core) \implies \\ & \quad \forall a \in A_{code} \cup A_{const} \cup (\mathcal{S} \setminus A_{guest})\ .\ \neg sbhit(cpu.sb, a[31:2]) \end{aligned}$$

Therefore, we instantiate the concrete machine unit invariant from Chapter 2 as follows:

$$uinv_{S_{\text{MIPS}}}^{S_{r\text{MIPS}}}(cpu, \mathcal{O}, \mathcal{S}) = extinv_{sb}(cpu, \mathcal{S})$$

Before proving the assumptions, we consider a few auxiliary lemmas.

Lemma 4.1 (Hypervisor Memory Addresses Covered by the Unit's SB Reduction Sequential Simulation). *For any configuration $E \in \mathbb{C}_{r\text{MIPS}}$ of the SB reduced MIPS* Cosmos *machine and a unit $p \in \mathbb{N}_{nu}$ the set of hypervisor memory addresses without non-shared owned addresses of all units except p is equal to the union of the hypervisor read-only, shared and owned by the unit p addresses if the ownership configuration satisfies the invariant $oinv(E)$ and the property $P_{S_{r\text{MIPS}}}(E)$.*

$$oinv(E) \wedge P_{S_{r\text{MIPS}}}(E) \implies A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) = S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest}) \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})$$

Proof: For the proof we first apply Lemma 2.8

$$S_{r\text{MIPS}}.\mathcal{A} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) = S_{r\text{MIPS}}.\mathcal{R} \cup E.\mathcal{S} \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})$$

From the *Cosmos* machine instantiation and the partitioning of the memory addresses we know $S_{r\text{MIPS}}.\mathcal{A} = \mathbb{B}^{32} = A_{guest} \cup A_{hyp}$ and $A_{guest} \cap A_{hyp} = \emptyset$. Moreover, $P_{S_{r\text{MIPS}}}(E)$ requires that the guests' addresses cannot be owned. Therefore, one trivially transforms the left-hand side

$$\begin{aligned} S_{r\text{MIPS}}.\mathcal{A} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) &= (A_{guest} \cup A_{hyp}) \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) \\ &= A_{guest} \cup \left(A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p)\right), \end{aligned}$$

where $A_{guest} \cap \left(A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p)\right) = \emptyset$.

Since $S_{r\text{MIPS}}.\mathcal{R} = A_{code} \cup A_{const}$, $S_{r\text{MIPS}}.\mathcal{R} \subset A_{hyp}$, and according to $P_{S_{r\text{MIPS}}}(E)$ the guest addresses belong to the shared memory and cannot be owned, we can re-write the right-hand side as

$$S_{r\text{MIPS}}.\mathcal{R} \cup E.\mathcal{S} \cup (E.\mathcal{O}_p \setminus E.\mathcal{S}) = A_{guest} \cup (S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest}) \cup (E.\mathcal{O}_p \setminus E.\mathcal{S}))$$

with $A_{guest} \cap (S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest}) \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})) = \emptyset$ and easily conclude

$$A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) = S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest}) \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})$$

□

Lemma 4.2 (Shared and Read-Only Memory Equality from SB Reduction Simulation Relation). *Given the concurrent simulation relation for a unit p between SB reduced and reference* Cosmos *machines with configurations E and D respectively, one can deduce the memory coupling present in the shared invariant, if the corresponding ownership states are coupled, the ownership invariant of the reduced machine and the restriction on the ownership transfer are fulfilled, and the unit invariant holds.*

$$\begin{aligned} &\forall D \in \mathbb{C}_{S_{\text{MIPS}}}, E \in \mathbb{C}_{S_{r\text{MIPS}}}, p \in \mathbb{N}_{nu}. \\ &\quad (i) \quad csim_p(D.M, E) \\ &\quad (ii) \quad D.\mathcal{S} = E.\mathcal{S} \wedge S_{\text{MIPS}}.\mathcal{R} = S_{r\text{MIPS}}.\mathcal{R} \\ &\quad (iii) \quad P_{S_{r\text{MIPS}}}(E) \wedge oinv(E) \wedge uinv_p(D) \\ &\quad \implies \\ &\quad E.m|_{E.\mathcal{S} \cup S_{r\text{MIPS}}.\mathcal{R}} = D.m|_{D.\mathcal{S} \cup S_{\text{MIPS}}.\mathcal{R}} \end{aligned}$$

Proof: Consider the SB reduction simulation relation for unit p:

$$\begin{aligned} csim_p(D.M, E) &= sim_p\left(D.M, E.M, \overline{\mathcal{SO}}(E.\mathcal{G}, p)\right) \\ &= sim_{r\text{MIPS}}\left((D.u_p, D.m), (E.u_p, E.m), \overline{\mathcal{SO}}(E.\mathcal{G}, p)\right) \end{aligned}$$

From the definition of $sim_{r\text{MIPS}}$ for unit p we get the relations between the memories of the reduced and reference MIPS machines:

$$\forall a \in A_{guest}.\ E.m(a) = D.m(a) \tag{4.1}$$

$$\forall a \in A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p).\ E.m(a) = \begin{cases} ms(D.u_p.sb, D.m)(a) & : \neg mode(E.u_p.core) \\ D.m(a) & : \text{otherwise} \end{cases} \tag{4.2}$$

Moreover, by Lemma 4.1 we know

$$A_{hyp} \setminus \overline{\mathcal{SO}}(E.\mathcal{G}, p) = S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest}) \cup (E.\mathcal{O}_p \setminus E.\mathcal{S})$$

To prove the claim $E.m|_{E.\mathcal{S} \cup S_{r\text{MIPS}}.\mathcal{R}} = D.m|_{D.\mathcal{S} \cup S_{\text{MIPS}}.\mathcal{R}}$ we distinguish cases depending on the processor mode.

1. Translated mode: $mode(E.u_p.core)$.
   Since the invariant $P_{S_{r\text{MIPS}}}(E)$ requires $A_{guest} \subset E.\mathcal{S}$, we get from equations (4.1) - (4.2)

   $$\forall a \in S_{r\text{MIPS}}.\mathcal{R} \cup E.\mathcal{S}.\ E.m(a) = D.m(a)$$

   Because we have $D.\mathcal{S} = E.\mathcal{S} \wedge S_{\text{MIPS}}.\mathcal{R} = S_{r\text{MIPS}}.\mathcal{R}$, the goal to be proven transforms to

   $$E.m|_{E.\mathcal{S} \cup S_{r\text{MIPS}}.\mathcal{R}} = D.m|_{E.\mathcal{S} \cup S_{r\text{MIPS}}.\mathcal{R}}$$

   This is exactly what we have just shown above.

2. System mode: $\neg mode(E.u_p.core)$.
   First, we concentrate on the read-only and shared hypervisor addresses. Unfolding the definition of the memory system ms in the equation (4.2) we get

   $$\forall a \in S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest})\ .\ E.m(a) = \begin{cases} sbv(D.u_p.sb, a) & : sbhit(D.u_p.sb, a[31:2]) \\ D.m(a) & : \text{otherwise} \end{cases}$$

   Since the processor of the reference machine is also in system mode (the cores are coupled via $csim_p(D.M, E)$), from $uinv_p(D)$ and the lemma hypothesis (ii) we conclude

   $$\forall a \in S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest})\ .\ \neg sbhit(D.u_p.sb, a[31:2])$$

   Therefore, the memory simulation for these addresses boils down to

   $$\forall a \in S_{r\text{MIPS}}.\mathcal{R} \cup (E.\mathcal{S} \setminus A_{guest})\ .\ E.m(a) = D.m(a)$$

   Using (4.1), the fact that the guest memory addresses are included into the shared ones, and the same argumentation about the ownership state as in the previous case, we easily conclude the claim of the lemma.

□

Now we prove a lemma corresponding to Assumption 2.1. For brevity we again skip the simulation parameter and consider only the premises needed in our case. Moreover, in order to avoid additional transformations later on, we state the lemma in the original notation from Chapter 2 on the consistency blocks though for the store buffer reduction such blocks boil down to one step transition information.

Lemma 4.3 (Safety Transfer and Invariants Preservation for Store Buffer Reduction).

$\forall D \in \mathbb{C}_{S_{\text{MIPS}}}, d' \in \mathbb{M}_{S_{\text{MIPS}}}, E, E' \in \mathbb{C}_{S_{r\text{MIPS}}}, \sigma \in \Theta^*_{S_{\text{MIPS}}}, \tau \in \Theta^*_{S_{r\text{MIPS}}}, o_\tau \in \Omega^*_{S_{r\text{MIPS}}}, p \in \mathbb{N}_{nu}.$

(i) $D.M \overset{\sigma}{\longmapsto} d' \wedge blk(\sigma, p) \wedge one\mathcal{IO}(\sigma, \tau) \wedge uinv_p(D)$

(ii) $E \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E' \wedge blk(\tau, p) \wedge P_{S_{r\text{MIPS}}}(E) \wedge safe_{P_{S_{r\text{MIPS}}}}(E, \langle \tau, o_\tau \rangle)$

(iii) $csim_p(D.M, E) \wedge sinv(D, E) \wedge csim_p(d', E')$

$\Longrightarrow$

$\exists o_\sigma \in (\Omega_{S_{\text{MIPS}}})^*, \mathcal{G}' \in \mathbb{G}_{S_{\text{MIPS}}}.$

(i) $D \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d', \mathcal{G}') \wedge safe(D, \langle \sigma, o_\sigma \rangle) \wedge uinv_p((d', \mathcal{G}'))$

(ii) $sinv((d', \mathcal{G}'), E')$

Proof: In the lemma we have to consider either empty steps in both machines, or one step of the reference machine because we deal with the store buffer reduction at each step of the MIPS model execution.

In the former case, one has only to show that $oinv(D.\mathcal{G})$ holds. This fact trivially follows from $safe(E, \varepsilon)$ and $sinv(D, E)$. Note, that $oinv(D.\mathcal{G})$ is also a part of the ownership safety for a non-empty step.

For the latter case, in Theorem 4.1 we have already seen that either a non-empty step or an empty step of the reduced machine corresponds to one step of the reference machine . The same is also easy to see from the lemma premises when we unfold the definitions of blk and $one\mathcal{IO}$. Formally, we have $|\sigma| = |\tau| = 1 \wedge \sigma_1.s = \tau_1.s = p$ or $|\sigma| = 1 \wedge \sigma_1.s = p \wedge \tau = o_\tau = \varepsilon$. Moreover, the reduced machine makes an empty step only in case of a store buffer step of the processor in system mode in the reference machine. This also follows from the consideration of steps leading to the configurations d' and E' with $csim_p(d', E')$.

To prove the lemma we make a case split on the step of the reference machine. Namely, we analyze whether it is suitable for the ownership transfer and an $\mathcal{IO}$-operation

Case 1: $\neg\sigma_1.ot$ – The reference machine performs a step that cannot be used for the ownership transfer.

For this case the reduced machine stays in the same configuration or makes a non-empty step. Therefore, using $one\mathcal{IO}(\sigma, \tau)$ and the definition of the sequence $\langle \tau, o_\tau \rangle$ we have $o_\tau = \varepsilon$ or $|o_\tau| = 1 \wedge \neg\tau_1.ot$. For the non-empty ownership transfer information sequence we easily get $o_{\tau 1} = (\emptyset, \emptyset, \emptyset)$ because of the safety policy $safe_{P_{S_{r\text{MIPS}}}}(E, \langle \tau, o_\tau \rangle)$. Therefore, for the implementation step we also set $o_{\sigma 1} = (\emptyset, \emptyset, \emptyset)$, which, in turn, does not violate the ownership transfer policy $policy_{trans}(\sigma_1.ot, D.\mathcal{O}_p, D.\mathcal{S}, D.\overline{\mathcal{O}}_p, o_{\sigma 1})$ with $D.\overline{\mathcal{O}}_p \equiv \bigcup_{q \neq p} D.\mathcal{O}_q$ and preserves the reference machine ownership state and the parts (i) - (iii) of the shared invariant after the step.

Assuming that the unit invariant $uinv_p((d', \mathcal{G}'))$ also holds after the reference machine step, we can apply Lemma 4.2 for the machine configurations $(d', \mathcal{G}')$ and E' and get the last part (iv) of the shared invariant $sinv((d', \mathcal{G}'), E')$.

Therefore, the claims left to be proven for this case are

- the transfer of the memory access policy

$$policy_{acc}(\sigma_1.io, reads_p(D, \sigma_1.in), writes_p(D, \sigma_1.in), D.\mathcal{O}_p, D.\mathcal{S}, S_{\text{MIPS}}.\mathcal{R}, D.\overline{\mathcal{O}}_p),$$

- and the unit invariant $uinv_p((d', \mathcal{G}'))$ after the step. Note that from the instantiation we have $uinv_p((d', \mathcal{G}')) = extinv_{sb}(d'.u(p), \mathcal{G}'.\mathcal{S})$.

For this we look closer at the operations matching $\neg\sigma_1.ot$.

Case 1.1: $\neg\sigma_1.io$ – The step is not matching an $\mathcal{IO}$-operation. From the definition of $\mathcal{IO}_{\text{MIPS}}$ we know that in this case the processor can only be in system mode and we have to consider either a store buffer step, or any core step except execution of the instructions $lw(I)$, $locksw(I)$, and $cas(I)$.

1. **Store buffer step**: $in = \texttt{sb}$

   As we already know, the reduced machine makes an empty step in this case. Therefore, we cannot directly transfer the memory access policy from $safe_{P_{S_{r\text{MIPS}}}}(E, \langle\tau, o_\tau\rangle)$. However, using the invariant $inv_{sb}(D.u_p, \overline{\mathcal{SO}}(E.\mathcal{G}, p))$[1], the ownership coupling in $sinv(D, E)$ as well as the unit invariant $uinv_p(D)$ we conclude

   $$\forall a \in S_{\text{MIPS}}.\mathcal{R} \cup D.\mathcal{S} \cup \overline{\mathcal{SO}}(D.\mathcal{G}, p).\ \neg sbhit(D.u_p.sb, a[31:2])$$

   Hence, since $oinv(D.\mathcal{G})$ also holds, we get

   $$\forall a \in \mathbb{B}^{32}.\ sbhit(D.u_p.sb, a[31:2]) \implies a \in D.\mathcal{O}_p \setminus D.\mathcal{S}$$

   For the store buffer step, the set $reads_p(D, \sigma_1.in)$ is empty, and $writes_p(D, \sigma_1.in)$ contains the byte addresses of the pending in the SB writes, what shows $writes_p(D, \sigma_1.in) \subseteq D.\mathcal{O}_p \setminus D.\mathcal{S}$ and proves the transfer of the memory access policy.

   Moreover, it is clear that the invariant $extinv_{sb}(d'.u(p), \mathcal{G}'.\mathcal{S})$ is not violated because the SB just commits one pending store.

2. **Core step**: $in = (\texttt{core}, w_I, w_D, eev)$

   In case of an external interrupt or the instruction misalignment no memory access is performed and the store buffer is unchanged. So, the memory access policy and the unit invariant after the step trivially hold.

   Otherwise, the processors of both machines fetch the same instruction at the same address, which, in turn, implies that the set of accessed addresses $AF((D.u_p, D.m), in)$ follow the memory access policy due to $safe_{P_{S_{r\text{MIPS}}}}(E, \langle\tau, o_\tau\rangle)$ and the coupling of the ownership states of both machines.

   If the instruction does not access the memory, or we have sw and data misalignment, the $write$-set is empty and we obviously conclude the claim of the lemma.

   Otherwise, we consider the instruction sw. The processor of the reduced machine writes to the memory $E.m$. The reference machine puts the data into its store buffer. Both operations are performed at the addresses

   $$writes_{core}((E.u_p, E.m), in) = writes_{core}((D.u_p, D.m), in)$$

   belonging to $E.\mathcal{O}_p \setminus E.\mathcal{S}$ and are safe because $safe_{P_{S_{r\text{MIPS}}}}(E, \langle\tau, o_\tau\rangle)$ holds and the ownership states are coupled.

   Moreover, the unit invariant is also preserved.

[1]It is present in $csim_p(D.M, E)$ according to the instantiation of the simulation framework $R^{S_{r\text{MIPS}}}_{S_{\text{MIPS}}}$.

Case 1.2: $\sigma_1.io$ – The step is matching an $\mathcal{IO}$-operation. The situation corresponds either to a guest step, or execution of $lw(I)$ in system mode.

In case of lw, instead of the $writes$-sets, which are empty for this instruction, we consider the $reads$-sets and the proof is similar to the step with sw. Moreover, the instruction lw does not change the SB configuration.

For any guest step, both machines perform the same operation because their configurations are equal. Since the memory access policy holds for the reduced machine and the ownership states are coupled, the access policy of the reference machine is also safe. If the machines stay in translated mode after the step, the unit invariant $uinv_p((d', \mathcal{G}'))$ is true by definition. Otherwise, after the guest step the store buffer of the non-reduced machine in system mode is empty and this invariant is also preserved.

Case 2: $\sigma_1.ot$ – The reference machine performs a step used for the ownership transfer.

This case corresponds to the execution of the instructions $locksw(I)$ and $cas(I)$ by the reference machine core in system mode. The reduced machine performs the same step updating the memory.

Since $safe_{P_{S_{r\text{MIPS}}}}(E, \langle\tau, o_\tau\rangle)$ holds and the ownership states of both machines are equal, we simply copy the ownership transfer information $o_{\sigma 1} = o_{\tau 1}$ which fulfills the ownership transfer policy for the reference machine and preserves the parts (i) - (iii) of the shared invariant after the step.

It is also easy to show as in the previous cases that the memory access policy is not violated because both machines read the instruction and write the data at the same addresses. Moreover, the unit invariant holds in the configuration after the step where the store buffer remain empty. Again, using the Lemma 4.2 for the machine configurations $(d', \mathcal{G}')$ and E', we get the last part (iv) of the shared invariant.

□

Since there is no specific well-formedness in our models, Assumptions 2.2, 2.3 hold by default and we continue with a lemma for the proof of Assumption 2.4.

Lemma 4.4 (Preservation of Simulation Relation and Unit's Invariant for SB Reduction).

$$\begin{aligned}
&\forall D, D' \in \mathbb{C}_{S_{\text{MIPS}}}, E, E' \in \mathbb{C}_{S_{r\text{MIPS}}}, p \in \mathbb{N}_{nu}.\\
&\quad \textit{(i)} \quad csim_p(D.M, E) \wedge uinv_p(D)\\
&\quad \textit{(ii)} \quad sinv(D, E) \wedge P_{S_{r\text{MIPS}}}(E) \wedge oinv(E) \wedge oinv(D)\\
&\quad \textit{(iii)} \quad sinv(D', E') \wedge P_{S_{r\text{MIPS}}}(E') \wedge oinv(E') \wedge oinv(D')\\
&\quad \textit{(iv)} \quad E \approx_p E' \wedge D \approx_p D'\\
&\quad \Longrightarrow\\
&\quad csim_p(D'.M, E') \wedge uinv_p(D')
\end{aligned}$$

Proof: According to our instantiation we have to prove the sequential simulation relation

$$sim_{r\text{MIPS}}\left((D'.u_p, D'.m), (E'.u_p, E'.m), \overline{\mathcal{SO}}(E'.\mathcal{G}, p)\right),$$

the store buffer invariant $inv_{sb}(D'.u_p, \overline{\mathcal{SO}}(E'.\mathcal{G}, p))$ with $\overline{\mathcal{SO}}(E'.\mathcal{G}, p) \subset A_{hyp}$, and the unit invariant $uinv_p(D')$.

From $csim_p(D.M, E)$ we have

$$E.u_p.core = D.u_p.core \tag{4.3}$$
$$E.u_p.tlb = D.u_p.tlb \tag{4.4}$$
$$E.u_p.sb = \begin{cases} \varepsilon & : \neg mode(E.u_p.core) \\ D.u_p.sb & : \text{otherwise} \end{cases} \tag{4.5}$$

Since $E \approx_p E'$ and $D \approx_p D'$ hold, we get $D'.u_p = D.u_p$, $E'.u_p = E.u_p$. Therefore, the equations (4.3) - (4.5) are also valid for configurations E' and D'.

The shared invariant $sinv(D', E')$ gives us the coupling of the read-only and shared memories as well as the ownership state in E' and D', what, in turn, implies

$$E'.m|_{E'.\mathcal{S} \cup S_{r\mathrm{MIPS}}.\mathcal{R}} = D'.m|_{E'.\mathcal{S} \cup S_{r\mathrm{MIPS}}.\mathcal{R}} \tag{4.6}$$

Having $A_{guest} \subset E'.\mathcal{S}$ in $P_{S_{r\mathrm{MIPS}}}(E')$, we also get $A_{guest} \subset D'.\mathcal{S}$ and, therefore,

$$\forall a \in A_{guest}.\ E'.m(a) = D'.m(a).$$

Now, we concentrate on the rest of the simulation relation, namely,

$$\forall a \in A_{hyp} \setminus \overline{\mathcal{SO}}(E'.\mathcal{G}, p).\ E'.m(a) = \begin{cases} ms(D'.u_p.sb, D'.m)(a) & : \neg mode(E'.u_p.core) \\ D'.m(a) & : \text{otherwise} \end{cases},$$

where because of $oinv(E')$ and $P_{S_{r\mathrm{MIPS}}}(E')$ the set of addresses covered by the relation is computed by Lemma 4.1 as

$$A_{hyp} \setminus \overline{\mathcal{SO}}(E'.\mathcal{G}, p) = S_{r\mathrm{MIPS}}.\mathcal{R} \cup (E'.\mathcal{S} \setminus A_{guest}) \cup (E'.\mathcal{O}_p \setminus E'.\mathcal{S}).$$

To prove this we split cases on the processor mode in E'.

1. **User mode**: $mode(E'.u_p.core)$

   From equation (4.6) we easily conclude

   $$\forall a \in S_{r\mathrm{MIPS}}.\mathcal{R} \cup (E'.\mathcal{S} \setminus A_{guest}).\ E'.m(a) = D'.m(a)$$

   and we only need to prove $\forall a \in E'.\mathcal{O}_p \setminus E'.\mathcal{S}.\ E'.m(a) = D'.m(a)$.

   The processor core in configuration E is also in user mode because of the local configuration equality $E \approx_p E'$. Therefore, the simulation relation $csim_p(D.M, E)$ and the invariants $P_{S_{r\mathrm{MIPS}}}(E)$, $oinv(E)$ give us

   $$\forall a \in E.\mathcal{O}_p \setminus E.\mathcal{S}.\ E.m(a) = D.m(a).$$

   By $E \approx_p E'$ and $D \approx_p D'$ we know $E.m|_{E.\mathcal{O}_p} = E'.m|_{E.\mathcal{O}_p}$ and $D.m|_{D.\mathcal{O}_p} = D'.m|_{D.\mathcal{O}_p}$. Moreover, the ownership states are coupled as $D.\mathcal{O}_p = E.\mathcal{O}_p$ in $sinv(D, E)$. Therefore, one concludes

   $$\forall a \in E.\mathcal{O}_p \setminus E.\mathcal{S}.\ E'.m(a) = D'.m(a).$$

   Since we have $E'.\mathcal{O}_p \cap E'.\mathcal{S} = E.\mathcal{O}_p \cap E.\mathcal{S}$ and $E.\mathcal{O}_p = E'.\mathcal{O}_p$ from $E \approx_p E'$, we can trivially show that the owned non-shared addresses do not change

   $$\begin{aligned} E'.\mathcal{O}_p \setminus E'.\mathcal{S} &= E'.\mathcal{O}_p \setminus (E'.\mathcal{O}_p \cap E'.\mathcal{S}) \\ &= E.\mathcal{O}_p \setminus (E.\mathcal{O}_p \cap E.\mathcal{S}) \\ &= E.\mathcal{O}_p \setminus E.\mathcal{S} \end{aligned}$$

and conclude the claim $\forall a \in E'.\mathcal{O}_p \setminus E'.\mathcal{S}.\ E'.m(a) = D'.m(a)$.

Obviously, the invariants to be shown are preserved by definition because the processor is in user mode. Using $P_{S_{r\text{MIPS}}}(E')$ and $oinv(E')$ we also have $\overline{\mathcal{SO}}(E'.\mathcal{G}, p) \subset A_{hyp}$.

2. **System mode**: $\neg mode(E'.u_p.core)$

   In this case we first prove the SB invariants and then consider the simulation relation.

   Using the invariant $inv_{sb}(D.u_p, \overline{\mathcal{SO}}(E.\mathcal{G}, p))$, the ownership coupling in $sinv(D, E)$, as well as the unit invariant $uinv_p(D)$ we conclude

   $$\forall a \in S_{\text{MIPS}}.\mathcal{R} \cup D.\mathcal{S} \cup \overline{\mathcal{SO}}(D.\mathcal{G}, p)).\ \neg sbhit(D.u_p.sb, a[31:2]) \tag{4.7}$$

   Therefore, since the ownership invariant $oinv(D.\mathcal{G})$ holds, we can obviously state

   $$\forall a \in \mathbb{B}^{32}.\ sbhit(D.u_p.sb, a[31:2]) \implies a \in D.\mathcal{O}_p \setminus D.\mathcal{S}.$$

   Analogously to the previous case we apply $D \approx_p D'$ to get $D.\mathcal{O}_p \setminus D.\mathcal{S} = D'.\mathcal{O}_p \setminus D'.\mathcal{S}$. Moreover, the store buffer configuration is unchanged, namely, $D.u_p.sb = D'.u_p.sb$. Hence, the following is also valid in the configuration D':

   $$\forall a \in \mathbb{B}^{32}.\ sbhit(D'.u_p.sb, a[31:2]) \implies a \in D'.\mathcal{O}_p \setminus D'.\mathcal{S}.$$

   Applying $oinv(D')$ we easily conclude

   $$\forall a \in S_{\text{MIPS}}.\mathcal{R} \cup D'.\mathcal{S} \cup \overline{\mathcal{SO}}(D'.\mathcal{G}, p)).\ \neg sbhit(D'.u_p.sb, a[31:2]),$$

   what, in turn, by reason of $P_{S_{r\text{MIPS}}}(E')$ and $sinv(E', D')$ shows $inv_{sb}(D'.u_p, \overline{\mathcal{SO}}(E'.\mathcal{G}, p))$ and $extinv_{sb}(D'.u_p, D'.\mathcal{S})$. Recall that we have $uinv_p(D') = extinv_{sb}(D'.u_p, D'.\mathcal{S})$ by the instantiation.

   Since the sets of addresses are equal in both machines because of $sinv(D', E')$, the rest of the simulation relation to be proven boils down to the claims:

   $$\forall a \in S_{r\text{MIPS}}.\mathcal{R} \cup (E'.\mathcal{S} \setminus A_{guest})\ .\ E'.m(a) = D'.m(a) \tag{4.8}$$
   $$\forall a \in E'.\mathcal{O}_p \setminus E'.\mathcal{S}.\ E'.m(a) = ms(D'.u_p.sb, D'.m)(a) \tag{4.9}$$

   The equation (4.8) follows directly from $sinv(D', E')$.

   From (4.7), $sinv(D, E)$, and $csim_p(D.M, E)$ we analogously get

   $$\forall a \in E.\mathcal{O}_p \setminus E.\mathcal{S}.\ E.m(a) = ms(D.u_p.sb, D.m)(a).$$

   Since we have $sinv(D, E)$ and the equalities $E.\mathcal{O}_p \setminus E.\mathcal{S} = E'.\mathcal{O}_p \setminus E'.\mathcal{S}$, $E.m|_{E.\mathcal{O}_p} = E'.m|_{E.\mathcal{O}_p}$, $D.m|_{D.\mathcal{O}_p} = D'.m|_{D.\mathcal{O}_p}$, $D'.u_p = D.u_p$, we easily conclude (4.9).

□

This lemma finished the proof of the simple store buffer reduction applied in the thesis. From now we can directly rely on Theorem 2.4 between *Cosmos* machines instantiated with the reduced and reference multi-core MIPS-86 models and will operate with the reduced model when we consider the upper layers of the verification stack.

5 Concurrent Mixed Machine Semantics for MIPS-86

In this chapter we consider a concurrent operational semantics for the combination of stack-based programming languages C-IL and Macro assembly on the multi-core MIPS-86 model from Chapter 3. Though the work is mostly based on the results from three doctoral thesise [Sha12, Sch13, Bau14b], we revise the semantics in detail, simplify and adapt it so that it can be easily used for the whole model stack covered in this thesis. The semantics covered here is suitable for implementation and verification of applications and systems where the context switch, stack substitution and operations on TLBs are not required. Later after stating the corresponding compiler correctness and concurrent simulation we will extend it to the one permitting such operations.

The C intermediate language (C-IL) was introduced by Sabine Schmaltz [Sch13] as an attempt to create a simpler language independent from the underlying architecture and compiler and suitable for the implementation of any standard-conforming C. In comparison to the C-dialect C0 [Lei08, PBLS15] the semantics of C-IL supports pointer arithmetic and does not have an explicit notion of a heap, what, in turn, gives the full control over the memory and allows to implement memory management on the low system level.

Since the pure C-IL semantics does not support operations for the implementation of the context switch usually performed by operating system kernels and hypervisors, Andrey Shadrin in his work [Sha12] formulated a macro assembly (MASM) semantics and integrated it with the C-IL on the base of the compiler calling convention. In this mixed programming model the implementation and verification of the context switch in a simple hypervisor on a single-core VAMP architecture became feasible, however, only in case of empty stacks.

Later, both C-IL and MASM semantics separately were slightly adapted for a simplified version of multi-core MIPS-86 model and considered in the concurrent setting by Christoph Baumann in his doctoral thesis [Bau14b].

5.1 Sequential Macro Assembly Semantics

5.1.1 Instructions and Programs

First, by $\mathbb{I}_{\mathrm{ASM}}$ we denote all *MIPS-86 assembly instructions* listed in Tables 3.1– 3.3 and corresponding to their syntax. The indices of registers and immediate constants can be represented in decimal notation. Additionally, we allow to provide the immediate constant imm and the instruction index $iindex$ in binary representation with the help of the prefix `0b`, e.g., `addi 5 7 0bimm`.

In order to convert any MIPS-86 assembly instruction from the set $\mathbb{I}_{\mathrm{ASM}}$ into its 32-bit representations, we introduce the function

$$code_{\mathrm{ASM}} : \mathbb{I}_{\mathrm{ASM}} \to \mathbb{B}^{32}$$

Definition 5.1 (MIPS-86 Assembly Instructions for MASM). Now, we can define the set of MIPS-86 assembly instructions supported by the MASM semantics:

$$\mathbb{I}_{\mathrm{ASM}}^{\mathrm{MASM}} \overset{def}{\equiv} \{instr \mid instr \in \mathbb{I}_{\mathrm{ASM}} \wedge plain\,(code_{\mathrm{ASM}}(instr))\}$$

where the predicate

$$plain(I) \overset{def}{\equiv} alu(I) \vee su(I) \vee mem(I) \vee movg2s(I) \vee movs2g(I)$$

denotes that the MIPS-86 instruction $I \in \mathbb{B}^{32}$ is not an instruction changing the control flow or operating directly on the store buffer or the TLB.

Definition 5.2 (MASM Instructions). Let $\mathbb{P}_{name}$ be a set of *procedure names* admissible in macro assembly. Then we define the set $\mathbb{S}_{\mathrm{MASM}}$ of MASM instructions in the following way:

- stack operations: $r \in \mathbb{N}_0 \implies (\textbf{push}\ r) \in \mathbb{S}_{\mathrm{MASM}} \wedge (\textbf{pop}\ r) \in \mathbb{S}_{\mathrm{MASM}}$
- parameter load/store: $i \in \mathbb{N} \wedge r \in \mathbb{N}_0 \implies (\textbf{lparam}\ r\ i) \in \mathbb{S}_{\mathrm{MASM}} \wedge (\textbf{sparam}\ r\ i) \in \mathbb{S}_{\mathrm{MASM}}$
- goto: $l \in \mathbb{N} \implies (\textbf{goto}\ l) \in \mathbb{S}_{\mathrm{MASM}}$
- conditional goto: $r \in \mathbb{N}_0 \wedge l \in \mathbb{N} \implies (\textbf{ifnez}\ r\ \textbf{goto}\ l) \in \mathbb{S}_{\mathrm{MASM}}$
- procedure call: $pn \in \mathbb{P}_{name} \implies (\textbf{call}\ pn) \in \mathbb{S}_{\mathrm{MASM}}$
- return from a procedure: $\textbf{ret} \in \mathbb{S}_{\mathrm{MASM}}$
- assembly instructions: $\mathbb{I}_{\mathrm{ASM}}^{\mathrm{MASM}} \subset \mathbb{S}_{\mathrm{MASM}}$
- inline MIPS-86 assembly[1]: $il \in \mathbb{I}_{\mathrm{ASM}}^{+} \implies \textbf{asm}\{il\} \in \mathbb{S}_{\mathrm{MASM}}$

Definition 5.3 (MASM Program). A MASM program is represented by a *procedure table* defined by the type $Prog_{\mathrm{MASM}}$ as a mapping from names of all procedures declared in the MASM program to the relevant procedure information:

$$Prog_{\mathrm{MASM}} \overset{def}{\equiv} \mathbb{P}_{name} \rightharpoonup ProcT$$

where

$$ProcT \overset{def}{\equiv} (npar \in \mathbb{N}_0, body \in \mathbb{S}_{\mathrm{MASM}}^{*} \cup \{\mathbf{extern}\}, uses \in \mathbb{N}^{*})$$

is a set of *procedure table entries* with the following components:

- *npar* – the number of input parameters for the procedure,
- *body* – either the procedure body given as a list of MASM instructions, or the keyword **extern** indicating that the procedure is declared as an external one.[2],
- *uses* – a list of indices for GPRs whose content is saved on the procedure entry and restored on the return from this procedure.

Definition 5.4 (MASM External Procedure Predicate). Given a MASM program $\pi_\mu \in Prog_{\mathrm{MASM}}$, we test whether a procedure $pn \in \mathbb{P}_{name}$ declared in π_μ (i.e. $pn \in \operatorname{dom}(\pi_\mu)$) is external or not with the help of the predicate

$$ext(pn, \pi_\mu) \overset{def}{\equiv} \pi_\mu(pn).body = \mathbf{extern}$$

[1]The semantics of inline assembly is not covered in this chapter and will be considered later in the thesis. However, in order to use the same definition of MASM programs later on when we deal with this extension in detail, we include the instruction **asm**$\{il\}$ into $\mathbb{S}_{\mathrm{MASM}}$ already here.

[2]Analogously to [Sha12] we do not formalize the execution of external procedures in the MASM semantics. Such procedures will be discussed later when we consider inter-language calls in the mixed machine semantics.

5.1.2 Machine Configuration

As we already mentioned, the semantics of macro assembly, like many other programming languages, does not consider the address translation and interrupt mechanism. Moreover, the store buffer is also supposed to be invisible. Basically, one can consider the state of the MASM machine as a combination of a simplified version of the single core MIPS-86 with a stack abstraction modelled as a sequence of procedure frames.

Definition 5.5 (MASM Stack Frame). We define the *MASM stack frames* as tuples from the set

$$\mathit{frame}_{\mathrm{MASM}} \overset{def}{\equiv} \big(p \in \mathbb{P}_{name}, loc \in \mathbb{N}, pars \in (\mathbb{B}^{32})^*, \mathit{lifo} \in (\mathbb{B}^{32})^*, saved : \mathbb{B}^5 \rightharpoonup \mathbb{B}^{32}\big)$$

comprising the following components:

- p – the name of the procedure the stack frame corresponds to,
- loc – the location counter denoting the index of the next instruction to be executed in the body of the MASM procedure,
- $pars$ – a list of input parameters passed to the procedure,
- lifo – the actual stack component used for storing temporary data and passing input parameters to procedures called from p,
- $saved$ – a buffer for keeping the content of registers listed in the $uses$ component of the procedure declaration.

Note that in the original MASM semantics from [Sha12] the stack could also be not set up and, therefore, not present. In this case it was modelled as a pair of the procedure name and the location counter. The semantics allowed to switch to the abstract stack with frames by writing initially specified stack frame base and stack pointers or to drop the stack abstraction by rewriting these pointers when the stack was completely empty. Since later in this thesis we provide a more powerful mechanism for stack substitution including the trivial cases treated in [Sha12], we consider here a shorter version of MASM where the stack is always present.

Given the abstraction of the stack frames, we now are able to define the state of the MASM machine.

Definition 5.6 (Sequential MASM Configuration). The MASM configuration is a tuple

$$\mathbb{C}_{\mathrm{MASM}} \overset{def}{\equiv} \big(stack \in \mathit{frame}^*_{\mathrm{MASM}},\ gpr : \mathbb{B}^5 \rightharpoonup \mathbb{B}^{32},\ spr : \mathbb{B}^5 \to \mathbb{B}^{32},\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8\big)$$

where $stack$ is the MASM stack with the top-most frame at the end of the sequence, gpr and spr are general and special purpose registers respectively, and $\mathcal{M}$ is the global byte-addressable memory.

In order to easier refer the stack and program components, we use a few overloaded shorthands. Given any MASM configuration $c_\mu \in \mathbb{C}_{\mathrm{MASM}}$, stack $st \equiv c_\mu.stack$, and program $\pi_\mu \in \mathit{Prog}_{\mathrm{MASM}}$, we introduce the following notation for:

- the index of the top-most frame:
$$top(c_\mu) \equiv top(st) \equiv |st|$$
- the components $X \in \{p, loc, pars, saved, \mathit{lifo}\}$ of the stack frame with an index $i \in \mathbb{N}_{top(st)}$:
$$X_i(c_\mu) \equiv X_i(st) \equiv st[i].X$$

index i	shorthand for i and i_5	description
0	$zero$	always contains 0^{32}
1	t_1	temporary values
2	rv	procedure return value
3	t_2	temporary values
$4, \ldots, 7$	$i_1 \ldots i_4$	input arguments for procedure calls
$8 \ldots 15, 24 \ldots 28$	$t_3 \ldots t_{15}$	temporary values
$16 \ldots 23$	$sv_1 \ldots sv_8$	callee-save registers
29	sp	stack pointer
30	bp	stack frame base pointer
31	ra	return address

Table 5.1: GPRs Usage Convention for MIPS-86

- the components X of the top-most frame:

$$X_{top}(c_\mu) \equiv X_{top}(st) \equiv X_{top(st)}(st)$$

- the components $Y \in \{npar, body, uses\}$ of the procedure table entry for the procedure $p_i(st)$:

$$Y_i(c_\mu, \pi_\mu) \equiv Y_i(st, \pi_\mu) \equiv \pi_\mu(p_i(st)).Y$$

- the components Y for the procedure of the top-most frame:

$$Y_{top}(c_\mu, \pi_\mu) \equiv Y_{top}(st, \pi_\mu) \equiv Y_{top(st)}(st, \pi_\mu)$$

- the next MASM instruction to be executed:

$$instr_{next}(c_\mu, \pi_\mu) \overset{def}{\equiv} body_{top}(c_\mu, \pi_\mu)[loc_{top}(c_\mu)]$$

5.1.3 Compiler Calling Convention for MIPS-86

In general, a *calling convention* describes certain rules for the low-level implementation of the interaction between a caller and a callee. These rules are followed by compilers and also especially important when one has to support calls between functions/procedures written in different programming languages.

In contrast to the programming in higher-level languages, which compilers usually allow programmers to abstract from concrete implementations, the macro assembly programming assumes the knowledge about the mechanism of passing the input parameters and returning the result. This is directly required by the MASM semantics. Moreover, one has to know which registers may be accessed and which must be preserved by the programmer. Hence, we discuss these rules already in the MASM semantics before we consider the compiler correctness.

Obviously, one has to agree how the general purpose registers could be used by the compiler and the programmer. In this thesis we rely on the usage of the GPRs (Table 5.1) borrowed from [Bau14b]. This setting is based on the MIPS calling convention applied in [Inc06] and differs from the original version introduced in [Sha12] for the VAMP processor.

In particular, additionally to the stack and base pointers a register containing the return address is introduced. These three registers are relevant for the implementation of the semantics in MIPS-86 assembly by the compiler, and should not be accessed by the MASM programmer. Furthermore, we explicitly distinguish between so called *calle-* and *caller-save registers*.

Caller-save (or *volatile*, *scratch*) *registers* are registers that can be used by the callee without any restrictions. The caller must save their content if it needs this content after return from the callee.

Callee-save (or *non-volatile*) *registers* are registers whose content must be saved by the callee before it overwrites them. The caller expects the calle-safe registers to be restored after return from the callee.

Finally, combining the specifications from [Sha12, SS12, Bau14b], we state the rules of the compiler calling convention for the MIPS-86:

1. The first four input parameters are passed through the registers $i_1 \dots i_4$ before the call instruction is executed. The caller does not expect the same register values after the callee returns.
2. The remaining parameters (if existent) are passed on the stack in right-to-left order before the call instruction is executed. There is also a space (so called *home addresses*[3]) reserved on the stack for parameters passed in registers.
3. The return value from a procedure call is passed through the register rv.
4. All callee-save registers must be restored before return.
5. A callee is responsible for cleaning up the stack from the parameters.

Note that as follows from Table 5.1 and the discussion above, we refer the registers $t_1 \dots t_{15}$, $i_1 \dots i_4$, and rv as caller-save registers available for the MASM programmer. The sets of indices for calle- and caller-save registers $Reg_{callee}, Reg_{caller} \subset \mathbb{B}^5$ respectively can then be formally defined as

$$Reg_{callee} \overset{def}{\equiv} \{sv_i \mid sv_i \in \mathbb{B}^5 \wedge i \in \mathbb{N}_8\}$$

$$Reg_{caller} \overset{def}{\equiv} \mathbb{B}^5 \setminus \{zero, sp, bp, ra\} \setminus Reg_{callee}$$

Definition 5.7 (Number of Parameters Passed on the MASM Stack and through Registers). In order to compute the number of parameters to be passed through registers for a call of a procedure $p \in \mathbb{P}_{name}$ declared in a MASM program $\pi_\mu \in Prog_{\text{MASM}}$, we define the auxiliary function

$$npar^{\pi_\mu}_{regs}(p) \overset{def}{\equiv} \begin{cases} 4 & : \pi_\mu(p).npar > 4 \\ \pi_\mu(p).npar & : \text{otherwise} \end{cases}$$

Hence, the number of parameters to be passed on the stack is computed as

$$npar^{\pi_\mu}_{stack}(p) \overset{def}{\equiv} \pi_\mu(p).npar - npar^{\pi_\mu}_{regs}(p)$$

5.1.4 Transition Function

First, we introduce functions preforming specific updates on the MASM configuration.

Definition 5.8 (MASM Configuration Update Functions). For a given MASM configuration $c_\mu \in \mathbb{C}_{\text{MASM}}$ with the stack $c_\mu.stack \neq \varepsilon$, we define the following operations on the top-most frame $t \equiv top(c_\mu)$ of the stack:

[3]According to Microsoft [Mic15], "Home addresses are required for the register arguments so a contiguous area is available in case the called function needs to take the address of the argument list ... or an individual argument. This area also provides a convenient place to save register arguments ... and may be used by the called function for other purposes besides saving parameter register values."

- incrementing the location counter[4]:

$$inc_{loc}(c_\mu) \overset{def}{\equiv} c_\mu \left[stack := c_\mu.stack[t \mapsto frame']\right]$$
$$\text{with } frame' \equiv c_\mu.stack[t]\,[loc := loc_{top}(c_\mu) + 1]$$

- setting the location counter to $l \in \mathbb{N}$:

$$set_{loc}(c_\mu, l) \overset{def}{\equiv} c_\mu \left[stack := c_\mu.stack[t \mapsto frame']\right]$$
$$\text{with } frame' \equiv c_\mu.stack[t]\,[loc := l]$$

- removing the top-most frame:

$$drop_{frame}(c_\mu) \overset{def}{\equiv} c_\mu\,[stack := c_\mu.stack[1 : t-1]]$$

- pushing a sequence $el \in (\mathbb{B}^{32})^*$ of elements on the lifo:

$$push_{lifo}(c_\mu, el) \overset{def}{\equiv} c_\mu \left[stack := c_\mu.stack[t \mapsto frame']\right]$$
$$\text{with } frame' \equiv c_\mu.stack[t] \left[lifo := lifo_{top}(c_\mu) \circ el\right]$$

- removing $n \in \mathbb{N}$ elements from the lifo if $|lifo_{top}(c_\mu)| \geq n$:

$$pop_{lifo}(c_\mu, n) \overset{def}{\equiv} c_\mu \left[stack := c_\mu.stack[t \mapsto frame']\right]$$
$$\text{with } frame' \equiv c_\mu.stack[t] \left[lifo := lifo_{top}(c_\mu)[1 : |lifo_{top}(c_\mu)| - n]\right]$$

- updating a parameter $i \in \mathbb{N}$ on the stack by a value $v \in \mathbb{B}^{32}$:

$$set_{pars}(c_\mu, i, v) \overset{def}{\equiv} c_\mu \left[stack := c_\mu.stack[t \mapsto frame']\right]$$
$$\text{with } frame' \equiv c_\mu.stack[t]\,[pars := pars'] \text{ and } pars' \equiv pars_{top}(c_\mu)\,[i \mapsto v]$$

Obviously, in the definition of the MASM semantics we will be interested only in valid, or, in other words, well-formed states of the MASM machine.

Definition 5.9 (Well-Formed MASM Stack). We call a MASM stack $st \in frame^*_{\text{MASM}}$ *well-formed* wrt. a MASM program $\pi_\mu \in Prog_{\text{MASM}}$ if and only if (i) the stack is not empty and (ii) every stack frame (a) belongs to a procedure defined in the program π_μ, (b) has the location counter pointing to an instruction in the body of this procedure, and exactly the declared (c) number of parameters and (d) the used registers are stored in the corresponding components of the frame.

$$\begin{aligned}
wfstack^{\pi_\mu}_{\text{MASM}}(st) \overset{def}{\equiv}\ & (i) && st \neq \varepsilon \\
& (ii) && \forall\, i \in \mathbb{N}_{top(st)}. \\
& && \quad (a) \quad p_i(st) \in \operatorname{dom}(\pi_\mu) \wedge \neg ext(p_i(st), \pi_\mu) \\
& && \quad (b) \quad loc_i(st) \in [1 : |body_i(st, \pi_\mu)|] \\
& && \quad (c) \quad |pars_i(st)| = npar_i(st, \pi_\mu) \\
& && \quad (d) \quad r \in \operatorname{dom}(saved_i(st)) \iff \langle r \rangle \in uses_i(st, \pi_\mu)
\end{aligned}$$

[4]One could have written the definition as: $inc_{loc}(c_\mu) \overset{def}{\equiv} c'_\mu$ such that $loc_{top}(c'_\mu) = loc_{top}(c_\mu) + 1$ and all other components of c'_μ are equal to the ones from c_μ. Instead, we present it in detail in the formal way as it is done in [Sha12], what better matches the computer-aided verification, e.g., in case of translating the definition into a tool specific language.

Moreover, as we have seen in the definition of the sequential MASM configurations, the general purpose registers are represented by a partial mapping. Now, after we have considered the GPRs usage convention, we make the following restriction on this MASM configuration component.

Definition 5.10 (Well-Formed MASM GPRs). We say that the mapping $gpr : \mathbb{B}^5 \rightharpoonup \mathbb{B}^{32}$ representing the general purpose registers in the MASM configuration is well-formed iff the GPRs contain all registers except *sp*, *bp*, *ra* which content is not modelled in the semantics.

$$\mathit{wfreg}_{\mathrm{MASM}}(gpr) \overset{def}{\equiv} \mathrm{dom}\,(gpr) = \mathbb{B}^5 \setminus \{sp, bp, ra\}$$

Definition 5.11 (Well-Formed MASM Configuration). Hence, a sequential MASM configuration $c_\mu \in \mathbb{C}_{\mathrm{MASM}}$ is considered to be *well-formed* wrt. a MASM program $\pi_\mu \in \mathit{Prog}_{\mathrm{MASM}}$ if and only if its stack is well-formed and the GPR component represents all registers except *sp*, *bp*, *ra*.

$$\mathit{wfconf}^{\pi_\mu}_{\mathrm{MASM}}(c_\mu) \overset{def}{\equiv} \begin{array}{ll} \textit{(i)} & \mathit{wfstack}^{\pi_\mu}_{\mathrm{MASM}}(c_\mu.stack) \\ \textit{(ii)} & \mathit{wfreg}_{\mathrm{MASM}}(c_\mu.gpr) \end{array}$$

Then, the operational semantics of the sequential MASM machine can be formalized in a way similar to [Sha12].

Definition 5.12 (Sequential MASM Transition Function). For a given MASM program $\pi_\mu \in \mathit{Prog}_{\mathrm{MASM}}$, we define the transitions of the sequential MASM machine by the function

$$\delta^{\pi_\mu}_{\mathrm{MASM}} : \mathbb{C}_{\mathrm{MASM}} \to \mathbb{C}_{\mathrm{MASM}\perp}$$

where $\mathbb{C}_{\mathrm{MASM}\perp} \overset{def}{\equiv} \mathbb{C}_{\mathrm{MASM}} \cup \{\perp\}$ and $\perp$ denotes here an *error state*.[5]

So, for a configuration $c_\mu \in \mathbb{C}_{\mathrm{MASM}}$, the function $\delta^{\pi_\mu}_{\mathrm{MASM}}(c_\mu)$ is defined by a case split on the instruction to be executed.

Macros Semantics First, we consider the macros execution:

- Push operation on stack: $\mathit{instr}_{next}(c_\mu, \pi_\mu) = \textbf{push}\ r$

$$\delta^{\pi_\mu}_{\mathrm{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} \mathit{inc}_{loc}\left(\mathit{push}_{\mathit{lifo}}(c_\mu, c_\mu.gpr(r_5))\right) & : r \notin \{sp, bp, ra\} \\ \perp & : \text{otherwise} \end{cases}$$

- Pop operation on stack: $\mathit{instr}_{next}(c_\mu, \pi_\mu) = \textbf{pop}\ r$

$$\delta^{\pi_\mu}_{\mathrm{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} \mathit{inc}_{loc}\left(\mathit{pop}_{\mathit{lifo}}(c'_\mu, 1)\right) & : \mathit{lifo}_{top}(c_\mu) \neq \varepsilon \wedge r \notin \{sp, bp, ra, zero\} \\ \perp & : \text{otherwise} \end{cases}$$

where $c'_\mu \equiv c_\mu\left[gpr := c_\mu.gpr\left[r_5 \mapsto \mathit{lifo}_{top}(c_\mu)[k]\right]\right]$ with $k \equiv |\mathit{lifo}_{top}(c_\mu)|$.

[5]The error state introduced in [Lei08] is used to indicate a run-time error appearing during the program execution (including errors that could be detected by the static check of the program). Instead of this additional state, one could define the transition function as a partial mapping. However, such a solution would not match the simulation theorem for *Cosmos* machines where one has to guarantee that there are no run-time errors for existing abstract computations.

- Parameter load: $instr_{next}(c_\mu, \pi_\mu) = \textbf{lparam}\ r\ i$

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} inc_{loc}\left(c'_\mu\right) & : \ i \le |pars_{top}(c_\mu)| \wedge r \notin \{sp, bp, ra, zero\} \\ \bot & : \ \text{otherwise} \end{cases}$$

where $c'_\mu \equiv c_\mu\left[gpr := c_\mu.gpr\left[r_5 \mapsto pars_{top}(c_\mu)[i]\right]\right]$

- Parameter store: $instr_{next}(c_\mu, \pi_\mu) = \textbf{sparam}\ r\ i$

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} inc_{loc}\left(set_{pars}\left(c_\mu, i, c_\mu.gpr(r_5)\right)\right) & : \ i \le |pars_{top}(c_\mu)| \wedge r \notin \{sp, bp, ra\} \\ \bot & : \ \text{otherwise} \end{cases}$$

- Goto: $instr_{next}(c_\mu, \pi_\mu) = \textbf{goto}\ l$

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} set_{loc}\left(c_\mu, l\right) & : \ l \le |body_{top}(c_\mu, \pi_\mu)| \\ \bot & : \ \text{otherwise} \end{cases}$$

- Conditional goto: $instr_{next}(c_\mu, \pi_\mu) = \textbf{ifnez}\ r\ \textbf{goto}\ l$

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} set_{loc}\left(c_\mu, l\right) & : \ l \le |body_{top}(c_\mu, \pi_\mu)| \wedge r \notin \{sp, bp, ra\} \wedge c_\mu.gpr(r_5) \ne 0^{32} \\ inc_{loc}\left(c_\mu\right) & : \ l \le |body_{top}(c_\mu, \pi_\mu)| \wedge r \notin \{sp, bp, ra\} \wedge c_\mu.gpr(r_5) = 0^{32} \\ \bot & : \ \text{otherwise} \end{cases}$$

- Procedure call: $instr_{next}(c_\mu, \pi_\mu) = \textbf{call}\ pn$

  The function $\delta^{\pi_\mu}_{\text{MASM}}(c_\mu)$ does not produce the error state if and only if the following transition conditions hold:

  – the procedure pn is declared as non-external in the program

$$pn \in \text{dom}\left(\pi_\mu\right) \wedge \neg ext(pn, \pi_\mu)$$

  – the lifo of the top-most frame contains at least as many elements as the number of procedure parameters to be passed on the stack:

$$|lifo_{top}(c_\mu)| \ge npar^{\pi_\mu}_{stack}(pn)$$

  Then the transition effect is computed as

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} c_\mu\left[stack := inc_{loc}\left(pop_{lifo}\left(c_\mu, npar^{\pi_\mu}_{stack}(pn)\right)\right).stack \circ frame'\right]$$

  where the components of the newly created stack frame are initialized as follows:

$$frame'.p = pn \qquad frame'.loc = 1 \qquad frame'.lifo = \varepsilon$$

$$frame'.saved(r) = \begin{cases} c_\mu.gpr(r) & : \ \langle r \rangle \in \pi_\mu(pn).uses \\ undefined & : \ \text{otherwise} \end{cases}$$

$$frame'.pars = pars' \circ pars''$$

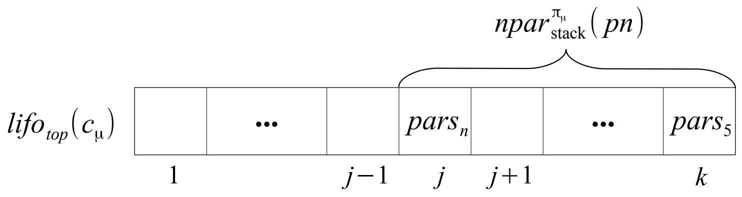

Figure 5.1: Passing parameters on the lifo during the MASM procedure call. Here, the number of input parameters $pars$ for pn is $n \equiv \pi_\mu(pn).npar$.

with the sequences[6] $pars' \equiv (0^{32}, \ldots, 0^{32})$ of length $|pars'| = npar^{\pi_\mu}_{regs}(pn)$ and $pars'' \equiv rev\left(lifo_{top}(c_\mu)\,[j : k]\right)$ for the indices $k \equiv |lifo_{top}(c_\mu)|,\ j \equiv k + 1 - npar^{\pi_\mu}_{stack}(pn)$ depicted on Figure 5.1.

If the transition conditions are not met, the result is $\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \bot$.

- Return from a procedure: $instr_{next}(c_\mu, \pi_\mu) = \textbf{ret}$

$$\delta^{\pi_\mu}_{\text{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} drop_{frame}\,(c_\mu\,[gpr := gpr']) & : \ top(c_\mu) > 1 \\ \bot & : \ \text{otherwise} \end{cases}$$

such that

$$gpr'(r) = \begin{cases} saved_{top}(c_\mu)(r) & : \ \langle r\rangle \in \pi_\mu(p_{top}(c_\mu)).uses \\ c_\mu.gpr(r) & : \ \text{otherwise} \end{cases}$$

Assembly Instructions In case of an assembly instruction $instr_{next}(c_\mu, \pi_\mu) \in \mathbb{I}^{\text{MASM}}_{\text{ASM}}$, we define $\delta^{\pi_\mu}_{\text{MASM}}(c_\mu)$ on the base of the transition function δ_{instr} for the non-interrupted instruction execution from Chapter 3 and the memory update semantics for the SB reduced MIPS-86 from Chapter 4. For this purpose we introduce or reload the following shorthands:

- the MIPS-86 instruction to be executed:

$$I_\mu \equiv I(c_\mu, \pi_\mu) \overset{def}{\equiv} code_{\text{ASM}}(instr_{next}(c_\mu, \pi_\mu))$$

- the processor core configuration $core_{\text{MIPS}}(c_\mu) \in \mathbb{C}_{core}$ constructed from the MASM configuration c_μ such that $core_{\text{MIPS}}(c_\mu) \overset{def}{\equiv} c$ with register files $c.spr = c_\mu.spr$, $c.gpr$ such that

$$c.gpr(r) = \begin{cases} c_\mu.gpr(r) & : \ r \notin \{sp, bp, ra\} \\ \epsilon\,\mathbb{B}^{32} & : \ \text{otherwise} \end{cases}$$

and any value of the program counter $c.pc = \epsilon\,\mathbb{B}^{32}$

- the data word read from the memory:

$$R_\mu \equiv R(c_\mu, \pi_\mu) \equiv c_\mu.\mathcal{M}_4(ea(c, I_\mu))$$

[6]In the original work [Sha12] $pars'$ is initialized with arbitrary values introducing the non-determinism in the MASM semantics. As we have mentioned in Section 1.2, in this work we model systems as deterministic automata and, therefore, in this case should have provided to the MASM transition function an extra input containing such initial values. However, in order to make the semantics simpler and avoid this extra alphabet, we initialize $pars'$ with zeros.

- the next processor core configuration: $c' \equiv \delta_{inst}(c, I_\mu, R_\mu)$

Similarly to the macros, we are not allowed to access the registers whose values are not modeled in the semantics. This registers are detected by the fields of the instruction to be executed. However, rs, rt, and rd are defined only for those instructions in which the corresponding fields are present. We define in detail a predicate indicating that the assembly instruction $instr_{next}(c_\mu, \pi_\mu) \in \mathbb{I}^{\mathrm{MASM}}_{\mathrm{ASM}}$ to be executed in the MASM configuration c_μ does not access the registers sp, bp, ra and does not write to the register $zero$:

$$\begin{aligned} accregs^{\mathrm{MASM}}_{\mathrm{ASM}}(c_\mu, \pi_\mu) \overset{def}{\equiv} \quad & \text{(i)} \quad rt(I_\mu) \notin \{sp, bp, ra\} \\ & \text{(ii)} \quad \neg(lui(I_\mu) \vee sll(I_\mu) \vee srl(I_\mu) \vee sra(I_\mu)) \implies \\ & \qquad rs(I_\mu) \notin \{sp, bp, ra\} \\ & \text{(iii)} \quad gprw(I_\mu) \implies cad(I_\mu) \notin \{sp, bp, ra, zero\} \end{aligned}$$

Now the result of the transition can be easily computed as

$$\delta^{\pi_\mu}_{\mathrm{MASM}}(c_\mu) \overset{def}{\equiv} \begin{cases} inc_{loc}\left(c'_\mu\right) & : accregs^{\mathrm{MASM}}_{\mathrm{ASM}}(c_\mu, \pi_\mu) \\ \bot & : \text{otherwise} \end{cases}$$

where

$$c'_\mu.gpr = c'.gpr \quad c'_\mu.spr = c'.spr \quad c'_\mu.stack = c_\mu.stack$$

$$c'_\mu.\mathcal{M} = \begin{cases} \delta_m(c_\mu.\mathcal{M}, (ea(c, I_\mu), sv(c, I_\mu))) & : locksw(I_\mu) \vee sw(I_\mu) \\ \delta_m(c_\mu.\mathcal{M}, (c.gpr(rd(I_\mu)), ea(c, I_\mu), sv(c, I_\mu))) & : cas(I_\mu) \\ c_\mu.\mathcal{M} & : \text{otherwise} \end{cases}$$

Inline Assembly As we mentioned before, we will consider the extended semantics with inline assembly as a separate topic later in this work. Therefore, for the time being, in case of $instr_{next}(c_\mu, \pi_\mu) = \mathbf{asm}\{il\}$ we simply generate the run-time error $\delta^{\pi_\mu}_{\mathrm{MASM}}(c_\mu) \overset{def}{\equiv} \bot$.

5.2 Sequential Intermediate C Semantics

5.2.1 Types and Qualifiers

Definition 5.13 (C-IL Types). Let $\mathbb{T}_C$ denote the set of *composite (struct) types* and $\mathbb{T}_P$ be the set of *primitive types* containing signed and unsigned 32-bit integers and the type **void**:

$$\mathbb{T}_P \overset{def}{\equiv} \{\mathbf{void}, \mathbf{i32}, \mathbf{u32}\}$$

Then the *set of C-IL types* $\mathbb{T}$ is constructed as follows:

- primitive types: $t \in \mathbb{T}_P \implies t \in \mathbb{T}$
- struct types: $t_c \in \mathbb{T}_C \implies (\mathbf{struct}\ t_c) \in \mathbb{T}$
- regular pointer types: $t \in \mathbb{T} \implies \mathbf{ptr}(t) \in \mathbb{T}$
- array types: $t \in \mathbb{T} \wedge n \in \mathbb{N} \implies \mathbf{array}(t, n) \in \mathbb{T}$

  An array type $\mathbf{array}(t, n)$ is characterized by the type t of elements and the number n of elements in the array.

- function pointer types: $t \in \mathbb{T} \wedge T \in (\mathbb{T} \setminus \{\mathbf{void}\})^* \implies \mathbf{funptr}(t, T) \in \mathbb{T}$

  A function pointer type $\mathbf{funptr}(t, T)$ is described by the type t of the return value and the list T of parameter types.

Note that we do not explicitly provide a type for boolean values, which could be modeled by the integer type in our case.

Besides the types, the C intermediate language supports *type qualifiers* **const** and **volatile** corresponding to the ones from regular C. The qualifiers annotate the type declaration and hint the compiler how to treat variables of such types.

Naturally, **const** is used to declare constant variables that cannot be re-written. In turn, **volatile** indicates that the content of the corresponding memory region might be modified by the environment, e.g., other threads, devices, or interrupt handlers. Therefore, in this case the compiler performs less optimization and does not reorder memory accesses to such variables.

Definition 5.14 (Type Qualifiers). We denote the set of type qualifiers as

$$\mathbb{Q} \overset{def}{\equiv} \{\mathbf{const}, \mathbf{volatile}\}$$

Any C-IL type can be annotated with a subset of qualifiers.

Definition 5.15 (Qualified C-IL Types). We define the *set of qualified types* $\mathbb{T}_{\mathbb{Q}}$ as a set containing the following elements:

- the type **void** cannot be qualified: $(\emptyset, \mathbf{void}) \in \mathbb{T}_{\mathbb{Q}}$
- qualified primitive types: $q \subseteq \mathbb{Q} \wedge t \in \{\mathbf{i32}, \mathbf{u32}\} \implies (q, t) \in \mathbb{T}_{\mathbb{Q}}$
- qualified struct types: $q \subseteq \mathbb{Q} \wedge t_c \in \mathbb{T}_C \implies (q, \mathbf{struct}\ t_c) \in \mathbb{T}_{\mathbb{Q}}$
- qualified regular pointer types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}} \implies (q, \mathbf{ptr}(t)) \in \mathbb{T}_{\mathbb{Q}}$
- qualified array types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}} \wedge n \in \mathbb{N} \implies (q, \mathbf{array}(t, n)) \in \mathbb{T}_{\mathbb{Q}}$
- qualified function pointer types:

$$q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_{\mathbb{Q}} \wedge T \in (\mathbb{T}_{\mathbb{Q}} \setminus \{(\emptyset, \mathbf{void})\})^* \implies (q, \mathbf{funptr}(t, T)) \in \mathbb{T}_{\mathbb{Q}}$$

As we mentioned before, the type qualifiers are only used by the compiler. In fact, they do not affect the C-IL semantics and can be dropped in formal definitions for brevity whenever possible. For this purpose we introduce a function converting qualified types to the corresponding unqualified ones.

Definition 5.16 (Conversion from Qualified Type to Unqualified Type). We define the function $qt2t : \mathbb{T}_{\mathbb{Q}} \to \mathbb{T}$ converting any qualified C-IL type to an unqualified type as follows:

$$qt2t(x) \overset{def}{\equiv} \begin{cases} t & : \quad x = (q, t) \wedge t \in \mathbb{T}_P \\ \mathbf{ptr}(qt2t(x')) & : \quad x = (q, \mathbf{ptr}(x')) \\ \mathbf{array}(qt2t(x'), n) & : \quad x = (q, \mathbf{array}(x', n)) \\ \mathbf{funptr}(qt2t(x'), map(qt2t, X)) & : \quad x = (q, \mathbf{funptr}(x', X)) \\ \mathbf{struct}\ t_c & : \quad x = (q, \mathbf{struct}\ t_c) \end{cases}$$

Definition 5.17 (**Type Predicates**)**.** We define predicates checking whether a given type $t \in \mathbb{T}$ is a pointer type, an array type, or a function pointer type:

$$\begin{aligned}
\mathit{isptr}(t) &\overset{def}{\equiv} \exists t'.\, t = \mathbf{ptr}(t') \\
\mathit{isarray}(t) &\overset{def}{\equiv} \exists t', n.\, t = \mathbf{array}(t', n) \\
\mathit{isfunptr}(t) &\overset{def}{\equiv} \exists t', T.\, t = \mathbf{funptr}(t', T)
\end{aligned}$$

5.2.2 Values

In contrast to many programming language semantics, most values used in C-IL are modeled as bit-strings of an appropriate length and a type.

Definition 5.18 (**Primitive Values**)**.** The set $\mathit{val}_{\mathbf{prim}}$ contains all values for variables of primitive type.

$$\mathit{val}_{\mathbf{prim}} \overset{def}{\equiv} \bigcup_{b \in \mathbb{B}^{32}} \{\mathbf{val}(b, \mathbf{i32}), \mathbf{val}(b, \mathbf{u32})\}$$

Note that we do not introduce a value for the type **void** because this type explicitly declares its absence.

Since any array value in C is treated as a pointer to the first element of the array, we also model array values as pointers with an array type. First, we introduce the values of *pointers to the global memory* represented as a couple of an address (which is 32-bit string according to the MIPS-86 ISA) and a corresponding type.

Definition 5.19 (**Regular Pointer and Array Values**)**.** We define the set $\mathit{val}_{\mathbf{ptr}}$ of values for regular pointers and arrays as

$$\mathit{val}_{\mathbf{ptr}} \overset{def}{\equiv} \bigcup_{t \in \mathbb{T} \wedge (\mathit{isptr}(t) \vee \mathit{isarray}(t))} \{\mathbf{val}(a, t) \mid a \in \mathbb{B}^{32}\}$$

Among with the regular pointer and array values we distinguish pointers to local variables residing on the stack. Such pointer values are called *local references*.

Definition 5.20 (**Local References**)**.** Let $\mathbb{V}$ be the set of all *variable names*. Then all local references are defined as tuples containing a name of a local variable v, a byte offset o in the byte representation of this variable, an index i of a stack frame corresponding to a call of a function in whose body this variable is used, and a type t of the variable.

$$\mathit{val}_{\mathbf{lref}} \overset{def}{\equiv} \bigcup_{t \in \mathbb{T} \wedge (\mathit{isptr}(t) \vee \mathit{isarray}(t))} \{\mathbf{lref}((v, o), i, t) \mid v \in \mathbb{V} \wedge o, i \in \mathbb{N}_0\}$$

In the C-IL semantics we provide two kinds of function pointer values: actual *function pointer value* and *symbolic function values*. The former one represents a memory address where the compiled code of a function starts. We will later introduce a specific environment parameter that depending on the compiler computes such addresses for given function names. The latter one is used when we do not need to store function pointers in the memory, e.g., for inline or external functions. In this case a symbolic value can only be used to perform a function call.

Definition 5.21 (Function Pointer Values). Let $\mathbb{F}_{name}$ be a set of *function names*. Then the sets $val_{\mathbf{fptr}}$ and $val_{\mathbf{fun}}$ of values for C-IL function pointers are defined as follows:

$$val_{\mathbf{fptr}} \overset{def}{\equiv} \bigcup_{t \in \mathbb{T} \wedge isfunptr(t)} \{\mathbf{val}(a, t) \mid a \in \mathbb{B}^{32}\}$$

$$val_{\mathbf{fun}} \overset{def}{\equiv} \{\mathbf{fun}(f, t) \mid f \in \mathbb{F}_{name} \wedge isfunptr(t)\}$$

Definition 5.22 (C-IL Values). Now, the set of all C-IL values can be defined as a union of all individual values considered above:

$$val \overset{def}{\equiv} val_{\mathbf{prim}} \cup val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup val_{\mathbf{fptr}} \cup val_{\mathbf{fun}}$$

5.2.3 Expressions and Statements

In order to define C-IL programs, we introduce expressions and statements supported in the C-IL semantics.

Definition 5.23 (Unary and Binary Operators). The sets $\mathbb{O}_1$ and $\mathbb{O}_2$ of *unary* and *binary operators* occurring in C-IL expressions are defined as follows:

$$\mathbb{O}_1 \subset \{\oslash \mid \oslash : val \rightharpoonup val\}$$

$$\mathbb{O}_2 \subset \{\otimes \mid \otimes : (val \times val) \rightharpoonup val\}$$

$$\mathbb{O}_1 \overset{def}{\equiv} \{-, \sim, !\}$$

$$\mathbb{O}_2 \overset{def}{\equiv} \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, |, \hat{}, \&\&, ||\}$$

The operators supported in C-IL fully correspond to the ones used C and we omit here their mathematical definitions. For more details, please, refer to [Sch13]. Note that the addition operation allows to perform the pointer arithmetic. When a primitive value is added to a regular pointer, array value, or local reference, this primitive value is multiplied by the size of the type pointed to.

First, additionally to the set $\mathbb{V}$ of variable names and the set $\mathbb{F}_{name}$ of function names present in our semantics, we introduce $\mathbb{F}$ denoting the set of all *field names* used in composite types.

Definition 5.24 (C-IL Expressions). The set $\mathbb{E}$ contains all possible *C-IL expressions* and defined as a set obeying the following rules:

- constants: $c \in val \implies c \in \mathbb{E}$
- variable names: $v \in \mathbb{V} \implies v \in \mathbb{E}$
- function names: $fn \in \mathbb{F}_{name} \implies fn \in \mathbb{E}$
- unary operations: $e \in \mathbb{E} \wedge \oslash \in \mathbb{O}_1 \implies \oslash e \in \mathbb{E}$
- binary operations: $e_1, e_2 \in \mathbb{E} \wedge \otimes \in \mathbb{O}_2 \implies (e_1 \otimes e_2) \in \mathbb{E}$
- ternary operation: $e, e_1, e_2 \in \mathbb{E} \implies (e \mathrel{?} e_1 : e_2) \in \mathbb{E}$
- type cast: $t \in \mathbb{T}_{\mathbb{Q}} \wedge e \in \mathbb{E} \implies (t)e \in \mathbb{E}$
- pointer dereferencing: $e \in \mathbb{E} \implies *(e) \in \mathbb{E}$

- address of: $e \in \mathbb{E} \implies \&(e) \in \mathbb{E}$
- field access: $e \in \mathbb{E} \wedge f \in \mathbb{F} \implies (e).f \in \mathbb{E}$
- size of a type: $t \in \mathbb{T}_{\mathbb{Q}} \implies \textbf{sizeof}(t) \in \mathbb{E}$
- size of an expression: $e \in \mathbb{E} \implies \textbf{sizeof}(e) \in \mathbb{E}$

The array and pointer field accesses typical for C do not belong to the C-IL expressions but can be easily translated into them:

$$a[i] \overset{def}{\equiv} *(a+i) \qquad\qquad b\text{->}f \overset{def}{\equiv} (*(b)).f$$

Definition 5.25 (C-IL Statements). We define the set $\mathbb{S}_{\text{CIL}}$ of C-IL *statements* inductively as follows:

- assignment: $e_0, e_1 \in \mathbb{E} \implies (e_0 = e_1) \in \mathbb{S}_{\text{CIL}}$
- goto: $l \in \mathbb{N} \implies (\textbf{goto}\ l) \in \mathbb{S}_{\text{CIL}}$
- if-not-goto: $l \in \mathbb{N} \wedge e \in \mathbb{E} \implies (\textbf{ifnot}\ e\ \textbf{goto}\ l) \in \mathbb{S}_{\text{CIL}}$
- function call with a return value: $e_0, e \in \mathbb{E} \wedge E \in \mathbb{E}^* \implies (e_0 = \textbf{call}\ e(E)) \in \mathbb{S}_{\text{CIL}}$
- function call without a return value: $e \in \mathbb{E} \wedge E \in \mathbb{E}^* \implies (\textbf{call}\ e(E)) \in \mathbb{S}_{\text{CIL}}$
- return from a function: $e \in \mathbb{E} \implies (\textbf{return}\ e) \in \mathbb{S}_{\text{CIL}}$ and $\textbf{return} \in \mathbb{S}_{\text{CIL}}$

For both kinds of function calls, $E \in \mathbb{E}^*$ represents a list of expressions passed to a function as parameters.

5.2.4 C-IL Programs

All relevant information about functions of a C-IL program is represented by a *function table* consisting of *function table entries*.

Definition 5.26 (C-IL Function Table Entries). The set $FunT$ of function table entries corresponding to each function of a C-IL program is defined as

$$FunT \overset{def}{\equiv} (rettype \in \mathbb{T}_Q, npar \in \mathbb{N}_0, V \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*, P \in \mathbb{S}_{\text{CIL}}^* \cup \{\textbf{extern}\})$$

with the following components:

- $rettype$ – the type of the function's return value which can be also **void** in case of the absence of the return value,
- $npar$ – the number of input parameters of the function,
- V – a list of parameter and local variable declarations consisting of pairs of a variable name and its type. The first $npar$ elements of the list represent the input parameters.
- P – either the function body containing a list of C-IL statements, or the keyword **extern** indicating that the function is declared as an external function.[7]

[7]In contrast to [Sch13] we do not formalize generally the external function execution in the pure C-IL semantics and will treat in the semantics only one particular intrinsic function supported by MISP-86 ISA. Later on in this thesis, however, we will consider external functions in detail for extended models (e.g. mixed semantics) in a way similar to [Sha12] .

Definition 5.27 (**C-IL Program**). Then all C-IL programs are represented by tuples from the set $\mathit{Prog}_{\mathrm{CIL}}$ such that

$$\mathit{Prog}_{\mathrm{CIL}} \overset{def}{\equiv} (V_G \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*, T_F : \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_{\mathbb{Q}})^*, \mathcal{F} : \mathbb{F}_{name} \rightharpoonup \mathit{FunT})$$

has the following elements:

- V_G – a list of global variable declarations consisting of names of global variables and their qualified types,
- T_F – a type table for struct types providing for each such type a list of fields and with their types,
- $\mathcal{F}$ – the function table modelled as a mapping from a function name to a corresponding function table entry.

5.2.5 Machine Configuration

In order to define the semantics for C-IL programs, we proceed with a definition of the C-IL machine state (or configuration). Basically, such a configuration includes the global byte-addressable memory and an abstract stack modelled as a sequence of frames corresponding to function calls.

Definition 5.28 (**C-IL Stack Frames**). *C-IL stack frames* are tuples from the set

$$\mathit{frame}_{\mathrm{CIL}} \overset{def}{\equiv} \big(f \in \mathbb{F}_{name}, \mathit{rds} \in \mathit{val}_{\mathbf{ptr}} \cup \mathit{val}_{\mathbf{lref}} \cup \{\bot\}, \mathit{loc} \in \mathbb{N}, \mathcal{M}_{\mathcal{E}} : \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*\big)$$

and contain the following components:

- f – the name of the function, which the stack frame belongs to,
- rds – the return value destination describing where the return value of a function call has to be stored on the return. The return destination can be a pointer, a local reference, or the value $\bot$ denoting the absence of the return destination.
- loc – the location counter showing the index of the next statement to be executed in the body of the function,
- $\mathcal{M}_{\mathcal{E}}$ – the memory for local variables and parameters in the call of the function f. This memory on the stack (or *local memory*) is modeled as a mapping from a name of a declared variable to the byte-string representation of its value.

Definition 5.29 (**Sequential C-IL Configuration**). A C-IL configuration is a tuple

$$\mathbb{C}_{\mathrm{CIL}} \overset{def}{\equiv} \big(\mathit{stack} \in \mathit{frame}^*_{\mathrm{CIL}}, \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8\big)$$

where stack is the C-IL stack with the top-most frame at the end of the sequence, and $\mathcal{M}$ is the global byte-addressable memory.

Analogously to MASM we introduce a number of overloaded shorthands for $c \in \mathbb{C}_{\mathrm{CIL}}$, $st \equiv c.\mathit{stack}$, and $\pi \in \mathit{Prog}_{\mathrm{CIL}}$:

- the index of the top-most frame:

$$\mathit{top}(c) \equiv \mathit{top}(st) \equiv |st|$$

- the components $X \in \{f, rds, loc, \mathcal{M}_\mathcal{E}\}$ of the stack frame with an index $i \in \mathbb{N}_{top(st)}$:
$$X_i(c) \equiv X_i(st) \equiv st[i].X$$
- the components X of the top-most frame:
$$X_{top}(c) \equiv X_{top}(st) \equiv X_{top(st)}(st)$$
- the components $Y \in \{rettype, npar, V, P\}$ of the function table entry for the function $f_i(st)$:
$$Y_i(c, \pi) \equiv Y_i(st, \pi) \equiv \pi.\mathcal{F}(f_i(st)).Y$$
- the components Y for the function of the top-most frame:
$$Y_{top}(c, \pi) \equiv Y_{top}(st, \pi) \equiv Y_{top(st)}(st, \pi)$$
- the next C-IL statement to be executed:
$$stmt_{next}(c, \pi) \equiv P_{top}(c, \pi)[loc_{top}(c)]$$

5.2.6 Environment Parameters

For program execution in C-IL we need additional information from the compiler and the underlying architecture, e.g., the composite types layout in the memory, base addresses of global variables, etc. In [Sch13] this information is collected in so called *environment parameters* because the C-IL semantics is designed in a general way without a particular connection to a certain C compiler and a processor architecture.

Here, we do not consider all parameters from [Sch13] and concentrate only on those required for our MIPS-86 based version of the C-IL. Note that the endianness of the MIPS-86 architecture is *little endian* and for brevity we do not include it in the environment parameters.

Definition 5.30 (**C-IL Environment Parameters**). We define the C-IL environment parameters $\theta \in Params_{\text{CIL}}$ as a tuple

$$Params_{\text{CIL}} \overset{def}{\equiv} (alloc_{gvar}, \mathcal{F}_{adr}, size_{struc}, offset_{struc}, size\_t, cast, intrinsics)$$

with the following components:

- $alloc_{gvar} : \mathbb{V} \rightharpoonup \mathbb{B}^{32}$ – a mapping from a name of a global variable to the variable's base addresses in the memory,
- $\mathcal{F}_{adr} : \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{32}$ – a mapping from a given C-IL function name to the staring address of the function's compiled code in the memory (undefined for inline and external functions),
- $size_{struc} : \mathbb{T}_C \rightharpoonup \mathbb{N}$ – the size of struct types in bytes,
- $offset_{struc} : \mathbb{T}_C \times \mathbb{F} \rightharpoonup \mathbb{N}_0$ – byte-offsets of fields in struct types,
- $size\_t \in \mathbb{T}_P$ – the type of the value returned by the operator **sizeof**,
- $cast : val \times \mathbb{T} \rightharpoonup val$ – a function describing how the compiler handles the type casting (an example of its in-depth definition can be found in [Sch13]). Note that C-IL is assumed to be type-correct, i.e., all required type casts are made explicitly in the code with the help of the corresponding type cast expression.

- $intrinsics : \mathbb{F}_{name} \rightharpoonup FunT$ – a function table for compiler intrinsic functions. According to [Sch13], such functions provide accesses to architecture specific instructions and usually inlined into the compiled code instead of performing ordinary function calls.

As it was done in [Bau14b], in this work we will consider the only intrinsic function corresponding to the MIPS-86 compare-and-swap cas. Therefore, for $cas \in \mathbb{F}_{name}$ we introduce

$$\theta.intrinsics(cas) \overset{def}{\equiv} fte_{cas}$$

such that the function table entry fte_{cas} is defined as

$$\begin{aligned} fte_{cas}.rettype &= (\emptyset, \mathbf{void}) \\ fte_{cas}.npar &= 4 \\ fte_{cas}.V &= (dest, (\emptyset, \mathbf{ptr}(\emptyset, \mathbf{i32}))) \circ (cmp, (\emptyset, \mathbf{i32})) \circ \\ &\quad\; (exch, (\emptyset, \mathbf{i32})) \circ (ret, (\emptyset, \mathbf{ptr}(\emptyset, \mathbf{i32}))) \\ fte_{cas}.P &= \mathbf{extern} \end{aligned}$$

where $dest$ is a pointer to the memory location at which the memory content shall be swapped to the value $exch$ if this content is equal to the compared value cmp. Additionally, the memory content pointed by $dest$ is stored at the location pointed by ret.

