TLB Virtualization in the Context of Hypervisor Verification

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Mikhail Kovalev

kovalev@wjpserver.cs.uni-saarland.de

Saarbrücken, März 2013

Tag des Kolloquiums:	27. März 2013
Dekan:	Prof. Dr. Mark Groves
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Kurt Mehlhorn
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	A/Prof. Dr. Gerwin Klein
3. Berichterstatter:	PD Dr. Thomas Santen
Akademischer Mitarbeiter:	Dr. Christian Schmaltz

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im März 2013

Acknowledgments

First and foremost I express my gratitude to Prof. Wolfgang J. Paul for giving me an opportunity to join possibly the world's leading research team in the field of formal verification, for his encouragement and guidance during my work.

I would like to thank many people from the Verisoft XT group and from the chair of Prof. Paul for their valuable advices, helpful discussions, and friendly atmosphere. Special thanks to Mark Hillebrant, Sabine Schmaltz, Christoph Baumann, Eyad Alkassar, and Ernie Cohen.

Abstract

In this thesis we address the challenges of hypervisor verification for multicore processors. As a first contribution we unite different pieces of hypervisor verification theory into a single theory comprising the stack of highly nontrivial computational models used. We consider multicore hypervisors for x86-64 architecture written in C. To make code verification in a C verifier possible, we define a reduced hardware model and show that under certain safety conditions it simulates the full model. We introduce an extension of the C semantics, which takes into consideration possible MMU and guest interaction with the memory of a program. We argue that the extended C semantics simulates the hardware machine, which executes compiled hypervisor code, given that the compiler is correct.

The second contribution of the thesis is the formal verification of a software TLB and memory virtualization approach, called SPT algorithm. Efficient TLB virtualization is one of the trickiest parts of building correct hypervisors. An SPT algorithm maintains dedicated sets of "shadow" page tables, ensuring memory separation and correct TLB abstraction for every guest. We use our extended C semantics to specify correctness criteria for TLB virtualization and to verify a simple SPT algorithm written in C. The code of the algorithm is formally verified in Microsoft's VCC automatic verifier, which is ideally suited for proofs performed on top of our semantic stack.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich eingehend mit der Verifikation von Hypervisorn und den Herausforderungen, die dabei auftreten. Als ein Hauptergebnis werden erstmalig die verschiedenen Teile der Hypervisor-Verifikationstheorie zu einer einheitlichen Theorie zusammengefasst, in der mehrere komplexen Rechenmodelle auf einander aufbauen. Als Zielplattform für die Virtualisierung wählten wir eine x86-64-Architektur und betrachten Hypervisoren für Multicore-Prozessoren, die in C implementiert sind. Um Code-Verifikation in einem C-Verifizierer zu ermöglichen, definieren wir ein reduziertes Hardware-Modell und zeigen, dass unter bestimmten Bedingungen das ursprüngliche Modell davon simuliert wird. Die C-Semantik wird so erweitert, dass mögliche MMU- und Gast-Interaktionen mit dem Speicher eines Programms berücksichtigt werden. Unter der Annahme, dass der Hyperviser-Code mit einem korrekten Compiler kompiliert wird, argumentieren wir, dass die erweiterte C-Semantik die Hardware-Maschine, welche den kompilierten Code ausführt, simuliert.

Ein weiterer Beitrag dieser Arbeit ist die formale Verifikation eines Algorithmus zur Speicher und TLB-Virtualisierung, der mit Shadow Page Tables (SPTs) arbeitet. Ein SPT-Algorithmus verwaltet Seitentabellen und garantiert Speicherseparierung sowie eine korrekte TLB-Abstraktion für alle Gäste. Wir benutzen unsere erweiterte C-Semantik, um die Korrektheitskriterien für die TLB-Virtualisierung zu spezifizieren und um einen einfachen SPT-Algorithmus zu verifizieren. Die Korrektheit des in C implementierten Algorithmus wurde formal bewiesen mit Hilfe des automatischen Beweiser VCC, der von Microsoft entwickelt wurde.

Contents

List of Theorems xiii

1 Introduction **1**
1.1 Motivation . . . 1
1.2 The Problem of TLB Virtualization . . . 4
1.3 Related Work . . . 6
1.4 Outline . . . 11
1.5 Notation . . . 12
1.5.1 Relations . . . 13
1.5.2 Functions . . . 15
1.5.3 Invariants . . . 15

2 I/O Automata and Simulation **17**
2.1 I/O Automaton . . . 18
2.2 Simulation Proofs . . . 18
2.3 Forward Simulation . . . 19

3 Abstract Hardware Model **21**
3.1 Multicore x64 Hardware Model . . . 22
3.1.1 The Scope of the Model . . . 25
3.1.2 Addressing Convention . . . 26
3.2 Instruction Core - Memory Core Interface . . . 28
3.2.1 Requests and Replies . . . 28
3.2.2 External Actions. . . . 30
3.3 Caches, Store Buffers and Main Memory . . . 31
3.3.1 Memory Types . . . 32
3.3.2 Abstract Cache . . . 33
3.3.3 Store Buffers . . . 38
3.4 Translation Lookaside Buffer . . . 41
3.4.1 Page Table Walks . . . 41
3.4.2 Page Tables and Page Table Entries . . . 42
3.4.3 TLB Model . . . 44
3.4.4 TLB Interface . . . 47
3.5 Memory Core . . . 50
3.5.1 Memory Accesses. . . . 52
3.5.2 TLB Operations . . . 55
3.5.3 Virtualization Actions. . . . 56
3.6 Instruction Automaton . . . 58

4 Reduced Hardware Model **59**
4.1 Specification 60
4.2 Cache Reduction 62
4.3 Ownership 66
4.3.1 Owned and Shared Addresses 66
4.3.2 Ownership Discipline 68
4.4 SB Reduction 70
4.5 TLB Reduction 74
4.5.1 Identity Mapped Page Tables 74
4.5.2 Registers 76
4.5.3 TLB-reduced Hardware Model 77
4.6 Putting It All Together 79
4.6.1 Ownership for Reduced Model 81
4.6.2 Ownership Transfer 82
4.6.3 Main Reduction Theorem 82

5 Intermediate C (C-IL) Semantics **89**
5.1 Sequential C-IL Semantics 90
5.1.1 Types 90
5.1.2 Values 92
5.1.3 Expressions and Statements 93
5.1.4 Configuration and Program 95
5.1.5 Context 96
5.1.6 Memory Semantics 97
5.1.7 Expression Evaluation 99
5.1.8 Operational Semantics 102
5.2 Concurrent C-IL Semantics 105
5.3 C-IL Program Safety 106
5.3.1 C-IL Ownership 106
5.3.2 Safe Expressions 107
5.3.3 Safe Statements 108
5.3.4 Safe Execution 111
5.4 Compiler Correctness 111
5.4.1 Hardware I/O Points 112
5.4.2 Consistency Points 113
5.4.3 Consistency-block Schedule 115
5.4.4 Consistency Relation 116
5.4.5 Software Consistency Points 119
5.4.6 Compiler Correctness Theorem 120

6 C-IL + Ghost Semantics **125**
6.1 Ghost Types and Values 126
6.1.1 Ghost Types 126
6.1.2 Ghost Values 126
6.2 Ghost Memory 128
6.3 Ghost Code 129
6.4 Configuration and Program 130
6.4.1 Configuration 130
6.4.2 Program and Context 131
6.5 Operational Semantics 131

6.6 Simulation Theorem 133

7 C-IL + HW Semantics 137
7.1 Configuration 138
7.2 Operational Semantics 139
7.2.1 C-IL Steps 140
7.2.2 Hardware Steps 141
7.2.3 C-IL + HW I/O Traces 145
7.3 C-IL + HW Program Safety 147
7.4 Simulation Theorem 148
7.4.1 HW Consistency 148
7.4.2 C-IL + HW Simulation 152
7.5 C-IL + HW + Ghost Semantics 159

8 TLB Virtualization 165
8.1 Specification and Implementation Models 167
8.1.1 Host Hardware Model 167
8.1.2 Guest Virtual Machines 168
8.1.3 Equality of Traces 169
8.1.4 VM Simulation 170
8.2 VM Configuration 171
8.2.1 ASIDs and ASID generations. 172
8.2.2 Processor Local Storage 173
8.2.3 Shadow Page Tables 175
8.2.4 SPT Properties 175
8.3 Coupling Invariant 176
8.3.1 Memory Coupling 176
8.3.2 SB Coupling 177
8.3.3 Memory Core Coupling 177
8.3.4 VTLB Coupling 179
8.3.5 Auxiliary VTLB Invariants 179
8.3.6 Reachable Walks 182
8.4 Simulation 185
8.4.1 Simulation for Hardware C-IL Steps 185
8.4.2 Correctness of VMRUN 192

9 Shadow Page Table Algorithm 195
9.1 Types and Data Structures 196
9.1.1 Constants and Types 196
9.1.2 VM Configuration 197
9.1.3 Processor Local Storage 197
9.1.4 Page Tables 198
9.2 Software Walks 199
9.3 Basic Functions on Page Tables 200
9.3.1 Creating an SPT 200
9.3.2 Shadowing a GPT 200
9.3.3 Walking SPTs 202
9.3.4 Walking GPTs 203
9.3.5 Comparing GPTEs and SPTEs 206
9.3.6 Reclaiming SPTs 206

9.4 TLB Lazy Flushing 206
9.5 Intercept Handlers 210
9.5.1 INVLPG Handler 211
9.5.2 MOVE TO *CR*3 Handler 212
9.5.3 PF Handler 214

10 Verification of the SPT Algorithm in VCC 219
10.1 The Verifying C Compiler 220
10.1.1 Memory Model 221
10.1.2 Objects, Invariants, and Ownership 221
10.1.3 Claims 222
10.1.4 Atomic Updates 223
10.1.5 Approvals 224
10.1.6 Scheduling 225
10.2 Modelling Hardware 225
10.2.1 Locating Invariants 226
10.2.2 Host Hardware 226
10.2.3 Virtual Hardware 229
10.3 Shadow Page Table 231
10.4 Virtualization Correctness 231
10.5 Virtual Hardware Simulation 233
10.6 Hardware Thread 235

11 Summary and Future Work 237

Bibliography 241

Index 251

List of Theorems

2.1 Theorem (**Soundness of forward simulation**) 19

4.1 Theorem (**Cache reduction**) 63
4.2 Lemma (**Consistent caches**) 64
4.3 Theorem (**Store buffer reduction**) 71
4.4 Theorem (**TLB reduction**) 78
4.5 Theorem (**Cache, SB, and TLB reduction**) 79
4.6 Lemma (**Safety transfer**) 85
4.7 Theorem (**Main reduction theorem**) 87

5.1 Theorem (**Consistency-block reordering**) 115
5.2 Theorem (**C-IL compiler correctness**) 120

6.1 Theorem (**C-IL + Ghost simulation (1 step)**) 134

7.1 Lemma (**Equality of expression evaluation**) 138
7.2 Lemma (**C-IL step transfer**) 141
7.3 Lemma (**C-IL local sequence safe**) 148
7.4 Lemma (**Safe C-IL + HW program transitive (HW step)**) 148
7.5 Lemma (**Safe and consistent guest step**) 149
7.6 Lemma (**Consistent VMRUN**) 151
7.7 Theorem (**C-IL + HW simulation**) 152
7.8 Lemma (**Safe C-IL + HW + Ghost step**) 161
7.9 Theorem (**C-IL + HW + Ghost simulation**) 162
7.10 Lemma (**Safety of C-IL + HW program.**) 163

8.1 Theorem (**Correct virtualization**) 170
8.2 Lemma (**Complete walks in HTLB**) 181
8.3 Theorem (**Virtualization of hardware steps**) 186
8.4 Lemma (**Correct virtualization of VMRUN**) 193

CHAPTER 1

Introduction

1.1 Motivation

Hardware virtualization is a technology used to provide a layer of abstraction between a computer system and the users utilizing this system. The first virtualization solutions appeared in 1960s and were designed to be used on large and expensive mainframes, usually consisting of multiple CPUs and operating on some sort of the shared memory. Today, with the intensive growth of hardware capabilities, shared multi-threading and shared multi-processing is becoming an integral part of the computer mainstream. As a result, hardware virtualization has recently emerged as a key technology in many areas. Virtualization solutions often provide good benefits in cost, efficiency, and security [HN09]. Virtualization is becoming an important part of safety and security-critical systems in avionics, medical, automotive, and military engineering [GWF10, Day10]. In order to fully achieve the benefits standing behind virtualization, one has to pay significant attention towards reliability of virtualization software.

A *hypervisor*, also called a virtual machine monitor (VMM) [SN05], is a piece of system software, that is responsible for hardware virtualization: it virtualizes system resources of the *host hardware* machine and makes them available for guest operating systems (OS) (Figure 1.1). A guest OS (or simply *guest*) in this case is running in the *virtual machine* (VM) (also called the *guest partition*) provided by the hypervisor. The clients are either aware of the underlying software layer (*para-virtualization*) or have an illusion of being the only system running on a physical machine (*full virtualization*). A hypervisor provides this illusion by saving the state of the virtual machine (VM) when it is not running, and by intercepting and virtualizing certain instructions and events occurring during the execution of the guest code. In case of para-virtualization the code of the guest OS has to be explicitly ported to comply with the API of the hypervisor, while in case of full virtualization the guest OS

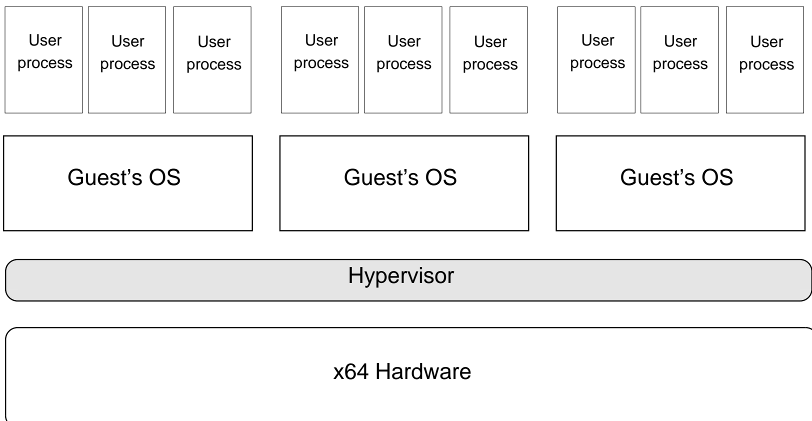

Figure 1.1: Running multiple VMs on a single hardware machine.

can run unmodified.

Conventional testing, when applied to hypervisors, does not always provide satisfactory results. Hypervisors are hard to debug and precise testing of hypervisor features in not always feasible. At the same time, hypervisor correctness is of critical importance for the reliability and safety of the whole computer system. Another important concern is hypervisor security. Hypervisors are often designed for use with general-purpose operating systems, which are allowed to run any code, including malicious or invalid one. The hypervisor's duty is to guarantee that such code does not escape the virtual environment and does not affect execution of other clients, which might only run security-critical trusted applications.

For these reasons, and because of their relatively small size, hypervisors make a viable and interesting target for formal verification. Formal software verification is an act of proving or disproving correctness of a piece of software w.r.t to its specification, using formal methods of mathematics. The main advantage of formal verification in comparison to testing is the fact that verification ensures correct behaviour of the program for all possible inputs and all possible traces, while testing can only guarantee absence of bugs for those inputs and traces, which have been included in the test suite.

Proving formal functional correctness of a hypervisor is not a trivial task. Hypervisor is said to be correct, if it simulates execution of its guest systems. Establishing this simulation formally in a theorem prover is challenging for a number of reasons:

- hypervisors are usually written in a high-level language, such as C, together with portions of assembler code. To verify such code one has to consider mixed semantics of C and assembler, while theorem provers are normally designed for verification of high-level program code only,
- in order to formally prove guest simulation, one has to come up with a realistic hardware model and to encode this model in a theorem prover. Formalizing hardware specifications of modern processors is itself a non-trivial task,
- a hypervisor is running in the most privileged hardware level. Like a

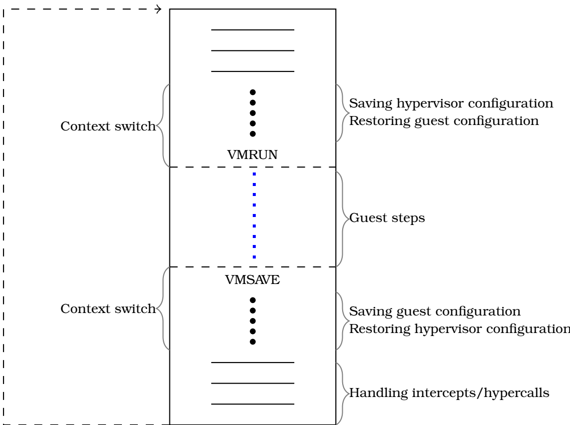

Figure 1.2: Execution thread of the hypervisor: virtualization layer.

regular OS kernel it is responsible for controlling address translations of clients and is normally running in a translated mode itself. Hence, when verifying hypervisor code one has to consider the presence of address translation,

- most hypervisors today are designed to run on multi-processor systems with shared memory. The shared memory of a modern multi-processor machine is not sequentially consistent: it has caches and store buffers. Both caches and store buffers are visible for the programmer writing the hypervisor code and have to be treated in the verification.

For the reasons stated above, hypervisor verification (as well as OS verification in general) is considered to be an important and challenging topic in the field of system and software verification.

The work presented in this thesis closely addresses the challenges of hypervisor verification. The main goals of this thesis are (i) to develop methodology and to build a formal model stack for verification of the virtualization layer[1] of a generic hypervisor for (a subset of) x64 architecture[2] and (ii) to apply this methodology for verification of a translation lookaside buffer (TLB) virtualization algorithm[3], called Shadow Page Table (SPT) algorithm in Microsoft's automatic verifier for concurrent C code (VCC).

We have chosen TLB virtualization as the main target for our verification for several reasons. First, efficient TLB virtualization is perhaps the trickiest part of building correct hypervisors (particularly for processors without hardware support for the second level of address translation (SLAT)). Second, precise

[1]Virtualization layer of the hypervisor consists of services, responsible for virtualization. This includes intercept handling, context switching, and hypercalls (Figure 1.2). In contrast to that, the kernel layer of the hypervisor is responsible for low-level features, such as thread switch and inter-processor communication.

[2]There is no standard naming convention for the 64-bit extension of the x86 hardware. AMD and Intel use the names "AMD64" and "Intel 64" (former "IA-32e" and "EM64T") for their vendor-specific implementations, while the names "x86-64" and "x64" are used in the industry as vendor-neutral terms.

[3]Though we call it "TLB vitualization", it is in fact "TLB, MMU, and memory virtualization" algorithm.

reasoning about memory management unit (MMU) and TLB behaviour is central to the correctness of the memory manager of the hypervisor: since flushing of the TLB is quite expensive, memory managers often use different tricks to avoid flushes whenever possible by allowing the hardware TLB to be out-of-sync with the page tables (PTs). Third, correctness of TLB virtualization is crucial for deriving such important security properties, as separation of guest partitions. Fourth, in spite of the critical importance of MMU behavior, it has never been seriously treated in kernel and hypervisor verification.

1.2 The Problem of TLB Virtualization

When the code is executed in the VM, address translation consists of two stages: first, a *guest virtual address* is translated into the *guest physical address*, which would be used for memory accesses if the code was run on the physical machine alone. Second, the guest physical address is translated into the *host physical address*, which is then used for accesses to the memory of the host machine. The second stage of address translation is controlled by the hypervisor and is transparent to the guest OS.

Translations of guest physical to guest virtual addresses are defined by the means of *guest page tables* (GPTs), which are located in the memory of the virtual machine. All accesses to the guest memory performed by the guest code are virtualized by the hypervisor with the help of the *virtual TLB*[4].

There are two main approaches for TLB and memory virtualization: a hardware-assisted solution and a software solution. In the hardware-assisted approach, which requires the hardware support for SLAT (called "nested paging" by AMD [Adv08] and "extended page table mechanism" by Intel [NSL+06], [Int11, Chapter 25]) MMU operates with two sets of page tables. The first one is the set of guest page tables and the second one is the set of nested page tables, which implement the guest physical to host physical translation. The hypervisor normally maintains a separate set of nested page tables for every guest OS. The hardware MMU walks two sets of page tables simultaneously: every guest physical address, obtained from the fetched GPT entry is translated to the host physical address using nested page tables. Thus, to perform a single translation of a virtual address in the long mode (with 4 levels of address translation), the TLB has to perform at most 20 fetches of PT entries (1 fetch of GPT entry and 4 fetches of nested PT entries for every level of translation), compared to at most 4 fetches for a regular translation.

A standard approach to software TLB virtualization in the hypervisor is to maintain a set of SPTs (Intel uses the term "active page table hierarchy" instead [Int11, Chapter 28]), where each SPT is a "shadow" of some GPT, which is linked (or was recently linked) to the page table graph of the guest (Figure 1.3) [Phi06].

SPTs are used by the host TLB to perform address translations when the machine is executing the guest code. They are maintained solely by the hypervisor and are not visible for a guest OS running in the VM. Guest

[4] Intel uses the term "virtual TLB" only as a name for software mechanisms for virtualized page translations [Int11], while we consider a more general meaning of a virtual TLB, as a virtual device being responsible for providing address translations for VMs independently on what TLB virtualization approach is used.

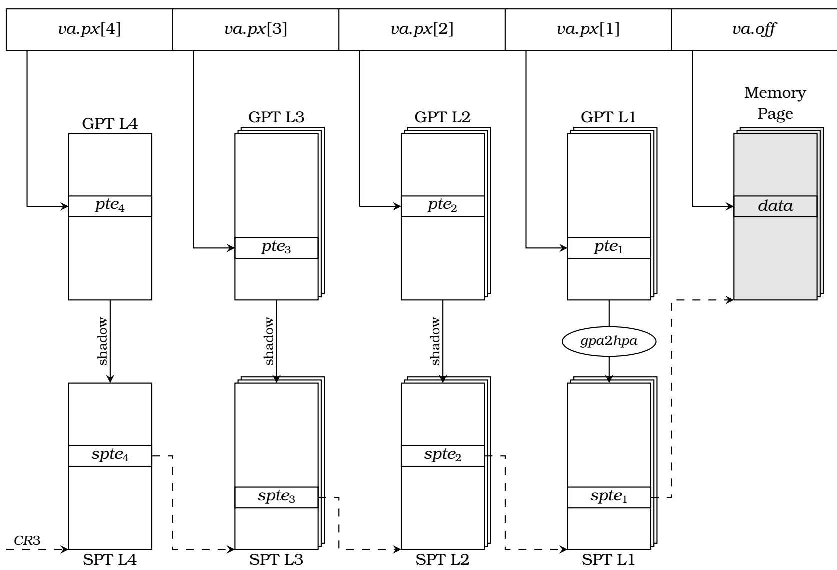

Figure 1.3: Software TLB virtualization: shadow page tables.

TLB-controlling instructions, such as TLB invalidation or modification of control registers (e.g., *CR*3 register), are intercepted by the host hardware and virtualized by the hypervisor. When a memory access by the guest results in a page fault, the hypervisor emulates the steps of the virtual MMU by walking GPTs, setting access and dirty bits in the GPTs, and caching the translations in the SPTs. Thus, the SPTs, the intercept handlers, and the host TLB act in concert to provide a virtual TLB abstraction to the VM.

While the hardware-assisted TLB virtualization today is the preferable choice in most cases (because of quite high costs for entering and exiting the hypervisor) [AA06], the support for a software VTLB implementation is still present in most of the modern hypervisors [Kiv07, BDF+03, Wal02]. Disadvantages of the hardware approach, for instance, include the large overhead in the number of memory accesses due to the required fetches of host PTEs. As a result, in certain user scenarios software TLB virtualization approach may perform faster than the hardware one [BLD+10, BLD11]. Even more promising looks the adaptive virtualization approach [WZW+11], which dynamically switches between SPTs and nested paging depending on the workload of the hypervisor.

Processor with the hardware-assisted virtualization support, besides the hardware SLAT feature (which is not necessarily supported), provide a number of other virtualization services [Adv11a, Int11, Int12], which normally include:

- support for multiple address spaces. Every translation in this case is tagged with an *address space identifier* (ASID); only one ASID can be active at a time,
- the mechanism to save the state of the VM to the memory (in AMD64 this is achieved by *VMSAVE* instruction),
- the mechanism to automatically load the saved state of the VM to the

registers of the host processor (in AMD64 this is achieved by *VMLOAD* and *VMRUN* instructions),

- a dedicated execution mode (we call it *guest mode* in contrast to *hypervisor mode*), where certain instructions and events are treated as a special kind of traps called *intercepts*, and
- an intercepts mechanism, which automatically stops execution of the guest code if a certain instruction or event has been encountered, and starts execution of an *intercept handler*, which is a part of the hypervisor (in AMD64 this is called a *VMEXIT* event).