Given these environment parameters $\theta \in Params_{\text{CIL}}$, we can now define a function calculating the size of a given type in bytes.

Definition 5.31 (Size of a C-IL Type). We define the function $size_\theta(t)$ for types $t \in \mathbb{T}$ as

$$size_\theta(t) \overset{def}{\equiv} \begin{cases} 4 & : \; t \in \{\mathbf{i32}, \mathbf{u32}\} \vee isptr(t) \vee isfunptr(t) \\ n \cdot size_\theta(t') & : \; t = \mathbf{array}(t', n) \\ \theta.size_{struc}(t_c) & : \; t = \mathbf{struct}\ t_c \end{cases}$$

Definition 5.32 (Zero Value). Moreover, having the size of a type for a given value $x \in val$, it is easy to check whether x is considered to be zero or not:

$$zero_\theta(x) \overset{def}{\equiv} \begin{cases} a = 0^{8 \cdot size_\theta(t)} & : \; x = \mathbf{val}(a, t) \\ undefined & : \; \text{otherwise} \end{cases}$$

Definition 5.33 (Function Predicate). Additionally, we would like to check whether a given function pointer $v \in val$ corresponds to a given function name $f \in \mathbb{F}_{name}$:

$$\begin{aligned} isfunc(v, f, \theta) \overset{def}{\equiv} &\; \exists t \in \mathbb{T}, a \in \mathbb{B}^{32}. \\ &\quad isfunptr(t) \wedge (v = \mathbf{val}(a, t) \wedge \theta.\mathcal{F}_{adr}(f) = a \vee v = \mathbf{fun}(f, t)) \end{aligned}$$

Definition 5.34 (Combined Function Table). Given a C-IL program $\pi \in Prog_{\text{CIL}}$, we can define the *combined function table* as

$$\mathcal{F}_\pi^\theta \overset{def}{\equiv} \pi.\mathcal{F} \cup \theta.intrinsics$$

Note, that the names of functions declared in the program must differ from the names of the compiler intrinsic functions.

Definition 5.35 (C-IL External Function Predicate). In order to test whether a given function $f \in \mathbb{F}_{name}$ declared in the program π (i.e. $f \in \text{dom}\left(\mathcal{F}_\pi^\theta\right)$) is an external one we use the predicate

$$ext(f, \pi, \theta) \overset{def}{\equiv} \mathcal{F}_\pi^\theta(f).P = \mathbf{extern}$$

5.2.7 Semantics of C-IL Memory Accesses

In C-IL we access the byte-addressable memory in order to read and write C-IL values. For converting values between the byte-string and C-IL representations, we provide auxiliary functions introduced in [Sch13].

Definition 5.36 (Converting Values to/from Byte-Strings). We define two partial functions

$$\mathit{val2bytes} : \mathit{val} \rightharpoonup (\mathbb{B}^8)^*$$
$$\mathit{bytes2val} : (\mathbb{B}^8)^* \times \mathbb{T} \rightharpoonup \mathit{val}$$

such that

$$\mathit{val2bytes}(v) \overset{def}{\equiv} \begin{cases} \mathit{bits2bytes}(b) & : \quad v = \mathbf{val}(b, t) \\ \mathit{undefined} & : \quad \text{otherwise} \end{cases}$$

$$\mathit{bytes2val}(bs, t) \overset{def}{\equiv} \begin{cases} \mathbf{val}(\mathit{bytes2bits}(bs), t) & : \quad t \neq \mathbf{struct}\ t_c \\ \mathit{undefined} & : \quad \text{otherwise} \end{cases}$$

Now we can define functions which read and write from/to the global and local memories of the C-IL machine.

Definition 5.37 (Reading Byte-Strings from Global Memory). For a memory $\mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8$ of the C-IL machine, an address $a \in \mathbb{B}^{32}$, and a length $s \in \mathbb{N}$, the function $\mathit{read}_{gm}(\mathcal{M}, a, s) \in (\mathbb{B}^8)^*$ returns a byte-string of length s computed as follows:

$$\mathit{read}_{gm}(\mathcal{M}, a, s) \overset{def}{\equiv} \begin{cases} \mathcal{M}(a) & : \quad s = 1 \\ \mathit{read}_{gm}(\mathcal{M}, a +_{32} 1_{32}, s - 1) \circ \mathcal{M}(a) & : \quad \text{otherwise} \end{cases}$$

Definition 5.38 (Writing Byte-Strings to Global Memory). Analogously, for a memory $\mathcal{M}$ and a byte-string $bs \in (\mathbb{B}^8)^*$ to be written to the memory at a starting address $a \in \mathbb{B}^{32}$ we define the function $\mathit{write}_{gm}(\mathcal{M}, a, bs) : \mathbb{B}^{32} \to \mathbb{B}^8$ updating the memory in a way such that for any byte address $x \in \mathbb{B}^{32}$ we have

$$\mathit{write}_{gm}(\mathcal{M}, a, bs)(x) \overset{def}{\equiv} \begin{cases} bs[\langle x \rangle - \langle a \rangle] & : \quad 0 \leq \langle x \rangle - \langle a \rangle < |bs| \\ \mathcal{M}(x) & : \quad \text{otherwise} \end{cases}$$

Definition 5.39 (Reading Byte-Strings from Local Memory). A byte-string of length $s \in \mathbb{N}$ read from a local memory $\mathcal{M}_{\mathcal{E}} : \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*$ for a local variable $v \in \mathbb{V}$ starting at an offset $o \in \mathbb{N}_0$ is computed by the partial function

$$\mathit{read}_{lm}(\mathcal{M}_{\mathcal{E}}, v, o, s) \overset{def}{\equiv} \mathcal{M}_{\mathcal{E}}(v)[o + s - 1] \circ \cdots \circ \mathcal{M}_{\mathcal{E}}(v)[o]$$

If $s + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin \mathrm{dom}\,(\mathcal{M}_{\mathcal{E}})$, then the function $\mathit{read}_{lm}(\mathcal{M}_{\mathcal{E}}, v, o, s)$ is undefined for the given parameters.

Definition 5.40 (Writing Byte-Strings to Local Memory). In order to write a byte-string $bs \in (\mathbb{B}^8)^*$ to a variable $v \in \mathbb{V}$ from a local memory $\mathcal{M}_{\mathcal{E}} : \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*$ at an offset $o \in \mathbb{N}_0$, we introduce the function $\mathit{write}_{lm}(\mathcal{M}_{\mathcal{E}}, v, o, bs)$ updating the local memory in a way such that for any $w \in \mathbb{V}$ and $i < |\mathcal{M}_{\mathcal{E}}(w)|$ one gets

$$\mathit{write}_{lm}(\mathcal{M}_{\mathcal{E}}, v, o, bs)(w)[i] \overset{def}{\equiv} \begin{cases} bs[i - o] & : \quad w = v \wedge o \leq i < o + |bs| \\ \mathcal{M}_{\mathcal{E}}(w)[i] & : \quad \text{otherwise} \end{cases}$$

If $|bs| + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin \mathrm{dom}\,(\mathcal{M}_{\mathcal{E}})$, then the function $\mathit{write}_{lm}(\mathcal{M}_{\mathcal{E}}, v, o, bs)$ is undefined.

Finally, we are ready to define similar functions on a C-IL configuration.

Definition 5.41 (Reading a Value from a C-IL Configuration). Given the environment parameters $\theta \in \mathit{Params}_{\mathrm{CIL}}$, a C-IL configuration $c \in \mathbb{C}_{\mathrm{CIL}}$, and a pointer value $x \in \mathit{val}$, the partial function $\mathit{read}(\theta, c, x) \in \mathit{val}$ returns a C-IL value read from the memory pointed to:

$$\mathit{read}(\theta, c, x) \overset{\mathit{def}}{\equiv} \begin{cases} \mathit{bytes2val}(\mathit{read}_{gm}(c.\mathcal{M}, a, \mathit{size}_\theta(t)), t) & : x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ \mathit{bytes2val}(\mathit{read}_{lm}(c.\mathit{stack}[i].\mathcal{M}_\mathcal{E}, v, o, \mathit{size}_\theta(t)), t) & : x = \mathbf{lref}((v,o), i, \mathbf{ptr}(t)) \\ \mathit{read}(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t))) & : x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ \mathit{read}(\theta, c, \mathbf{lref}((v,o), i, \mathbf{ptr}(t))) & : x = \mathbf{lref}((v,o), i, \mathbf{array}(t, n)) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

Definition 5.42 (Writing a Value to a C-IL Configuration). In order to write a given C-IL value $y \in \mathit{val}$ to a C-IL configuration $c \in \mathbb{C}_{\mathrm{CIL}}$ at the memory pointed to by a pointer $x \in \mathit{val}$ according to the environment parameters θ, we define the function

$$\mathit{write}(\theta, c, x, y) \overset{\mathit{def}}{\equiv} \begin{cases} c\,[\mathcal{M} := \mathit{write}_{gm}(c.\mathcal{M}, a, \mathit{val2bytes}(y))] & : x = \mathbf{val}(a, \mathbf{ptr}(t)) \land \\ & \; y = \mathbf{val}(b, t) \\ c' & : x = \mathbf{lref}((v,o), i, \mathbf{ptr}(t)) \land \\ & \; y = \mathbf{val}(b, t) \\ \mathit{write}(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & : x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ \mathit{write}(\theta, c, \mathbf{lref}((v,o), i, \mathbf{ptr}(t)), y) & : x = \mathbf{lref}((v,o), i, \mathbf{array}(t, n)) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

where $c'.\mathit{stack}[i].\mathcal{M}_\mathcal{E} = \mathit{write}_{lm}(c.\mathit{stack}[i].\mathcal{M}_\mathcal{E}, v, o, \mathit{val2bytes}(y))$ and all other components of c' are identical to c.

5.2.8 Type and Expression Evaluation

In order to define expression evaluation, we first introduce a few auxiliary functions computing types of values, variables, functions, and expressions.

Definition 5.43 (Type of a Value). For a given C-IL value $x \in \mathit{val}$, the function $\tau(x) \in \mathbb{T}$ extracts the type of x as

$$\tau(x) \overset{\mathit{def}}{\equiv} \begin{cases} t & : x = \mathbf{val}(y, t) \\ t & : x = \mathbf{fun}(y, t) \\ t & : x = \mathbf{lref}((v,o), i, t) \end{cases}$$

Definition 5.44 (Declared Variables). For a variable declaration list $V \in (\mathbb{V} \times \mathbb{T}_\mathbb{Q})^*$, we define the function $\mathit{decl}(V) \in 2^\mathbb{V}$ that returns a set of declared variable names:

$$\mathit{decl}(V) \overset{\mathit{def}}{\equiv} \begin{cases} \{v\} \cup \mathit{decl}(V') & : V = V' \circ (v, t) \\ \emptyset & : V = \varepsilon \end{cases}$$

Definition 5.45 (Type of a Variable in a Declaration List). Given a variable $v \in \mathbb{V}$, and a declaration list $T \in (\mathbb{V} \times \mathbb{T}_\mathbb{Q})^*$, the function $\tau_V(v, T) \in \mathbb{T}_\mathbb{Q}$ returns the type of this variable, if it is present in the list:

$$\tau_V(v, T) \overset{\mathit{def}}{\equiv} \begin{cases} t & : T = T' \circ (v, t) \\ \tau_V(v, T') & : T = T' \circ (v', t) \land v' \neq v \\ \mathit{undefined} & : T = \varepsilon \end{cases}$$

Definition 5.46 (Type of a Field in a Declaration List). Analogously, for a given field $f \in \mathbb{F}$, and a declaration list $T \in (\mathbb{F} \times \mathbb{T}_\mathbb{Q})^*$, the function $\tau_F(f, T) \in \mathbb{T}_\mathbb{Q}$ is defined as

$$\tau_F(f,T) \overset{def}{\equiv} \begin{cases} t & : T = T' \circ (f,t) \\ \tau_F(f,T') & : T = T' \circ (f',t) \wedge f' \neq f \\ \mathit{undefined} & : T = \varepsilon \end{cases}$$

Recall, that the declaration lists of global variables, fields for each struct type, and local variables for each function are given in the C-IL program. In the evaluation, we will often need to refer to the declaration of the local variables used in the top-most stack frame. For this purpose we use the already introduced shorthand $V_{top}(c, \pi)$ for a configuration $c \in \mathbb{C}_{\mathrm{CIL}}$ and a program $\pi \in \mathit{Prog}_{\mathrm{CIL}}$.

Definition 5.47 (Type of a Function). In order to extract the type information for a function with a name $f \in \mathbb{F}_{name}$ from a given function table $\mathcal{F} : \mathbb{F}_{name} \rightharpoonup \mathit{FunT}$ we define the function $\tau^{\mathcal{F}}_{\mathit{fun}}(f) \in \mathbb{T}_\mathbb{Q}$ such that

$$\tau^{\mathcal{F}}_{\mathit{fun}}(f) \overset{def}{\equiv} \begin{cases} (\emptyset, \mathbf{funptr}\,(\mathcal{F}(f).\mathit{rettype}, T)) & : f \in \mathrm{dom}\,(\mathcal{F}) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

where the list T of length $\mathit{npar} \equiv \mathcal{F}(f).\mathit{npar}$ contains the parameter types T_i such that for all $i \in \mathbb{N}_{\mathit{npar}}$ the predicate $\exists v_i \in \mathbb{V}.\ \mathcal{F}(f).V[i] = (v_i, T_i)$ holds.

Definition 5.48 (Qualified Type Evaluation). Now, for computing a qualified type of a given expression in a body of a non-external C-IL function $f \in \mathbb{F}_{name}$ of a program $\pi \in \mathit{Prog}_{\mathrm{CIL}}$ wrt. the environment parameters θ, we define the function

$$\tau^{\pi,\theta}_f : \mathbb{E} \to \mathbb{T}_\mathbb{Q}$$

by a case distinction over a given expression:

- constant: $x \in \mathit{val}$

$$\tau^{\pi,\theta}_f(x) \overset{def}{\equiv} (\emptyset, \tau(x))$$

- variable names: $v \in \mathbb{V}$

$$\tau^{\pi,\theta}_f(v) \overset{def}{\equiv} \begin{cases} \tau_V\,(v, \pi.\mathcal{F}(f).V) & : v \in \mathit{decl}\,(\pi.\mathcal{F}(f).V) \\ \tau_V(v, \pi.V_G) & : v \notin \mathit{decl}\,(\pi.\mathcal{F}(f).V) \wedge v \in \mathit{decl}(\pi.V_G) \\ (\emptyset, \mathbf{void}) & : \text{otherwise} \end{cases}$$

- function names: $fn \in \mathbb{F}_{name}$

$$\tau^{\pi,\theta}_f(fn) \overset{def}{\equiv} \tau^{\mathcal{F}^\theta_\pi}_{\mathit{fun}}(fn)$$

- unary operator: $e \in \mathbb{E}, \oslash \in \mathbb{O}_1$

$$\tau^{\pi,\theta}_f(\oslash e) \overset{def}{\equiv} \tau^{\pi,\theta}_f(e)$$

- binary operator: $e_1, e_2 \in \mathbb{E}, \otimes \in \mathbb{O}_2$

$$\tau^{\pi,\theta}_f(e_1 \otimes e_2) \overset{def}{\equiv} \tau^{\pi,\theta}_f(e_1)$$

  Note here that since the type is determined by the first operand, one has to exclude pointer arithmetic with the first operand of a primitive type.

- ternary operator: $e, e_1, e_2 \in \mathbb{E}$

$$\tau_f^{\pi,\theta}(e \mathrel{?} e_1 : e_2) \overset{def}{\equiv} \tau_f^{\pi,\theta}(e_1)$$

- type cast: $t \in \mathbb{T}_\mathbb{Q}, e \in \mathbb{E}$

$$\tau_f^{\pi,\theta}((t)e) \overset{def}{\equiv} t$$

- pointer dereferencing: $e \in \mathbb{E}$

$$\tau_f^{\pi,\theta}(*(e)) \overset{def}{\equiv} \begin{cases} t & : \tau_f^{\pi,\theta}(e) = (q, \mathbf{ptr}(t)) \\ t & : \tau_f^{\pi,\theta}(e) = (q, \mathbf{array}(t,n)) \\ (\emptyset, \mathbf{void}) & : \text{otherwise} \end{cases}$$

- address of: $e \in \mathbb{E}$

$$\tau_f^{\pi,\theta}(\&(e)) \overset{def}{\equiv} \begin{cases} \tau_f^{\pi,\theta}(e') & : e = *(e') \\ (\emptyset, \mathbf{ptr}(\tau_f^{\pi,\theta}(v))) & : e = v \\ (\emptyset, \mathbf{ptr}(q' \cup q'', X)) & : e = (e').z \wedge \tau_f^{\pi,\theta}(e') = (q', \mathbf{struct}\ t_C) \wedge \\ & \quad \tau_F(z, \pi.T_F(t_C)) = (q'', X) \\ (\emptyset, \mathbf{void}) & : \text{otherwise} \end{cases}$$

- field access: $e \in \mathbb{E}, z \in \mathbb{F}$

$$\tau_f^{\pi,\theta}((e).z) \overset{def}{\equiv} \tau_f^{\pi,\theta}(*(\&((e).z)))$$

- size of a type or an expression: $x \in \mathbb{T}_\mathbb{Q} \cup \mathbb{E}$

$$\tau_f^{\pi,\theta}(\mathbf{sizeof}(x)) \overset{def}{\equiv} (\emptyset, \theta.size\_t)$$

Note that this type evaluation was defined in [Sch13] for a given C-IL configuration instead of a given function. The version of the evaluation $\tau_f^{\pi,\theta}(e)$ considered here in detail was introduced in [Bau14b] without its definition. Obviously, one can easily define the type evaluation on a given configuration $c \in \mathbb{C}_{\text{CIL}}$ as

$$\tau_c^{\pi,\theta}(e) \overset{def}{\equiv} \tau_{f_{top}(c)}^{\pi,\theta}(e)$$

Definition 5.49 (Field Reference Computation). Given a C-IL program $\pi \in Prog_{\text{CIL}}$ and the environment parameters θ, we define the partial function $\sigma_\theta^\pi(x, f) \in val$ taking a pointer or a local reference $x \in val$ to a struct type and computing a pointer or a local reference to a given field $f \in \mathbb{F}$ in the following way

$$\sigma_\theta^\pi(x, f) \overset{def}{\equiv} \begin{cases} \mathbf{val}(a +_{32} o'_{32}, \mathbf{ptr}(t'')) & : x = \mathbf{val}(a, t) \wedge t \in \{\mathbf{ptr}(t'), \mathbf{array}(t', n)\} \\ \mathbf{lref}((v, o + o'), i, \mathbf{ptr}(t'')) & : x = \mathbf{lref}((v, o), i, t) \wedge t \in \{\mathbf{ptr}(t'), \mathbf{array}(t', n)\} \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

where the shorthands t', t'', and o' denote

$$t' \equiv \mathbf{struct}\ t_C \qquad t'' \equiv qt2t(\tau_F(f, \pi.T_F(t_C))) \qquad o' \equiv \theta.\mathit{offset}_{struc}(t_C, f)$$

The definition of $\sigma_\theta^\pi(x, f)$ was introduced in the original work [Sch13]. However, the computation was defined only for values x of pointer types. Therefore, the C-IL semantics did not allow to access a field in an array of structures. Since such arrays are pretty common in the programming practice, the definition of $\sigma_\theta^\pi(x, f)$ was extended here.

Definition 5.50 (Expression Evaluation). Finally, the function $[\![\cdot]\!]_c^{\pi,\theta} : \mathbb{E} \rightharpoonup val$ evaluates a given expression of a program $\pi \in Prog_{\mathrm{CIL}}$ in a C-IL configuration $c \in \mathbb{C}_{\mathrm{CIL}}$ wrt. the environment parameters θ in the following way:

- constant: $x \in val$

$$[\![x]\!]_c^{\pi,\theta} \overset{def}{\equiv} x$$

- variable names: $v \in \mathbb{V}$

$$[\![v]\!]_c^{\pi,\theta} \overset{def}{\equiv} [\![*(\&(v))]\!]_c^{\pi,\theta}$$

- function names: $fn \in \mathbb{F}_{name}$

$$[\![fn]\!]_c^{\pi,\theta} \overset{def}{\equiv} \begin{cases} \mathbf{val}\left(\theta.\mathcal{F}_{adr}(fn), qt2t\left(\tau_{fun}^{\mathcal{F}_\pi^\theta}(fn)\right)\right) & : fn \in \mathrm{dom}\left(\mathcal{F}_\pi^\theta\right) \wedge fn \in \mathrm{dom}\left(\theta.\mathcal{F}_{adr}\right) \\ \mathbf{fun}\left(fn, qt2t\left(\tau_{fun}^{\mathcal{F}_\pi^\theta}(fn)\right)\right) & : fn \in \mathrm{dom}\left(\mathcal{F}_\pi^\theta\right) \wedge fn \notin \mathrm{dom}\left(\theta.\mathcal{F}_{adr}\right) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

- unary operator: $e \in \mathbb{E}, \oslash \in \mathbb{O}_1$

$$[\![\oslash e]\!]_c^{\pi,\theta} \overset{def}{\equiv} \oslash [\![e]\!]_c^{\pi,\theta}$$

- binary operator: $e_1, e_2 \in \mathbb{E}, \otimes \in \mathbb{O}_2$

$$[\![e_1 \otimes e_2]\!]_c^{\pi,\theta} \overset{def}{\equiv} [\![e_1]\!]_c^{\pi,\theta} \otimes [\![e_2]\!]_c^{\pi,\theta}$$

- ternary operator: $e, e_1, e_2 \in \mathbb{E}$

$$[\![e \mathbin{?} e_1 : e_2]\!]_c^{\pi,\theta} \overset{def}{\equiv} \begin{cases} [\![e_1]\!]_c^{\pi,\theta} & : \neg zero_\theta\left([\![e]\!]_c^{\pi,\theta}\right) \\ [\![e_2]\!]_c^{\pi,\theta} & : zero_\theta\left([\![e]\!]_c^{\pi,\theta}\right) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

- type cast: $t \in \mathbb{T}_{\mathbb{Q}}, e \in \mathbb{E}$

$$[\![(t)e]\!]_c^{\pi,\theta} \overset{def}{\equiv} \theta.cast\left([\![e]\!]_c^{\pi,\theta}, qt2t(t)\right)$$

- pointer dereferencing: $e \in \mathbb{E}$

$$[\![*(e)]\!]_c^{\pi,\theta} \overset{def}{\equiv} \begin{cases} read(\theta, c, [\![e]\!]_c^{\pi,\theta}) & : \left(\tau\left([\![e]\!]_c^{\pi,\theta}\right) = \mathbf{ptr}(t) \wedge \neg isarray(t)\right) \vee \\ & \quad \tau\left([\![e]\!]_c^{\pi,\theta}\right) = \mathbf{array}(t,n) \\ \mathbf{val}(a, \mathbf{array}(t,n)) & : [\![e]\!]_c^{\pi,\theta} = \mathbf{val}(a, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{lref}((v,o), i, \mathbf{array}(t,n)) & : [\![e]\!]_c^{\pi,\theta} = \mathbf{lref}((v,o), i, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

- address of: $e \in \mathbb{E}$

$$[\![\&(e)]\!]_c^{\pi,\theta} \overset{def}{\equiv} \begin{cases} [\![e']\!]_c^{\pi,\theta} & : e = *(e') \\ \mathbf{lref}((v,0), |c.stack|, \mathbf{ptr}(t')) & : e = v \wedge v \in decl(V_{top}(c,\pi)) \\ \mathbf{val}(\theta.alloc_{gvar}(v), \mathbf{ptr}(t'')) & : e = v \wedge v \notin decl(V_{top}(c,\pi)) \wedge \\ & \quad v \in decl(\pi.V_G) \\ \sigma_\theta^\pi\left([\![\&(e')]\!]_c^{\pi,\theta}, f\right) & : e = (e').f \\ \mathit{undefined} & : \text{otherwise} \end{cases}$$

where

$$t' \equiv qt2t(\tau_V(v, V_{top}(c,\pi))) \qquad\qquad t'' \equiv qt2t(\tau_V(v, \pi.V_G))$$

- field access: $e \in \mathbb{E}, f \in \mathbb{F}$

$$[\![(e).f]\!]_c^{\pi,\theta} \overset{def}{\equiv} [\![*(\&((e).f))]\!]_c^{\pi,\theta}$$

- size of a type : $t \in \mathbb{T}_{\mathbb{Q}}$

$$[\![\mathbf{sizeof}(t)]\!]_c^{\pi,\theta} \overset{def}{\equiv} \mathbf{val}(size_\theta(qt2t(t))_{32}, \theta.size\_t)$$

- size of an expression: $e \in \mathbb{E}$

$$[\![\mathbf{sizeof}(e)]\!]_c^{\pi,\theta} \overset{def}{\equiv} [\![\mathbf{sizeof}(\tau_c^{\pi,\theta}(e))]\!]_c^{\pi,\theta}$$

5.2.9 Transition Function

First, we introduce functions performing specific updates on a C-IL configuration.

Definition 5.51 (C-IL Configuration Update Functions). For a given C-IL configuration $c \in \mathbb{C}_{\mathrm{CIL}}$ with a non-empty stack $c.stack \neq \varepsilon$, we define the following operations where $t \equiv top(c)$:

- incrementing the location counter:

$$inc_{loc}(c) \overset{def}{\equiv} c\left[stack := c.stack[t \mapsto frame']\right]$$

$$\text{with } frame' \equiv c.stack[t]\,[loc := loc_{top}(c) + 1]$$

- setting the location counter to $l \in \mathbb{N}$:

$$set_{loc}(c, l) \overset{def}{\equiv} c\left[stack := c.stack[t \mapsto frame']\right]$$

$$\text{with } frame' \equiv c.stack[t]\,[loc := l]$$

- removing the top-most frame:

$$drop_{frame}(c) \overset{def}{\equiv} c\,[stack := c.stack[1 : t-1]]$$

- setting the return destination to $x \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$:

$$set_{rds}(c, x) \overset{def}{\equiv} c\left[stack := c.stack[t \mapsto frame']\right]$$

$$\text{with } frame' \equiv c.stack[t]\,[rds := x]$$

Obviously, we are interested in the steps of the C-IL machine performed on well-formed (or valid) configurations.

Definition 5.52 (**Well-Formed C-IL Configuration**). A sequential C-IL configuration $c \in \mathbb{C}_{\text{CIL}}$ is *well-formed* wrt. a C-IL program $\pi \in \mathit{Prog}_{\text{CIL}}$ and the environment parameters θ iff its stack $st \equiv c.stack$ is *well-formed*. This means that (i) the stack st is not empty and (ii) every stack frame (a) corresponds to a function implemented in the given C-IL program, and (b) has the location inside the function. Moreover, (c) the local memory of each frame contains all variables and parameters declared in the function and (d) has the proper size. Additionally, (e) if the function is called with a return value, its destination corresponds to the type of the return value in the function declaration.

$$\begin{aligned}
\mathit{wfstack}^{\pi,\theta}_{\text{CIL}}(st) \overset{def}{\equiv}\ & (i)\quad st \neq \varepsilon \\
& (ii)\quad \forall\, i \in \mathbb{N}_{top(st)}. \\
& \qquad (a)\quad f_i(st) \in \operatorname{dom}\left(\mathcal{F}^{\theta}_{\pi}\right) \wedge \neg ext(f_i(st), \pi, \theta) \\
& \qquad (b)\quad loc_i(st) \in [1 : |P_i(st, \pi)|] \\
& \qquad (c)\quad \operatorname{dom}(\mathcal{M}_{\mathcal{E}\,i}(st)) = decl(V_i(st, \pi)) \\
& \qquad (d)\quad \forall\, (v, t) \in V_i(st, \pi).\ |\mathcal{M}_{\mathcal{E}\,i}(st)(v)| = size_{\theta}(qt2t(t)) \\
& \qquad (e)\quad i < top(st) \wedge \tau(rds_i(st)) \in \{\mathbf{ptr}(t), \mathbf{array}(t, n)\} \implies \\
& \qquad\qquad t = qt2t(rettype_{i+1}(st, \pi))
\end{aligned}$$

Finally, we can formalize the operational semantics of the sequential C-IL.

Definition 5.53 (**Sequential C-IL Transition Function**). For a given C-IL program $\pi \in \mathit{Prog}_{\text{CIL}}$ and the environment parameters $\theta \in \mathit{Params}_{\text{CIL}}$, the transitions of the sequential C-IL machine are defined by the function

$$\delta^{\pi,\theta}_{\text{CIL}} : \mathbb{C}_{\text{CIL}} \to \mathbb{C}_{\text{CIL}\perp}$$

with $\mathbb{C}_{\text{CIL}\perp} \overset{def}{\equiv} \mathbb{C}_{\text{CIL}} \cup \{\perp\}$ containing the error state $\perp$.

Note that the C-IL transition function in [Sch13] has an additional input resolving the non-deterministic choice of initial values for local variables in the newly created stack frame. In the scope a this work, however, analogously to the C0 semantics [PBLS15] we initialize the local variables with zeros. Moreover, we also use the error state to signal any run-time errors.

So, for a C-IL configuration $c \in \mathbb{C}_{\text{CIL}}$ we define $\delta^{\pi,\theta}_{\text{CIL}}(c)$ by a case split on the next C-IL statement to be executed and a kind of a function in case of a function call:

- Assignment: $stmt_{next}(c, \pi) = (e_0 = e_1)$

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} inc_{loc}\left(write(\theta, c, [\![\&(e_0)]\!]^{\pi,\theta}_{c}, [\![e_1]\!]^{\pi,\theta}_{c})\right)$$

- Goto: $stmt_{next}(c, \pi) = \textbf{goto}\ l$

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} \begin{cases} set_{loc}\,(c, l) & : \ l \leq |P_{top}(c, \pi)| \\ \perp & : \ \text{otherwise} \end{cases}$$

- If-Not-Goto: $stmt_{next}(c, \pi) = \textbf{ifnot}\ e\ \textbf{goto}\ l$

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} \begin{cases} set_{loc}\,(c, l) & : \ zero_{\theta}([\![e]\!]^{\pi,\theta}_{c}) \wedge l \leq |P_{top}(c, \pi)| \\ inc_{loc}(c) & : \ \neg zero_{\theta}([\![e]\!]^{\pi,\theta}_{c}) \\ \perp & : \ \text{otherwise} \end{cases}$$

- Function return without a return value: $stmt_{next}(c, \pi) = \textbf{return}$

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} \begin{cases} c' & : rds_{top}(c') = \bot \wedge top(c) > 1 \\ \bot & : \text{otherwise} \end{cases}$$

where $c' \equiv drop_{frame}(c)$

- Function return with a return value: $stmt_{next}(c, \pi) = \textbf{return e}$

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} \begin{cases} write\left(\theta, set_{rds}(c', \bot), rds_{top}(c'), [\![e]\!]^{\pi,\theta}_{c}\right) & : rds_{top}(c') \neq \bot \wedge top(c) > 1 \\ c' & : rds_{top}(c') = \bot \wedge top(c) > 1 \\ \bot & : \text{otherwise} \end{cases}$$

where $c' \equiv drop_{frame}(c)$

- Non-external function call:
  - the next statement is a function call with or without a return value

$$stmt_{next}(c, \pi) \in \{e_0 = \textbf{call } e(E), \textbf{call } e(E)\}$$

  - the expression e evaluates to some non-external function f

$$isfunc([\![e]\!]^{\pi,\theta}_{c}, f, \theta) \wedge f \in \text{dom}\left(\mathcal{F}^{\theta}_{\pi}\right) \wedge \neg ext(f, \pi, \theta)$$

  - the types of all parameters match the function declaration

$$|E| = \mathcal{F}^{\theta}_{\pi}(f).npar \wedge \forall i \in [1 : |E|].\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \implies \tau^{\pi,\theta}_{c}(E[i]) = t$$

  - for a call with a return value the function must return a result of a non-**void** type

$$stmt_{next}(c, \pi) = (e_0 = \textbf{call } e(E)) \implies qt2t\left(\mathcal{F}^{\theta}_{\pi}(f).rettype\right) \neq \textbf{void}$$

If these conditions hold, the non-error result of the transition is defined as

$$\delta^{\pi,\theta}_{\text{CIL}}(c) \overset{def}{\equiv} c'$$

such that

$$c'.stack = inc_{loc}(set_{rds}(c, rds')).stack \circ (f, \bot, 1, \mathcal{M}'_{\mathcal{E}})$$
$$c'.\mathcal{M} = c.\mathcal{M}$$

where

$$rds' \equiv \begin{cases} [\![\&(e_0)]\!]^{\pi,\theta}_{c} & : stmt_{next}(c, \pi) = (e_0 = \textbf{call } e(E)) \\ \bot & : stmt_{next}(c, \pi) = \textbf{call } e(E) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes\left([\![E[i]]\!]^{\pi,\theta}_{c}\right) & : \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \wedge i \leq \mathcal{F}^{\theta}_{\pi}(f).npar \\ (0^8)^{size_{\theta}(qt2t(t))} & : \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \wedge i > \mathcal{F}^{\theta}_{\pi}(f).npar \\ undefined & : \text{otherwise} \end{cases}$$

As follows from the computation of $\mathcal{M}'_{\mathcal{E}}$, in C-IL semantics any function cannot have parameters not reperesentable as byte-strings, e.i., local references and symbolic function values.

- Compare-and-swap call (external intrinsic function):
  - the next statement is a function call without a return value
    $$stmt_{next}(c, \pi) = \textbf{call } e(E)$$
  - the expression e evaluates to the compare-and-swap intrinsic function $cas \in \mathbb{F}_{name}$
    $$isfunc([\![e]\!]_c^{\pi,\theta}, cas, \theta)$$
  - the types of all parameters match the function declaration
    $$|E| = \mathcal{F}_\pi^\theta(f).npar \wedge \forall i \in \mathbb{N}_{|E|}.\ \mathcal{F}_\pi^\theta(f).V[i] = (v, t) \implies qt2t\left(\tau_c^{\pi,\theta}(E[i])\right) = qt2t\,(t)$$

  Let the input parameters be $E = e_{dest} \circ e_{cmp} \circ e_{exch} \circ e_{ret}$. Then the transition result is defined as

  $$\delta_{\text{CIL}}^{\pi,\theta}(c) \overset{def}{\equiv} \begin{cases} inc_{loc}\left(write\left(\theta, c', [\![e_{dest}]\!]_c^{\pi,\theta}, [\![e_{exch}]\!]_c^{\pi,\theta}\right)\right) & : \ val_{read} = [\![e_{cmp}]\!]_c^{\pi,\theta} \\ inc_{loc}(c') & : \ val_{read} \neq [\![e_{cmp}]\!]_c^{\pi,\theta} \end{cases}$$

  where $val_{read} \equiv read\left(\theta, c, [\![e_{dest}]\!]_c^{\pi,\theta}\right)$ and $c' \equiv write\left(\theta, c, [\![e_{ret}]\!]_c^{\pi,\theta}, val_{read}\right)$.

In all other cases, or if the evaluation of any expression present in the statement fails, or the application of any other functions involved into the transition is undefined, the computation also leads to the error state $\delta_{\text{CIL}}^{\pi,\theta}(c) \overset{def}{\equiv} \bot$.

5.3 Sequential Mixed Machine Semantics

5.3.1 MX Programs

Definition 5.54 (MX Program). A mixed machine program is simply represented by a couple of the C-IL and MASM programs:

$$Prog_{\text{MX}} \overset{def}{\equiv} Prog_{\text{MASM}} \times Prog_{\text{CIL}}$$

In this chapter we will mostly consider an MX program $\pi \in Prog_{\text{MX}}$ composed as $\pi = (\pi_\mu, \pi_{\text{cil}})$.

5.3.2 Environment Parameters

Since the environment parameters are only required for the C-IL semantics, we directly use the same $\theta \in Params_{\text{CIL}}$ in this section.

5.3.3 Machine Configuration

In order to obtain an integrated model of C-IL and MASM, one has to consider a list of stacks of both machines with a common byte-addressable memory and some additional individual components relevant for the machine execution and switching between C-IL and MASM. According to [Sha12] such stacks with additional information are called *execution contexts*. A context corresponding to a machine currently performing a step is an *active* one whereas all others are considered to be *inactive*. In fact, the active execution context is a configuration of the corresponding machine without the memory.

Before defining states of the MX machine, we first consider such contexts in detail.

Definition 5.55 (**Active MASM Context**). An active MASM execution context consists of the MASM stack and general purpose registers.

$$context^{active}_{\mathrm{MASM}} \overset{def}{\equiv} \left(stack \in frame^*_{\mathrm{MASM}},\ gpr : \mathbb{B}^5 \rightharpoonup \mathbb{B}^{32}\right)$$

Definition 5.56 (**Inactive MASM Context**). Instead of all GPRs used in the MASM configuration and the MASM active context, an inactive MASM context keeps additionally to the stack only the callee-save registers required for restoring the GPRs content after returning from the C-IL machine.

$$context^{inactive}_{\mathrm{MASM}} \overset{def}{\equiv} \left(stack \in frame^*_{\mathrm{MASM}},\ gpr_{callee} : Reg_{callee} \to \mathbb{B}^{32}\right)$$

Note that gpr_{callee} contains the state of registers before a C-IL function call from a MASM procedure in the topmost frame.

In contrast to the MASM execution contexts, active and inactive C-IL contexts are represented by the C-IL stack.

Definition 5.57 (**C-IL Context**). Therefore, in both cases we operate with the C-IL context containing a list of C-IL frames.

$$context_{\mathrm{CIL}} \overset{def}{\equiv} frame^*_{\mathrm{CIL}}$$

Note that in the original version of the MX model in [SS12, Sha12] the inactive C-IL context contains the callee-save registers too. However, during the work on this thesis together with the authors of [SS12, Sha12] it was discovered that this component is not needed for the operational semantics of the MX machine. Therefore, we do not include it into MX machine configurations here.

Definition 5.58 (**Sequential MX Configuration**). Then mixed machine configurations are represented by a set of tuples

$$\begin{aligned}\mathbb{C}_{\mathrm{MX}} \overset{def}{\equiv} \Big(& ac \in context_{\mathrm{CIL}} \cup context^{active}_{\mathrm{MASM}},\\ & ic \in \left(context_{\mathrm{CIL}} \cup context^{inactive}_{\mathrm{MASM}}\right)^*,\\ & spr : \mathbb{B}^5 \to \mathbb{B}^{32},\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8\Big)\end{aligned}$$

containing the active context ac, the list ic of inactive execution contexts, the byte addressable memory $\mathcal{M}$, and the special purpose registers spr.

We model SPRs as a component visible even during the C-IL execution. One of the reasons stated in [SS12] is future work modeling interrupts in the C-IL semantics. Moreover, according to the compiler calling convention, the compilers are not supposed to save and restore the special purpose registers during function/procedure calls and returns. Hence, it is reasonable to keep the values of the special purpose registers during C-IL steps of the mixed machine.

5.3.4 Transition Function

Before we consider the semantics of the mixed machine, we introduce a few auxiliary definitions and shorthands.

Definition 5.59 (Type of Context). We define predicates checking whether an execution context $k \in context_{\mathrm{CIL}} \cup context^{active}_{\mathrm{MASM}} \cup context^{inactive}_{\mathrm{MASM}}$ corresponds to C-IL or MASM respectively.

$$\begin{aligned} cil(k) &\overset{def}{\equiv} (k \in context_{\mathrm{CIL}}) \\ masm(k) &\overset{def}{\equiv} \left(k \in context^{active}_{\mathrm{MASM}} \cup context^{inactive}_{\mathrm{MASM}}\right) \end{aligned}$$

Definition 5.60 (Number of Frames). We compute the *number of frames in a single C-IL or MASM context k* as

$$nf_{cntx}(k) \overset{def}{\equiv} \begin{cases} |k.stack| & : masm(k) \\ |k| & : \text{otherwise} \end{cases}$$

Then for a given MX configuration $c \in \mathbb{C}_{\mathrm{MX}}$ the *number of frames in its list of inactive contexts* is denoted by

$$nf_{ic}(c) \overset{def}{\equiv} \begin{cases} \sum_{j=1}^{|c.ic|} nf_{cntx}\,(c.ic[j]) & : c.ic \neq \varepsilon \\ 0 & : \text{otherwise} \end{cases}$$

Similarly, the *number of frames in the active context* of the configuration c is

$$nf_{ac}(c) \overset{def}{\equiv} nf_{cntx}(c.ac)$$

The semantics of the sequential MX machine is obviously based on the earlier defined C-IL and MASM transition functions. Hence, in order to perform the corresponding steps of C-IL and MASM machines, we first compute their configurations.

Definition 5.61 (Construction of C-IL Configuration from MX Machine). Given an MX configuration $c \in \mathbb{C}_{\mathrm{MX}}$ with the active C-IL context, i.e., $cil(c.ac)$ holds, we construct a corresponding C-IL configuration $conf_{\mathrm{CIL}}(c) \in \mathbb{C}_{\mathrm{CIL}}$ from the MX configuration as follows:

$$conf_{\mathrm{CIL}}(c) \overset{def}{\equiv} c_{\mathrm{cil}}$$

such that the memory configuration is copied and the stack is formed from the frames of the active context and dummy frames

$$c_{\mathrm{cil}}.\mathcal{M} = c.\mathcal{M} \qquad c_{\mathrm{cil}}.stack = stack_{any} \circ c.ac$$

where $|stack_{any}| = nf_{ic}(c)$ and each dummy frame with index $i \in \mathbb{N}_{nf_{ic}(c)}$ has arbitrary value $stack_{any}[i] = \epsilon\, frame_{\mathrm{CIL}}$.

Note that the content of dummy frames does play any role in the definition of the MX semantics because the machine never relies on it. The dummy frames are introduced only for the proper numbering the C-IL frames in the overall stack of the mixed machine because the frame index is included into the local reference value. This fact was not taken into account in [Sha12].

In contrast to the C-IL configuration, the state of the MASM machine does not store indices of stack frames. Therefore, there is no need to add dummy frames in the construction of the MASM stack from the active and inactive contexts of the mixed machine.

Definition 5.62 (Construction of MASM Configuration from MX Machine). Given an MX configuration $c \in \mathbb{C}_{\mathrm{MX}}$ with the active MASM context, i.e., $masm(c.ac)$ holds, we construct a corresponding MASM configuration $conf_{\mathrm{MASM}}(c) \in \mathbb{C}_{\mathrm{MASM}}$ from the MX configuration as

$$conf_{\mathrm{MASM}}(c) \overset{def}{\equiv} c_{\mu}$$

where the components of the MASM configuration c_μ are

$$c_\mu.\mathcal{M} = c.\mathcal{M} \qquad c_\mu.stack = c.ac.stack$$

$$c_\mu.gpr = c.ac.gpr \qquad c_\mu.spr = c.spr$$

Definition 5.63 (Well-Formed MX Configuration). An active context ac and a list of inactive contexts ic are well-formed wrt. an MX program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in Prog_{\text{MX}}$ and the environment parameters θ iff (i) the stack of every context is well-formed, (ii) the GPRs of the active MASM context are well-formed, and (iii) – (v) the adjacent contexts are of different types. Let $ni \equiv |ic|$ be the number of inactive contexts. Then, the well-formedness predicate is defined as

$$\begin{aligned} wfcntx_{\text{MX}}^{\pi,\theta}(ac, ic) \overset{def}{\equiv} \quad & \textit{(i)} \quad \forall\, k \in (ic \circ ac). \\ & \qquad \big(masm(k) \implies wfstack_{\text{MASM}}^{\pi_\mu}(k.stack)\big) \wedge \Big(cil(k) \implies wfstack_{\text{CIL}}^{\pi_{\text{cil}},\theta}(k)\Big) \\ & \textit{(ii)} \quad masm(ac) \implies wfreg_{\text{MASM}}(ac.gpr) \\ & \textit{(iii)} \quad \forall\, j \in [1 : ni - 1]. \\ & \qquad (masm(ic_j) \iff cil(ic_{j+1})) \wedge (cil(ic_j) \iff masm(ic_{j+1})) \\ & \textit{(iv)} \quad masm(ac) \implies ic = \varepsilon \vee cil(ic_{ni}) \\ & \textit{(v)} \quad cil(ac) \implies ic = \varepsilon \vee masm(ic_{ni}) \end{aligned}$$

A sequential MX configuration $c \in \mathbb{C}_{\text{MX}}$ is well-formed iff its active and inactive contexts are well formed:

$$wfconf_{\text{MX}}^{\pi,\theta}(c) \overset{def}{\equiv} wfcntx_{\text{MX}}^{\pi,\theta}(c.ac, c.ic)$$

Definition 5.64 (Sequential MX Transition Function). For a given MX program $\pi \in Prog_{\text{MX}}$ such that $\pi = (\pi_\mu, \pi_{\text{cil}})$ and the environment parameters $\theta \in Params_{\text{CIL}}$, the transitions of the sequential mixed semantics machine are defined by the function

$$\delta_{\text{MX}}^{\pi,\theta} : \mathbb{C}_{\text{MX}} \times \Sigma_{\text{MX}} \to \mathbb{C}_{\text{MX}\perp}$$

where $\mathbb{C}_{\text{MX}\perp} \overset{def}{\equiv} \mathbb{C}_{\text{MX}} \cup \{\perp\}$ contains the error state $\perp$ and $\Sigma_{\text{MX}} \overset{def}{\equiv} \mathbb{B}^5 \to \mathbb{B}^{32}$ is an input alphabet used for non-deterministic choice of GPRs content during the call of a MASM procedure from C-IL context and the return to a MASM procedure from C-IL. [8]

In order to define the MX transition function on an MX configuration $c \in \mathbb{C}_{\text{MX}}$, we introduce the following shorthands:

- if the active context is MASM, i.e., $masm(c.ac)$:

$$\begin{aligned} c_\mu &\equiv conf_{\text{MASM}}(c) \\ c'_\mu &\equiv \delta_{\text{MASM}}^{\pi_\mu}(c_\mu) \\ instr_\mu &\equiv instr_{next}(c_\mu, \pi_\mu) \end{aligned}$$

- if the active context is C-IL, i.e., $cil(c.ac)$:

$$\begin{aligned} c_{\text{cil}} &\equiv conf_{\text{CIL}}(c) \\ c'_{\text{cil}} &\equiv \delta_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \\ stmt_{\text{cil}} &\equiv stmt_{next}(c_{\text{cil}}, \pi_{\text{cil}}) \end{aligned}$$

[8] Note that though this non-deterministic initialization was introduced in [Sha12], the original version of the transition function in [Sha12] has no additional input, and had to be formalized as a transition relation.

Then, the result of the computation $\delta_{\mathrm{MX}}^{\pi,\theta}(c, in)$ with $in \in \Sigma_{\mathrm{MX}}$ is considered for the cases given below and depending on the active context and the instruction/statement to be executed. In all other situations, additionally to the error-state generated by C-IL and MASM semantics, we set $\delta_{\mathrm{MX}}^{\pi,\theta}(c, in) \overset{def}{\equiv} \bot$. Note that the input in is ignored in all cases except switching from C-IL to MASM.

- Pure MASM step:
  - the active context is of MASM type, i.e., $masm(c.ac)$ holds
  - in case of a procedure call the procedure is defined in the MASM program

$$(instr_{\mu} = \textbf{call}\, pn) \implies pn \in \mathrm{dom}\,(\pi_{\mu}) \wedge \neg ext(pn, \pi_{\mu})$$

  - in case of a procedure return the active context contains more than one stack frame

$$(instr_{\mu} = \textbf{ret}) \implies nf_{ac}(c) > 1$$

  If the conditions hold, the result of the transition is computed as

$$\delta_{\mathrm{MX}}^{\pi,\theta}(c, in) \overset{def}{\equiv} \begin{cases} c' & : c'_{\mu} \neq \bot \\ \bot & : \text{otherwise} \end{cases}$$

  where the non-error state c' has the components

$$c'.ac = \left(c'_{\mu}.stack,\ c'_{\mu}.gpr\right) \qquad c'.spr = c'_{\mu}.spr$$
$$c'.ic = c.ic \qquad c'.\mathcal{M} = c'_{\mu}.\mathcal{M}$$

- Pure C-IL step:
  - the active context is of C-IL type, i.e., $cil(c.ac)$ holds
  - in case of a function call the function is either defined in the C-IL program or the external intrinsic function cas

$$stmt_{\mathrm{cil}} \in \{e_0 = \textbf{call}\, e(E), \textbf{call}\, e(E)\} \implies$$
$$isfunc\left([\![e]\!]_{c_{\mathrm{cil}}}^{\pi_{\mathrm{cil}},\theta}, f, \theta\right) \wedge f \in \mathrm{dom}\left(\mathcal{F}_{\pi_{\mathrm{cil}}}^{\theta}\right) \wedge (\neg ext(f, \pi_{\mathrm{cil}}, \theta) \vee f = cas)$$

  - in case of a function return the active context contains more than one stack frame

$$stmt_{\mathrm{cil}} \in \{e_0 = \textbf{return}\, e, \textbf{return}\} \implies nf_{ac}(c) > 1$$

  Again, if the conditions hold, the result of the transition is computed as

$$\delta_{\mathrm{MX}}^{\pi,\theta}(c, in) \overset{def}{\equiv} \begin{cases} c' & : c'_{\mathrm{cil}} \neq \bot \\ \bot & : \text{otherwise} \end{cases}$$

  where the non-error state c' has the components

$$c'.ac = c'_{\mathrm{cil}}.stack[nf_{ic}(c) + 1 : top(c'_{\mathrm{cil}})] \qquad c'.spr = c.spr$$
$$c'.ic = c.ic \qquad c'.\mathcal{M} = c'_{\mathrm{cil}}.\mathcal{M}$$

- Call from C-IL to MASM:
  - the active context is of C-IL type, i.e., $cil(c.ac)$ holds
  - the statement to be executed is the function/procedure call

$$stmt_{\text{cil}} \in \{e_0 = \textbf{call}\ e(E), \textbf{call}\ e(E)\}$$

  - the function/procedure is declared in the C-IL program as an external one

$$isfunc\left([\![e]\!]_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}, f, \theta\right) \land f \in \text{dom}\left(\mathcal{F}_{\pi_{\text{cil}}}^{\theta}\right) \land ext(f, \pi_{\text{cil}}, \theta) \land f \neq cas$$

  - and implemented in the MASM program

$$f \in \text{dom}\left(\pi_\mu\right) \land \neg ext(f, \pi_\mu)$$

  - the number of parameters matches the procedure declaration

$$|E| = \mathcal{F}_{\pi_{\text{cil}}}^{\theta}(f).npar = \pi_\mu(f).npar$$

  - the value of each parameter can be represented as a bit-string (what excludes local references and symbolic function values)

$$\forall j \in [1 : |E|].\ [\![E[j]]\!]_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta} = \textbf{val}(b_j, t_j)$$

Under these conditions the result of the transition is defined as

$$\delta_{\text{MX}}^{\pi,\theta}(c, in) \overset{def}{\equiv} c'$$

where the new configuration $c' \in \mathbb{C}_{\text{MX}}$ is formed as

$$c'.ac = \left(frame'_\mu, gpr'\right) \qquad c'.spr = c.spr$$
$$c'.ic = c.ic \circ (stack'_{\text{cil}}[nf_{ic}(c) + 1 : top(c_{\text{cil}})]) \qquad c'.\mathcal{M} = c.\mathcal{M}$$

such that the GPRs of the new active MASM context are initialized with values of the parameters according to the calling convention or the arbitrarily chosen values from the input in (meaning that a program written in MX must guarantee the expected result for any such input)

$$gpr'(r) = \begin{cases} b_j & : r = i_j \land j \in \left[1 : npar_{regs}^{\pi_\mu}(f)\right] \\ 0^{32} & : r = zero \\ in(r) & : \text{otherwise} \end{cases}$$

and the components of the newly created stack frame are initialized as follows:

$$frame'_\mu.p = f \qquad frame'_\mu.loc = 1 \qquad frame'_\mu.lifo = \varepsilon$$

$$frame'_\mu.saved(r) = \begin{cases} gpr'(r) & : \langle r \rangle \in \pi_\mu(f).uses \\ undefined & : \text{otherwise} \end{cases}$$

$$|frame'_\mu.pars| = |E|$$

$$\mathit{frame}'_\mu.\mathit{pars}[j] = \begin{cases} b_j & : j > \mathit{npar}^{\pi_\mu}_{\mathit{regs}}(f) \\ 0^{31} & : \text{otherwise} \end{cases}$$

Moreover, the C-IL context becomes inactive after storing the return value destination and increasing the location counter

$$\mathit{rds}' \equiv \begin{cases} [\![\&(e_0)]\!]^{\pi_{\mathrm{cil}},\theta}_{c_{\mathrm{cil}}} & : \mathit{stmt}_{\mathrm{cil}} = (e_0 = \textbf{call}\ e(E)) \\ \bot & : \mathit{stmt}_{\mathrm{cil}} = \textbf{call}\ e(E) \end{cases}$$

$$\mathit{stack}'_{\mathrm{cil}} \equiv \mathit{inc}_{\mathit{loc}}(\mathit{set}_{\mathit{rds}}(c_{\mathrm{cil}}, \mathit{rds}')).\mathit{stack}$$

- Return from MASM to C-IL:
  - the active context is of MASM type, i.e., $\mathit{masm}(c.\mathit{ac})$ holds
  - the instruction to be executed is the procedure return $\mathit{instr}_\mu = \textbf{ret}$
  - there is only one stack frame in the active MASM context and the list of inactive contexts is not empty
    $$\mathit{nf}_{\mathit{ac}}(c) = 1 \wedge c.\mathit{ic} \neq \varepsilon$$

In order to define the result of the computation $\delta^{\pi,\theta}_{\mathrm{MX}}(c, \mathit{in})$, we first form an intermediate MX configuration $\hat{c} \in \mathbb{C}_{\mathrm{MX}}$ with components

$$\hat{c}.\mathit{ac} = c.\mathit{ic}[\mathit{ni}] \qquad \hat{c}.\mathit{spr} = c.\mathit{spr}$$
$$\hat{c}.\mathit{ic} = c.\mathit{ic}[1 : \mathit{ni} - 1] \qquad \hat{c}.\mathcal{M} = c.\mathcal{M}$$

where $\mathit{ni} \equiv |c.\mathit{ic}|$ denotes the number of inactive contexts in the configuration. Hence, the C-IL configuration $\hat{c}_{\mathrm{cil}} \in \mathbb{C}_{\mathrm{CIL}}$ constructed from $\hat{c}$ is

$$\hat{c}_{\mathrm{cil}} \equiv \mathit{conf}_{\mathrm{CIL}}(\hat{c})$$

Now, we can compute the transition result on the C-IL configuration $\hat{c}$ similarly to the function return in the C-IL semantics. Let the return destination $a \in \mathit{val} \cup \{\bot\}$ and the value $v \in \mathit{val}$ returned for $a \neq \bot$ be

$$a \equiv \mathit{rds}_{\mathit{top}}(\hat{c}_{\mathrm{cil}})$$

$$v \equiv \textbf{val}\,(c.\mathit{ac}.\mathit{gpr}(\mathit{rv}), t) \quad \text{with} \quad \tau(a) \in \{\textbf{ptr}(t), \textbf{array}(t, n)\}$$

Then the resulting C-IL configuration $\hat{c}'_{\mathrm{cil}} \in \mathbb{C}_{\mathrm{CIL}}$ is

$$\hat{c}'_{\mathrm{cil}} \equiv \begin{cases} \mathit{write}\,(\theta, \mathit{set}_{\mathit{rds}}(\hat{c}_{\mathrm{cil}}, \bot), a, v) & : a \neq \bot \\ \hat{c}_{\mathrm{cil}} & : \text{otherwise} \end{cases}$$

and we easily define the effect of the return from MASM to C-IL as

$$\delta^{\pi,\theta}_{\mathrm{MX}}(c, \mathit{in}) \overset{\mathit{def}}{\equiv} c'$$

$$c'.\mathit{ac} = \hat{c}'_{\mathrm{cil}}.\mathit{stack}[\mathit{nf}_{\mathit{ic}}(\hat{c}) + 1 : \mathit{top}(\hat{c}'_{\mathrm{cil}})] \qquad c'.\mathit{spr} = \hat{c}.\mathit{spr}$$
$$c'.\mathit{ic} = \hat{c}.\mathit{ic} \qquad c'.\mathcal{M} = \hat{c}'_{\mathrm{cil}}.\mathcal{M}$$

- Call from MASM to C-IL:

- the active context is of MASM type, i.e., $masm(c.ac)$ holds
- the instruction to be executed is the procedure call

$$instr_\mu = \textbf{call}\ f$$

- the function/procedure is declared as an external one in the MASM program

$$f \in \operatorname{dom}(\pi_\mu) \wedge ext(f, \pi_\mu)$$

- and implemented in the C-IL program

$$f \in \operatorname{dom}\left(\mathcal{F}^{\theta}_{\pi_{\text{cil}}}\right) \wedge \neg ext(f, \pi_{\text{cil}}, \theta)$$

- the numbers of parameters in the MASM and C-IL declarations are equal

$$\pi_{\text{cil}}.\mathcal{F}(f).npar = \pi_\mu(f).npar$$

- the lifo of the top-most frame of the active context contains at least as many elements as the number of function parameters to be passed on the stack:

$$|lifo_{top}(c_\mu)| \geq npar^{\pi_\mu}_{stack}(f)$$

If these conditions hold, the result of the transition is defined as

$$\delta^{\pi,\theta}_{\text{MX}}(c, in) \overset{def}{\equiv} c'$$

$$\begin{aligned} c'.ac &= frame'_{\text{cil}} & c'.spr &= c.spr \\ c'.ic &= c.ic \circ \left(stack'_\mu, c.ac.gpr|_{Reg_{callee}}\right) & c'.\mathcal{M} &= c.\mathcal{M} \end{aligned}$$

where the components of the newly created stack frame are initialized as follows:

$$frame'_{\text{cil}}.f = f \qquad frame'_{\text{cil}}.loc = 1 \qquad frame'_{\text{cil}}.rds = \bot$$

$$frame'_{\text{cil}}.\mathcal{M}_{\mathcal{E}}(v) = \begin{cases} bits2bytes\,(c.ac.gpr(r)) & : V_f[j] = (v,t) \wedge r = i_j \wedge j \in \left[1 : npar^{\pi_\mu}_{regs}(f)\right] \\ bits2bytes\,(pars'[j-4]) & : V_f[j] = (v,t) \wedge j \in [5 : \pi_\mu(f).npar] \\ (0^8)^{size_\theta(qt2t(t))} & : V_f[j] = (v,t) \wedge j > \pi_\mu(f).npar \\ undefined & : \text{otherwise} \end{cases}$$

with the shorthands $V_f \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).V$ and $pars' \equiv rev\left(lifo_{top}(c_\mu)\,[m:n]\right)$ for the indices $n \equiv |lifo_{top}(c_\mu)|$, $m \equiv n + 1 - npar^{\pi_\mu}_{stack}(f)$.
Moreover, the active MASM context becomes inactive with the snapshot of the callee-save registers $c.ac.gpr|_{Reg_{callee}}$ and the updated stack

$$stack'_\mu \equiv inc_{loc}\left(pop_{lifo}\left(c_\mu, npar^{\pi_\mu}_{stack}(f)\right)\right).stack$$

- Return from C-IL to MASM:
  - the active context is of C-IL type, i.e., $cil(c.ac)$ holds
  - the statement to be executed is function return

$$stmt_{\text{cil}} \in \{\textbf{return}\ e, \textbf{return}\}$$

– there is only one stack frame in the active C-IL context and the list of inactive context is not empty

$$nf_{ac}(c) = 1 \wedge c.ic \neq \varepsilon$$

– the return value can be represented as a bit-string

$$(stmt_{\text{cil}} = \textbf{return } e) \implies [\![e]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \textbf{val}(b, t)$$

Under these conditions the result of the step is defined as

$$\delta^{\pi,\theta}_{\text{MX}}(c, in) \overset{def}{\equiv} c'$$

$$\begin{aligned} c'.ac &= (c.ic[ni].stack, gpr') & \qquad c'.spr &= c.spr \\ c'.ic &= c.ic[1 : ni - 1] & c'.\mathcal{M} &= c.\mathcal{M} \end{aligned}$$

where $ni \equiv |c.ic|$ is the number of inactive contexts in the configuration c and the content gpr' of the general-purpose registers in the active MASM context is initialized in the following way:

$$gpr'(r) = \begin{cases} c.ic[ni].gpr_{callee}(r) & : \ r \in Reg_{callee} \\ 0^{32} & : \ r = zero \\ b & : \ r = rv \wedge (stmt_{\text{cil}} = \textbf{return } e) \\ in(r) & : \ \text{otherwise} \end{cases}$$

5.4 Concurrent Mixed Machine Semantics

Finally, we can define the concurrent mixed machine semantics in which the steps of $nt \in \mathbb{N}$ sequential mixed machines (or *MX threads*) with the shared memory are interleaved non-deterministically.

First, we collect the MX machine components characterizing individual MX threads.

Definition 5.65 (MX Thread Configuration). A configuration of an MX thread consists of the active context, a list of inactive contexts, and the special purpose registers

$$\begin{aligned} \mathbb{K}_{\text{MX}} \overset{def}{\equiv} \Big(& ac \in context_{\text{CIL}} \cup context^{active}_{\text{MASM}}, \\ & ic \in \left(context_{\text{CIL}} \cup context^{inactive}_{\text{MASM}}\right)^*, \\ & spr : \mathbb{B}^5 \to \mathbb{B}^{32} \Big) \end{aligned}$$

Definition 5.66 (Configuration of Concurrent MX Machine). Then configurations of the concurrent mixed machine containing $nt \in \mathbb{N}$ MX threads are defined by the set

$$\mathbb{C}_{c\text{MX}} \overset{def}{\equiv} \left(k : \mathbb{N}_{nt} \to \mathbb{K}_{\text{MX}},\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8\right)$$

where k is a mapping from the index to an MX thread configuration, and $\mathcal{M}$ is the shared memory.

Definition 5.67 (Sequential MX Machine Configuration from Concurrent MX). For a given MX thread configuration $k \in \mathbb{K}_{\text{MX}}$ and a memory configuration $\mathcal{M}$ we introduce the auxiliary function $\mathit{conf}_{\text{MX}}(k, \mathcal{M}) \in \mathbb{C}_{\text{MX}}$ combining both arguments into a sequential MX machine configuration

$$\mathit{conf}_{\text{MX}}(k, \mathcal{M}) \overset{def}{\equiv} (k.ac,\ k.ic,\ k.spr,\ \mathcal{M})$$

Then we overload the same function for a given concurrent MX configuration $c \in \mathbb{C}_{c\text{MX}}$ and $t \in \mathbb{N}_{nt}$ as

$$\mathit{conf}_{\text{MX}}(c, t) \overset{def}{\equiv} \mathit{conf}_{\text{MX}}(c.k(t), c.\mathcal{M})$$

Obviously, we call the configuration of the MX thread t *well-formed* if and only if its corresponding sequential MX machine configuration is well-formed, i.e., $\mathit{wfconf}^{\pi,\theta}_{\text{MX}}(\mathit{conf}_{\text{MX}}(c, t))$ holds.

Definition 5.68 (Concurrent MX Transition Function). Now, for a given MX program $\pi \in \mathit{Prog}_{\text{MX}}$ and the environment parameters $\theta \in \mathit{Params}_{\text{CIL}}$, the transitions of the concurrent mixed machine are defined by the function

$$\delta^{\pi,\theta}_{c\text{MX}} : \mathbb{C}_{c\text{MX}} \times \mathbb{N}_{nt} \times \Sigma_{\text{MX}} \to \mathbb{C}_{c\text{MX}\perp}$$

such that for a configuration $c \in \mathbb{C}_{c\text{MX}}$, an index $t \in \mathbb{N}_{nt}$ of an MX thread performing a step, and an input $in \in \Sigma_{\text{MX}}$, the result of the transition (that can also lead to the run-time error $\perp$) is defined as

$$\delta^{\pi,\theta}_{c\text{MX}}(c, t, in) \overset{def}{\equiv} \begin{cases} (c.k[t \mapsto k'_t],\ c'_t.\mathcal{M}) & : c'_t \neq \perp \\ \perp & : \text{otherwise} \end{cases}$$

where the next sequential MX configuration c'_t is computed as $c'_t \equiv \delta^{\pi,\theta}_{\text{MX}}(\mathit{conf}_{\text{MX}}(c, t), in)$ and the next MX thread configuration is $k'_t \equiv (c'_t.ac,\ c'_t.ic,\ c'_t.spr)$.

6 Compiler Correctness and Justification of Concurrent Mixed Model

This chapter is devoted to the detailed definition of compiler correctness for our sequential mixed machine with all necessary requirements in the concurrent context. Using this correctness statement we will consider the justification of the concurrent MX model with np MX threads whose compiled code runs on the multi-core MIPS-86 ISA model with np processors.

Originally, MX compiler correctness for a single core VAMP processor was described by Shadrin in his doctoral thesis [Sha12] based on the previous work [Lei08, LPP05] from the Verisoft project [Ver10]. Having developed the model for concurrent systems and simulation, Baumann [Bau14b] restated these correctness criteria separately for the C-IL and MASM implemented on MIPS-86 and showed how one can establish the concurrent simulation in both cases. However, the work was done under assumption that store buffer reduction similar to the one considered in [Kov13] was applied to the reference MIPS-86 machine, and no software conditions and policy needed for the reduction to go through (see Chapter 4) were given.

Using the results from [Sha12, Kov13, Bau14b], we continue the research on this topic for the overall MX semantics. Particularly, since we will use the full mathematical statement of the MX compiler correctness in order to introduce more powerful semantics for system programming in the scope of this thesis, we adapt the results from the previous work and resolve issues not matching the argumentation about the higher levels in our model stack. Moreover, we especially concentrate on software conditions and safety policy enabling both sequential and concurrent simulation between the abstract MX semantics and its implementation.

6.1 Compiler Correctness for Sequential Mixed Machine

6.1.1 Mixed Program Compilation and Stack Layout

The compilation of any mixed machine program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in \mathit{Prog}_{\text{MX}}$ is done separately by C-IL and MASM compilers. The compiled code is placed into the memory by a linker at corresponding code base addresses $cba_\mu, cba_{\text{cil}} \in \mathbb{B}^{32}$ which are considered in this work as system parameters or determined by the linker. Later we will introduce conditions on the placement of the code.

The bird eye's view on the stack implementation of the mixed machine in given in Figure 6.1. The allocated stack consists of the space occupied by alternating portions of C-IL and MASP stack regions and the free space to be used if the stack is not full. Let $mss \in \mathbb{N}$ denote the maximal stack size in bytes. Then, the stack grows down starting from the mss-th byte of the stack (left corner) pointed by the stack base address and occupies the memory till the topmost frame. The first byte (right corner) of this frame is addressed by the stack pointer residing in the

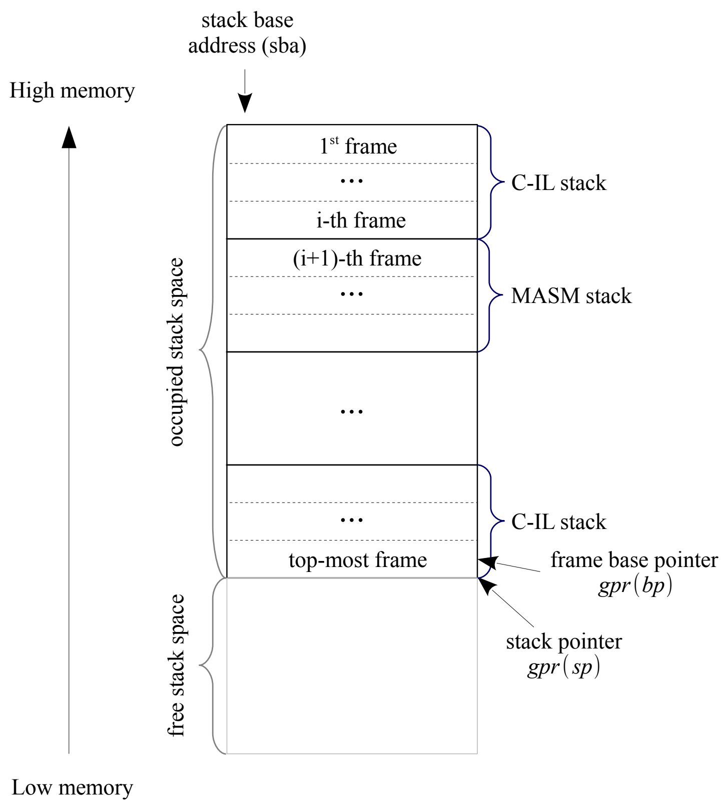

Figure 6.1: Overview of MX stack layout in memory.

GPRs. The frame base pointer in the corresponding register points to a special position inside the frame.

First, we consider the compilation of the C-IL and MASM programs, and the physical layout of stack frames for each case in detail.

6.1.1.1 Macro Assembly Compiler Information and Stack Frames

During the program compilation (or code generation) the MASM compiler produces static information.

Definition 6.1 (**Macro Assembly Compiler Information**). We define the set of tuples representing the MASM compiler information as

$$infoT_{\text{MASM}} \overset{def}{\equiv} (code \in \mathbb{I}^*_{\text{ASM}},\ \mathit{off} : \mathbb{P}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0)$$

where the components have the following meaning:

- *code* – a list of MIPS-86 assembly instructions representing a given compiled MASM program,
- *off* – a function calculating the offset in the compiled code for the first assembly instruction implementing a MASM instruction at a given location in a given procedure.

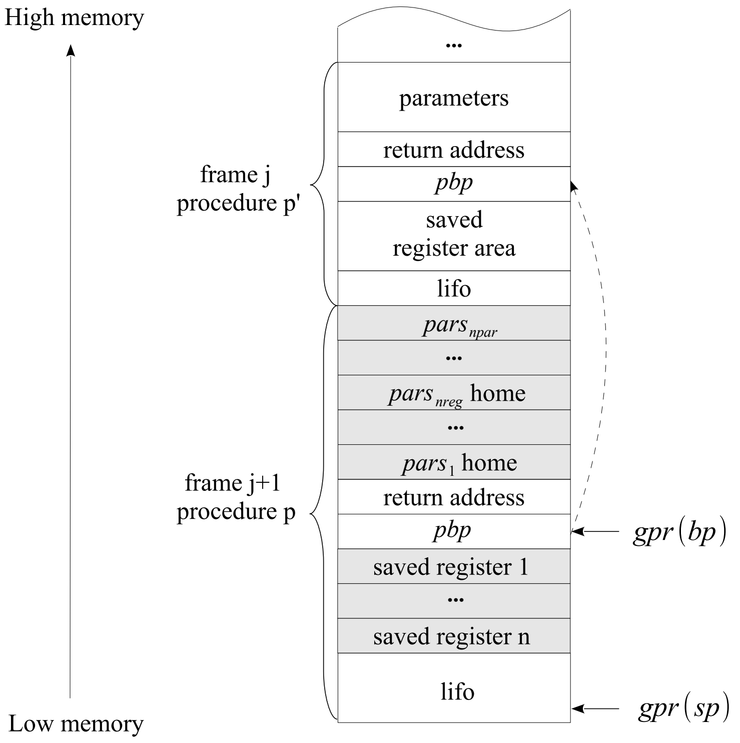

Figure 6.2: Layout of MASM stack frames. The shadowed parts represent in detail the space where the parameters and callee-saved registers are stored. The $i+1$-th frame is the topmost one here.

Definition 6.2 (**MASM Compilation Function**)**.** Through out the thesis we will use the uninterpreted function

$$cmpl_{\text{MASM}} : Prog_{\text{MASM}} \rightharpoonup infoT_{\text{MASM}}$$

generating the static compiler information for any given MASM program. The function is similar to the one considered in detail in [Sha12] and we do not provide its definition here.

Definition 6.3 (**Code Address of MASM Instruction/Macro**)**.** Given the compiler information $info_\mu \in infoT_{\text{MASM}}$ and the code base address cba_μ, one can easily compute the starting code address of a MASM instruction at the location $loc \in \mathbb{N}$ in a procedure $p \in \mathbb{P}_{name}$ as

$$ca_{\text{MASM}}(p, loc, info_\mu, cba_\mu) \overset{def}{\equiv} cba_\mu +_{32} (4 \cdot info_\mu.off(p, loc))_{32}$$

The layout of the MASM stack frames in the memory is depicted in Figure 6.2. Each frame contains the space for parameters, return code address from the procedure, callee-save registers, and the lifo. Any frame can be identified by its *base address* pointing to a word on the stack keeping the base address of the previous frame, or *previous base pointer pbp*. The base address of the topmost frame is kept in the GPR. The parameters are saved in the order defined by the compiler calling convention such that the home addresses for the first $nreg = npar^{\pi_\mu}_{regs}(p)$ parameters are reserved on the stack.

In the thesis we will particularly rely on the implementation of MASM instruction **ret** $\in \mathbb{S}_{\text{MASM}}$ given in [Sha12] and corresponding to the compiler calling convention for MIPS-86. The compiled code of this instruction called on the return from any procedure p performs the following steps:

1. restore callee-save registers present in $\pi_\mu(p).uses$,
2. adjust the stack pointer so that the stack is cleaned up from the parameters,
3. restore the base pointer from *pbp* saved in the frame,
4. return to the caller by jumping to the code at the return address stored in the frame.

Here, we do not present the formal definition of **ret** compilation and refer the interested reader to Section 3.2 of [Sha12] devoted to the topic of assembling MASM programs.

A memory word residing in the memory occupied by the stack is called a *stack item*. Obviously, addressing stack items is word-aligned.

Definition 6.4 (Stack Items in Memory). Given an ISA memory $m : \mathbb{B}^{32} \to \mathbb{B}^8$, an address $ad \in \mathbb{B}^{32}$ pointing to a stack item, and an index $i \in \mathbb{Z}$ of an another stack item above/below the given one (relative offset s.t. $i = 0$ for the item pointed by ad), we compute the value of the i-th item as

$$item(m, ad, i) \overset{def}{\equiv} m_4\left(bin_{32}\left(\langle ad\rangle + 4 \cdot i\right)\right)$$

Moreover, we can read a list of $n \in \mathbb{N}$ items starting at the address ad:

$$items_\uparrow(m, ad, n) \overset{def}{\equiv} \begin{cases} \varepsilon & : n = 0 \\ items_\uparrow(m, ad, n-1) \circ item(m, ad, n-1) & : \text{otherwise} \end{cases}$$

$$items_\downarrow(m, ad, n) \overset{def}{\equiv} \begin{cases} \varepsilon & : n = 0 \\ items_\downarrow(m, ad, n-1) \circ item(m, ad, -(n-1)) & : \text{otherwise} \end{cases}$$

Definition 6.5 (Computation of Common Stack Frame Components from Memory). Given an ISA memory $m : \mathbb{B}^{32} \to \mathbb{B}^8$ and a base address $ba \in \mathbb{B}^{32}$ of a corresponding frame in the memory, we retrieve the previous base pointer and the return address saved in the frame as

$$pbp(m, ba) \overset{def}{\equiv} item(m, ba, 0)$$
$$ra(m, ba) \overset{def}{\equiv} item(m, ba, 1)$$

Note that these items are common for MASM and C-IL frames (considered later).

Definition 6.6 (Computation of MASM Stack Frame Components from Memory). Analogously, other components of a MASM frame can be computed as follows:

- a list of $n \in \mathbb{N}_0$ input parameters

$$pars^{\pi_\mu}_{\text{MASM}}(m, ba, p) \overset{def}{\equiv} items_\uparrow\left(m, ba +_{32} 8_{32}, \pi_\mu(p).npar\right)$$

- saved registers (in order corresponding to the list)
  Using $ns \equiv |\pi_\mu(p).uses|$ we compute

$$saved^{\pi_\mu}_{\text{MASM}}(m, ba, p) : \mathbb{B}^5 \rightharpoonup \mathbb{B}^{32}$$

$$saved^{\pi_\mu}_{\text{MASM}}(m, ba, p)(r) \overset{def}{\equiv} \begin{cases} items_\downarrow\left(m, ba -_{32} 4_{32}, ns\right)[i] & : \langle r\rangle = \pi_\mu(p).uses[i] \wedge i \in \mathbb{N}_{ns} \\ undefined & : \text{otherwise} \end{cases}$$

- lifo

$$lifo^{\pi_\mu}_{\mathrm{MASM}}(m, ba, p, n) \overset{def}{\equiv} items_{\downarrow}(m, ba -_{32} ds_{32}, n)$$

where $ds \equiv (|\pi_\mu(p).uses| + 1) \cdot 4$.

Additionally to the retrieval of the stack components from the memory, we introduce a function evaluating one of the stack layout characteristics on the base of the abstract stack configuration.

Definition 6.7 (Distance between Frame Base Addresses in MASM Stack). Given a non-empty MASM stack $st \in \mathit{frame}^*_{\mathrm{MASM}}$, and a stack frame index $i \in \mathbb{N}_{top(st)}$, we define the function $dist_{\mathrm{MASM}}(st, i) \in \mathbb{N}$ computing the distance (in bytes) between the frame base addresses of the i-th and $i+1$-th frames for the non-topmost i-th frame, or the distance from the the base address of the topmost frame till the stack pointer as follows:

$$dist_{\mathrm{MASM}}(st, i) \overset{def}{\equiv} \begin{cases} 4 \cdot (\#\mathrm{dom}\,(saved_i(st)) + |lifo_i(st)|) & : i = top(s) \\ 4 \cdot \big(\#\mathrm{dom}\,(saved_i(st)) + |lifo_i(st)| + |pars_{i+1}(st)| + 2\big) & : i < top(s) \end{cases}$$

6.1.1.2 C-IL Compiler Information and Stack Frames

In comparison to MASM the C-IL stack frames in the memory (see Figure 6.3) have additionally the space for local variables (separate from the parameters), caller-save registers which are saved by the C-IL compiled code on a nested function call, return destination representing an address at which the return value must be saved on the function return, and a region for temporary values used by the compiler.

The compiled code of any function call (see Figure 6.4) consists of two parts: a *pre-call* executed before the jump to the callee's code, and an *epilogue* finishing the call after returning from the callee. The generated code of a function contains a starting portion of code (or *prologue*) preparing the frame before the function execution. The code is placed into the memory at the address determined for a given function by the C-IL environment parameters θ.

During the execution of the pre-call, the calling function saves the caller-save registers and return value destination on the stack, allocates the space for the callee's parameters and puts the arguments either on the stack or into registers according to the calling convention. Moreover, the return code address is saved before the jump to the callee is performed.

In turn, the callee's prologue is responsible for storing the frame base pointer of the caller, setting up the base pointer of the new frame in the GPR, allocation and initialization of local variables, as well as storing the callee-save registers on the stack.

The implementation of the return from the function is similar to the one considered for MASM above. After the return, the caller executes the epilogue storing the return value into the memory, restoring the content of caller-save registers, etc. The exact code generation depends on the compiler and we will not rely on it here. The only crucial fact we will use in the work is which items of the callee's frame must be prepared by the caller.

Now, similarly to the MASM compilation, we introduce the static compiler information for C-IL. However, since we assume an optimizing compiler, the information is more sophisticated.

Definition 6.8 (C-IL Compiler Information). We define the set of tuples representing the C-IL compiler information as

$$infoT_{\mathrm{CIL}} \overset{def}{\equiv} (code, \mathit{off}, \mathit{off}_{elog}, cp, \mathit{off}_{lvar}, reg_{lvar}, size_{crreg}, \mathit{off}_{crreg}, size_{tmp})$$

where the components have the following meaning:

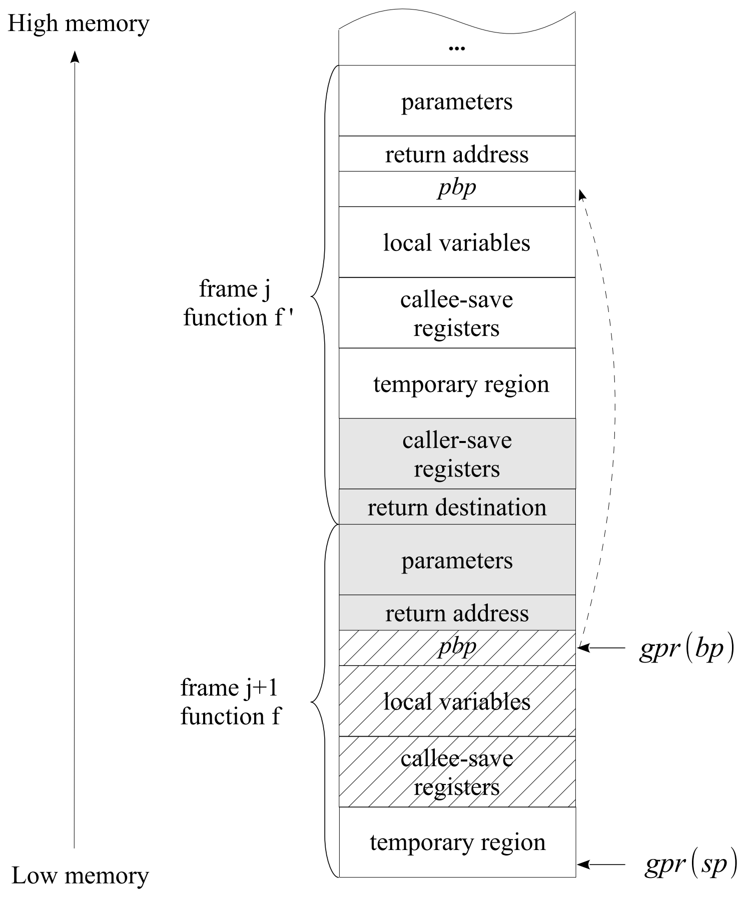

Figure 6.3: Layout of C-IL stack frames.

- $code \in \mathbb{I}^*_{\text{ASM}}$ – a list of MIPS-86 assembly instructions representing a given compiled C-IL program,

- $cp : \mathbb{F}_{name} \times \mathbb{N} \rightarrow \mathbb{B}$ – a flag indicating whether the C-IL machine is in the consistency point right before execution of a statement at a given location in a considered function,

- $\mathit{off} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ – the offset in the list $code$ for the first assembly instruction implementing a C-IL statement at a specified location (only a consistency point) in a given C-IL function,

- $\mathit{off}_{elog} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ – the offset in $code$ for the epilogue of a function call in a given function at a given location[1],

[1]The need of this piece of information not present in the original work [Sha12] was firstly discovered by the author of this thesis during the research on the correctness of the thread switch. Later, the corresponding component was added into $infoT_{\text{CIL}}$ by Christoph Baumann in his doctoral thesis [Bau14b]

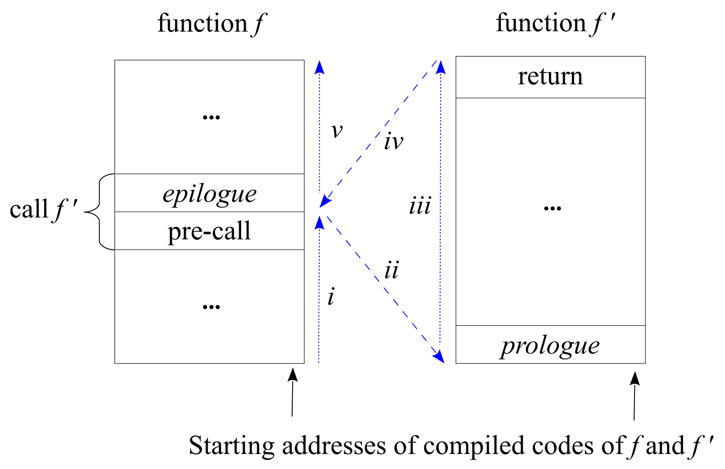

Figure 6.4: Execution of the compiled code during the function call and return. The numbers next to the arrows show the order of the execution.

- $\mathit{off}_{lvar} : \mathbb{V} \times \mathbb{F}_{name} \rightharpoonup \mathbb{N}$ – computes the offset (in bytes) of a local variable (excluding input parameters) on the stack relative to the frame base pointer for a given function.

  In contrast to this function in [Bau14b] additionally relying on being in a consistency point, we consider this offset to be static after the compilation because the allocation of the local variables is performed in the prologue part of the callee and cannot be changed during the function execution. Note that for simplicity we assume that the optimizing compiler allocates all local variables on the stack though their values may be kept in registers.[2]

- $\mathit{reg}_{lvar} : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{B}^5 \cup \{\bot\}$ – a function returning (if defined) a general purpose register in which the optimizing compiler keeps a value of a given local variable/parameter at the consistency point specified by a function name and a location in the body of the function. If the variable is not in a register, the function returns $\bot$.

  Note that we do not forbid the compiler to reallocate the registers for local variables (including parameters) during function calls. So, for a function call at a location loc in a function f and a local variable v we do not forbid $\mathit{reg}_{lvar}(v, f, loc) \neq \mathit{reg}_{lvar}(v, f, loc + 1)$. Naturally, the variables/parameters whose current values reside on the stack after the function call, must be stored to the the corresponding stack regions during this call. When a variable is supposed to be in a caller-save register after the function call, its value will be restored by the caller's epilogue from the caller-save region.

- $\mathit{size}_{crreg} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ – specifies the amount of bytes needed for the caller-save region on the stack in case of a function call in a specified function at a given location.

- $\mathit{off}_{crreg} : \mathit{Reg}_{caller} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ – the offset in bytes within the caller-save region (relative to its upper end) where a given register is supposed to be saved during the function call in a specified function at a given location.

[2]In fact this is similar to the parameter allocation performed according to the calling convention where the home addresses are always reserved on the stack (see Section 5.1.3).

- $size_{tmp} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ – the size (in bytes) of the temporary region on the stack before execution of a statement at a given location in a considered function. Here we rely on the fact that this size is known by static program analysis. Therefore, the stack must be used by the compiler in a restricted way.

- $\mathit{off}_{vol} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup 2^{\mathbb{N}_0}$ – the offsets (in the compiled code) for assembly instructions implementing memory volatile accesses in a C-IL statement at a specified location in a given C-IL function.

  Under a volatile access we understand an access of the memory/stack by a pointer value (computed during expression evaluation and belonging to $val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}$) to a volatile-qualified type. Since any C-IL statement can contain a few such accesses, the result of the function is a subset of offsets in the compiled code. We will consider volatile accesses later in the chapter and will use this compiler information under certain conditions.

  In [Bau14b] a similar function computing a single offset was not included into the compiler information, however was introduced as an uninterpreted one based on the compiler information and other parameters. Since the qualifier **volatile** of a type serves for the compiler in order to signal how an access to a memory may be implemented, in this work we consider this information directly available after the compilation. The idea of labeling processor or assembly instructions implementing memory accesses as volatile accesses comes from [CS10] and we use it in our work.

Definition 6.9 (C-IL Compilation Function). Similarly to MASM, we introduce the uninterpreted function

$$cmpl_{\mathrm{CIL}} : Prog_{\mathrm{CIL}} \rightharpoonup infoT_{\mathrm{CIL}}$$

generating the static compiler information for any given CIL program.

Finally, we can introduce an auxiliary function that will be used later in the statement of the compiler correctness.

Definition 6.10 (Code Address of C-IL Statement). Given the compiler information $info_{\mathrm{cil}} \in infoT_{\mathrm{CIL}}$ and the code base address cba_{cil}, one can easily compute the starting code address of a C-IL statement at the location $loc \in \mathbb{N}$ in a function $f \in \mathbb{F}_{name}$ as

$$ca_{\mathrm{CIL}}(f, loc, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \overset{def}{\equiv} cba_{\mathrm{cil}} +_{32} (4 \cdot info_{\mathrm{cil}}.\mathit{off}(f, loc))_{32}$$

Definition 6.11 (Return Address in C-IL Function Call). The return address saved on the stack during a function call is the starting address of the caller's epilogue and is computed as

$$rca_{\mathrm{CIL}}(f, loc, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \overset{def}{\equiv} cba_{\mathrm{cil}} +_{32} (4 \cdot info_{\mathrm{cil}}.\mathit{off}_{elog}(f, loc))_{32}$$

Definition 6.12 (Size of Local Variables and Parameters on C-IL Stack). For a given C-IL program $\pi_{\mathrm{cil}} \in Prog_{\mathrm{CIL}}$, the environment parameter $\theta \in Params_{\mathrm{CIL}}$, and a function $f \in \mathrm{dom}\,(\pi_{\mathrm{cil}}.\mathcal{F})$, the sizes (in bytes) of stack regions required for allocation of parameters and local variables of the function f are computed by $size^{\pi_{\mathrm{cil}},\theta}_{pars}(f) \in \mathbb{N}_0$ and $size^{\pi_{\mathrm{cil}},\theta}_{lvars}(f) \in \mathbb{N}_0$ respectively.

Formally, using the shorthands $(v_{f,j}, t_{f,j}) \equiv \pi_{\mathrm{cil}}.\mathcal{F}(f).V[j]$, $npar_f \equiv \pi_{\mathrm{cil}}.\mathcal{F}(f).npar$, and

$nlvar_f \equiv |\pi_{\text{cil}}.\mathcal{F}(f).V|$ we define

$$size^{\pi_{\text{cil}},\theta}_{pars}(f) \overset{def}{\equiv} \sum_{j=1}^{npar_f} size_\theta\left(qt2t(t_{f,j})\right)$$

$$size^{\pi_{\text{cil}},\theta}_{lvars}(f) \overset{def}{\equiv} \sum_{j=npar_f+1}^{nlvar_f} size_\theta\left(qt2t(t_{f,j})\right)$$

Definition 6.13 (**Distance between Frame Base Addresses in C-IL Stack**). Given a C-IL program $\pi_{\text{cil}} \in Prog_{\text{CIL}}$, the environment parameter $\theta \in Params_{\text{CIL}}$, the C-IL compiler information $info_{\text{cil}} \in infoT_{\text{CIL}}$, a non-empty C-IL stack $st \in frame^*_{\text{CIL}}$, and a stack frame index $i \in \mathbb{N}_{top(st)}$, by analogy with Definition 6.7 we provide the function

$$dist^{\pi_{\text{cil}},\theta}_{\text{CIL}}(st, i, info_{\text{cil}}) \overset{def}{\equiv} \begin{cases} dist' & : i = top(st) \\ dist' + dist'' & : i < top(st) \end{cases}$$

where the distances $dist'$, $dist''$ are computed as

$$dist' \equiv size^{\pi_{\text{cil}},\theta}_{lvars}\left(f_i(st)\right) + 4 \cdot \#Reg_{callee} + info_{\text{cil}}.size_{tmp}(f_i(st), loc_i(st))$$

$$dist'' \equiv info_{\text{cil}}.size_{crreg}\left(f_i(st), loc_i(st) - 1\right) + 4 + size^{\pi_{\text{cil}},\theta}_{pars}\left(f_{i+1}(st)\right) + 4 \cdot 2$$

Note that for the non-topmost frames i the size of the temporary region is computed *at the the location after the function call*, what, in turn, requires a consistency point to be at this location. Obviously, the size of this region during all callee executions must be equal to the one after the return. We do not require this size to be the same before and after the function call.

Moreover, similarly to [Bau14b] for simplicity we assume that C-IL functions are compiled in a way such that *in C-IL the callee's prologue stores all callee-save registers on the stack*. Otherwise, values of local variables for lower stack frames could be kept in registers during further function calls and saved by one of the callees in a much higher frame on the stack if the callee would modify the correspondent registers.

Definition 6.14 (**Local Variable Address on C-IL Stack**). Given a base address $ba \in \mathbb{B}^{32}$ for a frame of a C-IL function $f \in \mathbb{F}_{name}$ and the C-IL compiler information $info_{\text{cil}}$, we define the starting (or base) address of a local variable (excluding parameters) $v \in \mathbb{V}$ on the stack as

$$lva(v, ba, f, info_{\text{cil}}) \overset{def}{\equiv} ba -_{32} \left(info_{\text{cil}}.off_{lvar}(v, f)\right)_{32}$$

Definition 6.15 (**Function Parameter Address on C-IL Stack**). For $ba \in \mathbb{B}^{32}$, $f \in \mathbb{F}_{name}$, an index of the function parameter $j \in [1 : \pi_{\text{cil}}.\mathcal{F}(f).npar]$, and $(v_{f,j}, t_{f,j}) \equiv \pi_{\text{cil}}.\mathcal{F}(f).V[j]$ we define

$$para^{\pi_{\text{cil}},\theta}(j, ba, f) \overset{def}{\equiv} ba +_{32} \left(2 \cdot 4 + \sum_{k=1}^{j-1} size_\theta\left(qt2t(t_{f,k})\right)\right)_{32}$$

Definition 6.16 (**Address of a Caller-Save Register on C-IL Stack**). Let $loc \in \mathbb{N}$ be a location of a function call in a C-IL function $f \in \mathbb{F}_{name}$. Then the memory address at which a content of a caller-save register $r \in Reg_{caller}$ is saved on the stack during this function call is computed as

$$crra^{\pi_{\text{cil}},\theta}(r, ba, f, loc, info_{\text{cil}}) \overset{def}{\equiv} bin_{32}\left(crrba - info_{\text{cil}}.off_{crreg}(r, f, loc)\right)$$

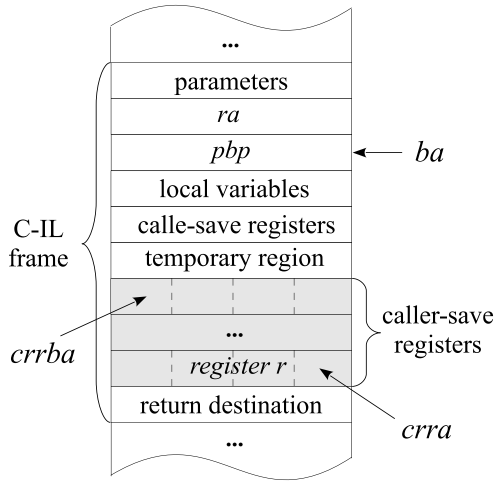

Figure 6.5: Address of a caller-save register on C-IL Stack.

where $crrba$ is the address (as a number) of the upper bound of the caller-save region (see Figure 6.5) such that

$$crrba \equiv \langle ba \rangle - size^{\pi_{\text{cil}},\theta}_{lvars}(f) - 4 \cdot \#Reg_{callee} - info_{\text{cil}}.size_{tmp}(f, loc+1) - 1$$

Note that this computation is used only for non-topmost frames. Moreover, as we mentioned before, the size of the temporary region is determined at the location after the function call.

Definition 6.17 (**Computation of C-IL Stack Frame Components from Memory).** Additionally, given an ISA memory $m : \mathbb{B}^{32} \to \mathbb{B}^8$, we can retrieve the following components from the C-IL stack in the memory:

- Destination address (binary word) for the return value in case of a call from C-IL.

  Using a base address ba' of a frame for a function/procedure f' as depicted on Figure 6.6, we get the destination address stored in the previous C-IL frame as

  $$rdsw^{\pi,\theta}(m, ba', f') \overset{def}{\equiv}$$
  $$\begin{cases} item\left(m, ba', 2 + size^{\pi_{\text{cil}},\theta}_{pars}(f')/4\right) & : f' \in \text{dom}\left(\mathcal{F}^{\theta}_{\pi_{\text{cil}}}\right) \wedge \neg ext(f', \pi_{\text{cil}}, \theta) \\ item\left(m, ba, 2 + \pi_{\mu}(f').npar\right) & : f' \in \text{dom}\left(\pi_{\mu}\right) \wedge \neg ext(f', \pi_{\mu}) \end{cases}$$

  where the number 2 corresponds to ra and pbp on the stack.

- Callee-saved registers stored in a C-IL frame:

  $$gpr_{callee}{}^{\pi_{\text{cil}},\theta}(m, ba, f) : Reg_{callee} \to \mathbb{B}^{32}$$

  such that for each $sv_i \in Reg_{callee}$

  $$gpr_{callee}{}^{\pi_{\text{cil}},\theta}(m, ba, f)(sv_i) \overset{def}{\equiv} items_{\downarrow}\left(m, ba -_{32} \left(size^{\pi_{\text{cil}},\theta}_{lvars}(f) + 4\right)_{32}, \#Reg_{callee}\right)[i]$$

Recall that for C-IL we require that all callee-save registers are saved on the stack, whereas for MASM – only if the MASM procedure is supposed to be called from C-IL.

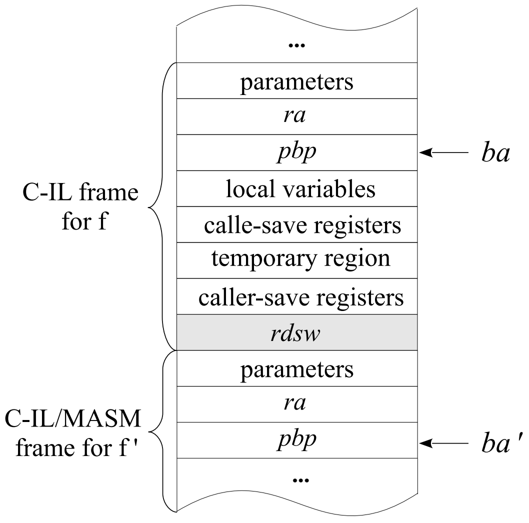

Figure 6.6: Destination address for the return value.

6.1.1.3 Mixed Compiler Information and Stack

Definition 6.18 (MX Compiler Information). Now, we combine the compile information for C-IL and MASM into a tuple

$$infoT_{\mathrm{MX}} \overset{def}{\equiv} infoT_{\mathrm{CIL}} \times infoT_{\mathrm{MASM}}$$

In this chapter we will often use $info = (info_{\mathrm{cil}}, info_{\mu}) \in infoT_{\mathrm{MX}}$.

Definition 6.19 (Code Region of MX Machine). The code region is determined by a set of byte addresses at which the compiled code resides in the memory. Given $cba = (cba_{\mathrm{cil}}, cba_{\mu}) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$ and $info$ we define

$$A^{code}_{\mathrm{CIL}}(info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \overset{def}{\equiv} \{cba_{\mathrm{cil}}\}_{4\cdot|info_{\mathrm{cil}}.code|}$$

$$A^{code}_{\mathrm{MASM}}(info_{\mu}, cba_{\mu}) \overset{def}{\equiv} \{cba_{\mu}\}_{4\cdot|info_{\mu}.code|}$$

$$A^{code}_{\mathrm{MX}}(info, cba) \overset{def}{\equiv} A^{code}_{\mathrm{CIL}}(info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \cup A^{code}_{\mathrm{MASM}}(info_{\mu}, cba_{\mu})$$

Obviously we require that both code regions do not overlap:

$$valid^{code}_{\mathrm{MX}}(info, cba) \overset{def}{\equiv} A^{code}_{\mathrm{CIL}}(info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \cap A^{code}_{\mathrm{MASM}}(info_{\mu}, cba_{\mu}) = \emptyset$$

Note that the code base address is a byte address of the first byte of the compiled code starting in the lower memory. In contrast, the stack base address is an highest byte address because the stack grows down from the higher to the lower memory.

Before we provide the computation of the distance between base pointers of the frames in the overall stack, we introduce auxiliary definitions and a unified notation.

Definition 6.20 (Full Stack of Mixed Machine). The full stack of the mixed machine (or MX stack) in a configuration $c \in \mathbb{C}_{\text{MX}}$ is a sequence of stacks of all contexts in c. In order to extract such a stack from the given MX configuration we use the following shorthands and definitions:

- the stack of a context $k \in \mathit{context}_{\text{CIL}} \cup \mathit{context}^{\mathit{active}}_{\text{MASM}} \cup \mathit{context}^{\mathit{inactive}}_{\text{MASM}}$

$$st(k) \overset{def}{\equiv} \begin{cases} k.stack & : masm(k) \\ k & : \text{otherwise} \end{cases}$$

- the full stack of an inactive context $k \in \left(\mathit{context}_{\text{CIL}} \cup \mathit{context}^{\mathit{inactive}}_{\text{MASM}}\right)^*$ is

$$st_{ic}(k) \in (\mathit{frame}_{\text{MASM}} \cup \mathit{frame}_{\text{CIL}})^*$$

$$st_{ic}(k) \overset{def}{\equiv} \begin{cases} \varepsilon & : k = \varepsilon \\ st_{ic}(k[1 : ni - 1]) \circ st(k[ni]) & : \text{otherwise} \end{cases}$$

  where $ni \equiv |k|$

- the stack of the MX configuration

$$st_{\text{MX}}(c) \overset{def}{\equiv} st_{ic}(c.ic) \circ st(c.ac)$$

To keep formal definitions in the similar shape throughout this work, we apply now the same overloaded shorthands for components of an MX stack $st \in (\mathit{frame}_{\text{MASM}} \cup \mathit{frame}_{\text{CIL}})^*$ as it was done for MASM and C-IL:

- the index of the topmost frame: $top(st) \equiv |st|$
- any components $X \in \{f, rds, loc, \mathcal{M}_{\mathcal{E}}, p, pars, saved, lifo\}$ of a stack frame with an index $i \in \mathbb{N}_{top(st)}$:

$$X_i(st) \equiv st[i].X$$

- the components of the topmost frame

$$X_{top}(st) \equiv X_{top(st)}(st)$$

- predicates checking whether a frame with an index i in the MX stack is of the C-IL or MASM type:

$$cil(st, i) \overset{def}{\equiv} (st[i] \in \mathit{frame}_{\text{CIL}})$$
$$masm(st, i) \overset{def}{\equiv} (st[i] \in \mathit{frame}_{\text{MASM}})$$

- the components $Y \in \{rettype, npar, V, P, body, uses\}$ of function/procedure table entry for a function/procedure in the i-th frame:

$$Y_i(st, \pi) \equiv \begin{cases} \pi_{\text{cil}}.\mathcal{F}(f_i(st)).Y & : cil(st, i) \\ \pi_{\mu}(p_i(st)).Y & : masm(st, i) \end{cases}$$

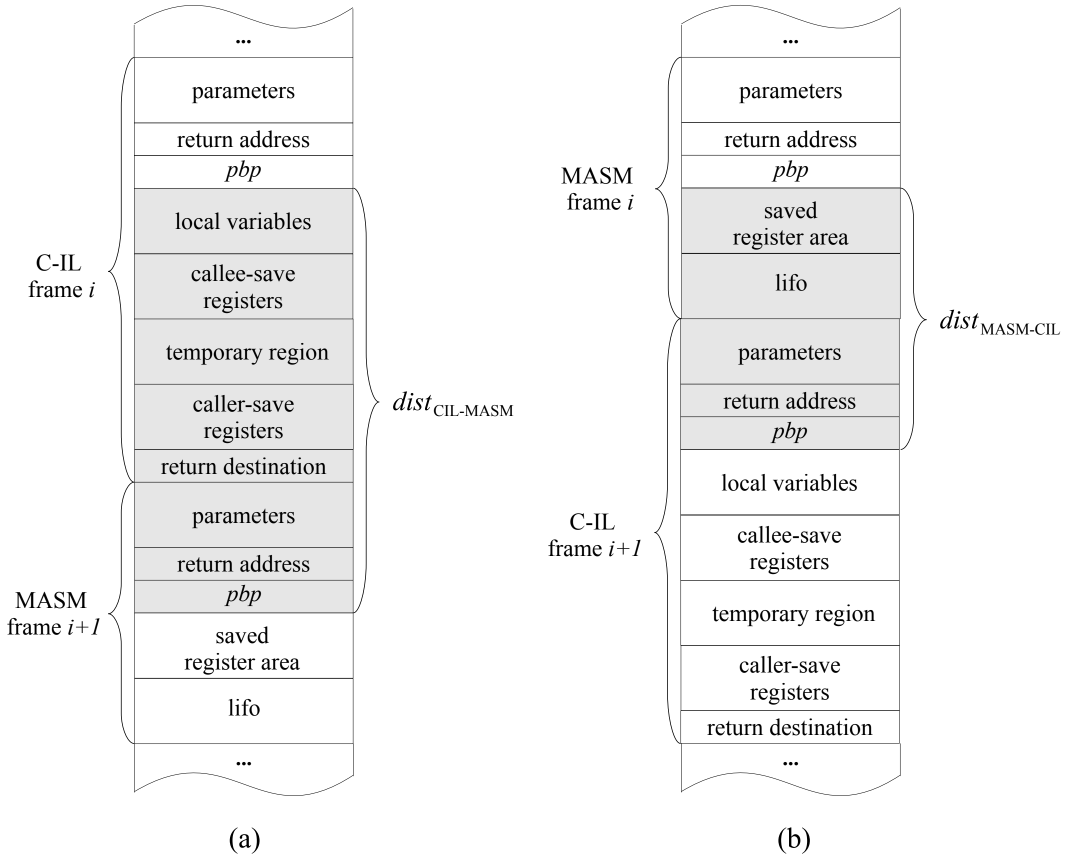

Figure 6.7: Computation of distances (a) $dist_{\text{CIL-MASM}}$ and (b) $dist_{\text{CIL}}^{\pi_{\text{cil}},\theta}$ in MX stack.

Definition 6.21 (Distance between Frame Base Addresses in MX Stack). Given an MX program $\pi \in Prog_{\text{MX}}$, $\pi = (\pi_\mu, \pi_{\text{cil}})$, the environment parameter $\theta \in Params_{\text{CIL}}$, the mixed machine compiler information $info \in infoT_{\text{MX}}$, $info = (info_{\text{cil}}, info_\mu)$, a non-empty MX stack $st \in (frame_{\text{MASM}} \cup frame_{\text{CIL}})^*$, and a stack frame index $i \in \mathbb{N}_{top(st)}$, we define the function

$$dist_{\text{MX}}^{\pi,\theta}(st, i, info) \overset{def}{\equiv}$$

$$\begin{cases} dist_{\text{CIL}}^{\pi_{\text{cil}},\theta}(st[i], 1, info_{\text{cil}}) & : \; i = top(st) \wedge cil(st, i) \\ dist_{\text{MASM}}(st[i], 1) & : \; i = top(st) \wedge masm(st, i) \\ dist_{\text{CIL}}^{\pi_{\text{cil}},\theta}(st[i : i+1], 1, info_{\text{cil}}) & : \; i < top(st) \wedge cil(st, i) \wedge cil(st, i+1) \\ dist_{\text{MASM}}(st[i : i+1], 1) & : \; i < top(st) \wedge masm(st, i) \wedge masm(st, i+1) \\ dist_{\text{CIL-MASM}} & : \; i < top(st) \wedge cil(st, i) \wedge masm(st, i+1) \\ dist_{\text{MASM-CIL}} & : \; i < top(st) \wedge masm(st, i) \wedge cil(st, i+1) \end{cases}$$

where the distances $dist_{\text{CIL-MASM}}$ and $dist_{\text{MASM-CIL}}$ between base addresses of two adjacent frames of different types are computed as depicted on Figure 6.7:

$$\begin{aligned}
dist_{\text{CIL-MASM}} \equiv\ & dist^{\pi_{\text{cil}},\theta}_{\text{CIL}}(st[i], 1, info_{\text{cil}}) + \\
& info_{\text{cil}}.size_{crreg}\,(f_i(st), loc_i(st) - 1) + \\
& 4 + 4 \cdot \big(npar_{i+1}(st, \pi) + 2\big) \\
dist_{\text{MASM-CIL}} \equiv\ & dist_{\text{MASM}}(st[i], 1) + size^{\pi_{\text{cil}},\theta}_{pars}\,(f_{i+1}(st)) + 4 \cdot 2
\end{aligned}$$

Finally, we can define the computation of base addresses of each frame in the mixed stack.

Definition 6.22 (Frame Base Addresses of MX Stack). Given the stack base address $sba \in \mathbb{B}^{32}$ and the arguments as in Definition 6.21, we compute the base address of the MX stack frame with an index $i \in \mathbb{N}_{top(st)}$ with the help of the following functions:

$$\widehat{base}^{\pi,\theta}_{\text{MX}}(st, i, sba, info) \overset{def}{\equiv} \begin{cases} \langle sba\rangle - size^{\pi_{\text{cil}},\theta}_{pars}\,(f_i(st)) - 4 \cdot 2 + 1 & : i = 1 \wedge cil(st, i) \\ \langle sba\rangle - 4 \cdot |pars_i(st)| - 4 \cdot 2 + 1 & : i = 1 \wedge masm(st, i) \\ \widehat{base}^{\pi,\theta}_{\text{MX}}(st, i-1, sba, info) - \\ dist^{\pi,\theta}_{\text{MX}}(st, i-1, info) & : i \in [2 : top(st)] \end{cases}$$

$$base^{\pi,\theta}_{\text{MX}}(st, i, sba, info) \overset{def}{\equiv} bin_{32}\left(\widehat{base}^{\pi,\theta}_{\text{MX}}(st, i, sba, info)\right)$$

where 1 is present in the computation for topmost frames because sba points to the last byte of the first word on the stack.

Note that we consider the stack base address also as a parameter of the system under which the compiled code is executed. Initially, the stack base address as well as the stack and base pointers are set during the booting and can be then changed when the software multi-threading is supported. Hence, we do not fix these parameters and state the compiler correctness for any of them. Later, we will show how these parameters are changed when we introduce the stack substitution.

Definition 6.23 (Stack Region of MX Machine). So, for given stack base address $sba \in \mathbb{B}^{32}$ and maximal stack size $mss \in \mathbb{N}$ we define the stack region as a set of byte addresses at which every byte of the stack resides in the memory.

First, we compute the maximal value of the stack pointer that does not cause the stack overflow:

$$sp_{max}(sba, mss) \overset{def}{\equiv} sba -_{32} (mss - 1)_{32}$$

Then, using the existing notation we define the stack region

$$A^{stack}_{\text{MX}}(sba, mss) \overset{def}{\equiv} \{sp_{max}(sba, mss)\}_{mss}$$

Moreover, from the stack st and the stack base address we can compute which address corresponds to the the item on the top of the stack. In fact, this address will be coupled with the stack pointer in GPRs later on.

$$sp^{\pi,\theta}_{top}(st, info, sba) \overset{def}{\equiv} base^{\pi,\theta}_{\text{MX}}(st, top(st), sba, info) -_{32} \left(dist^{\pi,\theta}_{\text{MX}}(st, top(st), info)\right)_{32}$$

Definition 6.24 (Stack Overflow). Obviously, the stack overflow can be easily indicated by the predicate

$$stackovf^{\pi,\theta}_{\text{MX}}(st, info, sba, mss) \overset{def}{\equiv} sp^{\pi,\theta}_{top}(st, info, sba) < sp_{max}(sba, mss)$$

6.1.2 Sequential Mixed Compiler Correctness

6.1.2.1 Compiler Consistency Relation

In order to guarantee that the SB reduced MIPS-86 machine implements the abstract MX semantics, we introduce the compiler consistency relation coupling configurations of both machines wrt. the static compiler information. Its definition is split into sub-relations for the program, stack, registers, etc. Obviously, we will require the compiler consistency to hold only at consistency points defined a bit later in this section.

In the definitions below we consider the following configurations and parameters: $c \in \mathbb{C}_{\mathrm{MX}}$, $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in \mathit{Prog}_{\mathrm{MX}}$, $\theta \in \mathit{Params}_{\mathrm{CIL}}$, $d \in \mathbb{C}_{\mathrm{MIPS}}$, $\mathit{info} = (\mathit{info}_{\mathrm{cil}}, \mathit{info}_\mu) \in \mathit{infoT}_{\mathrm{MX}}$, $\mathit{sba} \in \mathbb{B}^{32}$, $\mathit{mss} \in \mathbb{N}$, $\mathit{cba} = (\mathit{cba}_{\mathrm{cil}}, \mathit{cba}_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$. Moreover, when the we are not interested in the full MIPS-86 configuration, we will directly use its memory component m.

Definition 6.25 (**MX Code Consistency**). The MX compiler consistency includes the code consistency for the MASM and C-IL machines and for each of them requires that (i) the assembly code in the corresponding compiler information is obtained via the compilation of the given program, and (ii) each assembled instruction of this code has its binary representation residing in the MIPS-86 memory.

Formally, we define the relations for π_μ and π_{cil} separately as

$$\begin{aligned} &\mathit{consis}^{\mathit{code}}_{\mathrm{MASM}}(m, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \overset{\mathit{def}}{\equiv} \\ &\quad \text{(i)} \quad \mathit{info}_\mu = \mathit{cmpl}_{\mathrm{MASM}}(\pi_\mu) \\ &\quad \text{(ii)} \quad \forall j \in \mathbb{N}_{|\mathit{info}_\mu.\mathit{code}|}.\ \mathit{code}_{\mathrm{ASM}}\left(\mathit{info}_\mu.\mathit{code}[j]\right) = m_4(\mathit{ad}^j_\mu) \end{aligned}$$

with $\mathit{ad}^j_\mu \equiv \mathit{cba}_\mu +_{32} (4 \cdot (j-1))_{32}$

$$\begin{aligned} &\mathit{consis}^{\mathit{code}}_{\mathrm{CIL}}(m, \pi_{\mathrm{cil}}, \mathit{info}_{\mathrm{cil}}, \mathit{cba}_{\mathrm{cil}}) \overset{\mathit{def}}{\equiv} \\ &\quad \text{(i)} \quad \mathit{info}_{\mathrm{cil}} = \mathit{cmpl}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}) \\ &\quad \text{(ii)} \quad \forall j \in \mathbb{N}_{|\mathit{info}_{\mathrm{cil}}.\mathit{code}|}.\ \mathit{code}_{\mathrm{ASM}}\left(\mathit{info}_{\mathrm{cil}}.\mathit{code}[j]\right) = m_4(\mathit{ad}^j_{\mathrm{cil}}) \end{aligned}$$

with $\mathit{ad}^j_{\mathrm{cil}} \equiv \mathit{cba}_{\mathrm{cil}} +_{32} (4 \cdot (j-1))_{32}$

The full MX code consistency combines them into

$$\begin{aligned} &\mathit{consis}^{\mathit{code}}_{\mathrm{MX}}(m, \pi, \mathit{info}, \mathit{cba}) \overset{\mathit{def}}{\equiv} \\ &\quad \text{(i)} \quad \mathit{consis}^{\mathit{code}}_{\mathrm{MASM}}(m, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \\ &\quad \text{(ii)} \quad \mathit{consis}^{\mathit{code}}_{\mathrm{CIL}}(m, \pi_{\mathrm{cil}}, \mathit{info}_{\mathrm{cil}}, \mathit{cba}_{\mathrm{cil}}) \end{aligned}$$

Moreover, for the given full MIPS-86 configuration we overload the definition:

$$\mathit{consis}^{\mathit{code}}_{\mathrm{MX}}(d, \pi, \mathit{info}, \mathit{cba}) \overset{\mathit{def}}{\equiv} \mathit{consis}^{\mathit{code}}_{\mathrm{MX}}(d.m, \pi, \mathit{info}, \mathit{cba})$$

Definition 6.26 (**MX Memory Consistency**). The MX memory consistency states that the memory content of both machines is equal except the code and stack regions, as well as some addresses $\mathit{icm} \subset A_{\mathit{hyp}}$ at which the memory consistency might not hold because of environment steps.

$$\begin{aligned} &\mathit{consis}^{\mathit{mem}}_{\mathrm{MX}}(c, d, \mathit{info}, \mathit{cba}, \mathit{sba}, \mathit{mss}, \mathit{icm}) \overset{\mathit{def}}{\equiv} \\ &\quad \forall a \in \mathbb{B}^{32} \setminus \left(A^{\mathit{code}}_{\mathrm{MX}}(\mathit{info}, \mathit{cba}) \cup A^{\mathit{stack}}_{\mathrm{MX}}(\mathit{sba}, \mathit{mss}) \cup \mathit{icm}\right).\ d.m(a) = c.\mathcal{M}(a) \end{aligned}$$

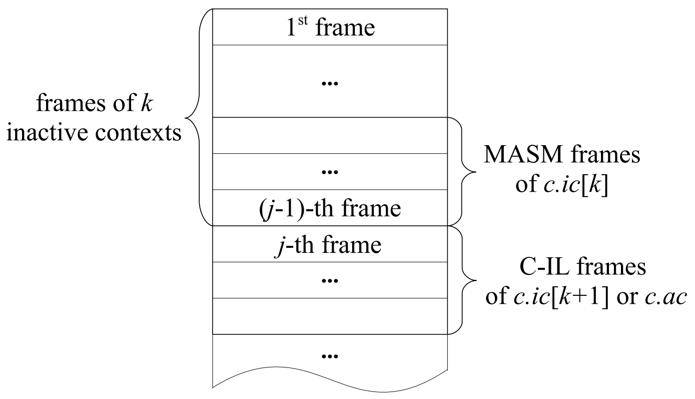

Figure 6.8: The position of the stack frame j containing the values of $c.ic[k].gpr_{callee}$.

Now, using the shorthands $st \equiv st_{\mathrm{MX}}(c)$ and $ba_q \equiv base^{\pi,\theta}_{\mathrm{MX}}(st, q, sba, info)$ for any $q \in \mathbb{N}_{top(st)}$ we define the sub-relation based on the stack configuration.

Definition 6.27 (**Frame Base and Stack Pointer Consistency**). By the frame base and stack pointer consistency we require that the value of the frame base pointer in the GPR is equal to the base address of the topmost frame of the MX stack. The stack pointer, in turn, must point to the item on the top of the stack.

$$consis^{bp/sp}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \overset{def}{\equiv}$$
$$\begin{aligned} &(i) && d.cpu.core.gpr(bp) = ba_{top(st)} \\ &(ii) && d.cpu.core.gpr(sp) = sp^{\pi,\theta}_{top}(st, info, sba) \end{aligned}$$

The content of all other registers except the program counter is covered by the registers consistency.

Definition 6.28 (**MX Register Consistency**). The register consistency demands that (i) GPRs of active MASM context are coupled with the corresponding registers of the core configuration, (ii) the content of all SPRs in both machines are always equal, and (iii) the callee-save registers of inactive MASM contexts in the MX machine configuration are stored in the MIPS-86 machine memory on the stack corresponding to the next context.[3]

$$consis^{reg}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \overset{def}{\equiv}$$
$$\begin{aligned} &(i) && masm(c.ac) \implies \forall r \in \mathbb{B}^5 \setminus \{sp, bp, ra\}.\ d.cpu.core.gpr(r) = c.ac.gpr(r) \\ &(ii) && d.cpu.core.spr = c.spr \\ &(iii) && \forall k \in \mathbb{N}_{|c.ic|}.\ masm(c.ic[k]) \implies c.ic[k].gpr_{callee} = gpr_{callee}{}^{\pi_{\mathrm{cil}},\theta}(d.m, ba_j, f_j(st)) \end{aligned}$$

where j is the index of the first stack frame after the context $c.ic[k]$ on the stack st such that $j \equiv 1 + \sum_{i=1}^{k} nf_{cntx}\,(c.ic[i])$ (see Figure 6.8).

Definition 6.29 (**MX Control Consistency**). The control consistency demands that (i) the program counter of the processor core points to the code address of the current C-IL statement or

[3]This part of the MX consistency was missing in the original work [Sha12].

MASM instruction depending on the MX context, and (ii) each code return addresses saved on the stack is either the code addresse of the next MASM statement after the procedure/function call, or the starting address of the epilogue in the C-IL function call.

$$consis^{ctrl}_{\mathrm{MX}}(c, d, \pi, \theta, info, cba, sba) \overset{def}{\equiv}$$

$$\textit{(i)} \quad d.cpu.core.pc = \begin{cases} ca_{\mathrm{MASM}}\left(p_{top}(st), loc_{top}(st), info_\mu, cba_\mu\right) & : masm(st, top(st)) \\ ca_{\mathrm{CIL}}\left(f_{top}(st), loc_{top}(st), info_{\mathrm{cil}}, cba_{\mathrm{cil}}\right) & : cil(st, top(st)) \end{cases}$$

$$\textit{(ii)} \quad \forall i \in [2 : top(st)].$$

$$ra(m, ba_i) = \begin{cases} ca_{\mathrm{MASM}}\left(p_{i-1}(st), loc_{i-1}(st), info_\mu, cba_\mu\right) & : masm(st, i) \\ rca_{\mathrm{CIL}}(f_{i-1}(st), loc_{i-1}(st) - 1, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) & : cil(st, i) \end{cases}$$

The rest of the abstract MX stack configuration is covered by the stack compiler consistency guaranteeing that the stack is properly implemented in the MIPS-86 memory.

Definition 6.30 (**Stack Consistency**)**.** Before we define the stack consistency, we introduce the consistency relations for the return destination and local memory of a single frame.

For a non-topmost frame i the *return destination consistency* requires that the return value destination address on the stack is either a global memory address, or an address of a corresponding local variable / parameter.[4]

$$consis^{rds_i}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \overset{def}{\equiv} rds_i(st) \neq \bot \implies$$

$$rdsw^{\pi,\theta}(d.m, ba_{i+1}, F_{i+1}) =$$

$$\begin{cases} a & : rds_i(st) = \mathbf{val}(a, t) \\ lva(v, ba_i, f_i(st), info_{\mathrm{cil}}) +_{32} o_{32} & : rds_i(st) = \mathbf{lref}((v, o), i, t') \wedge \\ & \quad (v, t'') = V_i(st, \pi)[j] \wedge j > npar_i(st, \pi) \\ para^{\pi_{\mathrm{cil}},\theta}(j, ba_i, f_i(st)) +_{32} o_{32} & : rds_i(st) = \mathbf{lref}((v, o), i, t') \\ & \quad (v, t'') = V_i(st, \pi)[j] \wedge j \leq npar_i(st, \pi) \end{cases}$$

where

$$F_{i+1} \equiv \begin{cases} p_{i+1}(st) & : masm(st, i+1) \\ f_{i+1}(st) & : cil(st, i+1) \end{cases}$$

Note that we do not make restrictions on t' and t'' because the configuration c is well-formed. In the simplest case one has $t' = \mathbf{ptr}(t'')$.

The *consistency relation for the local memory* of a frame i shows where the values of its local variables and parameters reside in the MIPS-86 configuration. Particularly, a local variable or a parameter of a topmost frame can be either in a processor register or on the physical stack where it is originally allocated. As for non-topmost frames, if the value is supposed to be in the register after the return from a callee, it must be kept in a corresponding region for callee- or caller-save registers on the stack during the callee's execution. Otherwise, as in the previous case, it resides on the stack at the allocated address.

Let the following shorthands denote

- a local variable/parameter declared in the body of the function $f_i(st)$

$$(v_{i,j}, t_{i,j}) \equiv V_i(st, \pi)[j]$$

[4]The fact that the return value can be stored into a variable used as a parameters in the function was not taken into account in [Bau14b].

- a register (if defined) containing its value either at the moment (for the topmost frame) or after the return from the callee $f_{i+1}(st)$ (for non-topmost frames)

$$r_{i,j} \equiv \mathit{info}_{\text{cil}}.\mathit{reg}_{lvar}\,(v_{i,j}, f_i(st), \mathit{loc}_i(st))$$

- a memory address at which during a function/procedure call the content of $r_{i,j}$ is saved on the stack if the register is a caller-save one

$$\mathit{crra}_{i,j} \equiv \mathit{crra}^{\pi_{\text{cil}},\theta}\,(r_{i,j}, \mathit{ba}_i, f_i(st), \mathit{loc}_i(st) - 1, \mathit{info}_{\text{cil}})$$

Then, this consistency relation is defined as

$$\begin{aligned}
&\mathit{consis}^{lm_i}_{\text{MX}}(c, d, \pi, \theta, \mathit{info}, \mathit{sba}) \overset{def}{\equiv} \forall j \in \mathbb{N}_{|V_i(st,\pi)|}.\\
&\mathcal{M}_{\mathcal{E}_i}(st)\,(v_{i,j}) =\\
&\begin{cases}
d.cpu.core.gpr(r_{i,j}) & : i = \mathit{top}(st) \wedge r_{i,j} \neq \bot\\
\mathit{gpr}_{callee}{}^{\pi_{\text{cil}},\theta}\,(d.m, \mathit{ba}_{i+1}, f_{i+1}(st))\,(r_{i,j}) & : i < \mathit{top}(st) \wedge r_{i,j} \in \mathit{Reg}_{callee} \wedge\\
& \quad \mathit{cil}(st, i+1)\\
\mathit{saved}^{\pi_\mu}_{\text{MASM}}\,(d.m, \mathit{ba}_{i+1}, p_{i+1}(st))\,(r_{i,j}) & : i < \mathit{top}(st) \wedge r_{i,j} \in \mathit{Reg}_{callee} \wedge\\
& \quad \mathit{masm}(st, i+1)\\
d.m_4\,(\mathit{crra}_{i,j}) & : i < \mathit{top}(st) \wedge r_{i,j} \in \mathit{Reg}_{caller}\\
d.m_{size_\theta(qt2t(t_{i,j}))}\,\left(\mathit{para}^{\pi_{\text{cil}},\theta}(j, \mathit{ba}_i, f_i(st))\right) & : r_{i,j} = \bot \wedge j \leq \mathit{npar}_i(st, \pi)\\
d.m_{size_\theta(qt2t(t_{i,j}))}\,(\mathit{lva}(v_{i,j}, \mathit{ba}_i, f_i(st), \mathit{info}_{\text{cil}})) & : r_{i,j} = \bot \wedge j > \mathit{npar}_i(st, \pi)
\end{cases}
\end{aligned}$$

Parameters passed through the registers must be only single data words. Moreover, for simplicity we assume that variables and parameters are not supposed to reside in the register $zero$ after the function/procedure return.

Finally, we can state the stack consistency demanding the following:

$$\begin{aligned}
&\mathit{consis}^{stack}_{\text{MX}}(c, d, \pi, \theta, \mathit{info}, \mathit{sba}) \overset{def}{\equiv} \forall i \in \mathbb{N}_{top(st)}.\\
&\quad \textit{(i)} \quad i > 1 \implies \mathit{pbp}(d.m, \mathit{ba}_i) = \mathit{ba}_{i-1}\\
&\quad \textit{(ii)} \quad \mathit{masm}(st, i) \implies\\
&\qquad\quad \textit{(a)} \quad \mathit{pars}_i(st) = \mathit{pars}^{\pi_\mu}_{\text{MASM}}\,(d.m, \mathit{ba}_i, p_i(st))\\
&\qquad\quad \textit{(b)} \quad \mathit{saved}_i(st) = \mathit{saved}^{\pi_\mu}_{\text{MASM}}\,(d.m, \mathit{ba}_i, p_i(st))\\
&\qquad\quad \textit{(c)} \quad \mathit{lifo}_i(st) = \mathit{lifo}^{\pi_\mu}_{\text{MASM}}\,(d.m, \mathit{ba}_i, p_i(st), |\mathit{lifo}_i(st)|)\\
&\quad \textit{(iii)} \quad \mathit{cil}(st, i) \implies\\
&\qquad\quad \textit{(a)} \quad i < \mathit{top}(st) \implies \mathit{consis}^{rds_i}_{\text{MX}}(c, d, \pi, \theta, \mathit{info}, \mathit{sba})\\
&\qquad\quad \textit{(b)} \quad \mathit{consis}^{lm_i}_{\text{MX}}(c, d, \pi, \theta, \mathit{info}, \mathit{sba})
\end{aligned}$$

Recall that we already stated the requirement that the prologue of any C-IL function stores all callee-save registers on the stack.

Definition 6.31 (Software Condition on MASM Procedures). As follows from the definition of the consistency relation for the local variables, we also assume that any MASM procedure called from a C-IL function is supposed to store the callee-save registers in the same manner. We formulate this requirement explicitly as a software condition:

$$\begin{aligned}
\mathit{sc}^{prog}_{\text{MX}}(\pi, \theta) \overset{def}{\equiv} \forall p \in \operatorname{dom}(\pi_\mu)\,.\,\neg ext(p, \pi_\mu) \wedge p \in \operatorname{dom}\left(\mathcal{F}^{\theta}_{\pi_{\text{cil}}}\right) \wedge ext(p, \pi_{\text{cil}}, \theta) \implies\\
\forall r \in \mathit{Reg}_{callee}.\ \langle r \rangle \in \pi_\mu(p).uses
\end{aligned}$$

Definition 6.32 (**MX Compiler Consistency Relation**)**.** The overall compiler consistency for the mixed machine is then a conjunction of all sub-relations defined above:

$$
\begin{aligned}
consis_{\mathrm{MX}}(c, d, \pi, \theta, info, cba, sba, mss, icm) \overset{def}{\equiv}& \\
\textit{(i)} \quad & consis^{code}_{\mathrm{MX}}(d, \pi, info, cba) \\
\textit{(ii)} \quad & consis^{ctrl}_{\mathrm{MX}}(c, d, \pi, \theta, info, cba, sba) \\
\textit{(iii)} \quad & consis^{bp/sp}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \\
\textit{(iv)} \quad & consis^{reg}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \\
\textit{(v)} \quad & consis^{stack}_{\mathrm{MX}}(c, d, \pi, \theta, info, sba) \\
\textit{(vi)} \quad & consis^{mem}_{\mathrm{MX}}(c, d, info, cba, sba, mss, icm)
\end{aligned}
$$

6.1.2.2 Consistency Points

Definition 6.33 (**Consistency Points for MX Machine**)**.** The mixed machine is at a consistency point if it is going to execute either a MASM instruction or a C-IL statement at a location determined by the C-IL compiler as the consistency point. Given an active context $ac \in context_{\mathrm{CIL}} \cup context^{active}_{\mathrm{MASM}}$ and the compiler information $info \in infoT_{\mathrm{MX}}$, we formally define

$$
\begin{aligned}
cp_{\mathrm{MX}}(ac, info) \overset{def}{\equiv}\ & masm(ac) \vee \\
& cil(ac) \wedge info_{\mathrm{cil}}.cp\left(f_{top}(ac), loc_{top}(ac)\right)
\end{aligned}
$$

Note that since the C-IL active context is represented by the C-IL stack, we directly use ac instead of $st(ac)$. We also reload the predicate for a given mixed machine configuration $c \in \mathbb{C}_{\mathrm{MX}}$ as

$$cp_{\mathrm{MX}}(c, info) \equiv cp_{\mathrm{MX}}(c.ac, info)$$

From the discussion about the C-IL compiler information and the MX consistency relation we already know that we have to require that the C-IL compiler guarantees consistency points at locations before and after function/procedure calls. This requirement is important for the MX compiler correctness and can be extended depending on the context in which our MX semantics is used. So, in order to be able to argue about the correctness of the kernel thread switch in the scope of this work, we will also require that the consistency points are present at the first statement of C-IL functions. This condition is weaker in comparison to [Bau14b] where the author additionally demanded them at the return statement.

Definition 6.34 (**Requirement on Consistency Points in C-IL Code**)**.** The C-IL compiler should at least guarantee consistency points before and after function/procedure calls as well as at the first statements in function bodies. Let $body_f \equiv \mathcal{F}^{\theta}_{\pi_{\mathrm{cil}}}(f).P$ and $stmt_{f,loc} \equiv body_f[loc]$ then

$$
\begin{aligned}
valid^{cp}_{\mathrm{MX}}(\pi, info, \theta) \overset{def}{\equiv}\ & \forall f \in \mathrm{dom}\left(\mathcal{F}^{\theta}_{\pi_{\mathrm{cil}}}\right), loc \in [1 : |body_f|]. \\
& \neg ext(f, \pi_{\mathrm{cil}}, \theta) \implies \\
& \quad \textit{(i)} \quad stmt_{f,loc} \in \{e_0 = \textbf{call}\ e(E), \textbf{call}\ e(E)\} \implies \\
& \quad \qquad info_{\mathrm{cil}}.cp(f, loc) \wedge info_{\mathrm{cil}}.cp(f, loc + 1) \\
& \quad \textit{(ii)} \quad info_{\mathrm{cil}}.cp(f, 1)
\end{aligned}
$$

A set of ISA instruction addresses of the first instructions in the compiled code of C-IL statements at consistency points is computed in the following way:

$$A^{cp}_{\text{CIL}}(\pi_{\text{cil}}, \mathit{info}_{\text{cil}}, \theta, \mathit{cba}_{\text{cil}}) \overset{def}{\equiv} \left\{ \mathit{ca}_{\text{CIL}}(f, \mathit{loc}, \mathit{info}_{\text{cil}}, \theta, \mathit{cba}_{\text{cil}}) \;\middle|\; \begin{array}{l} f \in \text{dom}\left(\mathcal{F}^{\theta}_{\pi_{\text{cil}}}\right) \wedge \\ \neg \mathit{ext}(f, \pi_{\text{cil}}, \theta) \wedge \\ \mathit{loc} \in [1 : |\mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).P|] \wedge \\ \mathit{info}_{\text{cil}}.\mathit{cp}(f, \mathit{loc}) \end{array} \right\}$$

Analogously, such a set of addresses for a MASM program is defined as

$$A^{cp}_{\text{MASM}}(\pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \overset{def}{\equiv} \left\{ \mathit{ca}_{\text{MASM}}(p, \mathit{loc}, \mathit{info}_\mu, \mathit{cba}_\mu) \;\middle|\; \begin{array}{l} p \in \text{dom}(\pi_\mu) \wedge \\ \neg \mathit{ext}(p, \pi_\mu) \wedge \\ \mathit{loc} \in [1 : |\pi_\mu(p).\mathit{body}|] \end{array} \right\}$$

Then, for an MX program π, $\mathit{cba} = (\mathit{cba}_{\text{cil}}, \mathit{cba}_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, and the compiler information $\mathit{info} = (\mathit{info}_{\text{cil}}, \mathit{info}_\mu) \in \mathit{infoT}_{\text{MX}}$ we combine both sets into

$$A^{cp}_{\text{MX}}(\pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv} A^{cp}_{\text{MASM}}(\pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \cup A^{cp}_{\text{CIL}}(\pi_{\text{cil}}, \mathit{info}_{\text{cil}}, \theta, \mathit{cba}_{\text{cil}})$$

Definition 6.35 (Consistency Points for MIPS-86 ISA). Therefore, the SB reduced MIPS-86 machine with a single core $\mathit{core} \in \mathbb{C}_{\mathit{core}}$ is at the MX consistency point if the following predicate holds:

$$\mathit{cp}^{\text{MX}}_{r\text{MIPS}}(\mathit{core}, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv} \mathit{core}.\mathit{pc} \in A^{cp}_{\text{MX}}(\pi, \mathit{info}, \theta, \mathit{cba})$$

We also reuse the definition for a given configuration $d \in \mathbb{C}_{\text{MIPS}}$

$$\mathit{cp}^{\text{MX}}_{r\text{MIPS}}(d, \pi, \mathit{info}, \theta, \mathit{cba}) \equiv \mathit{cp}^{\text{MX}}_{r\text{MIPS}}(d.\mathit{cpu}.\mathit{core}, \pi, \mathit{info}, \theta, \mathit{cba})$$

6.1.2.3 Accessed Addresses

Sets of Addresses for MASM In Section 4.3.1 we have already defined the sets of accessed addresses for the MIPS-86 model. However, since they rely on the MIPS-86 processor input, here we will not use them directly. Instead, we will reproduce only the needed parts of them, what allows to make the definitions clear and show the exact computations.

Definition 6.36 (Memory Addresses Accessed for Reading and Writing by MASM Instruction). The set of memory byte addresses at which the memory is read during a MASM instruction execution is

$$\mathit{reads}_{\text{MASM}}(c_\mu, \pi_\mu) \overset{def}{\equiv} \begin{cases} \{\mathit{ea}(\mathit{core}, I_\mu)\}_4 & : \mathit{instr}_{\mathit{next}}(c_\mu, \pi_\mu) \in \mathbb{I}^{\text{MASM}}_{\text{ASM}} \wedge \mathit{load}(I_\mu) \\ \emptyset & : \text{otherwise} \end{cases}$$

where $\mathit{core} \equiv \mathit{core}_{\text{MIPS}}(c_\mu)$ is the processor core constructed from the MASM configuration and $I_\mu \in \mathbb{B}^{32}$ is an assembled instruction such that $I_\mu \equiv I(c_\mu, \pi_\mu)$. Analogously, for the set of addresses at which the memory is written we define

$$\mathit{swap}_{\text{MASM}}(c_\mu, \pi_\mu) \overset{def}{\equiv} \mathit{cas}(I_\mu) \wedge c.\mathit{gpr}(\mathit{rd}(I_\mu)) = c.\mathcal{M}_4(\mathit{ea}(\mathit{core}, I_\mu))$$

$$\mathit{wr}_{\text{MASM}}(c_\mu, \pi_\mu) \overset{def}{\equiv} \mathit{sw}(I_\mu) \vee \mathit{locksw}(I_\mu) \vee \mathit{swap}_{\text{MASM}}(c_\mu, \pi_\mu)$$

$$\mathit{writes}_{\text{MASM}}(c_\mu, \pi_\mu) \overset{def}{\equiv} \begin{cases} \{\mathit{ea}(\mathit{core}, I_\mu)\}_4 & : \mathit{instr}_{\mathit{next}}(c_\mu, \pi_\mu) \in \mathbb{I}^{\text{MASM}}_{\text{ASM}} \wedge \mathit{wr}_{\text{MASM}}(c_\mu, \pi_\mu) \\ \emptyset & : \text{otherwise} \end{cases}$$

Sets of Addresses for C-IL First, analogously to [Bau14b] we define the computation of the set of memory byte addresses involved in the expression evaluation.

Definition 6.37 (Global Memory Footprint for Regular Pointer/Array Values). Given a regular pointer or array value $\mathbf{val}(a,t) \in val_{\mathbf{ptr}}$ such that $t \in \{\mathbf{ptr}(t'), \mathbf{array}(t',n)\}$, we define a set of accessed byte addresses (or footprint) for reading or writing a value pointed by $\mathbf{val}(a,t)$ in the C-IL global memory as

$$fp_\theta(a,t) \overset{def}{\equiv} \begin{cases} \{a\}_{size_\theta(t)} & : \neg isarray(t') \\ \emptyset & : \text{otherwise} \end{cases}$$

Definition 6.38 (Global Memory Footprint for Expression Evaluation). The function $A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} : \mathbb{E} \to 2^{\mathbb{B}^{32}}$ computes a subset of byte addresses accessed during the evaluation of a given expression wrt. a C-IL configuration $c_{\text{cil}} \in \mathbb{C}_{\text{CIL}}$, a program $\pi_{\text{cil}} \in Prog_{\text{CIL}}$, and the environment parameters θ in the following way:

- constant: $x \in val$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(x) \overset{def}{\equiv} \emptyset$$

- variable names: $v \in \mathbb{V}$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(v) \overset{def}{\equiv} \begin{cases} fp_\theta(a,t) & : [\![\&(v)]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a,t) \\ \emptyset & : \text{otherwise} \end{cases}$$

  Note that $[\![\&(v)]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}$ for any global variable v yields only the value $\mathbf{val}(a,t)$ with $t = \mathbf{ptr}(t')$ for some t' and we do not include this condition on the type in the definition. In all other cases the evaluation of the address is either undefined or gives a local reference.

- function names: $fn \in \mathbb{F}_{name}$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(fn) \overset{def}{\equiv} \emptyset$$

- unary operator: $e \in \mathbb{E}, \oslash \in \mathbb{O}_1$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\oslash e) \overset{def}{\equiv} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e)$$

- binary operator: $e_1, e_2 \in \mathbb{E}, \otimes \in \mathbb{O}_2$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_1 \otimes e_2) \overset{def}{\equiv} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_1) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_2)$$

  Note that in the definition we do not rely on the order in which the sub-expressions are evaluated.

- ternary operator: $e, e_1, e_2 \in \mathbb{E}$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e \mathbin{?} e_1 : e_2) \overset{def}{\equiv} \begin{cases} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_1) & : \neg zero_\theta\left([\![e]\!]^{\pi,\theta}_{c}\right) \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_2) & : \text{otherwise} \end{cases}$$

- type cast: $t \in \mathbb{T}_{\mathbb{Q}}, e \in \mathbb{E}$
$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}((t)e) \overset{def}{\equiv} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e)$$

- pointer dereferencing: $e \in \mathbb{E}$

$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(*(e)) \overset{def}{\equiv} \begin{cases} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) \cup fp_\theta(a,t) & : [\![e]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a,t) \wedge \\ & \quad (isptr(t) \vee isarray(t)) \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) & : \text{otherwise} \end{cases}$$

- address of: $e \in \mathbb{E}$

$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\&(e)) \overset{def}{\equiv} \begin{cases} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e') & : e = *(e') \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\&(e')) & : e = (e').f \\ \emptyset & : \text{otherwise} \end{cases}$$

- field access: $e \in \mathbb{E}, f \in \mathbb{F}$

$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}((e).f) \overset{def}{\equiv} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(*(\&((e).f)))$$

- size of a type or an expression: $x \in \mathbb{T}_\mathbb{Q} \cup \mathbb{E}$

$$A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\mathbf{sizeof}(x)) \overset{def}{\equiv} \emptyset$$

Definition 6.39 (Memory Addresses Accessed for Reading and Writing during C-IL Statement Execution). Let the statement to be executed in a C-IL configuration $c_{\text{cil}} \in \mathbb{C}_{\text{CIL}}$ be $s \equiv stmt_{next}(c_{\text{cil}}, \pi_{\text{cil}})$. Then we define

$$cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \equiv (s = \mathbf{call}\ e(E_{cas})) \wedge isfunc([\![e]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}, cas, \theta)$$

$$reads_{\text{CIL}}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) \overset{def}{\equiv} \begin{cases} A_{cas} & : cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) \cup A_E & : s = \mathbf{call}\ e(E) \wedge \\ & \quad isfunc([\![e]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}, f, \theta) \wedge f \neq cas \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) \cup A_E \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\&(e_0)) & : s = (e_0 = \mathbf{call}\ e(E)) \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(\&(e_0)) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_1) & : s = (e_0 = e_1) \\ A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e) & : s \in \{\mathbf{ifnot}\ e\ \mathbf{goto}\ l, \mathbf{return}\ e\} \\ \emptyset & : \text{otherwise} \end{cases}$$

where

$$E_{cas} \equiv e_{dest} \circ e_{cmp} \circ e_{exch} \circ e_{ret}$$

$$A_{cas} \equiv A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(*(e_{dest})) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_{cmp}) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_{exch}) \cup A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e_{ret})$$

$$A_E \equiv \bigcup_{e' \in E} A^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}(e')$$

Note that in comparison to the definition of the *reads*-set in [Bau14b], our version here is simpler and more intuitive in case of the assignment. It simply takes into account the definition of the C-IL transition function where we always compute the address of the left-hand side e_0. Moreover, we read the memory for the computation of the address during the function call with the return value.

For a configuration $c_{\text{cil}} \in \mathbb{C}_{\text{cil}}$ in which a call of the intrinsic function cas is going to be performed, i.e., $cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}})$ holds, we define the following predicates:

$$swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \overset{def}{\equiv} read\left(\theta, c_{\text{cil}}, [\![e_{dest}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}\right) = [\![e_{cmp}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}$$

Then, the set of addresses written during the call of cas depends on the condition $swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}})$ and whether the global memory or the local memory on the stack is written. In case of a write on the stack the addresses are not included into the set. Formally, we compute

$$writes^{cas}_{\text{CIL}}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) \overset{def}{\equiv} \begin{cases} fp_\theta(a_{ret}, \mathbf{ptr}(\mathbf{i32})) & : \neg swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \wedge \\ & \quad [\![e_{ret}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a_{ret}, \mathbf{ptr}(\mathbf{i32})) \\ fp_\theta(a_{ret}, \mathbf{ptr}(\mathbf{i32})) & : swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \wedge \\ & \quad [\![e_{ret}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a_{ret}, \mathbf{ptr}(\mathbf{i32})) \wedge \\ & \quad [\![e_{dest}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} \in val_{\mathbf{lref}} \\ fp_\theta(a_{dest}, \mathbf{ptr}(\mathbf{i32})) & : swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \wedge [\![e_{ret}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} \in val_{\mathbf{lref}} \wedge \\ & \quad [\![e_{dest}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a_{dest}, \mathbf{ptr}(\mathbf{i32})) \\ fp_\theta(a_{ret}, \mathbf{ptr}(\mathbf{i32})) \cup & : swap^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \wedge \\ fp_\theta(a_{dest}, \mathbf{ptr}(\mathbf{i32})) & \quad [\![e_{ret}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a_{ret}, \mathbf{ptr}(\mathbf{i32})) \wedge \\ & \quad [\![e_{dest}]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a_{dest}, \mathbf{ptr}(\mathbf{i32})) \\ \emptyset & : \text{otherwise} \end{cases}$$

In turn, the set of written addresses during a C-IL step is obtained as

$$writes_{\text{CIL}}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) \overset{def}{\equiv} \begin{cases} fp_\theta(a, t) & : s = (e_0 = e_1) \wedge [\![\&(e_0)]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}} = \mathbf{val}(a, t) \wedge \\ & \quad (isptr(t) \vee isarray(t)) \\ fp_\theta(a, t) & : s = \textbf{return } e \wedge top(c_{\text{cil}}) > 1 \wedge \\ & \quad rds_{top(c_{\text{cil}})-1}(c_{\text{cil}}) = \mathbf{val}(a, t) \\ writes^{cas}_{\text{CIL}}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) & : cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \\ \emptyset & : \text{otherwise} \end{cases}$$

Sets of Addresses for MX Finally, we define the same sets for the whole mixed machine.

Definition 6.40 (Memory Addresses Accessed for Reading and Writing during MX Machine Step). Let the shorthands be

$$c_\mu \equiv conf_{\text{MASM}}(c) \qquad c_{\text{cil}} \equiv conf_{\text{CIL}}(c)$$
$$instr_\mu \equiv instr_{next}(c_\mu, \pi_\mu) \qquad stmt_{\text{cil}} \equiv stmt_{next}(c_{\text{cil}}, \pi_{\text{cil}})$$

Then, we define

$$reads_{\text{MX}}(c, \pi, \theta) \overset{def}{\equiv} \begin{cases} reads_{\text{MASM}}(c_\mu, \pi_\mu) & : masm(c.ac) \\ reads_{\text{CIL}}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) & : cil(c.ac) \end{cases}$$

For the configuration c such that $masm(c.ac)$ holds we first introduce a predicate indicating a return from a MASM procedure to a C-IL function:

$$ret^{\text{CIL}}_{\text{MASM}}(c, \pi) \overset{def}{\equiv} nf_{ac}(c) = 1 \wedge c.ic \neq \varepsilon \wedge (instr_\mu = \mathbf{ret})$$

Analogously, for the return from C-IL to MASM we have

$$ret_{\text{CIL}}^{\text{MASM}}(c,\pi) \overset{def}{\equiv} nf_{ac}(c) = 1 \wedge c.ic \neq \varepsilon \wedge (stmt_{\text{cil}} = \textbf{return}\ e)$$

Finally, we compute

$$writes_{\text{MX}}(c,\pi,\theta) \overset{def}{\equiv} \begin{cases} writes_{\text{MASM}}(c_\mu,\pi_\mu) & : masm(c.ac) \wedge \neg ret_{\text{MASM}}^{\text{CIL}}(c,\pi) \\ fp_\theta(a,t) & : masm(c.ac) \wedge ret_{\text{MASM}}^{\text{CIL}}(c,\pi) \wedge \\ & \quad rds_{top}\,(st_{ic}(c.ic)) = \textbf{val}(a,t) \\ writes_{\text{CIL}}(c_{\text{cil}},\pi_{\text{cil}},\theta) & : cil(c.ac) \wedge \neg ret_{\text{CIL}}^{\text{MASM}}(c,\pi) \\ \emptyset & : \text{otherwise} \end{cases}$$

Definition 6.41 (No Access to icm by MX Machine). All addresses accessed during a MX machine step are combined in

$$accad_{\text{MX}}(c,\pi,\theta) \overset{def}{\equiv} reads_{\text{MX}}(c,\pi,\theta) \cup writes_{\text{MX}}(c,\pi,\theta)$$

Again, in order to show that the MX machine does not access a region icm, we define

$$noacc_{\text{MX}}^{\pi,\theta}(c,icm) \overset{def}{\equiv} accad_{\text{MX}}(c,\pi,\theta) \cap icm = \emptyset$$

6.1.2.4 Volatile Accesses, $\mathcal{IO}$- and $\mathcal{OT}$-Points

As we already know C-IL supports the concept of **volatile**-qualified types. Any access to a variable/field (or simply volatile object) of such a type is treated specifically by the compiler but does not influence the abstract semantics. Usually compilers are recommended to refrain from optimization of such accesses. This means, for example, a read of a volatile variable is always performed and the compiler cannot rely on the most recently read value because it can be changed by the environment. Moreover, the compiler can reorder the accesses to non-volatile variables, but the sequence of volatile accesses is supposed to be preserved.

Obviously, volatile accesses must be detected during the compilation, when it is not always possible to find the aliasing due to the pointer arithmetic. A typical example is an access to a volatile variable by a pointer to a non-volatile type. The aliasing could be recognized when an address assigned to the pointer is computed in the code directly from the volatile variable. However, if such a computation is based on the address of another non-volatile variable, the compiler would probably treat this access as non-volatile.

In fact, according to the C11 standard [ISO11], in case of an attempt to refer a volatile variable/field by a pointer to a type not qualified as volatile, the behaviour is undefined. To avoid such situations, many compilers introduce restrictions on how volatile variables/fields are allowed to be accessed. In this work, we also assume that volatile objects are not accessed via non-volatile pointers, what, in turn, makes it feasible to detect a volatile accesses directly from the type of the pointer. If a non-volatile object is accessed via a pointer to a volatile type, we will consider such an access also to be volatile, and will treat it here as an $\mathcal{IO}$-operation.

Any access to a volatile object is considered to be an $\mathcal{IO}$-operation. So, in order to define $\mathcal{IO}$-points for the C-IL machine, we first introduce a function computing a number of volatile accesses in a given C-IL expression.

Definition 6.42 (Pointer to a Volatile Type Suitable for Memory Access). We say that a qualified type $t_q \in \mathbb{T}_{\mathbb{Q}}$ represented as $t_q = (q,t)$ can be used for a volatile access of the C-IL memory if the following predicate hold

$$\begin{aligned} \mathit{volacc}(t_q) \overset{def}{\equiv}\ & t \in \{\mathbf{ptr}(q', t'), \mathbf{array}((q', t'), n)\} \wedge \mathbf{volatile} \in q' \wedge \\ & \neg \mathit{isarray}(t') \wedge (t' \neq \mathbf{struct}\ t_C) \end{aligned}$$

Note that we will use $\mathit{volacc}(t_q)$ only for machine states from which the computation does not lead to the run-time error. Therefore, e.g., though the access to a whole volatile variable of a composite type can be performed during the expression evaluation, the transition result for such an access is the error-state and we do not count here the number of accesses for all fields of the volatile-qualified composite type. Moreover, we can distinguish between a volatile array and an array of volatiles. An access to a variable of an volatile array type just computes the address using the environment parameter θ and, therefore, is not a volatile access.

Definition 6.43 (Number of Volatile Accesses in a C-IL Expression). The function $\mathit{Vol}_f^{\pi_{\text{cil}},\theta} : \mathbb{E} \to \mathbb{N}_0$ computes a number of volatile accesses that could appear during the evaluation of a given expression in a body of a non-external C-IL function $f \in \mathbb{F}_{name}$ of a program $\pi_{\text{cil}} \in \mathit{Prog}_{\text{CIL}}$ wrt. the environment parameters θ.

- constant: $x \in \mathit{val}$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(x) \overset{def}{\equiv} 0$$

- variable names: $v \in \mathbb{V}$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(v) \overset{def}{\equiv} \begin{cases} 1 & : \mathit{volacc}\left(\tau_f^{\pi_{\text{cil}},\theta}(\&v)\right) \\ 0 & : \text{otherwise} \end{cases}$$

- function names: $fn \in \mathbb{F}_{name}$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(fn) \overset{def}{\equiv} 0$$

- unary operator: $e \in \mathbb{E}, \oslash \in \mathbb{O}_1$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(\oslash e) \overset{def}{\equiv} \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e)$$

- binary operator: $e_1, e_2 \in \mathbb{E}, \otimes \in \mathbb{O}_2$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_1 \otimes e_2) \overset{def}{\equiv} \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_1) + \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_2)$$

- ternary operator: $e, e_1, e_2 \in \mathbb{E}$
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e\ ?\ e_1 : e_2) \overset{def}{\equiv} \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e)$$

  Here it is important to mention that the correct number of volatile accesses in the expression with the ternary operator can only be computed in the presence of the expression evaluation of e. So, depending on its value the number of accesses could be either $\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e) + \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_1)$ or $\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e) + \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_2)$. Since we compute this number from the expression without a given C-IL configuration, we have to make a restriction on the volatile accesses in the ternary operator. We chose the simplest solution such that volatile accesses are allowed to appear only in e, and not in e_1 or e_2, i.e., we require
$$\mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_1) = \mathit{Vol}_f^{\pi_{\text{cil}},\theta}(e_2) = 0$$

This definition differs from the version presented in [Bau14b] where the author tested all the expressions e, e_1, and e_2 together on the presence of the volatile accesses. However, for instance, if e and e_1 do not contain volatile accesses, while e_2 does, and during a step the condition e is evaluated to one, the volatile read in e_2 would be performed neither in the C-IL nor in the compiled code, though the whole expression with the ternary operator would contain a volatile access according to [Bau14b]. Therefore, matching $\mathcal{IO}$-points between the abstract and concrete machines wrt. Definition 2.32 would become a problem.

- type cast: $t \in \mathbb{T}_{\mathbb{Q}}, e \in \mathbb{E}$

$$Vol_f^{\pi_{\text{cil}},\theta}((t)e) \overset{def}{\equiv} Vol_f^{\pi_{\text{cil}},\theta}(e)$$

- pointer dereferencing: $e \in \mathbb{E}$

$$Vol_f^{\pi_{\text{cil}},\theta}(*(e)) \overset{def}{\equiv} \begin{cases} Vol_f^{\pi_{\text{cil}},\theta}(e) + 1 & : \mathit{volacc}\left(\tau_f^{\pi_{\text{cil}},\theta}(e)\right) \\ Vol_f^{\pi_{\text{cil}},\theta}(e) & : \text{otherwise} \end{cases}$$

- address of: $e \in \mathbb{E}$

$$Vol_f^{\pi_{\text{cil}},\theta}(\&(e)) \overset{def}{\equiv} \begin{cases} Vol_f^{\pi_{\text{cil}},\theta}(e') & : e = *(e') \\ Vol_f^{\pi_{\text{cil}},\theta}(\&(e')) & : e = (e').z \\ 0 & : \text{otherwise} \end{cases}$$

- field access: $e \in \mathbb{E}, z \in \mathbb{F}$

$$Vol_f^{\pi_{\text{cil}},\theta}((e).z) \overset{def}{\equiv} Vol_f^{\pi_{\text{cil}},\theta}(*(\&((e).f)))$$

  Note that if a composite type is volatile-qualified, then all its fields are treated as volatile independently from their explicitly given types.

- size of a type or an expression: $x \in \mathbb{T}_{\mathbb{Q}} \cup \mathbb{E}$

$$Vol_f^{\pi_{\text{cil}},\theta}(\textbf{sizeof}(x)) \overset{def}{\equiv} 0$$

If we have to compute the number of volatile accesses during the evaluation of the expression e in a C-IL configuration $c_{\text{cil}} \in \mathbb{C}_{\text{CIL}}$ we will use

$$Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(e) \overset{def}{\equiv} Vol_{f_{top}(c_{\text{cil}})}^{\pi_{\text{cil}},\theta}(e)$$

Definition 6.44 (Number of Volatile Reads and Writes in a C-IL Step). Given $c_{\text{cil}} \in \mathbb{C}_{\text{CIL}}$, θ, and π_{cil} we define the number of volatile reads and writes performed during a C-IL step from the given configuration and without the run-time error. For $s \equiv \mathit{stmt}_{next}(c_{\text{cil}}, \pi_{\text{cil}})$ we have the number of volatile reads during a C-IL step as

$$Vol_{read}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \overset{def}{\equiv} \begin{cases} Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(\&(e_0)) + Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(e_1) & : s = (e_0 = e_1) \\ Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(e) & : s = \textbf{ifnot } e \textbf{ goto } l \\ \mathit{cvol} & : s = \textbf{call } e(E) \\ \mathit{cvol} + Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(\&(e_0)) & : s = (e_0 = \textbf{call } e(E)) \\ Vol_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(e) & : s = \textbf{return } e \end{cases}$$

where the number of volatile accesses for a function call is

$$cvol \equiv Vol_f^{\pi_{\text{cil}},\theta}(e) + \sum_{e' \in E} Vol_f^{\pi_{\text{cil}},\theta}(e')$$

The number of volatile writes for a C-IL step is computed as

$$Vol_{write}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \overset{def}{\equiv} \begin{cases} \langle vw \rangle & : \ s = (e_0 = e_1) \\ \langle vw' \rangle & : \ s = \textbf{return } e \wedge top(c_{\text{cil}}) > 1 \wedge \\ & \quad\ s' = (e_0 = \textbf{call } e(E)) \\ 0 & : \ \text{otherwise} \end{cases}$$

where s' is the callee's C-IL statement that is the call of the function from which the return with a value is made

$$s' \equiv P_{top}(c'_{\text{cil}}, \pi_{\text{cil}})[loc_{top}(c'_{\text{cil}}) - 1] \quad \text{with} \quad c'_{\text{cil}} \equiv drop_{frame}(c_{\text{cil}})$$

and $vw, vw' \in \mathbb{B}$ are flags indicating that a write to a volatile type is performed in a corresponding case

$$vw \equiv volacc\left(\tau_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}(\&(e_0))\right) \qquad\qquad vw' \equiv volacc\left(\tau_{c'_{\text{cil}}}^{\pi_{\text{cil}},\theta}(\&(e_0))\right)$$

Note that according to the C-IL semantics, a C-IL step leading to the non-error state can make at most one write because the C-IL transition function cannot result in a simultaneous write to a few fields of any composite type or elements of an array. We consider the proof of this property to be just a bookkeeping exercise and to lay outside of the scope of the thesis.

Definition 6.45 (Number of Volatile Accesses during a C-IL Step). Then we define the total number of volatile accesses during a C-IL step not leading to the error-state as

$$Vol_{acc}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \overset{def}{\equiv} Vol_{read}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) + Vol_{write}^{\pi_{\text{cil}},\theta}(c_{\text{cil}})$$

Any volatile access is considered to be suitable for an $\mathcal{IO}$-operation, that's why a C-IL step containing an volatile access is at the $\mathcal{IO}$-point. However, we will also consider a step executing the function cas to be at $\mathcal{IO}$-point.