In the hardware models presented in this thesis we assume the presence of virtualization extensions, but no hardware SLAT. Yet, an SPT algorithm presented in Chapter 9 relies only on TLB support for multiple address spaces and with slight modifications can be also applied in hypervisors, which run on processors without virtualization extensions. Such hypervisors perform virtualization purely by software means, e.g., using a mechanism called *binary translation* [VMw07].

1.3 Related Work

Hardware Model. A formal definition of a (multi-processor) hardware model involves two main aspects: the shared memory model and the model of the instruction set architecture (ISA). In this thesis we focus on the memory model of the x64 architecture, while leaving the ISA part of the model as a black box.

The first *sequentially consistent* memory model for a multiprocessor machine was defined by Lamport in [Lam79]. Since then an extensive research in the field of memory models has been done, leading to the development of various relaxed memory models, which are not sequentially consistent [DSB86, AG96, HKV98]. Most of the modern, real-world architectures implement relaxed memory models due to the number of optimizations and speed-ups which they provide in comparison to a sequentially consistent memory model. The memory model of the x64 architecture is described in the Intel and AMD manuals and white papers [Adv11a, Adv11b, Int11, Int07]. The way how the memory model is described in vendor manuals is by listing the rules for reordering and execution of memory accesses. Several attempts have been made to come up with a formal model capturing these rules. Sarkal et. al. formalized the rules for accesses with a write-back memory type in [SSN+09]. Further, they developed the x86-CC model, which is a relaxed memory model of the x86 architecture with causal consistency. This modelled turned out to be too strict and to exclude certain execution traces, possible on the real hardware. As a result, the new model was developed, called x86-TSO [OSS09]. TSO stands for *total store ordering*, the memory model first introduced for the SPARC V8 processor [SI94]. The TSO model allows reads to return the value of its own processor's write before this write is made visible to other processors, while not allowing the read to return the value of other processor's write, which is not yet delivered to the memory[5]. The scope of the x86-TSO model covers typical user code and system code, which is using write-back memory type,

[5] It is believed, that the x86 memory model is by vendor intentions a variation of the TSO model.

does not have misalignment, self-modification of the code or the page tables, and which does not cause exceptions.

Degenbaev in [Deg11] presents a comprehensive model of the x64 architecture, including both the memory model and the ISA model. The work has started as part of the Verisoft XT project on Microsoft's Hyper-V hypervisor verification. Since the model was intended to be used for system-code verification, it includes low-level design features, such as TLBs, APICs, interrupts, different cache-modes, etc. The hardware model presented in this thesis is based upon Degenbaev's model.

Hardware Reduction and Ownership Discipline. Despite the fact that almost all modern hardware processors have relaxed memory models, most verification techniques for concurrent code still rely on sequentially consistent memory [App11, CMST10, O'H04]. To apply these approaches for programs running on a hardware machine with a relaxed memory model one has to ensure additional guarantees and to prove a number of hardware reduction theorems [DPS09]. A hardware reduction theorem is a simulation theorem between two hardware models, where one model has less visible components than the other. Applying proper reduction theorems, one can for instance ensure that a program verified for the sequentially consistent memory, also behaves correctly on a hardware machine with a store buffer, a cache system, and an address translation mechanism. Caches and MMUs are usually made invisible by asserting properties on page tables and hardware registers. In contrast to that, making store buffers invisible in a concurrent environment requires arguing about the code of the program itself.

When dealing with separation logic, a popular approach to store-buffer reduction is to show absence of data races in a program, by ensuring lock protection for all shared resources [AS07, OSS09]. If a program is data-race free, then one can make store buffers invisible simply by ensuring that all operations with synchronization primitives (e.g., locks) are performed with interlocked atomic instructions. O'Hearn [O'H04] uses "ownership" of memory locations for separation logic to make sure that dereferencing or disposing a memory cell does not cause a race condition. The ownership of a given cell can be transferred in and out of shared resources such as semaphores. In [BCHP05] the ownership concept for separation logic is replaced with more general "permissions". This allows arguing about shared memory cells, which can be written by one thread and read by many threads. The concept of fine-grained permissions is used by Appel in his Verified Software Toolchain project [App11] and has been recently integrated into Leroy's formally verified CompCert compiler [LABS12].

Though a mechanism of permissions for separation logic is powerful enough to argue about data-race free concurrency, including multiple-read single-write protocols, it is still not sufficient for fine-grained concurrency and "write-write" data races. In contrast to that, Ridge [Rid07] uses operational reasoning on top of a TSO memory model and guarantees sequential consistency by explicitly performing a store buffer flush after every write to shared data. Owens in [Owe10] shows sequential consistency for a TSO memory model by ensuring absence of so-called *triangular races*, i.e., races between a read and write operation where the read operation is preceded by

another write operation on the same thread, and there are no synchronization primitives in between (all other kinds of data races are allowed). Linden and Wolper in [LW11] use a similar approach and provide an algorithm for insertion of memory fences to guarantee that store buffer is appropriately flushed. Cohen and Schirmer in [CS10] generalize this approach by introducing an ownership discipline, which also ensures absence of triangular races for volatile data by requiring a store buffer flush to be performed in between a shared write and a subsequent shared read. At the same time their model allows sequential accesses (without any flushes) to lock-protected non-volatile data by allowing ownership transfer under certain conditions to occur. Both Owens' and Cohen-Schirmer's approaches avoid having to consider store buffers as an explicit part of the state of the target model.

The ownership model introduced in this thesis is done in the style of Cohen-Schirmer model, though enforcing stronger restrictions on the code (by requiring all volatile accesses to be performed with interlocked atomic instructions). As part of the future work, we plan to replace the ownership discipline in our framework with the Cohen-Schirmer ownership model.

OS Kernel Verification. A good survey on the OS verification has been given by Klein in [Kle09]. The first groundbreaking attempt in pervasive system verification was the famous CLI stack project [BHMY89a, BHMY89b], which included verification of the KIT kernel [Bev89b, Bev89a]. KIT stands for "Kernel for Isolated Tasks" and is a simple multitasking kernel implemented in assembler. The Flint project did not directly aim at the OS verification, but has contributed into the verification of the low-level context switching [NYS07, FSGD09] and into the treatment of hardware interrupts and preemptive threads.

Substantial progress towards the goal of a fully verified OS kernel was made in L4.verified and Verisoft projects. The main code verification technology used in both projects is the interactive environment in the theorem prover Isabelle [Sch05]. The Verisoft project [Ver08] was aimed at the pervasive formal verification of the entire computer system from the hardware level (VAMP processor [BJK+06]) up to application level [AHL+09, HP07, APST10]. As part of the project the functional correctness of the CVM (Communicating Virtual Machines) microkernel was proven [IdRT08]. CVM was implemented in a C dialect called C0 [LPP05] together with inline-assembly. Correctness of CVM was mainly stated in the form of a simulation theorem between the kernel implementation and abstractions of virtual user processes.

The L4.Verified project [KEH+09] focuses on the functional verification of high-performance C implementation of the seL4 (secure embedded L4) microkernel [EKD+07], which is an evolution of the classical L4 microkernel [Lie95]. In contrast to Verisoft, L4.Verified considers not a slightly changed variant of C, but rather a true subset of C including such unsafe features as pointer arithmetic and unchecked type casts. Hence, implementation of seL4 can be compiled with a regular C compiler. In L4.verified the compiler is considered as a part of the trusted code-base, while in Verisoft a non-optimizing C0 compiler has been verified [LP08a, Pet07].

Hypervisor Verification. Compared to the OS kernel verification, field of the hypervisor verification is less mature. While there is a number of verification projects dealing with hypervisors, most of them considered only certain safety and security properties leaving complete functional verification out of scope. The Nova micro-hypervisor verification project [TWV+08] aimed only at low-level properties of the code, such as memory and hardware safety and termination, and did not consider virtualization correctness at all [Tew07]. In [BBCL11] authors show isolation properties for a minimalistic model of a hypervisor running on a simplified hardware without MMUs and TLBs. [CVJ+12, VMQ+10] aim at showing memory integrity of the hypervisor, i.e., the fact that the hypervisor memory can not be modified by software running at a lower privilege level. Both isolation and integrity properties follow from the correctness of TLB and memory virtualization, which we address in this thesis.

Alkassar and Paul in [AP08] outline a virtualization correctness proof of a simple hypervisor for a single-core RISC machine with a single level address translation but without a TLB. The functional verification of this hypervisor was first presented in [AHPP10] and was completed with respect to the assembly portions in [Sha12]. The result of the verification is a simulation proof, carried out in Microsoft's VCC verifier. This work was done in the frame of the Verisoft XT project [The12] and was a precursor for the main target of the Verisoft XT, which was the complete verification of the Hyper-V hypervisor including virtualization correctness [LS09]. Yet, this goal was not fully achieved. The work presented in this thesis was started as another part of the Verisoft XT, which aimed at the development and verification of a prototypical academic hypervisor for the x86 architecture. A sketch of the top-level TLB virtualization proof from this thesis was previously presented in [ACH+10, ACKP12].

TLBs/MMUs in OS and Hypervisor Verification. MMU and TLB behavior has never been seriously treated in OS and hypervisor verification. For example, the Verisoft project used a synthetic hardware model without TLBs, while the L4.verified project explicitly assumed that the TLBs were kept in sync with the page tables, essentially making the TLBs transparent to software. A similar approach was chosen in the Nova micro-hypervisor verification project, which used an abstract model of IA-32 hardware with MMU, but without the TLB. To make this argumentation sound, page tables were assumed to be read-only and to provide non-aliasing address translations. In our verification framework we use an analogous approach to handle MMU behaviour when the hypervisor's own code is being executed. For the case when the guest code is running (and SPTs are used for address translations) we make the TLB component visible on the C level and allow the MMU to perform writes to the memory by setting access and dirty bits in page tables.

Integrated and Mixed Semantics. As part of the work presented in this thesis we extend the semantics of the C-IL language (C Intermediate Language [Sch12b]) with the hardware state, responsible for execution of the guest memory accesses. This involves modelling the behaviour of the hardware MMU on the C level and exposing the current TLB and register state in the

integrated C-IL + HW semantics. Previously the problem of arguing about the hardware state and the device behaviour on the source-code level has been treated in the projects mentioned above.

In the L4.verified project the state of the C machine is extended with the hardware components, which are accessible with the assembly functions. These assembly functions are not verified in a single framework with the rest of the code, but are isolated into separate functions and are verified separately against their specification. This specification is then used in the verification framework every time when an assembly function is called.

The Verisoft project followed a similar approach, but used a single formal framework for all proofs. The low-level hardware components were abstracted into an extension of the C0 state. The effect of inline assembly and device steps was modelled by so-called *XCalls* [AHL+09], which are atomic specifications updating both the extended and the original state of the C0 machine. Extension of the semantic stack with XCalls made it possible to verify assembly portions and device drivers in Hoare logic and to transfer the result of the verification down to VAMP assembly with devices. In order to justify XCall semantics a reordering theorem was proven, where all interleaved and non-interfering device steps are delayed until some inline assembler statement is encountered [Alk09]. In this thesis we also rely on a reordering theorem to justify the soundness of the C-IL + HW semantics. The difference between the XCalls and our approach is that we consider a different interleaving scheme, where the steps of "devices" (which in our case are the steps of the processors executing guest code) may interleave with the program steps only at the so-called *consistency points* (Chapter 5).

Schmaltz and Shadrin in [SS12] present an integrated operational small-step semantics model of C-IL language with macro-assembler code execution (C-IL + MASM). They sketch a theory connecting the semantic layer with an ISA-model executing the compiled code. C-IL + MASM semantic model was used to justify verification of assembly portions of a simple hypervisor for the VAMP processor [Sha12].

Theory of Multicore Hypervisor Verification. The overall theory of multicore hypervisor verification presented in this thesis is the result of the joint work, which started in the frame of the Verisoft XT project and continued afterwards on the chair of Prof. Paul in the Saarland University. In [DPS09] Degenbaev, Paul, and Schirmer outlined the pervasive theory of memory for TSO machines stating cache, SB, and TLB reduction theorems and basic compiler consistency. The general methodology for multicore hypervisor verification was sketched by Paul in talks given in Strasbourg and Kaiserslautern during meeting of the Verisoft XT project[6] and by Cohen in his talks and discussions summarized in [HP10]. The methodology and the overall theory of multicore hypervisor verification were further developed in numerous oberseminar talks and discussions at the chair. Cohen, Paul, and Schmaltz in [CPS13] outline the current state of this theory (including the topics which are not addressed in this thesis, such as e.g. interrupts and assembly code verification). Nevertheless, this thesis is the first document where different pieces of hypervisor verification theory are formally put together into a single,

[6]Slides of these talks can be provided by Paul upon request.

uniform paper-and-pencil theory and a formal semantic stack for multicore hypervisor verification is presented.

1.4 Outline

This chapter ends with the description of notation used throughout this thesis. The remainder of the thesis is structured as follows.

Chapter 2 gives a brief introduction on the general theory of I/O automata and simulation proofs.

In Chapter 3 we introduce the abstract hardware model of (the subset of) x64 architecture. The hardware is modelled as two communicating I/O automata, where one automaton is responsible for instruction execution and the other one is responsible for memory accesses and TLB operations. We model in detail only the second automaton, while leaving the first one as a "black box", which can be further instantiated with the x64 ISA.

In Chapter 4 we introduce a reduced hardware machine and prove hardware reduction theorems. We perform reduction in three phases: first we reduce caches, then we reduce store buffers, and finally - make address translation invisible by reducing TLBs. Caches are reduced for both the guest and the hypervisor execution modes, while store buffer and TLB reduction is done only for the hypervisor mode. Reduction is proven in the form of a step-by-step simulation theorem, between a reduced hardware machine and an original one. We introduce the safety properties, which have to be maintained on the reduced machine in order for the reduction theorems to go through. Caches are made invisible by requiring all memory accesses to be performed in a "write-back" memory mode. TLBs are made invisible in the hypervisor mode, by fixing the properties of the page tables used for hypervisor own address translations and ensuring that the memory region, where these page tables are located, stays unchanged afterwards. To prove a store-buffer reduction theorem we introduce a simple ownership discipline, which has to be maintained by all steps of the hardware machine.

In Chapter 5 we describe the operational semantics of the C-IL language, enriched with some virtualization primitives. We introduce a reordering theorem for execution sequences of reduced hardware machines. In a reordered execution sequence interleaving of steps of different processors can be done only at so-called consistency points. The set of consistency points in this case must include all hardware states before and/or after an access to a shared resource. We lift the safety properties defined for the reduced hardware model, including the ownership discipline, to the C-IL level and sketch a compiler correctness theorem for a generic, optimizing compiler.

In Chapter 6 we extend the C-IL semantics with the ghost state.

In Chapter 7 we make certain parts of the hardware model visible in another extension of the C-IL semantics, which we call C-IL + Hardware (C-IL + HW). We show that a regular C-IL program running in parallel with the guest code behaves exactly the same way, as defined by our C-IL + HW semantics. As a result, we can prove properties over such program in a C program verifier by extending the program with the hardware component (and a hardware thread) and verifying the combined program altogether. Further, we add the ghost

state to C-IL + HW and obtain the C-IL + HW + Ghost semantics, which we later use for verification of the SPT algorithm.

In Chapter 8 we specify correctness of TLB virtualization. We define the coupling invariant between abstract data structures of the hypervisor and the abstract configuration of VMs, which are modelled as instances of the hardware model introduced in Chapter 3. Correctness of TLB virtualization is stated in the form of a simulation theorem, between the execution sequence of the hypervisor program inside C-IL + HW + Ghost semantics and the execution of abstract VMs. In Chapter 8 we prove this theorem for non-deterministic transitions of the hardware component of the C-IL + HW + Ghost machine.

In Chapter 9 we provide implementation of a simple SPT algorithm and give the most crucial arguments on its correctness: we maintain the coupling invariant after every step of the program and show that the abstract VMs perform only those steps, which are supposed to be emulated by a given intercept handler.

In Chapter 10 we discuss verification of the SPT algorithm from the previous chapter in VCC. We focus on modelling of the hardware component of a thread from C-IL + HW semantics, modelling of the virtual hardware state in VCC, and simulation of steps of the abstract VMs. We provide the most crucial portions of VCC annotations, necessary for understanding of our approach and methodology.

In Chapter 11 we conclude and outline the future work.

1.5 Notation

The set of integers is denoted by $\mathbb{Z}$. The set of natural numbers including 0 is denoted by $\mathbb{N}$. The set of natural numbers in the range from 0 to $k-1$ is denoted by $\mathbb{N}_k$. The set of boolean values $\{0, 1\}$ is denoted by $\mathbb{B}$.

The type for a list of $n \in \mathbb{N}$ values of type $\mathbb{T}$ is denoted by $\mathbb{T}^n$. For a given list $l \in \mathbb{T}^n$, we use the functions $\mathbf{hd}(l)$ and $\mathbf{tl}(l)$ to return the head and the tail of list l respectively. The i-th element of list l is identified by $l[i]$ (we start counting from index 0) and the length of list l is obtained by the function $|l|$. The last element of list l is identified by $\mathbf{last}(l)$. The sublist from the element i to the element j is identified by $l[j : i]$. The concatenation of two lists l_1 and l_2 is denoted by $l_1 \circ l_2$. The reverse list of list l is denoted by $\mathbf{rev}(l)$.

The function $\mathbf{map}(f \in \mathbb{T}_1 \mapsto \mathbb{T}_2, l \in \mathbb{T}_1^n) \in \mathbb{T}_2^n$ returns list l', where every element is obtained by applying the function f to a respective element of the list l.

The set of all possible strings with the elements from the set $\mathbb{T}$ is denoted by $\mathbb{T}^*$:

$$\mathbb{T}^* \stackrel{\text{def}}{=} \bigcup_{n=1}^{\infty} \mathbb{T}^n \cup \{[]\}.$$

The power set (i.e., the set of all subsets) of a set S or of a type $\mathbb{T}$ is denoted by 2^S and $2^{\mathbb{T}}$ respectively. The power set of $\mathbb{T}$ can be also considered as a boolean map. Hence, the following types are considered equivalent:

$$2^{\mathbb{T}} = (\mathbb{T} \mapsto \mathbb{B}).$$

The pair of two elements $a \in \mathbb{T}_1$ and $b \in \mathbb{T}_2$ is denoted by $(a, b) \in (\mathbb{T}_1 \times \mathbb{T}_2)$. We access the first element of a pair with the function **fst** and the second element with the function **snd**.

The type of n-bit strings $\{0, 1\}^n$ is denoted by $\mathbb{B}^n$. We use the overloaded operators $+, -, \cdot, /$ to perform arithmetic operations (modulo 2^n) on bit strings of type B^n. When performing arithmetic operations on bit strings of different length, we zero-extend the shorter string to match the longer one and perform the operation modulo 2 in the power of the length of the longer string. To convert a bit string $a \in \mathbb{B}^n$ to a natural number we write $\langle a \rangle$, where

$$\langle a \rangle \stackrel{\text{def}}{=} \sum_{i=0}^{i<n} (a_i \times 2^i).$$

For conversion of a natural number $b \in \mathbb{N}$ to a bit-string with the length $n \in \mathbb{N}$ we write $bin_n(b)$.

A record R is defined as a tuple with named components and their types. For example, a record type containing two components of types $\mathbb{B}^n$ and $\mathbb{B}^m$ is defined as follows

$$R \stackrel{\text{def}}{=} [a \in \mathbb{B}^n, b \in \mathbb{B}^m].$$

The component a of a record $x \in R$ is obtained by $a.x$. The update of components of a record $x \in R$ with the values $a' \in \mathbb{B}^n$ and $b' \in \mathbb{B}^m$ is denoted as

$$x := x[a \mapsto a', b \mapsto b'].$$

For update of component a of nested record $(z.t) \in R$, we use a shorthand $z := z[t.a \mapsto a']$, which is equivalent to $z := z[t \mapsto t[a \mapsto a']]$. The construction of a new record $y \in R$ with component values a_0 and b_0 is denoted as $y := R[a \mapsto a_0, b \mapsto b_0]$.

We use maps to identify functions which can be passed as parameters to other functions. To distinguish an access to a map $m \in \mathbb{B}^n \mapsto \mathbb{B}$ from an application of a "normal" function, we use notation $m[i]$ for elements of the map. We update a map in the same way, as we update a record:

$$m := m[i \mapsto a', j \mapsto a'].$$

We use maps to boolean values for modelling sets. In this case we may also write $i \in m$, to denote that i is an element of the set m ($m[i] = 1$).

Let $m \in \mathbb{T}_1 \mapsto \mathbb{T}_2$ be a map and $\mathbb{T}_3 \subset \mathbb{T}_1$. Then we write $m[\mathbb{T}_3]$ to restrict m to $\mathbb{T}_3$:

$$m[\mathbb{T}_3] \in \mathbb{T}_3 \mapsto \mathbb{T}_2$$
$$\forall i \in \mathbb{T}_3 : m[i] = (m[\mathbb{T}_3])[i].$$

1.5.1 Relations

The hardware in this thesis is modelled as an I/O automaton (Section 2.1). An I/O automaton is a labeled transition system with input and output parameters. We define the hardware transition relation by splitting it into smaller transitions, each of which can happen nondeterministically, if the

precondition for its triggering is satisfied. The overall transition we denote by Δ.

To denote that hardware transition a from state h to h' is a part of Δ we write $(h, a, h') \in \Delta$, or simply $h \xrightarrow{a} h'$.

For every hardware transition we provide i) its *label* with the list of the input parameters, ii) the *guard* of the transition (i.e., the set of conditions under which the transition may occur), and iii) the *effect* of the transition on the resulting hardware configuration.

Each hardware transition has its own visibility scope, where the following names are visible:

- input parameters of the transition,
- the state of the hardware components before the transition has occurred,
- the state of the hardware components after the transition has finished,
- function names,
- free variables declared inside the transition relation, which are implicitly universally quantified.

As an example of a hardware transition relation, we consider the following transition of the abstract cache (Definition 3.23):

label	$\mathit{fetch\text{-}line\text{-}from\text{-}ca}(i \in \mathit{Pid}, j \in \mathit{Pid}, pa \in \mathbb{B}^{qpa})$
guard	$ca[j].v[pa]$, $mt = \mathit{ca\text{-}memtype}(p[i], tlb[i], pa)$, $\mathit{cacheable}(mt)$,
effect	$ca'[i].v[pa] = 1$, $ca'[i].data[pa] = ca[j].data[pa]$

In order for cache i to successfully fetch a line from cache j, the data has to be valid in cache j, and the address has to have a cacheable memory type. As a result of this transition, the data is copied from cache j to cache i, and is marked to be valid in cache i. We assume implicit framing for components of the hardware not stated to be changed explicitly.

Formally, the transition given above is transformed into the following statement:

$$\begin{aligned}
&\forall i, j \in \mathit{Pid} : \forall pa \in \mathbb{B}^{qpa} : \forall mt \in \mathit{MemType}\\
&\qquad h.ca[j].v[pa]\\
&\qquad \wedge\ mt = \mathit{ca\text{-}memtype}(h.p[i], h.tlb[i], pa)\\
&\qquad \wedge\ \mathit{cacheable}(mt)\\
&\qquad \wedge\ h' = h[ca[i].v[pa] \mapsto 1, ca[i].data[pa] \mapsto h.ca[j].data[pa]])\\
&\qquad \Longrightarrow\ (h, \mathit{fetch\text{-}line\text{-}from\text{-}ca}(i, j, pa), h') \in \Delta.
\end{aligned}$$

For the quantified variables inside function and transition definitions, we often omit their type if it can be clearly inferred from the context.

Given hardware states h and h', the expression $h \xrightarrow{\beta} h'$, where $|\beta| = n$ and $n > 0$ denotes a hardware execution sequence $h^0, \beta_0, h^1, \beta_1, \ldots, \beta_n, h^n$, where $h^0 = h$, $h^n = h'$ and every next hardware state is obtained from the previous

one by performing the corresponding step from β:

$$\forall i < n : h^i \xrightarrow{\beta_i} h^{i+1}.$$

In case when we define a function or a predicate on the hardware execution sequence $h^0 \xrightarrow{\beta} h^n$, we explicitly provide as a parameter only the sequence of actions β, assuming that hardware states h^i for this sequence can be derived from the context where the function/predicate is used. For instance, in the definition of a safe hardware sequence (Definition 4.39) we write

$$\mathit{safe\text{-}seq}(\beta, o, o') \stackrel{\text{def}}{=} \exists o^0, \ldots, o^n : o^0 = o \wedge o^n = o' \wedge \forall i \leq n : \mathit{safe\text{-}conf}(h^i, o^i),$$

assuming that the states $h^0, h^1, \ldots, h^n$ are provided implicitly. Further, we use the predicate $\mathit{safe\text{-}seq}(\beta, o, o')$ only in the context where the sequence $h^0 \xrightarrow{\beta} h^n$ is well-defined.

1.5.2 Functions

For every function used in the hardware model we provide its signature (function name, parameters, and the type of the result). We write a function body as a mathematical expression. Sometimes we only declare a function and leave the function body undefined e.g., if its definition varies depending on execution modes or some vendor-specific architectural features.

The functions which we use in the definition of hardware models often return a meaningful result only on a subset of possible inputs. For these functions we define the function domain as a predicate with the name $f_{\surd}$, where f is a function name. We overload the functions *read* and *write* to represent data accesses to different components of the hardware system.

When defining operational semantics of the C intermediate language we also sometimes use partial functions which we denote as $f \in \mathbb{T}_1 \rightharpoonup \mathbb{T}_2$, where f is the name of the function. The domain of such function is then denoted by $\mathit{dom}(f)$.

1.5.3 Invariants

Throughout this thesis we establish a number of properties over the hardware and the software, which are then later used in the proofs of theorems and lemmas. Since these properties are supposed to hold for all configurations of the system, we call them invariants.

When defining an invariant, we provide its name and the established property, e.g., the following definition of an invariant (Invariant 4.27)

name	$\mathit{inv\text{-}cr3\text{-}cacheable}(h \in \mathit{Hardware})$
property	$\forall i \in \mathit{Pid} : \neg h.p[i].\mathit{CR3}.\mathit{CD}$

is equivalent to

$$\mathit{inv\text{-}cr3\text{-}cacheable}(h) \stackrel{\text{def}}{=} \forall i \in \mathit{Pid} : \neg h.p[i].\mathit{CR3}.\mathit{CD}.$$

CHAPTER 2

I/O Automata and Simulation

2.1 I/O Automaton

2.2 Simulation Proofs

2.3 Forward Simulation

The correspondence (or equivalence) between two transition systems I and S, where I is regarded as implementation and S is considered as specification, is often expressed by the concept of *trace inclusion*, where the traces of the implementation system are included into the traces of the specification one [SAGG+93]. If all traces of I are contained in the traces of S, then we say that S (specification) *simulates* I (implementation), and call the correspondence between I and S *simulation*. The simulation proof formally captures the natural structure of many informal "paper-and-pencil" correctness proofs. Intuitively, a system S simulates the system I (or I is simulated by S) if the system S matches all steps of I. The existence of simulation between I and S allows to reduce the behaviour of I to the behaviour of S when showing some properties of I. More precisely, if S simulates I, then any property exhibited by I is also exhibited by S.

We use simulation as a base technique for different proofs presented in this thesis. This includes a hardware reduction proof, a compiler correctness theorem, and a TLB virtualization proof. In this chapter we introduce basic I/O automata and give a brief overview on simulation proofs.

2.1 I/O Automaton

An I/O automaton [LT87, LT89] is a labeled transition system, which performs *internal* and *external* actions. The internal actions are performed on the internal parts of the state and are not visible outside of the system. The external actions are divided into input and output actions and either require some input data to occur, or produce the output result.

An I/O automaton, or simply an automaton, A is a tuple consisting of four components, where

Definition 2.1 ► I/O automaton

- $\mathit{states}(A)$ is a set of states (either finite or infinite),
- $\mathit{start}(A)$ is a nonempty set of start states s.t. $\mathit{start}(A) \subseteq \mathit{states}(A)$,
- $\mathit{sig}(A)$ is an action signature $(\mathit{ext}(A), \mathit{int}(A))$, consisting of external actions $\mathit{ext}(A)$ and internal actions $\mathit{int}(A)$. The set $\mathit{ext}(A)$ of external actions consists of input actions $\mathit{in}(A)$ and output actions $\mathit{out}(A)$. The set of all actions $\mathit{acts}(A)$ is $\mathit{ext}(A) \cup \mathit{int}(A)$,
- $\mathit{steps}(A)$ is a transition relation of A s.t.

$$\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A).$$

For $s, s' \in \mathit{states}(A)$ and $a \in \mathit{acts}(A)$ we say that $(s, a, s') \in \mathit{steps}(A)$ is a *step* or a *transition* of the automaton A. The state s is called a *pre-state* and s' is a *post-state* of the transition.

An *execution fragment* $\omega = s_0, a_1, s_1, a_2, s_2, \ldots$ of A is a finite or infinite sequence of states and actions starting with a state s_0, ending in a state s_n (if the sequence is finite), and satisfying for all $i < n$

$$(s_i, a_{i+1}, s_{i+1}) \in \mathit{steps}(A).$$

For an execution fragment ω we use the functions $\mathit{first}(\omega)$ and $\mathit{last}(\omega)$ to obtain respectively the first and the last configuration (if the sequence is finite) of A in the fragment ω.

The *trace* (or the external behaviour) of an execution fragment ω of an automaton A is the sequence of external actions extracted from ω. We denote the trace of ω by $\mathit{trace}(\omega)$.

An *execution* of A is an execution fragment ω starting in a state $s_0 = \mathit{first}(\omega)$ s.t. $\mathit{first}(\omega) \in \mathit{start}(A)$. We say that a sequence of actions $\beta \in \mathit{acts}(A)^*$ is a trace of an automaton A if there exists an execution ω of A s.t.

$$\mathit{trace}(\omega) = \beta.$$

We denote the set of all traces of A by $\mathit{traces}(A)$.

2.2 Simulation Proofs

Different types of simulation, having generally the same goals, can be applied to different kinds of systems. The most commonly used types of simulation for software and hardware verification are *refinement*, *forward simulation* and *backward simulation*. The refinement is the most straightforward type of simulation, where every step of I has a corresponding step of S, which begins and ends in the respective images of the beginning and ending states of the

step in I [LV95]. As a result, every trace of I is at the same time a trace of S. The correspondence between the states of I and S is established by an abstraction function, which is called a *refinement* from I to S.

While the refinement is a powerful simulation technique for verification of sequential programs and for showing properties of deterministic automata, it is often inapplicable for verification of complex, distributed, non-deterministic transition systems in the concurrent environment. Particularly, constructing the refinement mapping from I to S is not always feasible. More general approaches, which could be applied to a broader set of problems, are forward and backward simulation proofs. The idea of both approaches is to construct an execution sequence of the specification system for every step of the implementation system in a way, that the simulation relation holds between the starting and ending states of the machines. The difference is in the way how these sequences are constructed: in the forward simulation the construction starts from the starting state, and in the backward simulation - from the ending state of the implementation system.

2.3 Forward Simulation

The simulation between I/O automata requires that for every step of the implementation system there exists an execution fragment of the specification machine, s.t.

- the trace of the fragment equals to the trace of the step of the implementation machine and,
- the simulation relation between the two systems holds after the step.

Let I and S be I/O automata. Then the *simulation relation* (or *coupling invariant*) between I and S is a binary relation $R \subset states(I) \times states(S)$, s.t. :

◄ **Definition 2.2** Forward simulation

- if $t \in start(I)$, then there exists $s \in start(S)$ such that $(t, s) \in R$,
- if $(t, a, t') \in steps(A)$, $s \in states(S)$, and $(t, s) \in R$, then there exists a finite execution fragment ω of S s.t.

$$first(\omega) = s \wedge (t', last(\omega)) \in R \wedge trace(a) = trace(\omega).$$

The soundness of forward simulation is defined with respect to trace inclusion.

Theorem 2.1 (Soundness of forward simulation). *Let there exist a forward simulation R between I/O automata I and S. Then all traces of I are included into the traces of S i.e.,*

$$traces(I) \subseteq traces(S).$$

Proof. Versions of the proof for the soundness of forward simulation appear in a variety of papers e.g., in [LT87, Sta86]. We omit it here. □

From theorem 2.1 it follows that any output produced by the implementation automaton is also produced by the specification automaton, under the condition that both automata are provided with the same inputs. In other words, execution of any sequence of external actions in implementation automaton is equivalent to execution of the same sequence on

the specification one. Hence, specification resembles any possible behaviour of the implementation.

Note, that even though forward simulation is sound with respect to trace inclusion, it is not complete. There exist automata, such that the traces of one are included among those of the other, but for which no forward simulation can be constructed [1].

Note also, that the existence of simulation is not sufficient to express the notion of correct implementation in general, because it does not rule out trivial implementations, which do nothing. Hence, the simulation can only show that *if* the implementation system does something, than this behaviour is correct.

In this thesis we use forward simulation as a technique for proving correctness of hardware virtualization in Chapters 8 and Chapter 9. We also use forward simulation as the base technique for step-by-step simulation when stating and proving reduction theorems in Chapter 4 and for compiler correctness theorems in Chapters 5 and 7.

[1] Even though forward simulation is incomplete in general, combinations of forward and backward simulations can be shown to be complete [LV92]. The completeness for some other types of simulation have also been shown. For instance, the completeness of refinement extended with history and prophecy variables is stated in [AL91].

CHAPTER 3

Abstract Hardware Model

3.1 Multicore x64 Hardware Model

3.2 Instruction Core - Memory Core Interface

3.3 Caches, Store Buffers, and Main Memory

3.4 Translation Lookaside Buffer

3.5 Memory Core

3.6 Instruction Automaton

Correctness of virtualization code is normally established by showing simulation between the actions performed by the code, and the respective steps of the abstract hardware machine implemented by the code [ACH+10, AHPP10]. In this setting, the choice of the proper hardware model is crucial. The model has to be small and abstract enough to make arguing about it feasible, especially in the context of an automatic verification. From the other side, this model has to capture all the hardware features, important for virtualization.

Moreover, when verifying system software one has to deal with the hardware features generally invisible on the pure C code level. For instance, this includes presence of the hardware MMU and the weak memory model of the real hardware machine [DPS09].

In this chapter we present an abstract model of the x64 hardware. Every processor in our model consists of two communicating I/O automata, where one automaton is responsible for memory and TLB accesses and the other one performs instruction execution. We model in detail only the first automaton, while leaving the second one as a “black box”. As the base for our model, we used the full abstract model of x64 hardware presented in [Deg11].

Our goal in this chapter is to define a (hopefully) sound model, which can simulate the TLB- and memory-related part of the x64 hardware defined in [Int11, Adv11a, Int07] and at the same time be small enough to be used for simulation proofs in a mechanical program verifier. In order to achieve this goal, we

1. argue only about the components of the hardware architecture which affect the behaviour of the memory subsystem (including TLB);
2. define a set of (software) rules, under which we can reduce the model and make certain components invisible (e.g., store buffers and caches);
3. support only a subset of x64 hardware features, comprehensive enough to describe the behavior of the memory subsystem of the real hardware in certain execution modes, but not covering all details of the hardware instruction set architecture. For instance, we support only long addressing mode (we do not argue about legacy addressing modes). Also we do not provide support for large memory pages and global page translations.

The simplifications stated above allow us to design a model tiny enough to perform formal automated proofs with it, while it still remains a realistic model of the (subset of) x64 hardware features.

The model we aim at should support reasoning about three types of operations:

1. steps of the memory management unit, which include traversal of SPTs, caching translations in TLB and performing address translations (later in this thesis we refer to those operation as *TLB steps*),
2. execution of accesses to the memory by the processor core,
3. execution of TLB controlling instructions s.t. TLB invalidation and writing to certain control registers, and
4. execution of a switch from the hypervisor mode to the guest mode and vice versa.

3.1 Multicore x64 Hardware Model

Degenbaev [Deg11] in his attempt to formalize the instruction set of the x64 architecture splits the hardware model into two disjoint parts: a nondeterministic abstract hardware, which includes memory, interrupt controllers, and devices, and a deterministic processor core executing instruction. The interface between these components is established by a set of rules, which describe how the processor core may interact with the memory system. For instance, if a processor needs to read the data from the memory, it issues a request and waits until this request is served by the memory system. The order of the requests issued by the processor core to other components of the hardware system depends on the order of instructions, executed in the core.

The multicore x64 hardware according to [Deg11] consists of the memory system, local APIC controllers, IPI controllers, external devices and processor cores. In the frame of this thesis we assume that external devices do not write to the memory regions where the code and the data of the hypervisor are

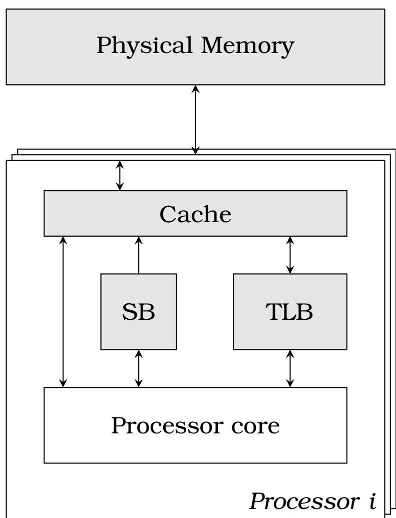

Figure 3.1: Hardware model: abstract view.

located. Hence, we abstract the devices away from the model. Moreover, we assume that the interrupts either never occur or do not affect the parts of the model we are interested in.

With the assumptions stated above we reduce the hardware model to contain only the core and the (nondeterministic) memory system (since we never use IPIs in our code and do not support interrupts we don't need to argue about the IPI controller).

The memory system of a multi-core machine consists of a shared physical memory and of the modules local to processor cores: caches, store buffers, load buffers, and TLBs. The data and code caches of real CPUs are modelled by processor-local abstract caches. TLBs are the components we are particularly interested in and are used for traversing page tables and producing address translations for memory accesses. Store buffers collect the store requests from the core to the caches/physical memory. Due to the delay, introduced by the store buffers, the processors may observe loads of the data before the old stores are completed. The load buffers produce a similar effect on the read requests from the processor core to the memory. More precisely, the load buffers non-deterministically pre-fetch data and instructions from the caches/ physical memory, which allows to model out-of-order/speculative instruction execution (Intel and AMD manuals [Int11, Adv11a] do not specify how exactly the speculative execution is done, which makes it non-deterministic by its nature).

Since we do not model instruction execution in details, we do not necessarily need to argue about the content of the load buffers explicitly and can incorporate them into the abstract core. We allow the core to fetch data non-deterministically. This simulates the pre-fetching behaviour of the load buffers. The very abstract view of the hardware machine we have is presented in Figure 3.1.

We model the hardware as a closed system (i.e., an automaton without inputs or outputs), which itself consists of two communicating I/O automata:

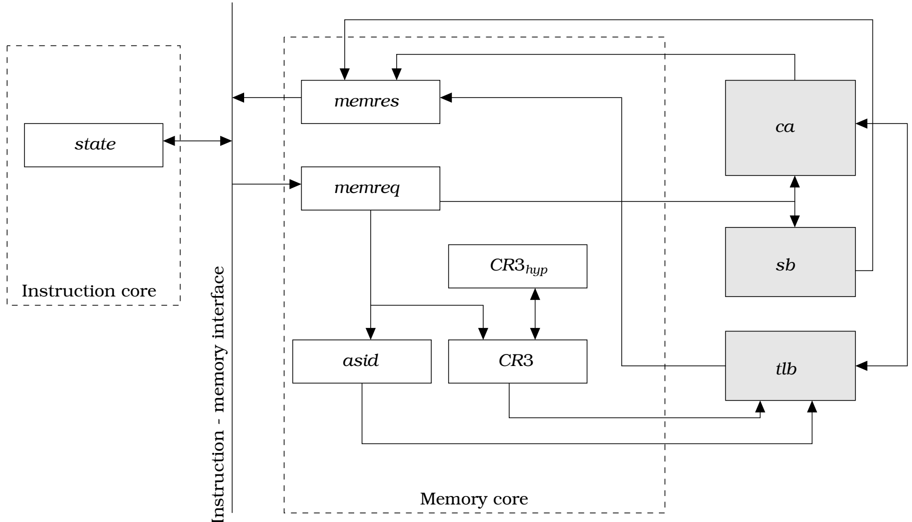

Figure 3.2: Data flow between components of processor i.

- the automaton responsible for memory accesses (which we later refer to as the *memory automaton*). As an input it gets a request for a memory access and provides the result of the memory access as an output. By a "memory access" here we understand not only memory reads/ writes, but also updates of memory-related registers and TLB controlling instructions. The state of this automaton consists of the main memory, caches, store buffers, TLBs, and certain parts of the abstract core (buffers and registers). We call the part of the core responsible for memory-related operations as *memory core*. We model the memory automaton in full detail, including the internal state and all possible internal actions,
- the automaton responsible for instruction execution (*instruction automaton*). This automaton issues memory requests and performs internal steps based on the input from the memory automaton. We do not model in detail internal state and internal steps of this automaton, but instead introduce an uninterpreted state and step function, which calculates the new internal state based on the current state and the outputs provided by the memory automaton. We refer to the part of the processor core responsible for instruction execution as *instruction core*.

In the closed hardware model input actions of one automaton are at the same time output actions of the other.

The configuration of the memory automaton is formally defined as a record of the following type:

Definition 3.1 ► Hardware state (memory automaton)

$$\begin{aligned} MemHw \stackrel{\text{def}}{=} [&p \in Pid \mapsto MemCore,\\ &mm \in Memory,\\ &ca \in Pid \mapsto Cache,\\ &sb \in Pid \mapsto SB,\\ &tlb \in Pid \mapsto Tlb],\end{aligned}$$

where $Pid \subset \mathbb{N}$ denotes the set of unique processor identifiers. Note, that component p here denotes only the memory managing part of the processor core.

The configuration of the instruction automaton is defined as a collection of memory automata of all cores:

$$InstrHw \stackrel{\text{def}}{=} [p_i \in Pid \mapsto InstrCore]$$

◀ **Definition 3.2**
Hardware state (instruction automaton)

The full hardware configuration then consists of the state of the instruction automaton, and the state of the memory automaton:

$$Hardware \stackrel{\text{def}}{=} [h_m \in MemHw, h_i \in InstrHw].$$

◀ **Definition 3.3**
Hardware state

To simplify the notation when talking about the state of the full hardware model $h \in Hardware$, we use the following shorthands for $x \in \{p, mm, ca, sb, tlb\}$ and for $y = p_i$:

$$h.x \stackrel{\text{def}}{=} h.h_m.x$$
$$h.y \stackrel{\text{def}}{=} h.h_i.y.$$

Moreover, we refer to a particular component of the hardware/processor state by writing the name of the component and the index of the processor. For instance, we write $ca[i]$ instead of $h.ca[i]$, and $memreq[i]$ instead of $h.p[i].memreq$, if the configuration h is clear from the context.

The detailed view on the communication between components and subsystems of our hardware model is presented in Figure 3.2.

Every step of the hardware transition system is parametrized with the index of the component making a step. When we need to identify the acting processor in a step $h \xrightarrow{a} h'$, we use the following shorthand:

$$pid(a) = i \stackrel{\text{def}}{=} (a \text{ is a step of component } i).$$

◀ **Definition 3.4**
Step of component i

Before we proceed with the formal definition of individual components of the hardware model, we summarize all the restrictions of the real hardware under which our model is valid.

3.1.1 The Scope of the Model

We define our abstract hardware model under the following assumptions:

- the interrupts either never occur, or are invisible to the program running on the hardware,
- the memory regions we argue about all belong to a memory system, memory mapped devices are not modelled[1],

[1] To integrate memory-mapped devices to our hardware model, one would have to treat the device mapped memory separately from the normal memory regions. For instance, our cache reduction theorem (where all addresses are made always cacheable) would not be applicable for the device mapped memory. As a result, caches would have to stay always visible for the range of

- all memory accesses are done in the long addressing mode,
- memory paging is always enabled ($CR0.PG$ bit is always set); as a consequence of this, segmentation is disabled,
- caching is always enabled ($CR0.CD$ bit is never set),
- write protection is always enabled ($CR0.WP$ bit is always set),
- large and global memory pages are not supported (the page tables should be set up accordingly),
- PAT and MTRR registers (responsible for the memory type computations) are never changed after initialization.

In the subsequent sections of this chapter we define the transition relation for the abstract hardware machine. Every transition from the transition system consists of a guard and effect of the transition. If the guard of the transition is satisfied, it can be triggered at any time nondeterministically. A triggered transition can modify the state of one or more components of the abstract machine.

3.1.2 Addressing Convention

As long as all memory accesses in our model are quadword (8-byte) aligned we normally argue about quadword addresses. Yet, sometimes we also have to argue about byte addresses (e.g., when defining byte-wise ownership discipline). The size of the memory page in our model is fixed to 4Kb. Thus, we consider six types of memory addresses:

- physical/virtual quadword addresses, which we simply call addresses later on,
- physical/virtual page addresses, which we call *page frame numbers* (PFNs) or *base addresses*, if talking about page addresses of page-aligned data structures, and
- physical/virtual byte addresses.

The x64 architecture in the long addressing mode supports physical (quadword) addresses up to 49 bits long (52 bits for byte addresses) and physical page frame numbers up to 40 bits long. The length of the virtual addresses depends on the addressing mode of the CPU. In the long addressing mode it is limited to 45 bits (48 bits for byte addresses) and virtual page frame numbers are limited to 36 bits. Since we model only a subset of the features of the real hardware, we argue only about a subset of physical memory addresses belonging to the physical memory (leaving the remaining ones e.g., for memory mapped devices). At the same time, in the abstract model to simplify TLB reduction we want to have physical and virtual addresses of the same length. Moreover, we want these addresses to be of the same length as the addresses in the C-IL semantics, introduced in Chapter 5. Hence, we define both sets of virtual and physical byte addresses as subsets of 64-bit integers; physical and virtual quadword addresses as subsets of 61-bit integers; physical and virtual

addresses which is assigned to devices. Alternatively, one could require all the accesses to device memory to be done in an uncacheable memory mode, which would extend the cache reduction theorem to be applicable for the devices.

PFNs as subsets of 52-bit integers:

$$\begin{aligned}
\mathbb{B}^{bpa} &\subset \mathbb{B}^{64},\\
\mathbb{B}^{qpa} &\subset \mathbb{B}^{61},\\
\mathbb{B}^{pfn} &\subset \mathbb{B}^{52},\\
\mathbb{B}^{bva} &\subset \mathbb{B}^{64},\\
\mathbb{B}^{qva} &\subset \mathbb{B}^{61},\\
\mathbb{B}^{vpfn} &\subset \mathbb{B}^{52}.
\end{aligned}$$

Given the sets $\mathbb{B}^{pfn}$ and $\mathbb{B}^{vpfn}$, we construct sets of virtual byte/quadword addresses by extending virtual PFNs with all possible page indices and byte indices using the following functions:

◀ **Definition 3.5** Address conversions

$$\begin{aligned}
&qword2bytes(pa \in \mathbb{B}^{61}) \in 2^{\mathbb{B}^{64}},\\
&pfn2qwords(pfn \in \mathbb{B}^{52}) \in 2^{\mathbb{B}^{61}},\\
&pfn2bytes(pfn \in \mathbb{B}^{52}) \in 2^{\mathbb{B}^{64}},\\
&qword2bytes(pa) \stackrel{\text{def}}{=} \lambda bpa : \exists a \in \mathbb{B}^{3} : bpa = pa \circ 0^{3} + a,\\
&pfn2words(pfn) \stackrel{\text{def}}{=} \lambda pa : \exists a \in \mathbb{B}^{9} : pa = pfn \circ 0^{9} + a,\\
&pfn2bytes(pfn) \stackrel{\text{def}}{=} \lambda bpa : \exists a \in \mathbb{B}^{12} : bpa = pfn \circ 0^{12} + a.
\end{aligned}$$

To make sure that addresses from the sets $\mathbb{B}^{pfn}$ and $\mathbb{B}^{vpfn}$ do not exceed the architecture limit, one has to enforce at least the following restrictions on these sets:

$$\begin{aligned}
&\forall a \in \mathbb{B}^{52} : a \in \mathbb{B}^{pfn} \implies a[51:40] = 0^{12},\\
&\forall a \in \mathbb{B}^{52} : a \in \mathbb{B}^{vpfn} \implies a[51:36] = 0^{16}.
\end{aligned}$$

Later in this thesis (starting from Chapter 4) we argue only about physical addresses which are identity-mapped by hypervisor page tables. Hence, we will consider sets of virtual and physical addresses to be equal, i.e.,

$$\mathbb{B}^{pfn} = \mathbb{B}^{vpfn}.$$

For a physical address $pa \in \mathbb{B}^{qpa}$ and for a virtual address $va \in \mathbb{B}^{qva}$ we use shorthands $pa.pfn$ and $va.vpfn$ to denote the page frame numbers of the addresses:

$$\begin{aligned}
pa.pfn &\stackrel{\text{def}}{=} 0^{12} \circ pa[48:9],\\
va.vpfn &\stackrel{\text{def}}{=} 0^{16} \circ va[44:9].
\end{aligned}$$

For an address $a \in \mathbb{B}^{qpa} \cup \mathbb{B}^{qva}$ shorthand $a.off$ denotes a *page offset*:

$$a.off \stackrel{\text{def}}{=} a[8:0].$$

We decompose a virtual page frame number $vpfn \in \mathbb{B}^{vpfn}$ into *page indices* 9 bits long each:

$$vpfn = 0^{16} \circ vpfn.px[4] \circ vpfn.px[3] \circ vpfn.px[2] \circ vpfn.px[1].$$

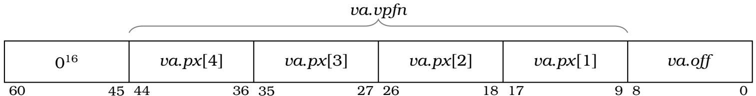

Figure 3.3: Decomposition of a virtual address $va \in \mathbb{B}^{qva} \subset \mathbb{B}^{61}$.