Definition 6.46 (Number of C-IL Global Memory Accesses Suitable for $\mathcal{IO}$-operations). For a C-IL step we compute the number of global memory accesses suitable for $\mathcal{IO}$-operations as a number of volatile accesses during a step plus an atomic write performed by the cas function:

$$nIO_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \overset{def}{\equiv} \begin{cases} Vol_{acc}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) + 1 & : \ cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \\ Vol_{acc}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) & : \ \text{otherwise} \end{cases}$$

Finally, we can introduce the software condition on the number of global memory accesses suitable for $\mathcal{IO}$-operations during a C-IL step

Definition 6.47 (Software Condition on Number of $\mathcal{IO}$-operations per C-IL Step). For a given C-IL configuration such that $\delta_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \neq \bot$ we require

$$sc_{\text{CIL}}^{nIO}(c_{\text{cil}}, \pi_{\text{cil}}, \theta) \overset{def}{\equiv} nIO_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \leq 1$$

Definition 6.48 (**$\mathcal{IO}$- and $\mathcal{OT}$-Points for C-IL Machine**). Given $c_{\text{cil}} \in \mathbb{C}_{\text{CIL}}$, θ, and π_{cil} we define whether the C-IL machine with the given configuration c_{cil} is at an $\mathcal{IO}$- or $\mathcal{OT}$-point respectively as follows

$$\begin{aligned}\mathcal{IO}^{\pi_{\text{cil}},\theta}_{\text{CIL}}(c_{\text{cil}}) &\overset{def}{\equiv} nIO^{\pi_{\text{cil}},\theta}_{\text{CIL}}(c_{\text{cil}}) = 1\\ \mathcal{OT}^{\pi_{\text{cil}},\theta}_{\text{CIL}}(c_{\text{cil}}) &\overset{def}{\equiv} cascall^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \vee \left(Vol^{\pi_{\text{cil}},\theta}_{write}(c_{\text{cil}}) = 1\right)\end{aligned}$$

Definition 6.49 (**$\mathcal{IO}$- and $\mathcal{OT}$-Points for MX Machine**). Let the shorthands be

$$\begin{aligned} c_\mu &\equiv conf_{\text{MASM}}(c) & st' &\equiv (st_{ic}(c.ic))\\ c_{\text{cil}} &\equiv conf_{\text{CIL}}(c) & s' &\equiv P_{top}(st',\pi)[loc_{top}(st')-1]\\ I_\mu &\equiv I(c_\mu,\pi_\mu) & f' &\equiv f_{top}(st')\end{aligned}$$

Then, the predicate $\mathcal{IO}^{\pi,\theta}_{\text{MX}}(c)$ indicates that the MX machine in a configuration $c \in \mathbb{C}_{\text{MX}}$ is in an $\mathcal{IO}$-point if (i) it is in an $\mathcal{IO}$-point during any C-IL step except the return to MASM, (ii) the return from C-IL to MASM performs a volatile read, (iii) a MASM step executes an assembly instruction cas or $locksw$[5], or (iv) the return from MASM to C-IL writes at a C-IL return value destination pointing to a volatile type:

$$\mathcal{IO}^{\pi,\theta}_{\text{MX}}(c) \overset{def}{\equiv} \begin{cases} \mathcal{IO}^{\pi_{\text{cil}},\theta}_{\text{CIL}}(c_{\text{cil}}) & : \ cil(c.ac) \wedge \neg ret^{\text{MASM}}_{\text{CIL}}(c,\pi)\\ Vol^{\pi_{\text{cil}},\theta}_{read}(c_{\text{cil}}) = 1 & : \ cil(c.ac) \wedge ret^{\text{MASM}}_{\text{CIL}}(c,\pi)\\ cas(I_\mu) \vee locksw(I_\mu) & : \ masm(c.ac) \wedge instr_{next}(c_\mu,\pi_\mu) \in \mathbb{I}^{\text{MASM}}_{\text{ASM}}\\ volacc\left(\tau^{\pi_{\text{cil}},\theta}_{f'}(\&(e_0))\right) & : \ masm(c.ac) \wedge ret^{\text{CIL}}_{\text{MASM}}(c,\pi) \wedge s' = (e_0 = \textbf{call } e(E))\\ 0 & : \ \text{otherwise}\end{cases}$$

In turn, we define $\mathcal{OT}$-points of the MX machine as

$$\mathcal{OT}^{\pi,\theta}_{\text{MX}}(c) \overset{def}{\equiv} \begin{cases} \mathcal{OT}^{\pi_{\text{cil}},\theta}_{\text{CIL}}(c_{\text{cil}}) & : \ cil(c.ac) \wedge \neg ret^{\text{MASM}}_{\text{CIL}}(c,\pi)\\ cas(I_\mu) \vee locksw(I_\mu) & : \ masm(c.ac) \wedge instr_{next}(c_\mu,\pi_\mu) \in \mathbb{I}^{\text{MASM}}_{\text{ASM}}\\ volacc\left(\tau^{\pi_{\text{cil}},\theta}_{f'}(\&(e_0))\right) & : \ masm(c.ac) \wedge ret^{\text{CIL}}_{\text{MASM}}(c,\pi) \wedge s' = (e_0 = \textbf{call } e(E))\\ 0 & : \ \text{otherwise}\end{cases}$$

Obviously, in analogy to the C-IL case we require a software condition on the number of $\mathcal{IO}$-accesses per a step of the mixed machine to hold.

Definition 6.50 (**Software Condition on Number of $\mathcal{IO}$-operations per a C-IL Step in the MX Machine**). For a given MX configuration $c \in \mathbb{C}_{\text{MX}}$, a program $\pi \in Prog_{\text{MX}}$ such that $\delta^{\pi,\theta}_{\text{MX}}(c,in) \neq \bot$ with $in \in \Sigma_{\text{MX}}$ we require

$$sc^{nIO}_{\text{MX}}(c,\pi,\theta) \overset{def}{\equiv} \begin{array}{ll} (i) & cil(c.ac) \wedge \neg ret^{\text{MASM}}_{\text{CIL}}(c,\pi) \implies sc^{nIO}_{\text{CIL}}(c_{\text{cil}},\pi_{\text{cil}},\theta)\\ (ii) & cil(c.ac) \wedge ret^{\text{MASM}}_{\text{CIL}}(c,\pi) \implies Vol^{\pi_{\text{cil}},\theta}_{read}(c_{\text{cil}}) \leq 1\end{array}$$

Recall, that in Definition 6.34 we have already introduced the conditions on the consistency points in the C-IL code. Now, additionally to them, we require that any $\mathcal{IO}$-point of the MX machine is a consistency point.

[5]Here, for simplicity, we do take into account accesses from the MASM program to volatile variables declared in the C-IL program.

Definition 6.51 (**MX $\mathcal{IO}$-Points are Consistency Points**).

$$\mathcal{IO}cp^{\pi,\theta}_{\text{MX}}(c, info) \overset{def}{\equiv} \mathcal{IO}^{\pi,\theta}_{\text{MX}}(c) \implies cp_{\text{MX}}(c, info)$$

This property is not needed for the justification of the MX semantics, but required for the correctness proof of the kernel threads. In fact, it demands that the C-IL compiler inserts consistency points before any volatile access. In case of the MASM steps, this condition trivially holds because we have consistency points at every MASM instruction.

In order to define $\mathcal{IO}$-points for the SB reduced ISA executing the compiled code of a mixed program, we first define a set of addresses of instructions implemented volatile accesses. Though we have already stated the software condition, we provide a definition in a general form, i.e., for all volatile accesses in the compiled code of any C-IL statement.

$$A^{vol}_{\text{CIL}}(\pi_{\text{cil}}, info_{\text{cil}}, \theta, cba_{\text{cil}}) \overset{def}{\equiv} \left\{ cba_{\text{cil}} +_{32} (4 \cdot o)_{32} \;\middle|\; \begin{array}{l} \exists f \in \mathbb{F}_{name}, loc \in \mathbb{N}. \\ f \in \text{dom}\left(\mathcal{F}^{\theta}_{\pi_{\text{cil}}}\right) \wedge \neg ext(f, \pi_{\text{cil}}, \theta) \wedge \\ loc \in [1 : |\mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).P|] \wedge \\ o \in info_{\text{cil}}.off_{vol}(f, loc) \end{array} \right\}$$

Definition 6.52 (**$\mathcal{IO}$- and $\mathcal{OT}$-Points for MIPS-86 ISA Executing MX**). Let a MIPS-86 instruction being executed in a configuration $d \in \mathbb{C}_{\text{MIPS}}$ be $I \equiv d.m_4(d.cpu.core.pc)$, then we define

$$\begin{aligned} \mathcal{IO}^{\text{MX}}_{r\text{MIPS}}(d, \pi, info, \theta, cba) \overset{def}{\equiv}\; & lw(I) \wedge \left(d.cpu.core.pc \in A^{vol}_{\text{CIL}}(\pi_{\text{cil}}, info_{\text{cil}}, \theta, cba_{\text{cil}})\right) \vee \\ & cas(I) \vee locksw(I) \end{aligned}$$

$$\mathcal{OT}^{\text{MX}}_{r\text{MIPS}}(d) \overset{def}{\equiv} cas(I) \vee locksw(I)$$

Note that the definition is basically rely on the core configuration and the code region, and no other memory. Moreover, $lw(I)$ is present in the formula in order to show that due to the simple store buffer reduction policy we are interested only in load operations used for volatile accesses. The predicate could also be stated without this explicit condition, though A^{vol}_{CIL} may include addresses of other memory instructions implementing volatile accesses in the code.

6.1.2.5 Addresses of Global Variables and Constants

We compute the memory byte addresses occupied by the C-IL global variables (including those which type is qualified as constant) declared in the program π_{cil}:

$$A^{gvar}_{\text{CIL}}(\pi_{\text{cil}}, \theta) \overset{def}{\equiv} \bigcup_{(v,t_q)\in\pi_{\text{cil}}.V_G} \{\theta.alloc_{gvar}(v)\}_{size_\theta(qt2t(t_q))}$$

Among these memory bytes we distinguish the memory addresses occupied by elements of arrays, variables, and fields with types qualified as constants. We collect such byte addresses in the set $A^{const}_{\text{CIL}}(\pi_{\text{cil}}, \theta) \in 2^{\mathbb{B}^{32}}$. In order to compute this set we introduce auxiliary computation. Let $a \in \mathbb{B}^{32}$ be a byte address at which a global variable, element of array, or a struct field of a qualified type $t_q \equiv (q, t)$, $t_q \in \mathbb{T}_{\mathbb{Q}}$ resides in the memory. Then we recursively define

$$A^{const}_{\text{CIL}}(a, t_q, \pi_{\text{cil}}, \theta) \overset{def}{\equiv} \begin{cases} \{a\}_{size_\theta(qt2t(t_q))} & : \textbf{const} \in q \wedge \\ & \quad \neg isarray(t) \wedge t \neq \textbf{void} \\ \bigcup_{i\in[0:n-1]} A^{const}_{\text{CIL}}\left(a_i, t'_q, \pi_{\text{cil}}, \theta\right) & : t = \textbf{array}(t'_q, n) \\ \bigcup_{(f,t'_q)\in\pi_{\text{cil}}.T_F(t_C)} A^{const}_{\text{CIL}}\left(a_f, t'_q, \pi_{\text{cil}}, \theta\right) & : \textbf{const} \notin q \wedge t = \textbf{struct}\ t_C \\ \emptyset & : \text{otherwise} \end{cases}$$

where the addresses of fields f and elements i of arrays are computes as

$$a_i \equiv a +_{32} \left(i \cdot size_\theta(qt2t(t'_q))\right)_{32} \qquad a_f \equiv a +_{32} (\theta.offset_{struc}(t_C, f))_{32}$$

Hence for all global variables we compute

$$A^{const}_{\text{CIL}}(\pi_{\text{cil}}, \theta) \overset{def}{\equiv} \bigcup_{(v,t_q)\in\pi_{\text{cil}}.V_G} A^{const}_{\text{CIL}}(\theta.alloc_{gvar}(v), t_q, \pi_{\text{cil}}, \theta)$$

6.1.2.6 Requirements and Conditions for MIPS-86 Machine

Now, we define requirements on steps of the SB reduced MIPS-86 machines which will enable the simulation for the MX model.

Definition 6.53 (Suitability of Inputs for Reduced MIPS-86 Executing MX). We consider an input $in \in \Sigma_{\text{MIPS}}$ to be suitable for the MX simulation when the reset signal is low:

$$suit^{\text{MX}}_{r\text{MIPS}}(in) \overset{def}{\equiv} in = (\texttt{core}, w_I, w_D, eev) \implies \neg eev[0]$$

Definition 6.54 (Configuration Well-Formedness for Reduced MIPS-86 Executing MX). The well-formedness of the MIPS-86 configuration for the MX simulation requires that the processor core must be in system mode, the store buffer is empty, and the maskable interrupts are masked:

$$\begin{aligned} wfconf^{\text{MX}}_{r\text{MIPS}}(d) \overset{def}{\equiv} \quad & \textit{(i)} && \neg mode(d.cpu.core) \\ & \textit{(ii)} && d.cpu.sb = \varepsilon \\ & \textit{(iii)} && \forall i \in \{1,7\}.\ d.cpu.core.spr(sr)[i] = 0 \end{aligned}$$

Definition 6.55 (Well-Behaviour of Reduced MIPS-86 Executing MX). The well-behaviour of a step of the reduced MIPS-86 machine encoding the MX semantics means that (i) the system mode is preserved (under software conditions considered later), (ii) the compiler guarantees that no illegal instructions appear in the code and memory accesses are properly aligned, and (iii) the software conditions needed for the store buffer reduction are transferred. Using the shorthands

$$\begin{aligned} core &\equiv d.cpu.core & I &\equiv d.m_4(core.pc) \\ d' &\equiv \delta_{r\text{MIPS}}(d, in) & core' &\equiv d'.cpu.core \end{aligned}$$

we formally define

$$\begin{aligned} wb^{\text{MX}}_{r\text{MIPS}}(d, in) \overset{def}{\equiv} \quad & \textit{(i)} && \neg mode(d'.cpu.core) \\ & \textit{(ii)} && \neg jisr(core, I, eev, 0, 0) \\ & \textit{(iii)} && sc_{r\text{MIPS}}(d, in) \end{aligned}$$

For brevity we provide here zeros instead of computation of page fault signals because we know that the computation is made in system mode.

6.1.2.7 MX Software Conditions

We distinguish between static and dynamic software conditions. The static software conditions make restrictions on the code of a given program and how the linker palaces the compiled program into the memory depending on the system. The dynamic software conditions are properties of steps and mostly cannot be detected during the compilation and placement of the compiled code into the memory.

Recall from Section 4.2, we split the memory into disjoint sets:

$$A_{hyp} \cup A_{guest} = \mathbb{B}^{32}, \quad A_{hyp} \cap A_{guest} = \emptyset, \quad A_{hyp} = A_{code} \cup A_{const} \cup A_{data}$$

$$\forall X, Y \in \{A_{code}, A_{const}, A_{data}\} \,.\, X \neq Y \implies X \cap Y = \emptyset$$

Definition 6.56 (**Static Software Conditions for MX**). Obviously, one has to require that the compiled code, data, and stack of the hypervisor/OS kernel resides in A_{hyp}. Taking into account that the aforementioned sets are disjoint we require the following static conditions: (i) the compiled code of the mixed program of the hypervisor/OS kernel is placed into the code region such that (ii) the binaries of C-IL and MASM programs do not overlap, (iii) – (iv) the stack and global variables not of the constant-qualified type belong to A_{data}, (v) the data C-IL constants reside in A_{const}[6]. Moreover, (vi) if a MASM procedure is supposed to be called from C-IL, it must save all callee-save registers.

$$\begin{aligned}
sc^{stat}_{\mathrm{MX}}(\pi, info, \theta, cba, sba, mss) \overset{def}{\equiv}& \\
\textit{(i)} \quad & A^{code}_{\mathrm{MX}}(info, cba) = A_{code} \\
\textit{(ii)} \quad & valid^{code}_{\mathrm{MX}}(info, cba) \\
\textit{(iii)} \quad & A^{stack}_{\mathrm{MX}}(sba, mss) \subset A_{data} \\
\textit{(iv)} \quad & A^{gvar}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) \setminus A^{const}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) \subset A_{data} \\
\textit{(v)} \quad & A^{const}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) = A_{const} \\
\textit{(vi)} \quad & sc^{prog}_{\mathrm{MX}}(\pi, \theta)
\end{aligned}$$

Definition 6.57 (**Dynamic Software Conditions for MX**). In order to define the dynamic software conditions, we consider $c' \in \mathbb{C}_{\mathrm{MX}\perp}$ such that $c' \equiv \delta^{\pi,\theta}_{\mathrm{MX}}(c, in)$ and $c_\mu \equiv conf_{\mathrm{MASM}}(c)$. Then for a step of the mixed machine we require: (i) no run-time error, (ii) – (iii) no access to stack and and code regions, (iv) no stack overflow, (v) at most one volatile access per a step, (vi) the mode and interrupt masking is not changed by the programmer, (vii) the access to the guest memory region is only performed by operations suitable for $\mathcal{IO}$-access, (viii) no misaligned data accesses from MASM are made.

$$\begin{aligned}
sc^{dyn}_{\mathrm{MX}}(c, in, \pi, info, \theta, cba, sba, mss) \overset{def}{\equiv}& \\
\textit{(i)} \quad & c' \neq \perp \\
\textit{(ii)} \quad & accad_{\mathrm{MX}}(c, \pi, \theta) \cap A^{stack}_{\mathrm{MX}}(sba, mss) = \emptyset \\
\textit{(iii)} \quad & accad_{\mathrm{MX}}(c, \pi, \theta) \cap A^{code}_{\mathrm{MX}}(info, cba) = \emptyset \\
\textit{(iv)} \quad & \neg stackovf^{\pi,\theta}_{\mathrm{MX}}(st_{\mathrm{MX}}(c'), info, sba, mss) \\
\textit{(v)} \quad & sc^{nIO}_{\mathrm{MX}}(c, \pi, \theta) \\
\textit{(vi)} \quad & \forall r \in \{mode, sr\}.\ c'.spr(r) = c.spr(r) \\
\textit{(vii)} \quad & accad_{\mathrm{MX}}(c, \pi, \theta) \cap A_{guest} \neq \emptyset \implies \mathcal{IO}^{\pi,\theta}_{\mathrm{MX}}(c) \\
\textit{(viii)} \quad & masm(c.ac) \wedge instr_{next}(c_\mu, \pi_\mu) \in \mathbb{I}^{\mathrm{MASM}}_{\mathrm{ASM}} \implies \\
& \neg dmal\,(core_{\mathrm{MIPS}}(c_\mu), I(c_\mu, \pi_\mu))
\end{aligned}$$

[6]Local variables are not forbidden from being constant-qualified. However, since they are allocated only on the stack and cannot not be used for shared accesses, we include them into A_{data}.

The condition (vii) will be guaranteed by the safety policy when we consider the concurrent model with the ownership. However, as we have seen from the software conditions in the store buffer reduction for a step of the single core MIPS-86, we need to guarantee that the guest memory is not written by the instruction sw in the compiled code. Since we can rely on the fact that volatile accesses are not translated to sw, we make this more general requirement

Moreover, we need to require the absence of date misalignment during MASM steps explicitly. As for instruction misalignment, it must be treated by the MASM compiler. Analogously, the C-IL compiler is responsible for catching both kinds of misalignment during C-IL step.

Note again that for brevity here we do not state formally the software condition requiring that volatile variables/fields are not accessed by pointers to non-volatile types (see Section 6.1.2.4).

Definition 6.58 (Software Conditions for MX Machine). Finally we combine the static and dynamic software conditions:

$$\begin{aligned} &sc_{\mathrm{MX}}(c, in, \pi, \mathit{info}, \theta, cba, sba, mss) \overset{def}{\equiv} \\ &\quad \text{(i)} \quad sc^{stat}_{\mathrm{MX}}(\pi, \mathit{info}, \theta, cba, sba, mss) \\ &\quad \text{(ii)} \quad sc^{dyn}_{\mathrm{MX}}(c, in, \pi, \mathit{info}, \theta, cba, sba, mss) \end{aligned}$$

6.1.2.8 Sequential MX Compiler Correctness in Concurrent Context

Now, we define that the software conditions hold for all defined computations from a given starting MX configuration $c_0 \in \mathbb{C}_{\mathrm{MX}}$ till the next consistency point in the following way:

$$\begin{aligned} &SCseq_{\mathrm{MX}}(c_0, \pi, \mathit{info}, \theta, cba, sba, mss, icm) \overset{def}{\equiv} \\ &\quad \forall n \in \mathbb{N}, c \in (\mathbb{C}_{\mathrm{MX}\bot})^{n+1}, \lambda \in (\Sigma_{\mathrm{MX}})^n . \\ &\qquad \text{(i)} \quad c_1 = c_0 \wedge \left(c_1 \longrightarrow^n_{\delta^{\pi,\theta}_{\mathrm{MX}}, \lambda} c_{n+1}\right) \\ &\qquad \text{(ii)} \quad \forall i \in [2:n].\ c_i \neq \bot \implies \neg cp_{\mathrm{MX}}(c_i, \mathit{info}) \\ &\qquad \text{(iii)} \quad c_{n+1} \neq \bot \implies cp_{\mathrm{MX}}(c_{n+1}, \mathit{info}) \\ &\qquad \implies \\ &\qquad \text{(i)} \quad \forall i \in \mathbb{N}_n.\ sc_{\mathrm{MX}}(c_i, \lambda_i, \pi, \mathit{info}, \theta, cba, sba, mss) \wedge \\ &\qquad\qquad\qquad noacc^{\pi,\theta}_{\mathrm{MX}}(c_i, icm) \wedge \\ &\qquad \text{(ii)} \quad c_{n+1} \neq \bot \implies wfconf^{\pi,\theta}_{\mathrm{MX}}(c_{n+1}) \end{aligned}$$

Note that the definition corresponds to one of the premises in the generalized sequential simulation theorem.

For non-empty sequences $d \in (\mathbb{C}_{\mathrm{MIPS}})^*$, $\sigma \in (\Sigma_{\mathrm{MIPS}})^*$, such that $|d| = |\sigma| + 1$, and $c \in (\mathbb{C}_{\mathrm{MX}})^*$, $\tau \in (\Sigma_{\mathrm{MX}})^*$ with $|c| = |\tau| + 1$ we define how $\mathcal{IO}$- and $\mathcal{OT}$-points are matched (wrt. Definition 2.32) between the MX machine and the MIPS-86 ISA implementing it:

$$\begin{aligned} &one\mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d, \sigma, c, \tau, \pi, \mathit{info}, \theta, cba) \overset{def}{\equiv} \\ &\quad \text{(i)} \quad \forall i, j \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}^{\pi,\theta}_{\mathrm{MX}}(c_i) \wedge \mathcal{IO}^{\pi,\theta}_{\mathrm{MX}}(c_j) \implies i = j \\ &\quad \text{(ii)} \quad \forall i, j \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_i, \pi, \mathit{info}, \theta, cba) \wedge \mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_j, \pi, \mathit{info}, \theta, cba) \implies i = j \\ &\quad \text{(iii)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}^{\pi,\theta}_{\mathrm{MX}}(c_i)\right) \implies \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_i, \pi, \mathit{info}, \theta, cba)\right) \\ &\quad \text{(iv)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{OT}^{\pi,\theta}_{\mathrm{MX}}(c_i)\right) \iff \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{OT}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_i)\right) \end{aligned}$$

Additionally we define the shorthands

$$nocp^{\mathrm{MX}}_{r\mathrm{MIPS}}(d, \pi, info, \theta, cba) \overset{def}{\equiv} \forall i \in \mathbb{N}_{|d|}.\ \neg cp^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_i, \pi, info, \theta, cba)$$

$$nocp_{\mathrm{MX}}(c, info) \overset{def}{\equiv} \forall i \in \mathbb{N}_{|c|}.\ \neg cp_{\mathrm{MX}}(c_i, info)$$

Finally, we can state the sequential mixed machine compiler correctness in the concurrent setting. Later, when we consider the justification of the concurrent MX model, we will use this compiler correctness in order to instantiate the sequential simulation framework and obtain Theorem 2.3 for the given models.

Theorem 6.1 (Sequential MX Compiler Correctness in the Concurrent Context). *Given any mixed program π, code base addresses cba, base address sba of the stack and its maximal size mss, C-IL environment parameters θ, compiler information $info$, MX machine configuration c_0, possible inconsistent portion of memory icm, and non-empty sequences d and ω of configurations and inputs respectively for the store buffer reduced MIPS-86 machine such that*

(i) the MX configuration c_0 and the starting MIPS-86 configuration d_1 of the sequence d are well-formed,

(ii) both machines are in consistency points in c_0 and d_1,

(iii) coupled via the MX compiler consistency relation wrt. the compiler information $info$ and possible inconsistent portion of memory icm belonging to the hypervisor or OS kernel addresses,

(iv) the sequence ω of MIPS-86 inputs is suitable for the simulation (has not active reset signal) and d is computed by steps of the reduced MIPS-86 machine with the inputs from ω.

(v) There are no consistency points between the first and the last configurations of d.

(vi) Moreover, the MX software conditions hold and addresses icm are not accessed during all defined MX steps from the given configuration c_0 till the next consistency point. Additionally, the MX machine has a well-formed configuration in this consistency point.

Then, there exist pairs of sequences (d', σ) and (c, τ) of configurations and inputs for the reduced MIPS-86 and MX machines respectively such that

(i) the sequences d and ω represent prefixes of d' and σ,

(ii) σ contains only suitable inputs and configurations d' are obtained by SB reduced MIPS-86 ISA steps with the inputs from σ,

(iii) these steps reach a consistency point at the end of d' not containing other consistency points in between,

(iv) are well-behaved and lead to the well-formed configuration at the end of d'.

(v) In turn, the computation of configurations in c with $c_1 = c_0$ is defined for the inputs τ by the semantics of the MX machine having at the end of the computation a well-formed state

(vi) with a consistency point next after the starting configuration c_0.

(vii) Moreover, the compiler guarantees that consistency points for the MX machine are inserted at least before and after function/procedure calls, at first statements of function bodies, at $\mathcal{IO}$-points, and

(viii) $\mathcal{IO}$- and $\mathcal{OT}$-points are properly matched between the abstract MX machine and its implementation by the reduced MIPS-86 model.

(ix) At the end of the computation both machines are also coupled via the MX compiler consistency relation wrt. info and the same icm.

$$\forall \pi \in \mathit{Prog}_{\mathrm{MX}}, \mathit{cba} \in \mathbb{B}^{32} \times \mathbb{B}^{32}, \mathit{sba} \in \mathbb{B}^{32}, \mathit{mss} \in \mathbb{N}, \theta \in \mathit{Params}_{\mathrm{CIL}},$$
$$c_0 \in \mathbb{C}_{\mathrm{MX}}, \mathit{info} \in \mathit{infoT}_{\mathrm{MX}}, \mathit{icm} \in 2^{\mathbb{B}^{32}}, k \in \mathbb{N}, d \in (\mathbb{C}_{\mathrm{MIPS}})^{k+1}, \omega \in (\Sigma_{\mathrm{MIPS}})^{k}.$$

(i) $\mathit{wfconf}^{\pi,\theta}_{\mathrm{MX}}(c_0) \wedge \mathit{wfconf}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_1)$

(ii) $\mathit{cp}_{\mathrm{MX}}(c_0, \mathit{info}) \wedge \mathit{cp}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d_1, \pi, \mathit{info}, \theta, \mathit{cba})$

(iii) $\mathit{icm} \subset A_{hyp} \wedge \mathit{consis}_{\mathrm{MX}}(c_0, d_1, \pi, \theta, \mathit{info}, \mathit{cba}, \mathit{sba}, \mathit{mss}, \mathit{icm})$

(iv) $\left(d_1 \longrightarrow^{k}_{\delta_{r\mathrm{MIPS}},\omega} d_{k+1}\right) \wedge \forall i \in \mathbb{N}_k.\ \mathit{suit}^{\mathrm{MX}}_{r\mathrm{MIPS}}(\omega_i)$

(v) $\mathit{nocp}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d[2:k], \pi, \mathit{info}, \theta, \mathit{cba})$

(vi) $\mathit{SCseq}_{\mathrm{MX}}(c_0, \pi, \mathit{info}, \theta, \mathit{cba}, \mathit{sba}, \mathit{mss}, \mathit{icm})$

$\Longrightarrow$

$$\exists n \in \mathbb{N}, d' \in (\mathbb{C}_{\mathrm{MIPS}})^{n+1}, \sigma \in (\Sigma_{\mathrm{MIPS}})^{n}, m \in \mathbb{N}, c \in (\mathbb{C}_{\mathrm{MX}})^{m+1}, \tau \in (\Sigma_{\mathrm{MX}})^{m}.$$

(i) $n \geq k \wedge d'[1:k+1] = d \wedge \sigma[1:k] = \omega$

(ii) $\left(d'_1 \longrightarrow^{n}_{\delta_{r\mathrm{MIPS}},\sigma} d'_{n+1}\right) \wedge \forall i \in \mathbb{N}_n.\ \mathit{suit}^{\mathrm{MX}}_{r\mathrm{MIPS}}(\sigma_i)$

(iii) $\mathit{cp}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d'_{n+1}, \pi, \mathit{info}, \theta, \mathit{cba}) \wedge \mathit{nocp}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d'[2:n], \pi, \mathit{info}, \theta, \mathit{cba})$

(iv) $\mathit{wfconf}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d'_{n+1}) \wedge \forall i \in \mathbb{N}_n . \mathit{wb}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d'_i, \sigma_i)$

(v) $c_1 = c_0 \wedge \left(c_1 \longrightarrow^{m}_{\delta^{\pi,\theta}_{\mathrm{MX}},\tau} c_{m+1}\right) \wedge \mathit{wfconf}^{\pi,\theta}_{\mathrm{MX}}(c_{m+1})$

(vi) $\mathit{cp}_{\mathrm{MX}}(c_{m+1}, \mathit{info}) \wedge \mathit{nocp}_{\mathrm{MX}}(c[2:m], \mathit{info})$

(vii) $\mathit{valid}^{cp}_{\mathrm{MX}}(\pi, \mathit{info}, \theta) \wedge \mathcal{IO}\mathit{cp}^{\pi,\theta}_{\mathrm{MX}}(c_{m+1}, \mathit{info})$

(viii) $\mathit{one}\mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}(d', \sigma, c, \tau, \pi, \mathit{info}, \theta, \mathit{cba})$

(ix) $\mathit{consis}_{\mathrm{MX}}(c_{m+1}, d'_{n+1}, \pi, \theta, \mathit{info}, \mathit{cba}, \mathit{sba}, \mathit{mss}, \mathit{icm})$

In comparison to the usual sequential compiler correctness proven for $icm = \emptyset$, the sequential compiler correctness needed for the concurrent systems require additional guarantees such as partial consistency in case of environment steps, the correct implementation of steps suitable for $\mathcal{IO}$-operations and ownership transfer, etc.

6.2 Concurrent MX Model Justification

6.2.1 Cosmos Model Instantiations

Concurrent Mixed Machine

Given a program $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in \mathit{Prog}_{\mathrm{MX}}$, the environment parameters $\theta \in \mathit{Params}_{\mathrm{CIL}}$, and the system information $\xi \equiv (\mathit{cba}, \mathit{sba}, \mathit{mss})$ with $\mathit{cba} \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, $\mathit{sba} \in \left(\mathbb{B}^{32}\right)^{np}$, $\mathit{mss} \in (\mathbb{N})^{np}$

wrt. the number $np \in \mathbb{N}$ of processors in the multi-core MIPS-86 machine such that

$$valid_{\text{MX}}^{stacks}(np, sba, mss) \overset{def}{\equiv} \forall p, q \in \mathbb{N}_{np}.\, p \neq q \implies A_{\text{MX}}^{stack}(sba_p, mss_p) \cap A_{\text{MX}}^{stack}(sba_q, mss_q) = \emptyset$$

we can now define the instantiation $S_{\text{MX}}^{\pi,\theta,\xi} \in \mathbb{S}$ of the *Cosmos* model for the mixed machine.

Let $c_{\text{mx}}(u, m) \equiv conf_{\text{MX}}\,(u, \lceil m \rceil_{\mathbb{B}^{32}})$ be a shorthand denoting a sequential MX machine configuration obtained on the base of the unit's configuration u and a memory m restricted by the set of addresses considered in the component of the *Cosmos* model signature. Moreover, the result of the compilation is $info \equiv (cmpl_{\text{MASM}}(\pi_\mu), cmpl_{\text{CIL}}(\pi_{\text{cil}}))$. Then the components are instantiated as follows:

- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{A} = \mathbb{B}^{32} \setminus \left(A_{\text{MX}}^{code}(info, cba) \cup \bigcup_{p \in \mathbb{N}_{np}} A_{\text{MX}}^{stack}(sba_p, mss_p)\right)$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{V} = \mathbb{B}^8$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{R} = A_{\text{CIL}}^{const}(\pi_{\text{cil}}, \theta)$
- $S_{\text{MX}}^{\pi,\theta,\xi}.nu = np$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{U} = \mathbb{K}_{\text{MX}\perp}$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{E} = \Sigma_{\text{MX}}$
- $S_{\text{MX}}^{\pi,\theta,\xi}.reads(u, m, in) = \begin{cases} reads_{\text{MX}}\big(c_{\text{mx}}(u, m), \pi, \theta\big) & : u \neq \perp \\ \emptyset & : \text{otherwise} \end{cases}$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\delta(u, m, in) = \begin{cases} \big(u',\ c'_{\text{mx}}.\mathcal{M}|_{writes_{\text{MX}}(c_{\text{mx}}(u,m),\pi,\theta)}\big) & : u \neq \perp \wedge c'_{\text{mx}} \neq \perp \\ (\perp, m_\emptyset) & : u \neq \perp \wedge c'_{\text{mx}} = \perp \\ undefined & : \text{otherwise} \end{cases}$

  where the next sequential MX machine configuration $c'_{\text{mx}} \equiv \delta_{\text{MX}}^{\pi,\theta}\,(c_{\text{mx}}(u, m), in)$ is used to compute the unit's next configuration $u' \equiv (c'_{\text{mx}}.ac, c'_{\text{mx}}.ic, c'_{\text{mx}}.spr)$ and $m_\emptyset$ in the empty function satisfying $\text{dom}\,(m_\emptyset) = \emptyset$ [7].
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{IP}(u, m, in) = (u \neq \perp \implies cp_{\text{MX}}(u.ac, info))$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{IO}(u, m, in) = \Big(u \neq \perp \implies \mathcal{IO}_{\text{MX}}^{\pi,\theta}\,(c_{\text{mx}}(u, m))\Big)$
- $S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{OT}(u, m, in) = \Big(u \neq \perp \implies \mathcal{OT}_{\text{MX}}^{\pi,\theta}\,(c_{\text{mx}}(u, m))\Big)$

Note that the memory m in the last three components represents the read-only memory, therefore we are not allowed to consider the whole memory for identification of consistency points and $\mathcal{IO}$-/$\mathcal{OT}$-points. In fact, we took this into account already in the definitions for the mixed machine, where we do not rely on the memory configuration at all. The whole MX configuration $c_{\text{mx}}(u, m)$ is used here only for brevity.

[7] Recall that according to *Cosmos* model signature from Definition 2.1 the result of the transition is a unit configuration (instantiated as $\mathbb{K}_{\text{MX}\perp}$ for our machine here) and a part of the memory modified during the step. Therefore, in order to match this formalism, we get $S_{\text{MX}}^{\pi,\theta,\xi}.\delta(u, m, in) = (\perp, m_\emptyset)$ in case of an MX run-time error.

SB Reduced Multi-Core MIPS-86 Implementing Concurrent MX Machine

In Section 4.3.2 for the proof of the store buffer reduction we have already considered the instantiation of the *Cosmos* model with the reduced MIPS-86 machine. Since we allowed the interleaving in every step and detected $\mathcal{IO}$-/$\mathcal{OT}$-points independently from the compiled code, we adapt the instantiation $S_{r\text{MIPS}} \in \mathbb{S}$ to $S_{r\text{MIPS}}^{\pi,\theta,\xi} \in \mathbb{S}$ taking into account the compilation of the MX program and the choice of the consistency points as well as the steps suitable for $\mathcal{IO}$-operations and ownership transfer.

- For components $X \in \{\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta\}$ the instantiation is equal to the one for the reduced machine
$$S_{r\text{MIPS}}^{\pi,\theta,\xi}.X = S_{r\text{MIPS}}.X$$
with $A_{code} = A_{\text{MX}}^{code}(info, cba)$ and $A_{const} = A_{\text{CIL}}^{const}(\pi_{\text{cil}}, \theta)$.
- $S_{r\text{MIPS}}^{\pi,\theta,\xi}.\mathcal{IP}(u, m, in) = cp_{r\text{MIPS}}^{\text{MX}}(u.core, \pi, info, \theta, cba)$
- $S_{r\text{MIPS}}^{\pi,\theta,\xi}.\mathcal{IO}(u, m, in) = \mathcal{IO}_{r\text{MIPS}}^{\text{MX}}\left((u, \lceil m \rceil_{\mathbb{B}^{32}}), \pi, info, \theta, cba\right)$
- $S_{r\text{MIPS}}^{\pi,\theta,\xi}.\mathcal{OT}(u, m, in) = \mathcal{OT}_{r\text{MIPS}}^{\text{MX}}\left((u, \lceil m \rceil_{\mathbb{B}^{32}})\right)$

Note again that the definitions of $\mathcal{IO}_{r\text{MIPS}}^{\text{MX}}$ and $\mathcal{OT}_{r\text{MIPS}}^{\text{MX}}$ rely on the memory region containing the compiled code which is, in turn, is treated as read-only memory in our instantiation.

6.2.2 Sequential Simulation Theorem

Now, taking into account the formulation of Theorem 6.1, we instantiate the sequential simulation framework $R_{S_{r\text{MIPS}}}^{S_{\text{MX}}}(\pi, \theta, \xi) \in \mathbb{R}$ for the two *Cosmos* models $S_{\text{MX}}^{\pi,\theta,\xi}, S_{r\text{MIPS}}^{\pi,\theta,\xi} \in \mathbb{S}$ defined wrt. the given π, θ, and ξ.

For $d \in \mathbb{C}_{proc} \times \mathbb{C}_m$, $d = (cpu, m)$, and $c \in \mathbb{K}_{\text{MX}\perp} \times \left(S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{A} \rightarrow S_{\text{MX}}^{\pi,\theta,\xi}.\mathcal{V}\right)$, $c = (k, \mathcal{M})$, and $info \in infoT_{\text{MX}}$, $icm \in 2^{\mathbb{B}^{32}}$ we define

$$R_{S_{r\text{MIPS}}}^{S_{\text{MX}}}(\pi, \theta, \xi). \left\{ \begin{array}{lcl} \mathcal{P} & = & infoT_{\text{MX}} \\ sim(d, c, info, icm) & = & icm \subset A_{hyp} \wedge valid_{\text{MX}}^{cp}(\pi, info, \theta) \wedge \mathcal{IO}cp_{\text{MX}}^{\pi,\theta}(c, info) \wedge \\ & & \big(k \neq \perp \implies \\ & & consis_{\text{MX}}\left(c_{\text{mx}}(k, \mathcal{M}), d, \pi, \theta, info, cba, sba, mss, icm'\right)\big) \\ \mathcal{CP}a(k, info) & = & (k \neq \perp \implies cp_{\text{MX}}(k.ac, info)) \\ \mathcal{CP}c(cpu, info) & = & cp_{r\text{MIPS}}^{\text{MX}}(cpu.core, \pi, info, \theta, cba) \\ wfa(c) & = & k \neq \perp \wedge wfconf_{\text{MX}}^{\pi,\theta}\left(c_{\text{mx}}(k, \mathcal{M})\right) \\ wfc(d) & = & wfconf_{r\text{MIPS}}^{\text{MX}}(d) \\ suit(in_d) & = & suit_{r\text{MIPS}}^{\text{MX}}(in_d) \\ sc(c, in_c, info) & = & valid_{\text{MX}}^{stacks}(np, sba, mss) \wedge (k \neq \perp \implies \\ & & sc_{\text{MX}}\left(c_{\text{mx}}(k, \mathcal{M}), in_c, \pi, info, \theta, cba, sba, mss\right)) \\ wb(d, in_d, info) & = & wb_{r\text{MIPS}}^{\text{MX}}(d, in_d) \end{array} \right.$$

where the $icm' \equiv icm \cup \bigcup_{p \in \mathbb{N}_{np}} A_{\text{MX}}^{stack}(sba_p, mss_p)$.

Note that since in Theorem 6.1 we use more predicates than are provided by the sequential simulation framework in Definition 2.30, we have to combine them properly in the framework instantiation. Moreover, the unit configuration k of the MX *Cosmos* machine can be the run-time error state $\perp$ and must be excluded in the application of MX predicates undefined for it. Additionally to the MX software conditions, we also require that the stacks of the concurrent MX machine do not overlap.

Having Theorem 6.1 proven by a compiler developer, it is easy to show (by unfolding the definitions) that the generalized sequential simulation Theorem 2.3 also holds for our *Cosmos* models $S_{\text{MX}}^{\pi,\theta,\xi}, S_{r\text{MIPS}}^{\pi,\theta,\xi} \in \mathbb{S}$ and the simulation framework $R_{S_{r\text{MIPS}}}^{S_{\text{MX}}}(\pi,\theta,\xi) \in \mathbb{R}$ instantiated wrt. the given program π and the parameters θ, ξ. Therefore, the sequential MX compiler correctness in the concurrent setting in terms of *Cosmos* machines can be stated in the following way:

Theorem 6.2 (Sequential MX Compiler Correctness for *Cosmos* Model Simulation). *The generalized sequential simulation Theorem 2.3 holds for any* Cosmos *models $S_{\text{MX}}^{\pi,\theta,\xi}, S_{r\text{MIPS}}^{\pi,\theta,\xi} \in \mathbb{S}$ and the simulation framework $R_{S_{r\text{MIPS}}}^{S_{\text{MX}}}(\pi,\theta,\xi) \in \mathbb{R}$ instantiated wrt. the any given mixed machine program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in Prog_{\text{MX}}$, the environment parameters $\theta \in Params_{\text{CIL}}$, and the system information $\xi \equiv (cba, sba, mss)$ with $cba \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, $sba \in \left(\mathbb{B}^{32}\right)^{np}$, $mss \in (\mathbb{N})^{np}$ and the number $np \in \mathbb{N}$ of processors in the multi-core MIPS-86 machine such that the condition $valid_{\text{cMX}}^{stack}(np, sba, mss)$ holds.*

6.2.3 Concurrent Model Simulation

Finally, in order to show that the execution of the compiled code of any mixed program on the multi-core MIPS-86 machine corresponds to the steps of the concurrent mixed machine, we instantiate other predicates needed for the *Cosmos* model simulation theorem and the proof of the required assumptions.

Again, we first consider the simulation for machines with the given program π and the parameters θ, ξ.

Definition 6.59 (Shared Invariant for Concurrent Mixed Machine Simulation).

$$sinv_{S_{r\text{MIPS}}}^{S_{\text{MX}}}(\pi,\theta,\xi)\big((m,\mathcal{S},\mathcal{R},\mathcal{O}),(m_{\text{mx}},\mathcal{S}_{\text{mx}},\mathcal{R}_{\text{mx}},\mathcal{O}_{\text{mx}}), info\big) \overset{def}{\equiv}$$

$$\begin{array}{ll} (i) & \mathcal{S} = \mathcal{S}_{\text{mx}} \\ (ii) & \mathcal{R} = \mathcal{R}_{\text{mx}} \cup A_{\text{MX}}^{code}(info, cba) \\ (iii) & \forall p \in \mathbb{N}_{nu}.\ \mathcal{O}(p) = \mathcal{O}_{\text{mx}}(p) \cup A_{\text{MX}}^{stack}(sba_p, mss_p) \\ (iv) & \forall p \in \mathbb{N}_{nu}.\ A_{\text{MX}}^{stack}(sba_p, mss_p) \cap \mathcal{S} = \emptyset \\ (v) & m\,|_{\mathcal{S}_{\text{mx}} \cup \mathcal{R}_{\text{mx}}} = m_{\text{mx}} \end{array}$$

The part (iv) was not explicitly required in [Bau14b]. However, it is needed in our work because of the further property transfer. Obviously, the stack currently used by a processor should not be shared among others.

Analogously to the store buffer reduction we restrict the ownership safety policy for the concurrent mixed machine.

Definition 6.60 (Restriction on the Ownership Transfer in Concurrent MX Machine). For any configuration $E \in \mathbb{C}_{S_{\text{MX}}^{\pi,\theta,\xi}}$ of the MX *Cosmos* machine we require that A_{guest} is always treated as shared and no addresses from A_{guest} can be owned by any execution unit. Formally, we define this by the property instantiating P_{S_a} in Theorem 2.4:

$$P_{S_{\text{MX}}^{\pi,\theta,\xi}}(E) \overset{def}{\equiv} A_{guest} \subset E.\mathcal{S} \wedge \forall p \in \mathbb{N}_{nu}.\ E.\mathcal{O}_p \cap A_{guest} = \emptyset$$

Since there is no need to make further restrictions on the unit's configuration of the SB reduced machine, we set

$$uinv^{S_{\mathrm{MX}}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,\xi)(cpu,\mathcal{O},\mathcal{S}) \overset{def}{\equiv} 1$$

Now, in order to guarantee that the *Cosmos* model simulation Theorem 2.4 holds for the given models, sequential simulation framework, and additional predicates instantiated for the given π, θ, ξ, one has to discharge Assumptions 2.1–2.4. Since the implementation of both C-IL and MASM compilers is out of the scope of this thesis, we do not consider the proof of the assumptions here. The main ideas of such proofs for MASM and C-IL separately, however, can be found in [Bau14b].

Obviously, we also have to guarantee that Theorem 2.4 holds for all our instantiations.

Theorem 6.3 (*Cosmos* Model Simulation Theorem for all Mixed Machine Programs and System/Environment Parameters). *Theorem 2.4 holds for any mixed machine programs $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in Prog_{\mathrm{MX}}$, the environment parameters $\theta \in Params_{\mathrm{CIL}}$, and the system information $\xi \equiv (cba, sba, mss)$ used for instantiation of the models $S^{\pi,\theta,\xi}_{\mathrm{MX}}, S^{\pi,\theta,\xi}_{r\mathrm{MIPS}} \in \mathbb{S}$.*

6.2.4 Application of the Concurrent MX Machine Simulation

The overall model stack for the justification of the concurrent MX machine implemented on the multi-core MIPS-86 is depicted on Figure 6.9 and considered in this section. We discuss the application of the theorems introduced by now and sketch the safety and properties transfer required for the overall simulation.

6.2.4.1 Properties Transfer from MX Machine to Reduced MIPS-86 ISA

Above we have stated the simulation theorem between the store buffer reduced multi-core MIPS-86 model $S^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ and the concurrent mixed machine $S^{\pi,\theta,\xi}_{\mathrm{MX}}$. This theorem can only be applied to the reordered steps of MIPS-86 composing incomplete consistency blocks.

The justification of this reordering preserving verified properties and safety were considered in detail in Section 2.4 where we transformed the concrete machine with the well-behaviour proven as a property of steps to an extended machine with the same property based only on a *Cosmos* machine configuration. In the context of the concurrent MX justification we denote this machine by $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ depicted on Figure 6.9. However, since we have made Assumption 2.5 on the well-behaviour and the transfer of any abstract property must satisfy the requirements introduced in Definitions 2.48 – 2.49, we have to discharge them here.

Recall that according to Assumption 2.5 the well-behaviour of the MIPS-86 machine should not rely on the memory outside of the *reads*-set. For the proof of this assumption we consider a similar lemma.

Lemma 6.1 (MIPS-86 Well-Behaviour Restriction for Concurrent MX Simulation). *For all configurations $(cpu, m) \in \mathbb{C}_{\mathrm{MIPS}}$ and inputs $in \in \Sigma_{\mathrm{MIPS}}$, we have*

$$wb^{\mathrm{MX}}_{r\mathrm{MIPS}}((cpu,m),in) \Longleftrightarrow \big(\forall m'.\ m'\,|_R = m\,|_R \implies wb^{\mathrm{MX}}_{r\mathrm{MIPS}}((cpu,m'),in)\big)$$

where the reads-set R is computed as $R \equiv reads_{\mathrm{MIPS}}((cpu,m),in)$.

Proof: The proof is a trivial bookkeeping and based on the fact that the definition of $wb^{\mathrm{MX}}_{r\mathrm{MIPS}}$ relies only on the processor core configuration and the instruction fetched in system mode. The

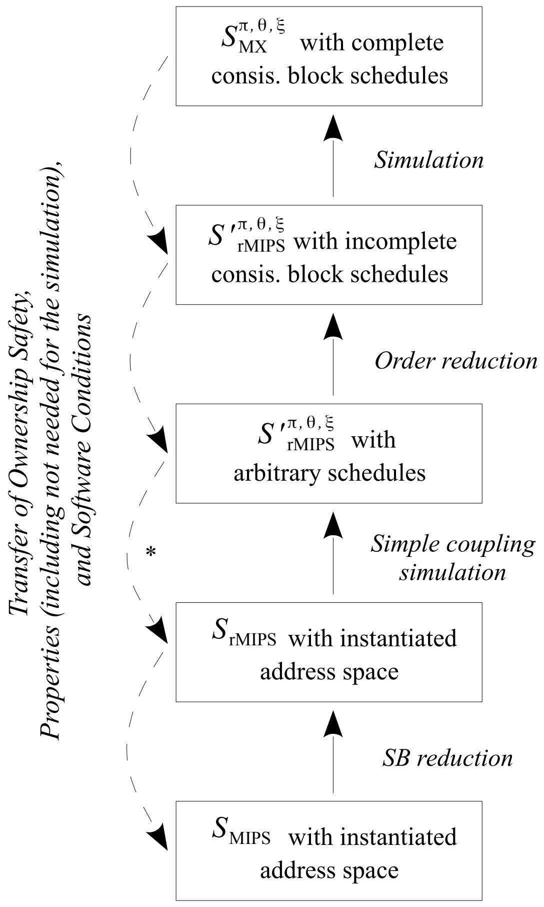
$S_{\mathrm{MX}}^{\pi,\theta,\xi}$ with complete consis. block schedules
Simulation
$S'^{\pi,\theta,\xi}_{\mathrm{rMIPS}}$ with incomplete consis. block schedules
Order reduction
$S'^{\pi,\theta,\xi}_{\mathrm{rMIPS}}$ with arbitrary schedules
Simple coupling simulation
S_{rMIPS} with instantiated address space
SB reduction
S_{MIPS} with instantiated address space
Transfer of Ownership Safety, Properties (including not needed for the simulation), and Software Conditions
*

Figure 6.9: Justification of the concurrent MX implemented on multi-core MIPS-86. As a special case of the properties transfer marked with * we consider obtaining the software conditions for store buffer reduction from the well-behavior proven for the reduced MIPS-86 machine implementing the concurrent MX semantics.

byte addresses at which the instruction resides in the memory are included into the $reads$-set by definition. □

Moreover, later for the application of the store buffer reduction we need the property $P_{S_{r\mathrm{MIPS}}}(E)$ for $E \in \mathbb{C}_{S_{r\mathrm{MIPS}}}$. We have already required $P_{S^{\pi,\theta,\xi}_{\mathrm{MX}}}(C)$ for the complete consistency blocks of the instantiated MX model with configurations $C \in \mathbb{C}_{S^{\pi,\theta,\xi}_{\mathrm{MX}}}$. Since there is a slight difference between $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ and $S_{r\mathrm{MIPS}}$, first we are interested in the transfer of the property $P_{S^{\pi,\theta,\xi}_{\mathrm{MX}}}(C)$ to a similar one for $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$, i.e., for $D \in \mathbb{C}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$

$$P_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(D) \overset{def}{\equiv} A_{guest} \subset D.\mathcal{S} \wedge \forall p \in \mathbb{N}_{nu}.\ D.\mathcal{O}_p \cap A_{guest} = \emptyset$$

Since both $P_{S^{\pi,\theta,\xi}_{\mathrm{MX}}}(C)$ and $P_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(D)$ are divisible according to Definition 2.48 and they are global properties, we easily prove the property transfer in a way stated in Definition 2.49.

Lemma 6.2 (Transfer of the Property $P_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}$).

$$\forall D, C, par.\ sinv(D, C, par) \implies \left(P_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(D) = P_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}(C)\right)$$

<u>**Proof**</u>: The proof follows directly from the shared invariant. Since the shared addresses of both machines are equal, one easily concludes

$$(A_{guest} \subset D.\mathcal{S}) = (A_{guest} \subset C.\mathcal{S}) \tag{6.1}$$

Moreover, using $sinv(D, C, par).(iii)$ for any $p \in \mathbb{N}_{nu}$ we can transform

$$(D.O_p \cap A_{guest} = \emptyset) = \big((C.O_p \cap A_{guest} = \emptyset) \wedge \big(A_{\mathrm{MX}}^{stack}(sba_p, mss_p) \cap A_{guest} = \emptyset\big) \big) \tag{6.2}$$

We proceed with the rest of the proof in both directions. Taking into account the equation (6.1), we need to discharge the following claims:

- $(A_{guest} \subset C.\mathcal{S} \wedge C.O_p \cap A_{guest} = \emptyset) \implies D.O_p \cap A_{guest} = \emptyset$

  By the equations (6.1) – (6.2) this claim to be proven is simplified to

  $$A_{guest} \subset D.\mathcal{S} \implies A_{\mathrm{MX}}^{stack}(sba_p, mss_p) \cap A_{guest} = \emptyset$$

  From $sinv(D, C, par).(iv)$ we know that the stack cannot be shared, i.e., $A_{\mathrm{MX}}^{stack}(sba_p, mss_p) \cap \mathcal{S} = \emptyset$ holds. Hence, using $A_{guest} \subset D.\mathcal{S}$ we conclude that it also does not overlap with the guests/processes address space A_{guest}.

- $D.O_p \cap A_{guest} = \emptyset \implies C.O_p \cap A_{guest} = \emptyset$

  Using the equation (6.2) we easily conclude this claim and finish the proof of the lemma.

□

Along with the property $P_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}$ for the MX *Cosmos* machine, one could be interested in the transfer of any other abstract property $P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}$ such that the simulation hypothesis of the concurrent simulation theorem hold for start configurations $C \in \mathbb{C}_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}$, $D \in \mathbb{C}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$, and a simulation parameter $par \in infoT_{\mathrm{MX}}$.

If this additional property $P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}$ is divisible and can be translated into the incompletely simulated *Cosmos* machine property $Q[P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}, par]$ wrt. Definitions 2.49 – 2.50, then applying Theorem 2.6 and Lemmas 6.1 – 6.2 we conclude that any suitable *Cosmos* machine schedule leaving D is safe wrt. not only the ownership and $P_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$, but also the translated property $Q[P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}, par]$. Moreover, all implementing computations of $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ are well-behaved.

Corollary 6.1 (Well-Behaviour and Safety Transfer for Arbitrary Schedules in MX Simulation). *Let the transferred property $\widetilde{P}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(d)$ be defined for $d \in \mathbb{C}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$ as*

$$\widetilde{P}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(d) \equiv W(d) \wedge P_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}(d) \wedge Q[P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}, par](d)$$

then formally we claim

$$\forall D, C, P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}, par.\ simh(D, C, P'_{S_{\mathrm{MX}}^{\pi,\theta,\xi}}, par) \implies safety\left(D, \widetilde{P}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}, suit\right)$$

where according to the shorthands from Section 2.3.2 and the instantiation of the sequential simulation framework $R^{S_{\mathrm{MX}}}_{S_{r\mathrm{MIPS}}}(\pi, \theta, \xi)$ *the suitability* $suit(\omega)$ *for* $\omega \in \Theta^*_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$ *is*

$$suit(\omega) \equiv \forall \beta \in \omega.\ R^{S_{\mathrm{MX}}}_{S'_{r\mathrm{MIPS}}}(\pi, \theta, \xi).suit(\beta.in)$$

6.2.4.2 Sketch for the Application of Store Buffer Reduction

Finally, in order to apply the store buffer reduction one has to consider an easy simulation between the SB reduced model $S_{r\mathrm{MIPS}}$ (in system mode) from Chapter 4 with the address space instantiated in the MX software conditions and the SB reduced model $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ involved into the concurrent MX machine simulation. The ownership safety policy including the property $\widetilde{P}_{S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}}$ from Corollary 6.1 must be transferred. Moreover, software conditions required for the store buffer reduction are derived from the well-bahaviour in $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$.

Both models slightly differ in the configuration and the instantiation of the $\mathcal{IO}$-points. In the store buffer reduction we treat any lw instruction as an $\mathcal{IO}$-operation. On the other hand, the SB machine considered in the concurrent MX simulation takes into account only those lw which are marked as volatile accesses by the compiler. Since we require the ownership safety on the level $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ having fewer $\mathcal{IO}$-points, it is easy to prove that the safety holds also for $S_{r\mathrm{MIPS}}$.

Note, however, that we proved the store buffer reduction in a general case without restrictions on the configurations and suitability. We also required that the software conditions and safety holds for any executions of the reduced machine. As we have just shown by Corollary 6.1, we can transfer such properties only for suitable schedules. Moreover, in the well-formedness for $S'^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ we require that all processors are in system mode and the store buffers are empty. Since the application of the store buffer reduction considered in this thesis is rather a technical task, we leave it for future work. The simulation and property transfer for the store buffer reduction wrt. a more sophisticated programming policy can be found in [Che16].

7 Semantics and Correctness of Concurrent Extended Mixed Machine for MIPS-86

In Chapters 5–6 we introduced the mixed machine semantics and considered in detail its compiler correctness and the soundness of the concurrent MX model implemented on the multi-core MIPS-86 machine where each processor executes the compiled code of programs without inline assembly portions.

As we mentioned before, system programming, however, requires more low-level operations than the MX semantics permits. Such operations can be implemented by using inline assembly in the mixed programs. This chapter is devoted exactly to the development of the semantics for this extended language and the justification of its concurrent version. We call the abstract machine able to execute the mixed programs enriched with inline assembly the *extended mixed machine* or simply *MXA machine* standing for MX+Assembly.

The idea of such a model, suggested by Prof. Wolfgang J. Paul and then studied independently in [PBLS15] for sequential C0+Assembly, is very intuitive. In fact, since we have already claimed the correctness of the sequential MX machine, one can easily consider the execution of the abstract MX machine as long as its compiled code runs on the processor. When an inline assembly portion is encountered in the MASM program, we can easily switch to the MIPS-86 configuration coupled by the MX compiler consistency relation. Therefore, one can continue the execution on this level until an MX consistency point at which an abstract MX configuration exists is reached. In order to get such an abstract configuration, one has to reconstruct a corresponding MX thread configuration from the MIPS-86 configuration, the executed program, and the compiler information.

Though the idea of the model considered in this thesis is similar to [PBLS15], the configuration and the semantics of our MXA machine is different because of a few reasons along with the MX language instead of C0. First of all, we have to justify the concurrent MXA model implemented by arbitrary interleaved steps of the hypervisor / OS kernel and user processes / guests respectively. For that purpose, we have to apply the theory of concurrent simulation and order reduction for *Cosmos* machines, which, in turn, allows less freedom in all formal definitions to match it. Another reason is the context in which our MXA model will be used. One should mention here, that in contrast to [PBLS15] where only one stack for the OS kernel is considered, we will argue about many stacks that can be added and deleted.

We proceed in the same way as it was done in the previous two chapters. Before we define the MXA machine transition function, we consider the reconstruction formally in detail.

7.1 Sequential Extended Mixed Machine (MXA) Semantics

7.1.1 MXA Programs and Environment Parameters

Since in Section 5.1.1 we have already introduced MASM programs containing inline assembly, we will consider the same MX programs $\pi = (\pi_\mu, \pi_{\text{cil}}) \in \mathit{Prog}_{\text{MX}}$ here. Obviously, the C-IL environment parameters $\theta \in \mathit{Params}_{\text{CIL}}$ are also as before.

7.1.2 Machine Configuration

In the definition of the semantics for the concurrent mixed machine from Section 5.4 we introduced the notion of the MX thread and its configuration. In the model of the extended mixed machine such a thread can be additionally represented by a processor configuration.

Definition 7.1 (MXA Thread Configuration). A configuration of the MXA thread is either the MIPS-86 processor configuration, or a tuple consisting of the MX thread configuration, base address and maximal size of the current stack abstracted in this configuration, and the configuration of the TLB.

$$\mathbb{K}_{\text{MXA}} \overset{def}{\equiv} \big(k_{\text{mx}} \in \mathbb{K}_{\text{MX}},\ sba \in \mathbb{B}^{32},\ mss \in \mathbb{N},\ tlb \in \mathbb{C}_{tlb}\big) \cup \mathbb{C}_{proc}$$

Though the MX thread cannot explicitly operate on the TLB, its configuration is included as a component so that it can be preserved during MX steps and later be used for the processor configuration.

Definition 7.2 (Sequential MXA Configuration). Therefore, the sequential MXA machine configuration is the MXA thread configuration accompanied by the byte addressable memory.

$$\mathbb{C}_{\text{MXA}} \overset{def}{\equiv} \big(k \in \mathbb{K}_{\text{MXA}},\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^{8}\big)$$

Definition 7.3 (Type of MXA Thread). Depending of the MXA thread configuration $k \in \mathbb{K}_{\text{MXA}}$, we distinguish whether the MXA machine makes MX or processor steps by the predicates

$$isa(k) \overset{def}{\equiv} k \in \mathbb{C}_{proc} \qquad mx(k) \overset{def}{\equiv} \neg isa(k)$$

Definition 7.4 (MX and SB Reduced Single Core MIPS-86 Configurations from MXA Configuration). For a given configuration $c \in \mathbb{C}_{\text{MXA}}$ performing mixed machine semantics steps, i.e., the predicate $mx(c.k)$ holds, we can compose the corresponding sequential MX configuration by the function

$$\mathit{conf}^{\text{MXA}}_{\text{MX}}(c) \overset{def}{\equiv} \mathit{conf}_{\text{MX}}(c.k.k_{\text{mx}}.ac,\ c.k.k_{\text{mx}}.ic,\ c.k.k_{\text{mx}}.spr,\ c.\mathcal{M})$$

Otherwise, if the MXA thread configuration is MIPS-86 ISA, one simply gets

$$\mathit{conf}^{\text{MXA}}_{r\text{MIPS}}(c) \overset{def}{\equiv} (c.k, c.\mathcal{M})$$

Definition 7.5 (Well-Formed MXA Configuration). We call an MXA configuration $c \in \mathbb{C}_{\text{MXA}}$ *well-formed* if its MXA thread configuration is *well-formed*, namely

$$\begin{aligned}\mathit{wfconf}^{\pi,\theta,cba}_{\text{MXA}}(c) \overset{def}{\equiv}\ & \textit{(i)} \quad mx(c.k) \implies \mathit{wfconf}^{\pi,\theta}_{\text{MX}}\Big(\mathit{conf}^{\text{MXA}}_{\text{MX}}(c)\Big)\\ & \textit{(ii)} \quad isa(c.k) \wedge \neg mode(c.k.core) \implies c.k.sb = \varepsilon\\ & \textit{(iii)} \quad consis^{code}_{\text{MX}}(c.\mathcal{M}, \pi, info, cba)\end{aligned}$$

with $\mathit{info} = (\mathit{info}_{\mathrm{cil}}, \mathit{info}_{\mu}) \in \mathit{infoT}_{\mathrm{MX}}$, computed as $\mathit{info}_{\mu} = \mathit{cmpl}_{\mathrm{MASM}}(\pi_{\mu})$ and $\mathit{info}_{\mathrm{cil}} = \mathit{cmpl}_{\mathrm{CIL}}(\pi_{\mathrm{cil}})$.

The last condition is added explicitly because we will treat the code region as read-only memory later on. Therefore, for the MXA semantics we have to require that the compiled code of the MX program with inline assembly is always in the memory.

7.1.3 Stack Information Abstraction

The configuration of the extended mixed machine contains the stack base address and its maximal size when the mixed machine steps are performed. With the start of the inline assembly, we havoc this information and allow the low-level operations on the stack to do what needed for the system programming. As soon as the control reaches the compiled code of the C-IL and MASM programs excluding inline assembly portions, the stack is supposed to be reconstructed. Since the stack substitution could be made, we need to find a matching pair of the stack base address and its length existing somewhere in system data structures in the memory.

Since on this level of abstraction we are not aware of concrete data structures containing the information about stacks, e.g., process control block, thread local storage, etc, we will operate here with general notions of *stack information* and *stack information regions*. In our interpretation the *stack information* includes all pairs of base addresses and maximal sizes of all stacks present in the system. In turn, a memory region containing the information about a single stack is called the *stack information region*. Each such region is identified by its base address.

The stack information can be stored in the memory in different ways (e.g., linked lists, arrays, etc.), as well as added and deleted by the programmer. Since we do not model such operations in the extended MX semantics, we introduce a stack information abstraction allowing to retrieve from the memory all available pairs of stack base addresses and maximal sizes every time they are needed. For the application of the semantics for particular cases one has to instantiate this abstraction wrt. its implementation.

Definition 7.6 (**Stack Information Abstraction**).

- *Base address of the stack information* in the memory is a starting address of the first stack information region and given by the constant
$$\mathit{StIba} \in \mathbb{B}^{32}$$
- The address of the next stack information region is computed by the uninterpreted function
$$\mathit{StIba}_{next}(a, m) \in \mathbb{B}^{32} \cup \{\bot\}$$
on the base of the memory configuration $m \in \mathbb{C}_m$ and a base address $a \in \mathbb{B}^{32}$ of a given stack information region. If the next region does not exist, the function returns $\bot$.
- The partial functions retrieving the stack base address and maximal stack size from a given stack information region identified by its base address are modeled as
$$\mathit{sba}(a, m) \in \mathbb{B}^{32} \qquad \mathit{mss}(a, m) \in \mathbb{N}$$
- Additionally to these parameters we introduce the constant
$$\mathit{StIsize} \in \mathbb{N}$$
representing the size of a single stack information region in bytes. This size is not needed for the semantics and will be later used for definition of conditions required for the correctness argumentation. Obviously, we assume that the addresses at which the sba and mss reside lay inside the stack information region.

7.1.4 Base Address and Maximal Size of the Current Stack

In order to find a matching pair of the stack base address and its maximal size we proceed as follows.

Definition 7.7 (Stack Information Computation). First, we recursively collect the stack information starting from a given stack information region identified by an address $a \in \mathbb{B}^{32} \cup \{\bot\}$:

$$StI(a, m) \overset{def}{\equiv} \begin{cases} (\varepsilon, \varepsilon) & : \ a = \bot \\ \big(sbas' \circ sba(a, m),\ msss' \circ mss(a, m)\big) & : \ \text{otherwise} \end{cases}$$

with $(sbas', mmss') \equiv StI\ (StIba_{next}(a, m), m)$.

Obviously, for retrieving the information about all stacks available in the memory m one calculates two sequences $(sbas, msss) = StI(StIba, m)$ and requires that all found stacks in the memory do not overlap by the predicate $valid^{stacks}_{\mathrm{MX}}(|sbas|, sbas, msss)$ (see Section 6.2.1).

Second, if possible we choose the result by comparing a content of the stack and base pointer registers of the MIPS-86 processor core.

Definition 7.8 (Search of Base Address and Maximal Size of the Current Stack). Given values of the stack and base pointers $spv, bpv \in \mathbb{B}^{32}$ residing in the GPRs, the memory $m \in \mathbb{C}_m$, and the base address $StIba$ of the stack information in m, we define the function

$$R_{StI}(spv, bpv, m, StIba) \in \big(\mathbb{B}^{32} \times \mathbb{N}\big) \cup \{\bot\}$$

such that on the base of the stack information $(sbas, msss) = StI(StIba, m)$ and the set of pairs

$$StI^{(sbas,msss)}_{pair}(spv, bpv) \overset{def}{\equiv} \left\{ (sba, mss) \;\middle|\; \begin{array}{l} (\exists i \in \mathbb{N}.\ sba = sbas_i \wedge mss = msss_i) \wedge \\ spv, bpv \in A^{stack}_{\mathrm{MX}}(sba, mss) \wedge \langle spv \rangle \leq \langle bpv \rangle \end{array} \right\}$$

the result is computed as

$$R_{StI}(spv, bpv, m, StIba) \overset{def}{\equiv} \begin{cases} \epsilon\ StI^{(sbas,msss)}_{pair}(spv, bpv) & : \ StI^{(sbas,msss)}_{pair}(spv, bpv) \neq \emptyset \\ \bot & : \ \text{otherwise} \end{cases}$$

One can easily prove that if all found stacks in the memory do not overlap, one can find at most one pair (sba, mss) corresponding to the given stack and base pointers.

Lemma 7.1 (Uniqueness of Found sba and mss). *For any m, $StIba$, spv, bpv from above, and computed non-empty $(sbas, msss) = StI(StIba, m)$ one has*

$$valid^{stacks}_{\mathrm{MX}}(|sbas|, sbas, msss) \implies \#StI^{(sbas,msss)}_{pair}(spv, bpv) \leq 1$$

7.1.5 MX Thread Configuration Reconstruction

Given a single core MIPS-86 configuration $d \in \mathbb{C}_{\mathrm{MIPS}}$, the mixed program $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in Prog_{\mathrm{MX}}$, the parameters $\theta \in Params_{\mathrm{CIL}}$, the compiler information $info = (info_{\mathrm{cil}}, info_\mu) \in infoT_{\mathrm{MX}}$, computed as $info_\mu = cmpl_{\mathrm{MASM}}(\pi_\mu)$ and $info_{\mathrm{cil}} = cmpl_{\mathrm{CIL}}(\pi_{\mathrm{cil}})$, the stack base address $sba \in \mathbb{B}^{32}$ and the code base addresses $cba = (cba_{\mathrm{cil}}, cba_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$ such that

$valid^{code}_{\mathrm{MX}}(\mathit{info}, \mathit{cba})$ holds, one can reconstruct a corresponding well-formed MX thread configuration of type $\mathbb{K}_{\mathrm{MX}}$ according to the MX compiler consistency.

Starting from the code address in the program counter and the frame base address from the general purpose register *bp*, we traverse the physical stack residing in the MIPS-86 memory from its top till we encounter the first its frame on its bottom. So, the reconstruction of the MX thread includes the following steps:

- collect the base addresses of all frames on the stack and determine current locations and functions/procedures which the frames belong to,
- using this control information traverse the stack in the memory and compose the abstract full MX stack,
- transform the MX stack into a sequence of MX execution contexts and compose the MX thread configuration.

Note, such a reconstruction is not always possible, e.g., in case an existing stack in the memory was destroyed, or a system programmer prepared a new stack with a wrong layout. We will indicate the result of the failed reconstruction with $\perp$.

Reconstruction of Control Information

First, we introduce two simple predicates indicating that a given function/procedure is defined in the C-IL or MASM program respectively:

$$\begin{aligned} masmp(p, \pi_\mu) &\overset{def}{\equiv} p \in \mathrm{dom}\,(\pi_\mu) \wedge \neg ext(p, \pi_\mu) \\ cilf(f, \pi_{\mathrm{cil}}, \theta) &\overset{def}{\equiv} f \in \mathrm{dom}\left(\mathcal{F}^{\theta}_{\pi_{\mathrm{cil}}}\right) \wedge \neg ext(f, \pi_{\mathrm{cil}}, \theta) \end{aligned}$$

Definition 7.9 (Code Address in a MASM Procedure). Given a code address $a \in \mathbb{B}^{32}$ (corresponding to the value of the program counter or the return address on the stack) and a pair $(p, loc) \in \mathbb{P}_{name} \times \mathbb{N}$ of the MASM procedure name p and the location loc inside its body we define a predicate indicating whether the address a corresponds to (p, loc):

$$proc^{\pi_\mu}(p, loc, a, info_\mu, cba_\mu) \overset{def}{\equiv} \begin{array}{ll} (i) & masmp(p, \pi_\mu) \\ (ii) & loc \in [1 : |\pi_\mu(p).body|] \\ (iii) & a = ca_{\mathrm{MASM}}\left(p, loc, info_\mu, cba_\mu\right) \end{array}$$

Definition 7.10 (Code Address in a C-IL Function). Analogously, for $(f, loc) \in \mathbb{F}_{name} \times \mathbb{N}$ we define a predicate showing that the code address $a \in \mathbb{B}^{32}$ is either the starting address of the compiled code of a C-IL statement at the location loc in the C-IL function f or the return address pointing to the epilogue of a function call at the location[1] $loc - 1$ in f:

$$func^{\pi_{\mathrm{cil}},\theta}(f, loc, a, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \overset{def}{\equiv} \begin{array}{ll} (i) & cilf(f, \pi_{\mathrm{cil}}, \theta) \\ (ii) & loc \in [1 : |\pi_{\mathrm{cil}}.\mathcal{F}(f).P|] \\ (iii) & a = ca_{\mathrm{CIL}}\,(f, loc, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \vee \\ & a = rca_{\mathrm{CIL}}\,(f, loc - 1, info_{\mathrm{cil}}, cba_{\mathrm{cil}}) \end{array}$$

[1]Recall that the location *loc* of non-topmost C-IL frames points to the next statement after the function call.

Definition 7.11 (Code Address in a Function/Procedure Compiled Code). Combining both definitions for $(f, loc) \in (\mathbb{F}_{name} \cup \mathbb{P}_{name}) \times \mathbb{N}$, *info*, *cba* from above, and a mixed machine program π we simply get

$$match^{\pi,\theta}_{ctrl}(f, loc, a, info, cba) \overset{def}{\equiv} proc^{\pi_\mu}(f, loc, a, info_\mu, cba_\mu) \vee func^{\pi_{\text{cil}},\theta}(f, loc, a, info_{\text{cil}}, cba_{\text{cil}})$$

In order to stop traversing the stack, we have to detect whether a frame base address points to the bottom frame on the stack.

Definition 7.12 (Bottom Frame on C-IL Stack). Given the stack base address $sba \in \mathbb{B}^{32}$, and a base address $ba \in \mathbb{B}^{32}$ of a stack frame of a non-external C-IL function $f \in \mathbb{F}_{name}$, we can test whether this frame is the bottom frame on the stack by the following comparison based on the physical layout of the C-IL stack:

$$botfr^{\pi_{\text{cil}},\theta}_{\text{CIL}}(sba, ba, f) \overset{def}{\equiv} \langle ba \rangle + size^{\pi_{\text{cil}},\theta}_{pars}(f) + 4 \cdot 2 - 1 = \langle sba \rangle$$

Definition 7.13 (Bottom Frame on MASM Stack). Analogously, for a non-external MASM procedure $p \in \mathbb{P}_{name}$ we define

$$botfr^{\pi_\mu}_{\text{MASM}}(sba, ba, p) \overset{def}{\equiv} \langle ba \rangle + 4 \cdot \pi_\mu(p).npar + 4 \cdot 2 - 1 = \langle sba \rangle$$

Finally, we can introduce a recursive function collecting the frame base addresses and simultaneously reconstructing the control information for every frame.

Definition 7.14 (Control Information Reconstruction Function). For the MIPS-86 memory $m \in \mathbb{C}_m$, frame base address $ba \in \mathbb{B}^{32}$, and a compiled code address $ca \in \mathbb{B}^{32}$ along with *info*, *cba*, *sba* from above, we define the function

$$R^{\pi,\theta}_{ctrl}(m, info, cba, sba, ca, ba) \in \left(\left(\mathbb{B}^{32}\right)^* \times \left((\mathbb{F}_{name} \cup \mathbb{P}_{name}) \times \mathbb{N}\right)^* \right) \cup \{\bot\}$$

such that using the shorthands

$$Ctrl \equiv Ctrl^{\pi,\theta}(ca, info, cba) \overset{def}{\equiv} \left\{ (f', loc') \,\middle|\, match^{\pi,\theta}_{ctrl}(f', loc', ca, info, cba) \right\}$$

$$ctrl' \equiv R^{\pi,\theta}_{ctrl}(m, info, cba, sba, ra(m, ba), pbp(m, ba))$$

$$bfr_{\text{cil}} \equiv botfr^{\pi_{\text{cil}},\theta}_{\text{CIL}}(sba, ba, f) \qquad bfr_\mu \equiv botfr^{\pi_\mu}_{\text{MASM}}(sba, ba, f)$$

the result is recursively computed as

$$R^{\pi,\theta}_{ctrl}(m, info, cba, sba, ca, ba) \overset{def}{\equiv}$$
$$\begin{cases} (ba,\ (f, loc)) & : (f, loc) = \epsilon\ Ctrl \wedge \\ & \quad (masmp(f, \pi_\mu) \implies bfr_\mu) \wedge \\ & \quad (cilf(f, \pi_{\text{cil}}, \theta) \implies bfr_{\text{cil}}) \\ \left(bas' \circ ba,\ flocs' \circ (f, loc)\right) & : (f, loc) = \epsilon\ Ctrl \wedge \\ & \quad (masmp(f, \pi_\mu) \implies \neg bfr_\mu) \wedge \\ & \quad (cilf(f, \pi_{\text{cil}}, \theta) \implies \neg bfr_{\text{cil}}) \wedge \\ & \quad ctrl' = (bas', flocs') \\ \bot & : \text{otherwise} \end{cases}$$

Therefore, for a given configuration $d \in \mathbb{C}_{\text{MIPS}}$ we start the reconstruction of the control information from the values of the program counter and the base pointer in the GPRs:

$$R^{\pi,\theta}_{ctrl}(d, info, cba, sba) \overset{def}{\equiv} R^{\pi,\theta}_{ctrl}(d.m, info, cba, sba, d.cpu.core.pc, d.cpu.core.gpr(bp))$$

Obviously, the reconstruction of the control information is unique if for any code address we can find at most one pair (f, loc). We state this requirement in the following lemma:

Lemma 7.2 (Unique Control Information Reconstruction).

$$valid_{\mathrm{MX}}^{code}(info, cba) \implies \forall ca \in A_{\mathrm{MX}}^{code}(info, cba).\ \#Ctrl^{\pi,\theta}(ca, info, cba) \leq 1$$

The proof of this lemma is out of the scope of this thesis and can be made in the presence of implementation details of the C-IL and MASM compiler. In fact, it should be easy to show that for any address pointing to a MIPS-86 instruction inside the compiled code of a MASM instruction or a C-IL statement except the function/procedure call, the set $Ctrl^{\pi,\theta}(ca, info, cba)$ is empty. Otherwise, we consider either a starting address of an instruction/statement or a return address pointing to the caller part of the epilogue, and find the only one (f, loc) corresponding to this code address.

Definition 7.15 (Valid Reconstructed Control Information). For a reconstructed list $flocs \in (\mathbb{F}_{name} \cup \mathbb{P}_{name}) \times \mathbb{N})^+$ we define the predicate $valid_{R_{ctrl}}^{\pi,\theta}(flocs) \in \mathbb{B}$ indicating whether $flocs$ is valid for the reconstruction of a corresponding MX thread configuration. Let the shorthands be $(f, loc) \equiv flocs_{i-1}$, $(f', loc') \equiv flocs_i$, $stmt_{\mathrm{cil}} \equiv \mathcal{F}_{\pi_{\mathrm{cil}}}^{\theta}(f).P[loc-1]$, $inst_\mu \equiv \pi_\mu(f).body[loc-1]$. Then we define $valid_{R_{ctrl}}(flocs)$ such that it requires: (i) if f is a C-IL function, then at the location $loc-1$ it must contain a call of a function of the same type as f' [2], (ii) if f is a MASM procedure, then f' is called explicitly at the same location, and (iii) for the C-IL function f the return value type in $stmt$ is the same as the return type of the C-IL function f':

$$\begin{aligned}
valid_{R_{ctrl}}^{\pi,\theta}(flocs) &\overset{def}{\equiv} \forall i \in [2 : |flocs|].\\
&\textit{(i)}\quad cilf(f, \pi_{\mathrm{cil}}, \theta) \implies stmt_{\mathrm{cil}} \in \{e_0 = \mathbf{call}\ e(E), \mathbf{call}\ e(E)\} \wedge \tau_f^{\pi_{\mathrm{cil}},\theta}(e) = \tau_{fun}^{\mathcal{F}_{\pi_{\mathrm{cil}}}^{\theta}}(f')\\
&\textit{(ii)}\quad masmp(f, \pi_\mu) \implies inst_\mu = \mathbf{call}\ f'\\
&\textit{(iii)}\quad cilf(f, \pi_{\mathrm{cil}}, \theta) \wedge stmt_{\mathrm{cil}} = (e_0 = \mathbf{call}\ f'(E)) \wedge\\
&\qquad t_{rds} \in \{\mathbf{ptr}(t), \mathbf{array}(t, n)\} \wedge cilf(f', \pi_{\mathrm{cil}}, \theta) \implies\\
&\qquad t = qt2t\left(\mathcal{F}_{\pi_{\mathrm{cil}}}^{\theta}(f').rettype\right)
\end{aligned}$$

Note that at least these requirements are needed for the reconstruction. Later we will have again the software condition stating that the MX execution from the reconstructed configuration does not cause run-time error. Therefore, all conditions present in the definition of the MX transition function do not need to be checked during the reconstruction.

Reconstruction of the MX Stack

Having the reconstructed control information $(bas, flocs)$ satisfying the predicate $valid_{R_{ctrl}}^{\pi,\theta}(flocs)$, we can proceed with the reconstruction of the full MX stack for the MX thread configuration.

In Definitions 6.6 and 6.17 we have already considered the computation of some stack frame components from the memory. Now, we complete the computation of the missing ones.

[2] In fact, since in C-IL a function/procedure can be called by a function pointer, the only information needed for the call and the later return to work correctly is the exact type of the function pointer which describes the parameters and the return type. Therefore, in case the stack is built by a system programmer and is not a result of the execution of the compiled mixed code, it is possible that the return from f' would be performed to the next statement after $stmt$ even if the call of f' is not present in π.

Definition 7.16 (Reconstruction of rds). Given are a memory $m \in \mathbb{C}_m$ and $ba, ba' \in \mathbb{B}^{32}$, $f \in \mathbb{F}_{name}$, $f' \in \mathbb{F}_{name} \cup \mathbb{P}_{name}$ corresponding to the frame base addresses and function/procedure names from Figure 6.6. Moreover, let the frame of the non-external C-IL function f have an index $i \in \mathbb{N}$ and $loc \in \mathbb{N}$ be a location after the function/procedure call in the body of f, namely for the statement $stmt \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).P[loc-1]$ we have $stmt \in \{e_0 = \textbf{call}\ e(E), \textbf{call}\ e(E)\}$. Then using the shorthands $a \equiv rdsw^{\pi,\theta}(m, ba', f')$ and $t_{rds} \equiv qt2t\left(\tau^{\pi,\theta}_{f}(\&(e_0))\right)$ we define the computation of the C-IL return value destination $rds_{val} \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$ in the frame i by the function

$$rds^{\pi,\theta}(m, ba, f, loc, i, ba', f', info) \overset{def}{\equiv} rds_{val}$$

such that the result depends on the following cases:

- No return value: $stmt = \textbf{call}\ e(E)$

  Then, the result of the reconstruction of obviously computed as

  $$rds_{val} = \bot$$

  Note that the C-IL semantics does not forbid such calls of f' even if the type of the function return value is not **void**.