For a virtual address $va \in \mathbb{B}^{qva}$ we use a shorthand $va.px[i], i \in [1 : 4]$ to identify page index i of $va.vpfn$ and $va.off$ to identify page offset (Figure 3.3).

3.2 Instruction Core - Memory Core Interface

In this section we define the interface between the memory and the instruction automata of our hardware model.

3.2.1 Requests and Replies

We consider three basic types of memory accesses: a memory read, a memory write, and an atomic compare exchange operation[2]. The compare exchange is implemented as an atomic read-modify-write access, which requires store buffers to be flushed before and after execution of the instruction. A memory write can either be a regular write (not sequentially consistent) or a locked write, which guarantees total ordering of stores by flushing the store buffer.

Definition 3.6 ► Memory access

$$MemAcc \stackrel{\text{def}}{=} \{read, write, atomic\text{-}cmpxchng, locked\text{-}write\}.$$

Another type of operations, which can be coming from the instruction automaton, is a TLB controlling request. This includes an address invalidation and a move to the $CR3$ register. Note, that we consider a move to $CR3$ to be a TLB controlling operation, because the side effect of this action is a TLB flush performed in the currently active address space:

Definition 3.7 ► TLB request

$$TlbReq \stackrel{\text{def}}{=} \{invlpg\text{-}asid, mov2cr3\}.$$

A complete TLB flush (across all address spaces) can not be requested explicitly, but is rather performed during a VMRUN execution if an appropriate bit in the memory request buffer is set.

The number of parameters possibly passed by the instruction core to the memory automaton differs depending on what mode is currently active (hypervisor or guest mode). To denote this distinction we split the memory request into the part which is used in both modes and into the part which is used solely in hypervisor mode.

[2] In the x64 architecture a memory accessing instruction is made atomic by adding the `lock` instruction prefix [Adv11b].

The following data type collects request parameters which are used in both modes:

$$MemReqMain \stackrel{\text{def}}{=} [active \in \mathbb{B}, va \in \mathbb{B}^{qva}, r \in Rights,\\ data \in \mathbb{B}^{64}, mask \in \mathbb{B}^{8},\\ cmp\text{-}data \in \mathbb{B}^{64},\\ type \in TlbReq \cup MemAcc \cup \{vmexit, vmrun\},\\ pf\text{-}flush\text{-}req \in \mathbb{B}].$$

◀ **Definition 3.8**
Main parameters of a request

Type *Rights* is defined in Section 3.4.1.

A request to the memory system from external environment is then modelled with the following data type:

$$MemReq \stackrel{\text{def}}{=} [main \in MemReqMain,\\ asid \in \mathbb{N},\\ cr3in \in RegCr3,\\ asidin \in \mathbb{N},\\ complete\text{-}flush \in \mathbb{B}\\ inject\text{-}data \in InjectData].$$

◀ **Definition 3.9**
Request to the memory subsystem

To simplify notation later in this thesis we write *req.x* instead of *req.main.x* when referring to the main parameters of the memory request $req \in MemReq$.

The parameters of the request $req \in MemReq$ have the following meaning:

- *req.type* - the type of the request,
- *req.active* - the flag denoting that the request is pending,
- *req.va* - the virtual address in case if the request is a memory access,
- *req.r* - the access rights in case if the request is a memory access,
- *req.data* - the data to be written to the memory in case if the request is a memory write, or a compare-exchange,
- *req.mask* - the byte mask in case if the request is a memory read, memory write or a compare-exchange. In case of a write or a compare-exchange byte i is written to the memory only if $req.mask[i]$ equals 1,
- *req.cmp-data* - the data for the comparison in case if the request is a compare-exchange,
- *req.pf-flush-req* - an internal bit, which is used to denote that a request for TLB invalidation after a page fault is raised. This flag is controlled internally by the memory core and is ignored if *req.active* bit is set. For details on how we use this control flag see Section 3.5.1,
- *req.asid* - the address space identifier (ASID) in which TLB invalidation has to be done,
- *req.cr3in* - the value which has to be written to the $CR3$ register in case of a move to $CR3$ or a VMRUN request,
- *req.asidin* - the new value of ASID in case of a VMRUN request,
- *req.complete-flush* - the flag which denotes a request for the complete TLB flush in case of a VMRUN request,
- *req.inject-data* - the data which has to be injected into the memory

request/result buffers during VMRUN, emulating a successful INVLPG, a move to $CR3$, or triggering a page fault (for details refer to Section 3.5.3).

A reply from the memory subsystem either contains the fetched data (in case of a memory read request) or contains the information about the page fault in case if the fault was trigged. Analogously to the memory request we distinguish the main parameters of the memory reply (which are used in both hypervisor and guest mode) and auxiliary parameters used only in hypervisor mode.

In both modes a reply from the memory subsystem contains the fetched data (in case of a memory read access), *ready* flag, which indicates that the request is served, and the page fault data provided in case if the page fault is raised as a result of the memory access:

Definition 3.10 ► Main parameters of a memory reply

$$MemResMain \stackrel{\text{def}}{=} [data \in \mathbb{B}^{64}, ready \in \mathbb{B}, pf \in PfData].$$

The type of the page fault data *PfData* is defined in Section 3.4.4.

A memory reply in hypervisor mode additionally contains a *vmexit* flag indicating that a memory request on a processor running in guest mode resulted in a VMEXIT event (e.g., a page fault has occurred, which has to be intercepted by the hypervisor), and the parameters of the memory request which was active at the time of the VMEXIT event:

Definition 3.11 ► Reply of the memory subsystem

$$MemRes \stackrel{\text{def}}{=} [main \in MemResMain, vmexit \in \mathbb{B}, vmexit\text{-}memreq \in MemReqMain].$$

To simplify notation later in this thesis we write $res.x$ instead of $res.main.x$ when referring to the main parameters of the memory reply $res \in MemRes$.

3.2.2 External Actions.

The memory and the instruction automata communicate with each other by the interface consisting of a number of external (input and output) actions. Each input action of one automaton is at the same time an output action of the other automaton.

The only input action to the memory automaton (and respectively the only output action of the instruction automaton) is issuing of a request $req \in MemReq$ to the memory subsystem of the processor $i \in Pid$:

$$core\text{-}issue\text{-}mem\text{-}req(i, req).$$

The only output action of the memory automaton (and respectively the input action of the instruction automaton) is sending a reply with the result $res \in MemRes$ of the memory operation of the processor $i \in Pid$:

$$core\text{-}send\text{-}mem\text{-}res(i, res).$$

We give semantics for these (external) actions separately for the memory automaton (Section 3.5) and for the instruction automaton (Section 3.6). In the transition system of the full hardware model, the effect and the guard of

these steps is defined as a conjunction of effects and guards of the instruction and memory automata.

Now we proceed with defining configurations and individual transitions of every component of our hardware model.

3.3 Caches, Store Buffers and Main Memory

We model physical memory as a map from quadword physical addresses to bit strings 8 bytes long:

$$Memory \stackrel{\text{def}}{=} \mathbb{B}^{qpa} \mapsto \mathbb{B}^{64}.$$

◀ **Definition 3.12** Physical Memory

In order to model byte-wise operations with the quadword addressable memory, including the update of selected bytes in the quadword and forwarding of selected bytes from the store buffer, we introduce the function

$$combine(old \in \mathbb{B}^{64}, (new \in \mathbb{B}^{64}, mask \in \mathbb{B}^{8})) \in \mathbb{B}^{64},$$
$$combine(old, (new, mask)) \stackrel{\text{def}}{=} data, \text{ where}$$
$$\forall i \in \mathbb{N}_{64} : data[i] = \begin{cases} new[i] & mask[\lfloor i/8 \rfloor] \\ old[i] & \text{otherwise.} \end{cases}$$

◀ **Definition 3.13** Combining quadwords

If we want to refer to byte $i \in [0:7]$ of quadword $data \in \mathbb{B}^{64}$ we use the following function:

$$byte_i(data \in \mathbb{B}^{64}) \in \mathbb{B}^{8},$$
$$byte_i(data) \stackrel{\text{def}}{=} data[8 \cdot (i+1) - 1 : 8 \cdot i].$$

◀ **Definition 3.14** Extracting a byte

Since we do not consider devices, we assume that reads from the memory do not have side effects. Accesses to the physical memory are performed through the following interface:

$$read_{\surd}(mm \in Memory, pa \in \mathbb{B}^{qpa}) \in \mathbb{B},$$
$$write_{\surd}(mm \in Memory, pa \in \mathbb{B}^{qpa}, data \in \mathbb{B}^{64}) \in \mathbb{B},$$
$$read(mm \in Memory, pa \in \mathbb{B}^{qpa}, mask \in \mathbb{B}^{8}) \in \mathbb{B}^{64},$$
$$write(mm \in Memory, pa \in \mathbb{B}^{qpa}, data \in \mathbb{B}^{64}, mask \in \mathbb{B}^{8}) \in Memory.$$

Domains of memory read and write operations denote whether the main memory is readable/writable at the time of the request. In the full hardware model the instruction automaton might want to perform a series of memory accesses knowing that no other processors will access the memory in between these accesses. This behaviour can be modelled by introducing a global lock for the memory and by allowing memory accesses to complete only when this lock is free or is acquired by the processor performing a memory operation [Deg11]. Since here we do not explicitly model the global lock, we leave the functions $read_{\surd}$ and $write_{\surd}$ undefined.

The results of read and write operation are defined in a straightforward

way:

Definition 3.15 ► Reading/writing main memory

$$read(mm, pa) \stackrel{\text{def}}{=} mm[pa],$$
$$write(mm, pa, data) \stackrel{\text{def}}{=} mm[pa \mapsto combine(mm[pa], (data, mask))].$$

The physical memory is connected via the common bus to a number of processor caches. Every memory access has a certain memory type associated with it, which determines how this memory access deals with caches.

3.3.1 Memory Types

The x64 architecture defines the following memory types:

- *UC* - Uncacheable: cache is bypassed and all accesses go directly to the memory, write-combining[3] and speculative reads are not allowed, memory accesses are strongly ordered;
- *WC* - Write-Combining: accesses are uncacheable, write-combining and speculative reads are allowed;
- *CD* - Cache-Disable: all accesses are uncacheable, on a cache hit the line is invalidated and written back to the memory;
- *WT* - Write-Through: writes update the physical memory independently of the state of the line in the cache, the line in the cache is updated on a write hit and is not cached in case of a write miss, reads are always cacheable;
- *WP* - Write-Protect: writes are uncacheable and a write hit invalidates the line, the reads are always cacheable;
- *WB* - Write-Back: all accesses are fully cacheable.

Formally, we define the set of memory types in the following way:

Definition 3.16 ► Memory type

$$MemType \stackrel{\text{def}}{=} \{UC, WC, CD, WT, WP, WB\}.$$

To distinguish cacheable memory types from uncacheable ones, we introduce the following function:

Definition 3.17 ► Cacheable memory

$$cacheable(mt \in MemType) \in \mathbb{B} \stackrel{\text{def}}{=} mt \in \{WT, WP, WB\}.$$

The type of a memory access is obtained by combining memory types for the virtual address and for the physical address of the access. The latter is defined by the Memory Type Range Registers (MTRR), which map ranges of physical addresses into memory types. The memory type of the virtual address is obtained during traversal of page tables by the MMU. Each page table entry contains an index to the Page Attribute Table (PAT), which maps 3-bit indices into memory types and is stored in the 64-bit PAT register.

In the scope of the thesis we assume that PAT and MTRR registers are never written during the program execution. Hence, we consider that PAT and MTRR

[3]Write-combining allows memory accesses to be reordered and grouped together.

memory type mappings are always fixed. We declare the functions, which map a PAT index and the physical base address into the memory type, and combine the two memory types into a single one:

$$\begin{aligned}
&\textit{pat-mt}(\textit{pat-idx} \in \mathbb{B}^3) \in \textit{MemType},\\
&\textit{mtrr-mt}(\textit{pfn} \in \mathbb{B}^{\textit{pfn}}) \in \textit{MemType},\\
&\textit{mt-combine}(\textit{mt}_1 \in \textit{MemType}, \textit{mt}_2 \in \textit{MemType}) \in \textit{MemType}.
\end{aligned}$$

3.3.2 Abstract Cache

The real x64 processor has a number of caches: L1, L2, L3 caches, separate caches for instruction and data. The hardware ensures that data in all these caches always stays consistent. Hence, we we can model all these caches as a single processor-local abstract cache.

In a multi-core system caches of different processors communicate via a certain protocol. This protocol maintains coherence between caches on all processors and tries to minimize the data flow between caches and the physical memory. In this thesis we do not define a specific cache coherence protocol, but rather consider abstract caches with a generic MOESI [SS86] communication protocol. Our generic protocol can be used to simulate different implementations of MOESI, for instance the one introduced and verified by Wolfgang J. Paul in [Pau11].

A cache line in an abstract MOESI cache can be in one of the following states:

- E - Exclusive: the line is present only in the current cache and is *clean* (i.e., it is equal to the content of the main memory if the user hasn't mixed cacheable/uncacheable memory types for this line),
- M - Modified: the line is present only in the current cache and is *dirty*,
- O - Owned: the line might be present in other caches and might be dirty; the current cache is the owner of this line, i.e., it is responsible for writing this line back to the memory and for sending this line to other caches if requested,
- S - Shared: the line might be present in other caches and might be dirty; the current cache is not the owner of the line and does not need to write it back to the memory or send it to other caches,
- I - Invalid: the line is invalid.

The abstract cache maps a physical address to the line data (64 bit string) and to the line state:

$$\textit{Cache} \stackrel{\text{def}}{=} [\textit{data} \in \mathbb{B}^{qpa} \mapsto \mathbb{B}^{64}, \textit{state} \in \mathbb{B}^{qpa} \mapsto \{M, O, E, S, I\}].$$

◄ **Definition 3.18**
Abstract cache

Cache Interface. The other components of the x64 machine communicate with the caches and the main memory via the following interface:

$$read_{\surd}(ca, mm, i, pa, mt) \in \mathbb{B}$$
$$read(ca, mm, i, pa, mt) \in \mathbb{B}^{64}$$
$$write_{\surd}(ca, mm, i, pa, mt, data, mask) \in \mathbb{B}$$
$$write(ca, mm, i, pa, mt, data, mask) \in (Cache, Memory),$$

where $ca \in Pid \mapsto Cache$, $mm \in Memory$, $p \in Core$, $i \in Pid$, $pa \in \mathbb{B}^{qpa}$, $mt \in MemType$, and $data \in \mathbb{B}^{64}$.

When the core performs an access to the cache, this access is either handled by the cache itself or is forwarded to the physical memory. For a read access to go through, a number of conditions have to hold:

- if the memory type of the access is cacheable the data in the cache for the requested line has to be valid,
- if the memory type of the access is uncacheable, then the data has to be readable from the main memory. Moreover, for the "Cache-Disable" memory type the line in the local cache has to be invalid:

Definition 3.19 ► Cache read domain

$$read_{\surd}(ca, mm, i, pa, mt) \stackrel{\text{def}}{=} \begin{cases} ca[i].state[pa] \neq I & cacheable(mt) \\ read_{\surd}(mm, pa) \wedge ca[i].state[pa] = I & mt = CD \\ read_{\surd}(mm, pa) & \text{otherwise.} \end{cases}$$

A read access to the cache is then handled in a straightforward way:

Definition 3.20 ► Cache read result

$$read(ca, mm, i, pa, mt) \stackrel{\text{def}}{=} \begin{cases} ca[i].data[pa] & \text{if } cacheable(mt) \\ read(mm, pa) & \text{otherwise.} \end{cases}$$

In case of a write access we proceed in the following way:

- if the memory type of the access is "Write-Back", then
  - check that the data is valid in the local cache and invalid in other caches,
  - write the new data to the cache line;
- if the memory type of the access is "Write-Through", then
  - check that the data is invalid in other caches,
  - if the line is valid in the local cache, update the data in the local cache,
  - forward the write request to the physical memory;
- if the memory type of the access is "Write-Protect", then
  - check that the line is invalid in other caches,
  - invalidate a line in the local cache (without writing the data back),
  - forward the write request to the physical memory;
- if the memory type of the access is "Cache-Disable", then
  - check that the line is invalid in the local cache,

- forward the write request to the physical memory;

- if the memory type of the access is "Uncacheable" or "Write-Combining", then forward the write request to the physical memory.

Note, that on a real x64 machine cache behaviour could be different, according to the particular choice of the coherence protocol. For instance, in case of a write hit, we could transmit the data from the master cache to other caches, instead of making the data in other caches invalid. This behaviour, however, has to guarantee data coherency between different caches at least for the case when the user doesn't perform accesses with different memory types to a single memory address.

Formally, we define the domain of a cache write access as follows:

◄ **Definition 3.21** Cache write domain

$$write_{\surd}(ca, mm, i, pa, mt, data, mask) \stackrel{\text{def}}{=} \begin{cases} ca[i].state[pa] \neq I \wedge \forall j \neq i : ca[j].state[pa] = I & mt = WB \\ write_{\surd}(mm, pa, data, mask) \wedge \forall j \neq i : ca[j].state[pa] = I & mt \in \{WT, WP\} \\ write_{\surd}(mm, pa, data, mask) \wedge ca[i].state[pa] = I & mt = CD \\ write_{\surd}(mm, pa, data, mask) & \text{otherwise.} \end{cases}$$

The result of a cache write operation is defined in the following way:

◄ **Definition 3.22** Cache write result

$$write(ca, mm, i, pa, mt, data, mask) \stackrel{\text{def}}{=} \begin{cases} (ca[.data[pa] \mapsto data', state[pa] \mapsto M], mm) & mt = WB \\ (ca[.data[pa] \mapsto data'], mm') & mt = WT \\ (ca[.state[pa] \mapsto I], mm') & mt = WP \\ (ca, mm') & \text{otherwise,} \end{cases}$$

where $mm' = write(mm, i, pa, data, mask)$ and $data' = combine(ca.data[pa], (data, mask))$. Note, that in case of a "write protect" memory access, the line gets invalidated without writing it back to the memory.

Transition relation. We allow an abstract cache to perform the following actions:

- nondeterministically fetch a line from the physical memory or from another cache,
- drop a clean line without writing it back to the memory,
- write back a dirty line to the memory,
- go from a shared to an exclusive state in case if all other caches do not have the line in a valid state,
- pass the ownership of a dirty line together with the content of the line to another cache.

A cache may fetch a line only if the physical address of a line has a cacheable memory type. The memory type of a physical base address $pfn \in \mathbb{B}^{pfn}$ is obtained from MTRR registers and from the TLB, which has walked the page tables and has determined that pfn is a translation of some virtual base

address $vpfn \in \mathbb{B}^{vpfn}$:

$$tlb\text{-}memtype(p \in Core, tlb \in Tlb, pfn \in \mathbb{B}^{pfn}) \in MemType \cup \{\bot\}.$$

The function *tlb-memtype*() is defined in Section 3.4.4.

Another source of the memory type information for a cache is the store buffer. If the store buffer contains a store to a cacheable memory address at the beginning of the queue, then the cache is allowed to fetch the line for this address. To denote the memory type of the first store in the store buffer we use the following function:

$$sb\text{-}memtype(sb \in SB, pa \in \mathbb{B}^{qpa}) \in MemType \cup \{\bot\}.$$

The function *sb-memtype* is defined in Section 3.3.3.

A cache may fetch a line from some other cache, if this cache has the line in a modified, exclusive, or owned state.

Definition 3.23 ► Fetching line from remote cache

label	$fetch\text{-}line\text{-}from\text{-}ca(i \in Pid, j \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].state[pa] = I,$ $ca[j].state[pa] \in \{M, O, E\},$ $mt = tlb\text{-}memtype(p[i], tlb[i], pa) \wedge cacheable(mt)$ $\vee\ mt = sb\text{-}memtype(sb[i], pa) \wedge cacheable(mt)$
effect	$ca'[i].state[pa] = S,$ $ca'[j].state[pa] = \begin{cases} S & ca'[j].state[pa] = E \\ O & \text{otherwise,} \end{cases}$ $ca'[i].data[pa] = ca[j].data[pa]$

If a given cache has a line in the invalid state and all other caches have this line either in a shared or in an invalid state, then the cache is allowed to fetch this line from the main memory. Note, that strictly speaking we do not need to fetch the data from the memory if at least one cache has it in a valid state. Yet, in case if no cache owns the line (i.e., when all caches have the line either in a valid or in a shared state) it is sometimes more efficient to get the data from the memory, rather than from other caches. For instance, this allows to implement a memory bus without additional arbitration between caches having data in a shared state [Pau11].

Definition 3.24 ► Fetching line from physical memory

label	$fetch\text{-}line\text{-}from\text{-}mm(i \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].state[pa] = I,$ $\forall j \neq i : ca[j].state[pa] \in \{S, I\},$ $mt = tlb\text{-}memtype(p[i], tlb[i], pa) \wedge cacheable(mt)$ $\vee\ mt = sb\text{-}memtype(sb[i], pa) \wedge cacheable(mt),$ $read_{\surd}(mm, pa)$
effect	$ca'[i].state[pa] = S,$ $ca'[i].data[pa] = read(ca, mm, i, pa, mt)$

When fetching the data from the main memory we always set a shared state for the cache line. If no other cache has the data for this line in a valid state, then

the cache may later change the state of a line to an exclusive one (Definition 3.27).

A cache may write back a line to the main memory, if this line is in a modified or in an owned state.

Definition 3.25 Writing-back cache line

label	$writeback\text{-}line\text{-}to\text{-}mm(i \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].state[pa] \in \{O, M\}$, $write_{\surd}(mm, pa, data, 1^8)$
effect	$ca'[i].state[pa] = \begin{cases} E & ca'[i].state[pa] = M \\ S & \text{otherwise}, \end{cases}$ $mm' = write(mm, i, pa, data, 1^8)$

A cache may drop any clean line without writing it back to the memory.

Definition 3.26 Dropping cache line

label	$drop\text{-}line(i \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].satate[pa] \in \{S, E\}$,
effect	$ca'[i].state[pa] = I$,

If a line in the cache is in the shared state, but no other cache has valid data for this line, then the cache may change the state of the line to an exclusive one.

Definition 3.27 Getting to an exclusive state

label	$make\text{-}exclusive(i \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].state[pa] \in \{O, S\}$, $\forall j \neq i : ca[j].state[pa] = I$
effect	$ca'[i].state[pa] = \begin{cases} M & ca'[i].state[pa] = O \\ E & \text{otherwise} \end{cases}$

A cache may pass the ownership of a cache line to another cache, which has the same line in a shared state.

Definition 3.28 Passing ownership of a cache line

label	$pass\text{-}ownership(i \in Pid, pa \in \mathbb{B}^{qpa})$
guard	$ca[i].state[pa] = O$, $ca[j].state[pa] = S$
effect	$ca[j].state[pa] = S$, $ca[i].state[pa] = O$

Cache coherence. The cache protocol defined above ensures, that the data is always consistent between different caches under an assumption that the user does not mix accesses with different memory types for a single address. In order to make cache consistency inductive and to perform cache reduction proof further in Section 4.2 we have to specify a number of invariants, giving the formal meaning to different states of cache lines [Pau11]:

1. if a cache has a line in one of the exclusive states (E or M), then all other caches have this line in an invalid state,
2. if a cache has a line in a clean exclusive state (E), then the data in this line is the same as the data in the memory for the address of the line,

3. if a cache has a line in state S, then either the data in this line is the same as the data in the memory or another cache has this line in state O,
4. if a cache has a line in state S and another cache has the same line in state S or O, then the data for the line in these caches is the same,
5. only one cache can have a given line in state O[4].

We formalize all these properties in the following invariant.

Invariant 3.29 ► Consistent caches

name	*inv-consistent-caches*$(ca \in Pid \mapsto Cache, mm \in Memory)$
property	$ca[i].state[pa] \in \{E, M\} \wedge j \neq i \implies ca[j].state[pa] = I,$ $ca[i].state[pa] \in E \implies ca[i].data[pa] = mm[pa],$ $ca[i].state[pa] = S \implies$ $\quad ca[i].data[pa] = mm[pa] \vee \exists j : ca[i].state[pa] = O,$ $ca[i].state[pa] = S \wedge ca[j].state[pa] \in \{S, O\} \implies$ $\quad ca[i].data[pa] = ca[j].data[pa],$ $ca[i].state[pa] = O \wedge j \neq i \implies ca[j].state[pa] \neq O$

In Section 4.2 we prove a simple lemma (Lemma 4.2), showing that once established this property is maintained by all cache transitions and memory accesses of a "Write-back" memory type.

3.3.3 Store Buffers

An x64 processor has several buffers, responsible for write optimizations and reordering. These include write buffers and write-combining buffers. We model all these buffers by a single (processor-local) store buffer, which accumulates and reorders writes from the core to the memory system.

A memory store is modelled with the following record type:

Definition 3.30 ► Memory Store

$$Store \overset{\text{def}}{=} [pa \in \mathbb{B}^{qpa}, data \in \mathbb{B}^{64}, mt \in MemType, mask \in \mathbb{B}^{8}].$$

We model a store buffer as a record, consisting of a queue of stores and store fences:

Definition 3.31 ► Store Buffer

$$SBItem \overset{\text{def}}{=} Store \cup \{SFENCE\},$$
$$SB \overset{\text{def}}{=} [buffer \in SBItem^{*}].$$

We introduce two auxiliary functions, which simplify data forwarding from a store buffer. The function *sb-cnt* is used to count the number of writes to a byte of a given physical address, which are present in the store buffer:

Definition 3.32 ► Counting writes in SB

$$sb\text{-}cnt(sb \in SB, pa \in \mathbb{B}^{qpa}, k \in \mathbb{N}_8) \in \mathbb{N},$$
$$sb\text{-}cnt(sb, pa, k) \overset{\text{def}}{=} |\{i \in \mathbb{N} \mid i < |sb.buffer| \wedge sb.buffer[i] \in Store \wedge sb.buffer[i].pa = pa \wedge sb.buffer[i].mask[k]\}|.$$

[4]Strictly speaking, we don't need this property for our cache reduction theorem. Yet, we leave it here, because it captures the intended meaning of the "owned" cache state.

The function *sb-data* is used to provide the data of the most recent stores to a physical address, which are still pending in the store buffer:

Definition 3.33 Recent store data

$$sb\text{-}data(sb \in SB, pa \in \mathbb{B}^{qpa}) \in \mathbb{B}^{64}$$

$$sb\text{-}data(sb, pa) \stackrel{\text{def}}{=} \begin{cases} 0^{64} & |sb.buffer| = 0 \\ combine(sb\text{-}data(sb[0 : |sb| - 2], pa), & |sb.buffer| > 0 \wedge s \in Store \\ \quad (s.data, s.mask)) & \wedge s = \mathbf{last}(sb.buffer) \wedge s.pa = pa \\ sb\text{-}data(sb[0 : |sb| - 2], pa) & \text{otherwise.} \end{cases}$$

Store buffer interface. The interface between the core and the store buffer consists from write and forwarding requests, and from auxiliary functions providing specific information about the state of the store buffer:

$$\begin{aligned} &pending\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}) \in \mathbb{B}, \\ &pending\text{-}byte\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}, byte \in \mathbb{N}_8) \in \mathbb{B}, \\ &pending\text{-}qword\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}) \in \mathbb{B}, \\ &is\text{-}empty(sb \in SB) \in \mathbb{B}, \\ &sb\text{-}memtype(sb \in SB, pa \in \mathbb{B}^{qpa}) \in MemType \cup \{\bot\}, \\ &forward(sb \in SB, pa \in \mathbb{B}^{qpa}) \in (\mathbb{B}^{64}, \mathbb{N}_8), \\ &write(sb \in SB, store \in SBItem) \in SB. \end{aligned}$$

The first function is used to identify whether the SB has a pending store to at least one byte of the given quadword physical address:

Definition 3.34 Pending store

$$pending\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}) \stackrel{\text{def}}{=} \exists k \in \mathbb{N}_8 : sb\text{-}cnt(sb, pa, k) > 0.$$

Another function is used to identify whether the SB has a pending store to a particular byte of the quadword physical address:

Definition 3.35 Pending byte store

$$pending\text{-}byte\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}, byte \in \mathbb{N}_8) \stackrel{\text{def}}{=} sb\text{-}cnt(sb, pa, byte) > 0.$$

To denote, whether the store buffer contains valid data for the whole quadword, we use the following function:

Definition 3.36 Pending quadword store

$$pending\text{-}qword\text{-}store(sb \in SB, pa \in \mathbb{B}^{qpa}) \stackrel{\text{def}}{=} \forall k \in \mathbb{N}_8 : sb\text{-}cnt(pa, k) > 0.$$

The function *sb-memtype* is used to provide the memory type of the first store in the queue to the cache, if the address of the store matches the provided physical address *pa*:

Definition 3.37 SB memory type

$$sb\text{-}memtype(sb, pa) \stackrel{\text{def}}{=} \begin{cases} sb.buffer[0].mt & |sb.buffer| > 0 \wedge sb.buffer[0].pa = pa \\ \bot & \text{otherwise.} \end{cases}$$

Data forwarding is defined in a straightforward way with the help of the components *data* and *cnt* of the store buffer. The returned mask identifies which bytes of the quad-word are valid (i.e., data for them in SB is meaningful):

Definition 3.38 ► SB forwarding result

$$forward(sb, pa) \stackrel{\text{def}}{=} (sb\text{-}data(sb, pa), mask), \quad \text{where}$$
$$mask = \lambda k \in \mathbb{B}_8 : sb\text{-}cnt(sb, pa, k) \neq 0.$$

Before executing certain instructions (e.g., atomic or serializing instructions), the core need to know that the store buffer is empty:

Definition 3.39 ► Empty store buffer

$$is\text{-}empty(sb \in SB) \stackrel{\text{def}}{=} |sb.buffer| = 0.$$

After certain hardware events the store buffer gets flushed. We introduce a simple function, returning an empty store buffer:

Definition 3.40 ► Constructing empty store buffer

$$empty\text{-}sb() \in SB$$
$$empty\text{-}sb() \stackrel{\text{def}}{=} SB[buffer \mapsto \{\}].$$

The result of a store request from the core to the store buffer is defined as

Definition 3.41 ► SB write result

$$write(sb, store) \stackrel{\text{def}}{=} sb[buffer \mapsto buffer \circ store].$$

Transition Relation. A store buffer is allowed to nondeterministically reorder stores, to write the stores to the cache/physical memory, and to drop the leading store fence.

Reordering of stores can be applied to any of two adjacent stores, if one of them has a "Write-Combining" memory type, none of them is a store fence, and the stores write data to different physical addresses. This step models the behaviour of the write-combining buffer of the real hardware.

Definition 3.42 ► Reordering of stores

label	$reorder\text{-}stores(i \in Pid, j \in \mathbb{N})$
guard	$j < \lvert sb[i].buffer\rvert - 1,$ $sb[i].buffer[j] \neq SFENCE,$ $sb[i].buffer[j+1] \neq SFENCE,$ $sb[i].buffer[j].pa \neq sb[i].buffer[j+1].pa,$ $sb[i].buffer[j].WC \vee sb[i].buffer[j+1].WC$
effect	$sb'[i].buffer[j] = sb[i].buffer[j+1],$ $sb'[i].buffer[j+1] = sb[i].buffer[j]$

A store buffer is allowed to drop a leading store fence at any time.

◂ **Definition 3.43**
Dropping *SFENCE*

label	$\mathit{drop\text{-}sfence}(i \in \mathit{Pid})$
guard	$0 < \lvert sb[i].\mathit{buffer}\rvert$, $sb[i].\mathit{buffer}[0] = \mathit{SFENCE}$,
effect	$sb'[i].\mathit{buffer} = \mathbf{tl}(sb[i].\mathit{buffer})$,

A normal store item in the front of the queue may be committed to the cache/physical memory.

◂ **Definition 3.44**
Committing a store

label	$\mathit{commit\text{-}store}(i \in \mathit{Pid})$
guard	$0 < \lvert sb[i].\mathit{buffer}\rvert$, $\mathit{store} = sb[i].\mathit{buffer}[0]$, $\mathit{store} \neq \mathit{SFENCE}$, $\mathit{write}_{\surd}(ca, mm, i, \mathit{store.pa}, \mathit{store.mt}, \mathit{store.data}, \mathit{store.mask})$
effect	$sb'[i].\mathit{buffer} = \mathbf{tl}(sb[i].\mathit{buffer})$, $(ca', mm') = \mathit{write}(ca, mm, i, \mathit{store.pa}, \mathit{store.mt}, \mathit{store.data}, \mathit{store.mask})$

3.4 Translation Lookaside Buffer

The purpose of a TLB is to cache address translations done by the MMU and to reuse them later without performing additional memory accesses to page tables. A modern TLB caches not only address translations themselves, which could by considered as complete page table traversals, but also intermediate states of such traversals, which we call *walks*.

3.4.1 Page Table Walks

A page table walk models either an address translation or an intermediate state of the page table traversal. A walk, which represents an address translation, is called *complete* and a walk, which models an intermediate state of the page table traversal is called *partial*. We model a walk as a record, storing all the information necessary for performing a next step of the address translation (for a partial walk) or a result of the translation (for a complete walk):

◂ **Definition 3.45**
Page table walk

$$\mathit{Walk} \stackrel{\text{def}}{=} [l \in \mathbb{N}, \mathit{asid} \in \mathbb{N}, \mathit{vpfn} \in \mathbb{B}^{\mathit{vpfn}}, \mathit{pfn} \in \mathbb{B}^{\mathit{pfn}}, r \in \mathit{Rights}, \mathit{mt} \in \mathit{MemType}].$$

A set of rights $r \in \mathit{Rights}$ contains the requested permissions for a memory access, where *ex* represents a right to execute, *us* states for the user access, and *rw* for a write permission:

◂ **Definition 3.46**
Translation rights

$$\mathit{Rights} \stackrel{\text{def}}{=} [ex \in \mathbb{B}, us \in \mathbb{B}, rw \in \mathbb{B}].$$

To compare two sets of translation rights we overload the operator "less or equal":

Definition 3.47 ► Rights comparison

$$op(\leq)(r_1 \in Rights, r_2 \in Rights) \in \mathbb{B}$$
$$r_1 \leq r_2 \stackrel{\text{def}}{=} r_1.ex \leq r_2.ex \wedge r_1.us \leq r_2.us \wedge r_1.rw \leq r_2.rw$$

To perform a bitwise "and" operation on two sets of rights we use the standard operator:

Definition 3.48 ► Rights addition

$$r_1 \wedge r_2 \stackrel{\text{def}}{=} Rights[ex \mapsto r_1.ex \wedge r_2.ex, us \mapsto r_1.us \wedge r_2.us, rw \mapsto r_1.rw \wedge r_2.rw].$$

The fields of a page table walk $w \in Walk$ have the following meaning:

- $w.l$: the level of the page table walk; a walk with $w.l = 0$ is a complete walk and a walk with $w.l \in [1:4]$ is a partial walk,
- $w.asid$: the address space identifier (ASID) of the walk,
- $w.vpfn$: the page frame number of the virtual address to be translated,
- $w.pfn$: the physical page frame number of the next page table to be traversed (for a partial walk) or the physical page frame number of the resulting address translation (for a complete walk),
- $w.r$: the permissions of the walk,
- $w.mt$: the type of the memory where the next level page table is located (for a partial walk) or the type of the memory for the resulted virtual address (for a complete walk).

A complete walk is identified by the following predicate:

Definition 3.49 ► Complete walk

$$complete(w \in Walk) \in \mathbb{B} \stackrel{\text{def}}{=} w.l = 0.$$

To give the formal definition for operations on walks performed by the MMU (which we call TLB *steps*), we first have to define the format of the page tables and page table entries.

3.4.2 Page Tables and Page Table Entries

A single page table occupies one page (4Kb) and consists of 512 page table entries (PTEs), each of which is 64 bits long. The x64 architecture for correct memory translation in the long addressing mode requires page tables to form a graph, where each path has length 4.[5] The $CR3$ register points to the top-level (level 4) page table, which contains references to page tables of the next level (level 3). Page tables of level 1 are called *terminal* page tables and contain the mappings to physical addresses.

Definition 3.50 ► Page table

$$Pt \stackrel{\text{def}}{=} [0:511] \mapsto Pte$$

[5] Address translations for large pages, which are left out of the scope of the thesis, require less then 4 PTEs in a path.

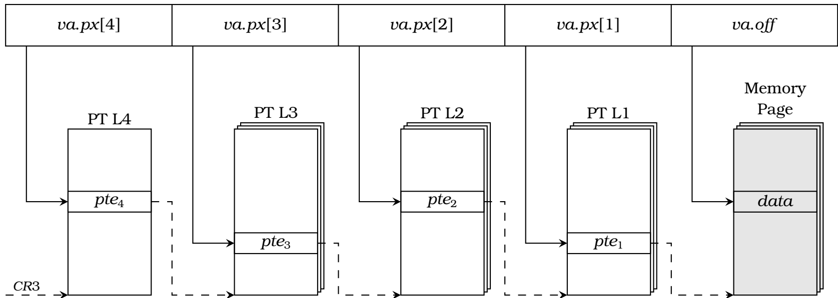

Figure 3.4: Selection of PTEs during address translation.

A page table entry in the long address translation mode is a bit string (or a union) 64 bits long:

$$Pte \stackrel{\text{def}}{=} \mathbb{B}^{64}.$$

During page table traversal, a page index $a.px[i], i \in [1:4]$ of a virtual address $a \in \mathbb{B}^{qva}$ is used to select a PTE from the i-th level page table (Figure 3.4).

For a page table with the (page aligned) base address $ba \in \mathbb{B}^{pfn}$, we obtain the (quadword aligned) address of the j-th PTE, where $j \in [0:511]$, in a straightforward way:

Definition 3.51 Base address of a PTE

$$pte\text{-}addr(pfn \in \mathbb{B}^{pfn}, j \in \mathbb{N}_{512}) \in \mathbb{B}^{pfn}$$
$$pte\text{-}addr(pfn, j) \stackrel{\text{def}}{=} (pfn \circ 0^9) + j.$$

The MMU accesses PTEs with the help of the following functions:

Definition 3.52 Reading/writing a PTE

$$pte\text{-}read(ca \in Cache, mm \in Memory, i \in Pid, w \in Walk) \in Pte,$$
$$pte\text{-}write(ca, mm, i, w, pte \in Pte) \in (Cache, Memory),$$
$$pte\text{-}read(ca, mm, i, w) \stackrel{\text{def}}{=} read(ca, mm, i, pa, w.mt, 1^8),$$
$$pte\text{-}write(ca, mm, i, w, pte) \stackrel{\text{def}}{=} write(ca, mm, i, pa, w.mt, pte, 1^8),$$

where $pa = pte\text{-}addr(w.pfn, w.vpfn.px[w.l])$ is the base address of the PTE.

The respective predicates $pte\text{-}read_{\surd}$ and $pte\text{-}write_{\surd}$ are defined in a straightforward way, using the read/write domains of the cache interface.

To simplify reasoning about fields of a PTE and to hide implementation details we introduce the function

$$abs\text{-}pte(pte \in Pte) \in AbsPte,$$

which converts binary representation of a PTE to an abstract representation,

where an abstract PTE is a record of the following type:

Definition 3.53 ► Abstract PTE

$$AbsPte \stackrel{\text{def}}{=} [p \in \mathbb{B}, a \in \mathbb{B}, d \in \mathbb{B}, r \in Rights, pfn \in \mathbb{B}^{pfn}, pat\text{-}idx \in \mathbb{B}^3, valid \in \mathbb{B}].$$

The fields of an abstract $pte \in Pte$ have the following meaning:

- $pte.p$: the present bit, denotes whether a given PTE has meaningful data;
- $pte.a$: the access bit, identifies whether MMU has already used the PTE for an address translation;
- $pte.d$: the dirty bit, identifies whether MMU has already used the PTE for a translation with the write request; is meaningful only for terminal page tables;
- $pte.r$: the access permissions, may restrict the set of non-faulty translations through this PTE (e.g., make write requests produce a page fault);
- $pte.pfn$: the page frame number of the next level page table (for a non-terminal page table) or a page frame number of the resulting address translation (for a terminal page table),
- $pte.pat\text{-}idx$: the index to the PAT table, identifying the memory type of the address stored in $w.pfn$;
- $pte.valid$: the flag indicating whether the reserved bits of the binary representation of this PTE have the allowed values, specified by the architecture.

Note, that due to the chosen restrictions on the address translation mode and features, we omit some of the fields of PTEs, specified by the x64 architecture (such as flags for global or large pages).

To convert an abstract PTE into a concrete one, we use the following function:

$$concrete\text{-}pte(pte \in AbsPte) \in Pte.$$

Since the PFN field of the concrete PTE is limited to 40 bits and the set $\mathbb{B}^{pfn}$ contains 52-bit strings, we have to do conversion by throwing away the leading 12 bits of the abstract PFN value[6].

3.4.3 TLB Model

We model a TLB as a set of walks:

Definition 3.54 ► Translation Lookaside Buffer

$$Tlb \stackrel{\text{def}}{=} 2^{Walk}.$$

In order to perform an address translation for a virtual address $va \in \mathbb{B}^{qva}$ with initial permissions $r \in Rights$, MMU initializes a walk w with $w.vpfn = va.pfn$ and $w.r = r$, and sets the $w.pfn$ field to point to the top-level page table. Then it performs a number of walk extensions, fetching page table entries

[6]When defining the set $\mathbb{B}^{pfn}$, one has to make sure that it only contains addresses which do not exceed the length defined by the architecture (see Section 3.1.2).

and updating the state of the walk. In the end, it either ends in a situation where walk extension is not possible anymore due to a page-fault situation or it produces a complete walk, which identifies a successful address translation.

Below we define all possible nondeterministic TLB steps, each of which is a part of the transition relation of the abstract hardware model.

Creating a walk. To start an address translation, the TLB first has to create a new walk with the initial parameters for page table traversal. The level of the new walk is set to the depth of translation, which in our case (for the long addressing mode) equals four. The physical base address of the top-level page table and the memory type of that address are calculated from the value of the *CR*3 register of the core.

◀ **Definition 3.55** Creating a walk

label	$\mathit{create\text{-}walk}(i \in \mathit{Pid}, w \in \mathit{Walk})$
guard	$w.l = 4$, $w.\mathit{asid} = \mathit{asid}[i]$, $w.r = \mathit{Rights}[\mathit{ex} \mapsto 1, \mathit{us} \mapsto 1, \mathit{rw} \mapsto 1]$, $\mathit{CR3}[i].\mathit{valid}$, $w.\mathit{pfn} = \mathit{CR3}[i].\mathit{pfn}$, $w.\mathit{mt} = \mathit{root\text{-}pt\text{-}memtype}(\mathit{CR3}[i])$
effect	$\mathit{tlb}'[i][w] = 1$

The structure of the *CR*3 register and the definition of *root-pt-memtype* are given in Section 3.5.

Note, that we do not fix the initial *vpfn* field for the new walk, allowing the TLB to start an address translation for any virtual address. By giving this freedom to the TLB we model speculative address translations.

Extending a walk. To extend a partial walk we use the field *pfn* of the walk together with the page index, obtained from the field *vpfn*, to fetch the next PTE in the page table traversal path (Figure 3.4). The fields of the PTE are used to calculate the new walk with the level of the original walk, decremented by one. The memory type of the new walk is obtained by combining memory types from the PAT and MTRR tables for the newly obtained physical PFN, which is either the resulting physical PFN of the translation (if this is the last level of walk extension) or the base address of the next level page table:

◀ **Definition 3.56** Walk extension

$$\mathit{wext}(w \in \mathit{Walk}, \mathit{pte} \in \mathit{AbsPte}, r \in \mathit{Rights}) \in \mathit{Walk}$$
$$\mathit{wext}(w, \mathit{pte}, r) \stackrel{\text{def}}{=} w[l \mapsto (w.l - 1), \mathit{pfn} \mapsto \mathit{pte.pfn}, r \mapsto r, \mathit{mt} \mapsto \mathit{mt}'],$$

where $\mathit{mt}' = \mathit{mt\text{-}combine}(\mathit{pat\text{-}mt}(\mathit{pte.pat\text{-}idx}), \mathit{mtrr\text{-}mt}(\mathit{pte.pfn}))$.

The number of conditions has to be met for the walk extension over a given PTE to be successful:

- access permissions of the walk being extended should be broad enough to satisfy the rights restrictions of the fetched PTE;
- the present bit has to be set in the PTE;
- the valid flag has to be set in the PTE;
- the access bit has to be set in the PTE;

- for the last level of the walk extension with the write permission the dirty bit has to be set in the PTE (PTE is terminal in this case);
- the walk to be extended has to be incomplete and should have at least the same rights as the new walk (we do allow the rights of a walk to be reduced arbitrarily during walk extension).

Violation of any of the first three conditions triggers a page fault during the walk extension:

Definition 3.57 ► Page faulty PTE

$$\begin{aligned} &\mathit{page\text{-}fault}(r \in \mathit{Rights}, \mathit{pte} \in \mathit{AbsPte}) \in \mathbb{B}, \\ &\mathit{page\text{-}fault}(r, \mathit{pte}) \stackrel{\text{def}}{=} \neg(r \leq \mathit{pte}.r) \vee \neg \mathit{pte}.p \vee \neg \mathit{pte}.\mathit{valid}. \end{aligned}$$

Note though, that for a page fault to be reported to the core, the walk chosen for the extension should be suitable for an address translation with the requested parameters (for details see Section 3.5.1).

The domain of the walk extension is then stated as follows:

Definition 3.58 ► Walk extension domain

$$\begin{aligned} &\mathit{wext}_{\surd}(w \in \mathit{Walk}, \mathit{pte} \in \mathit{AbsPte}, r \in \mathit{Rights}) \in \mathbb{B} \\ &\mathit{wext}_{\surd}(w, \mathit{pte}, r) \stackrel{\text{def}}{=} \neg \mathit{page\text{-}fault}(w.r, \mathit{pte}) \wedge \neg \mathit{complete}(w) \wedge \mathit{pte}.a \\ &\qquad \wedge (w.r.rw \wedge w.l = 1 \implies \mathit{pte}.d) \wedge r \leq w.r. \end{aligned}$$

If all conditions for the extension of a walk w are satisfied, and the PTE pointed to by w is readable, the TLB may perform a walk extension.

Definition 3.59 ► Extending a walk

label	$\mathit{extend\text{-}walk}(i \in \mathit{Pid}, w \in \mathit{Walk}, r \in \mathit{Rights})$
guard	$\mathit{tlb}[i][w] = 1$, $w.\mathit{asid} = \mathit{asid}[i]$, $\mathit{pte\text{-}read}_{\surd}(\mathit{ca}, \mathit{mm}, i, w)$, $\mathit{wext}_{\surd}(w, \mathit{pte}, r)$, $\mathit{pte} = \mathit{abs\text{-}pte}(\mathit{pte\text{-}read}(\mathit{ca}, \mathit{mm}, i, w))$, $w' = \mathit{wext}(w, \mathit{pte}, r)$
effect	$\mathit{tlb}'[i][w'] = 1$

During the walk extension we never add faulty walks to the TLB. This means that in order to report a page fault, TLB has to fetch a faulty PTE from memory. This allows to model silent rights granting in page tables i.e., when the user grants more rights in a PTE without a consequent TLB flush, and setting of present bit in a PTE without TLB flushing. In a real TLB the same behaviour is achieved by performing a re-walk of page tables in case of a page fault. Our model does not allow to (nicely) model the full traversal in case of a page fault. Thus, we stick to modelling only the last level of this traversal by not storing faulty walks in the TLB and by forcing the MMU to always fetch a faulty PTE from the memory.

Note, that in the real hardware machine the TLB is probably not allowed to store multiple complete walks for a given physical address. We consider a more general TLB model, where this restriction is not enforced. This allows us to use the same TLB model both for the host hardware and for the virtual hardware when we later prove correctness of the SPT algorithm. Our virtual

TLB might have multiple complete walks for a given physical address due to the fact that the virtual TLB contains the translated version of all the walks which could have been possibly added to the host TLB since the last flush.

Setting access and dirty bits. Before performing a walk extension, the MMU must set access and dirty bits in the PTE chosen for a walk extension. The MMU in this case fetches the entry, checks whether the entry is valid, updates access and dirty bits, and writes the entry back to the memory. All these actions are performed in one atomic step. The access bit is always set (for a valid PTE). The dirty bit is set only for a terminal PTE in case if the walk has the write permission, and the write is allowed by the PTE. The following function returns the updated PTE:

◄ **Definition 3.60** PTE with A/D bits set

$$pte\text{-}set\text{-}ad\text{-}bits(pte \in AbsPte, w \in Walk) \in AbsPte$$

$$pte\text{-}set\text{-}ad\text{-}bits(pte, w) \stackrel{\text{def}}{=} \begin{cases} pte[a \mapsto 1, d \mapsto 1] & w.r.rw \wedge w.l = 1 \wedge pte.r.rw \\ pte[a \mapsto 1] & \text{otherwise.} \end{cases}$$

The step of setting access and dirty bits is defined in the following way.

◄ **Definition 3.61** Setting access and dirty bits

label	$set\text{-}access\text{-}dirty(i \in Pid, w \in Walk)$
guard	$tlb[i][w] = 1$, $w.asid = asid[i]$, $\neg complete(w)$, $pte\text{-}read_{\surd}(ca, mm, i, w)$, $pte = abs\text{-}pte(pte\text{-}read(ca, mm, i, w))$, $pte.p$, $pte.valid$, $pte' = pte\text{-}set\text{-}ad\text{-}bits(pte, w)$, $pte\text{-}write_{\surd}(ca, mm, i, w, concrete\text{-}pte(pte'))$
effect	$(ca'[i], mm') = pte\text{-}write(ca, mm, i, w, concrete\text{-}pte(pte'))$

Dropping a walk. The MMU may nondeterministically drop any number of walks, present in the TLB, at any time.