- Return value address corresponds to a local reference:
  - the C-IL function / MASM procedure is called in the statement with return value

    $$stmt = (e_0 = \textbf{call}\ e(E))$$

  - there exist a pair (v, o) of a local variable/parameter and an offset inside it such that the byte address computed for (v, o) is equal to the destination address a read from the stack.

    Using the shorthands $V_f \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).V$ and $npar_f \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).npar$ we define

    $$\begin{aligned} exist^{\pi_{\text{cil}},\theta}_{lref}(v, o, a, ba, f) \overset{def}{\equiv}\ & \exists\, j \in [1 : |V_f|],\ t \in \mathbb{T}_{\mathbb{Q}}. \\ &\textit{(i)} \quad (v, t) = V_f[j] \\ &\textit{(ii)} \quad o \in [0 : size_{\theta}(qt2t(t)) - 1] \\ &\textit{(iii)} \quad j \leq npar_f \implies para^{\pi_{\text{cil}},\theta}(j, ba, f) +_{32} o_{32} = a \\ &\textit{(iv)} \quad j > npar_f \implies lva(v, ba, f, info_{\text{cil}}) +_{32} o_{32} = a \end{aligned}$$

    and require

    $$(v, o) = \epsilon\left\{(\hat{v}, \hat{o}) \in \mathbb{V} \times \mathbb{N}_0 \,\middle|\, exist^{\pi_{\text{cil}},\theta}_{lref}(\hat{v}, \hat{o}, a, ba, f)\right\}$$

    If the conditions hold, the result of the reconstruction is

    $$rds_{val} = \mathbf{lref}((v, o), i, t_{rds})$$

- Return value address does not correspond to a local reference:
  - the C-IL function / MASM procedure is called in the statement with return value

    $$stmt = (e_0 = \textbf{call}\ e(E))$$

– a local variable/ parameter is not found

$$\left\{(\hat{v}, \hat{o}) \in \mathbb{V} \times \mathbb{N}_0 \;\middle|\; exist^{\pi_{\text{cil}},\theta}_{lref}(\hat{v}, \hat{o}, a, ba, f)\right\} = \emptyset$$

Under the conditions above we get

$$rds_{val} = \mathbf{val}(a, t_{rds})$$

Note that in this case $a \in \mathbb{B}^{32}$ might also be an address inside the code region or of a word on the stack where the local variables and parameters of the frame i do not reside. However, we will not be interested in such reconstructions later on because the software conditions will require that the code and stack regions are not explicitly accessed by the MX machine.

Now, using the following shorthands

- control information $ctrl \equiv R^{\pi,\theta}_{ctrl}(d, info, cba, sba)$
- for non-empty $ctrl = (bas, flocs)$, $top \equiv |bas|$, and $i \in \mathbb{N}_{|ctrl|}$:

$$ba_i \equiv bas_i \qquad (f_i, loc_i) \equiv flocs_i$$

- for a C-IL function f_i:

$$V_i \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f_i).V \qquad npar^{\text{cil}}_i \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f_i).npar \qquad (v_{i,j}, t_{i,j}) \equiv V_i[j]$$
$$r_{i,j} \equiv info_{\text{cil}}.reg_{lvar}\,(v_{i,j}, f_i, loc_i) \qquad crra_{i,j} \equiv crra^{\pi_{\text{cil}},\theta}\,(r_{i,j}, ba_i, f_i, loc_i - 1, info_{\text{cil}})$$

- for a MASM procedure f_i:

$$uses_i \equiv \pi_\mu(f_i).uses \qquad npar^{\mu}_i \equiv \pi_\mu(f_i).npar$$

we can define the reconstruction of the full MX stack.

Definition 7.17 (MX Stack Reconstruction Starting from a Given Frame). For the considered d, $info$, $(bas, flocs)$ such that $flocs$ is valid for the reconstruction, and an index $i \in [0 : top]$ we define the auxiliary stack reconstruction function

$$\hat{R}^{\pi,\theta}_{stack}(d, info, bas, flocs, i) \in (frame_{\text{MASM}} \cup frame_{\text{CIL}})^*$$

$$\hat{R}^{\pi,\theta}_{stack}(d, info, bas, flocs, i) \overset{def}{\equiv} \begin{cases} \varepsilon & : \; i = 0 \\ \hat{R}^{\pi,\theta}_{stack}(d, info, bas, flocs, i-1) \circ frame^i_{\text{mx}} & : \; \text{otherwise} \end{cases}$$

where the frame $frame^i_{\text{mx}}$ is reconstructed as follows:

$$frame^i_{\text{mx}} \equiv \begin{cases} frame^i_{\text{cil}} & : \; cilf(f_i, \pi_{\text{cil}}, \theta) \\ frame^i_{\mu} & : \; masmp(f_i, \pi_\mu) \end{cases}$$

- MASM stack frame $frame^i_\mu$:

$$frame^i_\mu.p = f_i \qquad frame^i_\mu.pars = pars^{\pi_\mu}_{\text{MASM}}\,(d.m, ba_i, f_i)$$
$$frame^i_\mu.loc = loc_i \qquad frame^i_\mu.saved = saved^{\pi_\mu}_{\text{MASM}}\,(d.m, ba_i, f_i)$$

In contrast to the frame components restored above on the base of the procedure information, the size of the *lifo* has to be computed from the actual layout of the stack in the memory:

$$size_{lifo} \equiv \begin{cases} \langle ba_i \rangle - \langle d.cpu.core.gpr(sp) \rangle - 4 \cdot |uses_i| & : i = top \\ dist - size^{\pi_{\text{cil}},\theta}_{pars}(f_{i+1}) & : i < top \wedge cilf(f_{i+1}, \pi_{\text{cil}}, \theta) \\ dist - 4 \cdot npar^{\mu}_{i+1} & : i < top \wedge masmp(f_{i+1}, \pi_{\mu}) \end{cases}$$

where

$$dist \equiv \langle ba_i \rangle - \langle ba_{i+1} \rangle - 4 \cdot |uses_i| - 2 \cdot 4$$

Note that $size_{lifo}$ cannot be correctly computed if the stack is prepared by a programmer in a wrong way. For the time being we ignore such situations and set empty *lifo* for $size_{lifo} < 0$. Later after the stack reconstruction we will introduce another validity condition intended to exclude such issues. So, by now we compute

$$frame^i_{\mu}.lifo = \begin{cases} lifo^{\pi_\mu}_{\text{MASM}}(d.m, ba_i, f_i, size_{lifo}/4) & : size_{lifo} \geq 0 \\ \varepsilon & : \text{otherwise} \end{cases}$$

- C-IL stack frame $frame^i_{\text{cil}}$:

$$frame^i_{\text{cil}}.f = f_i \qquad frame^i_{\text{cil}}.loc = loc_i$$

$$frame^i_{\text{cil}}.rds = \begin{cases} rds^{\pi,\theta}(d.m, ba_i, f_i, loc_i, i, ba_{i+1}, f_{i+1}, info) & : i < top \\ \perp & : \text{otherwise} \end{cases}$$

In order to reconstruct the local memory we simply chose the values for the local variables/parameters $v_{i,j}$ according to the compiler consistency relation:

$$frame^i_{\text{cil}}.\mathcal{M}_{\mathcal{E}}(v_{i,j}) =$$

$$\begin{cases} d.cpu.core.gpr(r_{i,j}) & : i = top \wedge r_{i,j} \neq \perp \\ gpr_{callee}(d.m, ba_{i+1}, f_{i+1})(r_{i,j}) & : i < top \wedge r_{i,j} \in Reg_{callee} \wedge cilf(f_{i+1}, \pi_{\text{cil}}, \theta) \\ saved^{\pi_\mu}_{\text{MASM}}(d.m, ba_{i+1}, f_{i+1})(r_{i,j}) & : i < top \wedge r_{i,j} \in Reg_{callee} \wedge masmp(f_{i+1}, \pi_\mu) \\ d.m_4(crra_{i,j}) & : i < top \wedge r_{i,j} \in Reg_{caller} \\ d.m_{size_\theta(qt2t(t_{i,j}))}\left(para^{\pi_{\text{cil}},\theta}(j, ba_i, f_i)\right) & : r_{i,j} = \perp \wedge j \leq npar^{\text{cil}}_i \\ d.m_{size_\theta(qt2t(t_{i,j}))}\left(lva(v_{i,j}, ba_i, f_i, info_{\text{cil}})\right) & : r_{i,j} = \perp \wedge j > npar^{\text{cil}}_i \end{cases}$$

Definition 7.18 (Full MX Stack Reconstruction). Finally, for the full MX stack reconstruction we define the function wrt. the shorthands introduced above.

$$R^{\pi,\theta}_{stack}(d, info, cba, sba) \in (frame_{\text{MASM}} \cup frame_{\text{CIL}})^* \cup \{\perp\}$$

$$R^{\pi,\theta}_{stack}(d, info, cba, sba) \overset{def}{\equiv} \begin{cases} \hat{R}^{\pi,\theta}_{stack}(d, info, bas, flocs, top) & : ctrl = (bas, flocs) \wedge \\ & \quad valid^{\pi,\theta}_{R_{ctrl}}(flocs) \\ \perp & : \text{otherwise} \end{cases}$$

As we mentioned before, in the reconstruction of frames we mostly rely on function/procedure declarations and the compiler information. The only exception is the computation of *lifo*,

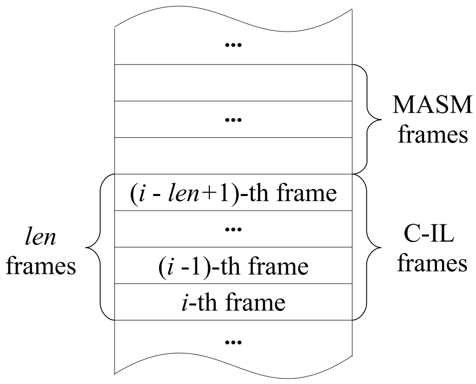

Figure 7.1: C-IL context reconstruction from the full MX stack.

though we have ignored the wrong stack layout yet that could be created by a system programmer. In fact, during the stack reconstruction we have not checked whether each frame of the stack in the memory has a size proper for the further execution of the compiled code. Without such a condition one could get the reconstructed components with values taken from adjacent stack frames. We formulate this validity condition using the earlier introduced notion of the distance between frame base addresses.

Definition 7.19 (Valid Reconstruction of MX Stack). We call a reconstructed MX stack $st = R^{\pi,\theta}_{stack}(d, info, cba, sba)$ with $st \neq \bot$ valid wrt. the valid control information $(bas, flocs) = R^{\pi,\theta}_{ctrl}(d, info, cba, sba)$ iff the following predicate holds:

$$valid^{\pi,\theta}_{R_{stack}}(st, bas, d, info) \overset{def}{\equiv} \forall i \in \mathbb{N}_{top}.$$
$$dist^{\pi,\theta}_{\mathrm{MX}}(st, i, info) = \begin{cases} \langle ba_i \rangle - \langle ba_{i+1} \rangle & : \ i < top \\ \langle ba_i \rangle - \langle d.cpu.core.gpr(sp) \rangle & : \ \text{otherwise} \end{cases}$$

Reconstruction of Execution Contexts and MX Thread Configuration

In order to reconstruct the sequence of the MX execution contexts on the base of the reconstructed stack we introduce an auxiliary recursive function traversing the stack from a frame with a given index.

Definition 7.20 (Reconstruction of the MX Contexts Starting from a Given Frame). Given a reconstructed non-empty MX stack $st \in (frame_{\mathrm{MASM}} \cup frame_{\mathrm{CIL}})^*$, a sequence of reconstructed frame base addresses $bas \in (\mathbb{B}^{32})^*$, a memory $m \in \mathbb{C}_m$, general purpose registers $gpr : \mathbb{B}^8 \to \mathbb{B}^{32}$, and an index $i \in [0 : top(st)]$ of the frame from which the reconstruction is performed, we define the partial function

$$\hat{R}^{\pi,\theta}_{cntx}(st, bas, m, gpr, i) \in \left(context_{\mathrm{CIL}} \cup context^{active}_{\mathrm{MASM}} \cup context^{inactive}_{\mathrm{MASM}}\right)^*$$

such that using the shorthands:

- the C-IL or MASM stack of an individual context:

$$s \equiv st[i - len + 1 : i]$$

such that all its adjacent frames are of the same type, what is taken into account in the computation of its length (see example for a C-IL frame with the index i on Figure 7.1)

$$len \equiv max \left\{ l \in \mathbb{N}_i \;\middle|\; \begin{array}{r} \forall j \in [i-l+1 : i].\ (cil(st,i) \implies cil(st,j)) \wedge \\ (masm(st,i) \implies masm(st,j)) \end{array} \right\}$$

- the sequence of reconstructed contexts starting from the frame $i - len$:

$$k \equiv \hat{R}_{cntx}(st, bas, m, gpr, i - len)$$

the result is computed as

$$\hat{R}^{\pi,\theta}_{cntx}(st, bas, m, gpr, i) \overset{def}{\equiv} \begin{cases} \varepsilon & : i = 0 \\ k \circ s & : i > 0 \wedge cil(st,i) \\ k \circ (s,\ gpr') & : i = top(st) \wedge masm(st,i) \\ k \circ (s,\ gpr'') & : 0 < i < top(st) \wedge \\ & \quad masm(st,i) \wedge cil(st, i+1) \\ undefined & : \text{otherwise} \end{cases}$$

where the GPRs of active and inactive MASM contexts are

$$gpr' \equiv gpr|_{\mathbb{B}^8 \setminus \{sp,bp,ra\}} \qquad gpr'' \equiv gpr_{callee}{}^{\pi_{\text{cil}},\theta}(m, bas_{i+1}, f_{i+1}(st))$$

Definition 7.21 (Full Reconstruction of MX Execution Contexts). Hence, using the definition above we reconstruct the whole sequence of the MX execution contexts by the function

$$R^{\pi,\theta}_{cntx}(d, info, cba, sba) \in \left(context_{\text{CIL}} \cup context^{active}_{\text{MASM}} \cup context^{inactive}_{\text{MASM}}\right)^* \cup \{\bot\}$$

defined for the shorthands

$$\begin{aligned} ctrl &\equiv R^{\pi,\theta}_{ctrl}(d, info, cba, sba) \\ st &\equiv R^{\pi,\theta}_{stack}(d, info, cba, sba) \\ k &\equiv \hat{R}^{\pi,\theta}_{cntx}(st, bas, d.m, d.cpu.core.gpr, top(st)) \end{aligned}$$

as follows

$$R^{\pi,\theta}_{cntx}(d, info, cba, sba) \overset{def}{\equiv} \begin{cases} k & : st \neq \bot \wedge ctrl = (bas, flocs) \wedge \\ & \quad valid^{\pi,\theta}_{R_{stack}}(st, bas, d, info) \\ \bot & : \text{otherwise} \end{cases}$$

As the last step of the reconstruction, we easily construct the MX thread configuration if possible. Obviously, the last element in the computed sequence of contexts correspond to the active execution context.

Definition 7.22 (Reconstruction of the MX Thread Configuration). The function for the MX thread configuration reconstruction is defined as

$$R^{\pi,\theta}_{\mathbb{K}_{\text{MX}}}(d, info, cba, sba) \in \mathbb{K}_{\text{MX}\bot}$$

$$R^{\pi,\theta}_{\mathbb{K}_{\text{MX}}}(d, info, cba, sba) \overset{def}{\equiv} \begin{cases} (ac,\ ic,\ d.cpu.core.spr) & : R^{\pi,\theta}_{cntx}(d, info, cba, sba) = ic \circ ac \\ \bot & : \text{otherwise} \end{cases}$$

such that the active context is $ac \in context_{\text{CIL}} \cup context^{active}_{\text{MASM}}$, and the sub-sequence ic corresponds to the list of inactive contexts and can be empty.

7.1.6 Transition Function

In order to define the transition function for the extended mixed machine, one should know when the MXA machine starts the inline assembly, and when it should be possible to return back to a corresponding MX thread configuration with the abstract stack.

Definition 7.23 (Start of Inline Assembly). The predicate $start_{isa}^{\pi}(c)$ indicates that the MXA machine with a configuration $c \in \mathbb{C}_{\mathrm{MXA}}$ is about to start execution of inline assembly in the program π:

$$start_{isa}^{\pi}(c) \overset{def}{\equiv} mx(c.k) \wedge masm(c_{\mathrm{mx}}.ac) \wedge \exists il \in \mathbb{I}_{\mathrm{ASM}}^{+}.\ instr_{next}(c_{\mu}, \pi_{\mu}) = \mathbf{asm}\{il\}$$

where the shorthand are $c_{\mathrm{mx}} \equiv conf_{\mathrm{MX}}^{\mathrm{MXA}}(c)$ and $c_{\mu} \equiv conf_{\mathrm{MASM}}(c_{\mathrm{mx}})$.

Definition 7.24 (End of ISA Steps). Conversely, one can attempt to apply the reconstruction for the MXA machine with an MXA thread $k \in \mathbb{K}_{\mathrm{MXA}}$ represented by the MIPS-86 processor configuration if the following predicate holds:

$$end_{isa}^{\pi,\theta,cba}(k) \overset{def}{\equiv} isa(k) \wedge \neg mode(k.core) \wedge cp_{r\mathrm{MIPS}}^{\mathrm{MX}}(k.core, \pi, info, \theta, cba)$$

Obviously, we will use here and further in this section the information $info = (info_{\mathrm{cil}}, info_{\mu}) \in infoT_{\mathrm{MX}}$, computed as before, i.e, $info_{\mu} = cmpl_{\mathrm{MASM}}(\pi_{\mu})$ and $info_{\mathrm{cil}} = cmpl_{\mathrm{CIL}}(\pi_{\mathrm{cil}})$.

Note that as we have seen before in the definition of the reconstruction, $end_{isa}^{\pi,\theta,cba}(k)$ only says when we can apply it. Though we may reach a consistency point, it is not always possible to return to the abstract MX thread configuration and one needs to continue the ISA execution until one reaches another consistency point at which the reconstruction is possible. This issue and the solution were discovered by the author of this thesis in the time when the original version of C0 semantics with inline assembly [PBLS15] based on the reconstruction did not take this fact into account and, therefore, had to be corrected later.

Definition 7.25 (Stack Addresses Occupied by the Abstract Stack). Additionally, for a given MXA configuration $c \in \mathbb{C}_{\mathrm{MXA}}$ such that $mx(c.k)$ holds, we define a set of memory addresses that actually should be occupied by the abstract stack of the configuration c:

$$A_{stack}^{\pi,\theta}(c, info, sba) \overset{def}{\equiv} \{sp_{top}\}_{\langle sba\rangle - \langle sp_{top}\rangle + 1}$$

witch $sp_{top} \equiv sp_{top}^{\pi,\theta}(st_{\mathrm{MX}}(c_{\mathrm{mx}}), info, sba)$ and $c_{\mathrm{mx}} \equiv conf_{\mathrm{MX}}^{\mathrm{MXA}}(c)$.

Definition 7.26 (Sequential MXA Transition Function). Finally, for a given MX program $\pi \in Prog_{\mathrm{MX}}$ with inline assembly such that $\pi = (\pi_{\mu}, \pi_{\mathrm{cil}})$, the environment parameters $\theta \in Params_{\mathrm{CIL}}$, the code base addresses $cba = (cba_{\mathrm{cil}}, cba_{\mu}) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, and the base address of the stack information $StIba \in \mathbb{B}^{32}$, combined together into $\iota = (cba, StIba)$, the transitions of the sequential extended mixed machine are defined by the partial function

$$\delta_{\mathrm{MXA}}^{\pi,\theta,\iota} : \mathbb{C}_{\mathrm{MXA}} \times \Sigma_{\mathrm{MXA}} \rightharpoonup \mathbb{C}_{\mathrm{MXA}\perp}$$

where $\mathbb{C}_{\mathrm{MXA}\perp} \overset{def}{\equiv} \mathbb{C}_{\mathrm{MXA}} \cup \{\perp\}$ contains the error state, coming from the mixed machine semantics. The input alphabet

$$\Sigma_{\mathrm{MXA}} \overset{def}{\equiv} \Sigma_{\mathrm{MX}} \cup \Sigma_{\mathrm{A}} \cup \Sigma_{\mathrm{MIPS}}$$

$$\Sigma_{\mathrm{A}} \overset{def}{\equiv} \big(pc \in \mathbb{B}^{32},\ gpr : \{sp, bp, ra\} \to \mathbb{B}^{32},\ m_{st} : \mathbb{B}^{32} \rightharpoonup \mathbb{B}^{8}\big)$$

includes the inputs needed for the MX machine step, or the single core MIPS-86 ISA, or, in case of switching to inline assembly, the parts of MIPS-86 configuration not modeled in the mixed machine configuration.

Definition 7.27. For constructing a single core MIPS-86 machine configuration from $c \in \mathbb{C}_{\mathrm{MXA}}$ with $start^{\pi}_{isa}(c)$ holding in c, and $in \in \Sigma_{\mathrm{A}}$ we define the function $conf^{start}_{r\mathrm{MIPS}}(c, in) \in \mathbb{C}_{\mathrm{MIPS}}$ such that

$$conf^{start}_{r\mathrm{MIPS}}(c, in) \overset{def}{\equiv} d$$

$$d.cpu.sb = \varepsilon \qquad d.cpu.tlb = c.k.tlb$$

$$d.cpu.core = (in.pc,\ gpr,\ c.k.k_{\mathrm{mx}}.spr)$$

$$gpr(r) = \begin{cases} c.k.k_{\mathrm{mx}}.ac.gpr(r) & : r \notin \{sp, bp, ra\} \\ in.gpr(r) & : \text{otherwise} \end{cases}$$

$$d.m(a) = \begin{cases} c.\mathcal{M}(a) & : a \notin \mathrm{dom}\,(in.m_{st}) \\ in.m_{st}(a) & : \text{otherwise} \end{cases}$$

Then, depending on the MXA machine configuration, for $c \in \mathbb{C}_{\mathrm{MXA}}$, $in \in \Sigma_{\mathrm{MXA}}$ we consider:

- Mixed machine step:
  $$mx(c.k) \wedge in \in \Sigma_{\mathrm{MX}} \wedge \neg start^{\pi}_{isa}(c)$$
  We perform the step of the corresponding MX machine
  $$c_{\mathrm{mx}} \equiv conf^{\mathrm{MXA}}_{\mathrm{MX}}(c) \qquad c'_{\mathrm{mx}} \equiv \delta^{\pi,\theta}_{\mathrm{MX}}(c_{\mathrm{mx}}, in)$$
  and compose the new configuration of the MX thread if $c'_{\mathrm{mx}} \neq \bot$:
  $$k'_{\mathrm{mx}} \equiv (c'_{\mathrm{mx}}.ac,\ c'_{\mathrm{mx}}.ic,\ c'_{\mathrm{mx}}.spr)$$
  Hence, the result of the MXA machine step is defined as
  $$\delta^{\pi,\theta,\iota}_{\mathrm{MXA}}(c, in) \overset{def}{\equiv} \begin{cases} \big(c.k[k_{\mathrm{mx}} := k'_{\mathrm{mx}}],\ c'_{\mathrm{mx}}.\mathcal{M}\big) & : c'_{\mathrm{mx}} \neq \bot \\ \bot & : \text{otherwise} \end{cases}$$
- Switch to inline assembly:
  - inline assembly code is to be executed: $start^{\pi}_{isa}(c)$
  - the input is $in \in \Sigma_{\mathrm{A}}$
  - the input memory corresponds to the actual stack region occupied by the abstract stack
    $$\mathrm{dom}\,(in.m_{st}) = A^{\pi,\theta}_{stack}(c, info, c.k.sba)$$
  - The input parameters for the step satisfy the compiler consistency, namely, the predicate $suit^{\pi,\theta,cba}_{start}(c, in, info)$ defined using $d \equiv conf^{start}_{r\mathrm{MIPS}}(c, in)$ and $c_{\mathrm{mx}} \equiv conf^{\mathrm{MXA}}_{\mathrm{MX}}(c)$ holds:
    $$suit^{\pi,\theta,cba}_{start}(c, in, info) \overset{def}{\equiv}$$
    $$\begin{array}{ll} (i) & consis^{ctrl}_{\mathrm{MX}}(c_{\mathrm{mx}}, d, \pi, \theta, info, cba, c.k.sba) \\ (ii) & consis^{bp/sp}_{\mathrm{MX}}(c_{\mathrm{mx}}, d, \pi, \theta, info, c.k.sba) \\ (iii) & consis^{reg}_{\mathrm{MX}}(c_{\mathrm{mx}}, d, \pi, \theta, info, c.k.sba) \\ (iv) & consis^{stack}_{\mathrm{MX}}(c_{\mathrm{mx}}, d, \pi, \theta, info, c.k.sba) \end{array}$$

Note that the input is not unique because of the general purpose register ra not covered by the MX compiler consistency and temporaries on the physical C-IL stack which are purely used by the compiler and have no counterparts in the abstract C-IL configuration. Since values of ra and temporaries are non-deterministic here, any program using the MXA semantics has to be written in a way such that it works for any of them.

In order to define the result of the MXA step in this case we first compute the result of the execution of the first inline assembly instruction

$$d' \equiv \delta_{r\text{MIPS}}\left(d, \left(\texttt{core}, w_I, w_D, 0^{256}\right)\right)$$

where $w_I = w_D = \epsilon\ \mathbb{C}_{walk}$ are just ignored in the transition function.

Since the inline assembly may contain a single instruction or a jump to a compiled code where the reconstruction might be possible, namely $end_{isa}^{\pi,\theta,cba}(d'.cpu)$ holds, we first try to find

– a new pair of the stack base address and stack maximal size

$$StI' \equiv R_{StI}\,(gpr'(sp), gpr'(bp), d'.m, StIba)$$

with $gpr' \equiv d'.cpu.core.gpr$.

– a reconstructed MX thread configuration for $StI' = (sba', mss')$

$$k'_{\text{mx}} \equiv R^{\pi,\theta}_{\mathbb{K}_{\text{MX}}}(d', info, cba, sba')$$

– the new MXA thread configuration for $k'_{\text{mx}} \neq \bot$

$$k' \equiv (k'_{\text{mx}},\ sba',\ mss',\ d'.cpu.tlb)$$

Hence, the transition function for this case is defined as

$$\delta^{\pi,\theta,\iota}_{\text{MXA}}(c, in) \overset{def}{\equiv} \begin{cases} (k',\ d'.m) & : end_{isa}^{\pi,\theta,cba}(d'.cpu) \wedge \\ & \quad StI' = (sba', mss') \wedge k'_{\text{mx}} \neq \bot \\ (d'.cpu,\ d'.m) & : \text{otherwise} \end{cases} \tag{7.1}$$

- Inline assembly, compiled code, or guest/process step:
  - the MXA thread is represented by the ISA configuration: $isa(c.k)$
  - the input matches the step:

$$in \in \Sigma_{\text{MIPS}} \wedge (in = (\texttt{core}, w_I, w_D, eev) \implies \neg eev[0])$$

  - the single core MIPS-86 transition is defined for some d':

$$\delta_{r\text{MIPS}}\left(conf^{\text{MXA}}_{r\text{MIPS}}(c), in\right) = d'$$

Then, using the shorthands StI', k'_{mx}, and k' from above, we define $\delta^{\pi,\theta,\iota}_{\text{MXA}}(c, in)$ by the same equation (7.1).

In all other cases the transition function $\delta^{\pi,\theta,\iota}_{\text{MXA}}(c, in)$ is undefined.

7.2 Concurrent Extended Mixed Machine Semantics

Analogously to the concurrent MX semantics in Section 5.4, we define the concurrent MXA machine.

Definition 7.28 (Configuration of Concurrent MXA Machine). Configurations of the concurrent extended mixed machine with $nt \in \mathbb{N}$ MXA threads are defined by the set

$$\mathbb{C}_{c\text{MXA}} \overset{def}{\equiv} \left(k : \mathbb{N}_{nt} \to \mathbb{K}_{\text{MXA}},\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^{8}\right)$$

Definition 7.29 (Stack Information Abstraction for Concurrent MXA Machine).

- For every MXA thread with index $\mathbb{N}_{nt}$ we consider a distinct base address of the corresponding stack information, namely, we have the sequence

$$StIbas \in \left(\mathbb{B}^{32}\right)^{nt}$$

- All other uninterpreted functions $StIba_{next}$, sba, mss and the constant $StIsize$ treated as parameters for the MXA model are common for all threads.

Definition 7.30 (Sequential MXA Configuration from Concurrent MXA). For a given concurrent MXA configuration $c \in \mathbb{C}_{c\text{MXA}}$ and $t \in \mathbb{N}_{nt}$ we define $conf_{\text{MXA}}(c,t) \in \mathbb{C}_{\text{MXA}}$ such that

$$conf_{\text{MXA}}(c,t) \overset{def}{\equiv} (c.k(t), c.\mathcal{M})$$

Definition 7.31 (Concurrent MXA Transition Function). Now, for a given MX program $\pi \in Prog_{\text{MX}}$ with inline assembly, the environment parameters $\theta \in Params_{\text{CIL}}$, and code base addresses $cba = (cba_{\text{cil}}, cba_{\mu}) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$ determined by the linker placing the compiled program π into the memory, the transitions of the concurrent extended mixed machine are defined by the function

$$\delta^{\pi,\theta,cba}_{c\text{MXA}} : \mathbb{C}_{c\text{MXA}} \times \mathbb{N}_{nt} \times \Sigma_{\text{MXA}} \rightharpoonup \mathbb{C}_{c\text{MXA}\perp}$$

such that for a configuration $c \in \mathbb{C}_{c\text{MXA}}$, an index $t \in \mathbb{N}_{nt}$ of an MXA thread performing a step, and an input $in \in \Sigma_{\text{MXA}}$, the result of the transition is defined as

$$\delta^{\pi,\theta,cba}_{c\text{MXA}}(c,t,in) \overset{def}{\equiv} \begin{cases} (c.k[t \mapsto c'_t.k],\ c'_t.\mathcal{M}) & : c'_t \neq \perp \\ \perp & : \text{otherwise} \end{cases}$$

where the next sequential MXA configuration c'_t is computed for $\iota_t \equiv (cba, StIbas_t)$ as

$$c'_t \equiv \delta^{\pi,\theta,\iota_t}_{\text{MXA}}\left(conf_{\text{MXA}}(c,t), in\right)$$

If c'_t does not exist (recall that the sequential MXA function is partial), then the result of the step $\delta^{\pi,\theta,cba}_{c\text{MXA}}(c,t,in)$ is undefined.

Obviously, in the semantics we require that any stepping MXA thread t has a well-formed configuration:

$$wfconf^{\pi,\theta,cba}_{\text{MXA}}\left(conf_{\text{MXA}}(c,t)\right)$$

7.3 Compiler Correctness for the Sequential Extended Mixed Machine

Similarly to Chapter 6 we proceed with the justification of the MXA machine encoded by the SB reduced MIPS-86. First, one has to consider a sequential case adapted for further concurrent context.

7.3.1 Sequential Simulation Relation

Since the MXA machine performs either MX, or MIPS-86 steps, the simulation relation is either the MX compiler consistency, or the coupling of MIPS-86 components such that the compiled code resides in the memory. Again, in order to apply this simulation in the presence of environment steps, we have to exclude the region of possibly inconsistent memory. Recall, that he memory inconsistency might be caused by MX steps of another core that has not reached an MX consistency point, or by writing to a store buffer of another processor in the non-reduced ISA machine.

Definition 7.32. For any sequential MXA machine configuration $c \in \mathbb{C}_{\text{MXA}}$, a SB reduced MIPS-86 configuration $d \in \mathbb{C}_{\text{MIPS}}$, code base addresses $cba = (cba_{\text{cil}}, cba_{\mu}) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, compiler information $info = (info_{\text{cil}}, info_{\mu}) \in infoT_{\text{MX}}$, and memory addresses $icm \subset A_{hyp} \setminus A^{code}_{\text{MX}}(info, cba)$ we define the sequential MXA simulation relation as

$$consis_{\text{MXA}}(c, d, \pi, \theta, info, cba, icm) \overset{def}{\equiv}$$

$$\begin{aligned} &(i) \quad mx(c.k) \implies consis_{\text{MX}}\left(conf^{\text{MXA}}_{\text{MX}}(c), d, \pi, \theta, info, cba, c.k.sba, c.k.mss, icm\right) \\ &(ii) \quad isa(c.k) \implies d.cpu = c.k \wedge \forall a \in \mathbb{B}^{32} \setminus icm.\ d.m(a) = c.\mathcal{M}(a) \end{aligned}$$

7.3.2 Consistency Points

In order to define the consistency points for the MXA machine, and the SB reduced MIPS-86 executing its compiled code, we first define the predicates indicating which code is executed: compiled code of hypervisor / OS kernel MXA program, inline assembly portion of this code, or some compiled code not belonging to the considered MXA program. In the last case the code may belong to user processes / guests, or to compiled libraries used by hypervisor / OS kernel.

Definition 7.33 (Addresses of Instructions in Inline Assembly). The set of addresses of instructions in inline assembly portions of a MASM program π_{μ} which compiled code contained in $info_{\mu}$ is placed in the memory at the code base address cba_{μ} is computed by the function

$$A^{inline}_{\text{MASM}}(\pi_{\mu}, info_{\mu}, cba_{\mu}) \overset{def}{\equiv} \left\{ adr_{loc,i} \;\middle|\; \begin{array}{l} p \in \text{dom}(\pi_{\mu}) \wedge \\ \neg ext(p, \pi_{\mu}) \wedge \\ loc \in [1 : |\pi_{\mu}(p).body|] \wedge \\ \pi_{\mu}(p).body[loc] = \textbf{asm}\{il\} \wedge \\ i \in [1 : |il| - 1] \end{array} \right\}$$

where $adr_{loc,i} \equiv ca_{\text{MASM}}(p, loc, info_{\mu}, cba_{\mu}) +_{32} (4 \cdot i)_{32}$ is an address of an instruction in the inline assembly sequence il.

Note here that we exclude the address of the first instruction in il because it corresponds exactly to the starting addressed of the compiled $\textbf{asm}\{il\}$.

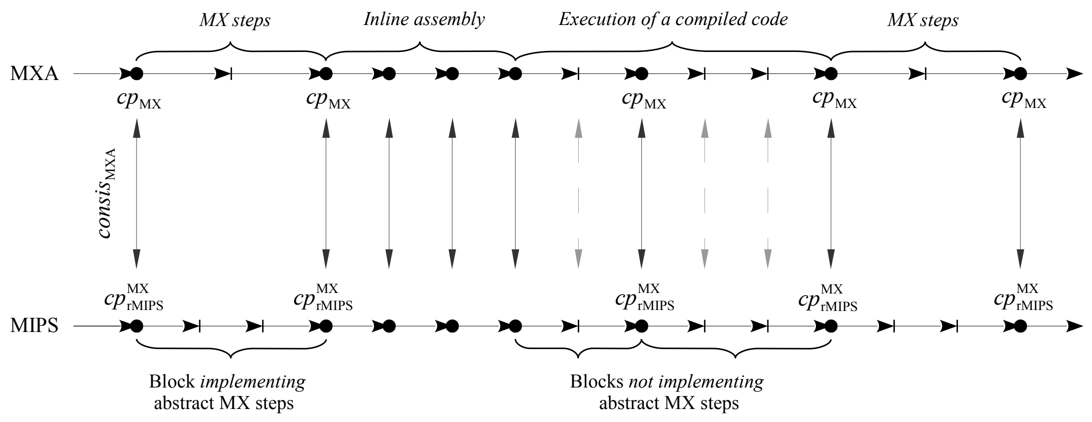
MX steps
Inline assembly
Execution of a compiled code
MX steps
MXA
cp_{MX}
$consis_{\text{MXA}}$
$cp^{\text{MX}}_{\text{rMIPS}}$
MIPS
Block *implementing* abstract MX steps
Blocks *not implementing* abstract MX steps

Figure 7.2: Consistency points for MXA machine and MIPS implementing it. The dots depict the consistency points chosen for both machines such that the unnamed points are additional ones along with those inserted by the MX compiler. The arrows between the machines represent the consistency relation where the shaded dashed ones denote that the coupling holds but it is out of our interest.

Definition 7.34 (Processor Runs Inline Assembly). Then, we define that the processor core with a configuration $core \in \mathbb{C}_{core}$ executes inline assembly by

$$inline(core, \pi_\mu, info_\mu, cba_\mu) \overset{def}{\equiv} core.pc \in A^{inline}_{\text{MASM}}(\pi_\mu, info_\mu, cba_\mu)$$

Definition 7.35 (Processor Runs Inside/Outside Hypervisor/OS Kernel Code). Similarly, we test whether the core runs the compiled code of hypervisor / OS kernel MXA program or not by the predicates

$$inside(core, info, cba) \overset{def}{\equiv} core.pc \in A^{code}_{\text{MX}}(info, cba)$$
$$outside(core, info, cba) \overset{def}{\equiv} \neg inside(core, info, cba)$$

Now, we can choose consistency points for the extended mixed machine. These consistency points will be treated as interleaving points when we apply the order reduction for the arbitrarily interleaved steps of MIPS implementing our MXA machine and consider the concurrent simulation. On first sight, when the extended machine makes MX steps, one would like to consider the MX consistency points. In all other cases, since we would be able to couple the MIPS configuration present in the MXA machine with the one executing the code, one might wish to have this consistency in every step. A corresponding computation of consistency points must be also applied on the MIPS-86 level. However, things are more complex.

In fact, reaching an MX consistency point does not guarantee that we manage to reconstruct a corresponding MX thread configuration. A typical case will be considered later when we switch to a newly created stack where a return address residing on the stack is the starting address of a callee's epilogues instead of a caller's prologue. In this case one continues MIPS-86 ISA steps until we reach another consistency point and try to apply the reconstruction again (see the execution of the compiled code on Figure 7.2).

If one would like to have a consistency point at every step in both machines during the execution of the compiled code not implementing abstract MX steps, it would be possible to detect it from the MXA thread configuration, but not in the implementing MIPS-86 machine. The reason is that one would need to distinguish between steps in the blocks implementing and not implementing MX steps. In contrast to the latter case, in the former case we have to consider only the points inserted by the compiler. In order to detect the case purely on the MIPS-86 level, one has to test whether the reconstruction is possible and keep a ghost history indicating whether we still implement the abstraction, what, in turn, requires to take into account a memory region occupied by the stack. Using such memory addresses for detecting the interleaving and consistency points, however, is not allowed by the *Cosmos* model and sequential simulation framework from Chapter 2.

Therefore, we will require the simulation relation to hold at every MIPS-86 step only when the processor executes inline assembly or some code outside of our compiled program. This means that even if a corresponding abstract MX steps does not exist during the execution of the compiled code, we will still use only the consistency points inserted by the compiler and consider the simulation at these points (see non-dashed arrows on Figure 7.2) though the trivial coupling for the abstract and implementing machines is possible in between.

Definition 7.36 (**Consistency Points for MXA Machine).** Given an MXA thread configuration $k \in \mathbb{K}_{\mathrm{MXA}}$ and π, *info*, θ, *cba*, we define whether the corresponding MXA machine is at the consistency point by the predicate

$$\begin{aligned} &cp_{\mathrm{MXA}}(k, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv} \\ &\qquad \textit{(i)} \quad mx(k) \implies cp_{\mathrm{MX}}(k.k_{\mathrm{mx}}.ac, \mathit{info}) \\ &\qquad \textit{(ii)} \quad isa(k) \implies \mathit{outside}(k.core, \mathit{info}, \mathit{cba}) \vee \\ &\qquad\qquad\qquad\qquad \mathit{inline}(k.core, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \vee \\ &\qquad\qquad\qquad\qquad cp^{\mathrm{MX}}_{r\mathrm{MIPS}}(k.core, \pi, \mathit{info}, \theta, \mathit{cba}) \end{aligned}$$

For a configuration $c \in \mathbb{C}_{\mathrm{MXA}}$ we simply reload the definition for brevity as

$$cp_{\mathrm{MXA}}(c, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv} cp_{\mathrm{MXA}}(c.k, \pi, \mathit{info}, \theta, \mathit{cba})$$

Definition 7.37 (**Consistency Points for MIPS-86 ISA Executing MXA).** Analogously, for the reduced MIPS-86 machine with a processor core configuration $core \in \mathbb{C}_{core}$ we define

$$\begin{aligned} cp^{\mathrm{MXA}}_{r\mathrm{MIPS}}(core, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv}\ & \mathit{outside}(core, \mathit{info}, \mathit{cba}) \vee \\ & \mathit{inline}(core, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \vee \\ & cp^{\mathrm{MX}}_{r\mathrm{MIPS}}(core, \pi, \mathit{info}, \theta, \mathit{cba}) \end{aligned}$$

Then, for $d \in \mathbb{C}_{\mathrm{MIPS}}$ we rewrite

$$cp^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{def}{\equiv} cp^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d.cpu.core, \pi, \mathit{info}, \theta, \mathit{cba})$$

7.3.3 Accessed Addresses

Definition 7.38 (**Memory Addresses for Stack Information Retrieving).** Similarly to the computation of the stack information in Definition 7.7 for a given $a \in \mathbb{B}^{32} \cup \{\bot\}$ we recursively collect all byte addresses occupied by stack information regions

$$A_{StI}(a, m) \overset{def}{\equiv} \begin{cases} \emptyset & : \ a = \bot \vee a = 0^{32} \\ \{a\}_{StIsize} \cup A_{StI}\ (StIba_{next}(a, m), m) & : \ \text{otherwise} \end{cases}$$

Therefore, $A_{StI}(StIba, m)$ contains addresses belonging to all stack information regions.

Definition 7.39 (Memory Addresses Accessed for Reading and Writing during an MXA Machine Step). For a configuration $c \in \mathbb{C}_{\mathrm{MXA}}$ and input $in \in \Sigma_{\mathrm{MXA}}$ such that the extended machine step $\delta^{\pi,\theta,\iota}_{\mathrm{MXA}}(c, in)$ with $\iota \equiv (cba, StIba)$ is defined we compute sets $reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba)$, $writes^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba)$ of addresses at which the memory is read and written during the MXA step. So, using the shorthands

$$c_{\mathrm{mx}} \equiv conf^{\mathrm{MXA}}_{\mathrm{MX}}(c) \qquad d \equiv conf^{\mathrm{MXA}}_{r\mathrm{MIPS}}(c) \qquad d' \equiv \delta_{r\mathrm{MIPS}}(d, in)$$

$$\widehat{d} \equiv conf^{start}_{r\mathrm{MIPS}}(c, in) \qquad \widehat{in} \equiv \left(\texttt{core}, \epsilon\, \mathbb{C}_{walk}, \epsilon\, \mathbb{C}_{walk}, 0^{256}\right) \qquad \widehat{d}' \equiv \delta_{r\mathrm{MIPS}}(\widehat{d}, \widehat{in})$$

and for any $x \in \mathbb{C}_{\mathrm{MIPS}}$ with $gpr_x \equiv x.cpu.core.gpr$

$$sti(x, StIba) \equiv R_{StI}\,(gpr_x(sp), gpr_x(bp), x.m, StIba)$$

the set $reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba)$ is obtained as (i) the MX $reads$-set for a mixed machine step, and (ii) in case of the switch to inline assembly or a pure ISA step it includes (a) the $reads$-set of the MIPS-86 model if there is no end of ISA steps, (b) this $reads$-set together with addresses for stack information if the end of ISA steps is reached and a matching base address of the stack and its maximal size are not found, and (c) if they are, $reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba)$ is computed as the MIPS-86 $reads$-set together with addresses for stack information and the stack:

$$reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba) \overset{def}{\equiv}$$

$$\begin{cases} reads_{\mathrm{MX}}\,(c_{\mathrm{mx}}, \pi, \theta) & : \ mx(c.k) \wedge \neg start^{\pi}_{isa}(c) \\ reads_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) & : \ start^{\pi}_{isa}(c) \wedge \neg end^{\pi,\theta,cba}_{isa}(\widehat{d}'.cpu) \\ reads_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) \cup A_{StI}(StIba, \widehat{d}'.m) & : \ start^{\pi}_{isa}(c) \wedge end^{\pi,\theta,cba}_{isa}(\widehat{d}'.cpu) \wedge \\ & \quad sti(\widehat{d}', StIba) = \bot \\ reads_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) \cup A_{StI}(StIba, \widehat{d}'.m) \cup A^{stack}_{\mathrm{MX}}(\widehat{sba'}, \widehat{mss'}) & : \ start^{\pi}_{isa}(c) \wedge end^{\pi,\theta,cba}_{isa}(\widehat{d}'.cpu) \wedge \\ & \quad sti(\widehat{d}', StIba) = (\widehat{sba'}, \widehat{mss'}) \\ reads_{\mathrm{MIPS}}\,(d, in) & : \ isa(c.k) \wedge \neg end^{\pi,\theta,cba}_{isa}(d'.cpu) \\ reads_{\mathrm{MIPS}}\,(d, in) \cup A_{StI}(StIba, d'.m) & : \ isa(c.k) \wedge end^{\pi,\theta,cba}_{isa}(d'.cpu) \wedge \\ & \quad sti(d', StIba) = \bot \\ reads_{\mathrm{MIPS}}\,(d, in) \cup A_{StI}(StIba, d'.m) \cup A^{stack}_{\mathrm{MX}}(sba', mss') & : \ isa(c.k) \wedge end^{\pi,\theta,cba}_{isa}(d'.cpu) \wedge \\ & \quad sti(d', StIba) = (sba', mss') \end{cases}$$

Note that the definition of $reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba)$ depends on the intermediate MIPS-86 step because it can change the stack information in the memory.[3]

The set $writes^{\pi,\theta}_{\mathrm{MXA}}(c, in)$ is defined as (i) the MX $writes$-set for a mixed machine step, (ii) the MIPS-86 $writes$-set together with the set of stack addresses[4] in case of the switch to inline

[3]In order to compute $reads^{\pi,\theta,cba}_{\mathrm{MXA}}(c, in)$ only on the base of the input configuration, one would need to introduce software conditions requiring that the stack information in the memory is not changed during the step before a possible reconstruction:

$$isa(c.k) \wedge end^{\pi,\theta,cba}_{isa}(d'.cpu) \implies A_{StI}(StIba, d.m) = A_{StI}(StIba, d'.m)$$

$$start^{\pi}_{isa}(c) \wedge end^{\pi,\theta,cba}_{isa}(\widehat{d}'.cpu) \implies A_{StI}(StIba, \widehat{d}.m) = A_{StI}(StIba, \widehat{d}'.m)$$

[4]Recall that during the switch to inline assembly the memory region occupied by the stack is provided by the input for the MXA step and put into the memory of the MXA machine.

assembly, and (iii) the MIPS-86 $writes$-set for a pure ISA step:

$$writes^{\pi,\theta}_{\mathrm{MXA}}(c, in) \overset{def}{\equiv}$$
$$\begin{cases} writes_{\mathrm{MX}}\left(c_{\mathrm{mx}}, \pi, \theta\right) & : \ mx(c.k) \land \neg start^{\pi}_{isa}(c) \\ A^{\pi,\theta}_{stack}(c, info, c.k.sba) \cup writes_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) & : \ start^{\pi}_{isa}(c) \\ writes_{\mathrm{MIPS}}\left(d, in\right) & : \ isa(c.k) \end{cases}$$

where the compiler information $info$ is computed as before.

Definition 7.40 (No Access to icm by MXA Machine). We combine both sets into

$$accad^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba) \overset{def}{\equiv} reads^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba) \cup writes^{\pi,\theta}_{\mathrm{MXA}}(c, in)$$

and define a predicate indicating that addresses icm are not accessed

$$noacc^{\pi,\theta}_{\mathrm{MXA}}(c, in, icm, cba, StIba) \overset{def}{\equiv} accad^{\pi,\theta}_{\mathrm{MXA}}(c, in, cba, StIba) \cap icm = \emptyset$$

7.3.4 $\mathcal{IO}$- and $\mathcal{OT}$-Points

Definition 7.41 ($\mathcal{IO}$- and $\mathcal{OT}$-Points for MXA Machine). Using the same shorthands from Definition 7.39, namely,

$$\widehat{d} \equiv conf^{start}_{r\mathrm{MIPS}}(c, in) \qquad d \equiv conf^{\mathrm{MXA}}_{r\mathrm{MIPS}}(c)$$

we define the MXA $\mathcal{IO}$-points as

$$\mathcal{IO}^{\pi,\theta}_{\mathrm{MXA}}(c, in, info, cba) \overset{def}{\equiv}$$
$$\begin{cases} \mathcal{IO}^{\pi,\theta}_{\mathrm{MXA}}\left(c_{\mathrm{mx}}\right) & : \ mx(c.k) \land \neg start^{\pi}_{isa}(c) \\ \mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}\left(\widehat{d}, \pi, info, \theta, cba\right) & : \ start^{\pi}_{isa}(c) \\ \mathcal{IO}_{\mathrm{MIPS}}(d, in) \land & : \ isa(c.k) \\ \left(\neg mode(d.cpu.core) \implies \mathcal{IO}^{\mathrm{MX}}_{r\mathrm{MIPS}}\left(d, \pi, info, \theta, cba\right)\right) & \end{cases}$$

The last case means either any step in user mode, or a processor step in system mode without interrupts such that the core executes cas, $locksw$, or a a volatile access by lw in the compiled code.

The MXA $\mathcal{OT}$-points are simply

$$\mathcal{OT}^{\pi,\theta}_{\mathrm{MXA}}(c, in) \overset{def}{\equiv} \begin{cases} \mathcal{OT}^{\pi,\theta}_{\mathrm{MXA}}\left(c_{\mathrm{mx}}\right) & : \ mx(c.k) \land \neg start^{\pi}_{isa}(c) \\ \mathcal{OT}^{\mathrm{MX}}_{r\mathrm{MIPS}}\left(\widehat{d}\right) & : \ start^{\pi}_{isa}(c) \\ \mathcal{OT}_{\mathrm{MIPS}}(d, in) & : \ isa(c.k) \end{cases}$$

Definition 7.42 (C-IL $\mathcal{IO}$-Points are Consistency Points in MXA). Additionally, we apply Definition 6.51 for the MXA machine.

$$\mathcal{IO}cp^{\pi,\theta}_{\mathrm{MXA}}(c, info) \overset{def}{\equiv} mx(c.k) \implies \mathcal{IO}cp^{\pi,\theta}_{\mathrm{MX}}\left(conf^{\mathrm{MXA}}_{\mathrm{MX}}(c), info\right)$$

Definition 7.43 ($\mathcal{IO}$- and $\mathcal{OT}$-Points for MIPS-86 ISA Executing MXA). For the SB reduced single core MIPS-86 in a configuration $d \in \mathbb{C}_{\text{MIPS}}$ the corresponding predicates are defined as

$$\mathcal{IO}^{\text{MXA}}_{r\text{MIPS}}(d, in, \pi, \mathit{info}, \theta, \mathit{cba}) \overset{\mathit{def}}{\equiv} \mathcal{IO}_{\text{MIPS}}(d, in) \wedge \big(\neg \mathit{mode}(d.\mathit{cpu}.\mathit{core}) \implies \mathcal{IO}^{\text{MX}}_{r\text{MIPS}}(d, \pi, \mathit{info}, \theta, \mathit{cba})\big)$$

$$\mathcal{OT}^{\text{MXA}}_{r\text{MIPS}}(d, in) \overset{\mathit{def}}{\equiv} \mathcal{OT}_{\text{MIPS}}(d, in)$$

7.3.5 Requirements and Conditions for MIPS-86 Machine

Now, similarly to the sequential MX compiler correctness, we define requirements on steps of the SB reduced MIPS-86 machines which will enable the simulation for the extended MX model.

Definition 7.44 (Suitability of Inputs for Reduced MIPS-86 Executing MXA). We consider an input $in \in \Sigma_{\text{MIPS}}$ to be suitable for the MXA simulation when the reset signal is low. Using the previous definition for the MIPS-86 encoding the MX machine, we define

$$\mathit{suit}^{\text{MXA}}_{r\text{MIPS}}(in) \overset{\mathit{def}}{\equiv} \mathit{suit}^{\text{MX}}_{r\text{MIPS}}(in)$$

Recall that according to Definition 6.54 the well-formedness of the MIPS-86 configuration for the MX simulation required that the processor core must be in system mode, the store buffer is empty, and the maskable interrupts are masked. Moreover, the software conditions guaranteed that MASM steps could not change the mode and status registers.

In the extended mixed machine semantics applicable for system programming, one might need to unmask the interrupts already during the hypervisor / OS kernel execution, therefore we will require the same well-formedness only when the processor executes the compiled MXA programm except its online assembly portions. Moreover, since the inline assembly belongs to the system code, the processor must be in system mode and have the empty store buffer what is guaranteed by the SB reduction.

Definition 7.45 (Configuration Well-Formedness for Reduced MIPS-86 Executing MXA). Then, for $d \in \mathbb{C}_{\text{MIPS}}$, given π, info, cba, and the shorthand $\mathit{core} \equiv d.\mathit{cpu}.\mathit{core}$, we define

$$\begin{aligned}
&\mathit{wfconf}^{\text{MXA}}_{r\text{MIPS}}(d, \pi, \mathit{info}, \mathit{cba}) \overset{\mathit{def}}{\equiv}\\
&\quad \textit{(i)}\quad \mathit{inside}(\mathit{core}, \mathit{info}, \mathit{cba}) \wedge \neg \mathit{inline}(\mathit{core}, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \implies \mathit{wfconf}^{\text{MX}}_{r\text{MIPS}}(d)\\
&\quad \textit{(ii)}\quad \mathit{inline}(\mathit{core}, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \implies \neg \mathit{mode}(\mathit{core}) \wedge d.\mathit{cpu}.\mathit{sb} = \varepsilon
\end{aligned}$$

We have already conditions that no reset occurs and the device and overflow interrupts are masked. As we already know, in order to simulate the MX machine, the compiler has to guarantee that no illegal instructions appear in the code and memory accesses are properly aligned. We require this in the well-behaviour of the MIPS-86 machine executed MXA program. Moreover, again one has to show that the software conditions needed for the store buffer reduction are transferred.

Definition 7.46 (Well-Behaviour of Reduced MIPS-86 Executing MXA). For the same arguments as in the previous definition, an input $in \in \Sigma_{\text{MIPS}}$, and the shorthands $\mathit{core} \equiv d.\mathit{cpu}.\mathit{core}$, $I \equiv d.m_4(\mathit{core}.\mathit{pc})$ we define

$$\begin{aligned}
&\mathit{wb}^{\text{MXA}}_{r\text{MIPS}}(d, in, \pi, \mathit{info}, \mathit{cba}) \overset{\mathit{def}}{\equiv}\\
&\quad \textit{(i)}\quad \mathit{sc}_{r\text{MIPS}}(d, in)\\
&\quad \textit{(ii)}\quad \mathit{inside}(\mathit{core}, \mathit{info}, \mathit{cba}) \wedge \neg \mathit{inline}(\mathit{core}, \pi_\mu, \mathit{info}_\mu, \mathit{cba}_\mu) \implies \neg \mathit{jisr}(\mathit{core}, I, \mathit{eev}, 0, 0)
\end{aligned}$$

where eev comes from the only possible for this case input $in = (\texttt{core}, w_I, w_D, \mathit{eev})$.

7.3.6 MXA Sofware Conditions

The static software conditions for the extended mixed machine differ from those stated for the MX machine in Definition 6.56 only for the code and stack regions. Namely, since the hypervisor / OS kernel can use compiled libraries, the compiled program π is included to the code region among with other binaries. Moreover, the stack region can be changed in the MXA semantics.

Definition 7.47 (Static Software Conditions for MXA). Therefore, for given π, $info$, θ, cba, we adapt the needed static conditions from Definition 6.56 as follows:

$$
\begin{aligned}
& sc^{stat}_{\mathrm{MXA}}(\pi, info, \theta, cba) \overset{def}{\equiv} \\
& \quad \textit{(i)} \quad A^{code}_{\mathrm{MX}}(info, cba) \subseteq A_{code} \\
& \quad \textit{(ii)} \quad valid^{code}_{\mathrm{MX}}(info, cba) \\
& \quad \textit{(iii)} \quad A^{gvar}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) \setminus A^{const}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) \subset A_{data} \\
& \quad \textit{(iv)} \quad A^{const}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta) = A_{const} \\
& \quad \textit{(v)} \quad sc^{prog}_{\mathrm{MX}}(\pi, \theta)
\end{aligned}
$$

Definition 7.48 (Dynamic Software Conditions for MXA). Given an MXA machine configuration $c \in \mathbb{C}_{\mathrm{MXA}}$ and an input $in \in \Sigma_{\mathrm{MXA}}$ such that the MXA step $c' \equiv \delta^{\pi,\theta,\iota}_{\mathrm{MXA}}(c, in)$ with $\iota \equiv (cba, StIba)$ is defined, we state the dynamic software conditions for this step and require that: (i) for any MX machine step the dynamic MX software conditions hold, (ii) for any MIPS-86 step the software conditions needed for the store buffer reduction hold, (iii) before a successful reconstruction of an MX thread configuration, the masked interrupts must be masked by the programmer again, (iv) all stacks found in the memory do not overlap, and (v) – (vi) the current stack as well as all stack information regions belong to the data region of the memory.

Let c_{mx}, d, $\widehat{d}$, $\widehat{in}$ be the shorthands from Definition 7.39, then using $\overline{d}$, $\overline{in}$, $\overline{d}'$, $sbas$, $msss$ considered below, we define

$$
\begin{aligned}
& sc^{dyn}_{\mathrm{MXA}}(c, in, \pi, info, \theta, cba, StIba) \overset{def}{\equiv} \\
& \quad \textit{(i)} \quad mx(c.k) \wedge \neg start^{\pi}_{isa}(c) \implies \\
& \quad \qquad sc^{dyn}_{\mathrm{MX}}(c_{\mathrm{mx}}, in, \pi, info, \theta, cba, c.k.sba, c.k.mss) \\
& \quad \textit{(ii)} \quad mx(c.k) \wedge start^{\pi}_{isa}(c) \vee isa(c.k) \implies sc_{r\mathrm{MIPS}}(\overline{d}, \overline{in}) \\
& \quad \textit{(iii)} \quad isa(c.k) \wedge mx(c'.k) \implies \forall i \in \{1, 7\}.\ \overline{d}'.cpu.core.spr(sr)[i] = 0 \\
& \quad \textit{(iv)} \quad isa(c.k) \wedge mx(c'.k) \implies valid^{stacks}_{\mathrm{MX}}(|sbas|, sbas, msss) \\
& \quad \textit{(v)} \quad mx(c.k) \implies A^{stack}_{\mathrm{MX}}(c.k.sba, c.k.mss) \subset A_{data} \\
& \quad \textit{(vi)} \quad A_{StI}(StIba, c.\mathcal{M}) \subset A_{data}
\end{aligned}
$$

where the corresponding MIPS-86 configuration $\overline{d}$ and the input $\overline{in}$ are

$$
(\overline{d}, \overline{in}) \equiv \begin{cases} (\widehat{d}, \widehat{in}) & : \ mx(c.k) \wedge start^{\pi}_{isa}(c) \\ (d, in) & : \ isa(c.k) \end{cases}
$$

and, hence, the next MIPS-86 configuration is computed as $\overline{d}' \equiv \delta_{r\mathrm{MIPS}}(\overline{d}, \overline{in})$. Moreover, the information about all available stacks used during the reconstruction is $(sbas, msss) \equiv StI(StIba, \overline{d}'.m)$.

Definition 7.49 (MXA Software Conditions). Combining both definition from above, we get

$$\begin{aligned} & sc_{\text{MXA}}(c, in, \pi, info, \theta, cba, StIba) \overset{def}{\equiv} \\ & \quad \textit{(i)} \quad sc^{stat}_{\text{MXA}}(\pi, info, \theta, cba) \\ & \quad \textit{(ii)} \quad sc^{dyn}_{\text{MXA}}(c, in, \pi, info, \theta, cba, StIba) \end{aligned}$$

7.3.7 Sequential MXA Compiler Correctness in Concurrent Context

Similarly how it was done in Section 6.1.2.8 we introduce a few auxiliary definitions and state the sequential MXA compiler correctness needed for the justification of the concurrent MXA model. The definitions here are simply obtained from Section 6.1.2.8 by substituting the predicates for the MX machine by the corresponding ones defined in this chapter.

For existing MXA steps from a given configuration $c_0 \in \mathbb{C}_{\text{MXA}}$ till the next consistency point, we require that the MXA software conditions hold, the possible inconsistent region icm of memory is not accessed, and the configuration of the MXA machine at the consistency point is well-formed. Formally, using $\iota \equiv (cba, StIba)$, we define

$$\begin{aligned} & SCseq_{\text{MXA}}(c_0, \pi, info, \theta, cba, StIba, icm) \overset{def}{\equiv} \\ & \quad \forall n \in \mathbb{N}, c \in (\mathbb{C}_{\text{MXA}\bot})^{n+1}, \lambda \in (\Sigma_{\text{MXA}})^n . \\ & \qquad \textit{(i)} \quad c_1 = c_0 \wedge \left(c_1 \longrightarrow^n_{\delta^{\pi,\theta,\iota}_{\text{MXA}}, \lambda} c_{n+1}\right) \\ & \qquad \textit{(ii)} \quad \forall i \in [2:n].\ c_i \neq \bot \implies \neg cp_{\text{MXA}}(c_i, \pi, info, \theta, cba) \\ & \qquad \textit{(iii)} \quad c_{n+1} \neq \bot \implies cp_{\text{MXA}}(c_{n+1}, \pi, info, \theta, cba) \\ & \qquad \implies \\ & \qquad \textit{(i)} \quad \forall i \in \mathbb{N}_n.\ sc_{\text{MXA}}(c_i, \lambda_i, \pi, info, \theta, cba, StIba) \wedge \\ & \qquad \qquad \qquad noacc^{\pi,\theta}_{\text{MXA}}(c_i, \lambda_i, icm, cba, StIba) \\ & \qquad \textit{(ii)} \quad c_{n+1} \neq \bot \implies wfconf^{\pi,\theta,cba}_{\text{MXA}}(c_{n+1}) \end{aligned}$$

In order to make the restrictions on the number of $\mathcal{IO}$-points and require the proper implementation of MXA steps suitable for $\mathcal{IO}$-operations and ownership transfer, we define a predicate for non-empty sequences $d \in (\mathbb{C}_{\text{MIPS}})^*$, $\sigma \in (\Sigma_{\text{MIPS}})^*$, such that $|d| = |\sigma| + 1$, and $c \in (\mathbb{C}_{\text{MXA}})^*$, $\tau \in (\Sigma_{\text{MXA}})^*$ with $|c| = |\tau| + 1$:

$$\begin{aligned} & one\mathcal{IO}^{\text{MXA}}_{r\text{MIPS}}(d, \sigma, c, \tau, \pi, info, \theta, cba) \overset{def}{\equiv} \\ & \quad \textit{(i)} \quad \forall i, j \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}^{\pi,\theta}_{\text{MXA}}(c_i, \tau_i, info, cba) \wedge \mathcal{IO}^{\pi,\theta}_{\text{MXA}}(c_j, \tau_j, info, cba) \implies i = j \\ & \quad \textit{(ii)} \quad \forall i, j \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}^{\text{MXA}}_{r\text{MIPS}}(d_i, \sigma_i, \pi, info, \theta, cba) \wedge \mathcal{IO}^{\text{MXA}}_{r\text{MIPS}}(d_j, \sigma_j, \pi, info, \theta, cba) \implies i = j \\ & \quad \textit{(iii)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}^{\pi,\theta}_{\text{MXA}}(c_i, \tau_i, info, cba)\right) \implies \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}^{\text{MXA}}_{r\text{MIPS}}(d_i, \sigma_i, \pi, info, \theta, cba)\right) \\ & \quad \textit{(iv)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{OT}^{\pi,\theta}_{\text{MXA}}(c_i, \tau_i)\right) \iff \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{OT}^{\text{MXA}}_{r\text{MIPS}}(d_i, \sigma_i)\right) \end{aligned}$$

Additionally, we use the following shorthands to indicate that there are no consistency points in the given sequences:

$$\begin{aligned} nocp^{\text{MXA}}_{r\text{MIPS}}(d, \pi, info, \theta, cba) &\overset{def}{\equiv} \forall i \in \mathbb{N}_{|d|}.\ \neg cp^{\text{MXA}}_{r\text{MIPS}}(d_i, \pi, info, \theta, cba) \\ nocp_{\text{MXA}}(c, \pi, info, \theta, cba) &\overset{def}{\equiv} \forall i \in \mathbb{N}_{|c|}.\ \neg cp_{\text{MXA}}(c_i, \pi, info, \theta, cba) \end{aligned}$$

Finally, the sequential compiler correctness for the extended mixed machine can be formulated in the way needed for its application in the concurrent context. Its statement is very similar to the correctness for the mixed machine given in Theorem 6.1 in detail. The only major difference is that instead of the base address of the stack and its maximal size we consider the base address of the stack information. Moreover, the possible inconsistent memory icm for the MXA machine does not include the code region.

Theorem 7.1 (Sequential MXA Compiler Correctness in Concurrent Context).

$$\forall \pi \in \mathit{Prog}_{\mathrm{MX}}, \mathit{cba} \in \mathbb{B}^{32} \times \mathbb{B}^{32}, \mathit{StIba} \in \mathbb{B}^{32}, \theta \in \mathit{Params}_{\mathrm{CIL}},$$
$$c_0 \in \mathbb{C}_{\mathrm{MXA}}, \mathit{info} \in \mathit{infoT}_{\mathrm{MX}}, \mathit{icm} \in 2^{\mathbb{B}^{32}}, k \in \mathbb{N}, d \in (\mathbb{C}_{\mathrm{MIPS}})^{k+1}, \omega \in (\Sigma_{\mathrm{MIPS}})^{k}.$$

(i) $\mathit{wfconf}^{\pi,\theta,\mathit{cba}}_{\mathrm{MXA}}(c_0) \wedge \mathit{wfconf}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d_1, \pi, \mathit{info}, \mathit{cba})$

(ii) $\mathit{cp}_{\mathrm{MXA}}(c_0, \pi, \mathit{info}, \theta, \mathit{cba}) \wedge \mathit{cp}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d_1, \pi, \mathit{info}, \theta, \mathit{cba})$

(iii) $\mathit{icm} \subset A_{hyp} \setminus A^{code}_{\mathrm{MX}}(\mathit{info}, \mathit{cba}) \wedge \mathit{consis}_{\mathrm{MXA}}(c_0, d_1, \pi, \theta, \mathit{info}, \mathit{cba}, \mathit{icm})$

(iv) $\left(d_1 \longrightarrow^{k}_{\delta_{r\mathrm{MIPS}},\omega} d_{k+1}\right) \wedge \forall i \in \mathbb{N}_k.\ \mathit{suit}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(\omega_i)$

(v) $\mathit{nocp}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d[2:k], \pi, \mathit{info}, \theta, \mathit{cba})$

(vi) $\mathit{SCseq}_{\mathrm{MXA}}(c_0, \pi, \mathit{info}, \theta, \mathit{cba}, \mathit{StIba}, \mathit{icm})$

$\Longrightarrow$

$$\exists n \in \mathbb{N}, d' \in (\mathbb{C}_{\mathrm{MIPS}})^{n+1}, \sigma \in (\Sigma_{\mathrm{MIPS}})^{n}, m \in \mathbb{N}, c \in (\mathbb{C}_{\mathrm{MXA}})^{m+1}, \tau \in (\Sigma_{\mathrm{MXA}})^{m}.$$

(i) $n \geq k \wedge d'[1:k+1] = d \wedge \sigma[1:k] = \omega$

(ii) $\left(d'_1 \longrightarrow^{n}_{\delta_{r\mathrm{MIPS}},\sigma} d'_{n+1}\right) \wedge \forall i \in \mathbb{N}_n.\ \mathit{suit}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(\sigma_i)$

(iii) $\mathit{cp}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d'_{n+1}, \pi, \mathit{info}, \theta, \mathit{cba}) \wedge \mathit{nocp}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d'[2:n], \pi, \mathit{info}, \theta, \mathit{cba})$

(iv) $\mathit{wfconf}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d'_{n+1},, \pi, \mathit{info}, \mathit{cba}) \wedge \forall i \in \mathbb{N}_n. \mathit{wb}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d'_i, \sigma_i, \pi, \mathit{info}, \mathit{cba})$

(v) $c_1 = c_0 \wedge \left(c_1 \longrightarrow^{m}_{\delta^{\pi,\theta,\iota}_{\mathrm{MXA}},\tau} c_{m+1}\right) \wedge \mathit{wfconf}^{\pi,\theta,\mathit{cba}}_{\mathrm{MXA}}(c_{m+1})$

(vi) $\mathit{cp}_{\mathrm{MXA}}(c_{m+1}, \pi, \mathit{info}, \theta, \mathit{cba}) \wedge \mathit{nocp}_{\mathrm{MXA}}(c[2:m], \pi, \mathit{info}, \theta, \mathit{cba})$

(vii) $\mathit{valid}^{cp}_{\mathrm{MX}}(\pi, \mathit{info}, \theta) \wedge \mathcal{IO}\mathit{cp}^{\pi,\theta}_{\mathrm{MXA}}(c_{m+1}, \mathit{info})$

(viii) $\mathit{one}\mathcal{IO}^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d', \sigma, c, \tau, \pi, \mathit{info}, \theta, \mathit{cba})$

(ix) $\mathit{consis}_{\mathrm{MXA}}(c_{m+1}, d'_{n+1}, \pi, \theta, \mathit{info}, \mathit{cba}, \mathit{icm})$

with $\iota \equiv (\mathit{cba}, \mathit{StIba})$.

We leave the proof of this theorem out of the scope of the thesis because we do not consider the compiler implementation in this thesis. In order to prove the claim, along with the application of Theorem 6.1 one has to argue about additional cases. For instance, if after an inline assembly step the control returns to the compiled code of the program π not at a C-IL consistency point, the compiler has also to guarantee that the next consistency point is reachable. This claim, however, is not covered by Theorem 6.1 because we consider there only execution starting from MX consistency points.

7.4 Justification of the Concurrent Extended Mixed Model

7.4.1 Cosmos Model Instantiations

Concurrent Extended Mixed Machine

Given a program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in \mathit{Prog}_{\text{MX}}$ with inline assembly, the environment parameters $\theta \in \mathit{Params}_{\text{CIL}}$, the system information $c\iota \equiv (\mathit{cba}, \mathit{StIbas})$ with $\mathit{cba} \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, $\mathit{StIbas} \in \left(\mathbb{B}^{32}\right)^{np}$ wrt. the number $np \in \mathbb{N}$ of processors in the multi-core MIPS-86 machine, and the compiler information $\mathit{info} \equiv (\mathit{cmpl}_{\text{MASM}}(\pi_\mu), \mathit{cmpl}_{\text{CIL}}(\pi_{\text{cil}}))$, we define the instantiation $S_{\text{MXA}}^{\pi,\theta,c\iota} \in \mathbb{S}$ of the *Cosmos* model for the extended mixed machine.

Let $c_{\text{mxa}}(u, m) \equiv (u.k,\ \lceil m \rceil_{\mathbb{B}^{32}})$ be an MXA configuration composed from the MXA thread configuration $u.k$, and the partial memory $m : \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$ extended with dummy values. Then, the signature of the *Cosmos* model $S_{\text{MXA}}^{\pi,\theta,c\iota}$ is defined as:

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{A} = \mathbb{B}^{32}$

  In contrast to $S_{\text{MX}}^{\pi,\theta,\xi}$ from Section 6.2.1, the code and stack regions are visible in the concurrent MXA model.

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{V} = \mathbb{B}^8$

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{R} = A_{\text{CIL}}^{\mathit{const}}(\pi_{\text{cil}}, \theta) \cup A_{\mathit{code}}$

  with $A_{\text{MX}}^{\mathit{code}}(\mathit{info}, \mathit{cba}) \subseteq A_{\mathit{code}}$ following from the MXA software conditions.

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathit{nu} = np$

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{U} = \left(k \in \mathbb{K}_{\text{MXA}},\ \mathit{StIba} \in \mathbb{B}^{32}\right) \cup \{\bot\}$

  The second components is introduce in the unit configuration in order to match the formalism of *Cosmos* model because we never use an index of the unit in the *Cosmos* model instantiation. Therefore, for any unit configuration $u \in S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{U}$ we will refer the MXA thread configuration by $u.k$, and the stack information base address of this unit by $u.\mathit{StIba}$. Obviously, we will require for $E \in \mathbb{C}_{S_{\text{MXA}}^{\pi,\theta,c\iota}}$

  $$\mathit{StIbainv}_{\text{MXA}}(E) \overset{\mathit{def}}{\equiv} \forall p \in \mathbb{N}_{nu}.\ E.u_p.\mathit{StIba} = \mathit{StIbas}_p$$

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{E} = \Sigma_{\text{MXA}}$

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathit{reads}(u, m, in) = \begin{cases} \mathit{reads}_{\text{MXA}}^{\pi,\theta}\left((u.k, m), in, \mathit{cba}, u.\mathit{StIba}\right) & : u \neq \bot \\ \emptyset & : \text{otherwise} \end{cases}$

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\delta(u, m, in) = \begin{cases} \left(u',\ c'_{\text{mxa}}.\mathcal{M}|_{\mathit{writes}_{\text{MXA}}^{\pi,\theta}(c_{\text{mxa}}(u,m), in)}\right) & : u \neq \bot \land c'_{\text{mxa}} \neq \bot \\ (\bot, m_\emptyset) & : u \neq \bot \land c'_{\text{mxa}} = \bot \\ \mathit{undefined} & : \text{otherwise} \end{cases}$

  where $c'_{\text{mxa}} \equiv \delta_{\text{MXA}}^{\pi,\theta,\iota}\left(c_{\text{mxa}}(u, m), in\right)$ with $\iota \equiv (\mathit{cba}, u.\mathit{StIba})$ is the next sequential MXA machine configuration, $u' \equiv (c'_{\text{mxa}}.k, u.\mathit{StIba})$ is the next unit configuration, and $m_\emptyset$ denotes the empty function with $\operatorname{dom}(m_\emptyset) = \emptyset$.

- $S_{\text{MXA}}^{\pi,\theta,c\iota}.\mathcal{IP}(u, m, in) = (u \neq \bot \implies \mathit{cp}_{\text{MXA}}(u.k, \pi, \mathit{info}, \theta, \mathit{cba}))$

- $S^{\pi,\theta,c\iota}_{\mathrm{MXA}}.\mathcal{IO}(u,m,in) = \Big(u \neq \bot \implies \mathcal{IO}^{\pi,\theta}_{\mathrm{MXA}}\left(c_{\mathrm{mxa}}(u,m), in, \mathit{info}, \mathit{cba}\right)\Big)$
- $S^{\pi,\theta,c\iota}_{\mathrm{MXA}}.\mathcal{OT}(u,m,in) = \Big(u \neq \bot \implies \mathcal{OT}^{\pi,\theta}_{\mathrm{MXA}}\left(c_{\mathrm{mxa}}(u,m), in\right)\Big)$

SB Reduced Multi-Core MIPS-86 Implementing Concurrent MXA Machine

The instantiation $S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}} \in \mathbb{S}$ of the *Cosmos* model with the SB reduced MIPS-86 concurrently running the compiled code of the MXA program π is very similar to $S^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ from the previous chapter and defined as:

- For components $X \in \{\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, \mathit{reads}, \delta\}$ the instantiation is equal to the one for the reduced machine
$$S^{\pi,\theta,\xi}_{r\mathrm{MIPS}}.X = S_{r\mathrm{MIPS}}.X$$
with $A^{code}_{\mathrm{MX}}(\mathit{info}, \mathit{cba}) \subseteq A_{code}$ and $A_{const} = A^{const}_{\mathrm{CIL}}(\pi_{\mathrm{cil}}, \theta)$.
- $S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}}.\mathcal{IP}(u,m,in) = cp^{\mathrm{MXA}}_{r\mathrm{MIPS}}(u.core, \pi, \mathit{info}, \theta, \mathit{cba})$
- $S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}}.\mathcal{IO}(u,m,in) = \mathcal{IO}^{\mathrm{MXA}}_{r\mathrm{MIPS}}\left((u, \lceil m \rceil_{\mathbb{B}^{32}}), in, \pi, \mathit{info}, \theta, \mathit{cba}\right)$
- $S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}}.\mathcal{OT}(u,m,in) = \mathcal{OT}^{\mathrm{MXA}}_{r\mathrm{MIPS}}\left((u, \lceil m \rceil_{\mathbb{B}^{32}}, in)\right)$

7.4.2 Sequential Simulation Theorem

As the next step, we instantiate the sequential simulation framework $R^{S_{\mathrm{MXA}}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,c\iota) \in \mathbb{R}$ for the two *Cosmos* models $S^{\pi,\theta,c\iota}_{\mathrm{MXA}}, S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}} \in \mathbb{S}$ defined wrt. the given π, θ, and $c\iota \equiv (\mathit{cba}, \mathit{StIbas})$.

For $d \in \mathbb{C}_{proc} \times \mathbb{C}_m$, $d = (cpu, m)$, $c \in S^{\pi,\theta,c\iota}_{\mathrm{MXA}}.\mathcal{U} \times \left(\mathbb{B}^{32} \to \mathbb{B}^8\right)$, $c = (u, \mathcal{M})$, and $\mathit{info} \in \mathit{infoT}_{\mathrm{MX}}$, $icm \in 2^{\mathbb{B}^{32}}$ we define

$$R^{S_{\mathrm{MXA}}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,c\iota).\begin{cases} \mathcal{P} & = \mathit{infoT}_{\mathrm{MX}} \\ sim(d,c,\mathit{info},icm) & = icm \subset A_{hyp} \setminus A^{code}_{\mathrm{MX}}(\mathit{info},\mathit{cba}) \wedge \\ & \quad valid^{cp}_{\mathrm{MX}}(\pi,\mathit{info},\theta) \wedge \mathcal{IO}cp^{\pi,\theta}_{\mathrm{MXA}}(c,\mathit{info}) \wedge \\ & \quad \big(u \neq \bot \implies \\ & \quad consis_{\mathrm{MXA}}\left((u.k,\mathcal{M}), d, \pi, \theta, \mathit{info}, \mathit{cba}, icm\right)\big) \\ \mathcal{CP}a(u,\mathit{info}) & = (u \neq \bot \implies cp_{\mathrm{MXA}}(u.k, \pi, \mathit{info}, \theta, \mathit{cba}, \mathit{info})) \\ \mathcal{CP}c(cpu,\mathit{info}) & = cp^{\mathrm{MXA}}_{r\mathrm{MIPS}}(cpu.core, \pi, \mathit{info}, \theta, \mathit{cba}) \\ wfa(c) & = u \neq \bot \wedge wfconf^{\pi,\theta,cba}_{\mathrm{MXA}}((u.k,\mathcal{M})) \\ wfc(d) & = wfconf^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d, \pi, \mathit{info}, \mathit{cba}) \\ suit(in_d) & = suit^{\mathrm{MXA}}_{r\mathrm{MIPS}}(in_d) \\ sc(c, in_c, \mathit{info}) & = (u \neq \bot \implies \\ & \quad sc_{\mathrm{MXA}}((u.k,\mathcal{M}), in, \pi, \mathit{info}, \theta, \mathit{cba}, u.\mathit{StIba})) \\ wb(d, in_d, \mathit{info}) & = wb^{\mathrm{MXA}}_{r\mathrm{MIPS}}(d, in_d, \pi, \mathit{info}, \mathit{cba}) \end{cases}$$

Obviously, having proven Theorem 7.1, one easily gets that the generalized sequential simulation Theorem 2.3 also holds for $S^{\pi,\theta,c\iota}_{\mathrm{MXA}}, S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}} \in \mathbb{S}$ and the framework $R^{S_{\mathrm{MXA}}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,c\iota) \in \mathbb{R}$ instantiated wrt. the given program π and the parameters $\theta, c\iota$. Hence, the sequential MXA compiler correctness in the concurrent setting in terms of *Cosmos* machines is stated similarly to Theorem 6.2.