◄ **Definition 3.62** Dropping a walk

label	$drop\text{-}walks(i \in Pid, walks \in 2^{Walk})$
guard	
effect	$tlb'[i].walks = \lambda w \in Walk : tlb[i][w] \wedge \neg walks[w]$

3.4.4 TLB Interface

The TLB interface provides the core with the ability to perform address translations and gives a limited control over the TLB state. Parameters of a TLB request are determined by the current state of the memory request buffer. For instance, if *memreq* buffer contains a request for memory read, write, or compare-exchange then the TLB is requested to either provide a successful translation or to signal a page fault.

To perform an address translation, the TLB nondeterministically selects a walk, suitable either for a successful or a faulty translation. Note, that there could be multiple walks, suitable for a particular translation, sitting in the TLB at the same time. In this case an arbitrary walk is chosen.

All address translations are performed only in the currently active address space and the ASID field of the request is ignored.

Successful address translation. For a successful address translation the chosen walk has to be complete and has to have the address space identifier, as well as the virtual base address, equal to the ones of the translation request. The walk should have access permissions not less than the rights of the request.

The following function denotes that a given walk in the TLB of a processor can be used for successful address translation:

Definition 3.63 ► Successful translation ready

$$tlb\text{-}transl\text{-}ready(memreq \in MemReqMain, asid \in \mathbb{N}, tlb \in Tlb, w \in Walk) \in \mathbb{B}$$

$$\begin{aligned} tlb\text{-}transl\text{-}ready(memreq, asid, tlb, w) \stackrel{\text{def}}{=}\ & tlb[w] = 1 \wedge complete(w) \\ & \wedge w.vpfn = memreq.va.vpfn \\ & \wedge w.asid = p.asid \\ & \wedge memreq.active = 1 \\ & \wedge memreq.type \in MemAcc \\ & \wedge memreq.r \leq w.r. \end{aligned}$$

Faulting address translation. For a page fault to be triggered, the TLB must contain a (non-faulty) partial walk and extension of this walk must produce a page fault i.e., one of the following conditions has to hold: the PTE for a walk extension is not present, it is not valid, or it has less rights than required by the translation. If a page fault is signaled, the TLB also provides the 4-bit code of the page fault. The following function computes the code of the page fault based on the rights of the translation access and on the present and valid fields of the PTE:

$$page\text{-}fault\text{-}code(r \in Rights, present \in \mathbb{B}, valid \in \mathbb{B}) \in \mathbb{B}^4.$$

In our model we allow the selected walk for a translation to have more rights, than the translation request. In this scenario, if we use the rights of the selected walk to check PTE for a page fault, we may produce page faults which should have never been triggered. Thus, for a rights-violation page fault we check the original rights of the issued request, rather than the rights of the chosen walk.

The following predicate denotes that a walk w can be used for triggering of a page fault over a given PTE (in the context where this function is used one

has to ensure that walk w points to this PTE):

$$\begin{aligned}
&tlb\text{-}fault\text{-}ready(memreq \in MemReqMain, asid \in ASID,\\
&\qquad tlb \in Tlb, pte \in AbsPte, w \in Walk) \in \mathbb{B},\\
&tlb\text{-}fault\text{-}ready(memreq, asid, tlb, pte, w) \stackrel{\text{def}}{=} tlb[w] = 1 \wedge \neg complete(w)\\
&\qquad \wedge w.vpfn = p.memreq.va.vpfn\\
&\qquad \wedge w.asid = asid\\
&\qquad \wedge memreq.active = 1\\
&\qquad \wedge memreq.type \in MemAcc\\
&\qquad \wedge memreq.r \leq w.r\\
&\qquad \wedge page\text{-}fault(memreq.r, pte).
\end{aligned}$$

◀ **Definition 3.64** Faulty translation ready

The page fault data, accumulated in the *memres* buffer in case of a page fault, contains the following information:

$$PfData \stackrel{\text{def}}{=} [fault \in \mathbb{B}, fault\text{-}code \in \mathbb{B}^4, r \in Rights, va \in \mathbb{B}^{qva}].$$

◀ **Definition 3.65** Page fault data

When a step of the memory core completes a memory access which is not causing a page fault only the *fault* bit of the *memres.pf* buffer has to be written. Yet, to simplify arguing about equality of outputs of memory automata when proving hardware virtualization, we set the whole *memres.pf* buffer to a dummy “zeroed” value (where only *fault* bit is meaningful):

$$\begin{aligned}
&no\text{-}page\text{-}fault() \in PfData,\\
&no\text{-}page\text{-}fault() \stackrel{\text{def}}{=} PfData[fault \mapsto 0, fault\text{-}code \mapsto 0,\\
&\qquad r.\{ex, us, rw\} \mapsto 0, va \mapsto 0].
\end{aligned}$$

◀ **Definition 3.66** No page fault

TLB flushing. We model four types of TLB flushes: a complete flush across all address spaces, a full flush in the running address space (performed as part of the move to *CR*3 register), a tagged address invalidation, and a flush in case of a page fault. Here we define the predicates, which are later used as guarding conditions in these steps:

$$\begin{aligned}
&tlb\text{-}empty\text{-}asid(tlb, asid) \stackrel{\text{def}}{=} \forall w \in Walk : tlb[w] \implies w.asid \neq asid,\\
&tlb\text{-}invalidated(tlb, vpfn, asid) \stackrel{\text{def}}{=} \forall w \in Walk :\\
&\qquad tlb[w] \wedge w.asid = asid \implies (w.vpfn \neq vpfn \wedge complete(w)),\\
&tlb\text{-}invalidated\text{-}pf(tlb, vpfn, asid) \stackrel{\text{def}}{=} \forall w \in Walk :\\
&\qquad tlb[w] \wedge w.asid = asid \implies w.vpfn \neq vpfn,
\end{aligned}$$

◀ **Definition 3.67** TLB flushing guards

where $tlb \in Tlb$ is the flushed TLB, $asid \in \mathbb{N}$ is the ASID in which flushing is performed, and $vpfn \in \mathbb{B}^{vpfn}$ is the invalidated address.

Additionally, we introduce functions, which return an empty TLB and an

invalidated TLB state:

Definition 3.68 ► Empty/invalidated TLB

$$empty\text{-}tlb() \stackrel{\text{def}}{=} \lambda w \in Walk : 0,$$
$$inval\text{-}tlb(tlb \in Tlb, vpfn, asid) \stackrel{\text{def}}{=} \lambda w \in Walk : tlb[w] \wedge w.asid \neq asid \wedge w.vpfn \neq vpfn \wedge complete(w),$$
$$pf\text{-}inval\text{-}tlb(tlb \in Tlb, vpfn, asid) \stackrel{\text{def}}{=} \lambda w \in Walk : tlb[w] \wedge w.asid \neq asid \wedge w.vpfn \neq vpfn.$$

Cache - TLB interface. In addition to providing address translations to the core, TLB is also used for calculating memory types of physical addresses. These memory types are used by caches to decide whether this memory region is cacheable or not. The following function[7] obtains the memory type of a given physical address. Note, that in every hardware state the function is defined only for a subset of physical addresses.

Definition 3.69 ► TLB memory type

$$tlb\text{-}memtype(p \in MemCore, tlb \in Tlb, pfn \in \mathbb{B}^{pfn}) \in MemType \cup \{\bot\}$$
$$tlb\text{-}memtype(p, tlb, pfn) \stackrel{\text{def}}{=} \begin{cases} w.mt & tlb[w] \wedge w.pfn = pfn \wedge w.asid = p.asid \\ & \wedge complete(w) \\ \bot & \text{otherwise} \end{cases}$$

3.5 Memory Core

The memory core is modelled as a record, containing the $CR3$ register and memory request/result buffers. We also introduce a register, containing the identifier of the address space (ASID) currently being active on the processor[8], and the register $CR3_{hyp}$, which is an auxiliary register storing the old value of $CR3$ when the processor performs a switch to the guest mode. When the processor switches back to the hypervisor mode, the value from $CR3_{hyp}$ is restored to $CR3$.

Definition 3.70 ► Memory Core

$$MemCore \stackrel{\text{def}}{=} [CR3 \in RegCr3, asid \in \mathbb{N}, memreq \in MemReq, memres \in MemRes, CR3_{hyp} \in RegCr3]$$

The $CR3$ register contains the base address of the top-level page table, the flags for the memory type of this address and the reserved bits, which we

[7] Actually, *tlb-memtype* is a relation, because TLB could contain multiple complete walks for a given address. Yet, we later restrict TLBs to contain only walks with "write-back" memory type, which makes *tlb-memtype* a well-defined function.

[8] On the x64 architecture with hardware virtualization extensions this register is not accessed explicitly and can be modified only by switching to and from guest mode.

abstract into a validity flag:

$$RegCr3 \stackrel{\text{def}}{=} [pfn \in \mathbb{B}^{pfn}, WT \in \mathbb{B}, CD \in \mathbb{B}, valid \in \mathbb{B}].$$

◄ **Definition 3.71**
*CR*3 Register

The following functions are used to convert a 64-bit string to an instance of the type *RegCr*3 and vice versa:

$$cr3\text{-}2\text{-}uint(cr3 \in RegCr3) \in \mathbb{B}^{64},$$
$$uint2cr3(a \in \mathbb{B}^{64}) \in RegCr3.$$

Flag *CD* denotes whether the top level page table is cacheable or not. Flag *WT* identifies a "Write-through" memory type when it is set, or a "Write-back" type when it is not set. The combined memory type of the top-level page table is calculated by the following function:

$$root\text{-}pt\text{-}memtype(CR3 \in RegCr3) \in MemType$$
$$root\text{-}pt\text{-}memtype(CR3) \stackrel{\text{def}}{=} mt\text{-}combine(pat\text{-}mt(0 \circ CR3.CD \circ CR3.WT), mtrr\text{-}mt(CR3.pfn)).$$

◄ **Definition 3.72**
Root PT memory type

Register *asid* contains the ID of the currently active address space. The TLB of the core may perform all its operations, with the exception of walk removal, only with the walks from the active address space. When the currently active ASID equals 0, we say that the processor is running in *hypervisor mode*. Otherwise, it is running in *guest mode*.

The memory request buffer contains the data only for a single pending TLB or memory request (per processor). In the real hardware there could be multiple pending TLB and memory requests at the same time. Since we do not aim at providing the details of instruction execution, we leave the request queues hidden inside the uninterpreted part of the core and assume that there exists some ordering of these requests[9].

External actions. Here we define the effect of external actions on the memory core. The effect on the instruction core we define in Section 3.6. The memory core accepts a request by writing it to the buffer *memreq*.

label	$core\text{-}issue\text{-}mem\text{-}req(i \in Pid, req \in MemReq)$
guard	
effect	$memreq'[i] = req,$

◄ **Definition 3.73**
Accepting a memory request

The result of the memory operation is sent from the memory core to the instruction core in case if the result in *memres* buffer is ready.

label	$core\text{-}send\text{-}mem\text{-}res(i \in Pid, res \in MemRes)$
guard	$memres[i].ready = 1,$ $res = memres[i]$
effect	$memres'[i].ready = 0$

◄ **Definition 3.74**
Sending a memory result

[9] For instance, one can order memory accesses based on their end time as shown in [Pau11].

3.5.1 Memory Accesses.

A request for a memory access may get served if there exists a walk w which can be used for successful address translation. After the request is served, we clear the *active* bit in the *memreq* buffer to be sure that the memory access will not be performed several times for a single request (this would not hurt in case of read accesses, but would be unsound for memory writes).

Memory read. If the served request is a read request, the memory result buffer gets the result of the read access. For a memory read to succeed, there either has to be a pending write request to the required physical address in the store buffer, or the address has to be readable in the caches/physical memory. Note, that our memory read is masked. Later we rely on this fact when introducing byte-wise ownership and proving a store-buffer reduction theorem in Section 4.4. For simplicity in further arguing, we set all bits in the result of the memory read to 0, if they are not supposed to be read from the memory by the provided mask.

Definition 3.75 ► Core memory read

label	$\mathit{core\text{-}memory\text{-}read}(i \in \mathit{Pid}, w \in \mathit{Walk})$
guard	$\mathit{tlb\text{-}transl\text{-}ready}(p[i].\mathit{memreq.main}, p[i].\mathit{asid}, \mathit{tlb}[i], w)$, $\mathit{memreq}[i].\mathit{type} = \mathit{read}$, $\mathit{pa} = w.\mathit{pfn} \circ \mathit{memreq}[i].\mathit{va.off}$, $\mathit{pending\text{-}qword\text{-}store}(\mathit{sb}[i], \mathit{pa}) \vee \mathit{read}_{\surd}(\mathit{ca}, \mathit{mm}, i, \mathit{pa}, w.\mathit{mt})$, $\mathit{data} = \mathit{combine}(\mathit{read}(\mathit{ca}, \mathit{mm}, i, \mathit{pa}, w.\mathit{mt}), \mathit{forward}(\mathit{sb}[i], \mathit{pa}))$
effect	$\mathit{memres}'[i].\mathit{data} = \mathit{combine}(0^{64}, (\mathit{data}, \mathit{memreq}[i].\mathit{mask}))$, $\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\mathit{vmexit} = 0$, $\mathit{memreq}'[i].\mathit{active} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

Memory write. A memory write, in contrast to a memory read, does not go directly to the caches/main memory but is rather committed to the store buffer.

Definition 3.76 ► Core memory write

label	$\mathit{core\text{-}memory\text{-}write}(i \in \mathit{Pid}, w \in \mathit{Walk})$
guard	$\mathit{tlb\text{-}transl\text{-}ready}(p[i].\mathit{memreq.main}, p[i].\mathit{asid}, \mathit{tlb}[i], w)$, $\mathit{memreq}[i].\mathit{type} = \mathit{write}$, $\mathit{data} = \mathit{memreq}[i].\mathit{data}$, $\mathit{mask} = \mathit{memreq}[i].\mathit{mask}$, $\mathit{pa} = w.\mathit{pfn} \circ \mathit{memreq}[i].\mathit{va.off}$, $\mathit{store} = \mathit{Store}[\mathit{pa} \mapsto \mathit{pa}, \mathit{data} \mapsto \mathit{data}, \mathit{mt} \mapsto w.\mathit{mt}, \mathit{mask} \mapsto \mathit{mask}]$
effect	$\mathit{sb}'[i] = \mathit{write}(\mathit{sb}[i], \mathit{store})$, $\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\{\mathit{data}, \mathit{vmexit}\} = 0$, $\mathit{memreq}'[i].\mathit{active} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

Note, that setting of fields *data* and *pf* of the *memres* buffer in this step does not effect the execution in any way (these fields must be ignored by the instruction automaton in this case). Yet, we prefer to set these fields to some default value so that we can know the exact state of the *memres* buffer after the step is performed.

Atomic compare exchange. An atomic memory write guarantees that all stores of previous instructions are written to the memory before any memory access of the current instruction occurs. In the real hardware atomic compare exchange is implemented by a sequence of memory accesses. The memory lock, acquired before the start of the first memory access guarantees that no other memory operations are performed in between the steps of the atomic instruction. Hence, the result of all memory accesses of an atomic instruction is equivalent to the effect of a single complex atomic memory action, which we model below.

Regardless of whether the comparison was successful or not, the data fetched from the memory is written to the memory result buffer. The predicate $meq(data_1 \in \mathbb{B}^{64}, data_2 \in \mathbb{B}^{64}, mask \in \mathbb{B}^8) \in \mathbb{B}$ compares only bytes of the data, which are set in the given mask:

$$\begin{aligned} &meq(data_1, data_2, mask) \stackrel{\text{def}}{=} \forall k \in \mathbb{N}_8 : \\ &\quad mask[k] \implies data_1[8*(k+1)-1, 8*k] = data_2[8*(k+1)-1, 8*k]. \end{aligned}$$

◂ **Definition 3.77**
Atomic compare exchange

label	$core\text{-}atomic\text{-}cmpxchng(i \in Pid, w \in Walk)$
guard	$tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$, $memreq[i].type = atomic\text{-}cmpxchng$, $is\text{-}empty(sb[i])$, $pa = w.pfn \circ memreq[i].va.off$, $mask = memreq[i].mask$, $cmp\text{-}data = memreq[i].cmp\text{-}data$, $mem\text{-}data = read(ca, mm, i, pa, w.mt)$, $store\text{-}data = \begin{cases} memreq[i].data & meq(mem\text{-}data, cmp\text{-}data, mask) \\ mem\text{-}data & \text{otherwise} \end{cases}$, $read_{\surd}(ca, mm, i, pa, w.mt)$, $write_{\surd}(ca, mm, i, pa, w.mt, store\text{-}data, mask)$
effect	$(ca', mm') = write(ca, mm, i, pa, w.mt, store\text{-}data, mask)$, $memres'[i].data = combine(0^{64}, (mem\text{-}data, mask))$ $memres'[i].pf = no\text{-}page\text{-}fault()$, $memres'[i].vmexit = 0$, $memreq'[i].active = 0$, $memres'[i].ready = 1$

Locked write. In addition to the atomic compare-exchange operation we introduce another step, which performs a locked memory write[10].

Definition 3.78 ► Locked memory write

label	$core\text{-}locked\text{-}memory\text{-}write(i \in Pid, w \in Walk)$
guard	$tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$, $is\text{-}empty(sb[i])$, $memreq[i].type = locked\text{-}write$, $data = memreq[i].data, mask = memreq[i].mask$, $pa = w.pfn \circ memreq[i].va.off$, $write_{\surd}(ca, mm, i, pa, w.mt, store\text{-}data, mask)$
effect	$(ca', mm') = write(ca, mm, i, pa, w.mt, data, mask)$, $memres'[i].pf = no\text{-}page\text{-}fault()$, $memres'[i].\{data, vmexit\} = 0$, $memreq'[i].active = 0$, $memres'[i].ready = 1$

Triggering a page fault exception. If a TLB translation for the requested virtual address is faulting, the core acknowledges a page fault and writes page fault data to the *memres* buffer. The page fault is reported if there is an active memory request and walk w, which can be used for triggering of a page fault, is present in the TLB. At the same time, Intel [Int11, p. 4-56] and AMD [Adv11a, p. 144] specifications additionally guarantee that all entries (complete and incomplete ones) for a faulty virtual address are flushed from the TLB after a page fault is reported. As a result, we have to split page fault triggering into two steps: first identifying the faulty entry and reporting page fault information, and then performing a TLB invalidation. In the first stage of the page fault triggering we write the result of the page fault to the *memres* buffer, but do not set the *ready* bit. Instead, we raise an "internal" request for a page-fault address invalidation by setting the *pf-flush-req* flag in the *memreq* buffer.

Definition 3.79 ► Triggering page fault (stage 1)

label	$core\text{-}prepare\text{-}page\text{-}fault(i \in Pid, w \in Walk)$
guard	$memreq[i].active = 1$, $memreq[i].type \in MemAcc$, $pte\text{-}read_{\surd}(ca, mm, i, w)$, $pte = abs\text{-}pte(pte\text{-}read(ca, mm, i, w))$, $tlb\text{-}fault\text{-}ready(memreq[i].main, asid[i], tlb[i], pte, w)$
effect	$memres'[i].pf.fault\text{-}code = page\text{-}fault\text{-}code(req.r, pte.p, pte.v)$, $memres'[i].pf.r, va = memreq[i].r, va$, $memres'[i].pf.fault = 1$, $memres[i].\{ready, data\} = 0$, $memreq'[i].active = 0$, $memreq'[i].pf\text{-}flush\text{-}req = 1$

[10] In the x64 instruction set a locked memory write can be implemented by an `xchg` instruction, where one operand is a register and another one is a memory address. An `xchg` operation implicitly has a `lock` prefix, which ensures atomicity of the memory write and acts as a serializing event [Adv11b].

Note, that in the first stage of the PF triggering we set the *ready* bit in the *memres* buffer to zero. This is necessary, because we are overwriting certain fields of the *memres* buffer and we want to make sure, that the instruction core never reads a part of the memory result from one access and another part from another access. (We currently do allow a new request to be issued, while the result of the previous request has not been acknowledged. The result of the old request in this case might get overwritten.)

In the second stage of the page fault triggering we wait until TLB is invalidated and set the *ready* bit in the *memres* buffer.

Definition 3.80 Triggering page fault (stage 2)

label	$\mathit{core\text{-}trigger\text{-}page\text{-}fault}(i \in \mathit{Pid})$
guard	$\mathit{memreq}[i].\mathit{active} = 0$, $\mathit{memreq}[i].\mathit{pf\text{-}flush\text{-}req} = 1$, $\mathit{memreq}[i].\mathit{type} \in \mathit{MemAcc}$, $\mathit{tlb\text{-}invalidated\text{-}pf}(\mathit{tlb}[i], \mathit{memreq}[i].\mathit{va}.\mathit{vpfn}, \mathit{asid}[i])$, $\mathit{asid}[i] = 0$
effect	$\mathit{memreq}'[i].\mathit{pf\text{-}flush\text{-}req} = 0$, $\mathit{memres}'[i].\mathit{vmexit} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

Note, that a regular page fault can be triggered only on a processor running in hypervisor mode. In case if a processor is running in guest mode VMEXIT event is triggered instead.

3.5.2 TLB Operations

The TLB actions, which can be requested from the instruction automaton, include an address invalidation and a move to the $CR3$ register. All these operations can be performed only on a processor running in hypervisor mode.

TLB address invalidation. Address invalidation removes not only all walks for the invalidated virtual address, but also all partial walks.

Definition 3.81 Tagged TLB address invalidation

label	$\mathit{core\text{-}tlb\text{-}invlpga}(i \in \mathit{Pid})$
guard	$\mathit{asid}[i] = 0$, $\mathit{memreq}[i].\mathit{active} = 1$, $\mathit{memreq}[i].\mathit{type} = \mathit{invlpg\text{-}asid}$, $\mathit{tlb\text{-}invalidated}(\mathit{tlb}[i], \mathit{memreq}[i].\mathit{va}.\mathit{vpfn}, \mathit{memreq}[i].\mathit{asid})$,
effect	$\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\{\mathit{data}, \mathit{vmexit}\} = 0$, $\mathit{memreq}'[i].\mathit{active} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

Move to $CR3$**.** If a move to $CR3$ register is requested, we wait until the TLB is completely flushed in the currently active ASID and update the value of the $CR3$ register.

Definition 3.82 ► Move to $CR3$

label	$\mathit{core\text{-}mov2cr3}(i \in \mathit{Pid})$
guard	$\mathit{asid}[i] = 0$, $\mathit{memreq}[i].\mathit{active} = 1$, $\mathit{memreq}[i].\mathit{type} = \mathit{mov2cr3}$, $\mathit{tlb\text{-}empty\text{-}asid}(\mathit{tlb}[i], \mathit{asid}[i])$
effect	$CR3'[i] = \mathit{memreq}[i].\mathit{cr3in}$, $\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\{\mathit{data}, \mathit{vmexit}\} = 0$, $\mathit{memreq}'[i].\mathit{active} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

3.5.3 Virtualization Actions.

VMEXIT. A VMEXIT event is triggered on a processor running in guest mode in one of the following cases:

- VMEXIT is requested by the instruction core,
- TLB contains a walk which can be used for page fault triggering and a memory request is pending,
- TLB address invalidation or a move to $CR3$ is pending.

Additionally, we have to ensure that the store buffer is flushed at the time when VMEXIT is triggered (VMEXIT is a serializing event, which requires flushing of the store buffer).

Definition 3.83 ► VMEXIT

label	$\mathit{core\text{-}vmexit}(i \in \mathit{Pid}, w \in \mathit{Walk})$
guard	$\mathit{asid}[i] \neq 0$, $\mathit{memreq}[i].\mathit{type} \in \{\mathit{mov2cr3}, \mathit{invlpg\text{-}asid}, \mathit{vmexit}\} \cup \mathit{MemAcc}$, $\mathit{memreq}[i].\mathit{type} \notin \mathit{MemAcc} \implies$ $\quad \mathit{memreq}[i].\mathit{active} = 1$, $\mathit{memreq}[i].\mathit{type} \in \mathit{MemAcc} \implies$ $\quad \mathit{memreq}[i].\mathit{pf\text{-}flush\text{-}req} = 1$ $\quad \wedge\ \mathit{memreq}[i].\mathit{active} = 0$ $\quad \wedge\ \mathit{tlb\text{-}invalidated\text{-}pf}(\mathit{tlb}[i], \mathit{memreq}[i].\mathit{va}.\mathit{vpfn}, \mathit{asid}[i])$, $\mathit{is\text{-}empty}(\mathit{sb}[i])$, $\mathit{memres}[i].\mathit{ready} = 0$
effect	$CR3'[i] = CR3_{hyp}[i]$, $\mathit{asid}'[i] = 0$, $\mathit{memres}'[i].\{\mathit{ready}, \mathit{vmexit}\} = 1$, $\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\mathit{data} = 0$, $\mathit{memreq}'[i].\{\mathit{active}, \mathit{pf\text{-}flush\text{-}req}\} = 0$, $\mathit{memres}'[i].\mathit{vmexit\text{-}memreq} = \mathit{memreq}[i]$

Note, that as a precondition for VMEXIT we require the *memres* buffer to have no pending result of the previous operation (i.e., the flag *ready* does not have to be 0). Strictly speaking, we could allow a new request to be accepted by the memory automaton only when the result of the previous request is acknowledged and the *ready* bit is reset. In this case we could be sure that the buffers *memreq* and *memres* never contain an active request and a pending result at the same time. Yet, in the proofs presented further in this thesis we don't need this requirement, and the only step where we have to know that this bit equals 0 is the VMEXIT step (one would need this knowledge to satisfy preconditions of Lemma 8.4).

VMRUN. In case the instruction automaton requests a VMRUN, we write the provided values to the $CR3$ and ASID registers, and wait until the store buffer is flushed. Additionally, we may inject a page fault to the *memres* buffer (if required by the instruction automaton) and inject a pending memory request to the *memreq* buffer. The data which might be injected in the memory request/result buffers consists of the following fields:

$$InjectData \stackrel{\text{def}}{=} [req \in MemReqMain, pf \in PfData, ready \in \mathbb{B}].$$

◄ **Definition 3.84** VMRUN injection data

To simplify notation later in this thesis we sometimes write *idata.x* instead of *idata.req.x* when referring to the parameters of the injection data $idata \in InjectData$.

If a bit $memreq[i].complete\text{-}flush$ is set, then we wait until TLB removes all walks with ASIDs other than zero.

◄ **Definition 3.85** VMRUN

label	$core\text{-}vmrun(i \in Pid)$
guard	$asid[i] = 0$, $memreq[i].active = 1$, $memreq[i].type = VMRUN$, $is\text{-}empty(sb[i])$, $memreq[i].complete\text{-}flush \implies$ $\quad \forall asid \neq 0 : tlb\text{-}empty\text{-}asid(tlb[i], asid)$
effect	$memreq'[i].main = memreq[i].inject\text{-}data.req$, $CR3'_{hyp}[i] = CR3[i]$, $CR3'[i] = memreq[i].cr3in$, $asid'[i] = memreq[i].asidin$, $memres'[i].pf = memreq[i].inject\text{-}data.pf$, $memres'[i].\{data, vmexit\} = 0$, $memres'[i].ready = memreq[i].inject\text{-}data.ready$

After a VMRUN event is completed and the core continues execution of guest instructions, the guest will see the result of his memory access without knowing that it was interrupted. Injected data could contain information for a successful INVLPG (in the ASID of the guest), or a move to $CR3$, or a page fault which caused VMEXIT, was virtualized by the hypervisor, and has to be propagated to the guest.

The value of the field $memres'[i].data$ is irrelevant, because we never inject a result of a memory read operation at the VMRUN. Yet, instead of leaving this filed undefined (or unchanged) after the step, we assign a zero value to it so that we can later specify the respective VMRUN step in C-IL + HW semantics (Section 7.2.1).

Note, that the instruction automaton has to guarantee, that the value of the ASID to switch to is different from 0.

3.6 Instruction Automaton

The configuration of the instruction core contains a single component, which denotes the internal state of the automaton:

Definition 3.86 ▶ Instruction core

$$InstrCore \stackrel{\text{def}}{=} [state \in InstrCoreState].$$

To argue about updates of the internal state of the instruction automaton we introduce two uninterpreted function. One function is used to perform an internal step of the instruction automaton and the other is used to perform an input action (from the point of view of the instruction automaton) accepting the result of the memory operation received from the memory core:

$$\begin{aligned}&next\text{-}instr\text{-}state(state \in InstrCoreState) \in InstrCoreState,\\&next\text{-}instr\text{-}mem\text{-}state(state \in InstrCoreState,\\&\qquad\qquad memres \in MemRes) \in InstrCoreState.\end{aligned}$$

An internal step of the instruction automaton is defined in the following way.

Definition 3.87 ▶ Internal step of instruction automaton

label	$core\text{-}instr\text{-}step(i \in Pid)$
guard	
effect	$state[i]' = next\text{-}instr\text{-}state(state[i])$

The next memory request to be issued by the instruction automaton is obtained with the following function:

$$next\text{-}mem\text{-}req(state \in InstrCoreState, memres \in MemRes) \in MemReq \cup \{\bot\}.$$

The step of issuing a memory request is defined in the following way.

Definition 3.88 ▶ Issuing a memory request

label	$core\text{-}issue\text{-}mem\text{-}req(i \in Pid, req \in MemReq)$
guard	$next\text{-}mem\text{-}req(state[i], memres[i]) = req$
effect	

The effect of an input action from memory automaton involves updating the internal state of instruction automaton based on the obtained result of the memory operation.

Definition 3.89 ▶ Accepting memory reply

label	$core\text{-}send\text{-}mem\text{-}res(i \in Pid, res \in MemRes)$
guard	
effect	$state[i]' = next\text{-}instr\text{-}mem\text{-}state(state[i], res)$

CHAPTER 4

Reduced Hardware Model

4.1 Specification

4.2 Cache Reduction

4.3 Ownership

4.4 SB Reduction

4.5 TLB Reduction

4.6 Putting It All Together

One of our goals in this thesis is to define a hardware model, which can be later used for verification of system software code using an automated C verifier. The very first and crucial restriction on the hardware model introduced by the C verifier is the sequentially consistent memory model. The C verifier can operate only with the memory, where store buffers, caches, and TLBs are not visible. The hardware model defined thus far does not fit the aforementioned requirements. In this Chapter we define a *reduced hardware model* without caches, SBs, and TLBs, which simulates the full abstract machine (referred later as a *reference hardware model*) presented in Chapter 3.

To perform SB reduction we partition the memory into ownership sets and define an ownership discipline, which has to be maintained in order for simulation to go through. For TLB reduction we introduce the set of identity mapped (hypervisor) page tables and define properties on them. To perform cache reduction, we restrict our hardware model to operate only with "write-back" memory types.

A sketch of the reduction theorems presented in this chapter was outlined by Degenbaev, Paul, and Schirmer in [DPS09].

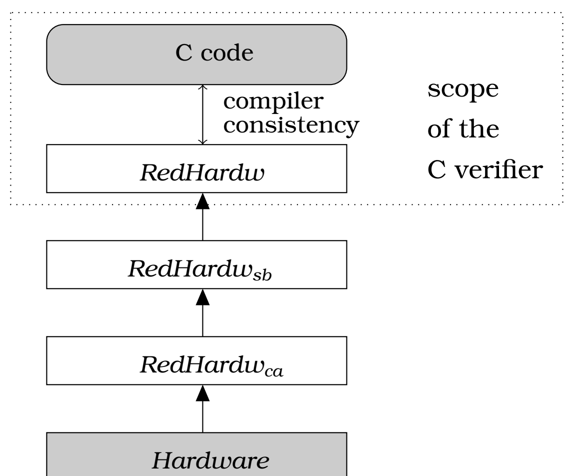

Figure 4.1: The stack of reduced hardware models.

Hypervisor code is often running in parallel with guest code being executed on other processors of the system. As soon as we want to provide the full hardware model to the guest, we cannot reduce SBs and TLBs on the processors executed in virtualziation mode. Moreover, we need to have TLBs on processors running the guest code in order to virtualize the guest memory. Hence, we define a reduced hardware model where SBs and TLBs are invisible for processors running in hypervisor mode and are visible otherwise.

At the same time, since we are controlling guest memory translations (by setting shadow page tables), we can control the type of the guest memory. This allows us to reduce caches on all processors, including the ones running the guest code. Since we do not consider devices, we make this reduction by assigning a "write-back" type to the whole guest memory[1].

We do the hardware reduction in three stages: first we reduce caches, then store buffers, and finally - TLBs. As a result, we have three different reduced models and three simulation theorems (Figure 4.1). Two intermediate reduced models we call *cache-reduced* and *SB-reduced* hardware respectively. Cache, SB, and TLB reduction theorems from Sections 4.2, 4.4, and 4.5 are stated for a single step of the hardware machine and are not inductive, i.e., we do not show that preconditions for the reduction are maintained after every step of the machine. In Section 4.6 we unite three reduction theorems into a single one and make it inductive.

4.1 Specification

The only hardware components, which are reduced completely for all processors are caches. Store buffers and TLBs are reduced only for processors operating in hypervisor mode. Any processor operating in hypervisor mode may at some point enter guest mode, which makes its SB and TLB visible

[1] In the presence of devices one would have to ensure that I/O mapped memory regions always have non-cacheable memory types.

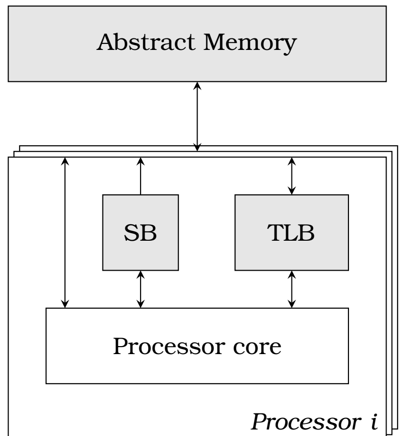

Figure 4.2: Hardware model after cache reduction

again. Moreover, the TLB content is not flushed after the mode switch. This means that we have to argue about TLBs of all processors in the reduced model (including the ones running in hypervisor mode), even though some of them do not participate in address translations. More precisely, we abstract away only the part of the TLB, where the walks with the ASID tag 0 are stored.

All reductions are done only to the memory automaton of the hardware system. Instruction automata in all reduced hardware models are the same as in the reference model. Moreover, memory automata of all reduced models have the same components of the state, which are identical to the components of the reference hardware with the exception of caches which are not visible in the reduced models (Figure 4.2).

The state of the memory automaton of reduced hardware is fixed by the following data type:

$$\begin{aligned} \mathit{RedMemHardw} \overset{\text{def}}{=} [&p \in \mathit{Pid} \mapsto \mathit{MemCore}, mm \in \mathit{Memory}, \\ &tlb \in \mathit{Pid} \mapsto \mathit{Core}, sb \in \mathit{Pid} \mapsto \mathit{SB}]. \end{aligned}$$

◂ **Definition 4.1** Reduced hardware state (memory automaton)

The state of the reduced hardware machine is then obtained by combining the reduced memory automaton with the instruction automaton of the reference model.

$$\mathit{RedHardw} \overset{\text{def}}{=} [h_m \in \mathit{RedMemHw}, h_i \in \mathit{InstrHw}].$$

◂ **Definition 4.2** Reduced hardware state

The part where all reduced hardware models are different, is the transition relation. In order to distinguish the models from each other, we denote the transition relation of the first reduced model (after cache reduction) by Δ_{ca}, the relation of the second reduced model (after cache and SB reduction) by Δ_{sb}, and the relation of the fully reduced model (after cache, SB, and TLB reduction) by Δ (the same symbol as used for the original hardware model).

Additionally, we introduce the types

$$RedHardw_{ca} \stackrel{\text{def}}{=} RedHardw,$$
$$RedHardw_{sb} \stackrel{\text{def}}{=} RedHardw,$$

to distinguish instances of different kind of reduced models.

4.2 Cache Reduction

Caches are made invisible by requiring all memory addresses to always have a "write-back" memory type[2]. This prevents the hardware from mixing memory types of accesses to a given address and making the cache content inconsistent with the data in the physical memory. With this requirement on the program enforced, caches can be abstracted in a straightforward manner:

Definition 4.3 ► Memory abstraction (reducing caches)

$$reduced\text{-}ca\text{-}mm(mm \in Memory, ca \in Pid \mapsto Cache) \in Memory,$$
$$reduced\text{-}ca\text{-}mm(mm, ca)[pa] \stackrel{\text{def}}{=} \begin{cases} ca[i].data[pa] & ca[i].state[pa] \neq I \\ mm[pa] & \text{otherwise.} \end{cases}$$

Cache-reduced hardware is obtained by applying the memory abstraction function to the components of the reference hardware model:

Definition 4.4 ► Hardware reduction (caches)

$$reduced\text{-}ca\text{-}hw(h \in Hardware) \in RedHardw_{ca},$$
$$reduced\text{-}ca\text{-}hw(h) \stackrel{\text{def}}{=} RedHardw_{ca}[p \mapsto h.p, p_i \mapsto h.p_i, tlb \mapsto h.tlb, sb \mapsto h.sb, mm \mapsto reduced\text{-}ca\text{-}memory(h.mm, h.ca)].$$

Transitions of the cache-reduced hardware are equivalent to the transitions of the reference model with the following exceptions:

- all cache steps are empty (i.e., perform stuttering),
- memory accessing steps operate directly on the physical memory (by the means of $read(mm, \ldots)$, $write(mm, \ldots)$ functions) rather than on the cache/memory system,
- the functions *pte-read* and *pte-write* operate directly on the physical memory, and
- the shared memory is considered to be always accessible i.e., we do not require $read_{\surd}(mm, \ldots)$ to hold for the access to succeed. This weakening of the model is fine since we only argue about terminating traces of the reference hardware.

In the reference model the memory type of a memory access is obtained from the memory type of the walk, chosen for the address translation of this access. The following predicate denotes that all walks with a given ASID in a

[2]Requiring just cacheable memory type is not enough, because performing an access with a "write-protect" memory type may lead to the loss of data, in case one of the previous writes to this address was done with a "write-back" or a "write-through" type.

given TLB have a "write-back" memory type:

$$cacheable\text{-}walks(tlb \in Tlb, asid \in \mathbb{N}) \in \mathbb{B},$$
$$cacheable\text{-}walks(tlb, asid) \stackrel{\text{def}}{=} \forall w \in Walk : tlb[w] \wedge w.asid = asid \implies w.mt = WB.$$

◄ **Definition 4.5** Cacheable (write-back) walks

To guarantee that all accesses in the system are always performed to addresses with a "write-back" memory type, we have to maintain the following invariant on all walks with the active ASID in all TLBs. Additionally, we maintain the same property on all walks with ASID 0 (this part of invariant we use in Lemma 4.6 to make the invariant inductive in case of VMEXIT).

name	$inv\text{-}tlb\text{-}cacheable(h \in Hardware)$
property	$cacheable\text{-}walks(tlb[i], asid[i]) \wedge cacheable\text{-}walks(tlb[i], 0)$

◄ **Invariant 4.6** Cacheable TLB memory types

We also need to maintain an analogous invariant for SBs to guarantee that all stores they commit to the memory have a "write-back" memory type.

name	$inv\text{-}sb\text{-}cacheable(sb \in Pid \mapsto SB)$
property	$\forall j < \lvert sb[i].buffer \rvert : sb[i].buffer[j].mt = WB$

◄ **Invariant 4.7** Cacheable SB memory types

With the help of Invariant 4.6 and Invariant 4.7 we can now prove a cache reduction theorem.

Theorem 4.1 (Cache reduction). *Let all TLBs and SBs provide only "write-back" memory types and the data in all caches be consistent. Moreover, let reduction hold between states $h \in Hardware$ and $h_r \in RedHardw_{ca}$. Then reduction is maintained after any step of the reference machine.*

$$\begin{aligned}
&h \xrightarrow{a} h' \\
&\wedge\ inv\text{-}consistent\text{-}caches(h.ca, h.mm) \\
&\wedge\ inv\text{-}tlb\text{-}cacheable(h) \\
&\wedge\ inv\text{-}sb\text{-}cacheable(h.sb) \\
&\wedge\ h_r = reduced\text{-}ca\text{-}hw(h) \\
&\implies h_r \xrightarrow{a} h_r' \\
&\qquad \wedge\ h_r' = reduced\text{-}ca\text{-}hw(h')
\end{aligned}$$

Proof. If step $h \xrightarrow{a} h'$ does not interfere with the caches or the main memory, the step of the reduced machine is equivalent to $h \xrightarrow{a} h'$ and the theorem holds. Otherwise we do a case split on the type of the hardware step from h to h':

Case 1: $h \xrightarrow{a} h'$ involves a read from the main memory on the processor i and the physical address pa . The reduced machine performs the same kind of a step, reading the physical memory instead of the cache/ memory system. From *inv-tlb-cacheable* we know that the memory read is done from the "write-back" memory address. Hence,

$$read(h_r.mm, i, pa) = h.ca[i].data[pa].$$

From *inv-consistent-caches* we know that the content of all caches, which have the data for the pa in a valid state, is the same. It follows, that the reduced memory abstraction is well-defined and $h_r.mm[pa] = h.ca[i].data[pa]$. Thus, the results of the memory reads on two machines are the same:

$$read(h_r.mm, i, pa) = read(h.ca, h.mm, i, pa, mt).$$

Case 2: $h \xrightarrow{a} h'$ involves a write to the memory:

$$(h'.ca, h'.mm) = write(h.ca, h.mm, i, pa, mt, data, mask).$$

The reduced machine performs the same kind of a step, writing the main memory instead of the cache/memory system. Since the write is done to a "write-back" memory address, we know that it does not bypass the cache. Moreover, all other caches have the data for pa in an invalid state. Hence, we get

$$h'_r.mm = write(h_r.mm, i, pa, data, mask).$$

Case 3: $h \xrightarrow{a} h'$ is a step of the abstract cache i for the address pa. Our cache coherence protocol guarantees that the data stays consistent between all caches (see Lemma 4.2). If some cache is in a valid state in h and stays in a valid state in h', then its data is unchanged. If a cache is invalid in h and goes to a valid state in h', then it either gets the data from the main memory or from another cache. In all these cases the result of the memory abstraction function is not affected by the step and the value of the abstracted memory is unchanged. A corresponding step of the reduced machine $h_r \xrightarrow{a} h'_r$ is an empty step i.e., $h'_r = h_r$.

□

Additionally, we state an easy lemma showing that the cache consistency is maintained after every step of the machine.

Lemma 4.2 (Consistent caches). *Let all TLBs and SBs provide only "write-back" memory types and the data in all caches be consistent in state $h \in$ Hardware. Further, let $h \xrightarrow{a} h'$ be a hardware step. Then cache consistency also holds in state h':*

$$\begin{aligned} & h \xrightarrow{a} h' \\ & \wedge\ inv\text{-}consistent\text{-}caches(h.ca, h.mm) \\ & \wedge\ inv\text{-}tlb\text{-}cacheable(h) \\ & \wedge\ inv\text{-}sb\text{-}cacheable(h.sb) \\ & \implies inv\text{-}consistent\text{-}caches(h'.ca, h'.mm). \end{aligned}$$

Proof. The proof is done by a case split on the type of the hardware step:

Case 1: $h \xrightarrow{a} h'$ is a step of writing a line with the address pa to the cache i by the processor core (either *core-locked-memory-write* or *core-atomic-cmpxchng*). Invariant *inv-tlb-cacheable* guarantees that the write is performed with a "write-back" memory type. It follows

that no other cache has the same line in the valid state and the state of the line in cache i is changed to M. Consistency invariants in this case are trivially maintained,

Case 2: $h \xrightarrow{a} h'$ is a *commit-store* step of the store buffer i to the address pa. Invariant *inv-sb-cacheable* guarantees that the write is performed with a "write-back" memory type. The state of the cache line is changed to M and all consistency invariants are trivially maintained,

Case 3: $h \xrightarrow{a} h'$ is a step of cache i of fetching a line with the address pa from cache j. The state of the line in cache i is set to S. We now consider three sub-cases:

Case 3.1: $h.ca[j].state[pa] = E$. During the transition the state for pa is changed to S. From *inv-consistent-caches*$(h.ca, h.mm)$ it follows that no caches other than i and j have the data for the line in a valid state. Moreover, the memory contains the same data as the caches do. Hence, cache consistency is preserved in h',

Case 3.2: $h.ca[j].state[pa] = M$. During the transition the state for pa is changed to O. From *inv-consistent-caches*$(h.ca, h.mm)$ it follows that no caches other than i and j have the data for the line in a valid state. Hence, cache consistency is preserved in h',

Case 3.3: $h.ca[j].state[pa] = O$. During the transition the state for pa is left unchanged and cache consistency is trivially preserved,

Case 4: $h \xrightarrow{a} h'$ is a step of cache i of fetching a line with the address pa from the main memory. The guard of the step guarantees that no other cache has the data in state M or O. From *inv-consistent-caches*$(h.ca, h.mm)$ it follows that the data in the memory for the address pa is the same, as the data in all valid cache lines. Hence, fetching the data from the memory does not break cache consistency,

Case 5: $h \xrightarrow{a} h'$ is a step of cache i of writing back a line with the address pa to the main memory. After the step we have

$$h'.ca[i].data[pa] = h'.mm[pa].$$

We again consider two sub-cases:

Case 5.1: $h.ca[i].state[pa] = M$. During the transition the state for pa is changed to E. From *inv-consistent-caches*$(h.ca, h.mm)$ it follows that no cache other than i has the data in a valid state. The content of cache i and the main memory is the same after the step and cache consistency is preserved,

Case 5.2: $h.ca[i].state[pa] = O$. During the transition the state for pa is changed to S. From *inv-consistent-caches*$(h.ca, h.mm)$ it follows that all other caches have the data in a shared or invalid state. For the cashes which have this line in a shared state the data in the line is the same as the data in cache i. After the step all caches have this line in state

I or S and the data for the valid line is the same one as the data in the main memory. Hence, cache consistency is preserved,

Case 6: $h \overset{a}{\rightarrow} h'$ is a step of dropping a line, bringing a line to an exclusive state, or passing the ownership of a line to another cache. In all these cases cache consistency is trivially maintained.

All the other steps do not affect the state of the cache and can not possibly break the invariant. □

Ensuring that Invariant 4.6 holds after a step of the machine requires arguing about the content of page tables, the value of control bits of the $CR3$ register, and the content of the MTRR registers. We fix these properties in Section 4.5.

4.3 Ownership

In order to prove SB and TLB reduction theorems and to verify memory safety of concurrent programs we need to introduce an ownership discipline for memory addresses. More precisely, we aim at partitioning the memory address space into a set of disjoint ownership domains of different cores and a set of shared addresses.

4.3.1 Owned and Shared Addresses

In the context of a hypervisor program running atop of the hardware machine we distinguish several sets of addresses, which we assume to be statically fixed[3]. This partitioning is done from the point of view of the hypervisor program and comprise the following sets of byte addresses:

- the set of shared writable addresses (e.g., used for storing shared global data of the program):

$$SharedAddr \subset \mathbb{B}^{bpa},$$

- the set of shared read-only-addresses (e.g., used for storing the code of the hypervisor program):

$$ReadOnlyAddr \subset \mathbb{B}^{bpa},$$

- the set of hypervisor addresses where local stacks of every hypervisor thread is located:

$$StackAddr \subset \mathbb{B}^{bpa}.$$

  This set is subdivided into subsets $StackAddr_i$, where $i \in Pid$ and all subsets are disjoint from each other. A set $StackAddr_i$ is always

[3]In general, sets of read-only and shared addresses are not fixed and may change. Yet, we are interested only in those execution traces, where these sets are already fixed at the start of execution and do not change afterwards.

exclusively owned by a processor i, which means that other processors can never access addresses from this set.

- the set of hypervisor addresses, where global non-shared variables of the program are located (including the heap region):

$$PrivateAddr \subset \mathbb{B}^{bpa},$$

- the set of addresses allocated to guest partitions. From the hypervisor point of view these addresses are also shared:

$$GuestAddr \subset \mathbb{B}^{bpa}.$$

All these sets have to be pairwise disjoint:

$$\begin{aligned} \forall A, B \in \{&SharedAddr, ReadOnlyAddr, StackAddr, \\ &PrivateAddr, GuestAddr\} : A \cap B = \emptyset \\ \forall i, j : i \neq j &\implies StackAddr_i \cap StackAddr_j = \emptyset. \end{aligned}$$

◄ **Invariant 4.8**
Disjoint sets of addresses

Further in this thesis we assume that partitioning of memory into sets of addresses is correct and is statically fixed. Hence, we assume that Invariant 4.8 always holds.

Further, we introduce local *ownership sets* for every processor in the system. Since the ownership discipline is defined purely by software, we do not keep ownership sets of addresses in the hardware configuration, but rather introduce a separate data type

$$Ownership \in Pid \mapsto 2^{\mathbb{B}^{bpa}}.$$

For $o \in Ownership$ the set $o[i]$ keeps the addresses owned by the core with the index i, when it is running in hypervisor mode. Only addresses from the set *PrivateAddr* can be present in the set $o[i]$. Addresses from the set $StackAddr_i$ are considered to be always owned by a thread i and we do not include them to the set $o[i]$, which may change during execution. Since we do not do SB reduction for processors running in guest mode, we do not need to argue explicitly about addresses owned by these processors (from the set *GuestAddr*).

A processor in hypervisor mode is allowed to read any address, except those addresses which are in ownership domains of other processors. It can write either an owned, a shared writable, or a guest address. A processor in guest mode is allowed to access only guest addresses.

Note, that we require the set of shared writable addresses to be disjoint from sets of owned addresses. Yet, on top of our ownership model one can implement another model, where a shared address can be in the ownership domain of some processor. In this case only this processor can write this address and others can only read it (including MMUs which would not be able to write to this address).

Further in this Chapter we consider ownership setting o to be changing during hardware execution (ownership transfer). To denote a sequence of ownership setting from o^0 to o^n, which consists of $n + 1$ states we write

$o^0, \ldots, o^n$.