Theorem 7.2 (Sequential MXA Compiler Correctness for *Cosmos* Model Simulation). *The generalized sequential simulation Theorem 2.3 holds for any* Cosmos *models* $S^{\pi,\theta,c\iota}_{\mathrm{MXA}}, S^{\pi,\theta,c\iota}_{r\mathrm{MIPS}} \in \mathbb{S}$ *and the simulation framework* $R^{S_{\mathrm{MXA}}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,c\iota) \in \mathbb{R}$ *instantiated wrt. any given mixed machine program* $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in \mathit{Prog}_{\mathrm{MX}}$ *with inline assembly, the environment parameters* $\theta \in \mathit{Params}_{\mathrm{CIL}}$*, and the system information* $c\iota \equiv (\mathit{cba}, \mathit{StIbas})$ *with* $\mathit{cba} \in \mathbb{B}^{32} \times \mathbb{B}^{32}$*,* $\mathit{StIbas} \in \left(\mathbb{B}^{32}\right)^{np}$*, and the number* $np \in \mathbb{N}$ *of processors in the multi-core MIPS-86 machine.*

7.4.3 Concurrent Model Simulation

Finally, we consider the shared and unit invariants as well as other properties which enable the simulation between MXA and SB reduced MIPS-86 *Cosmos* machines for the given program π and parameters θ, $c\iota$.

First, in order to define the property $P_{S^{\pi,\theta,c\iota}_{\mathrm{MXA}}}$ instantiating P_{S_a} in Theorem 2.4, we introduce auxiliary predicates.

Definition 7.50 (Guest Addresses are Shared in MXA *Cosmos* machine). Obviously, as before, for any configuration $E \in \mathbb{C}_{S^{\pi,\theta,c\iota}_{\mathrm{MX}}}$ of the MXA *Cosmos* machine we require that A_{guest} is always treated as shared and no addresses from A_{guest} can be owned by any execution unit.

$$\mathit{oginv}_{\mathrm{MXA}}(E) \overset{def}{\equiv} A_{guest} \subset E.\mathcal{S} \wedge \forall p \in \mathbb{N}_{nu}.\ E.\mathcal{O}_p \cap A_{guest} = \emptyset$$

Definition 7.51 (Current Stacks and Stack Information are Local in MXA *Cosmos* machine). Moreover, for such configurations E we require that the memory regions occupied by the stacks having the MX abstraction are locally owned. The same holds also for all stack information regions.

$$\begin{aligned} &\mathit{ostinv}_{\mathrm{MXA}}(E) \overset{def}{\equiv} \\ &\quad \text{(i)} \quad \forall p \in \mathbb{N}_{nu}.\ mx(E.u_p.k) \implies A^{stack}_{\mathrm{MX}}(\mathit{sba}_p, \mathit{mss}_p) \subset E.\mathcal{O}_p \setminus E.\mathcal{S} \\ &\quad \text{(ii)} \quad \forall p \in \mathbb{N}_{nu}.\ A_{\mathit{StI}}(E.u_p.\mathit{StIba}, E.m) \subset E.\mathcal{O}_p \setminus E.\mathcal{S} \end{aligned}$$

where $\mathit{sba}_p \equiv E.u_p.k.sba$ and $\mathit{mss}_p \equiv E.u_p.k.mss$ denote the stack base address and maximal stack size for the unit p.

Therefore, the property $P_{S^{\pi,\theta,c\iota}_{\mathrm{MXA}}}$ on configurations of the MXA *Cosmos* machine is formalized as

$$P_{S^{\pi,\theta,c\iota}_{\mathrm{MXA}}}(E) \overset{def}{\equiv} \mathit{oginv}_{\mathrm{MXA}}(E) \wedge \mathit{ostinv}_{\mathrm{MXA}}(E) \wedge \mathit{StIbainv}_{\mathrm{MXA}}(E)$$

Since the sets of memory addresses in both machines are equal, the shared invariant is trivial.

Definition 7.52 (Shared Invariant for Concurrent MXA Machine Simulation). For the *Cosmos* models instantiated with the extended mixed machine and the SB reduced MIPS-86 we demand the equality of their sets of shared $\mathcal{S}$, $\mathcal{S}_{\mathrm{mxa}}$, read-only $\mathcal{R}$, $\mathcal{R}_{\mathrm{mxa}}$, and owned $\mathcal{O}(p)$, $\mathcal{O}_{\mathrm{mxa}}(p)$ by each unit p addresses, as well as the contents of shared and read-only memories m, m_{mxa}.

$$\begin{aligned} &\mathit{sinv}^{\mathrm{MXA}}_{S_{r\mathrm{MIPS}}}(\pi,\theta,c\iota)\big((m,\mathcal{S},\mathcal{R},\mathcal{O}),(m_{\mathrm{mxa}},\mathcal{S}_{\mathrm{mxa}},\mathcal{R}_{\mathrm{mxa}},\mathcal{O}_{\mathrm{mxa}})\big) \overset{def}{\equiv} \\ &\quad \text{(i)} \quad \mathcal{S} = \mathcal{S}_{\mathrm{mxa}} \\ &\quad \text{(ii)} \quad \mathcal{R} = \mathcal{R}_{\mathrm{mxa}} \\ &\quad \text{(iii)} \quad \forall p \in \mathbb{N}_{nu}.\ \mathcal{O}(p) = \mathcal{O}_{\mathrm{mxa}}(p) \\ &\quad \text{(iv)} \quad m = m_{\mathrm{mxa}} \end{aligned}$$

Like in the concurrent MX machine simulation, we do not need to introduce any restrictions on the unit's configuration of the SB reduced machine and set for $cpu \in \mathbb{C}_{proc}$

$$uinv^{S_{\text{MXA}}}_{S_{r\text{MIPS}}}(\pi, \theta, c\iota)(cpu, \mathcal{O}, \mathcal{S}) \overset{def}{\equiv} 1$$

Now, given that Assumptions 2.1– 2.4 are discharged for the instantiated machines wrt. any π, θ, $c\iota$, one easily guarantees that the *Cosmos* model simulation theorem holds for each case.

Theorem 7.3 (*Cosmos* Model Simulation Theorem for all MX Programs with Inline Assembly and System/Environment Parameters). *Theorem 2.4 holds for any programs $\pi = (\pi_\mu, \pi_{\text{cil}}) \in Prog_{\text{MX}}$ with inline assembly, the environment parameters $\theta \in Params_{\text{CIL}}$, and the system information $c\iota \equiv (cba, StIbas)$ used for instantiation of the models $S^{\pi,\theta,c\iota}_{\text{MXA}}, S^{\pi,\theta,c\iota}_{r\text{MIPS}} \in \mathbb{S}$.*

7.4.4 Application of the Concurrent MXA Machine Simulation

The application of the concurrent MXA machine simulation for the order reduction is pretty similar to the one covered in Section 6.2.4.

Particularly, one can easily prove the claim of Lemma 6.1 for $wb^{\text{MXA}}_{r\text{MIPS}}(d, in, \pi, info, cba)$ by the same argumentation as before.

Moreover, for the store buffer reduction we need the property $P_{S_{r\text{MIPS}}}(E)$ for $E \in \mathbb{C}_{S_{r\text{MIPS}}}$. In fact, we have $oginv_{\text{MXA}}(C)$ for $C \in \mathbb{C}_{S^{\pi,\theta,c\iota}_{\text{MX}}}$ and the shared invariant $sinv(D, C, par)$ requires that the sets of addresses and memory regions involved into the coupling are the same for C and $D \in \mathbb{C}_{S'^{\pi,\theta,c\iota}_{r\text{MIPS}}}$ of the extended SB reduced MIPS-86 *Cosmos* machine. Therefore, the claim of Lemma 6.2 for this case directly follows from the shared invariant and does not need other argumentation.

Additionally, one can also consider the transfer of any other divisible property $P'_{S^{\pi,\theta,c\iota}_{\text{MXA}}}$ of the MXA *Cosmos* machine (such that the simulation hypothesis hold) into the incompletely simulated property $Q[P'_{S^{\pi,\theta,c\iota}_{\text{MXA}}}, par]$ of $S'^{\pi,\theta,c\iota}_{r\text{MIPS}}$.

Having transferred the well-behaviour and safety properties from suitable block schedule computations to any arbitrary interleaved computations of the SB reduced multi-core MIPS-86, one can apply the order reduction and the store buffer reduction in the way sketched in Section 6.2.4. We leave these technical details here as a simple bookkeeping exercise.

8 Concurrent Kernel Threads: Model, Implementation, and Correctness Criteria

In the previous chapter we considered and justified the semantics of the concurrent machine with MXA threads being executed on the multi-core MIPS-86. This semantics can be used for the implementation of hypervisor and operating system kernels where multi-threading can be restricted by the number of processor cores. However, as we already know, it is typical to have more threads operating in the address space of a given system or user process.

In this chapter we proceed with the extension of the concurrent MXA semantics with *cooperative*[1] POSIX-like threads [POS95, But97], provide their abstract model, implementation, and consider the correctness criteria in detail. We chose the minimal number of operations on kernel threads which should be sufficient for multi-theraded system programming as well as implementation of other functions of POSIX API. Particularly, we pay attention on the semantics and correctness of the thread switch based of stack substitution, and special functions working on shared resources requiring lock protection.

To the best of our knowledge, though the stack based switch is a classical method used by system programmers (e.g., see Chapter 20 in [Han96]), the detailed semantics of such a thread switch and correctness of its implementation for the industrial-like higher-level programming languages on multi-core machines is not considered in any known scientific works (e.g., [FS05, FSV$^+$06, FSDG08, NYS07, GFSS12]). Moreover, in order to specify operations using the lock protection, we take into account our model for concurrent simulation and split the operations into phases similar to those considered in [CL98].

The semantics of kernel threads given here is more detailed than just the concurrent MXA machine containing a number of threads greater than the number of processor cores and additionally specifying operations on them. Some details (e.g., information about the stack in the call of the thread creation function, thread identifiers explicitly provided for the thread operations) are needed by the thread manager responsible for the scheduling policy, cleaning the system from the finished thread, queuing requests between threads, etc., and can be further abstracted away in the presence of its implementation and the memory manager performing the memory allocation.

[1]In comparison to *preemptive threads* being switched by the scheduler that can take a decision about the time slot for the thread execution and also rely on the timer interrupts, the *cooperative threads* decide by themselves when and to which next thread they give the control.

8.1 Abstract Model of Kernel Threads

8.1.1 Program and Parameters

In the model we consider a program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in Prog_{\text{MX}}$ corresponding to the code of the hypervisor / OS kernel implementation. Since the context switch and virtualization are supposed to be performed on this level, the MASM program π_μ is allowed to contain inline assembly portions.

The C-IL program π_{cil} has the declaration of the function pointer type *thfun_t* for thread entry functions

```
typedef void (*thfun_t)(u32);
```

and the external functions

```
extern void thread_create(u32 tid, u32 pid, thfun_t fn, u32 arg, void *sba, u32 mss);
extern void thread_acquire();
extern void thread_run(u32 tid);
extern void thread_exit_to(u32 tid);
extern void thread_delete(u32 tid);
```

implementing the operations on the kernel threads, namely

- *thread_create* – creation of a new thread for a given processor,
- *thread_acquire* – acquiring all newly created threads by the running processor,
- *thread_run* – switch a thread with a given identifier,
- *thread_exit_to* – finishing the running thread and a switch to a given thread,
- *thread_delete* – deletion of a given finished thread.

We call them as *special functions* or *primitives* and define the set $\mathbb{F}_{name}^{prim} \subset \mathbb{F}_{name}$ of corresponding function names as

$$\mathbb{F}_{name}^{prim} \overset{def}{\equiv} \{thread\_create, thread\_acquire, thread\_run, thread\_exit\_to, thread\_delete\}$$

A typical lifecycle of a thread is depicted on Figure 8.1 where *schedule* is an operation performed by either *thread_run* or *thread_exit_to* which we call *scheduling primitives*. A new thread with a free (i.e., not used by other threads) identifier is created by the call of *thread_create* taking additionally as parameters an ID of a processor on which the thread will run, the pointer to the thread entry function, its argument, and the information about the allocated stack. A new thread after its creation resides in a global pool of new threads visible for all processors and can only be locally scheduled after its acquisition by a corresponding processor. When the primitive *thread_acquire* is executed on a processor, all existing new threads with a corresponding processor ID are acquired and become ready for scheduling. A new thread starts to run on a processor or a sleeping thread is restored when one of the scheduling primitives with its ID is called. Moreover, the running thread calling such a primitive either switches to the sleeping state in case of *thread_run* or becomes finished after the execution of *thread_exit_to*. A finished thread cannot be scheduled again and can only be deleted by *thread_delete* executed in the code of any running thread present in the model.

Along with the program, as before we keep a corresponding C-IL environment parameter $\theta \in Params_{\text{CIL}}$.

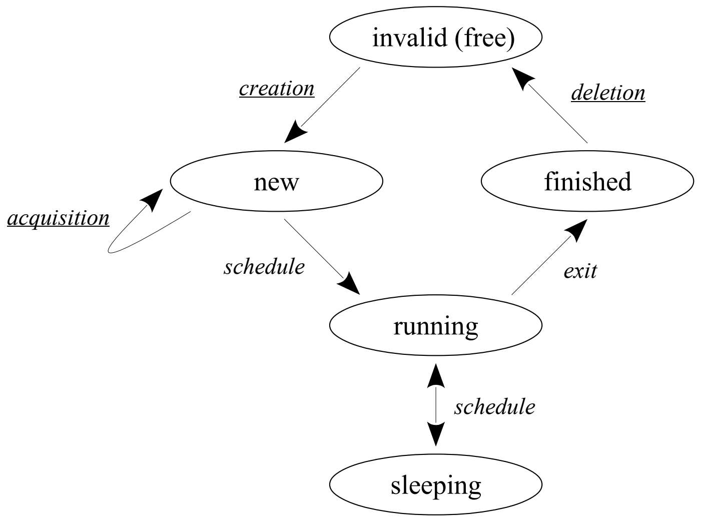

Figure 8.1: Thread lifecycle. The circles on figure denote thread state whereas the edges are operations executed by the primitives. The underlined operations are performed on data that can be shared between processors and, therefore, are considered to be global in comparison to the schedule and exit operation being local.

We consider the model with $np \in \mathbb{N}$ running threads[2] corresponding to the number of processors with identifiers from the set $\mathbb{N}_{np}$. The constant $mtid \in \mathbb{N}$ represents the maximal thread identifier. Each thread ID belongs to $\mathbb{N}_{mtid}$. Moreover, we require $mtid \geq np$.

Since in the model of kernel threads we will also consider the execution of inline assembly as well as guest / process steps, we will basically rely on a simplified version of the extended mixed machine (from Chapter 7) where only the stack of the running thread may be reconstructed after the pure MIPS-86 ISA steps. Therefore, we introduce here again the code base addresses $cba = (cba_{\text{cil}}, cba_{\mu}) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$ such that $valid^{code}_{\text{MX}}(info, cba)$ holds, and the compiler information $info = (info_{\text{cil}}, info_{\mu}) \in infoT_{\text{MX}}$ computed as $info_{\mu} = cmpl_{\text{MASM}}(\pi_{\mu})$ and $info_{\text{cil}} = cmpl_{\text{CIL}}(\pi_{\text{cil}})$.

8.1.2 Sequential Semantics in Concurrent Setting

8.1.2.1 Machine Configuration

Though in the model for clarity we will explicitly distinguish between running, sleeping, finished, and newly created threads, we introduce only two kinds of thread configurations, namely, *running* and *non-running*. Intuitively, new, sleeping, and finished threads have a configuration of the non-running thread, which, however, must be restricted by the well-formedness depending on the state.

In Definition 7.1 we have already introduced the configuration of the MXA thread represented by the MX thread or the processor configuration in case of inline assembly and guest / process steps. The configuration of the running thread is obtained by extending this definition

[2]In this thesis we do not consider the system startup during which at least one thread is created for each processor of the multi-core machine.

so that the processor configuration is accompanied by the base address and maximal stack size too. These components are needed in order to allow the context switch in the model.

Definition 8.1 (Configuration of the Running Thread). The configuration of the running thread

$$\mathbb{K}_{Th}^{run} \overset{def}{\equiv} \mathbb{K}_{Th}^{mx} \cup \mathbb{K}_{Th}^{isa}$$

is either the MXA configuration in case of the mixed machine steps

$$\mathbb{K}_{Th}^{mx} \overset{def}{\equiv} \big(k_{\text{mx}} \in \mathbb{K}_{\text{MX}},\ sba \in \mathbb{B}^{32},\ mss \in \mathbb{N},\ tlb \in \mathbb{C}_{tlb}\big)$$

or the extended processor configuration during inline assembly, guest, or user steps

$$\mathbb{K}_{Th}^{isa} \overset{def}{\equiv} \big(cpu \in \mathbb{C}_{proc},\ sba \in \mathbb{B}^{32},\ mss \in \mathbb{N}\big)$$

Since we do not support thread migration in this work, the processor affinity of any running thread can be directly taken from the corresponding special purpose register.

Definition 8.2 (Processor Affinity of the Running Thread). Given a configuration $th \in \mathbb{K}_{Th}^{run}$ of a running thread, we obtain its processor affinity as

$$pid_{run}(th) \overset{def}{\equiv} \begin{cases} \langle th.k_{\text{mx}}.ac.spr(pid)\rangle & : th \in \mathbb{K}_{Th}^{mx} \\ \langle th.cpu.core.spr(pid)\rangle & : \text{otherwise} \end{cases}$$

In comparison to the running threads, the configuration of a non-running thread is simpler. Such a thread can be newly created or be the result of the switch performed by a call of the corresponding primitive from the C-IL context. Therefore, the active context of the non-running thread can be only of the C-IL type. Moreover, usually during the thread switch the special purpose registers are not saved and restored, and we do not keep them in the configuration either.

Definition 8.3 (Configuration of a Non-Running Thread). Therefore, we model any non-running thread by the list of inactive execution contexts, the active C-IL context, the same components sba, mss characterizing the allocation of the stack in the memory, and the processor affinity pid.

$$\begin{aligned}\mathbb{K}_{Th}^{nrun} \overset{def}{\equiv} \Big(&ac \in context_{\text{CIL}},\ ic \in \big(context_{\text{CIL}} \cup context_{\text{MASM}}^{inactive}\big)^*,\\ &sba \in \mathbb{B}^{32},\ mss \in \mathbb{N},\ pid \in \mathbb{N}_{np}\Big)\end{aligned}$$

In the manner as it was done before, we define a configuration corresponding to the execution unit. Such a configuration contains all threads belonging to a single processor, except those threads that are still not acquired.

Definition 8.4 (Configuration of Processor's Threads). The configuration of threads belonging to a given processor is represented by a tuple containing two mappings *ready* and *fin* from threads identifiers to configurations of threads *ready* for scheduling and a scheduled one in the first case, and finished in the second one. The identifier of the scheduled (or current) thread is kept in the component ct.

$$\begin{aligned}\mathbb{K}_{Th}^{proc} \overset{def}{\equiv} \Big(&ready : \mathbb{N}_{mtid} \rightharpoonup \mathbb{K}_{Th}^{run} \cup \mathbb{K}_{Th}^{nrun},\\ &fin : \mathbb{N}_{mtid} \rightharpoonup \mathbb{K}_{Th}^{nrun},\ ct \in \mathbb{N}_{mtid}\Big)\end{aligned}$$

Finally, we define the configuration of the machine with kernel threads. Though we still consider the sequential case, the definition is adapted for the use in the concurrent setting.

Definition 8.5 (**Configuration of Sequential Machine for Kernel Threads**). Full configurations of the sequential machine with kernel threads are defined by the set $\mathbb{C}_{Th}$ of tuples

$$\mathbb{C}_{Th} \overset{def}{\equiv} \left(k \in \mathbb{K}_{Th}^{proc},\ new : \mathbb{N}_{mtid} \rightharpoonup \mathbb{K}_{Th}^{nrun},\ free \in 2^{\mathbb{N}_{mtid}},\ ap \in [0 : np],\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^{8}\right)$$

containing the following components:

- k – the local configuration of the processor's threads,
- new – the mapping from the identifier of newly created threads to their configurations,
- $free$ – a set of free thread identifiers that can be used for the thread creation,
- ap – the identifier of a processor currently working on global new and $free$ (or accessing processor). If no processor accesses them, we have $ap = 0$. The component ap models a lock used in the implementation of the kernel threads.
- $\mathcal{M}$ – the global byte-addressable memory.

8.1.2.2 Transition Function

In order to define the semantics we first introduce a few reloaded auxiliary functions needed for using the semantics from lower levels of the model stack.

Definition 8.6 (**Current Thread Configuration**). For a given configuration $k \in \mathbb{K}_{Th}^{proc}$ of processor's threads we define a function computing the configuration of the current thread as

$$th_{cur}(k) \overset{def}{\equiv} k.ready(k.ct)$$

Therefore, for a configuration $c \in \mathbb{C}_{Th}$ of the sequential machine for kernel threads we simply have

$$th_{cur}(c) \overset{def}{\equiv} th_{cur}(c.k)$$

Using this definition we can introduce the shorthands for the base address and maximal size of the stack for the current thread:

$$sba_{cur}(c) \overset{def}{\equiv} th_{cur}(c).sba$$
$$mss_{cur}(c) \overset{def}{\equiv} th_{cur}(c).mss$$

Given a configuration $c \in \mathbb{C}_{Th}$ and Definition 8.2 we can also denote the processor affinity of the current thread by

$$pid_{cur}(c) \overset{def}{\equiv} pid_{run}(th_{cur}(c))$$

Definition 8.7 (**Current Thread to MXA Machine**). We also easily transform the configuration of the current thread in $k \in \mathbb{K}_{Th}^{proc}$ into the MXA thread configuration from Section 7.1.2 by the function

$$conf_{\mathbb{K}_{\mathrm{MXA}}}^{curT}(k) \overset{def}{\equiv} \begin{cases} th & : th \in \mathbb{K}_{Th}^{mx} \\ th.cpu & : \text{otherwise} \end{cases}$$

with $th \equiv th_{cur}(k)$.

Hence, the configuration of the current thread in a configuration $c \in \mathbb{C}_{Th}$ can be easily transformed into an MXA machine configuration by

$$conf^{curT}_{\mathrm{MXA}}(c) \overset{def}{\equiv} \left(conf^{curT}_{\mathbb{K}_{\mathrm{MXA}}}(c.k),\ c.\mathcal{M}\right)$$

Moreover, we introduce functions transforming a non-running thread into a running one and the other way around.

Definition 8.8 (Thread Transformation). Given a configuration $th \in \mathbb{K}^{nrun}_{Th}$ of a non-running thread, special purpose registers spr, and $tlb \in \mathbb{C}_{tlb}$ from the configuration of the current threads, we get a corresponding configuration of a running thread by the function

$$wakeup(th, spr, tlb) \overset{def}{\equiv} (k_{\mathrm{mx}},\ th.sba,\ th.mss,\ tlb)$$

with $k_{\mathrm{mx}} = (th.ac,\ th.ic,\ spr)$.

For the reverse operation on a running thread with a configuration $th \in \mathbb{K}^{mx}_{Th}$ we define

$$sleep(th) \overset{def}{\equiv} (th.k_{\mathrm{mx}}.ac,\ th.k_{\mathrm{mx}}.ic,\ th.sba,\ th.mss,\ pid_{run}(th))$$

As we mentioned before, we consider the model of threads after the system startup when at least one thread is created for each processor. We will distinguish such threads by their identifiers.

Definition 8.9 (Initial Thread). Under the *initial thread* we understand a thread created for a given processor during initialization of the kernel or a process creation. Such a thread can be always scheduled and never exits. For clarity in this work, we assume that *the initial thread of a given processor has the identifier equal to the ID of this processor*.

Obviously, we are interested in the semantics of the machine defined on well-formed configurations. First, we introduce an auxiliary definition for the configuration of the processor's threads.

Definition 8.10 (Well-Formed Processor's Threads). A configuration $k \in \mathbb{K}^{proc}_{Th}$ of processor's threads is considered to be well-formed if (i) the current thread is present and has a configuration of the running thread, (ii) all other ready for scheduling threads have configurations of non-running ones, (iii) – (iv) all non-running threads have the processor affinity of the running thread, (v) threads cannot be ready for scheduling and finished simultaneously, (vi) the contexts of ready for scheduling threads are well-formed, (vii) the stacks of all threads do not overlap, and (viii) the initial thread for the processor is always present in the model.

Formally, using the shorthands

$$\begin{aligned} thr(t) &\equiv k.ready(t) & Dr &\equiv \mathrm{dom}\,(k.ready) \\ thf(t) &\equiv k.fin(t) & Df &\equiv \mathrm{dom}\,(k.fin) \\ sba_t &\equiv thr(t).sba & mss_t &\equiv thr(t).mss \end{aligned}$$

and for $ths : \mathbb{N}_{mtid} \rightharpoonup \mathbb{K}^{run}_{Th} \cup \mathbb{K}^{nrun}_{Th}$

$$\begin{aligned} valid^{stack}_{Th}(ths) \overset{def}{\equiv} &\ \forall p, q \in \mathrm{dom}\,(ths)\,.\, p \neq q \implies \\ &\ A^{stack}_{\mathrm{MX}}(sba_p, mss_p) \cap A^{stack}_{\mathrm{MX}}(sba_q, mss_q) = \emptyset \end{aligned}$$

we define

$$
\begin{aligned}
&\mathit{wfth}^{\pi,\theta}_{proc}(k) \overset{def}{\equiv}\\
&\quad \text{(i)} \quad k.ct \in \mathit{Dr} \wedge \mathit{thr}(k.ct) \in \mathbb{K}^{run}_{Th}\\
&\quad \text{(ii)} \quad \forall t \in \mathit{Dr} \setminus \{k.ct\}.\ \mathit{thr}(t) \in \mathbb{K}^{nrun}_{Th}\\
&\quad \text{(iii)} \quad \forall t \in \mathit{Dr} \setminus \{k.ct\}.\ \mathit{thr}(t).pid = pid_{run}(\mathit{thr}(k.ct))\\
&\quad \text{(iv)} \quad \forall t \in \mathit{Df}.\ \mathit{thf}(t).pid = pid_{run}(\mathit{thr}(k.ct))\\
&\quad \text{(v)} \quad \mathit{Dr} \cap \mathit{Df} = \emptyset\\
&\quad \text{(vi)} \quad \forall t \in \mathit{Dr} \setminus \{k.ct\}.\ \mathit{wfcntx}^{\pi,\theta}_{\mathrm{MX}}(\mathit{thr}(t).ac, \mathit{thr}(t).ic)\\
&\quad \text{(vii)} \quad \mathit{valid}^{stack}_{Th}(k.ready \uplus k.\mathit{fin})\\
&\quad \text{(viii)} \quad pid_{run}(\mathit{thr}(k.ct)) \in \mathit{Dr}
\end{aligned}
$$

As a special case of non-running threads we consider newly created threads that can be either in the list new of the sequential machine configuration or even in $ready$ after acquiring (shown in the transition function later). The stacks of such threads have one C-IL frame of the corresponding entry function.

Definition 8.11 (Well-Formed Configuration of New Thread). Given a configuration $th \in \mathbb{K}^{nrun}_{Th}$ of a non-running thread, we say that it is a well-formed new thread if it has an empty list of the inactive context and the active context is represented by a single well-formed C-IL frame of a thread entry function f implemented in π. The location counter of the frame points to the beginning of the function's body. All local variables (except the parameter) in $\mathcal{M}'_{\mathcal{E}}$ are initialized with zeros.

$$
\begin{aligned}
&\mathit{wfth}^{\pi,\theta}_{new}(th) \overset{def}{\equiv}\\
&\quad \text{(i)} \quad |th.ac| = 1 \wedge th.ac[1] \in \mathit{frame}_{\mathrm{CIL}} \wedge th.ic = \varepsilon\\
&\quad \text{(ii)} \quad \mathit{wfstack}^{\pi,\theta}_{\mathrm{CIL}}(th.ac)\\
&\quad \text{(iii)} \quad th.ac[1] = (f, \bot, 1, \mathcal{M}'_{\mathcal{E}})
\end{aligned}
$$

The formal definition of the initialization for these components f and $\mathcal{M}'_{\mathcal{E}}$ will be considered in the transition function and we skip it here for brevity.

Definition 8.12 (Well-Formed Configuration of Sequential Machine for Kernel Threads). Finally, the well-formedness of machine configurations $c \in \mathbb{C}_{Th}$ requires that (i) the configuration of the processor's threads is well-formed, (ii) the MXA machine configuration $c_{\mathrm{mxa}} \equiv \mathit{conf}^{curT}_{\mathrm{MXA}}(c)$ obtained for the current thread is well-formed, and (iii) if the thread can access the shared resources new and free, then (a) all new threads have well-fomed configurations, (b) all stacks in the machine do not overlap, and (c) the same thread cannot be in a few states simultaneously.

Using the shorthands Dr, Df from Definition 8.10, we have

$$
\begin{aligned}
&\mathit{wfconf}^{\pi,\theta,cba}_{Th}(c) \overset{def}{\equiv}\\
&\quad \text{(i)} \quad \mathit{wfth}^{\pi,\theta}_{proc}(c.k)\\
&\quad \text{(ii)} \quad \mathit{wfconf}^{\pi,\theta,cba}_{\mathrm{MXA}}(c_{\mathrm{mxa}})\\
&\quad \text{(iii)} \quad c.ap = 0 \vee c.ap = pid_{cur}(c) \implies\\
&\qquad \text{(a)} \quad \forall t \in \mathrm{dom}\,(c.new)\,.\ \mathit{wfth}^{\pi,\theta}_{new}(c.new(t))\\
&\qquad \text{(b)} \quad \mathit{valid}^{stack}_{Th}(c.k.ready \uplus c.k.\mathit{fin} \uplus c.new)\\
&\qquad \text{(c)} \quad \forall T, T' \in \{\mathit{Dr}, \mathit{Df}, \mathrm{dom}\,(c.new)\,, c.\mathit{free}\}.\ T \neq T' \implies T \cap T' = \emptyset
\end{aligned}
$$

Note that condition in (iii) is used in order to satisfy the requirements on the well-formedness introduced in Assumption 2.2 from Chapter 2.

Definition 8.13 (Transition Function of Sequential Machine for Kernel Threads). For the given program $\pi = (\pi_\mu, \pi_{\text{cil}}) \in \mathit{Prog}_{\text{MX}}$ from Section 8.1.1, the environment parameters $\theta \in \mathit{Params}_{\text{CIL}}$, and the code base addresses $\mathit{cba} = (\mathit{cba}_{\text{cil}}, \mathit{cba}_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, the transitions of the sequential machine for kernel threads are defined by the partial function

$$\delta_{Th}^{\pi,\theta,cba} : \mathbb{C}_{Th} \times \Sigma_{\text{MXA}} \rightharpoonup \mathbb{C}_{Th\perp}$$

where $\mathbb{C}_{Th\perp} \overset{def}{\equiv} \mathbb{C}_{Th} \cup \{\perp\}$ contains the error state, coming from the mixed machine semantics as well as the kernel threads semantics being defined here. The input alphabet Σ_{MXA} serves for the same purposes as in Definition 7.26 because the current thread can execute inline assembly and guest / process steps.

Given a configuration $c \in \mathbb{C}_{Th}$, an input $in \in \Sigma_{\text{MXA}}$, and using the shorthands

$$c_{\text{mxa}} \equiv \mathit{conf}_{\text{MXA}}^{\mathit{curT}}(c) \qquad c_{\text{mx}} \equiv \mathit{conf}_{\text{MX}}^{\text{MXA}}(c_{\text{mxa}})$$

$$c_{\text{cil}} \equiv \mathit{conf}_{\text{CIL}}(c_{\text{mx}}) \qquad \mathit{stmt}_{\text{cil}} \equiv \mathit{stmt}_{next}(c_{\text{cil}}, \pi_{\text{cil}})$$

we now define in detail the steps of the sequential machine for kernel threads. Among those we distinguish the primitives' execution and the cases similar to the extended mixed machine MXA but simplified because we will restrict the stack substitution in our model.

MXA Machine Steps

- Mixed machine step:

$$mx(c_{\text{mxa}}.k) \wedge in \in \Sigma_{\text{MX}} \wedge \neg \mathit{start}_{isa}^{\pi}(c_{\text{mxa}}) \wedge$$
$$\big(\mathit{cil}(c_{\text{mx}}.ac) \wedge \mathit{stmt}_{\text{cil}} = \textbf{call } f(E) \implies f \notin \mathbb{F}_{name}^{prim}\big)$$

When the machine performs a pure MX step without a primitive call or switching to inline assembly, the semantics is the same as for MXA in this case.

So, performing the MX step

$$c'_{\text{mx}} \equiv \delta_{\text{MX}}^{\pi,\theta}(c_{\text{mx}}, in)$$

we get again the new configuration of the MX thread for $c'_{\text{mx}} \neq \perp$

$$k'_{\text{mx}} \equiv (c'_{\text{mx}}.ac,\ c'_{\text{mx}}.ic,\ c'_{\text{mx}}.spr)$$

and compose the new configuration $c' \in \mathbb{C}_{Th}$ such that only the configuration of the current thread and the memory are updated as

$$c'.k.\mathit{ready}(c.k.ct).k_{\text{mx}} = k'_{\text{mx}} \qquad c'.\mathcal{M} = c'_{\text{mx}}.\mathcal{M}$$

All other components of c' are the same as in c' and we skip here the full formal definition for brevity.

Hence, the result of the step of the machine for kernel threads is defined as

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c' & : c'_{\text{mx}} \neq \perp \\ \perp & : \text{otherwise} \end{cases}$$

- Switch to inline assembly:

$$start^{\pi}_{isa}(c_{\mathrm{mxa}}) \wedge in \in \Sigma_{\mathrm{A}} \wedge$$
$$\Big(\mathrm{dom}\,(in.m_{st}) = A^{\pi,\theta}_{stack}(c_{\mathrm{mxa}}, info, c_{\mathrm{mxa}}.k.sba)\Big) \wedge suit^{\pi,\theta,cba}_{start}(c_{\mathrm{mxa}}, in, info)$$

  Similarly to the definition of the same step for the MXA machine we execute the first instruction of the inline assembly:

$$d' \equiv \delta_{r\mathrm{MIPS}}\left(d, \left(\mathtt{core}, w_I, w_D, 0^{256}\right)\right)$$

  with $d \equiv conf^{start}_{r\mathrm{MIPS}}(c_{\mathrm{mxa}}, in)$ and $w_I = w_D = \epsilon\, \mathbb{C}_{walk}$.

  However, if $end^{\pi,\theta,cba}_{isa}(d'.cpu)$ holds, in contrast to the MXA machine semantics where one attempts to find a new pair of the stack base address and stack maximal size, here we try to reconstruct the stack identified only by $sba_{cur}(c)$ and $mss_{cur}(c)$ belonging to the current thread, what is indicated by

$$\begin{aligned} matchst_{cur}(c, d') \overset{def}{\equiv}\ & spv', bpv' \in A^{stack}_{\mathrm{MX}}\,(sba_{cur}(c), mss_{cur}(c)) \wedge \\ & \langle spv' \rangle \leq \langle bpv' \rangle \end{aligned}$$

  with $spv' \equiv d'.cpu.core.gpr(sp)$ and $bpv' \equiv d'.cpu.core.gpr(bp)$. If this condition does not hold, we generate the run-time error $\perp$. Otherwise, one tries to reconstruct the MX thread configuration

$$k'_{\mathrm{mx}} \equiv R^{\pi,\theta}_{\mathbb{K}_{\mathrm{MX}}}\,(d', info, cba, sba_{cur}(c))$$

  and in case of $k'_{\mathrm{mx}} \neq \perp$ computes the new configuration of the running thread

$$th' \equiv (k'_{\mathrm{mx}}, sba_{cur}(c), mss_{cur}(c), d'.cpu.tlb)$$

  Hence, the transition function for this step is defined as follows:

$$\delta^{\pi,\theta,cba}_{Th}(c, in) \overset{def}{\equiv} \begin{cases} c' & : end^{\pi,\theta,cba}_{isa}(d'.cpu) \wedge \\ & \quad matchst_{cur}(c, d') \wedge k'_{\mathrm{mx}} \neq \perp \\ \perp & : end^{\pi,\theta,cba}_{isa}(d'.cpu) \wedge \\ & \quad \neg matchst_{cur}(c, d') \\ c'' & : \text{otherwise} \end{cases} \tag{8.1}$$

  where depending on the case we get from c either the machine configuration $c' \in \mathbb{C}_{Th}$ with the updated MX thread such that

$$c'.k.ready(c.k.ct) = th' \qquad c'.\mathcal{M} = d'.m$$

  or the configuration $c'' \in \mathbb{C}_{Th}$ where only the processor and memory components are updated

$$c''.k.ready(c.k.ct).cpu = d'.cpu \qquad c''.\mathcal{M} = d'.m$$

- Inline assembly, compiled code, or guest/process step:

$$isa(c_{\mathrm{mxa}}.k) \wedge in \in \Sigma_{\mathrm{MIPS}} \wedge$$
$$(in = (\mathtt{core}, w_I, w_D, eev) \implies \neg eev[0]) \wedge$$
$$\delta_{r\mathrm{MIPS}}\left(conf^{\mathrm{MXA}}_{r\mathrm{MIPS}}(c_{\mathrm{mxa}}), in\right) = d'$$

where d' is some SB reduced MIPS-86 configuration such that the transition function $\delta_{r\mathrm{MIPS}}$ is defined.

Similarly to the MXA semantics, we use this d' and compute k'_{mx} and th' as in the previous case. Then, $\delta_{Th}^{\pi,\theta,cba}(c, in)$ is again defined by the equation (8.1).

Primitives' Execution

In order to indicate the execution of any of the aforementioned primitives, we define a predicate requiring that (i) the active context of the current thread is C-IL and (ii) a primitive is called by its name such that the types of all parameters match the function declaration:

$$
\begin{aligned}
&prim_{Th}^{\pi,\theta}(c) \overset{def}{\equiv} \\
&\quad \textit{(i)} \quad mx(c_{\mathrm{mxa}}.k) \wedge cil(c_{\mathrm{mx}}.ac) \\
&\quad \textit{(ii)} \quad \exists f \in \mathbb{F}_{name}^{prim}.\ stmt_{\mathrm{cil}} = \mathbf{call}\ f(E) \wedge \\
&\qquad\qquad |E| = \mathcal{F}_{\pi}^{\theta}(f).npar \wedge \forall i \in [1:|E|].\ \mathcal{F}_{\pi}^{\theta}(f).V[i] = (v,t) \implies \tau_{c_{\mathrm{cil}}}^{\pi,\theta}(E[i]) = t
\end{aligned}
$$

We also introduce the function $inc_{loc}^{cur}(c)$ incrementing the location counter of the current thread in this case:

$$inc_{loc}^{cur}(c) \overset{def}{\equiv} c'$$

where c' is obtained from c by only updating

$$c'.k.ready(c.k.ct).k_{\mathrm{mx}}.ac = inc_{loc}(c_{\mathrm{cil}}).stack$$

Then, if the predicate $prim_{Th}^{\pi,\theta}(c)$ holds, depending of the primitive's name we distinguish between the following cases:

- Thread switch: $stmt_{\mathrm{cil}} = \mathbf{call}\ thread\_run(E)$

  From the list of arguments E with a single argument $e_{tid} = E$ we compute by expression evaluation the value $[\![e_{tid}]\!]_{c_{\mathrm{cil}}}^{\pi_{\mathrm{cil}},\theta} = \mathbf{val}(a, \mathbf{u32})$ and the identifiers of threads to be switched:

  $$tid_{to} \equiv \langle a \rangle \qquad tid_{from} \equiv c.k.ct$$

  The thread switch can be performed successfully if tid_{to} is present among threads ready for scheduling, what is indicated by the predicate

  $$thswitch^{\pi,\theta}(c) \overset{def}{\equiv} tid_{to} \neq tid_{from} \wedge tid_{to} \in \mathrm{dom}\,(c.k.ready)$$

  In this case, by incrementing the location counter of the current thread and setting up the current thread component to tid_{to} we get intermediate configurations

  $$c' \equiv inc_{loc}^{cur}(c) \qquad c'' \equiv c'\big[k := c'.k[ct := tid_{to}]\big]$$

  In turn, the configurations of the threads before and after the transformation needed for the switch are

  $$
  \begin{aligned}
  &th_{from} \equiv th_{cur}(c') && th_{to} \equiv th_{cur}(c'') \\
  &th'_{from} \equiv sleep\,(th_{from}) && th'_{to} \equiv wakeup\,(th_{to}, spr', tlb')
  \end{aligned}
  $$

  with $spr' \equiv th_{from}.k_{\mathrm{mx}}.spr$ and $tlb' \equiv th_{from}.tlb$.

Therefore, the final configuration c''' of the machine with kernel threads in case of the successful switch is obtained from c'' by updating the thread configurations involved into the switch as follows:

$$c'''.k.ready(tid_{from}) = th'_{from} \qquad c'''.k.ready(tid_{to}) = th'_{to}$$

All other components of c''' are equal to the same ones from c''.

Hence, the result of the thread switch is defined as

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c''' & : \ thswitch^{\pi,\theta}(c) \\ c' & : \ tid_{to} = tid_{from} \\ \bot & : \ \text{otherwise} \end{cases}$$

- Thread exit: $stmt_{\text{cil}} = \textbf{call}\ thread\_exit\_to(E)$

  In order to define the semantics of the thread exit, we use tid_{to}, tid_{from}, c', c'', th'_{from}, and th'_{to} computed in the previous case.

  In comparison to the thread switch, the successful exit to another thread can only be performed by a thread that is not initial for the processor (see Definitions 8.9 and 8.10). Therefore, if the conditions

$$thexit^{\pi,\theta}(c) \overset{def}{\equiv} thswitch^{\pi,\theta}(c) \wedge tid_{from} \neq pid_{cur}(c)$$

  hold, we obtain the new machine configuration c''' by moving the exiting thread to the finished ones and updating c'' in the following way:

$$c'''.k.ready(x) = \begin{cases} th'_{to} & : \ x = tid_{to} \\ c''.k.ready(x) & : \ x \in \text{dom}\,(c.k.ready) \setminus \{tid_{to}, tid_{from}\} \\ undefined & : \text{otherwise} \end{cases}$$

$$c'''.k.fin(x) = \begin{cases} th'_{from} & : \ x = tid_{from} \\ c''.k.fin(x) & : \ \text{otherwise} \end{cases}$$

  Finally, the transition function for the thread exit step is defined as

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c''' & : \ thexit^{\pi,\theta}(c) \\ c' & : \ tid_{to} = tid_{from} \\ \bot & : \ \text{otherwise} \end{cases}$$

  Note that the introduction of this primitive in the semantics allows to return from the function even in the bottom frame of the stack, what is not possible in MASM, C-IL, and MX considered in the previous chapters of this thesis and [Sha12, Sch13].

- Thread deletion: $stmt_{\text{cil}} = \textbf{call}\ thread\_delete(E)$

  Performing the expression evaluation of the single argument $E = e_{tid}$ we get the identifier of a thread to be deleted

$$[\![e_{tid}]\!]_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta} = \textbf{val}(a, \textbf{u32}) \qquad tid \equiv \langle a \rangle$$

The result of the computation depends whether the thread can access the component $c.\mathit{free}$. If it is not used by some other processor (or free) and the thread tid is finished

$$c.ap = 0 \wedge \mathit{tid} \in \mathrm{dom}\,(c.k.\mathit{fin})$$

one computes the resulting configuration c' by updating the following components in c:

$$c'.\mathit{free} = c.\mathit{free} \cup \{\mathit{tid}\} \qquad\qquad c'.ap = \mathit{pid}_{cur}(c)$$

$$c'.k.\mathit{fin}(x) = \begin{cases} c.k.\mathit{fin}(x) & : \ x \neq \mathit{tid} \\ \mathit{undefined} & : \ \text{otherwise} \end{cases}$$

where $c'.ap$ indicates that $c'.\mathit{free}$ is now in use (successfully acquired and becomes locked) by the processor $\mathit{pid}_{cur}(c)$ and cannot be accessed by others because the execution of the primitive is not complete yet.

Hence, the semantics of the step is defined by case split on phases corresponding to

- $c.ap \neq 0$: an unsuccessful attempt to acquire $c.\mathit{free}$,
- $c.ap = 0$: its acquisition with the configuration update considered above, and
- $c.ap = \mathit{pid}_{cur}(c)$: finishing the execution of the primitive by releasing $c.\mathit{free}$ and increasing the location.

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c' & : \ c.ap = 0 \wedge \mathit{tid} \in \mathrm{dom}\,(c.k.\mathit{fin}) \\ \perp & : \ c.ap = 0 \wedge \mathit{tid} \notin \mathrm{dom}\,(c.k.\mathit{fin}) \\ \mathit{inc}_{loc}^{cur}(c)[ap := 0] & : \ c.ap = \mathit{pid}_{cur}(c) \\ c & : \ \text{otherwise} \end{cases}$$

Note that according to the last case in the definition of $\delta_{Th}^{\pi,\theta,cba}(c, in)$, we stay at the primitive call as long as we cannot access $c.\mathit{free}$ occupied by some other processor. At the beginning of each phase we will obviously require that the machine configuration c is well-formed wrt. Definition 8.10.

It is important to mention that in comparison to the Verisoft project [Ver10] and [PBLS15] where the execution of a CVM primitive (see also related works in Chapter 1) on a single-core machine is modelled as a single atomic step, this thesis is the first document giving the specification of primitives using the lock protection in their implementation on the multi-core processor architecture.

We proceed with the semantics of the tread creation and acquisition, which also perform global operations (recall Figure 8.1) and, therefore, are specified in a similar way.

- Thread creation: $\mathit{stmt}_{\mathrm{cil}} = \textbf{call}\ \mathit{thread\_create}(E)$

  Let the list E of expressions passed into the function as parameters be

  $$E = e_{tid} \circ e_{pid} \circ e_{fn} \circ e_{arg} \circ e_{sba} \circ e_{mss}$$

  We evaluate the expressions with subscripts $s \in \{\mathit{sba}, \mathit{mss}, \mathit{pid}, \mathit{tid}\}$ as

  $$[\![e_s]\!]_{c_{\mathrm{cil}}}^{\pi_{\mathrm{cil}},\theta} = \mathbf{val}(a_s, t_s)$$

  Then, if the conditions

  $$c.ap = 0 \wedge \langle a_{tid} \rangle \in \mathrm{dom}\,(c.\mathit{free})$$

hold, we compute a new machine configuration $c' \in \mathbb{C}_{Th}$ by updating c in the following way:

$$c'.free = c.free \setminus \{\langle a_{tid} \rangle\} \qquad c'.ap = pid_{cur}(c)$$

$$c'.new(x) = \begin{cases} th & : \ x = \langle a_{tid} \rangle \\ c.new(x) & : \ \text{otherwise} \end{cases}$$

where th is a configuration of the new thread

$$th \equiv (frame, a_{sba}, \langle a_{mss} \rangle, \langle a_{pid} \rangle)$$

with the C-IL frame composed as

$$frame \equiv (f, \perp, 1, \mathcal{M}'_{\mathcal{E}})$$

such that the function name f satisfies

$$isfunc([\![e_{fn}]\!]_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}, f, \theta) \wedge f \in \text{dom}\left(\mathcal{F}_{\pi_{\text{cil}}}^{\theta}\right) \wedge \neg ext(f, \pi_{\text{cil}}, \theta)$$

and the local memory is initialized wrt. the input argument

$$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes\left([\![e_{arg}]\!]_{c_{\text{cil}}}^{\pi_{\text{cil}},\theta}\right) & : \ \mathcal{F}_{\pi_{\text{cil}}}^{\theta}(f).V[1] = (v, t) \\ (0^8)^{size_\theta(qt2t(t))} & : \ \mathcal{F}_{\pi_{\text{cil}}}^{\theta}(f).V[i] = (v, t) \wedge i > 1 \\ undefined & : \ \text{otherwise} \end{cases}$$

Hence, using the predicate $vst(c') \equiv valid_{Th}^{stack}(c'.k.ready \uplus c'.k.fin \uplus c'.new)$ indicating that the stack of the new thread does not overlap with stacks of other threads, the step is defined as

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c' & : \ c.ap = 0 \wedge \langle a_{tid} \rangle \in \text{dom}\,(c.free) \wedge vst(c') \\ inc_{loc}^{cur}(c)[ap := 0] & : \ c.ap = pid_{cur}(c) \\ \perp & : \ c.ap = 0 \wedge (\langle a_{tid} \rangle \notin \text{dom}\,(c.free) \vee \neg vst(c')) \\ c & : \ \text{otherwise} \end{cases}$$

where the processor $pid_{cur}(c)$ locks/releases the components *free* and new.

- Thread acquisition: $stmt_{\text{cil}} = \textbf{call } thread\_acquire()$

  By the thread acquisition we move all new threads with affinity equal to $pid_{cur}(c)$ to the configuration of the processor's threads. So, if the new threads can be accessed, i.e. $c.k.ac = 0$ holds, we update the configuration c in a way such that c' differs from c in the following:

$$c'.k.ready(x) = \begin{cases} c.new(x) & : \ c.new(x).pid = pid_{cur}(c) \\ c.k.ready(x) & : \ \text{otherwise} \end{cases}$$

$$c'.new(x) = \begin{cases} c.new(x) & : \ c.new(x).pid \neq pid_{cur}(c) \\ undefined & : \ \text{otherwise} \end{cases}$$

$$c'.ap = pid_{cur}(c)$$

  Hence, the step of the machine is defined as

$$\delta_{Th}^{\pi,\theta,cba}(c, in) \overset{def}{\equiv} \begin{cases} c' & : \ c.ap = 0 \\ inc_{loc}^{cur}(c)[ap := 0] & : \ c.ap = pid_{cur}(c) \\ c & : \ \text{otherwise} \end{cases}$$

In other cases or when the expression evaluation cannot be successfully performed, the result of the step $\delta_{Th}^{\pi,\theta,cba}(c, in)$ is undefined.

8.1.3 Concurrent Semantics of Kernel Threads

Now, the concurrent model of kernel threads can be easily defined on the base of the definitions from the previous section.

Definition 8.14 (Configuration of Concurrent Machine for Kernel Threads). Configurations of the concurrent machine for kernel threads with np running threads (equal to the number of processors) are represented by tuples that differ from $\mathbb{C}_{Th}$ only by the mapping k from the processor identifier to the configuration of the processor's threads.

$$\mathbb{C}_{cTh} \overset{def}{\equiv} \left(k : \mathbb{N}_{np} \to \mathbb{K}_{Th}^{proc},\ new : \mathbb{N}_{mtid} \rightharpoonup \mathbb{K}_{Th}^{nrun},\ free \in 2^{\mathbb{N}_{mtid}},\ ap \in [0 : np],\ \mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^{8}\right)$$

Definition 8.15 (Sequential Machine Configuration from the Concurrent Machine for Kernel Threads). For a configuration $c \in \mathbb{C}_{cTh}$ of the concurrent machine for kernel threads and $t \in \mathbb{N}_{np}$ we define $conf_{Th}(c, t) \in \mathbb{C}_{Th}$ such that

$$conf_{Th}(c, t) \overset{def}{\equiv} (c.k(t), c.new, c.free, c.ap, c.\mathcal{M})$$

Definition 8.16 (Transition Function of the Concurrent Machine for Kernel Threads). Then, for $\pi = (\pi_\mu, \pi_{\text{cil}}) \in Prog_{\text{MX}}$, $\theta \in Params_{\text{CIL}}$, and $cba = (cba_{\text{cil}}, cba_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, the transitions of the concurrent machine for kernel threads are defined by the function

$$\delta_{cTh}^{\pi,\theta,cba} : \mathbb{C}_{cTh} \times \mathbb{N}_{np} \times \Sigma_{\text{MXA}} \rightharpoonup \mathbb{C}_{cTh\perp}$$

such that for a configuration $c \in \mathbb{C}_{cTh}$, a processor with index $t \in \mathbb{N}_{np}$ running a kernel thread, and an input $in \in \Sigma_{\text{MXA}}$, the result of the transition is computed as

$$\delta_{cTh}^{\pi,\theta,cba}(c, t, in) \overset{def}{\equiv} \begin{cases} \left(c.k[t \mapsto c'_t.k],\ c'_t.new,\ c'_t.free,\ c'_t.ap,\ c'_t.\mathcal{M}\right) & : c'_t \neq \perp \\ \perp & : \text{otherwise} \end{cases}$$

where c'_t is the next configuration of the sequential machine for threads belonging to the processor t

$$c'_t \equiv \delta_{Th}^{\pi,\theta,cba}\left(conf_{Th}(c, t), in\right)$$

If c'_t does not exist, the result of the step $\delta_{cTh}^{\pi,\theta,cba}(c, t, in)$ is undefined.

Naturally, the transition function for t assumes that the corresponding configuration $conf_{Th}(c, t)$ is well-formed according to Definition 8.12.

8.2 Implementation of Kernel Threads

The abstract machine for kernel threads considered above is implemented by the MXA machine executing a program obtained from π and the framework for kernel threads.

8.2.1 Framework for Kernel Threads

The kernel threads framework is a program $\pi_{th} = \left(\pi_\mu^{th}, \pi_{\text{cil}}^{th}\right) \in Prog_{\text{MX}}$ containing the implementation of the aforementioned primitives and all data structures and parameters needed for it. This program is linked [Tsy09, IdR09] with π from Section 8.1.1, compiled together, and placed into the memory.

8.2.1.1 Data Structures

First, we consider the data structures and constants declared in the C-IL program π^{th}_{cil}.

In the program we fix the number of running kernel threads to the constant NP equal to np. In turn, the constant TID_MAX corresponds to the maximal thread identifier $mtid$. Moreover, we denote thread states by the following constants:

```
#define TS_FREE 0
#define TS_NEW 1
#define TS_RUNNING 2
#define TS_SLEEPING 3
#define TS_FINISHED 4
```

The C-IL composite type **struct** *TCB* represents the *thread control block (TCB)* containing all data characterizing a kernel thread:

```
typedef struct TCB {
  u32 tid; // thread ID
  u32 state; // thread state
  u32 pid; // processor ID
  thfun_t fn; // thread entry function
  u32 arg; // thread function argument
  void *sba; // stack base address
  u32 mss; // maximal stack size
  thcntx_t cntx; // thread context
} TCB_t;
```

where the *thread context* contains values of stack and frame base pointers of the topmost frame if the corresponding thread is non-running:

```
typedef struct Thcntx {
  void *sp;
  void *bp;
} thcntx_t;
```

The components of the abstract machine configuration are implemented as doubly linked lists with nodes containing TCBs:

```
typedef struct Thread {
  TCB_t tcb; // thread control block
  struct Thread *next; // link to the next node
  struct Thread *prev; // link to the previous node
} thread_t;
```

such that the nodes for all threads available in the kernel are kept in an array *threads* with elements numbered by thread identifiers starting from 1:

```
thread_t threads [TID_MAX+1];
```

The element with index 0 is not used for simplicity here. Obviously, if a thread is not present, its state is marked as TS_FREE and a new thread can be created with the same identifier.

For modeling the lists we support pointers and arrays of pointers to tails and heads of the lists:

```
// Pointer to a global list of new threads
thread_t *new_hd;
thread_t *new_tl;

// Pointer to a global list of free threads
thread_t *free_hd;
thread_t *free_tl;
```

Listing 8.1: Stack switch procedure in π_{μ}^{th}.

```
switch_stack USES sv1 sv2 sv3 sv4 sv5 sv6 sv7 sv8
1: asm {
   // save sp and bp to the thread context
   sw sp i1 0
   sw bp i2 0
   // restore sp and bp from another thread context
   addi sp i3 0
   addi bp i4 0
   // load argument for the entry function of a new thread
   lw i1 bp 28 };
2: ret;
```

```
// Pointers to local lists of running and ready for scheduling threads
thread_t *ready_hd [NP+1];
thread_t *ready_tl [NP+1];

// Pointers to local lists of finished threads
thread_t *fin_hd [NP+1];
thread_t *fin_tl [NP+1];
```

Note again, that we assume the processors to be numbered starting from 1 and, therefore, do not use the elements with index 0.

For the work on such lists we implement the classical operations (see Appendix B):

- *search_by_tid* – search for a list node of a thread with a given identifier,
- *search_by_pid* – search for a list node of a thread with a given processor identifier,
- *remove* – removing a thread list node,
- *insert_to_end* – inserting a node to the end of a thread list.

The accesses to the global lists of new and free threads are locked and implemented with the help of acquiring and releasing the spinlock [3]

```
lock_t lock;
```

by the C-IL functions (see Appendix A):

```
void acquire_lock(lock_t *lock);
void release_lock(lock_t *lock);
```

In order to determine the current thread for each processor, in the program π_{cil}^{th} we also declare an array of the thread identifiers:

```
u32 current [NP+1];
```

8.2.1.2 Implementation of Primitives

The thread switch is based on the stack substitution implemented in the MASM program π_{μ}^{th} with inline assembly by the procedure *switch_stack* (Listing 8.1)[4]. The idea of using such a procedure was taken from [Han96] and the former joint work of the author of this thesis with Ulan Degenbaev in the Verisoft XT project [The11].

[3]The verification of operations on locks can be found in [HL09]

[4]For register names used in MASM programs, please, refer Table 5.1 in Section 5.1.3.

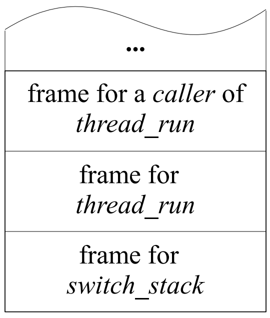

Figure 8.2: Layout of the stacks for the running and sleeping threads in *switch_stack* before the return.

The procedure is declared as external in the C-IL program π_{cil}^{th}:

```
extern void switch_stack(void **oldsp, void **oldbp, void *newsp, void *newbp);
```

The first two parameters are the pointers to the fields in the context of the current thread. The other ones are the values of the stack and frame base pointers of a thread to be scheduled.

Loading the argument is only needed for switching to the stack of a newly created thread because we have to provide the argument via register i_1 according to the compiler calling convention introduced in Section 5.1.3.

Except the MASM procedure *get_pid* (see Appendix A) reading the processor ID from the corresponding SPR (Table 3.4), all other functions of the framework for kernel threads are implemented in π_{cil}^{th}. Here, we present the implementation of every primitive and describe them shortly.

The *thread switch* (Listing 8.2) is performed on the local data structures and comprises the following steps. One obtains pointers (variables $from$ and to in the code) to the nodes for the current thread and the one we switched to if it is present among the threads ready for scheduling (lines 1-4). In case of success, we change the states of the threads and modify the ID of the thread current for the running processor (lines 6-8). The most tricky part of the thread switch is made after the call of *switch_stack* (line 9) in the body of this procedure.

First, in the inline assembly (line 1) we substitute the stacks by saving the current content of the registers sp and bp and restoring them from the context belonging to the thread pointed by to. Note that at this moment the stack of the thread pointed by $from$ has at least three frames on the top (Figure 8.2): for the caller of *thread_run*, and the calls of *thread_run*, *switch_stack* respectively. Its layout will be preserved in the memory until the thread is scheduled again. Second, by executing **ret** (line 2) one either returns the control to the sleeping thread we switch to, or starts executing the entry function of a new thread.

In case of the sleeping thread, we continue the execution of the code of this thread starting from the C-IL statement **return** in the function *thread_run* (line 10). This operation is feasible because the stack of the sleeping thread has also the layout formed by the call of *thread_run* and depicted on Figure 8.2. Therefore, after the further return from *thread_run*, one restores the thread's execution at the next statement after this call.

In case of the new thread, after **ret**, the compiled code of the prologue of the entry function is executed. Obviously, the stack of the newly created thread must be prepared properly by the primitive *thread_create* so that one can correctly return from *switch_stack*.

During the *thread creation* (Listing 8.3) we access the global lists of free and new threads, and, therefore, guarantee the atomicity of the operation via acquiring (line 1) and releasing (line 24)

Listing 8.2: Thread switch.

```
void thread_run(u32 tid)
{
// Local variables:
u32 pid;
u32 ct;
thread_t *from, *to;
// Function body:
1: pid = call get_pid();
2: ct = current[pid];
3: from = &threads[ct];
4: to = call search_by_tid(ready_hd[pid], tid);
5: ifnot (to != 0 && ct != tid) goto 10;
6: (from->tcb).state = TS_SLEEPING;
7: (to->tcb).state = TS_RUNNING;
8: current[pid] = tid;
9: switch_stack(&((from->tcb).cntx.sp), &((from->tcb).cntx.bp),
               (to->tcb).cntx.sp, (to->tcb).cntx.bp);
10: return;
}
```

Listing 8.3: Thread creation.

```
void thread_create(u32 tid, u32 pid, thfun_t fn, u32 arg, void *sba, u32 mss)
{
// Local variables:
thread_t *th;
i32 *sp;
// Function body:
1: call acquire_lock(&lock);
2: th = call search_by_tid(free_hd, tid);
3: ifnot (th != 0) goto 24;
// – initialization of TCB
4: (th->tcb).tid = tid;
5: (th->tcb).state = TS_NEW;
6: (th->tcb).pid = pid;
7: (th->tcb).fn = fn;
8: (th->tcb).arg = arg;
9: (th->tcb).sba = sba;
10: (th->tcb).mss = mss;
// –– prepare frame part for fn
11: sp = (i32*)((i32)sba - 3);   // address for arg
12: *sp = arg;         // storing arg on the stack
13: sp = sp + (-1);       // space for dummy return address
// –– prepare frame for switch_stack
15: sp = sp + (-4);       // home addresses for arguments of switch_stack
16: sp = sp + (-1);       // space for return address
17: *sp = (i32*) fn;      // storing address of fn as return address
18: sp = sp + (-1);       // space for dummy pbp
19: (th->tcb).cntx.bp = (void*) sp; // save frame base pointer to the thread context
20: sp = sp + (-8);       // dummy space for all callee–save registers
21: (th->tcb).cntx.sp = (void*) sp; // save stack pointer to the thread context
// – remove from list of free threads
22: call remove(&free_hd, &free_tl, th);
// – add to list of new threads
23: call insert_to_end(&new_hd, &new_tl, th);
24: call release_lock(&lock);
25: return;
}
```

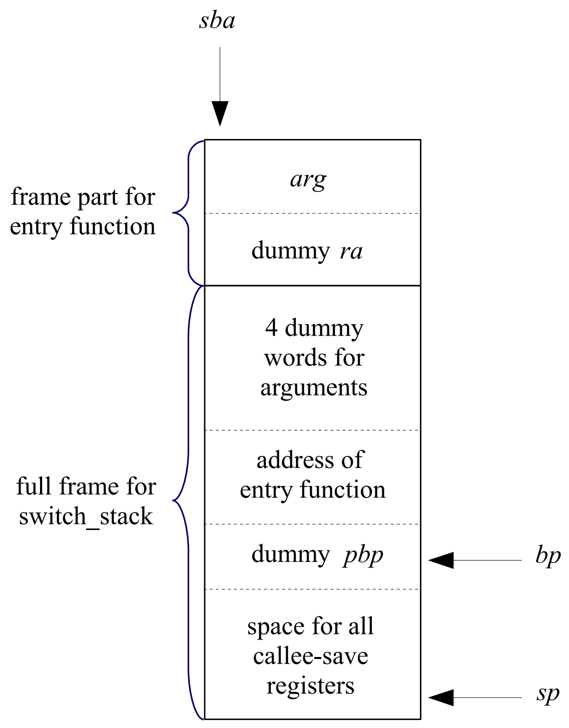
sba
frame part for entry function
arg
dummy ra
4 dummy words for arguments
address of entry function
dummy pbp
space for all callee-save registers
full frame for switch_stack
bp
sp

Figure 8.3: Physical stack layout of a newly created thread.

the spinlock. After the successful acquisition of the lock, we check whether the given thread ID is free (line 2) and then prepare the TCB (lines 4-10) as well as the stack of the new thread including the stack pointer and the frame base pointer (lines 11-21) so that the thread can be correctly scheduled. Both pointers are saved into the tread context in the TCB (lines 19 and 21). The operation finishes by adding the thread node into the list of new threads and removing it from the list of free threads (lines 22-23).

The physical stack layout of the new thread is depicted on Figure 8.3. Since the thread switch is performed by returning from *switch_stack*, the stack has the full frame for this procedure where the return address is the starting address of the entry function. Note that the value of *pbp* in this frame will not be used later and, therefore, it is left uninitialized. The prologue of the entry function will compute the actual frame base pointer on the base of the stack pointer. Moreover, the entry function is not called explicitly from C-IL. Hence, we also have to prepare the part of its frame that is supposed to be created by a caller, i.e., allocation of home addresses for arguments and the space for the return address. Since we never return from the entry function by the C-IL statement **return**, the value of the return address does not matter for the correct execution of the function. The argument, however, is stored on the stack at the home address so that the procedure *switch_stack* can put it into the register later on during the thread switch.

The *thread exit* (Listing 8.4) is similar to the thread switch except moving the thread to the list of finished ones. Note, since a thread initial for a processor is not allowed to exit, the execution of the function has no effect in such a case (lines 6 and 13).

During the *acquisition of newly created threads* (Listing 8.5) the global list of new threads is accessed in a loop. Obviously, the operation requires the protection by the same lock used in

Listing 8.4: Thread exit.

```
void thread_exit_to(u32 tid)
{
// Local variables:
u32 pid;
u32 ct;
thread_t *from, *to;
// Function body:
1: pid = call get_pid();
2: ct = current[pid];
3: ifnot (ct != pid) goto 13; // do not exit from initial threads
4: from = &threads[ct];
5: to = call search_by_tid(ready_hd[pid], tid);
6: ifnot (to != 0 && ct != tid) goto 13;
7: (from->tcb).state = TS_FINISHED;
8: call remove(&ready_hd[pid], &ready_tl[pid], from);
9: call insert_to_end(&fin_hd[pid], &fin_tl[pid], from);
10: (to->tcb).state = TS_RUNNING;
11: current[pid] = tid;
12: switch_stack(&((from->tcb).cntx.sp), &((from->tcb).cntx.bp),
                  (to->tcb).cntx.sp, (to->tcb).cntx.bp);
13: return;
}
```

the thread creation. All new threads with the matching processor affinity are deleted from this list and added into the list of threads ready for scheduling.

The implementation of the *thread deletion* working on the local and global lists of finished and free threads respectively is intuitive and given in Listing 8.6.

8.2.2 Program Linking and Obtaining the Implementation Model

As the next step, we consider how the program π of the hypervisor / OS system kernel is linked together with the framework for the kernel threads π_{th}.

For simplicity, we assume that names of declared global variables, and composite types are distinct in both programs. Moreover, the function and procedure tables are different except the additional declaration of external primitives in π_{cil}.

Definition 8.17 (Program Linking Operator). For the given programs $\pi = (\pi_\mu, \pi_{\text{cil}})$ and $\pi_{th} = \left(\pi_\mu^{th}, \pi_{\text{cil}}^{th}\right)$ we define the program linking operator

$$link(\pi, \pi_{th}) \overset{def}{\equiv} \pi_{imp}$$

where the resulting linked program $\pi_{imp} \equiv \left(\pi_\mu^{imp}, \pi_{\text{cil}}^{imp}\right)$ is computed as follows:

- linked MASM program
$$\pi_\mu^{imp}(p) = \pi_\mu \uplus \pi_\mu^{th}$$
- linked C-IL program
$$\begin{aligned}\pi_{\text{cil}}^{imp}.V_G &= \pi_{\text{cil}}.V_G \circ \pi_{\text{cil}}^{th}.V_G\\ \pi_{\text{cil}}^{imp}.T_F &= \pi_{\text{cil}}.T_F \uplus \pi_{\text{cil}}^{th}.T_F\\ \pi_{\text{cil}}^{imp}.\mathcal{F} &= \pi_{\text{cil}}.\mathcal{F}|_{D'} \uplus \pi_{\text{cil}}^{th}.\mathcal{F}\end{aligned}$$

Listing 8.5: Acquisition of newly created threads.

```
void thread_acquire()
{
// Local variables:
u32 pid;
thread_t *th;
thread_t *list;
// Function body:
1: pid = call get_pid();
2: list = new_hd;
3: call acquire_lock(&lock);
4: th = call search_by_pid(list, pid);
5: ifnot (th != 0) goto 10;
6: list = th->next;
7: call remove(&new_hd, &new_tl, th);
8: call insert_to_end(&ready_hd[pid], &ready_tl[pid], th);
9: goto 4;
10: call release_lock(&lock);
11: return;
}
```

Listing 8.6: Thread deletion.

```
void thread_delete(u32 tid)
{
// Local variables:
u32 pid;
thread_t *th;
thread_t *list;
// Function body:
1: pid = call get_pid();
2: call acquire_lock(&lock);
3: th = call search_by_tid(fin_hd[pid], tid);
4: ifnot (th != 0) goto 8;
5: (th->tcb).state = TS_FREE;
6: call remove(&fin_hd[pid], &fin_tl[pid], th);
7: call insert_to_end(&free_hd, &free_tl, th);
8: call release_lock(&lock);
9: return;
}
```

where the domain $D' \equiv \mathrm{dom}\,(\pi_{\mathrm{cil}}) \setminus \mathbb{F}^{prim}_{name}$ is restricted to all declared as well as implemented in π_{cil} functions excluding the primitives because they are present in the function table of the framework.

A more general form of linking operator for *any* C0 programs can be found in [Tsy09, IdR09].

Since the concurrent MXA machine implementing the abstract machine for kernel threads executes the linked program π_{imp}, its environment parameters $\theta_{imp} \in Params_{\mathrm{CIL}}$ are computed wrt. this program and placing the compiled code into the memory. Obviously, we require that θ can be fully obtained from θ_{imp}:

$$\begin{aligned} valid^{par}_{Th}(\theta, \theta_{imp}) \overset{def}{\equiv} & \\ \textit{(i)} \quad & \forall X \in \{size\_t, cast, intrinsics\}\,.\,\theta.X = \theta_{imp}.X \\ \textit{(ii)} \quad & \forall v \in \mathrm{dom}\,(\theta.alloc_{gvar})\,.\,\theta.alloc_{gvar}(v) = \theta_{imp}.alloc_{gvar}(v) \\ \textit{(iii)} \quad & \forall f \in \mathrm{dom}\,(\theta.\mathcal{F}_{adr})\,.\,\theta.\mathcal{F}_{adr}(f) = \theta_{imp}.\mathcal{F}_{adr}(f) \\ \textit{(iv)} \quad & \forall t \in \mathrm{dom}\,(\theta.size_{struc})\,.\,\theta.size_{struc}(t) = \theta_{imp}.size_{struc}(t) \\ \textit{(v)} \quad & \forall t \in \mathrm{dom}\,(\theta.offset_{struc})\,.\,\theta.offset_{struc}(t) = \theta_{imp}.offset_{struc}(t) \end{aligned}$$

The whole compiled program is placed at the same base addresses $cba = (cba_{\mathrm{cil}}, cba_{\mu})$ such that $valid^{code}_{\mathrm{MX}}(info_{imp}, cba)$ holds for the full compiler information $info_{imp} = (info^{imp}_{\mathrm{cil}}, info^{imp}_{\mu}) \in infoT_{\mathrm{MX}}$, computed as $info^{imp}_{\mu} = cmpl_{\mathrm{MASM}}(\pi^{imp}_{\mu})$ and $info^{imp}_{\mathrm{cil}} = cmpl_{\mathrm{CIL}}(\pi^{imp}_{\mathrm{cil}})$. Similarly to θ_{imp} and θ above, we introduce the predicate

$$valid^{cmpl}_{Th}(\pi, \pi_{th}, \pi_{imp}) \in \mathbb{B}$$

with requirements on the compilation of π and π_{imp} such that (i) the compiled codes of the kernel and the framework for kernel threads are placed after each other

$$info^{imp}_{\mathrm{cil}}.code = info_{\mathrm{cil}}.code \circ cmpl_{\mathrm{CIL}}(\pi^{th}_{\mathrm{cil}}).code$$

$$info^{imp}_{\mu}.code = info_{\mu}.code \circ cmpl_{\mathrm{MASM}}(\pi^{th}_{\mu}).code$$

and (ii) all other components of $info$ are included into $info_{imp}$. The formal definition is intuitive and we do not provide it here for brevity.

Definition 8.18 (Memory Addresses Occupied by Global Variables of the Framework for Kernel Threads). Now, having π_{th} and θ_{imp} one can compute all memory byte addresses of the global variables declared in π_{th} (see Section 8.2.1.1) and used for the implementation of the kernel threads. We denote this set by

$$A^{data}_{th}(\pi_{th}, \theta_{imp}) \subset \mathbb{B}^{32}$$

and do not provide its formal definition here for brevity.

In order to determine other parameters for the extended mixed machine, we introduce auxiliary functions computing base addresses of elements of arrays and fields of structures.

Definition 8.19 (Base Addresses of an Array's Element). For a declared array variable $(v, t) \in \mathbb{V} \times \mathbb{T}_{\mathbb{Q}}$ with $qt2t(t) = \mathbf{array}(t', n)$ for some $n \in \mathbb{N}$ we define the base address of an array's element with index $i \in \mathbb{N}_n$:

$$ba^{\theta_{imp}}_{elem}(v, t', i) \overset{def}{\equiv} \theta_{imp}.alloc_{gvar}(v) +_{32} \left(i \cdot size_{\theta_{imp}}(t')\right)_{32}$$

Definition 8.20 (Base Addresses of a Field). Given an address $a \in \mathbb{B}^{32}$ of a composite type $t \in \mathbb{T}_C$, we compute the base address of its field $f \in \mathbb{F}$ as

$$ba^{\theta_{imp}}_{field}(a, t, f) \overset{def}{\equiv} a +_{32} (\theta_{imp}.offset_{struc}(t, f))_{32}$$

Therefore, the stack information abstraction is instantiated according to the the linked program as follows:

- Base address of the stack information: $StIbas \in \left(\mathbb{B}^{32}\right)^{np}$ such that

  $$StIbas_i = ba^{\theta_{imp}}_{elem}(threads, Thread, i)$$

  where $threads \in decl(\pi_{imp}.V_G)$ is a global variable and $Thread \in \mathbb{T}_C$ is the composite type in the program.

- Size of a single stack information region in bytes:

  $$StIsize = size_{\theta_{imp}}(Thread)$$

- Address of the next stack information region:

  $$StIba_{next}(a, m) = \begin{cases} a_{next} & : \; a_{next} \neq 0^{32} \\ \bot & : \; \text{otherwise} \end{cases}$$

  where the retrieved address a_{next} is computed as

  $$a_{next} \equiv m_4\left(ba^{\theta_{imp}}_{field}(a, Thread, next)\right)$$

- Partial functions retrieving the stack base address and maximal stack size:

  $$sba(a, m) = m_4\left(ba^{\theta_{imp}}_{field}(a_{tcb}, TCB, sba)\right)$$

  $$mss(a, m) = \left\langle m_4\left(ba^{\theta_{imp}}_{field}(a_{tcb}, TCB, mss)\right)\right\rangle$$

  where a_{tcb} is the address of the TCB field in the structure $Thread$ computed as

  $$a_{tcb} \equiv ba^{\theta_{imp}}_{field}(a, Thread, tcb)$$

8.3 Sequential Correctness in Concurrent Setting

As we have seen above, since for given π and π_{th} we can easily compute the linked program and the corresponding compiler information, for brevity the following shorthands depending on π will be often used without explicit mentioning them as parameters in definitions till the end of the chapter:

$$\pi_{imp} \equiv link(\pi, \pi_{th})$$

$$info \equiv (info_{\text{cil}}, info_{\mu}) \equiv (cmpl_{\text{CIL}}(\pi_{\text{cil}}), cmpl_{\text{MASM}}(\pi_{\mu}))$$

$$info_{imp} \equiv (info^{imp}_{\text{cil}}, info^{imp}_{\mu}) \equiv \left(cmpl_{\text{CIL}}(\pi^{imp}_{\text{cil}}), cmpl_{\text{MASM}}(\pi^{imp}_{\mu})\right)$$

In turn, explicitly, we will operate with π, θ, θ_{imp}, and some abstract and implementation configurations $c \in \mathbb{C}_{Th}$ and $c_{imp} \in \mathbb{C}_{\text{MXA}}$ as well at their components.

Additionally, we will also often refer to the shorthands c_{mxa}, c_{mx}, c_{cil}, and $stmt_{\text{cil}}$ computed for $c \in \mathbb{C}_{Th}$ in Definition 8.13.

8.3.1 Consistency Points

The consistency points of the machine for kernel threads executing the program π are easily defined on the base of the MXA thread configuration obtained for the current kernel thread.

Definition 8.21 (Consistency Points of Machine for Kernel Threads). Given $k \in \mathbb{K}_{Th}^{proc}$, π, θ, and cba from above, we compute

$$cp_{Th}(k, \pi, \theta, cba) \overset{def}{\equiv} cp_{\mathrm{MXA}}\left(conf_{\mathbb{K}_{\mathrm{MXA}}}^{curT}(k), \pi, info, \theta, cba\right)$$

For $c \in \mathbb{C}_{Th}$ we simply reload the definition as

$$cp_{Th}(c, \pi, \theta, cba) \overset{def}{\equiv} cp_{Th}(c.k, \pi, \theta, cba)$$

Recall that using Definition 6.34 we already required the C-IL compiler to insert the consistency point directly before any function call. Therefore, as long as the location counter points to the call of a primitive, the current thread stays in the consistency point.

For the consistency points of the MXA machine implementing the kernel threads we will choose only those MXA consistency points from Definition 7.36 at which the simulation relation between the MXA and the machine for kernel threads will hold. The natural candidates for them are the consistency points in the program π, at every ISA step executing any compiled code of libraries used by the kernel, guest/user steps, and the $\mathcal{IO}$-points inside the implementation of the primitives in π_{th}, except the cas call in the first iteration during the lock acquisition.

Definition 8.22 (Consistency Points for MXA Machine Implementing Kernel Threads). Given an MXA thread configuration $k_{imp} \in \mathbb{K}_{\mathrm{MXA}}$ in the machine implementing the kernel threads, we define whether the thread at a consistency point by the predicate

$$\begin{aligned}
&cp_{\mathrm{MXA}}^{Th}(k_{imp}, \pi, \theta_{imp}, cba) \overset{def}{\equiv} \\
&\quad (i) \quad isa(k_{imp}) \implies \\
&\qquad\qquad outside(k_{imp}.core, info_{imp}, cba) \vee \\
&\qquad\qquad inline(k_{imp}.core, \pi_{\mu}, info_{\mu}, cba_{\mu}) \vee \\
&\qquad\qquad cp_{r\mathrm{MIPS}}^{\mathrm{MX}}(k_{imp}.core, \pi, info, \theta, cba) \\
&\quad (ii) \quad mx(k_{imp}) \implies \\
&\qquad\qquad (a) \quad masm(ac_{imp}) \implies p_{top}(ac_{imp}.stack) \in \mathrm{dom}\,(\pi_{\mu}) \\
&\qquad\qquad (b) \quad cil(ac_{imp}) \implies info_{\mathrm{cil}}.cp(f_{top}(ac_{imp}), loc_{top}(ac_{imp})) \vee \\
&\qquad\qquad\qquad\qquad cp_{\mathrm{MXA}}^{acqcas}(k_{imp}) \vee cp_{\mathrm{MXA}}^{relvol}(k_{imp})
\end{aligned}$$

where

- ac_{imp} – the active context of the MXA thread

$$ac_{imp} \equiv k_{imp}.k_{\mathrm{mx}}.ac$$

- $cp_{\mathrm{MXA}}^{acqcas}(k_{imp})$ – the consistency point inside the spinlock acquisition function

$$cp_{\mathrm{MXA}}^{acqcas}(k_{imp}) \overset{def}{\equiv} f_{top}(ac_{imp}) = acquire\_lock \wedge loc_{top}(ac_{imp}) = 3 \wedge \langle it \rangle \neq 1$$

such that

$$it \equiv bytes2bits(read_{lm}(\mathcal{M}_{\mathcal{E}\,top}(ac_{imp}), i, 0, 4))$$

is a bit-string value of the local variable i indicating whether the first iteration with the call of cas in the spinlock is being performed.

- $cp^{relvol}_{\mathrm{MXA}}(k_{imp})$ – the consistency point inside the spinlock release

$$cp^{relvol}_{\mathrm{MXA}}(k_{imp}) \overset{def}{\equiv} f_{top}(ac_{imp}) = release\_lock \wedge loc_{top}(ac_{imp}) = 1$$

Since in the compiler correctness for the MX machine we already required the cas calls and accesses to volatile to be $\mathcal{IO}$-points, we refer here explicitly the functions acquiring and releasing locks and the corresponding locations inside them (see Appendix A).

For a configuration $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$ of the MXA machine we also provide the shorthand

$$cp^{Th}_{\mathrm{MXA}}(c_{imp}, \pi, \theta_{imp}, cba) \overset{def}{\equiv} cp^{Th}_{\mathrm{MXA}}(c_{imp}.k, \pi, \theta_{imp}, cba)$$

8.3.2 Well-Formed Implementation

Before we consider the simulation relation between the sequential machine for threads and the MXA implementing them, we pay attention to the well-formedness of the MXA machine. In particular, since the stacks of threads must have specific layout allowing the thread switch operation, we will define it formally here. Obviously, such layout will be preserved by the implementation and under some software conditions.

First, we introduce a few auxiliary definitions for operating on the data structures in the MXA machine.

For retrieving the values of variables and elements/fields of the data structures from the framework for kernel threads, we will not use the C-IL expression evaluation. Instead, we will get their bit-string values directly from the memory of the MXA machine. The reason is that a corresponding C-IL configuration is not always available. Moreover, the correctness criteria and required conditions can be easier stated without considering the typed C-IL values. So, given the MXA machine memory $\mathcal{M}$ and a thread ID $t \in \mathbb{N}_{mtid}$, we define the shorthands:

- base address of the TCB

$$ba^{\theta_{imp}}_{TCB}(\mathcal{M}, t) \equiv ba^{\theta_{imp}}_{field}\left(ba^{\theta_{imp}}_{elem}(threads, Thread, t), Thread, tcb\right)$$

- bit-string value of a TCB field $f \in \mathbb{F}$

$$va^{\theta_{imp}}_{TCB}(\mathcal{M}, t, f) \equiv \mathcal{M}_4\left(ba^{\theta_{imp}}_{field}\left(ba^{\theta_{imp}}_{TCB}(\mathcal{M}, t), TCB, f\right)\right)$$

- thread ID, state, affinity, argument, and stack size $X \in \{tid, state, pid, arg, mss\}$

$$X^{\theta_{imp}}(\mathcal{M}, t) \equiv \left\langle va^{\theta_{imp}}_{TCB}(\mathcal{M}, t, X)\right\rangle$$

- the address of the thread entry function

$$ad^{\theta_{imp}}_{fn}(\mathcal{M}, t) \equiv va^{\theta_{imp}}_{TCB}(\mathcal{M}, t, fn)$$

- stack base address

$$sba^{\theta_{imp}}(\mathcal{M}, t) \equiv va^{\theta_{imp}}_{TCB}(\mathcal{M}, t, sba)$$

- base address of the thread context

$$ba^{\theta_{imp}}_{cntx}(\mathcal{M}, t) \equiv ba^{\theta_{imp}}_{field}\left(ba^{\theta_{imp}}_{TCB}(\mathcal{M}, t), TCB, cntx\right)$$

- stack pointer and frame base pointer

$$sp^{\theta_{imp}}(\mathcal{M}, t) \equiv \mathcal{M}_4\left(ba^{\theta_{imp}}_{field}\left(ba^{\theta_{imp}}_{cntx}(\mathcal{M}, t), \mathit{Thcntx}, sp\right)\right)$$

$$bp^{\theta_{imp}}(\mathcal{M}, t) \equiv \mathcal{M}_4\left(ba^{\theta_{imp}}_{field}\left(ba^{\theta_{imp}}_{cntx}(\mathcal{M}, t), \mathit{Thcntx}, bp\right)\right)$$

For a given processor ID $pid \in \mathbb{N}_{np}$ we denote the index of the current thread obtained from the memory $\mathcal{M}$ as

$$ct^{\theta_{imp}}(\mathcal{M}, pid) \equiv \left\langle \mathcal{M}_4\left(ba^{\theta_{imp}}_{elem}(current, \mathbf{u32}, pid)\right)\right\rangle$$

The identifier of the processor accessing the shared resources is computed by

$$ap^{\theta_{imp}}(\mathcal{M}) \equiv \langle \mathcal{M}_4\left(\theta_{imp}.alloc_{gvar}(lock)\right)\rangle$$

Along with computation of the fields from the TCB of a given thread, we also collect all thread identifiers for lists of ready, new and free threads declared in π_{th}.

Definition 8.23 (Sets of Thread Identifiers from Lists). Given an address $a \in \mathbb{B}^{32} \cup \{\bot\}$ of the node of the list, we define $\mathit{TIDs}^{\theta_{imp}}(a, \mathcal{M}) \in 2^{\mathbb{N}_{mtid}}$ inductively as

$$\mathit{TIDs}(a, \mathcal{M}) \overset{def}{\equiv} \begin{cases} \emptyset & : a = \bot \vee a = 0^{32} \\ \{tid\} \cup \mathit{TIDs}^{\theta_{imp}}\left(\mathit{StIba}_{next}(a, \mathcal{M}), \mathcal{M}\right) & : \text{otherwise} \end{cases}$$

where the thread ID is computed as above

$$tid \equiv \left\langle \mathcal{M}_4\left(ba^{\theta_{imp}}_{field}\left(ba_{TCB}, \mathit{TCB}, tid\right)\right)\right\rangle$$

$$ba_{TCB} \equiv ba^{\theta_{imp}}_{field}\left(a, \mathit{Thread}, tcb\right)$$

Hence, for $X \in \{new, free\}$ we easily compute

$$\mathit{TIDs}^{\theta_{imp}}_{X}(\mathcal{M}) \overset{def}{\equiv} \mathit{TIDs}^{\theta_{imp}}\left(\mathcal{M}_4\left(\theta_{imp}.alloc_{gvar}(X\_hd)\right), \mathcal{M}\right)$$

In turn, for $Y \in \{ready, fin\}$ and $pid \in \mathbb{N}_{np}$ we have

$$\mathit{TIDs}^{\theta_{imp}}_{Y}(\mathcal{M}, pid) \overset{def}{\equiv} \mathit{TIDs}^{\theta_{imp}}\left(\mathcal{M}_4\left(ba^{\theta_{imp}}_{elem}(Y\_hd, \mathbf{ptr}(thread\_t), pid)\right), \mathcal{M}\right)$$

The well-formedness of the stack and data structures implementing a kernel thread depends on the state of this thread.

Sleeping Thread Recall that a thread switch is performed when a current kernel thread represented by an MXA thread in the MXA machine executes the function *thread_run* which, in turn, calls the MASM procedure *switch_stack*.

According to Listing 8.1 the MX stack abstraction of the current thread is lost at the beginning of the inline assembly (line 1) because of switching to the consistent MIPS-86 configuration. If the physical stack residing in the memory is not modified later, we can reconstruct almost the same MX thread configuration when this thread is scheduled again. The only components that can differ are the location counter in the topmost frame and the values of GPRs and SPRs because they are not saved on the stack or in TCB. The reconstruction is performed according to the MXA semantics before returning from $switch\_stack$ (line 2).

In order to show the correctness of the thread switch we will require that this reconstruction on the base of the stack residing in the memory and the TCB fields is always possible for any sleeping thread.

First, we consider which reconstructed MX thread configurations we are interested in.

Definition 8.24 (Reconstructed MX Thread Configuration Well-Formed for Sleeping Thread). For $k_{\text{mx}} \in \mathbb{K}_{\text{MX}}$ we require that (i) the active context is represented by a well-formed MASM frame for $switch\_stack$ with the location at the return from the procedure, and (ii) the list of inactive contexts has a C-IL context on its top where the stack contains at least two frames. (iii) Its well-formed topmost frame belongs to the function $thread\_run$ with the location after the call of $switch\_stack$. Using $ic_{top} \equiv k_{\text{mx}}.ic[|k_{\text{mx}}.ic|]$ we define

$$
\begin{aligned}
wf^{sleep}_{\mathbb{K}_{\text{MX}}}(k_{\text{mx}}, \theta_{imp}) &\overset{def}{\equiv} \\
\textit{(i)} \quad & masm(k_{\text{mx}}.ac) \wedge k_{\text{mx}}.ac.stack = fr \\
\textit{(ii)} \quad & cil(ic_{top}) \wedge |ic_{top}| \geq 2 \\
\textit{(iii)} \quad & ic_{top}[|ic_{top}|] = fr'
\end{aligned}
$$

such that the frame of the active context satisfies the conditions

$$
fr.p = switch\_stack \qquad fr.loc = 2 \qquad |fr.pars| = 4 \qquad fr.lifo = \varepsilon
$$
$$
\text{dom}\,(fr.saved) = \#Reg_{callee}
$$

and the frame fr' has the properties

$$
fr'.f = thread\_run \qquad fr'.rds = \bot \qquad fr'.loc = 10
$$
$$
\text{dom}\left(fr'.\mathcal{M}_{\mathcal{E}}\right) = decl\left(\pi^{th}_{\text{cil}}.\mathcal{F}(thread\_run).V\right)
$$
$$
\forall\, (v,t) \in \pi^{th}_{\text{cil}}.\ \mathcal{F}(thread\_run).V.\ |fr'.\mathcal{M}_{\mathcal{E}}(v)| = size_{\theta_{imp}}(qt2t(t))
$$

Note that the values of the arguments and local variables do not matter because they are not used during scheduling the sleeping thread. The same applies to the content of the general and special purpose registers in k_{mx}.

As the next step, we have to choose the MIPS ISA configuration for which we will state that the reconstruction of the MX thread conforming $wf^{sleep}_{\mathbb{K}_{\text{MX}}}$ is always possible. Obviously, as long as the thread remains sleeping, the MXA configuration is irrelevant except for the memory region with the TCB and the stack of this thread. Fortunately, according to the $wf^{sleep}_{\mathbb{K}_{\text{MX}}}$ we do not have to care about the irrelevant parts of the configuration and can easily compose the ISA configuration needed for the reconstruction.

Definition 8.25 (ISA Configuration for the MX Thread Reconstruction of the Sleeping Thread). Given an identifier $t \in \mathbb{N}_{mtid}$ of a sleeping thread and the memory configuration $\mathcal{M}$ of the MXA machine we define

$$
conf^{sleep}_{\text{MIPS}}(t, \mathcal{M}, \pi_{imp}, \theta_{imp}, cba) \overset{def}{\equiv} (cpu, \mathcal{M})
$$

such that $cpu \in \mathbb{C}_{proc}$ satisfies the following conditions:

$$
cpu.core.pc = ca_{\text{MASM}}\left(switch\_stack, 2, info^{imp}_{\mu}, cba_{\mu}\right)
$$

$$cpu.core.gpr(sp) = sp^{\theta_{imp}}(\mathcal{M}, t)$$
$$cpu.core.gpr(bp) = bp^{\theta_{imp}}(\mathcal{M}, t)$$

All other components of the processor are either not involved into the reconstruction or irrelevant and, therefore, may have arbitrary configurations.

Definition 8.26 (Well-Formed TCB and Stack for Sleeping Thread). Finally, we can define that the TCB and the stack residing in the the memory $\mathcal{M}$ of the MXA machine implementing the kernel threads are well-formed for a sleeping thread $t \in \mathbb{N}_{mtid}$ with $state^{\theta_{imp}}(\mathcal{M}, t) =$ TS_SLEEPING.

$$\begin{aligned} wf^{sleep}_{\text{MXA}}(t, \mathcal{M}, \pi_{imp}, \theta_{imp}, cba) \overset{def}{\equiv}& \\ \exists k_{\text{mx}} \in \mathbb{K}_{\text{MX}}.& \\ &(i) \quad k_{\text{mx}} = R^{\pi_{imp},\theta_{imp}}_{\mathbb{K}_{\text{MX}}}\left(d, info_{imp}, cba, sba^{\theta_{imp}}(\mathcal{M}, t)\right) \\ &(ii) \quad wf^{sleep}_{\mathbb{K}_{\text{MX}}}(k_{\text{mx}}, \theta_{imp}) \end{aligned}$$

with $d \equiv conf^{sleep}_{\text{MIPS}}(t, \mathcal{M}, \pi_{imp}, \theta_{imp}, cba)$.

New Thread The well-formedness of the TCB and the physical stack of any new thread reflects the physical stack layout on Figure 8.3.

Definition 8.27 (Well-Formed TCB and Stack for New Thread). For a new thread with an identifier $t \in \mathbb{N}_{mtid}$ such that $state^{\theta_{imp}}(\mathcal{M}, t) =$ TS_NEW holds we require that (i) the argument passed to its entry function is stored on the bottom of the physical stack, (ii) the return address in the topmost frame is the address of the entry function, and (iii) – (iv) the stack base pointer and the frame base pointer in the thread context are set properly.

$$\begin{aligned} wf^{new}_{\text{MXA}}(t, \mathcal{M}, \theta_{imp}) \overset{def}{\equiv}& \\ &(i) \quad arg^{\theta_{imp}}(\mathcal{M}, t) = \langle \mathcal{M}_4(ad_{bot})\rangle \\ &(ii) \quad ad^{\theta_{imp}}_{fn}(\mathcal{M}, t) = ra\left(\mathcal{M}, bp^{\theta_{imp}}(\mathcal{M}, t)\right) \\ &(iii) \quad bp^{\theta_{imp}}(\mathcal{M}, t) = ad_{bot} -_{32} (4 \cdot 7)_{32} \\ &(iv) \quad sp^{\theta_{imp}}(\mathcal{M}, t) = bp^{\theta_{imp}}(\mathcal{M}, t) -_{32} (4 \cdot \#Reg_{callee})_{32} \end{aligned}$$

where $ad_{bot} \equiv sba^{\theta_{imp}}(\mathcal{M}, t) -_{32} 3_{32}$ is the address of the word on the bottom of the stack (or the first item).

Running Thread In this case we define the well-formedness on the whole configuration of the sequential MXA machine that must hold at consistency points from Definition 8.22.

Let $pid(c_{imp})$ be the processor identifier in a configuration $c_{imp} \in \mathbb{C}_{\text{MXA}}$ retrieved as

$$pid(c_{imp}) \equiv \begin{cases} \langle c_{imp}.k.k_{\text{mx}}.spr(pid)\rangle & : mx(c_{imp}) \\ \langle c_{imp}.k.core.spr(pid)\rangle & : isa(c_{imp}) \end{cases}$$

Definition 8.28 (Well-Formed MXA Configuration for Running Thread). For a running (or current) thread with an identifier $t \in \mathbb{N}_{mtid}$ such that $state^{\theta_{imp}}(\mathcal{M}, t) =$ TS_RUNNING holds and a configuration c_{imp} of the MXA machine encoding this thread we require that (i) the MXA configuration is well-formed, (ii.a) before the lock acquisition in *acquire_lock* and at the C-IL calls of primitives implemented with the lock protection the lock is not acquired by the considered processor yet, and (ii.b) at the moment of the lock release it belongs to this processor.

$$
\begin{array}{lll}
\multicolumn{3}{l}{wf^{run}_{\mathrm{MXA}}(t, c_{imp}, \pi, \pi_{imp}, \theta_{imp}, cba) \overset{def}{\equiv}} \\
(i) & \multicolumn{2}{l}{wfconf^{\pi_{imp},\theta_{imp},cba}_{\mathrm{MXA}}(c_{imp})} \\
(ii) & \multicolumn{2}{l}{mx(c_{imp}.k) \wedge cil(ac_{imp}) \implies} \\
 & (a) & cp^{acqcas}_{\mathrm{MXA}}(c_{imp}.k) \vee info_{\mathrm{cil}}.cp\,(f_{top}(ac_{imp}), loc_{top}(ac_{imp})) \implies \\
 & & ap^{\theta_{imp}}(c_{imp}.\mathcal{M}) \neq pid(c_{imp}) \\
 & (b) & cp^{relvol}_{\mathrm{MXA}}(c_{imp}.k) \implies ap^{\theta_{imp}}(c_{imp}.\mathcal{M}) = pid(c_{imp})
\end{array}
$$

with $ac_{imp} \equiv c_{imp}.k.k_{\mathrm{mx}}.ac$.

Note that in (ii.a) for brevity we consider all consistency points in the C-IL program of the hypervisor / OS kernel though we are only interested in those at which the mentioned functions are called.

MXA Machine Implementing Kernel Threads Finally, in order to define the well-formedness of an MXA machine configuration for kernel thread implementation, we consider it separately for the local lists belonging to the processor and the global ones as declared in π_{th}.

Definition 8.29 (Well-Formed MXA Configuration for All Processor's Threads). Then, the well-formedness of $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$ for all processor's threads requires that (i) – (ii) the current thread has the proper state in its TCB and c_{imp} is well-formed for it, (iii) all other threads from the list of ready ones have the states of sleeping or new threads and their TCBs as well as physical stacks in the memory are well-formed, (iv) the list of finished threads contains only finished threads.

Using $ct \equiv ct^{\theta_{imp}}(\mathcal{M}, pid(c_{imp}))$ we formally define

$$
\begin{array}{llll}
\multicolumn{4}{l}{wf^{proc}_{\mathrm{MXA}}(c_{imp}, \pi_{imp}, \theta_{imp}, cba) \overset{def}{\equiv}} \\
(i) & \multicolumn{3}{l}{state^{\theta_{imp}}(c_{imp}.\mathcal{M}, ct) = \mathtt{TS\_RUNNING}} \\
(ii) & \multicolumn{3}{l}{wf^{run}_{\mathrm{MXA}}(ct, c_{imp}, \pi_{imp}, \theta_{imp}, cba)} \\
(iii) & \multicolumn{3}{l}{\forall t \in TIDs^{\theta_{imp}}_{ready}(c_{imp}.\mathcal{M}, pid(c_{imp})) \setminus \{ct\}.} \\
 & (a) & \multicolumn{2}{l}{state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) \in \{\mathtt{TS\_SLEEPING,\ TS\_NEW}\}} \\
 & (b) & \multicolumn{2}{l}{state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \mathtt{TS\_SLEEPING} \implies} \\
 & & & wf^{sleep}_{\mathrm{MXA}}(t, c_{imp}.\mathcal{M}, \pi_{imp}, \theta_{imp}, cba) \\
 & (c) & \multicolumn{2}{l}{state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \mathtt{TS\_NEW} \implies} \\
 & & & wf^{new}_{\mathrm{MXA}}(t, c_{imp}.\mathcal{M}, \theta_{imp}) \\
(iv) & \multicolumn{3}{l}{\forall t \in TIDs^{\theta_{imp}}_{fin}(c_{imp}.\mathcal{M}, pid(c_{imp})).} \\
 & \multicolumn{3}{l}{\quad state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \mathtt{TS\_FINISHED}}
\end{array}
$$

Since the access to the global lists of new and free threads is lock protected and is only possible when the lists are unlocked or locked by the considered processor, we define the well-formedness for them only under these conditions.

Definition 8.30 (Well-Formed MXA Configuration for New and Free Threads). The well-formedness of a configuration c_{imp} in this case assumes that all threads in both global lists have proper states in their TCBs, and the TCBc and stacks for new threads are well-formed.

$$\begin{aligned}
& wf^{free/new}_{\mathrm{MXA}}(c_{imp}, \theta_{imp}) \overset{def}{\equiv} \\
& \quad ap^{\theta_{imp}}(c_{imp}.\mathcal{M}) \in \{0, pid(c_{imp})\} \implies \\
& \qquad \textit{(i)} \quad \forall t \in TIDs^{\theta_{imp}}_{new}(c_{imp}.\mathcal{M}). \\
& \qquad\qquad state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \texttt{TS\_NEW} \wedge wf^{new}_{\mathrm{MXA}}(t, c_{imp}.\mathcal{M}, \theta_{imp}) \\
& \qquad \textit{(ii)} \quad \forall t \in TIDs^{\theta_{imp}}_{free}(c_{imp}.\mathcal{M}). \\
& \qquad\qquad state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \texttt{TS\_FREE}
\end{aligned}$$

Definition 8.31 (Well-Formed MXA Configuration for Kernel Threads Implementation). Finally, we combine Definitions 8.29 and 8.30 into the well-formedness of MXA machine configurations for kernel thread implementation as

$$\begin{aligned}
& wfconf^{Th}_{\mathrm{MXA}}(c_{imp}, \pi_{imp}, \theta_{imp}, cba) \overset{def}{\equiv} \\
& \quad \textit{(i)} \quad wf^{proc}_{\mathrm{MXA}}(c_{imp}, \pi_{imp}, \theta_{imp}, cba) \\
& \quad \textit{(ii)} \quad wf^{free/new}_{\mathrm{MXA}}(c_{imp}, \theta_{imp}) \\
& \quad \textit{(iii)} \quad valid^{cp}_{\mathrm{MX}}(\pi_{imp}, info_{imp}, \theta_{imp})
\end{aligned}$$

The last condition is in fact needed for proving the correctness of the switch to a new thread where the simulation relation must hold at the beginning of the body of the thread entry function.

8.3.3 Sequential Simulation Relation

First, we separately define the consistency relation for each kind of kernel threads.

Definition 8.32 (Consistency Relation for Sleeping Thread). Given a configuration $th \in \mathbb{K}^{nrun}_{Th}$ of a kernel thread with an identifier $t \in \mathbb{N}_{mtid}$ and the memory $\mathcal{M}$ of the MXA machine implementing the kernel threads such that the thread is sleeping, i.e., $state^{\theta_{imp}}(\mathcal{M}, t) = \texttt{TS\_SLEEPING}$ holds, we introduce the simulation relation $consis^{sleep}_{Th}(th, t, \mathcal{M}, \pi, \theta_{imp}, cba)$ stating that the active and inactive contexts in th are equal to the corresponding parts of an existing MX thread configuration, and other components of th are coupled with the fields in the TCB.

Formally, for $k_{\mathrm{mx}} \in \mathbb{K}_{\mathrm{MX}}$ existing by Definition 8.26 and computed as

$$k_{\mathrm{mx}} \equiv R^{\pi_{imp}, \theta_{imp}}_{\mathbb{K}_{\mathrm{MX}}}\left(d, info_{imp}, cba, sba^{\theta_{imp}}(\mathcal{M}, t)\right)$$

$$d \equiv conf^{sleep}_{\mathrm{MIPS}}(t, \mathcal{M}, \pi_{imp}, \theta_{imp}, cba)$$

and the shorthands

$$ni \equiv |k_{\mathrm{mx}}.ic| \qquad ic_{top} \equiv k_{\mathrm{mx}}.ic[ni]$$

we define

$$\begin{aligned}
& consis^{sleep}_{Th}(th, t, \mathcal{M}, \pi, \theta_{imp}, cba) \overset{def}{\equiv} \\
& \quad \textit{(i)} \quad th.ic = k_{\mathrm{mx}}.ic[1 : ni - 1] \\
& \quad \textit{(ii)} \quad th.ac = ic_{top}[1 : top(ic_{top}) - 1] \\
& \quad \textit{(iii)} \quad t = tid^{\theta_{imp}}(\mathcal{M}, t) \\
& \quad \textit{(iv)} \quad th.pid = pid^{\theta_{imp}}(\mathcal{M}, t) \\
& \quad \textit{(v)} \quad th.sba = sba^{\theta_{imp}}(\mathcal{M}, t) \\
& \quad \textit{(vi)} \quad th.mss = mss^{\theta_{imp}}(\mathcal{M}, t)
\end{aligned}$$

In contrast to the sleeping threads that can run again on the processor, the finished threads are never scheduled. Therefore, we are not interested in their stack abstraction and can skip the coupling relation for it.

Definition 8.33 (Consistency Relation for Finished Thread). Given a configuration $th \in \mathbb{K}_{Th}^{nrun}$ of a finished thread $t \in \mathbb{N}_{mtid}$ and the MXA machine memory $\mathcal{M}$ such that $state^{\theta_{imp}}(\mathcal{M}, t) =$ `TS_FINISHED` holds, we define the simulation relation

$$
\begin{aligned}
consis_{Th}^{fin}(th, t, \mathcal{M}, \theta_{imp}) \overset{def}{\equiv}& \\
\text{(i)} \quad & t = tid^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(ii)} \quad & th.pid = pid^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(iii)} \quad & th.sba = sba^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(iv)} \quad & th.mss = mss^{\theta_{imp}}(\mathcal{M}, t)
\end{aligned}
$$

Definition 8.34 (Consistency Relation for New Thread). Given a new thread $th \in \mathbb{K}_{Th}^{nrun}$ with an identifier $t \in \mathbb{N}_{mtid}$ and the memory $\mathcal{M}$ such that $state^{\theta_{imp}}(\mathcal{M}, t) =$ `TS_NEW` holds, the simulation relation $consis_{Th}^{new}(th, t, \mathcal{M}, \pi, \theta, \theta_{imp}, cba)$ specifies that the function name and the argument in the frame of the active context as well as all other components of th not belonging to the stack abstraction are coupled with the corresponding fields in the TCB of the given thread. Moreover, we include the well-formedness of the new thread from Definition 8.11. Though this well-formedness is required for all threads in new in Definition 8.12, it is formally lost after the thread is moved to $ready$ (see Definition 8.10) and we need to keep it here for the correctness proof.

Using the shorthands $st \equiv th.ac$ and $(v, t) \equiv \pi_{imp}.\mathcal{F}(f_{top}(st)).V[1]$ we formally define

$$
\begin{aligned}
consis_{Th}^{new}(th, t, \mathcal{M}, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv}& \\
\text{(i)} \quad & f_{top}(st) = \epsilon\left\{f' \in \mathbb{F}_{name} \;\middle|\; \theta_{imp}.\mathcal{F}_{adr}(f') = ad_{fn}^{\theta_{imp}}(\mathcal{M}, t)\right\} \\
\text{(ii)} \quad & \left\langle bytes2bits\left(\mathcal{M}_{\mathcal{E}\,top}(st)(v)\right)\right\rangle = arg^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(iii)} \quad & t = tid^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(iv)} \quad & th.pid = pid^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(v)} \quad & th.sba = sba^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(vi)} \quad & th.mss = mss^{\theta_{imp}}(\mathcal{M}, t) \\
\text{(vii)} \quad & wfth_{new}^{\pi,\theta}(th)
\end{aligned}
$$

The simulation relation for the current thread depends on whether it performs MXA machine steps or executes a primitive. Since the semantics of the primitives implemented with locks is split into phases, we must support the simulation relation at consistency points inside their function bodies. Recall that during any primitive execution we do not create a frame in the configuration of the current thread. In the implementation, however, we get two additional frames for the special function call and the lock acquire/release operation. If such a primitive has parameters and the implementation MXA machine is at a consistency point inside the lock acquisition function, one should couple the local parameters in the frame of the primitive with the values of the arguments passed during the call by the current thread. Moreover, the location in the caller's frame in the implementation is one position ahead in comparison with the configuration of the thread. All these facts are taken into account in the following definition.

Definition 8.35 (Consistency Relation for Running Thread). Given a configuration $c \in \mathbb{C}_{Th}$ of the sequential machine for kernel threads and a configuration $c_{imp} \in \mathbb{C}_{\text{MXA}}$ of the implementation machine, we define the coupling relation $consis^{run}_{Th}(c, c_{imp}, \pi, \theta_{imp}, cba)$ specifying how the MXA machine encodes the current kernel thread.

Namely, we require that (i) the processor configurations are fully coupled during inline assembly steps in π or user/guest transitions, (ii) if no primitive is called during MX machine steps, the configuration of the running kernel thread is equal to the configuration of the MXA thread, (iii) the same holds before the call of the thread switch or exit. (iv) In case of any other primitive we distinguish whether the call is already performed in the implementation or not and couple the configurations wrt. the discussion above. (v) Especially, as mentioned before, if $thread\_create$ or $thread\_delete$ is already called in the MXA machine, we couple their parameters on the stack with the arguments in the call of the primitive. (vi) – (viii) Obviously, the thread identifier, stack base pointer, and frame base pointer in the thread configuration should be equal to the fields in its TCB.

Let the stacks of active context in c and c_{imp} be denoted by

$$st \equiv th_{cur}(c).k_{\text{mx}}.ac \qquad st_{imp} \equiv c_{imp}.k.k_{\text{mx}}.ac$$

and the difference between the lengths of these stacks be computed as

$$dif \equiv top(st_{imp}) - top(st)$$

Moreover, let c'_{imp} be a MXA machine configuration obtained from c_{imp} by dropping dif topmost frames from the stack of the active context as

$$c'_{imp}.k.k_{\text{mx}}.ac = st_{imp}[1 : top(st)]$$

Then, using additional shorthands $st'_{imp} \equiv st_{imp}[1 : top(st_{imp}) - 1]$, $(v_i, t_i) \equiv \mathcal{F}^{\theta}_{\pi_{\text{cil}}}(f).V[i]$, and $stmt_{\text{cil}}$, c_{cil} from Definition 8.13, we formally define the simulation relation as

$$
\begin{aligned}
consis^{run}_{Th}(c, c_{imp}, \pi, \theta_{imp}, cba) &\overset{def}{\equiv} \\
\textit{(i)} \quad & th_{cur}(c) \in \mathbb{K}^{isa}_{Th} \implies th_{cur}(c).cpu = c_{imp}.k \\
\textit{(ii)} \quad & th_{cur}(c) \in \mathbb{K}^{mx}_{Th} \wedge \neg prim^{\pi,\theta}_{Th}(c) \implies th_{cur}(c) = c_{imp}.k \\
\textit{(iii)} \quad & prim^{\pi,\theta}_{Th}(c) \wedge stmt_{\text{cil}} = \textbf{call } f(E) \wedge \\
& f \in \{thread\_run, thread\_exit\_to\} \implies th_{cur}(c) = c_{imp}.k \\
\textit{(iv)} \quad & prim^{\pi,\theta}_{Th}(c) \wedge stmt_{\text{cil}} = \textbf{call } f'(E) \wedge \\
& f' \in \mathbb{F}^{prim}_{name} \setminus \{thread\_run, thread\_exit\_to\} \implies \\
& (dif = 0 \wedge th_{cur}(c) = c_{imp}.k) \vee \\
& \big(dif = 2 \wedge f_{top}(st'_{imp}) = f' \wedge th_{cur}\,(inc^{cur}_{loc}(c)) = c'_{imp}.k\big) \\
\textit{(v)} \quad & prim^{\pi,\theta}_{Th}(c) \wedge stmt_{\text{cil}} = \textbf{call } f(E) \wedge \\
& f \in \{thread\_create, thread\_delete\} \wedge cp^{acqcas}_{\text{MXA}}(c_{imp}.k) \implies \\
& \forall i \in \mathbb{N}_{|E|}.\ val2bytes\left([\![E[i]]\!]^{\pi_{\text{cil}},\theta}_{c_{\text{cil}}}\right) = \mathcal{M}_{\mathcal{E}\,top}(st'_{imp})(v_i) \\
\textit{(vi)} \quad & c.k.ct = tid^{\theta_{imp}}(\mathcal{M}, c.k.ct) \\
\textit{(vii)} \quad & th_{cur}(c).sba = sba^{\theta_{imp}}(\mathcal{M}, c.k.ct) \\
\textit{(viii)} \quad & th_{cur}(c).mss = mss^{\theta_{imp}}(\mathcal{M}, c.k.ct)
\end{aligned}
$$

Finally, we can define the consistency relation for all threads belonging to the processor.

Definition 8.36 (Consistency Relation for Processor's Threads). Given configurations $c \in \mathbb{C}_{Th}$ and $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$ of the sequential machine for kernel threads and the MXA machine implementing it as well as π, θ, θ_{imp}, and cba considered above, we define the simulation relation for the processor's threads $c.k$ claiming that (i) the current thread identifier resides in the corresponding element in the array $current$ of the framework π_{th}, (ii) the sets of identifiers of all threads ready for scheduling or scheduled are equal in the specification and the implementation, (iii) – (iv) the corresponding consistency relations for the current, new, and sleeping threads belonging to the processor hold, and (v) – (vi) the same finished threads covered by the consistency relation are considered in both machines.

Using $p \equiv pid_{cur}(c)$ we define

$$
\begin{aligned}
&consis^{proc}_{Th}(c, c_{imp}, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv}\\
&\quad \textit{(i)}\quad c.k.ct = ct^{\theta_{imp}}(c_{imp}.\mathcal{M}, p)\\
&\quad \textit{(ii)}\quad \mathrm{dom}\,(c.k.ready) = TIDs^{\theta_{imp}}_{ready}(c_{imp}.\mathcal{M}, p)\\
&\quad \textit{(iii)}\quad consis^{run}_{Th}(c, c_{imp}, \pi, \theta_{imp}, cba)\\
&\quad \textit{(iv)}\quad \forall t \in \mathrm{dom}\,(c.k.ready) \setminus \{c.k.ct\}.\\
&\qquad\qquad \textit{(a)}\quad state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \texttt{TS\_SLEEPING} \implies\\
&\qquad\qquad\qquad consis^{sleep}_{Th}(c.k.ready(t), t, c_{imp}.\mathcal{M}, \pi, \theta_{imp}, cba)\\
&\qquad\qquad \textit{(b)}\quad state^{\theta_{imp}}(c_{imp}.\mathcal{M}, t) = \texttt{TS\_NEW} \implies\\
&\qquad\qquad\qquad consis^{new}_{Th}(c.k.ready(t), t, c_{imp}.\mathcal{M}, \pi, \theta, \theta_{imp}, cba)\\
&\quad \textit{(v)}\quad \mathrm{dom}\,(c.k.fin) = TIDs^{\theta_{imp}}_{fin}(c_{imp}.\mathcal{M}, p)\\
&\quad \textit{(vi)}\quad \forall t \in \mathrm{dom}\,(c.k.fin)\,.\\
&\qquad\qquad consis^{fin}_{Th}(c.k.fin(t), t, c_{imp}.\mathcal{M}, \theta_{imp})
\end{aligned}
$$

Note that $state^{\theta_{imp}}(c_{imp}.\mathcal{M}, c.k.ct) = \texttt{TS\_RUNNING}$ is a part of the well-formedness in Definition 8.29 and we do not repeat it here.

As for the components new and $free$ of the machine configuration $c \in \mathbb{C}_{Th}$, we can couple them with the implementation only when we can access them according to the semantics of the machine for kernel threads.

Definition 8.37 (Consistency Relation for New and Free Threads). If the condition $c.ap = 0 \vee c.ap = pid_{cur}(c)$ holds, we can require that (i) the free identifiers of threads are equal in c and c_{imp}, and (ii) – (iii) the same newly created threads satisfying the simulation relation from Definition 8.34 are present in both machines.

$$
\begin{aligned}
&consis^{free/new}_{Th}(c, c_{imp}, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv}\\
&\quad c.ap = 0 \vee c.ap = pid_{cur}(c) \implies\\
&\qquad \textit{(i)}\quad c.free = TIDs^{\theta_{imp}}_{free}(c_{imp}.\mathcal{M})\\
&\qquad \textit{(ii)}\quad \mathrm{dom}\,(c.new) = TIDs^{\theta_{imp}}_{new}(c_{imp}.\mathcal{M})\\
&\qquad \textit{(iii)}\quad \forall t \in \mathrm{dom}\,(c.new)\,.\\
&\qquad\qquad consis^{new}_{Th}(c.new(t), t, c_{imp}.\mathcal{M}, \pi, \theta, \theta_{imp}, cba)
\end{aligned}
$$

In the definition of the memory consistency we have to exclude the memory regions occupied by stacks having the counterparts in the configuration of the machine for kernel threads.

Definition 8.38 (Memory Addresses Occupied by Stacks of Processor's Threads). Given a configuration $k \in \mathbb{K}_{Th}^{proc}$ we define the set $A_{proc}^{stacks}(k)$ of memory byte addresses occupied by stacks for which the corresponding MX stack abstraction is present in k.

Let $D(k)$ be the set of identifies for threads with MX stack abstraction

$$D_{Th}(k) \overset{def}{\equiv} \operatorname{dom}(k.fin) \cup \operatorname{dom}(k.ready) \setminus \begin{cases} \{k.ct\} & : \ th_{cur}(k) \in \mathbb{K}_{Th}^{\mathrm{mx}} \\ \emptyset & : \ th_{cur}(k) \in \mathbb{K}_{Th}^{\mathrm{isa}} \end{cases}$$

Then, using the shorthand

$$th(t) \equiv (\operatorname{dom}(k.fin) \uplus \operatorname{dom}(k.ready))(t)$$

we formally define

$$A_{proc}^{stacks}(k) \overset{def}{\equiv} \bigcup_{t \in D_{Th}(k)} A_{\mathrm{MX}}^{stack}(th(t).sba, th(t).mss)$$

Analogously, we define the set of addresses for the stacks of the new threads.

Definition 8.39 (Memory Addresses Occupied by Stacks of New Threads). Given a configuration $c \in \mathbb{C}_{Th}$, we compute

$$A_{new}^{stacks}(c) \overset{def}{\equiv} \bigcup_{t \in \operatorname{dom}(c.new)} A_{\mathrm{MX}}^{stack}(c.new(t).sba, c.new(t).mss)$$

Now, using Definitions 8.18, 8.38, and 8.39 we define the memory consistency for the sequential machine for kernel threads in the concurrent setting. Recall that according to the semantics of kernel threads (see MXA machine steps in Section 8.1.2.2 and the well-formedness in Definitions 7.5, 8.12) the compiled code of the hypervisor / OS kernel program π is visible in the memory of the machine for kernel threads. In turn, the MXA machine implementing the kernel threads contains in the memory the compiled code of the whole linked program π_{imp}. Therefore, the memory region where the code of π resides should be coupled by the simulation relation.

Definition 8.40 (Memory Consistency for Sequential Machine for Kernel Threads). For configurations $c \in \mathbb{C}_{Th}$, $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$, other parameters as above, and memory addresses $icm \subset A_{hyp} \setminus A_{\mathrm{MX}}^{code}(info, cba)$, the memory consistency relation $consis_{Th}^{mem}(c, c_{imp}, \pi, \theta_{imp}, cba, icm)$ requires that the content of the memory in both machines is equal except the regions occupied by the compiled code of the framework π_{th}, all global variables declared in π_{th}, the stacks having consistent counterparts in c, and the region icm

Computing the memory addresses occupied by the compiled code of the framework π_{th}

$$A^{code} \equiv A_{\mathrm{MX}}^{code}(info_{imp}, cba) \setminus A_{\mathrm{MX}}^{code}(info, cba)$$

and the stacks in c

$$A^{stacks} \equiv A_{proc}^{stacks}(c.k) \cup A_{new}^{stacks}(c)$$

we formally define

$$\begin{aligned} &consis_{Th}^{mem}(c, c_{imp}, \pi, \theta_{imp}, cba, icm) \overset{def}{\equiv} \\ &\qquad \forall a \in \mathbb{B}^{32} \setminus \left(A^{code} \cup A_{th}^{data}(\pi_{th}, \theta_{imp}) \cup A^{stacks} \cup icm\right). \ c.\mathcal{M}(a) = c_{imp}.\mathcal{M}(a) \end{aligned}$$

Finally, we can define the simulation relation for the full machine for kernel threads.

Definition 8.41 (Consistency Relation for Sequential Machine for Kernel Threads). Given configurations $c \in \mathbb{C}_{Th}$, $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$ of the sequential machine for kernel threads and the implementation MXA machine, a program π of the hypervisor / OS kernel, environment parameters θ, θ_{imp}, code base addresses $cba = (cba_{\mathrm{cil}}, cba_{\mu})$, and a set $icm \subset A_{hyp} \setminus A^{code}_{\mathrm{MX}}(info, cba)$ of memory addresses that cannot be covered by the memory consistency due to environment steps, we define the simulation relation between these machines as

$$\begin{aligned}
consis_{Th}(c, c_{imp}, \pi, \theta, \theta_{imp}, cba, icm) \ &\overset{def}{\equiv} \\
(i) \quad & consis^{proc}_{Th}(c, c_{imp}, \pi, \theta, \theta_{imp}, cba) \\
(ii) \quad & c.ap = ap^{\theta_{imp}}(c_{imp}.\mathcal{M}) \\
(iii) \quad & consis^{free/new}_{Th}(c, c_{imp}, \pi, \theta, \theta_{imp}, cba) \\
(iv) \quad & consis^{mem}_{Th}(c, c_{imp}, \pi, \theta_{imp}, cba, icm) \\
(v) \quad & valid^{par}_{Th}(\theta, \theta_{imp}) \wedge valid^{cmpl}_{Th}(\pi, \pi_{th}, \pi_{imp})
\end{aligned}$$

Note that (v) is included into the simulation relation because it is rather a property of the compiler than an assumption for the linker placing the compiled code into the memory.

8.3.4 Accessed Addresses

The set of addresses read and written during steps of the machine for kernel threads are computed similarly to Definition 7.39. The only difference is that we do not take into account the addresses for stack information retrieving (see Definition 7.38). In contrast to Definition 7.39 where in case of the end of ISA steps we search (in the memory) for the base address and the maximal size of the current stack, here, we only test whether the stack pointer and the frame base pointer belong to the stack of the current thread.

Definition 8.42 (Memory Addresses Accessed for Reading and Writing during Steps of Sequential Machine for Kernel Therads). For a configuration $c \in \mathbb{C}_{Th}$ and an input $in \in \Sigma_{\mathrm{MXA}}$ we use the following shorthands

$$c_{\mathrm{mxa}} \equiv conf^{curT}_{\mathrm{MXA}}(c) \qquad c_{\mathrm{mx}} \equiv conf^{\mathrm{MXA}}_{\mathrm{MX}}(c_{\mathrm{mxa}})$$

$$d \equiv conf^{\mathrm{MXA}}_{r\mathrm{MIPS}}(c_{\mathrm{mxa}}) \qquad d' \equiv \delta_{r\mathrm{MIPS}}(d, in)$$

$$\widehat{d} \equiv conf^{start}_{r\mathrm{MIPS}}(c_{\mathrm{mxa}}, in)$$

$$\widehat{in} \equiv (\mathtt{core}, \epsilon\, \mathbb{C}_{walk}, \epsilon\, \mathbb{C}_{walk}, 0^{256}) \qquad \widehat{d}' \equiv \delta_{r\mathrm{MIPS}}(\widehat{d}, \widehat{in})$$

and define the set of read addresses $reads^{\pi,\theta}_{Th}(c, in, cba)$ obtained as (i) the MX $reads$-set for a mixed machine step, and (ii) in case of the switch to inline assembly or a pure ISA step, computed as (a) the $reads$-set of the MIPS-86 model if there is no end of ISA steps or the end of ISA steps is reached and the stack pointer and the frame base pointer do not match the stack of the current thread, and (b) if they do, $reads^{\pi,\theta}_{Th}(c, in, cba)$ additionally includes the memory addresses occupied by this stack:

$$reads_{Th}^{\pi,\theta}(c, in, cba) \overset{def}{\equiv}$$
$$\begin{cases} reads_{\mathrm{MX}}\,(c_{\mathrm{mx}}, \pi, \theta) & : mx(c_{\mathrm{mxa}}.k) \wedge \neg start_{isa}^{\pi}(c_{\mathrm{mxa}}) \\ reads_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) & : start_{isa}^{\pi}(c_{\mathrm{mxa}}) \wedge \big(\neg end_{isa}^{\pi,\theta,cba}(\widehat{d}'.cpu) \vee \\ & \quad end_{isa}^{\pi,\theta,cba}(\widehat{d}'.cpu) \wedge \neg matchst_{cur}(c, \widehat{d}')\big) \\ reads_{\mathrm{MIPS}}\left(\widehat{d}, \widehat{in}\right) \cup & : start_{isa}^{\pi}(c_{\mathrm{mxa}}) \wedge end_{isa}^{\pi,\theta,cba}(\widehat{d}'.cpu) \wedge \\ A_{\mathrm{MX}}^{stack}\,(sba_{cur}(c), mss_{cur}(c)) & \quad matchst_{cur}(c, \widehat{d}') \\ reads_{\mathrm{MIPS}}\,(d, in) & : isa(c_{\mathrm{mxa}}.k) \wedge \big(\neg end_{isa}^{\pi,\theta,cba}(d'.cpu) \vee \\ & \quad end_{isa}^{\pi,\theta,cba}(d'.cpu) \wedge \neg matchst_{cur}(c, d')\big) \\ reads_{\mathrm{MIPS}}\,(d, in) \cup & : isa(c_{\mathrm{mxa}}.k) \wedge end_{isa}^{\pi,\theta,cba}(d'.cpu) \wedge \\ A_{\mathrm{MX}}^{stack}\,(sba_{cur}(c), mss_{cur}(c)) & \quad matchst_{cur}(c, d')\big) \end{cases}$$

In turn, the set of written addresses is the same as for the MXA machine:

$$writes_{Th}^{\pi,\theta}(c, in) \overset{def}{\equiv} writes_{\mathrm{MXA}}^{\pi,\theta}(c_{\mathrm{mxa}}, in)$$

Definition 8.43 (No Access to icm by the Machine for Kernel Threads). We combine both sets into

$$accad_{Th}^{\pi,\theta}(c, in, cba) \overset{def}{\equiv} reads_{Th}^{\pi,\theta}(c, in, cba) \cup writes_{Th}^{\pi,\theta}(c, in)$$

and define a predicate indicating that addresses icm are not accessed

$$noacc_{Th}^{\pi,\theta}(c, in, icm, cba) \overset{def}{\equiv} accad_{Th}^{\pi,\theta}(c, in, cba) \cap icm = \emptyset$$

8.3.5 $\mathcal{IO}$- and $\mathcal{OT}$-Points

The $\mathcal{IO}$- and $\mathcal{OT}$-points for our machine modeling kernel threads are similar to the ones for the MXA machine except steps of the primitives different from thread scheduling and exit. Since these primitives perform global operations, the machine executing such a primitive is considered to be at $\mathcal{IO}$- and $\mathcal{OT}$-points.

Let the call of the primitive distinct from the thread switch and exit (that are scheduling primitives) be indicated by the predicate

$$primnsch_{Th}^{\pi,\theta}(c) \overset{def}{\equiv}$$
$$\begin{aligned} &(i) \quad prim_{Th}^{\pi,\theta}(c) \\ &(ii) \quad stmt_{\mathrm{cil}} = \textbf{call}\ f(E) \wedge \\ &\qquad f \in \mathbb{F}_{name}^{prim} \setminus \{thread\_run, thread\_exit\_to\} \end{aligned}$$

Analogously, for the scheduling primitives we have

$$primsch_{Th}^{\pi,\theta}(c) \overset{def}{\equiv}$$
$$\begin{aligned} &(i) \quad prim_{Th}^{\pi,\theta}(c) \\ &(ii) \quad stmt_{\mathrm{cil}} = \textbf{call}\ f(E) \wedge \\ &\qquad f \in \{thread\_run, thread\_exit\_to\} \end{aligned}$$

Definition 8.44 ($\mathcal{IO}$- and $\mathcal{OT}$-Points for Kernel Threads). Then, for $c \in \mathbb{C}_{Th}$ and $in \in \Sigma_{\mathrm{MXA}}$ we define

$$\mathcal{IO}^{\pi,\theta}_{Th}(c, in, cba) \overset{def}{\equiv} \begin{cases} 1 & : \mathit{primnsch}^{\pi,\theta}_{Th}(c) \\ \mathcal{IO}^{\pi,\theta}_{\mathrm{MXA}}(c_{\mathrm{mxa}}, in, \mathit{info}, cba) & : \text{otherwise} \end{cases}$$

where info is computed as before.

$$\mathcal{OT}^{\pi,\theta}_{Th}(c, in) \overset{def}{\equiv} \begin{cases} 1 & : \mathit{primnsch}^{\pi,\theta}_{Th}(c) \\ \mathcal{OT}^{\pi,\theta}_{\mathrm{MXA}}(c, in) & : \text{otherwise} \end{cases}$$

For the implementation machine in configuration $c_{imp} \in \mathbb{C}_{\mathrm{MXA}}$ with an input $in \in \Sigma_{\mathrm{MXA}}$ we will directly use

$$\mathcal{IO}^{\pi_{imp},\theta_{imp}}_{\mathrm{MXA}}(c_{imp}, in, \mathit{info}_{imp}, cba) \quad \text{and} \quad \mathcal{OT}^{\pi_{imp},\theta_{imp}}_{\mathrm{MXA}}(c_{imp}, in)$$

8.3.6 Requirements and Conditions for MXA Machine

The requirements and conditions on steps of the MXA machine implementing the kernel threads are quite simple and are based on the definitions introduced before.

Since the MXA semantics already excludes non-suitable inputs for which the transitions are not defined, we simply set

$$\mathit{suit}^{Th}_{\mathrm{MXA}}(in) \overset{def}{\equiv} 1$$

As for the well-formedness of the MXA machine, we have ready defined it in Section 8.3.2 as

$$\mathit{wfconf}^{Th}_{\mathrm{MXA}}(c_{imp}, \pi_{imp}, \theta_{imp}, cba)$$

Finally, the well-behaviour of the MXA machine represents the software conditions from Section 7.3.6 needed for the simulation of the extended mixed machine. Therefore, we consider

$$\mathit{wb}^{Th}_{\mathrm{MXA}}(c_{imp}, in, \pi_{imp}, cba) \overset{def}{\equiv} \mathit{sc}_{\mathrm{MXA}}(c_{imp}, in, \pi_{imp}, \mathit{info}_{imp}, \theta_{imp}, cba, \mathit{StIba})$$

8.3.7 Software Conditions for Kernel Threads

For the simulation between the machine for kernel threads and the MXA machine implementing it, we still need to keep the static software conditions introduced in Definition 7.47 because they represent the requirements on the placement of the compiled code and data into the memory. Therefore, we restate here the same conditions with a slight modification.

Definition 8.45 (Static Software Conditions for Kernel Threads). For given π, θ_{imp}, and cba, we define

$$\mathit{sc}^{stat}_{Th}(\pi, \theta_{imp}, cba) \overset{def}{\equiv}$$

$$\begin{array}{ll} \text{(i)} & A^{code}_{\mathrm{MX}}(\mathit{info}_{imp}, cba) \subseteq A_{code} \\ \text{(ii)} & \mathit{valid}^{code}_{\mathrm{MX}}(\mathit{info}_{imp}, cba) \\ \text{(iii)} & A^{gvar}_{\mathrm{CIL}}(\pi^{imp}_{\mathrm{cil}}, \theta) \setminus A^{const}_{\mathrm{CIL}}(\pi^{imp}_{\mathrm{cil}}, \theta) \subset A_{data} \\ \text{(iv)} & A^{const}_{\mathrm{CIL}}(\pi^{imp}_{\mathrm{cil}}, \theta) = A_{const} \\ \text{(v)} & \mathit{sc}^{prog}_{\mathrm{MX}}(\pi, \theta) \end{array}$$

Note, that *(v)* is the requirement only for the program of the hypervisor / OS kernel. It will be used for proving the same property for the linked program in the presence of π_{th}.

In comparison to the dynamic software conditions for the MXA machine in Definition 7.48, the conditions for the kernel threads are not applied for the stack information regions that are abstracted away in the semantics. Recall that the stack base pointer and the stack maximal size are components of the kernel thread configuration. Moreover, we introduce restrictions on the primitive calls, forbid accesses to the memory containing the implementation of the primitives, TCBs, etc.

Definition 8.46 (**Dynamic Software Conditions for Kernel Threads).** Given a configuration $c \in \mathbb{C}_{Th}$ of the machine for kernel threads, an input $in \in \Sigma_{\text{MXA}}$, and the existing step $c' \equiv \delta_{Th}^{\pi,\theta,cba}(c, in)$, we state the dynamic software conditions requiring the following: (i) no run-time error is generated by the step, (ii) the current thread does not access the memory regions occupied by the compiled code of the framework, stacks of threads (except the case when the stack abstraction is lost) and by the global variables of the framework π_{th}, (iii) the current thread does not modify the compiled code of the hypervisor / OS kernel program π, (iv) during a step of a primitive other than thread switch and exit no volatile accesses in the evaluation of the primitive's arguments are allowed, (v) evaluation of arguments for thread switch and exit primitives may perform up to one volatile access, (vi) for any MX machine step (except the primitive call) the dynamic MX software conditions hold, (vii) for any MIPS-86 step the software conditions needed for the store buffer reduction hold, (viii) before a successful reconstruction of an MX thread configuration, the masked interrupts must be masked by the programmer again, and (ix) stacks of threads belong to the data region of the memory.

Let A_{imp} be the set of addresses at which the memory is not allowed to be accessed by the thread according to (ii)

$$\begin{aligned} A_{imp} \overset{def}{\equiv}\; & \left(A_{\text{MX}}^{code}(info_{imp}, cba) \setminus A_{\text{MX}}^{code}(info, cba)\right) \cup \\ & A_{proc}^{stacks}(c.k) \cup A_{new}^{stacks}(c) \cup A_{th}^{data}(\pi_{th}, \theta_{imp}) \end{aligned}$$

Then, using the shorthands d, $\widehat{d}$, $\widehat{in}$ from Definition 8.42, and $\overline{d}$, $\overline{in}$, $\overline{d}'$ given below, we define

$$\begin{aligned} & sc_{Th}^{dyn}(c, in, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv} \\ & \quad \textit{(i)} \quad c' \neq \bot \\ & \quad \textit{(ii)} \quad accad_{Th}^{\pi,\theta}(c, in, cba) \cap A_{imp} = \emptyset \\ & \quad \textit{(iii)} \quad writes_{Th}^{\pi,\theta}(c, in) \cap A_{\text{MX}}^{code}(info, cba) = \emptyset \\ & \quad \textit{(iv)} \quad primnsch_{Th}^{\pi,\theta}(c) \implies nIO_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) = 0 \\ & \quad \textit{(v)} \quad primsch_{Th}^{\pi,\theta}(c) \implies nIO_{\text{CIL}}^{\pi_{\text{cil}},\theta}(c_{\text{cil}}) \leq 1 \\ & \quad \textit{(vi)} \quad mx(c_{\text{mxa}}.k) \wedge \neg start_{isa}^{\pi}(c_{\text{mxa}}) \wedge \neg prim_{Th}^{\pi,\theta}(c) \implies \\ & \qquad\quad sc_{\text{MX}}^{dyn}(c_{\text{mx}}, in, \pi, info, \theta, cba, c_{\text{mxa}}.k.sba, c_{\text{mxa}}.k.mss) \\ & \quad \textit{(vii)} \quad mx(c_{\text{mxa}}.k) \wedge start_{isa}^{\pi}(c_{\text{mxa}}) \vee isa(c_{\text{mxa}}.k) \implies \\ & \qquad\quad sc_{r\text{MIPS}}(\overline{d}, \overline{in}) \\ & \quad \textit{(viii)} \quad isa(c_{\text{mxa}}.k) \wedge mx\left(conf_{\text{MXA}}^{curT}(c').k\right) \implies \\ & \qquad\quad \forall i \in \{1, 7\}.\ \overline{d}'.cpu.core.spr(sr)[i] = 0 \\ & \quad \textit{(ix)} \quad A_{proc}^{stacks}(c.k) \cup A_{new}^{stacks}(c) \subset A_{data} \end{aligned}$$

where the corresponding MIPS-86 configuration $\overline{d}$ and the input $\overline{in}$ are

$$(\overline{d}, \overline{in}) \equiv \begin{cases} (\widehat{d}, \widehat{in}) & : \ mx(c_{\text{mxa}}.k) \wedge start_{isa}^{\pi}(c_{\text{mxa}}) \\ (d, in) & : \ isa(c_{\text{mxa}}.k) \end{cases}$$

and the next MIPS-86 configuration is computed as $\overline{d}' \equiv \delta_{r\text{MIPS}}(\overline{d}, \overline{in})$.

Definition 8.47 (Software Conditions for Kernel Threads). Combining both definition from above, we get

$$\begin{aligned} &sc_{Th}(c, in, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv} \\ &\quad \textit{(i)} \quad sc_{Th}^{stat}(\pi, \theta_{imp}, cba) \\ &\quad \textit{(ii)} \quad sc_{Th}^{dyn}(c, in, \pi, \theta, \theta_{imp}, cba) \end{aligned}$$

8.3.8 Sequential Correctness for Kernel Threads in Concurrent Context

Finally, as it was done in the previous chapters, in order to state the sequential correctness required for the justification of the concurrent model of kernel threads, we introduce auxiliary definitions which are obtained from Section 7.3.7 by substituting the predicates for MXA by the corresponding ones defined for the machine with kernel threads in this chapter.

For defined steps of the machine for kernel threads from a configuration $c_0 \in \mathbb{C}_{Th}$ till the next consistency point, we require that the software conditions hold, the memory region icm is not accessed, and the machine configuration at the next consistency point is well-formed.

$$\begin{aligned} &SCseq_{Th}(c_0, \pi, \theta, \theta_{imp}, cba, icm) \overset{def}{\equiv} \\ &\quad \forall n \in \mathbb{N}, c \in (\mathbb{C}_{Th\perp})^{n+1}, \lambda \in (\Sigma_{\text{MXA}})^n . \\ &\qquad \textit{(i)} \quad c_1 = c_0 \wedge \left(c_1 \longrightarrow^n_{\delta_{Th}^{\pi,\theta,cba},\lambda} c_{n+1}\right) \\ &\qquad \textit{(ii)} \quad \forall i \in [2:n].\ c_i \neq \bot \implies \neg cp_{Th}(c_i, \pi, \theta, cba) \\ &\qquad \textit{(iii)} \quad c_{n+1} \neq \bot \implies cp_{Th}(c_{n+1}, \pi, \theta, cba) \\ &\qquad \implies \\ &\qquad \textit{(i)} \quad \forall i \in \mathbb{N}_n.\ sc_{Th}(c_i, \lambda_i, \pi, \theta, \theta_{imp}, cba) \wedge \\ &\qquad\qquad\qquad noacc_{Th}^{\pi,\theta}(c_i, \lambda_i, icm, cba) \\ &\qquad \textit{(ii)} \quad c_{n+1} \neq \bot \implies wfconf_{Th}^{\pi,\theta,cba}(c_{n+1}) \end{aligned}$$

The restrictions on the number of $\mathcal{IO}$-points between two consistency points and the requirement on their proper implementation are formulated in a predicate for non-empty sequences $c_{imp} \in (\mathbb{C}_{\text{MXA}})^*$, $\sigma \in (\Sigma_{\text{MXA}})^*$, with $|c_{imp}| = |\sigma| + 1$, and $c \in (\mathbb{C}_{Th})^*$, $\tau \in (\Sigma_{\text{MXA}})^*$ with $|c| = |\tau| + 1$:

$$\begin{aligned} &one\mathcal{IO}_{\text{MXA}}^{Th}(c_{imp}, \sigma, c, \tau, \pi, \theta, \theta_{imp}, cba) \overset{def}{\equiv} \\ &\quad \textit{(i)} \quad \forall i, j \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}_{Th}^{\pi,\theta}(c_i, \tau_i, cba) \wedge \mathcal{IO}_{Th}^{\pi,\theta}(c_j, \tau_j, cba) \implies i = j \\ &\quad \textit{(ii)} \quad \forall i, j \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}_{\text{MXA}}^{\pi_{imp},\theta_{imp}}(c_{imp\,i}, \sigma_i, info_{imp}, cba) \wedge \\ &\qquad\qquad \mathcal{IO}_{\text{MXA}}^{\pi_{imp},\theta_{imp}}(c_{imp\,j}, \sigma_j, info_{imp}, cba) \implies i = j \\ &\quad \textit{(iii)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{IO}_{Th}^{\pi,\theta}(c_i, \tau_i, cba)\right) \implies \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{IO}_{\text{MXA}}^{\pi_{imp},\theta_{imp}}(c_{imp\,i}, \sigma_i, info_{imp}, cba)\right) \\ &\quad \textit{(iv)} \quad \left(\exists i \in \mathbb{N}_{|\tau|}.\ \mathcal{OT}_{Th}^{\pi,\theta}(c_i, \tau_i)\right) \iff \left(\exists i \in \mathbb{N}_{|\sigma|}.\ \mathcal{OT}_{\text{MXA}}^{\pi_{imp},\theta_{imp}}(c_{imp\,i}, \sigma_i)\right) \end{aligned}$$

Additionally, the shorthands indicating that there are no consistency points in the given sequence are define as

$$nocp_{Th}(c, \pi, \theta, cba) \overset{def}{\equiv} \forall i \in \mathbb{N}_{|c|}.\ \neg cp_{Th}(c_i, \pi, \theta, cba)$$

$$nocp^{Th}_{\text{MXA}}(c_{imp}, \pi, \theta_{imp}, cba) \overset{def}{\equiv} \forall i \in \mathbb{N}_{|c_{imp}|}.\ \neg cp^{Th}_{\text{MXA}}(c_{imp_i}, \pi, \theta_{imp}, cba)$$

Finally, we formally state the theorem for the sequential correctness for kernel threads in concurrent context in the way done before for lower layers of our model stack. Recall that in the previous chapter we have presented the simulation between the reduced MIPS-86 and MXA machines such that the existence of the MXA consistency points (see Definition 7.36) satisfying the requirements from Definitions 6.34, 7.42 is guaranteed. Therefore, in the correctness of the kernel threads implementation we consider steps of the MXA machine with such consistency points. For brevity, the condition on their existence is silently assumed in the simulation theorem here.

Theorem 8.1 (Sequential Correctness for Kernel Threads in Concurrent Context).

$\forall \pi \in Prog_{\text{MX}}, cba \in \mathbb{B}^{32} \times \mathbb{B}^{32}, \theta, \theta_{imp} \in Params_{\text{CIL}},$

$c_0 \in \mathbb{C}_{Th}, icm \in 2^{\mathbb{B}^{32}}, k \in \mathbb{N}, c_{imp} \in (\mathbb{C}_{\text{MXA}})^{k+1}, \omega \in (\Sigma_{\text{MXA}})^{k}.$

(i) $wfconf^{\pi,\theta,cba}_{Th}(c_0) \wedge wfconf^{Th}_{\text{MXA}}(c_{imp}[1], \pi_{imp}, \theta_{imp}, cba)$

(ii) $cp_{Th}(c_0, \pi, \theta, cba) \wedge cp^{Th}_{\text{MXA}}(c_{imp}[1], \pi, \theta_{imp}, cba)$

(iii) $icm \subset A_{hyp} \setminus A^{code}_{\text{MX}}(info, cba) \wedge consis_{Th}(c_0, c_{imp}[1], \pi, \theta, \theta_{imp}, cba, icm)$

(iv) $c_{imp}[1] \longrightarrow^{k}_{\left(\delta^{\pi_{imp},\theta_{imp},\iota}_{\text{MXA}},\omega\right)} c_{imp}[k+1]$

(v) $nocp^{Th}_{\text{MXA}}(c_{imp}[2:k], \pi, \theta_{imp}, cba)$

(vi) $SCseq_{Th}(c_0, \pi, \theta, \theta_{imp}, cba, icm)$

$\Longrightarrow$

$\exists n \in \mathbb{N}, c'_{imp} \in (\mathbb{C}_{\text{MXA}})^{n+1}, \sigma \in (\Sigma_{\text{MXA}})^{n}, m \in \mathbb{N}, c \in (\mathbb{C}_{Th})^{m+1}, \tau \in (\Sigma_{\text{MXA}})^{m}.$

(i) $n \geq k \wedge c'_{imp}[1:k+1] = c_{imp} \wedge \sigma[1:k] = \omega$

(ii) $c'_{imp}[1] \longrightarrow^{n}_{\left(\delta^{\pi_{imp},\theta_{imp},\iota}_{\text{MXA}},\omega\right)} c'_{imp}[n+1]$

(iii) $cp^{Th}_{\text{MXA}}(c'_{imp}[n+1], \pi, \theta_{imp}, cba) \wedge nocp^{Th}_{\text{MXA}}(c'_{imp}[2:n], \pi, \theta_{imp}, cba)$

(iv) $wfconf^{Th}_{\text{MXA}}(c'_{imp}[n+1], \pi_{imp}, \theta_{imp}, cba) \wedge$
$\forall i \in \mathbb{N}_n. wb^{Th}_{\text{MXA}}(c'_{imp}[i], \sigma_i, \pi_{imp}, cba)$

(v) $c_1 = c_0 \wedge \left(c_1 \longrightarrow^{m}_{\delta^{\pi,\theta,cba}_{Th},\tau} c_{m+1}\right) \wedge wfconf^{\pi,\theta,cba}_{Th}(c_{m+1})$

(vi) $cp_{Th}(c_{m+1}, \pi, \theta, cba) \wedge nocp_{Th}(c[2:m], \pi, \theta, cba)$

(vii) $one\mathcal{IO}^{Th}_{\text{MXA}}(c'_{imp}, \sigma, c, \tau, \pi, \theta, \theta_{imp}, cba)$

(viii) $consis_{Th}(c_{m+1}, c'_{imp}[n+1], \pi, \theta, \theta_{imp}, cba, icm)$

with $\iota \equiv (cba, StIbas[pid_{cur}(c_0)])$ *and* $\pi_{imp} \equiv link(\pi, \pi_{th})$.

Proof: Here, we show in detail the correctness proof only for the thread creation and thread switch primitives. The argumentation about other primitives is similar and left as a bookkeeping exercise. In fact, all conditions introduced in this chapter are sufficient for showing the correctness of the whole kernel threads implementation as well as the required property transfer needed for the pervasive verification in the overall model stack considered in this work.

Thread creation: Let the machine for kernel threads be at the call of the primitive $\mathit{thread\_create}$ in the configuration c_0. In turn, according to Definition 8.22 the MXA machine in the configuration $c_{imp}[1]$ can be at one of the following consistency points:

1. the call of $\mathit{thread\_create}$:

   The primitive is called with the same values of the function arguments in both machines because the consistency relation $\mathit{consis}_{Th}(c_0, c_{imp}[1], \pi, \theta, \theta_{imp}, cba, icm)$ holds.

   According to the well-formedness wf^{run}_{MXA} in Definition 8.28 the MXA machine is at a C-IL consistency point and, therefore, we have $ap^{\theta_{imp}}(c_{imp}[1].\mathcal{M}) \neq pid(c_{imp}[1])$. Since by consis_{Th} from Definition 8.41 we get $c_0.ap = ap^{\theta_{imp}}(c_{imp}[1].\mathcal{M})$ and by $\mathit{consis}^{run}_{Th}$ in Definition 8.35 we conclude $pid(c_{imp}[1]) = pid_{cur}(c_0)$ because the SPRs in both machines are equal, we also have $c_0.ap \neq pid_{cur}(c_0)$. The further argumentation depends on the value of $c_0.ap$:

   a) $c_0.ap \notin \{0, pid_{cur}(c_0)\}$: another processor holds the lock.

      The machine for kernel threads performs a step from $c_1 = c_0$ and stays in the same well-formed configuration, i.e, we have $m = 1$, $\delta^{\pi,\theta,cba}_{Th}(c_1, in) = c_2$, and $c_1 = c_2 = c_0$. In turn, in the implementation if the computation $c_{imp}[1] \longrightarrow^{k}_{\left(\delta^{\pi_{imp},\theta_{imp},\iota}_{\text{MXA}},\omega\right)} c_{imp}[k+1]$ has not reached the next consistency point, we continue the steps till the configuration $c'_{imp}[n+1]$ with $n > k$ where the MXA machine is at the second call of cas in $\mathit{acquire\_lock}$ and, therefore, $cp^{Th}_{\text{MXA}}(c'_{imp}[n+1], \pi, \theta_{imp}, cba)$ holds. Otherwise, we have already $n = k$.

      By a simple bookkeeping one can easily show that the configuration $c'_{imp}[n+1]$ is well-formed because we have only made operations on the stack of the MXA machine and moved to the next consistency point. Moreover, all performed MXA steps are well-behaved, what can be seen from the executed code.

      Since we did not change the configuration of the machine for kernel threads, and the MXA machine configuration $c'_{imp}[n+1]$ differs from $c_{imp}[1]$ by only two additional frames for the functions $\mathit{thread\_create}$ and $\mathit{acquire\_lock}$ on the top of its stack, the consistency relation $\mathit{consis}^{run}_{Th}$ is not violated and the overall $\mathit{consis}_{Th}(c_{m+1}, c'_{imp}[n+1], \pi, \theta, \theta_{imp}, cba, icm)$ holds.

      It is also clear that the predicate $\mathit{one}\mathcal{IO}^{Th}_{\text{MXA}}(c'_{imp}, \sigma, c, \tau, \pi, \theta, \theta_{imp}, cba)$ is satisfied because the call of the primitive in c_1 is marked as $\mathcal{IO}$-, $\mathcal{OT}$-points, and the MXA machine has executed only a single call of cas.

   b) $c_0.ap = 0$: the lock is free.

      The machine for kernel threads performs a step from $c_1 = c_0$ to $c_2 \neq \bot$ (no run-time error by the software conditions) where a new thread th with an ID $\langle a_{tid}\rangle \in \operatorname{dom}(c_1.\mathit{free})$ (see the notation in the semantics of the step in Section 8.1.2.2) is created with a stack not overlapping with stacks of other threads and $\mathit{wfth}^{\pi,\theta}_{new}(th)$ holds.

      $$c_2.\mathit{free} = c_1.\mathit{free} \setminus \{\langle a_{tid}\rangle\} \qquad c_2.ap = pid_{cur}(c_1)$$

      $$c_2.new(x) = \begin{cases} th & : \ x = \langle a_{tid}\rangle \\ c_1.new(x) & : \ \text{otherwise} \end{cases}$$

      One also concludes $\langle a_{tid}\rangle \neq c_1.k.ct$ because we have $c_1.k.ct \in \operatorname{dom}(c_1.k.ready)$ by wfth_{proc} in Definition 8.10 and $\operatorname{dom}(c_1.k.ready) \cap c_1.\mathit{free} = \emptyset$ by Definition 8.12 for

$\mathit{wfconf}_{\mathit{Th}}$. Hence, it is easy to show that the step leads to the well-formed configuration, i.e., $\mathit{wfconf}^{\pi,\theta,cba}_{\mathit{Th}}(c_2)$ holds.

In the implementation the lock is also acquired by the processor $\mathit{pid}(c_{imp}[1])$ and we have to consider the next consistency point, which is, in contrast to the previous case, the assignment $*\mathit{lock} = 0$ inside *release_lock*. Again, we have either $n = k$, or we continue the execution of the MXA machine till $c'_{imp}[n+1]$ with $n > k$ if we have not reached the consistency point in $c_{imp}[k+1]$.

During these steps by the lock acquisition we get

$$ap^{\theta_{imp}}(c'_{imp}[n+1].\mathcal{M}) = \mathit{pid}(c'_{imp}[n+1])$$

By the call of *search_by_tid* we find a pointer to a node of the free thread with the same ID $\langle a_{tid}\rangle$. This operation is feasible because we know $\langle a_{tid}\rangle \in \mathrm{dom}\,(c_1.\mathit{free})$, by Definition 8.37 of $\mathit{consis}^{\mathit{free/new}}_{\mathit{Th}}$ we have $c_1.\mathit{free} = \mathit{TIDs}^{\theta_{imp}}_{\mathit{free}}(c_{imp}[1].\mathcal{M})$, and we have not changed the memory $c_{imp}[1].\mathcal{M}$ till the call of *search_by_tid*.

After the preparation of the well-formed stack and TCB for the new thread according to $\mathit{wf}^{\mathit{new}}_{\mathrm{MXA}}$ in Definition 8.27, the thread is deleted from the list of free threads by the function *remove* and inserted into the list of new threads by *insert_to_end* such that at the end of the steps we have

$$\begin{aligned}\mathit{TIDs}^{\theta_{imp}}_{\mathit{free}}(c'_{imp}[n+1].\mathcal{M}) &= \mathit{TIDs}^{\theta_{imp}}_{\mathit{free}}(c_{imp}[1].\mathcal{M}) \setminus \{\langle a_{tid}\rangle\}\\ \mathit{TIDs}^{\theta_{imp}}_{\mathit{new}}(c'_{imp}[n+1].\mathcal{M}) &= \mathit{TIDs}^{\theta_{imp}}_{\mathit{new}}(c_{imp}[1].\mathcal{M}) \cup \{\langle a_{tid}\rangle\}\end{aligned}$$

Taking into account the changes of the MXA machine configuration from above and the fact that the steps have reached the consistency point indicated by $cp^{\mathit{relvol}}_{\mathrm{MXA}}(c'_{imp}[n+1].k)$ we conclude that the well-formedness $\mathit{wfconf}^{\mathit{Th}}_{\mathrm{MXA}}(c'_{imp}[n+1], \pi_{imp}, \theta_{imp}, cba)$ holds. Moreover, as in the previous case, the steps are well-behaved.

Finally, since the stack of $c'_{imp}[n+1]$ differs from the one in $c_{imp}[1]$ by two additional frames for the functions *thread_create* and *release_lock* and in c_2 and $c'_{imp}[n+1]$ we have made only the updates of the component considered above, using the simulation relation $\mathit{consis}_{\mathit{Th}}(c_0, c_{imp}[1], \pi, \theta, \theta_{imp}, cba, icm)$ we conclude that the simulation relation $\mathit{consis}_{\mathit{Th}}(c_{m+1}, c_{imp}[n+1], \pi, \theta, \theta_{imp}, cba, icm)$ after the steps in both machines still holds.

As in the previous case, the invariant about the $\mathcal{IO}$- and $\mathcal{OT}$-points is also preserved.

2. the call of *cas* in *acquire_lock* except for the first iteration:

   In this case, the stack of the MXA machine in the configuration $c_{imp}[1]$ has already two frames for the functions *thread_create* and *acquire_lock* on its top. Moreover, by the relation $\mathit{consis}^{\mathit{run}}_{\mathit{Th}}(c_0, c_{imp}[1], \pi, \theta_{imp}, cba)$ the parameters of *thread_create* on the stack are coupled with the values of the corresponding arguments in the primitive call in the running thread. The further proof is the same as for case (1) except for the frames of the stack. Namely, in case (a) the stack layout is preserved when we reach in $c'_{imp}[n+1]$ the next call of *cas* in *acquire_lock*, and in case (b) the topmost frame is substituted by the frame for the function *acquire_lock*.

3. the assignment $*\mathit{lock} = 0$ inside *release_lock*:

   Since $cp^{\mathit{relvol}}_{\mathrm{MXA}}(c_{imp}[1].k)$ holds, by the well-formedness $\mathit{wf}^{\mathit{run}}_{\mathrm{MXA}}$ in Definition 8.28 we get $ap^{\theta_{imp}}(c_{imp}[1].\mathcal{M}) = \mathit{pid}(c_{imp}[1])$ and by the same argumentation as in case (1) we conclude $c_0.ap = \mathit{pid}_{cur}(c_0)$.

Therefore, after a single step from $c_1 = c_0$ the machine for kernel threads has the configuration

$$c_2 = inc^{cur}_{loc}(c_1)[ap := 0]$$

The configuration c_2 is well-formed because no other components (except for ap and the location in the topmost frame) have been changed.

As the next consistency point for the MXA machine we consider a C-IL statement directly after the call of *thread_create* in π. Again, if in $c_{imp}[k+1]$ we have not reached this consistency point, we continue the execution of *thread_create* until $c'_{imp}[n+1]$ after the return from the primitive. Otherwise, we set $n = k$. After these steps the lock is released and we get

$$ap^{\theta_{imp}}(c'_{imp}[n+1].\mathcal{M}) = 0$$

The configuration $c'_{imp}[n+1]$ is well-formed because we have only dropped two topmost frames of the stack of the current thread that was well-formed in $c_{imp}[1]$. The well-behaviour trivially follows from the executed steps.

Since by $consis^{run}_{Th}$ from Definition 8.35 before the steps we had $th_{cur}\ (inc^{cur}_{loc}(c)) = \tilde{c}_{imp}[1]$, where $\tilde{c}_{imp}[1]$ is obtained from $c_{imp}[1]$ by deleting these two frames, and the MXA machine has only released the lock and returned from *thread_create*, we get $th_{cur}(c_2) = c'_{imp}[n+1].k$ and conclude that $consis_{Th}(c_2, c'_{imp}[n+1], \pi, \theta, \theta_{imp}, cba, icm)$ holds.

Obviously, the condition $one\mathcal{IO}^{Th}_{\mathrm{MXA}}(c'_{imp}, \sigma, c, \tau, \pi, \theta, \theta_{imp}, cba)$ is satisfied because a single $\mathcal{IO}$-/$\mathcal{OT}$-operation is executed in both machines.

Thread switch: Both machines in the configurations c_0 and $c_{imp}[1]$ are at the call of *thread_run* with the same (by $consis_{Th}$) identifier tid_{to} of the thread we switch to (see the semantics of the step in Section 8.1.2.2).

By the software conditions we know that any step from c_0 does not generate the run-time error and, therefore, $tid_{to} \in \mathrm{dom}\,(c_0.k.ready)$ holds. From Definition 8.36 of $consis^{proc}_{Th}$ we have $\mathrm{dom}\,(c_0.k.ready) = TIDs^{\theta_{imp}}_{ready}(c_{imp}[1].\mathcal{M}, pid_{cur}(c_0))$. Hence, in the memory of the MXA machine the local list of threads ready for scheduling also contains a node of the thread tid_{to}. Moreover, the current thread ID is equal in both machines: $c_0.k.ct = ct^{\theta_{imp}}(c_{imp}[1].\mathcal{M}, pid_{cur}(c_0)) = tid_{from}$.

By performing a single step of the machine for kernel threads from $c_1 = c_0$ we compute its new configuration c_2. In turn, in the implementation, if $c_{imp}[k+1]$ is a configuration of the MXA machine not after the call of *thread_run*, we continue its steps until $c'_{imp}[n+1]$ similarly to the proof of the thread creation. The result of the computation depends on whether we try to switch to the same current thread, or to another thread that can be new or sleeping.

During the steps of the MXA machine we obtain pointers to the nodes of the same threads tid_{to} and tid_{from} because the list of ready threads remains unchanged in the MXA memory since the call of the primitive.

If *both pointers are equal*, the case corresponds to $tid_{to} = tid_{from}$ and in the implementation we return from *thread_run*. The configuration $c'_{imp}[n+1]$ differs from $c'_{imp}[1]$ only by the increased location of the topmost frame. The configuration c_2 of the machine for kernel threads is computed as $c_2 = inc^{cur}_{loc}(c_1)$. Since by Definition 8.35 of $consis^{run}_{Th}$ we have $th_{cur}(c_1) = c'_{imp}[1].k$ and no other components as well as the memory were changed, the simulation relation between c_2 and $c'_{imp}[n+1]$ is preserved. The argumentation about the well-formedness of c_2 and $c'_{imp}[n+1]$, the well-behaviour of the MXA steps, and $\mathcal{IO}$-/$\mathcal{OT}$-points is trivial too.

If *the pointers* to the nodes of the threads being switched *are not equal*, the case corresponds to $tid_{to} \neq tid_{from}$. The configuration c_2 of the machine for kernel threads is obtained from c_1 by updating $c_2.k.ct = tid_{to}$, increasing the location in the topmost frame of the previously running thread tid_{from}, the transformation of its configuration to a configuration of the non-running thread, and enriching the configuration of tid_{to} by the state of the SPRs and TLB (see Definition 8.8) from c_1. Let the configurations of the threads in c_1 and c_2 be

$$th_{from} \equiv th_{cur}(c_1) \qquad th_{to} \equiv c_1.k.ready(tid_{to})$$

$$th'_{from} \equiv c_2.k.ready(tid_{from}) \qquad th'_{to} \equiv th_{cur}(c_2)$$

Hence, in the configuration c_2 we have

$$\forall X \in \{ac, ic\}.\ th'_{to}.k_{\mathrm{mx}}.X = th_{to}.X \qquad th'_{to}.k_{\mathrm{mx}}.spr = th_{from}.k_{\mathrm{mx}}.spr$$

$$\forall Y \in \{sba, mss\}.\ th'_{to}.Y = th_{to}.Y \qquad th'_{to}.tlb = th_{from}.tlb$$

In the implementation the MXA machine performs further MX steps where the states of threads in their TCBs are changed, the current thread ID is also updated by tid_{to}, and the MASM procedure $switch\_stack$ is called. Note that after this call the stack of the thread tid_{from} has frames for $thread\_run$ and $switch\_stack$ on its top. By starting the inline assembly we switch to a MIPS-86 configuration containing a consistent physical stack of the thread tid_{from} in the memory, the program counter pointing to the start of the inline assembly, the actual stack pointer and frame base pointer, etc. After saving these pointers to the thread context of tid_{from}, one can easily conclude that the well-formedness $wf^{sleep}_{\mathrm{MXA}}$ (see Definition 8.26) of the TCB and the stack for the sleeping thread tid_{from} is satisfied because one can again reconstruct a corresponding MX thread configuration well-formed for this sleeping thread according to Definition 8.24. During the execution of the next inline assembly instructions we restore the stack pointer and frame base pointer and begin to operate on the stack of the thread tid_{to}. After loading a word from this stack by $lw\ i1\ bp\ 28$ into the register i_1 (used for passing the first argument during function/procedure calls according to the calling convention from Section 5.1.3), before the return from $switch\_stack$ we attempt to reconstruct an abstract MX thread configuration $k''_{\mathrm{mx}} \in \mathbb{K}_{\mathrm{MX}}$ because the MIPS-86 machine is at a consistency point corresponding to the start of the compiled code of **ret**. The further execution depends on the state of the thread tid_{to}:

1. the thread tid_{to} is sleeping:

   In this case we know that before the steps $wf^{sleep}_{\mathrm{MXA}}(tid_{to}, c'_{imp}[1].\mathcal{M}, \pi_{imp}, \theta_{imp}, cba)$ holds and, therefore, for the sleeping threads there exists $k'_{\mathrm{mx}} \in \mathbb{K}_{\mathrm{MX}}$ satisfying $wf^{sleep}_{\mathbb{K}_{\mathrm{MX}}}(k'_{\mathrm{mx}}, \theta_{imp})$. Then, by $consis^{sleep}_{Th}(th_{to}, tid_{to}, c'_{imp}[1].\mathcal{M}, \pi, \theta_{imp}, cba)$ for $ni' \equiv |k'_{\mathrm{mx}}.ic|$, $ic'_{top} \equiv k'_{\mathrm{mx}}.ic[ni']$ we have

   $$th_{to}.ic = k'_{\mathrm{mx}}.ic[1 : ni' - 1] \qquad th_{to}.ac = ic_{top}[1 : top(ic'_{top}) - 1] \tag{8.2}$$

   Since until the execution of **ret** in $switch\_stack$ we have not modified the stack and fields of TCB for the thread tid_{to} except for its state, we successfully obtain the abstract MX thread configuration k''_{mx} (mentioned above) and by the reconstruction conclude

   $$k''_{\mathrm{mx}}.ic = k'_{\mathrm{mx}}.ic \tag{8.3}$$

   The active context of k''_{mx} is of MASM type and contains a single stack frame for the procedure $switch\_stack$, the current state of SPRs equal to the registers content in $c'_{imp}[1]$ because we have not rewritten them, and the GPRs with i_1 updated in the inline assembly. The topmost frame of its last inactive context corresponds to the function $thread\_run$. The further steps are performed in the MX semantics, namely, the MASM step for **ret** and the C-IL return from $thread\_run$. Note that the register i_1 is not used during these steps and, therefore, its loaded value is ignored. Formally, after these steps in the configuration $c'_{imp}[n+1]$ we get for $ni'' \equiv |k''_{\text{mx}}.ic|$, $ic''_{top} \equiv k''_{\text{mx}}.ic[ni'']$:

$$\begin{aligned}
c'_{imp}[n+1].k.k_{\text{mx}}.ic &= k''_{\text{mx}}.ic[1 : ni'' - 1] \\
c'_{imp}[n+1].k.k_{\text{mx}}.ac &= ic_{top}[1 : top(ic''_{top}) - 1] \\
c'_{imp}[n+1].k.k_{\text{mx}}.spr &= c'_{imp}[1].k.k_{\text{mx}}.spr \\
c'_{imp}[n+1].k.tlb &= c'_{imp}[1].k.tlb \\
c'_{imp}[n+1].k.sba &= sba^{\theta_{imp}}(c'_{imp}[1].\mathcal{M}, tid_{to}) \\
c'_{imp}[n+1].k.mss &= mss^{\theta_{imp}}(c'_{imp}[1].\mathcal{M}, tid_{to})
\end{aligned}$$

Note that along with k''_{mx} the values of sba and mss belonging to the sleeping thread are also correctly reconstructed from the stack information abstraction instantiated in Section 8.2.2 for the MXA machine. Since we do not change these fields in the TCB of the thread, they are equal to the values fetched from $c'_{imp}[1].\mathcal{M}$.

Now, using equations (8.2) – (8.3), $th_{from} = c'_{imp}[1].k$ from $consis^{run}_{Th}(c_1, c'_{imp}[1], \pi, \theta_{imp}, cba)$, and $th_{to}.sba = sba^{\theta_{imp}}(c'_{imp}[1].\mathcal{M}, tid_{to})$, $th_{to}.mss = mss^{\theta_{imp}}(c'_{imp}[1].\mathcal{M}, tid_{to})$ from the simulation relation $consis^{sleep}_{Th}(th_{to}, tid_{to}, c'_{imp}[1].\mathcal{M}, \pi, \theta_{imp}, cba)$ we easily obtain

$$\begin{aligned}
c'_{imp}[n+1].k.k_{\text{mx}}.ic &= th_{to}.ic \\
c'_{imp}[n+1].k.k_{\text{mx}}.ac &= th_{to}.ac \\
c'_{imp}[n+1].k.k_{\text{mx}}.spr &= th_{from}.k_{\text{mx}}.spr \\
c'_{imp}[n+1].k.tlb &= th_{from}.tlb \\
c'_{imp}[n+1].k.sba &= th_{to}.sba \\
c'_{imp}[n+1].k.mss &= th_{to}.mss
\end{aligned}$$

From the computation of th'_{to} considered above we conclude

$$th_{cur}(c_2) = c'_{imp}[n+1].k$$

Therefore, after the steps of both machines the simulation relation $consis^{run}_{Th}(c_2, c'_{imp}[n+1], \pi, \theta_{imp}, cba)$ holds. It is also easy to see that we have $consis^{sleep}_{Th}(th'_{from}, tid_{from}, c'_{imp}[n+1].\mathcal{M}, \pi, \theta_{imp}, cba)$ because, as mentioned above, during the execution of the inline assembly the well-formedness wf^{sleep}_{MXA} of the TCB and the stack for the thread tid_{from} is satisfied and these data structures in the MXA memory are preserved during the MXA steps until $c'_{imp}[n+1]$. Since in the machines we have updated only the components mentioned during the proof, we finally conclude that the simulation relation $consis_{Th}$ between c_2 and $c'_{imp}[n+1]$ holds. Moreover, $c'_{imp}[n+1]$ is well-formed for the kernel threads implementation.

2. the thread tid_{to} is new:

   In this case before **ret** we cannot reconstruct k''_{mx} because the physical stack (see Figure 8.3) of the new thread tid_{to} satisfies $wf^{new}_{\text{MXA}}(tid_{to}, c'_{imp}[1].\mathcal{M}, \theta_{imp})$ and no corresponding abstract MX machine stack consistent wrt. the MX compiler consistency exists.

   Therefore, we continue MIPS-86 ISA steps executing the compiled code of **ret**. During these steps, according to the implementation of **ret** given in Section 6.1.1.1, we know that the topmost frame corresponding to the procedure $switch\_stack$ is dropped and the program counter is set to the return address stored in this frame.

Since by wf^{new}_{MXA} we know that this return address is equal to the address of the entry function in the TCB, and by $consis^{new}_{Th}$ its is coupled with the function name of the stack frame in the abstract configuration of the thread tid_{to}, the MIPS-86 steps proceed with the execution of the prologue (see Section 6.1.1.2) of this function until the next consistency point, where the program counter points to the beginning of the compiled code of the first statement in the body of the thread entry function. Moreover, we know that the argument of the entry function resides in the register i_1. Therefore, by reconstructing an abstract MX thread configuration with the base address and maximal size of the stack belonging to the thread tid_{to}, we obtain the well-formed configuration $c'_{imp}[n+1].k$ with $c'_{imp}[n+1].k = th_{cur}(c_2)$.

By taking into account the argumentation about the thread tid_{from} from the previous case, we finally conclude that the simulation relation $consis_{Th}$ between c_2 and $c'_{imp}[n+1]$ holds also in case of the thread switch to the new thread.

In both cases from above the condition $one\mathcal{IO}^{Th}_{\mathrm{MXA}}(c'_{imp}, \sigma, c, \tau, \pi, \theta, \theta_{imp}, cba)$ is satisfied because we have not performed $\mathcal{IO}$-operations. This finishes the correctness proof of the thread switch considered in this thesis.

As for the usual MXA steps of the machine for kernel threads, one can easily show that both machines execute the same steps. In case the stack substitution is performed by the kernel in an inline assembly of the program π, the software condition for the kernel threads guarantee that the configurations of all threads except for the running one are preserved and no thread switch is performed without the execution of the corresponding primitive.

$\square$

8.4 Justification of the Concurrent Model

8.4.1 Cosmos Model Instantiations

Given a program $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in Prog_{\mathrm{MX}}$ of the hypervisor / OS kernel, the kernel threads framework $\pi_{th} = \left(\pi^{th}_\mu, \pi^{th}_{\mathrm{cil}}\right) \in Prog_{\mathrm{MX}}$ such that the linked implementation program $\pi_{imp} \equiv \left(\pi^{imp}_\mu, \pi^{imp}_{\mathrm{cil}}\right)$ is computed as $\pi_{imp} = link(\pi, \pi_{th})$, the environment parameters θ, θ_{imp} satisfying $valid^{par}_{Th}(\theta, \theta_{imp})$, code base addresses $cba = (cba_{\mathrm{cil}}, cba_\mu) \in \mathbb{B}^{32} \times \mathbb{B}^{32}$, and $c\iota \equiv (cba, StIbas)$, we define the instantiations $S^{\pi,\theta,\theta_{imp},cba}_{Th}, S^{\pi_{imp},\theta_{imp},c\iota}_{imp} \in \mathbb{S}$ of the *Cosmos* models for the kernel threads machine and the extended mixed machine implementing it on $np \in \mathbb{N}$ processors.

Concurrent Machine for Kernel Threads

- $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{A}$ – the set of memory addresses of the *Cosmos* model for kernel threads must include not only byte addresses, but also the names new, $free$, ap of the abstract components present in the kernel machine configuration. The reason is that all resources that can be shared have to be modelled by the memory $m : \mathcal{A} \to \mathcal{V}$ in the concurrent machine state. Moreover, in order to see which threads are owned or shared, we include $\mathbb{N}_{mtid}$ into this set. Since we get configurations of threads from new, $free$, and processor's threads, for the *Cosmos* machine memory we will set $\forall tid \in \mathbb{N}_{mtid}.\ m(tid) = \bot$.

Let the set of memory addresses not occupied by the compiled code and global variables of the framework be denoted as

$$\mathcal{A}' \equiv \mathbb{B}^{32} \setminus \big(A^{code}_{\text{MX}}(\mathit{info}_{imp}, cba) \setminus A^{code}_{\text{MX}}(\mathit{info}, cba)\big) \cup A^{data}_{th}(\pi_{th}, \theta_{imp})$$

Then, we instantiate the set of addresses $\mathcal{A}$ as

$$S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{A} = \mathcal{A}' \cup \{new, \mathit{free}, ap\} \cup \mathbb{N}_{mtid}$$

where new, free, ap show the abstract component to be addressed.

- $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{V}$ – the set of *Cosmos* model memory values includes then all possible values for the addressed components:

$$S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{V} = \mathbb{B}^{8} \cup (\mathbb{N}_{mtid} \rightharpoonup \mathbb{K}^{nrun}_{Th}) \cup 2^{\mathbb{N}_{mtid}} \cup [0 : np] \cup \{\perp\}$$

- $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{R} = A^{const}_{\text{CIL}}(\pi_{\text{cil}}, \theta) \cup A_{code} \setminus \big(A^{code}_{\text{MX}}(\mathit{info}_{imp}, cba) \setminus A^{code}_{\text{MX}}(\mathit{info}, cba)\big)$
- $S^{\pi,\theta,\theta_{imp},cba}_{Th}.nu = np$
- $S^{\pi,\theta,\theta_{imp}cba}_{Th}.\mathcal{U} = \mathbb{K}^{proc}_{Th} \cup \{\perp\}$
- $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{E} = \Sigma_{\text{MXA}}$

Before we instantiate other components, we introduce the function

$$c_{Th}(u, m) \overset{def}{\equiv} c$$

composing the configuration $c \in \mathbb{C}_{Th}$ of the sequential machine for the kernel threads on the base of the unit's configuration u and the *Cosmos* model memory m mapping either read-only addresses $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{R}$ or all $S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{A}$ to the corresponding values.

The result c depends whether the abstract components ap, new, free are in the given portion of the *Cosmos* machine memory or not. In the latter case we choose any values of the components because they are not involved into the underlying computations and are needed only for the formalism used in definitions.

$$c.k = u \qquad c.\mathcal{M}(a) = \begin{cases} m(a) & : \ a \in \operatorname{dom}(m) \\ \epsilon\mathbb{B}^{32} & : \ \text{otherwise} \end{cases} \qquad c.ap = \begin{cases} m(ap) & : \ ap \in \operatorname{dom}(m) \\ \epsilon\mathbb{N}_{np} & : \ \text{otherwise} \end{cases}$$

$$c.new = \begin{cases} m(new) & : \ new \in \operatorname{dom}(m) \\ \epsilon\,(\mathbb{N}_{mtid} \rightharpoonup \mathbb{K}^{nrun}_{Th}) & : \ \text{otherwise} \end{cases} \qquad c.\mathit{free} = \begin{cases} m(\mathit{free}) & : \ \mathit{free} \in \operatorname{dom}(m) \\ \epsilon 2^{\mathbb{N}_{mtid}} & : \ \text{otherwise} \end{cases}$$

Therefore, using $c = c_{Th}(u, m)$ we can now apply the functions and predicates defined for the kernel threads machine from the previous section.

- $$S^{\pi,\theta,\theta_{imp},cba}_{Th}.reads(u, m, in) = \begin{cases} reads^{\pi,\theta}_{Th}(c, in, cba) & : \ u \neq \perp \wedge \neg primnsch^{\pi,\theta}_{Th}(c) \\ reads^{\pi,\theta}_{Th}(c, in, cba) \cup \{ap\} & : \ u \neq \perp \wedge primnsch^{\pi,\theta}_{Th}(c) \wedge \\ & \quad c.ap \neq 0 \\ reads^{\pi,\theta}_{Th}(c, in, cba) \cup & : \ u \neq \perp \wedge primnsch^{\pi,\theta}_{Th}(c) \wedge \\ \{ap, new, \mathit{free}\} & \quad a.ap = 0 \\ \emptyset & : \ \text{otherwise} \end{cases}$$

Note that we do not include thread identifies into the $reads$-set because we do not modify $\bot$ in m. For arguing about the ownership memory access policy it is sufficient to consider new and *free* in this set.

- $S_{Th}^{\pi,\theta,\theta_{imp},cba}.\delta(u,m,in)$ – the transition function of the unit is instantiated wrt. the computation $c' \equiv \delta_{Th}^{\pi,\theta,cba}(c,in)$ of the next configuration of the sequential machine for kernel threads. If the step is undefined, we obviously have $c' \notin \mathbb{C}_{Th\bot}$.

$$S_{Th}^{\pi,\theta,\theta_{imp},cba}.\delta(u,m,in) = \begin{cases} (c'.k,m') & : u \neq \bot \wedge \neg primnsch_{Th}^{\pi,\theta}(c) \wedge c' \neq \bot \\ (c'.k,m'') & : u \neq \bot \wedge primnsch_{Th}^{\pi,\theta}(c) \wedge c' \neq \bot \wedge \\ & \quad c.ap = pid_{cur}(c) \\ (c'.k,m''') & : u \neq \bot \wedge primnsch_{Th}^{\pi,\theta}(c) \wedge c' \neq \bot \wedge c.ap = 0 \\ (c'.k,m_\emptyset) & : u \neq \bot \wedge primnsch_{Th}^{\pi,\theta}(c) \wedge c' \neq \bot \wedge \\ & \quad c.ap \notin \{0, pid_{cur}(c)\} \\ (\bot, m_\emptyset) & : u \neq \bot \wedge c' = \bot \\ undefined & : \text{otherwise} \end{cases}$$

where m', m'', and m''' are the updated portions of the *Cosmos* machine memory computed as

$$m' \equiv c'.\mathcal{M}|_{writes_{Th}^{\pi,\theta}(c,in)}$$

$$m''(x) = \begin{cases} c'.ap & : x = ap \\ undefined & : \text{otherwise} \end{cases} \qquad m'''(x) = \begin{cases} c'.x & : x \in \{ap, new, free\} \\ undefined & : \text{otherwise} \end{cases}$$

and $m_\emptyset$ is the empty function satisfying $\operatorname{dom}(m_\emptyset) = \emptyset$.

- $S_{Th}^{\pi,\theta,\theta_{imp},cba}.\mathcal{IP}(u,m,in) = (u \neq \bot \implies cp_{Th}(u,\pi,\theta,cba))$

- $S_{Th}^{\pi,\theta,\theta_{imp},cba}.\mathcal{IO}(u,m,in) = \Big(u \neq \bot \implies \mathcal{IO}_{Th}^{\pi,\theta}(c,in,cba)\Big)$

- $S_{Th}^{\pi,\theta,\theta_{imp},cba}.\mathcal{OT}(u,m,in) = \Big(u \neq \bot \implies \mathcal{OT}_{Th}^{\pi,\theta}(c,in)\Big)$

MXA Machine Implementing Concurrent Kernel Threads

The instantiation $S_{imp}^{\pi_{imp},\theta_{imp},c\iota} \in \mathbb{S}$ of the *Cosmos* model with the mixed machine implementing the kernel threads is almost equal to the MXA instantiation $S_{\text{MX}}^{\pi_{imp},\theta_{imp},c\iota}$ from Chapter 7 for π_{imp}, θ_{imp}, and stack information abstraction obtained in Section 8.2.2. The only exception is the interleaving points determined by Definition 8.22.

- For components $X \in \{\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{OT}\}$ the instantiation is the same as for the MXA machine

$$S_{imp}^{\pi_{imp},\theta_{imp},c\iota}.X = S_{\text{MX}}^{\pi_{imp},\theta_{imp},c\iota}.X$$

- $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}.\mathcal{IP}(u,m,in) = \big(u \neq \bot \implies cp_{\text{MXA}}^{Th}(u,\pi,\theta_{imp},cba)\big)$

8.4.2 Sequential Simulation Theorem

Now, we instantiate the sequential simulation framework $R^{S_{Th}}_{S_{imp}}(\pi, \theta, \theta_{imp}, cba) \in \mathbb{R}$ for our *Cosmos* models $S^{\pi,\theta,\theta_{imp},cba}_{Th}, S^{\pi_{imp},\theta_{imp},c\iota}_{imp} \in \mathbb{S}$. Since we do not have a specific simulation parameter, we consider it equal to $\perp$ and skip it in all predicates.

For configurations $\tilde{c} = (u, m)$, $c \equiv c_{Th}(u, m)$, $\tilde{c}_{imp} = (u_{imp}, m_{imp})$, a subset icm of the *Cosmos* model memory, and $icm' \equiv icm \setminus (\{new, free, ap\} \cup \mathbb{N}_{mtid})$ we define

$$R^{S_{Th}}_{S_{imp}}(\pi, \theta, \theta_{imp}, cba). \begin{cases} \mathcal{P} & = \perp \\ sim(\tilde{c}_{imp}, \tilde{c}, icm) & = icm' \subset A_{hyp} \setminus A^{code}_{\mathrm{MX}}(info, cba) \wedge \\ & \quad \big(u \neq \perp \implies \\ & \quad consis_{Th}(c, (u_{imp}.k, m_{imp}), \pi, \theta, \theta_{imp}, cba, icm')\big) \\ \mathcal{CP}a(u) & = (u \neq \perp \implies cp_{Th}(u, \pi, \theta, cba)) \\ \mathcal{CP}c(u_{imp}) & = \big(u \neq \perp \implies cp^{Th}_{\mathrm{MXA}}(u_{imp}.k, \pi, \theta_{imp}, cba)\big) \\ wfa(\tilde{c}) & = u \neq \perp \wedge wfconf^{\pi,\theta,cba}_{Th}(c_{Th}(u, m)) \\ wfc(\tilde{c}_{imp}) & = u \neq \perp \wedge wfconf^{Th}_{\mathrm{MXA}}((u_{imp}.k, m_{imp}), \pi_{imp}, \theta_{imp}, cba) \\ suit(in) & = 1 \\ sc(\tilde{c}, in) & = (u \neq \perp \implies sc_{Th}(c, in, \pi, \theta, \theta_{imp}, cba)) \\ wb(\tilde{c}_{imp}, in) & = wb^{Th}_{\mathrm{MXA}}((u_{imp}.k, m_{imp}), in, \pi_{imp}, cba) \end{cases}$$

Note that we use $icm' \in 2^{\mathbb{B}^{32}}$ because in the sequential correctness of the machine for kernel threads (see Section 8.3.8) the corresponding predicates are defined for byte addresses of possible inconsistent memory regions.

Using Theorem 8.1, one can easily prove the generalized sequential simulation Theorem 2.3 for our case and, therefore, can state the correctness of the kernel thread implementation for any hypervisor / OS kernel program and corresponding parameters.

Theorem 8.2 (Sequential Kernel Threads Correctness for *Cosmos* Model Simulation). *The generalized sequential simulation Theorem 2.3 holds for any* Cosmos *models* $S^{\pi,\theta,\theta_{imp},cba}_{Th}, S^{\pi_{imp},\theta_{imp},c\iota}_{imp} \in \mathbb{S}$ *and the simulation framework* $R^{S_{Th}}_{S_{imp}}(\pi, \theta, \theta_{imp}, cba) \in \mathbb{R}$ *instantiated wrt. any given hypervisor / OS kernel program* $\pi = (\pi_\mu, \pi_{\mathrm{cil}}) \in Prog_{\mathrm{MX}}$ *with inline assembly, the environment parameters* $\theta, \theta_{imp} \in Params_{\mathrm{CIL}}$*, system information* $\iota \equiv (cba, StIba)$*, the framework* π_{th}*, and the number* $np \in \mathbb{N}$ *of processors in the multi-core MIPS-86 machine.*

8.4.3 Concurrent Model Simulation and Its Application Overview

As the last step for the concurrent model simulation between the machine for kernel threads and the MXA machine implementing them, we instantiate the missing predicates.

As it was done in the previous chapter, we will use the predicate $oginv_{Th}(E)$ requiring for $E \in \mathbb{C}_{S_a}$ with $S_a \equiv S^{\pi,\theta,\theta_{imp},cba}_{Th}$ that the gust addresses are always shared and not owned. Its formulation can be found in Definition 7.50 and we do not repeat it here.

Definition 8.48 (Ownership Invariant for Kernel Threads Machine). Given a configuration E of the *Cosmos* machine for kernel threads, we require that (i) – (ii) threads belonging to a given processor as well as their stacks are local, (iii) in case the components *new* and *free* are accessed by a processor, *ap* is shared and owned by this processor whereas *new*, *free*, all threads present in these components, and stacks of the new threads are local. (iv) Otherwise, these components and addresses are shared and unowned.

Let $c_p \in \mathbb{C}_{Th}$ be computed as $c_p \equiv c_{Th}(E.u_p, E.m)$. Then, we formally define

$$
\begin{aligned}
&\mathit{ocinv}_{Th}(E) \overset{def}{\equiv}\\
&\quad \textit{(i)} \quad \forall p \in \mathbb{N}_{nu}.\ \mathrm{dom}\,(E.u_p.\mathit{ready}) \cup \mathrm{dom}\,(E.u_p.\mathit{fin}) \subset E.\mathcal{O}_p \setminus E.\mathcal{S}\\
&\quad \textit{(ii)} \quad \forall p \in \mathbb{N}_{nu}.\ A^{\mathit{stacks}}_{\mathit{proc}}(E.u_p) \subset E.\mathcal{O}_p \setminus E.\mathcal{S}\\
&\quad \textit{(iii)} \quad \forall p \in \mathbb{N}_{nu}.\ E.m(ap) = p \implies\\
&\qquad\quad \textit{(a)} \quad \{\mathit{new}, \mathit{free}\} \subset E.\mathcal{O}_p \setminus E.\mathcal{S} \wedge ap \in E.\mathcal{O}_p \cap E.\mathcal{S}\\
&\qquad\quad \textit{(b)} \quad \mathrm{dom}\,(E.m(\mathit{new})) \cup E.m(\mathit{free}) \subset E.\mathcal{O}_p \setminus E.\mathcal{S}\\
&\qquad\quad \textit{(c)} \quad A^{\mathit{stacks}}_{\mathit{new}}(c_p) \subset E.\mathcal{O}_p \setminus E.\mathcal{S}\\
&\quad \textit{(iv)} \quad E.m(ap) = 0 \implies\\
&\qquad\quad \textit{(a)} \quad \{ap, \mathit{new}, \mathit{free}\} \subset E.\mathcal{S} \setminus \textstyle\bigcup_{p \in \mathbb{N}_{np}} E.\mathcal{O}_p\\
&\qquad\quad \textit{(b)} \quad \mathrm{dom}\,(E.m(\mathit{new})) \cup E.m(\mathit{free}) \subset E.\mathcal{S} \setminus \textstyle\bigcup_{p \in \mathbb{N}_{np}} E.\mathcal{O}_p\\
&\qquad\quad \textit{(c)} \quad A^{\mathit{stacks}}_{\mathit{new}}(c_p) \subset E.\mathcal{S} \setminus \textstyle\bigcup_{p \in \mathbb{N}_{np}} E.\mathcal{O}_p
\end{aligned}
$$

Therefore, the property P_{S_a} on configurations of the *Cosmos* machine for kernel threads is specified as

$$P_{S_a}(E) \overset{def}{\equiv} \mathit{oginv}_{Th}(E) \wedge \mathit{ocinv}_{Th}(E)$$

In the shared invariant we have to couple the ownership states of both machines and the shared portions of their memories, as well as to indicate that such shared memories are well-formed.

First, we introduce shorthands for the computation of addresses allocated for the data structures declared in the framework π_{th} (see Section 8.2.1.1):

- pointers to the head and the tail of the global list of new threads:

$$A_{\mathit{new}} \equiv \bigcup_{x \in \{hd, tl\}} \{\theta_{imp}.\mathit{alloc}_{gvar}(\mathit{new\_x})\}_4$$

- pointers to the global list of free threads:

$$A_{\mathit{free}} \equiv \bigcup_{x \in \{hd, tl\}} \{\theta_{imp}.\mathit{alloc}_{gvar}(\mathit{free\_x})\}_4$$

- the spinlock:

$$A_{\mathit{lock}} \equiv \{\theta_{imp}.\mathit{alloc}_{gvar}(\mathit{lock})\}_4$$

- for a given processor $p \in \mathbb{N}_{np}$ an element of the array of current threads:

$$A_{\mathit{cur}}(p) \equiv \left\{ ba^{\theta_{imp}}_{\mathit{elem}}(\mathit{current}, \mathbf{u32}, p) \right\}_4$$

- for a given processor $p \in \mathbb{N}_{np}$ pointers to the list of the ready threads:

$$A_{\mathit{ready}}(p) \equiv \bigcup_{x \in \{hd, tl\}} \left\{ ba^{\theta_{imp}}_{\mathit{elem}}(\mathit{ready\_x}, \mathbf{ptr}(\mathit{Thread}), p) \right\}_4$$

- for a given processor $p \in \mathbb{N}_{np}$ pointers to the list of the finished threads:

$$A_{\mathit{fin}}(p) \equiv \bigcup_{x \in \{hd,tl\}} \left\{ ba^{\theta_{imp}}_{elem}(\mathit{fin\_x}, \mathbf{ptr}(\mathit{Thread}), p) \right\}_4$$

- addresses occupied by an element of the type *Thread* for a given thread $tid \in \mathbb{N}_{mtid}$ in the array *threads*:

$$A_{th}(tid) \equiv \left\{ ba^{\theta_{imp}}_{elem}(\mathit{threads}, \mathit{Thread}, tid) \right\}_{size_{\theta_{imp}}(\mathit{Thread})}$$

- addresses occupied by elements *Thread* for threads with IDs from a set $A \subset S^{\pi,\theta,\theta_{imp},cba}_{Th}.\mathcal{A}$:

$$A_{threads}(A) \equiv \bigcup_{tid \in A \cap \mathbb{N}_{mtid}} A_{th}(tid)$$

Definition 8.49 (Shared Invariant for Concurrent Kernel Threads Simulation). Given parts $(m, \mathcal{S}, \mathcal{R}, \mathcal{O})$ and $(m_{imp}, \mathcal{S}_{imp}, \mathcal{R}_{imp}, \mathcal{O}_{imp})$ of configurations of the *Cosmos* machines for kernel threads and their implementation, we define the shared invariant wrt. Definition 8.48. Using the following shorthands

$$\tilde{m} \equiv \lceil m_{imp} \rceil_{\mathbb{B}^{32}}$$

$$A_{abs} \equiv \{new, \mathit{free}, ap\} \cup \mathbb{N}_{mtid}$$

$$\mathcal{S}' \equiv \mathcal{S} \setminus A_{abs} \qquad \mathcal{O}'(p) \equiv \mathcal{O}(p) \setminus A_{abs}$$

$$\widetilde{\mathcal{S}}_{imp}(\mathcal{S}) \equiv \begin{cases} A_{new} \cup A_{\mathit{free}} & : \{new, \mathit{free}\} \subset \mathcal{S} \\ \emptyset & : \text{otherwise} \end{cases}$$

$$\widetilde{\mathcal{O}}_{imp}(\mathcal{O}, p) \equiv \begin{cases} A_{lock} \cup A_{new} \cup A_{\mathit{free}} & : \{ap, new, \mathit{free}\} \subset \mathcal{O}(p) \\ \emptyset & : \text{otherwise} \end{cases}$$

and denoting the thread identifiers not existing in the system and memory addresses occupied by array elements with such IDs [5]

$$\mathit{TIDs}_{\perp}(\mathcal{S}, \mathcal{O}) \equiv \{0\} \cup \mathbb{N}_{tid} \setminus \left(\mathcal{S} \cup \bigcup_{p \in \mathbb{N}_{np}} \mathcal{O}(p) \right)$$

$$A_{\perp}(\mathcal{S}, \mathcal{O}) \equiv A_{ready}(0) \cup A_{\mathit{fin}}(0) \cup A_{threads}\left(\mathit{TIDs}_{\perp}\right)$$

[5] Recall from Section 8.2.1.1 that the arrays declared in π_{th} have elements with indices 0 which are not used for simplicity. Moreover, in the abstract model for kernel threads we do not require that all thread identifiers from $\mathbb{N}_{mtid}$ are present, therefore, the array elements with such indices are not accessed in the implementation. This is guaranteed by the software conditions assuming the absence of run-time errors generated in case of an operation on a thread with a non-existing identifier.

we formally have

$$sinv^{Th}_{imp}(\pi, \theta, \theta_{imp}, cba)\big((m, \mathcal{S}, \mathcal{R}, \mathcal{O}), (m_{imp}, \mathcal{S}_{imp}, \mathcal{R}_{imp}, \mathcal{O}_{imp})\big) \overset{def}{\equiv}$$

$$\begin{array}{rl}
(i) & \mathcal{R}_{imp} = \mathcal{R} \cup \big(A^{code}_{\mathrm{MX}}(info_{imp}, cba) \setminus A^{code}_{\mathrm{MX}}(info, cba)\big) \\
(ii) & \mathcal{S}_{imp} = \mathcal{S}' \cup A_{lock} \cup A_{threads}(\mathcal{S}) \cup \widetilde{\mathcal{S}}_{imp}(\mathcal{S}) \cup A_{\perp}(\mathcal{S}, \mathcal{O}) \\
(iii) & \forall p \in \mathbb{N}_{nu}. \\
 & \mathcal{O}_{imp}(p) = \mathcal{O}'(p) \cup A_{cur}(p) \cup A_{ready}(p) \cup A_{fin}(p) \cup A_{threads}(\mathcal{O}(p)) \cup \widetilde{\mathcal{O}}_{imp}(\mathcal{O}, p) \\
(iv) & \forall a \in \mathrm{dom}\,(m) \setminus A_{abs}.\ m(a) = m_{imp}(a) \\
(v) & m(ap) = ap^{\theta_{imp}}(\tilde{m}) \\
(vi) & m(ap) = 0 \implies \\
 & \quad (a)\quad m(free) = TIDs^{\theta_{imp}}_{free}(\tilde{m}) \\
 & \quad (b)\quad \forall t \in m(free).\ state^{\theta_{imp}}(\tilde{m}, t) = \texttt{TS\_FREE} \\
 & \quad (c)\quad \mathrm{dom}\,(m(new)) = TIDs^{\theta_{imp}}_{new}(\tilde{m}) \\
 & \quad (d)\quad \forall t \in \mathrm{dom}\,(m(new))\,.\ consis^{new}_{Th}(m(new, t), t, \tilde{m}, \pi, \theta, \theta_{imp}, cba) \\
 & \quad (e)\quad \forall t \in \mathrm{dom}\,(m(new))\,.\ wfth^{\pi,\theta}_{new}(m(new, t)) \wedge \\
 & \qquad\qquad wf^{new}_{\mathrm{MXA}}(t, \tilde{m}, \theta_{imp}) \wedge state^{\theta_{imp}}(\tilde{m}, t) = \texttt{TS\_NEW} \\
 & \quad (f)\quad valid^{stack}_{Th}(m(new)) \\
 & \quad (g)\quad \mathrm{dom}\,(m(new)) \cap \mathrm{dom}\,(m(free)) = \emptyset
\end{array}$$

Note that the addresses $A_{\perp}(\mathcal{S}, \mathcal{O})$ not used for the implementation of the machine with kernel threads have to be present in the ownership state for the preservation of the ownership invariant in the MXA machine. Therefore, as one of solutions for this issue we set them shared and not owned.

Moreover, (vi) in Definition 8.49 repeats for the shared components the corresponding parts of the well-formedness for both machines and the simulation relation in case of the free lock.

Since the instantiated concurrent simulation relation and Definitions 8.48, 8.49 provide everything needed for the argumentation about the unit's configuration of the MXA machine implementing concurrent threads, we instantiate the unit invariant as

$$uinv^{Th}_{imp}(\pi, \theta, \theta_{imp}, cba)(u_p, \mathcal{O}_p, \mathcal{S}) \overset{def}{\equiv} 1$$

By setting $S_a \equiv S^{\pi,\theta,\theta_{imp},cba}_{Th}$ and $S_c \equiv S^{\pi_{imp},\theta_{imp},c\iota}_{imp}$ we restate and prove here Assumptions 2.1–2.4 needed for the concurrent simulation.

First, we show that the ownership invariant for the MXA machine follows from $ocinv_{Th}(E)$, the ownership invariant of the machine for kernel threads, and the shared invariant.

Lemma 8.1 (Ownership Invariant for the MXA Machine Implementing Kernel Threads).

$$\forall E \in \mathbb{C}_{S_a}, \forall D \in \mathbb{C}_{S_c}.\ oinv(E.G) \wedge ocinv_{Th}(E) \wedge sinv(D, E) \implies oinv(D.G)$$

Proof: In order to prove the lemma, we recall from Definition 8.49 how the ownership states of both machines in E and D are coupled and consider every component in the computation of the ownership state of the MXA machine.

The owned addresses of each unit p in the MXA machine contain the addresses $E.\mathcal{O}_p \setminus A_{abs}$, which are disjoint according to $oinv(E.G)$ for the machine with threads:

$$\forall p, q.\, p \neq q \Longrightarrow E.\mathcal{O}_p \cap E.\mathcal{O}_q = \emptyset \tag{8.4}$$

Moreover, by the shared invariant any unit p owns only its element of the array of current threads as well as the pointers to the lists of ready and finished threads. Since according to $ocinv_{Th}(E)$ and (8.4) the abstract components ap, new, and $free$ can also be owned only by a single unit in the machine, this holds also for the lock and the pointers to the lists of free and new threads in the implementation. By (8.4) the sets of owned thread identifiers are disjoint for $p \neq q$. Moreover, in the implementation each thread is represented by a separate element of type *Thread* in the array *threads*. Hence, $A_{threads}(D.\mathcal{O}_p) \neq A_{threads}(D.\mathcal{O}_q)$ holds and we conclude the first part of $oinv(D.G)$: $\forall p, q.\, p \neq q \Longrightarrow D.\mathcal{O}_p \cap D.\mathcal{O}_q = \emptyset$.

By the shared invariant the owned and shared addresses in the implementation do not contain the code region. Moreover, in the framework π_{th} we do not have variables declared as constants. All constants present in the linked program come from the program π of the hypervisor / OS kernel and, therefore, belong to the read-only addresses of the machine for kernel threads. Since these addresses are unowned and not shared according to $oinv(E.G)$, we obtain the second part of $oinv(D.G)$:

$$\forall p.\, D.O_p \cap S_c.\mathcal{R} = \emptyset \;\wedge\; D.\mathcal{S} \cap S_c.\mathcal{R} = \emptyset$$

As follows from the instantiations of both machines, their address spaces differ only by the addresses occupied by the compiled code and data structures of the framework π_{th}. According to the shared invariant, the read only memory of the MXA machine contains this compiled code and all data structures are mentioned in the computation of the ownership state in the implementation. Therefore, the last part of $oinv(D.G)$ holds:

$$S_c.\mathcal{A} = S_c.\mathcal{R} \cup D.\mathcal{S} \cup \bigcup_{p \in \mathbb{N}_{nu}} D.O_p$$

□

We discharge Assumption 2.1 by proving the following lemma.

Lemma 8.2 (Safety Transfer and Invariants Preservation for Correctness of Concurrent Kernel Threads).

$$\forall D \in \mathbb{C}_{S_c}, d' \in \mathbb{M}_{S_c}, E, E' \in \mathbb{C}_{S_a}, \sigma \in \Theta^*_{S_c}, \tau \in \Theta^*_{S_a}, o_\tau \in \Omega^*_{S_a}, p \in \mathbb{N}_{nu}.$$

(i) $D.M \overset{\sigma}{\longmapsto} d' \wedge blk(\sigma, p) \wedge one\mathcal{IO}(\sigma, \tau) \wedge wb(D.M, \sigma) \wedge wf_p(d')$

(ii) $E \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E' \wedge blk(\tau, p) \wedge P_{S_a}(E) \wedge safe_{P_{S_a}}(E, \langle \tau, o_\tau \rangle) \wedge sc(E.M, \tau) \wedge wf_p(E'.M)$

(iii) $csim_p(D.M, E) \wedge sinv(D, E) \wedge csim_p(d', E')$

$\Longrightarrow$

$\exists o_\sigma \in (\Omega_{S_c})^*, \mathcal{G}' \in \mathbb{G}_{S_c}.$

(i) $D \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d', \mathcal{G}') \wedge safe(D, \langle \sigma, o_\sigma \rangle)$

(ii) $sinv((d', \mathcal{G}'), E')$

Proof: The safety $safe(D, \langle\sigma, o_\sigma\rangle)$ requires that the ownership invariant $oinv(D.G)$ holds, what follows directly from Lemma 8.1.

In order to prove the ownership safety during the MXA steps and the shared invariant after them, one should consider cases depending on the step of the abstract machine and the consistency point in the implementation in the way similar to the proof of Theorem 8.1.

Thread switch: During the thread switch both machines operate on local data structures and no ownership transfer is performed. Therefore, we choose the ownership transfer information o_σ containing empty sets of addresses for each step of the MXA machine.

From $ocinv_{Th}(E)$ and the shared invariant we know that all threads from $\mathrm{dom}\,(E.u_p.ready)$ and their elements in the array $threads$ are locally owned. Therefore, any access to a TCB by such a thread identifier is safe. Moreover, the index of the current thread to be rewritten resides in the memory at the addresses $A_{cur}(p)$, which are local according to the shared invariant.

Since from $consis_{Th}^{proc}$ we have $\mathrm{dom}\,(E.u_p.ready) = TIDs_{ready}^{\theta_{imp}}(D.m, p)$, we conclude that the local list of the ready threads in the implementation contains only locally owned nodes of all threads from $\mathrm{dom}\,(E.u_p.ready)$. Therefore, the function $search\_by\_tid$ traversing this list accesses local addresses. During the thread reconstruction after the execution of the inline assembly in the procedure $switch\_stack$ we access the stack information abstraction (in the memory) also instantiated by the list of these ready threads.

Moreover, any operations on the stacks of the threads to be switched are safe because the stacks of all processor's threads are allocated at the memory addresses owned and not shared according to $ocinv_{Th}(E)$.

Since the ownership state and the shared memory are not changed during the steps in both machines, the shared invariant is trivially preserved.

The proof of the lemma for the primitive $therad\_exit\_to$ is the same except for the additional argumentation about the locally owned list of finished threads.

Pure MXA step of the abstract machine: Since both machines perform the same steps, we choose o_σ with the ownership transfer applied for the abstract machine.

The safety of the memory accesses and the ownership transfer in the implementation machine then mostly follows from $P_{S_a}(E) \wedge safe_{P_{S_a}}(E, \langle\tau, o_\tau\rangle)$. The only exception is the context switch back to the current thread performed in the abstract machine. In the implementation during the reconstruction we access the stack information abstraction. Hence, the argumentation about the safety of the memory accesses is the same as in the case for the thread switch above.

The shared invariant holds after the steps because we have $csim_p(d', E')$, well-formedness $wf_p(d')$, $wf_p(E'.M)$, and the ownership states in both machines are changes in the same way.

Thread creation: During the thread creation by the unit p the ownership state of both machines is changed when we acquire and release the lock. The lemma is proven by a case split on $E.m(ap)$.

1. $E.m(ap) \notin \{0, p\}$: the lock is held by another processor.

   After reading only $E.m(ap)$ the abstract machine stays in the same configuration and the ownership state remains unchanged. In the implementation the machine either calls the primitive and executes its code until the second call of cas or performs a single iteration in the loop of the function $acquire\_lock$. Therefore, we set o_σ with empty sets for all steps of the MXA machine.

   From $ocinv_{Th}(E)$, $oinv(E.G)$, and the shared invariant we conclude that the addresses A_{lock} are shared and not owned by p. Since the MXA machine reads the memory at these addresses at an $\mathcal{IO}$-point, the memory access policy is not violated. If the primitive is called in the implementation, the expression evaluation of the arguments is also safe because the same memory addresses are accessed in both machines and $safe_{P_{S_a}}(E, \langle \tau, o_\tau \rangle)$ holds.

Moreover, the shared invariant after the steps follows from the fact that no ownership state and no shared memory are changed.

2. $E.m(ap) = 0$: the lock is free.

   In this case according to $ocinv_{Th}(E)$ we know that ap, new, $free$, all threads from $E.m(free)$, $\mathrm{dom}\,(E.m(new))$, and their stack addresses $A^{stacks}_{new}\,(Th(E.u_p, E.m))$ are shared and not owned. Not violating the ownership safety of the step, the machine for kernel threads acquires them and makes them local except for the shared and owned ap. After updating these components and setting $E.m(ap) = p$, the ownership is no more changed.

   Using $ocinv_{Th}(E)$, the shared invariant for E and D, and the simulation relation we also conclude that the memory addresses occupied by the lock, the pointers to the lists of free and new threads, the array elements corresponding to these threads as well as stacks of new threads are shared and not owned by any unit. Moreover, the lists of new and free threads contain only such array elements.

   Therefore, we choose o_σ in a way such that during the execution of cas in the function $acquire\_lock$ all theses addresses are acquired and made local except for the addresses of the lock. For all other steps the ownership transfer information has empty sets and does not change the ownership state of the MXA machine. Directly from this setting we conclude the ownership safety of the MXA steps implementing the step of the thread in this case.

   After the lock acquisition all memory accesses during the traversal of the lists of new and free threads, the accesses to the TCB of the new thread (found in the list) and its stack are safe.

   From $sinv(D, E)$, $csim_p(d', E')$, and the updates of the ownership states in both machines one easily concludes that the ownership invariant is preserved after the steps.

3. $E.m(ap) = p$: the lock is held by processor p.

   According to $ocinv_{Th}(E)$ we know that new, $free$, all threads from $E.m(free)$, new threads from $\mathrm{dom}\,(E.m(new))$, and their stack addresses $A^{stacks}_{new}\,(Th(E.u_p, E.m))$ are shared and not owned by any unit. The components ap is shared and owned by p. During the step of the abstract machine these components are released and ap is set to $E.m(ap) = 0$.

   By $ocinv_{Th}(E)$, $csim_p(D.M, E)$, and $sinv(D, E)$ one easily sees that in the implementation the memory addresses occupied by the aforementioned abstract components belong to the sets of owned and shared addresses of the MXA machine in the same way as it is mentioned above for the machine with threads.

   By executing the assignment $*lock = 0$ inside $release\_lock$ we release all these addresses. Obviously, the lock write is a safe memory access because it is owned by unit p. The ownership transfer information o_σ contains the addresses to be released by the step performing $*lock = 0$ and empty sets for all other MXA steps. Therefore, the ownership transfer policy is not violated.

   One can easily show that $sinv((d', \mathcal{G}'), E')$ holds by $sinv(D, E)$, $csim_p(d', E')$, the updates of the ownership states, and well-formedness $wf_p(d')$, $wf_p(E'.M)$ for unit p after its steps.

$\square$

The proof of Assumption 2.4 is shown in Lemma 8.3.

Lemma 8.3 (Preservation of Simulation Relation for Concurrent Kernel Therads).

$$\begin{array}{rl} \multicolumn{2}{l}{\forall D, D' \in \mathbb{C}_{S_c}, E, E' \in \mathbb{C}_{S_a}, p \in \mathbb{N}_{nu}.} \\ (i) & csim_p(D.M, E) \\ (ii) & sinv(D, E) \wedge P_{S_a}(E) \wedge oinv(E) \wedge oinv(D) \\ (iii) & sinv(D', E') \wedge P_{S_a}(E') \wedge oinv(E') \wedge oinv(D') \\ (iv) & E \approx_p E' \wedge D \approx_p D' \\ & \implies csim_p(D'.M, E') \end{array}$$

Proof: First, we consider the preservation of parts of $consis_{Th}$ not depending on the value of the lock. From $E \approx_p E'$ and $D \approx_p D'$ we know that in both machines the configurations of unit p, the memory owned by p, and its sets of owned addresses are equal. Since $csim_p(D.M, E)$ holds and the consistency for the shared memory as well as $E'.m(ap)$ is covered by $sinv(D', E')$ we conclude that for D' and E' the simulation relations $consis_{Th}^{proc}$,$consis_{Th}^{mem}$, and the equality for the lock are preserved.

In order to prove $consis_{Th}^{free/new}$ between D' and E', we make a case split on values of $E.m(ap)$ and $E'.m(ap)$.

1. $E.m(ap) = 0 \wedge E'.m(ap) \notin \{0, p\}$: The components *free* and new are locked by another unit $q \neq p$. Therefore, the unit p does not cover them in its simulation relation and $consis_{Th}^{free/new}$ holds by definition (the implication in the relation is true).

2. $E.m(ap) = 0 \wedge E'.m(ap) = 0$: The components *free*, new are not locked in E'. By $ocinv_{Th}(E)$ they belong to the shared memory of the *Cosmos* machine and are covered by the shared invariant in the same way stated in $consis_{Th}^{free/new}$.

3. $E.m(ap) = 0 \wedge E'.m(ap) = p$: By $ocinv_{Th}(E)$ and $ocinv_{Th}(E')$ we obtain $ap \notin E.\mathcal{O}_p$ and $ap \in E'.\mathcal{O}_p$. Since the owned addresses of p are changed, it contradicts the premise $E \approx_p E'$ requiring the equality of the owned addresses between E and E'. Therefore, the preservation of the simulation relation does not have to be proven.

4. $E.m(ap) = p \wedge E'.m(ap) \notin \{0, p\}$: By $ocinv_{Th}(E)$, $ocinv_{Th}(E')$, and $oinv(E')$ we conclude $ap \in E.\mathcal{O}_p$ and $ap \notin E'.\mathcal{O}_p$ and become a contradiction as in the previous case.

5. $E.m(ap) = p \wedge E'.m(ap) = 0$: Again, in this case we get a contradiction to $E \approx_p E'$, because the owned sets with $ap \in E.\mathcal{O}_p$ and $ap \notin E'.\mathcal{O}_p$ are not equal.

6. $E.m(ap) = p \wedge E'.m(ap) = p$: In both configurations E, E' the components ap, new, *free*, the identifiers of new and free threads, and addresses occupied by their stacks are owned by p. The same holds for the corresponding addresses in the implementation machine. By $E.m|_{E.\mathcal{O}_p} = E'.m|_{E.\mathcal{O}_p}$ we conclude $E.m(free) = E'.m(free)$ and $E.m(new) = E'.m(new)$. Moreover, we also have $D.m|_{E.\mathcal{O}_p} = D'.m|_{E.\mathcal{O}_p}$. Since $consis_{Th}^{free/new}$ holds between E and D, it is also preserved in E' and D'.

7. $E.m(ap) \notin \{0, p\} \wedge E'.m(ap) \notin \{0, p\}$: The unit p does not to cover the new and free thread in its simulation relation (the implication in $consis_{Th}^{free/new}$ is true).

8. $E.m(ap) \notin \{0, p\} \wedge E'.m(ap) = 0$: In this case, ap, *free*, new are shared in E' and the relation $consis_{Th}^{free/new}$ trivially follows from $sinv(D', E')$.

9. $E.m(ap) \notin \{0, p\} \wedge E'.m(ap) = p$: The component ap is not owned by p in E. However, it is owned in E'. This contradicts again the premise $E \approx_p E'$.

□

For the proof of Assumptions 2.2 and 2.3 about the well-formedness preservation in both machines we now consider Lemma 8.4 and Lemma 8.5.

Lemma 8.4 (Preservation of Well-Formedness for Other Units in Machine with Kernel Threads).

$$\begin{aligned} &\forall E, E' \in \mathbb{C}_{S_a}, p \in \mathbb{N}_{nu}.\\ &\quad \textit{(i)} \quad \mathit{wf}_p(E.M) \wedge E \approx_p E'\\ &\quad \textit{(ii)} \quad \mathit{sinv}(D', E')\\ &\quad \textit{(iii)} \quad \mathit{oinv}(E) \wedge P_{S_a}(E) \wedge \mathit{oinv}(E') \wedge P_{S_a}(E')\\ &\qquad \Longrightarrow \mathit{wf}_p(E'.M) \end{aligned}$$

<u>Proof</u>: Since the configuration of unit p in E and E' are equal by $E \approx_p E'$, we conclude that $\mathit{wfth}^{\pi,\theta}_{proc}$ and $\mathit{wfconf}^{\pi,\theta,cba}_{\text{MXA}}$ in well-formedness $\mathit{wfconf}^{\pi,\theta,cba}_{Th}$ are preserved. For the rest depending on the value of $E'.m(ap)$ we consider the same case split from Lemma 8.3:

1. $E'.m(ap) \notin \{0, p\}$: the rest of well-formedness holds by definition because it not covered if the components ap, new, free are locked by some other unit $q \neq p$.

2. $E.m(ap) = 0 \wedge E'.m(ap) = 0$: From the shared invariant we get

$$\forall t \in \operatorname{dom}(E'.m(new))\,.\, \mathit{wfth}^{\pi,\theta}_{new}(E'.m(new, t))$$
$$\operatorname{dom}(E'.m(new)) \cap E'.m(\mathit{free}) = \emptyset$$
$$\mathit{valid}^{stack}_{Th}(E'.m(new))$$

   By $\mathit{ocinv}_{Th}(E')$ we know that the identifiers of the free and new threads as well as their stack addresses are shared and unowned. Moreover, IDs of ready, finished threads, and their stacks are locally owned by p. Therefore, we get that the sets $\operatorname{dom}(E'.m(new))$, $E'.m(\mathit{free})$, and $\operatorname{dom}(E'.u_p.\mathit{ready}) \cup \operatorname{dom}(E'.u_p.\mathit{fin})$ do not intersect pairwise.

   Since the unit configuration is equal in E, E' and by $\mathit{wf}_p(E.M)$ the sets of the ready and finished threads are disjoint, we conclude that the sets of new, free, ready, and finished threads in E' are disjoint too. By the same argumentation for the stack addresses one shows that the stacks of the ready, finished and new threads do not overlap.

3. $E.m(ap) = p \wedge E'.m(ap) = p$: In both configurations E, E' the components ap, new, free, the identifiers of the new and free threads and addresses occupied by their stacks are owned by p. By $E.m|_{E.\mathcal{O}_p} = E'.m|_{E.\mathcal{O}_p}$ we conclude $E.m(\mathit{free}) = E'.m(\mathit{free})$ and $E.m(new) = E'.m(new)$. The same equality holds also for the finished and ready threads. Therefore, $\mathit{wf}_p(E'.M)$ follows directly from $\mathit{wf}_p(E.M)$.

4. $E.m(ap) \notin \{0, p\} \wedge E'.m(ap) = 0$: The proof is absolutely the same as for the case with $E.m(ap) = 0 \wedge E'.m(ap) = 0$.

As shown in Lemma 8.3, all other cases contradict $E \approx_p E'$.

□

Lemma 8.5 (Preservation of Well-Formedness for Other Units in Machine Implementing Kernel Threads).

$$\begin{aligned}
&\forall D, D' \in \mathbb{C}_{S_c}, E, E' \in \mathbb{C}_{S_c}, p \in \mathbb{N}_{nu}.\\
&\quad \textit{(i)} \quad \mathit{wf}_p(D.M) \wedge D \approx_p D'\\
&\quad \textit{(ii)} \quad \mathit{csim}_p(D.M, E) \wedge \mathit{sinv}(D, E) \wedge \mathit{sinv}(D', E')\\
&\quad \textit{(iii)} \quad \mathit{oinv}(E) \wedge P_{S_a}(E) \wedge \mathit{oinv}(E') \wedge P_{S_a}(E')\\
&\quad \Longrightarrow \mathit{wf}_p(D'.M)
\end{aligned}$$

<u>Proof</u>: From $\mathit{ocinv}_{Th}(E)$ and the shared invariant $\mathit{sinv}(D, E)$ we know that all threads from the domains $\operatorname{dom}(E.u_p.\mathit{ready}) \cup \operatorname{dom}(E.u_p.\mathit{fin})$, their stack addresses, and their elements in the array *threads* are locally owned. Since by $\mathit{consis}^{proc}_{Th}$ we have $\operatorname{dom}(E.u_p.\mathit{ready}) = \mathit{TIDs}^{\theta_{imp}}_{ready}(D.m, p)$ and $\operatorname{dom}(E.u_p.\mathit{fin}) = \mathit{TIDs}^{\theta_{imp}}_{\mathit{fin}}(D.m, p)$, we conclude that the local lists of ready and finished threads in the implementation contain only locally owned nodes of all threads from the aforementioned domains. Moreover, according to $\mathit{sinv}(D, E)$ the addresses $A_{cur}(p) \cup A_{ready}(p) \cup A_{\mathit{fin}}(p)$ are also local.

Therefore, using the equality of unit configurations for p, its owned addresses, owned memory assumed by $D \approx_p D'$, and $\mathit{wf}^{proc}_{\text{MXA}}$ in D, we easily conclude that $\mathit{wf}^{proc}_{\text{MXA}}$ for p in D' is preserved.

The proof of $\mathit{wf}^{free/new}_{\text{MXA}}$ in D' depends on $ap^{\theta_{imp}}(D'.m)$, $ap^{\theta_{imp}}(D.m)$ equal to $E'.m(ap)$ and $E.m(ap)$ respectively by $\mathit{sinv}(D', E')$ and $\mathit{sinv}(D, E)$. Hence, we make a case split from the proof of Lemma 8.3:

1. $ap^{\theta_{imp}}(D'.m) \notin \{0, p\}$: Well-formedness $\mathit{wf}^{free/new}_{\text{MXA}}$ trivially holds by definition.
2. $ap^{\theta_{imp}}(D'.m) = 0$: Well-formedness $\mathit{wf}^{free/new}_{\text{MXA}}$ is fully covered by $\mathit{sinv}(D', E')$.
3. $ap^{\theta_{imp}}(D.m) = p \wedge ap^{\theta_{imp}}(D'.m) = p$: Analogously to the proof for ready and finished threads, we use $\mathit{ocinv}_{Th}(E)$, $\mathit{sinv}(D, E)$, and $\mathit{consis}^{free/new}_{Th}$ in order to show that the corresponding lists in the implementation contain only addresses locally owned by p. Moreover, the stacks of new threads as well as the addresses $A_{new} \cup A_{\mathit{free}}$ are owned by p too.

   Therefore, using $D \approx_p D'$ and $\mathit{wf}^{free/new}_{\text{MXA}}$ in D for p, we conclude that for unit p well-formedness of the MXA machine for new and free threads is preserved in D'.

As follows from premise (iii), all other cases contradict $D \approx_p D'$ because the owned addresses of p in D and D' are not equal. For instance, in case of $ap^{\theta_{imp}}(D.m) = p \wedge ap^{\theta_{imp}}(D'.m) = 0$ using $\mathit{ocinv}_{Th}(E)$ and $\mathit{ocinv}_{Th}(E')$ we conclude that ap, $free$, new are owned by p in E and not owned in E. By the shared invariant in D and D' the same holds for the corresponding address occupied by these components in the implementation. Therefore, we conclude $D.\mathcal{O}_p \neq D'.\mathcal{O}_p$.

This finishes the proof of the well-formedness transfer in the MXA machine implementing kernel threads.

□

Having discharged the assumptions for all instantiations one can state that *Cosmos* model simulation theorem holds for each case.

Theorem 8.3 (***Cosmos*** **Model Simulation Theorem for all Programs of Hypervisor / OS Kernels Using Threads).** *Theorem 2.4 holds for any kernel programs π with inline assembly, the environment parameters $\theta, \theta_{imp} \in Params_{\mathrm{CIL}}$, and the system information $c\iota \equiv (cba, StIbas)$ used for instantiation of the models $S_{Th}^{\pi,\theta,\theta_{imp},cba}$, $S_{imp}^{\pi_{imp},\theta_{imp},c\iota} \in \mathbb{S}$ wrt. the framework π_{th}.*

Finally, in order to justify the concurrent model of kernel threads executed on the multi-core MIPS-86 machine, one has to prove additionally the following:

- the safety and properties transfer from incomplete consistency blocks to the arbitrary schedules of $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}$ in the way done in the previous chapters,
- a simple simulation guaranteeing that for any steps of $S_{\mathrm{MXA}}^{\pi_{imp},\theta_{imp},c\iota}$ there exist the same steps of $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}$ such that the machine configurations are coupled and the interleaving points of $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}$ are in the set of interleaving points of the underlying machine,
- the transfer of the safety and all other properties from $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}$ to the complete consistency block schedules of $S_{\mathrm{MXA}}^{\pi_{imp},\theta_{imp},c\iota}$ required by the concurrent simulation theorem from Chapter 7.

Then, applying Theorem 7.3 we get complete consistency block schedules of $S_{\mathrm{MXA}}^{\pi_{imp},\theta_{imp},c\iota}$ for which we prove the existence of steps of the *Cosmos* machine for concurrent kernel threads in the following way:

- transform a block schedule to a corresponding $\mathcal{IP}$-schedule,
- switch to the corresponding schedule of $S_{imp}^{\pi_{imp},\theta_{imp},c\iota}$ via the simulation introduced above,
- perform the order reduction so that one gets the consistency blocks suitable for the concurrent threads simulation[6],
- apply Theorem 8.3 for the reordered sequence.

We leave this technical tasks for an interested reader and do not consider them formally in this thesis.

[6]Recall that this is the second application of the order reduction in our model stack. The first one was made on the steps of the SB reduced MIPS-86 for obtaining the consistency blocks implementing steps of the MXA machine.

9 Conclusion and Future Work

To the best of our knowledge, the results provided in this doctoral thesis contribute to a number of important research topics concerning the formal verification of hypervisors and operating systems.

- The work represents progress towards modeling and verification of multi-threaded kernels of hypervisors and OS kernels as well as formalization of C semantics wrt. the C11 standard dealing with threads. Note that the Verisoft [Ver10] and VerisoftXT [The11] projects considered the introduction of threads as a topic for future work (see Chapter 7 in [Dö10]). Moreover, Kovalev assuming in [Kov13] the multi-threaded hypervisor kernel implementation, claimed that "probably the most complicated part of the kernel layer verification is the proof of a thread switch mechanism".

- In the thesis, we give the first argumentation about the correctness of threads implemented by the mixture of high- and low-level programming languages with an optimizing to some degree compiler. In contrast to existing work [FS05, FSV$^+$06, FSDG08, NYS07, GFSS12], where the thread switch is similar to the normal context switch saving/restoring the content of the whole register file, having stated the compiler correctness and relying on it we managed to implement the correct switch with much less operations required from the system programmer.

- In the attempt to formalize the multi-threaded programming model and to find a clean implementation of threads simple enough for reasoning about their formal correctness, we discovered that classical *forking* typical for Lunix kernel programming has an unnecessarily complicated semantics and operates rather on processes than threads. As a result, we came up with a simpler implementation that turned to be similar to the threads given by the POSIX standard. Based on that and using the achievements of the Verisoft project concerning the CVM model, where special functions on the model are formalized as primitives, we managed to provide a clean concurrent semantics of kernel threads revealing hardware features (except for external interrupts) needed for system programming. The integration of inter-processor interrupts into the concurrent pure C-IL semantics can be found in [Pen16]. Moreover, the C-dialect C0 with inline assembly for systems containing disks and allowing external interrupts is considered in [PBLS15].

- In order to argue about the correctness of our threads implementation, we developed the extended concurrent C-IL+MASM+Asm semantics that allows to describe any stack substitution not possible in [Sha12] and, in fact, is powerful enough for the implementation of hypervisor/OS kernels, however, restricted to a case without device steps and interrupts appearing during C-IL steps.

- In comparison to [Kov13, Pen16] where the authors abstracted away from some important technical issues, our work contains a complete model stack where *all* layers are connected by the same general concurrent simulation theory, the compiler correctness is fully exposed, and all (hopefully) needed well-formedness requirements, software conditions, and safety properties are discovered.

Among the directions for future work one can point to:

- the formal mechanized proof of theorems stated in the thesis with the full check of properties transfer and auxiliary technical, though simple, simulations (e.g., between ${S'}^{\pi,\theta,\xi}_{r\mathrm{MIPS}}$ and $S_{r\mathrm{MIPS}}$ in Section 6.2.4) only sketched in the scope of the work,
- the implementation and verification of the kernel memory manager responsible for the memory allocation,
- the implementation of the threads manager based on the introduced model of kernel threads,
- the integration of interrupts and preemption in the thread scheduling via combination of the results from [Pen16] and [FSDG08, GFSS12] achieved by the FLINT group,
- the creation of initial threads in the kernel for each processor as well inside any user process. Talking about the initialization of the kernel on the multi-core machine, the initial stacks are usually allocated for each core during the booting process and then associated with such boot-strap threads. This association can be performed by the implementation of a function preparing a TCB corresponding to a calling thread.
- relaxing the restrictions and limitations of the *Cosmos* model and order reduction applied as the main theory in the scope of our work. For instance, one of such strong restrictions is the fixed set of read-only memory addresses excluding the self modification of the code. Another one is the dependence of the $\mathcal{IO}$- and $\mathcal{IP}$-points only on this memory. The discussion on the solutions for these and other issues of the *Cosmos* model can be found in [Bau14b]. The development of a more flexible order reduction is in progress at the chair of Prof. Paul in Saarland University.
- the application of Oberhauser's [Obe16] effective programming discipline for the store buffer reduction and substituting our safety policy by the new required conditions.
- Last but not least, one has to mention that the introduction of the multi-threaded semantics opens a new direction for the work on the CVM model for the multi-core machine. Even though one could implement CVM with a number of kernel threads equal to the number of processor cores (what probably does not need the full model from Chapter 8), each user process must be represented by a few virtual processors instead of a single one considered in [PBLS15] because the execution of a user process can now be split into concurrent threads. Obviously, having such a model of the user process with concurrent virtual processors, one can easily apply our semantics of threads for the verification of user programs.

Appendix A: Implementation of Spinlocks with Processor ID

Listing 9.1: Getting processor ID.

```
get_pid USES sv1 sv2 sv3 sv4 sv5 sv6 sv7 sv8
1: movs2g pid rv
2: ret
```

Listing 9.2: Spinlock type.

```
typedef volatile i32 lock_t;
```

Listing 9.3: Spinlock initialization.

```
void init_lock(lock_t *lock)
{
1: *lock = 0;
2: return;
}
```

Listing 9.4: Acquisition of spinlock

```
void acquire_lock(lock_t *lock)
{
u32 pid;
i32 acquired;
u32 i;
1: i = 1; // the first cas will be executed
2: pid = call get_pid();
3: call cas(lock, 0, (i32)pid, &acquired);
4: i = 0; // the first cas is executed
5: ifnot (acquired == 0) goto 3;
6: return;
}
```

Listing 9.5: Release of spinlock

```
void release_lock(lock_t *lock)
{
1: *lock = 0;
2: return;
}
```

Appendix B: Implementation of Operations on Doubly Linked Lists of Threads

Listing 9.6: Search for a list node of a thread with a given ID.

```
thread_t *search_by_tid(thread_t *hd, u32 tid)
{
1: ifnot (hd != 0 && (hd->tcb).tid != tid) goto 4;
2: hd = hd->next;
3: goto while;
4: return hd;
}
```

Listing 9.7: Search for a list node of a thread with a given processor ID.

```
thread_t *search_by_pid(thread_t *hd, u32 pid)
{
1: ifnot (hd != 0 && (hd->tcb).pid != pid) goto 4;
2: hd = hd->next;
3: goto while;
4: return hd;
}
```

Listing 9.8: Removing a thread list node.

```
void remove(thread_t **hd, thread_t **tl, thread_t *th)
{
1: ifnot (th->prev = 0) goto 4;
2: *hd = th->next;
3: goto 5;
4: (th->prev)->next = th->next;
5: ifnot (th->next = 0) goto 7;
5: *tl = th->prev;
6: goto 8;
7: (th->next)->prev = th->prev;
8: return;
}
```

Listing 9.9: Inserting a node to the end of a thread list.

```
void insert_to_end(thread_t **hd, thread_t **tl, thread_t *th)
{
1: ifnot (*tl == 0) goto 6;
2: *tl = th;
3: *hd = th;
4: th->prev = 0;
5: th->nex = 0;
5: goto 10;
6: th->next = (*tl)->next;
7: th->prev = *tl;
8: (*tl)->next = th;
9: *tl = th;
10: return;
}
```

Appendix C: Long List of "Small" Mistakes in PhD Dissertations / Books

Shadrin's PhD Thesis [Sha12]

1. *Definition of the inactive C-IL context of the MX machine* (Definition 5.5 on page 71):

   The callee-save registers in the inactive C-IL context are not needed for the operational semantics of the MX machine.

2. *Computation of the C-IL configuration from the MX machine* (Definition 5.7 on page 73, Pure C-IL step on page 74):

   The C-IL configuration obtained from the MX machine does not take into account the frames of inactive contexts. The proper numbering of the C-IL frames in the overall stack of the mixed machine is required because the frame index is included into the local reference value.

3. *Run-time errors during pure C-IL and MASM steps of the MX machine* (pages 74, 85):

   The run-time errors during pure C-IL and MASM steps of the MX machine are not detected and not shown in the MX semantics though they are present in C-IL and MASM semantics separately. Moreover, in the MX compiler correctness, Shadrin requires "the execution of c_{MX} does not get stuck" in the theorem, what is not defined by the MX semantics.

4. *Definition of an inter-language step* (pages 73-74):

   The predicate $ext(c_{\mathrm{MX}}, \pi)$ indicating the inter-language step is not formally defined. Then, in the definition of the pure C-IL and MASM steps it is used with a stack as a parameter instead of the MX configuration.

5. *Writing the return value during the return from MASM to C-IL* (page 75):

   The function $write^{\theta}$ is used with a bit-string argument $gpr[rv]$ though it is defined only on typed C-IL values.

6. *Compiler correctness for C-IL and MX* (pages 67, 85)

   No software conditions, under which the claims of compiler correctness for C-IL and MX should hold, are stated (in contrast to MASM correctness given in a different form on page 46). The formulations of theorems are partially informal. A claim about the existence of the next consistency point is missing.

7. *C-IL and MX return address consistency* (Definition 4.17, 5.17 on pages 63, 81)

   Shadrin claims that a memory word at the return address taken from the stack is equal to the starting address of the compiled code of the C-IL function call. This statement has no sense. The correct one must be: the return address taken from the stack is equal to the starting address of the epilogue. The fact about the epilogue was discovered by Alekhin and later fixed by Baumann in his thesis [Bau14b].

8. *C-IL stack consistency* (Definition 4.22 on page 65)

   No consistency for parameters and local variables of non-topmost frames is given.

9. *MX compiler consistency* (page 81)

   Callee-save registers present in inactive contexts of the MX machine are not coupled with the implementation.

Schmaltz's PhD Thesis [Sch13]

1. *Computation of misalignment interrupt* (page 68)

   Misalignments on fetch and load/store $iev(c, I, pff, pfls)[3]$ correspond to the same interrupt level and are computed together without taking into account the page fault on fetch having a lower interrupt level. Therefore, misalignment on load/store can be generated even in case of page fault on fetch, what, in turn, provides a wrong cause of interrupts.

2. *Field reference function for C-IL* (Definition 5.35 on page 123):

   The function $\sigma_\theta^\pi(x, f)$ is defined only for values x of pointer types. Therefore, the C-IL semantics does not allow to access a field in an array of structures though such arrays are pretty common in the programming practice.

3. *C-IL transition function for goto and if-not-goto* (Definition 5.41 on page 127)

   The C-IL transition function is defined for any statements **ifnot** e **goto** l and **goto** l though the label l must be only in the body of the function. This condition is missing.

4. *Return from C-IL function* (Definition 5.41 on page 127)

   The semantics of the C-IL return step does not have a condition requiring that the stack contains more than one stack frame.

Baumann's PhD Thesis [Bau14b]

1. *Sequential simulation relation in concurrent settings* (Definition 79 on page 155)

   The simulation relation $sim : \mathbb{L}_c \times \mathbb{L}_a \times \mathcal{P} \to \mathbb{B}$ of the sequential simulation framework does not take into account sets of addresses of possible inconsistent regions of memory modified by concurrent steps of other units. Therefore, it is not possible to state the correct sequential simulation theorem needed for the concurrent simulation.

2. *Requirement on $\mathcal{IO}$-points in consistency blocks* (page 158, already reported in [Bau14a])

   The definition $one\mathcal{IO}(\sigma, \tau)$ does not exclude the case with $\sigma|_{io} \neq \varepsilon \wedge \tau|_{io} = \varepsilon$ and, therefore, is incorrect.

3. *Generalized sequential simulation theorem* (Theorem 5 on page 159)

   The generalized sequential simulation theorem does not consider environment steps and, hence, is not suitable for the concurrent simulation.

4. *Concurrent simulation relation* (page 162, Theorem 6 on page 166)

   The simulation relation $sim(d, par, e) = \forall p \in U_c(d, par).\ sim_p(d, par, e)$ for units at consistency points is only based on machine configurations without the ownership state of the abstract machine and, therefore, for any unit p it does not exclude memory addresses locally owned by other units $q \neq p$. The reason is the absence of inconsistent memory as a parameter in the sequential simulation relation. When not every unit reaches a consistency point (how it is considered in the Cosmos model simulation theorem), it is not possible to show this simulation relation holds. In the Cosmos model simulation theorem it is also claimed to hold for any existing executions of the abstract machine, though it is only possible for the safe ones.

5. *Safe abstract schedules in Cosmos model simulation theorem* (Theorem 6 on page 166)

   Ownership safety transfer and shared invariant are claimed to hold for any safe abstract schedules. The additional safety property P_a of the abstract machine is not considered in these schedules though it is needed for the concurrent simulation.

6. *Coupling of units' indices in Cosmos model simulation theorem* (Theorem 6 on page 166)

   The theorem does not guarantee that a consistency block in the implementation and specification belongs to the unit with the same index.

7. *Assumption 1* (page 164)

   Safety of computations in the consistency block does not respect the property P_a needed for the simulation. The concurrent simulation relation does not take the ownership as a parameter. Moreover, well-formedness for both machines at the end of their consistency block computations is not present among the premises in the assumption though it is needed for proving the shared invariant.

8. *Assumption 2* (page 165)

   The ownership invariant (see Definition 12) for machine states is not present. The sequential simulation relation is used without the ownership state. In the assumption for the abstract machine, P_a and $oinv$ allowing to derive needed properties about owned and shared components and the memory in configurations E and E', are not present. The same holds for the assumption for the concrete machine, where one additionally needs the simulation relation and the shared invariant for E and D in order to transfer P_a from the abstract machine to the concrete level.

9. *Transfer of a few properties* (Theorem 8, page 182)

   The conjunction of predicates in $safety(D, Q[P, par] \wedge W, suit)$ is not a standard notation and it is not defined in the thesis.

10. *Transfer of well-behavior from block schedules to arbitrary interleaved schedules* (pages 175, 182)

    The computation of the well-behaviour flag $d'.u(p).wb$ is shown incorrectly because it cannot be based on the full memory. Instead, one can only use the memory covered by the $reads$-set. No such restriction on the instantiation of the well-behaviour predicate is stated.

11. *Offset of local variables in C-IL compiler information* (page 128)

    The C-IL compiler information function $info_{IL}.off_{lvar}$ computing the offset of a given local variable on the stack relies on the location in the body of a given function. However, this offset is static after the compilation because the allocation of the local variables is performed in the prologue part of the callee and cannot be changed during the function execution. Therefore, the location as a parameter for $info_{IL}.off_{lvar}$ is redundant. This parameter is not used in [Sha12] and was introduced in [Bau14b].

12. *Computation of the distance between frame base addresses in C-IL stack* (page 143)

    In the computation $dist(i)$ for a non-topmost frame i the size of the caller-save region is $size_{CrS}(f_i, loc_i)$ where loc_i points to a statement after the call. However, this size must be calculated at the location $loc_i - 1$ of the function call.

13. *C-IL local variable consistency* (Definition 70 on page 134, already reported in [Bau14a])

    In the computation of $crsa_{i,j}$ the author uses $info_{IL}$ instead of $info_{IL}.crso$. Moreover, $csrbase_i$ must be $crsbase_i$.

14. *C-IL local variable consistency* (Definition 70 on page 134, already reported in [Bau14a])

    Local variables of a stack frame i are in callee-save registers saved in the same frame, i.e. $\mathcal{M}_{\mathcal{E}}{}_i(v_{i,j}) = h.m_4(csa_{i,j})$ is claimed. Instead, one has to consider the values $h.m_4(csa_{i+1,j})$ in the next frame.

15. *Return value destination in C-IL stack consistency* (Definition 71 on page 135)

    The fact that the return value can be stored into a variable used as a parameters in the function was not taken into account.

16. *Volatiles in expressions* (Definition 64 on page 130)

    In the ternary operator $e\ ?\ e_1 : e_2$ the author tests all the expressions e, e_1, and e_2 together on the presence of the volatile accesses. However, for instance, if e and e_1 do not contain volatile accesses, while e_2 does, and during a step the condition e is evaluated to one, the volatile read in e_2 would be performed neither in the C-IL nor in the compiled code, though the whole expression with the ternary operator would contain a volatile access according to [Bau14b]. Therefore, matching $\mathcal{IO}$-points between the abstract and concrete machines would become a problem.

17. *Stack regions in the sequential simulation framework* (page 190)

    Though every unit of the concurrent C-IL machine should exclude all stacks (its own and others belonging to all other units) from its memory consistency, only a single stack region is treated by every units. The reason is that the stack base address and the stack maximal size of a single stack are in the C-IL compiler information which is used by all units in their simulation relations.

18. *Ownership for C-IL and MASM stacks in shared invariants* (Definitions 86, 87 on pages 187, 192)

    An individual stack region (in the memory of the multi-core MIPS-86 machine) of a unit is owned by this unit. However, since owned addresses may be shared, additionally, all stack addresses should be excluded from the shared addresses.

System Architecture as an Ordinary Engineering Discipline [PBLS15]

1. *Returning to translated C in the inline assembly semantics* (Section 13.3.3, page 327)

   Though we may reach a consistency point, it is not always possible to return to the abstract C0 configuration and one needs to continue the ISA execution until one reaches another consistency point at which the reconstruction is possible. This issue and the solution were discovered by Alekhin in the time when the original version of C0 semantics with inline assembly [PBLS15] based on the reconstruction did not take this fact into account and, therefore, had to be corrected later in [PBLS16].

Bibliography

[Adv11a] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.19 edition, September 2011.

[Adv11b] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 3.16 edition, September 2011.

[AHPP10] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 40–54, Edinburgh, UK, 2010. Springer.

[App11] Andrew Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2011.

[APST10] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, 2010. Springer.

[Bau14a] Christoph Baumann. Known errata in the dissertation. `http://www-wjp.cs.uni-saarland.de/publikationen/Ba14err.pdf`, 2014.

[Bau14b] Christoph Baumann. *Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems*. PhD thesis, Saarland University, Saarbrücken, 2014.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[BJK$^+$06] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[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.

[CDH$^+$09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Invited paper.

[Che16] Geng Chen. *Store Buffer Reduction Theorem and Application*. PhD thesis, Saarland University, Saarbrücken, 2016.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin Heidelberg, 1998.

[CMST10] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.

[CPS13] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of multi core hypervisor verification. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, volume 7741 of *Lecture Notes in Computer Science*, pages 1–27. Springer Berlin Heidelberg, 2013.

[CS10] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In Matt Kaufmann, Lawrence Paulson, and Michael Norrish, editors, *Interactive Theorem Proving (ITP 2010)*, Lecture Notes in Computer Science, Edinburgh, UK, 2010. Springer.

[Dö10] Jan Dörrenbächer. *Formal Specification and Verification of a Microkernel*. PhD thesis, Saarland University, Saarbrücken, 2010.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Saarbrücken, 2011.

[FS05] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. *SIGPLAN Not.*, 40(9):254–267, September 2005.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. *SIGPLAN Not.*, 43(6):170–182, June 2008.

[FSV$^+$06] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. *SIGPLAN Not.*, 41(6):401–414, June 2006.

[GFSS12] Yu Guo, Xinyu Feng, Zhong Shao, and Peizhi Shi. Modular verification of concurrent thread management. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 315–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[GHLP05] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *Theorem Proving in High Order Logics (TPHOLs) 2005*, LNCS. Springer, 2005.

[Han96] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[HL09] M. Hillebrand and D. 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.

[IdR09] Thomas In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Saarbrücken, 2009.

[IdRT08] T. In der Rieden and A. Tsyban. Cvm - a verified framework for microkernel programmers. In G. Klein R. Huuck and B. Schlich, editors, *3rd intl Workshop on Systems Software Verification (SSV08)*, volume 217 of *ENTCS*, pages 151–168. Elsevier Science B.V., 2008.

[Inc06] The Santa Cruz Operation Inc. System V Application Binary Interface – MIPS RISC Processor Supplement 3rd Edition. Technical report, SCO Inc., February 2006.

[ISO11] ISO/IEC 9899:201x (Draft N1570): Programming languages — C. Standard, ISO/IEC, April 2011.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[KMP14] Mikhail Kovalev, Silvia M. Müller, and Wolfgang J. Paul. *A Pipelined Multi-core MIPS Machine - Hardware Implementation and Correctness Proof*, volume 9000 of *Lecture Notes in Computer Science*. Springer, 2014.

[Kov13] Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, 2013.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[Lei08] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[LPP05] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 2005.

[LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical report, SRC Research Report 44, 1989.

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer. Invited paper.

[Mic15] Microsoft Corp. MSDN Library: Development Tools and Languages. x64 Software Conventions. `https://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx`, October 2015.

[Mic16] Microsoft Corp. VCC: A C Verifier. `http://vcc.codeplex.com`, 2016.

[Moo03] J.S. Moore. A grand challenge proposal for formal methods: A verified stack. In *Formal Methods at the Crossroads. From Panacea to Foundational Support.*, volume 2757 of *LNCS*, pages 161–172, Heidelberg, 2003. Springer.

[MPR07] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 218–232, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007. Proceedings*, pages 189–206, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Obe16] Jonas Oberhauser. A simpler reduction theorem for x86-TSO. In Arie Gurfinkel and A. Sanjit Seshia, editors, *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, pages 142–164, Cham, 2016. Springer International Publishing.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer Berlin / Heidelberg, 2010.

[PBLS15] Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. System Architecture as an Ordinary Engineering Discipline. `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/info2/ss15/protected_material/sysbook.pdf`, 2015.

[PBLS16] Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. *System Architecture. An Ordinary Engineering Discipline.* Springer, 2016.

[Pen16] Hristo Pentchev. *Sound Semantics of a High-Level Language with Interprocessor Interrupts*. PhD thesis, Saarland University, Saarbrücken, 2016.

[POS95] IEEE POSIX 1003.1c-1995. Standard, IEEE Computer Society, 1995.

[Sch13] Sabine Schmaltz. *Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C*. PhD thesis, Saarland University, Saarbrücken, 2013.

[SF10] Zhong Shao and Bryan Ford. Advanced Development of Certified OS Kernels, 2010.

[Sha12] Andrey Shadrin. *Mixed Low- and High Level Programming Languages Semantics. Automated Verification of a Small Hypervisor: Putting It All Together.* PhD thesis, Saarland University, Saarbrücken, 2012.

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, Philadelphia, USA, 2012. Springer Berlin / Heidelberg.

[The11] The Verisoft XT Consortium. The Verisoft XT Project. `http://www.verisoftxt.de`, 2011.

[TMu03] White Paper: Processor Affinity. Multiple CPU Scheduling. TMurgent Technologies, November 2003.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken, 2009.

[Ver10] Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`, 2010.