4.3.2 Ownership Discipline

The ownership discipline consists of a number of invariants which have to be maintained by any step of the system. This discipline is later used to justify reordering of hardware steps to I/O points (see Section 5.4.3). We also use the ownership discipline to prove a store buffer reduction theorem in Section 4.4.

First, we formalize the disjointness of ownership sets.

Invariant 4.9 ► Disjoint ownership domains

name	$inv\text{-}disjoint\text{-}ownership\text{-}domains(p \in Pid \mapsto MemCore, o \in Ownership)$
property	$pa \in o[i] \implies pa \in PrivateAddr,$ $pa \in o[i] \wedge pa \in o[j] \implies i = j$

Next, we define restrictions on reading and writing operations performed by the core. In order to identify all byte addresses participating in a given read or write, we introduce the following function:

Definition 4.10 ► Affected byte addresses

$$affected\text{-}byte\text{-}addr(pa \in \mathbb{B}^{61}, mask \in \mathbb{B}^8) \in 2^{\mathbb{B}^{64}},$$
$$affected\text{-}byte\text{-}addr(pa, mask) \stackrel{\text{def}}{=} \{bpa \mid \exists bx \in \mathbb{B}^3 : bpa = pa \circ 0^3 + bx \wedge mask[\langle bx \rangle]\}.$$

Memory reads can be performed from any address which is not owned by others if a processor is running in hypervisor mode and from guest addresses if the processors is running in guest mode.

Invariant 4.11 ► Ownership for reads

name	$inv\text{-}owned\text{-}reads(p \in Pid \mapsto MemCore, tlb \in Pid \mapsto Tlb, o \in Ownership)$
property	$p[i].memreq.type = read$ $\wedge\ tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$ $\wedge\ pa = w.pfn \circ p[i].memreq.va.off$ $\wedge\ bpa \in affected\text{-}byte\text{-}addr(pa, p[i].memreq.mask)$ $\implies (p[i].asid = 0 \implies bpa \notin \bigcup_{j \neq i}(o[j] \cup StackAddr_j))$ $\wedge\ (p[i].asid \neq 0 \implies bpa \in GuestAddr)$

In contrast to regular reads, writes in hypervisor mode can be performed only to owned addresses.

Invariant 4.12 ► Ownership for writes

name	$inv\text{-}owned\text{-}writes(p \in Pid \mapsto MemCore, tlb \in Pid \mapsto Tlb, o \in Ownership)$
property	$p[i].memreq.type = write$ $\wedge\ tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$ $\wedge\ pa = w.pfn \circ p[i].memreq.va.off$ $\wedge\ bpa \in affected\text{-}byte\text{-}addr(pa, p[i].memreq.mask)$ $\implies (p[i].asid = 0 \implies bpa \in o[i] \cup StackAddr_i)$ $\wedge\ (p[i].asid \neq 0 \implies bpa \in GuestAddr)$

Writes to the memory performed by an atomic compare-exchange operation or by a locked memory write have to be done to an owned, shared, or guest address (for processors running the hypervisor) or to a guest address (for processors in guest mode).

name	$inv\text{-}owned\text{-}atomic(p \in Pid \mapsto MemCore, tlb \in Pid \mapsto Tlb,$ $o \in Ownership)$
property	$p[i].memreq.type \in \{atomic\text{-}cmpxchng, locked\text{-}write\}$ $\wedge\ pa = w.pfn \circ p[i].memreq.va.off$ $\wedge\ bpa \in affected\text{-}byte\text{-}addr(pa, p[i].memreq.mask)$ $\wedge\ tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$ $\implies (p[i].asid = 0 \implies bpa \in SharedAddr \cup GuestAddr$ $\cup\ o[i] \cup StackAddr_i)$ $\wedge\ (p[i].asid \neq 0 \implies bpa \in GuestAddr)$

◄ **Invariant 4.13**
Ownership for interlocked operations

Note, that Invariants 4.12 and 4.13 require all writes to shared addresses to be performed with an interlocked operation, which has a side effect of flushing the store buffer. Shared variable of a C program have to be marked with the **volatile** type qualifier and the compiler is responsible for executing a locked write or a locked compare-exchange for every update of volatile data in a C program.

In contrast to memory reads, regular writes are not done to the cache-memory subsystem directly, but are at first committed to store buffers. Hence, we have to talk about all stores which are currently pending in SBs.

name	$inv\text{-}owned\text{-}stores(p \in Pid \mapsto MemCore, sb \in Pid \mapsto SB,$ $o \in Ownership)$
property	$pending\text{-}byte\text{-}store(sb[i], pa, \langle byte \rangle)$ $\wedge\ bpa = pa \circ 0^3 + byte \wedge byte \in \mathbb{B}^3$ $\implies (p[i].asid = 0 \implies bpa \in o[i] \cup StackAddr_i)$ $\wedge\ (p[i].asid = 0 \implies bpa \in GuestAddr),$

◄ **Invariant 4.14**
SB stores owned

The memory of the hardware machine may be accessed not only by processor cores, but also by MMUs. Hence, we need to be sure that MMUs also obey the ownership discipline. More precisely, when an MMU is writing a PTE in the memory, we have to be sure that there are no stores to this PTE pending in any of the store buffers.

name	$inv\text{-}tlb\text{-}ownership(p \in Pid \mapsto MemCore, tlb \in Pid \mapsto Tlb,$ $o \in Ownership)$
property	$tlb[i][w] \wedge w.l \neq 0 \wedge w.asid = p[i].asid$ $\wedge\ bpa \in qword2bytes(pte\text{-}addr(w.pfn, w.vpfn.px[w.l]))$ $\implies bpa \notin \bigcup_{i \neq j}(o[j] \cup StackAddr_j)$

◄ **Invariant 4.15**
Ownership for walks

We group all the invariants defined in this section into a single property.

Invariant 4.16 ► Ownership discipline

name	$inv\text{-}ownership\text{-}discipline(h \in Hardware, o \in Ownership)$
property	$inv\text{-}disjoint\text{-}ownership\text{-}domains(h.p, o)$, $inv\text{-}owned\text{-}reads(h.p, h.tlb, o)$, $inv\text{-}owned\text{-}writes(h.p, h.tlb, o)$, $inv\text{-}owned\text{-}atomic(h.p, h.tlb, o)$, $inv\text{-}owned\text{-}stores(h.p, h.sb, o)$, $inv\text{-}tlb\text{-}ownership(h.p, h.tlb, o)$

Note, that maintaining the ownership discipline is user's and compiler's responsibility. The correct ownership scenario first has to be established for the original program. The compiler has to guarantee that the ownership discipline of the original code is then transfered to the hardware ISA level.

The ownership discipline presented in this section is quite strict. In particular, it does not allow store buffers to contain stores to the shared data. As a result, the user has to flush the SB every time when he does a write to the shared data. However, weaker ownership disciplines could be defined, which still ensure sequential consistency of the memory/SBs system and can be used for the store buffer reduction. One of such disciplines [CS10] requires that the store buffer is flushed not after every shared write, but before a shared read and only in case if the store buffer is "dirty" (i.e., there were writes to shared data after the last flush).

4.4 SB Reduction

Generally, defining an abstraction function for SB reduction is a non-trivial task, because of the unknown ordering of stores, committed by SBs to the memory (due to the nondeterministic nature of SB behaviour). For instance, consider a programming discipline where the store buffer flushes are done not after writes to shared data, but before reads to shared data [CS10]. In this case, one could end up with having two different chunks of data for one physical address residing in different store buffers without any clue, which data will be committed to the memory first.

A simpler ownership discipline used in this thesis allows us to overcome this problem and to abstract store-buffers in a straightforward way. Our discipline guarantees, that no two store buffers contain a pending store request to a given physical byte address at the same time. Hence, the memory abstraction for the SB-reduced model can be constructed analogously to the abstraction function for cache reduction:

Definition 4.17 ► Memory abstraction (reducing store buffers)

$$reduced\text{-}sb\text{-}mm(mm \in Memory, sb \in Pid \mapsto SB, p \in Pid \mapsto MemCore) \in Memory,$$

$$reduced\text{-}sb\text{-}mm(mm, sb)[pa] \stackrel{\text{def}}{=} data, \quad \text{where}$$

$$\forall i \in \mathbb{N}_{64} : data[i] = \begin{cases} (sb[j].data[pa])[i] & pending\text{-}byte\text{-}store(sb[j], pa, \lfloor i/8 \rfloor) \\ & \wedge p[i].asid = 0 \\ mm[pa] & \text{otherwise.} \end{cases}$$

In the SB-reduced model SBs of processors running in hypervisor mode are considered to be always empty, while SBs of other processors are simply

copied from the reference model:

$$reduced\text{-}sb(sb \in Pid \mapsto SB, p \in Pid \mapsto MemCore) \in Pid \mapsto SB,$$
$$reduced\text{-}sb(mm, sb)[i] \stackrel{\text{def}}{=} \begin{cases} empty\text{-}sb() & p[i].asid = 0 \\ sb[i] & \text{otherwise.} \end{cases}$$

◄ **Definition 4.18** SB abstraction

The SB-reduced machine is then defined in the following way:

$$reduced\text{-}sb\text{-}hw(h \in RedHardw_{ca}) \in RedHardw_{sb},$$
$$reduced\text{-}sb\text{-}hw(h) \stackrel{\text{def}}{=} h[sb \mapsto reduced\text{-}sb(h.sb, h.p),$$
$$mm \mapsto reduced\text{-}sb\text{-}mm(h.mm, h.sb, h.p)].$$

◄ **Definition 4.19** Hardware reduction (store buffers)

The transition system of the SB-reduced hardware is equivalent to the transition system of the cache-reduced model with the exception of the steps of processors running in hypervisor mode (i.e., with current ASID set to 0). The steps of these processors differ in the following way:

- all store buffer steps are empty (i.e., perform stuttering),
- the core memory write operation is done directly to the main memory of the SB-reduced machine, rather than committed to a store buffer,
- the core memory read is always done from the main memory (no store buffer forwarding),
- VMRUN step does not require the SB to be flushed.

As an example of the core memory access of a processor running in hypervisor mode, consider a memory write operation.

◄ **Definition 4.20** Core memory write ($RedHardw_{sb}$)

label	$core\text{-}memory\text{-}write(i \in Pid, w \in Walk)$
guard	$asid[i] = 0$, $tlb\text{-}transl\text{-}ready(p[i].memreq.main, p[i].asid, tlb[i], w)$, $memreq[i].type = write$, $data = memreq[i].data$, $mask = memreq[i].mask$, $pa = w.pfn \circ memreq[i].va.off$
effect	$mm' = write(mm, pa, data, mask)$, $memres'[i].pf = no\text{-}page\text{-}fault()$, $memres'[i].\{data, vmexit\} = 0$, $memreq'[i].active = 0$, $memres'[i].ready = 1$

Theorem 4.3 (Store buffer reduction). *Let the ownership discipline hold in a state $h \in RedHardw_{ca}$. Moreover, let SB-reduction hold between states h and $h_r \in RedHardw_{sb}$. Then reduction is maintained after any step of the cache-*

reduced machine:

$$\begin{aligned}
& h \xrightarrow{a} h' \\
& \wedge\ \textit{inv-ownership-discipline}(h, o) \\
& \wedge\ h_r = \textit{reduced-sb-hw}(h) \\
& \implies h_r \xrightarrow{a} h'_r \\
& \qquad \wedge\ h'_r = \textit{reduced-sb-hw}(h').
\end{aligned}$$

Proof. If step $h \xrightarrow{a} h'$ does not interfere with the store buffers or the main memory, the step of the SB-reduced machine is equivalent to $h \xrightarrow{a} h'$ and the theorem holds. Otherwise, we do a case split on the type of a step performed by the host hardware.

Case 1: $h \xrightarrow{a} h'$ is a compare exchange step to physical address pa on processor i. The byte addresses affected by this step are

$$bpa \in \textit{affected-byte-addr}(pa, \textit{memreq}[i].\textit{mask}).$$

The reduced machine performs the same kind of a step. From *inv-owned-atomic* (Invariant 4.13) we get that

$$bpa \in \textit{SharedAddr} \vee bpa \in \textit{GuestAddr}.$$

Using *inv-owned-stores* (Invariant 4.14) we conclude that there are no stores to *bpa* pending in any of the store buffers of processors running in hypervisor mode. Hence,

$$\begin{aligned}
h'.p[i].\textit{memres.data} &= \textit{combine}(0^{64}, (h.\textit{mm}[pa], h.p[i].\textit{memreq.mask})) \\
&= \textit{combine}(0^{64}, (h_r.\textit{mm}[pa], h_r.p[i].\textit{memreq.mask})) \\
&= h'_r.p[i].\textit{memres.data}, \quad \text{and} \\
h'.\textit{mm}[pa] &= \textit{combine}(h.\textit{mm}[pa], (h.p[i].\textit{memreq.data}, \\
&\quad h.p[i].\textit{memreq.mask})) \\
&= \textit{combine}(h_r.\textit{mm}[pa], (h.p[i].\textit{memreq.data}, \\
&\quad h.p[i].\textit{memreq.mask})) \\
&= h'_r.\textit{mm}[pa].
\end{aligned}$$

Case 2: $h \xrightarrow{a} h'$ is a locked memory write to physical address pa on processor i. The proof for this case is completely analogous to the previous case.

Case 3: $h \xrightarrow{a} h'$ is a TLB step of setting access/dirty bits in a PTE or a walk extension (which involves fetching of the PTE). From *inv-shared-ptes* (Invariant 4.15) we get that the address of the PTE is shared and complete the proof analogously to Case 1.

Now we consider the hardware steps performed by processors running in hypervisor mode ($h.p[i].\textit{asid} = 0$).

Case 3: $h \xrightarrow{a} h'$ is a core memory read from physical address pa on processor i. The reduced machine performs the same step, reading the physical memory. The byte addresses, which are supposed to be read from the memory are

$$bpa \in \textit{affected-byte-addr}(pa, \textit{memreq}[i].\textit{mask}).$$

From the ownership discipline it follows that the data for *bpa* may be present only in the store buffer of the processor i. If the memory read doesn't involve forwarding from the store buffer, then the data is not present in any of the store-buffers at all and the theorem holds. Otherwise, the copy of the data is taken from the store buffer and we have

$$\begin{aligned} h'.p[i].memres.data &= combine(0^{64}, (combine(h.mm[pa],\\ &\quad forward(h.sb[i], pa)), h.p[i].memreq.mask))\\ &= combine(0^{64}, (combine(h.mm[pa],\\ &\quad forward(h.sb[i], pa)), h.p[i].memreq.mask))\\ &= combine(0^{64}, (h_r.mm[pa], h.p[i].memreq.mask))\\ &= h'_r.p[i].memres.data. \end{aligned}$$

Case 4: $h \xrightarrow{a} h'$ is a core memory write to the (owned) physical address *pa* on processor i. For the bytes of *pa* which are not affected by the write nothing is changed. The write is committed to the store buffer together with the write mask. The other store buffers do not have the data for the bytes of *pa*, which are modified by this write. It follows for all bytes $i < 8$ affected by the memory write:

$$byte_i(h'.sb[i].data[pa]) = byte_i(h'_r.mm[pa]).$$

The reduced machine performs the same step, writing directly to the physical memory and the theorem holds.

Case 5: $h \xrightarrow{a} h'$ is a step of store buffer $h.sb[i]$. The reduced machine makes an empty step. The proof for this case requires a case split on the type of the store buffer step and is analogous to previous cases.

Case 6: $h \xrightarrow{a} h'$ is a VMRUN step of processor i. The reduced machine performs the same step. The store buffer becomes now visible on the reduced machine. Since store buffer $h.p[i].sb$ is empty at the time when the step is triggered and buffer $h_r.p[i].sb$ is also empty according to the abstraction relation, we get

$$h'.p[i].sb = h'_r.p[i].sb = empty\text{-}sb()$$

and the abstraction relation holds after the step.

Now we do a case split on steps of the processor running in guest mode ($h.p[i].asid \neq 0$).

Case 7: $h \xrightarrow{a} h'$ is a core memory write to (guest) physical address *pa* on processor i. The reduced machine performs the same step, committing a store to the SB. From the ownership discipline it follows that no store buffers of processors in hypervisor mode contain stores to the bytes affected by this write. Hence, the main memory abstraction of the reduced machine is maintained.

Case 8: $h \xrightarrow{a} h'$ is an SB write to the main memory. The proof for this case is analogous to the previous case .

Case 9: $h \xrightarrow{a} h'$ is a VMEXIT step of processor i. Before the step the store buffers are empty on both machines. Hence, after the step the

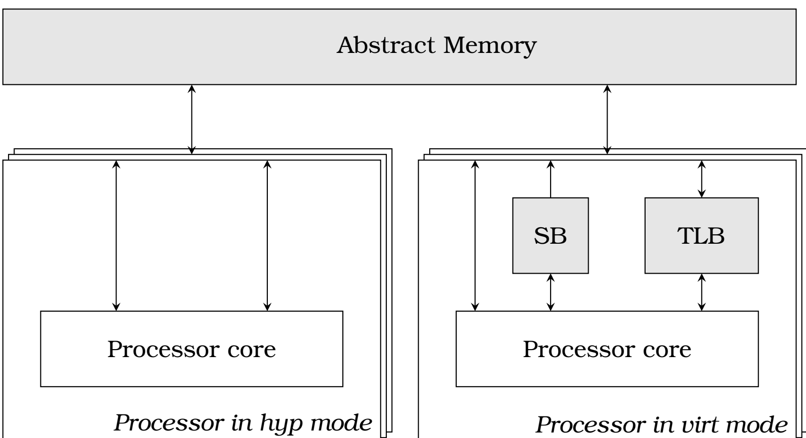

Figure 4.3: Reduced hardware machine.

abstraction relation holds.

All steps, which are not explicitly mentioned above, are equivalent on both machines and there is nothing to prove for them. □

4.5 TLB Reduction

A compiler is normally not aware of any address translation performed by the hardware MMU: it produces the code, which behaves as intended only when it is executed on a machine with sequentially consistent memory. In order to talk about compiler consistency of multi-threaded programs (where every thread is executed on a dedicated processor), we introduce another reduced hardware model, where address translation is not visible on processors running in hypervisor mode.

Note, that although the pointer to the top-level page table is set separately for every processor, MMU reduction can not be done on a per-processor basis. In order to run a multi-threaded program on a machine with reduced MMUs, we need to be sure that all processors do the same address mapping. Only then we can run programs operating on a shared memory. Moreover, we want to have a number of other processors running in the "translated" mode, which have their MMUs operating in the same address space as the "untranslated" processors do.

The simplest solution to this problem is to set up page tables for "untranslated" processors so that they provide the identity mapping. Further, we require the hypervisor code to run under the identity mapping. In the next section we formalize the properties of *the identity mapped page tables* (IMPTs).

4.5.1 Identity Mapped Page Tables

We require IMPTs to be located in the dedicated range of physical addresses:

$$IMPTAddr \subset \mathbb{B}^{bpa}.$$

We require this set of addresses to be disjoint from all other sets (i.e., from shared writable, read only, processor owned, and guest addresses) introduced in Section 4.3.1.

$$\forall bpa \in IMPTAddr : bpa \notin SharedAddr \cup ReadOnlyAddr \cup StackAddr \\ \wedge\ bpa \notin PrivateAddr \cup GuestAddr.$$

◄ **Invariant 4.21** Disjoint IMPT addresses

Further in this thesis we assume that partitioning of memory into sets of addresses is correct and is statically fixed. Hence, we assume that Invariant 4.21 always holds.

For IMPTs we introduce a predicate, which denotes that addresses of all entries of a page table are located in the set *IMPTAddr*:

$$impt\text{-}in\text{-}IMPTAddr(ba \in \mathbb{B}^{pfn}) \in \mathbb{B},$$
$$impt\text{-}in\text{-}IMPTAddr(ba) \stackrel{\text{def}}{=} pfn2bytes(ba) \subseteq IMPTAddr.$$

◄ **Definition 4.22** IMPT in *IMPTAddr*

The base address of the root IMPT is fixed by the constant *IMPTRootBA* s.t.

$$impt\text{-}in\text{-}IMPTAddr(IMPTRootBA).$$

For simplicity, we require all physical addresses to be mapped i.e., we have

$$\mathbb{B}^{pfn} = \mathbb{B}^{vpfn}.$$

We say that an IMPT is valid, if the following properties are satisfied:

- all entries of the IMPT are located at the address from *IMPTAddr*,
- all entries of the IMPT are marked as present and valid with A and D bits (for terminal PTs) set,
- all entries of the IMPT have *pat-idx* value identifying a "write-back" memory type,
- all entries of the IMPT have all rights enabled,
- if the IMPT is non-terminal, then all its entries point to other valid IMPTs,
- if the IMPT is non-terminal, then all its entries have the *pfn* field equal to the virtual PFN, which leads to this terminal IMPT.

With the following function we check that a given base address *ba* points to a valid subtree of IMPTs with the depth l and the virtual prefix *vpfn*. Let

$$pte = abs\text{-}pt(read(h.mm, ba \circ 0^9)).pte[vpfn.px[l]]$$

be the next PTE to be fetched for a given *vpfn*. Then the valid IMPT is defined

in the following way:

Definition 4.23 ► Valid IMPT tree

$$valid\text{-}im\text{-}tree(h \in RedHard_{sb}, vpfn \in \mathbb{B}^{vpfn}, ba \in \mathbb{B}^{pfn}, l) \in \mathbb{B}$$

$$\begin{aligned} valid\text{-}im\text{-}tree(h, vpfn, ba, l) \stackrel{\text{def}}{=} \\ & impt\text{-}in\text{-}IMPTAddr(ba) \\ & \wedge pte.p \wedge pte.a \wedge pte.valid \\ & \wedge pat\text{-}mt(pte.pat\text{-}idx) = WB \\ & \wedge pte.r.\{us, rw, ex\} = 1 \\ & \wedge (l > 1 \implies valid\text{-}im\text{-}tree(h, vpfn, pte.pfn, l-1)) \\ & \wedge (l = 1 \implies pte.d \wedge pte.pfn = vpfn). \end{aligned}$$

Now we can define an invariant, which ensures that all address translations for any virtual address from $\mathbb{B}^{vpfn}$ go only through valid IMPTs.

Invariant 4.24 ► Valid IM translations

name	$inv\text{-}valid\text{-}im\text{-}translations(h \in RedHardw_{sb})$
property	$\forall vpfn \in \mathbb{B}^{vpfn} : valid\text{-}im\text{-}tree(h, vpfn, IMPTRootBA, 4)$

To make sure that the properties of IMPTs don't get violated during the code execution, we need to know that the core never writes to the addresses in *IMPTAddr*. This is guaranteed by our ownership discipline for the reduced model, which is defined in the next section.

All incomplete walks with ASID 0 in TLBs of the reference hardware model have to be walks through the IMPTs. The $CR3$ register of processors running in hypervisor mode should always point to the root IMPT. All complete walks with ASID 0 should have $w.pfn$ and $w.vpfn$ fields equal (this guarantees identity mapping for address translations). Moreover, when the hypervisor is sleeping and the guest is running, the $CR3_{hyp}$ register should point to the root IMPT. We fix these properties in the following invariant.

Invariant 4.25 ► TLB walks through IMPTs

name	$inv\text{-}tlb\text{-}walks\text{-}impts(h \in RedHardw_{sb})$
property	$h.p[i].asid = 0 \implies h.p[i].CR3.pfn = IMPTRootBA,$ $h.p[i].asid \neq 0 \implies h.p[i].CR3_{hyp}.pfn = IMPTRootBA,$ $h.p[i].tlb[w] \wedge w.asid = 0 \wedge w.l \neq 0$ $\implies valid\text{-}im\text{-}tree(h, i, w.vpfn, w.pfn, w.l),$ $h.p[i].tlb[w] \wedge w.asid = 0 \wedge w.l = 0 \implies w.pfn = w.vpfn$

4.5.2 Registers

To make sure that all accesses in the system have "write-back" memory types we additionally have to maintain invariants over the registers, used in memory type calculations.

Since the content of the MTRR registers is considered to be fixed during initialization and may not change during machine execution, we simply require all physical addresses to have a "write-back" MTRR memory type:

Invariant 4.26 ► MTRR memory types

name	$mtrr\text{-}cacheable()$
property	$\forall i \in Pid, pfn \in \mathbb{B}^{pfn} : mtrr\text{-}mt(pfn) = WB$

To ensure that the top-level page table has a "write-back" memory type, we maintain an invariant on the value of the $CR3$ register. Additionally, we maintain the same property on the $CR3_{hyp}$ register making the invariant inductive after a VMEXIT event.

name	$\mathit{inv\text{-}cr3\text{-}cacheable}(h \in \mathit{Hardware})$
property	$\mathit{root\text{-}pt\text{-}memtype}(h.p[i].CR3) = WB$, $h.p[i].\mathit{asid} \neq 0 \implies \mathit{root\text{-}pt\text{-}memtype}(h.p[i].CR3_{hyp}) = WB$

◄ **Invariant 4.27** $CR3$ memory type

4.5.3 TLB-reduced Hardware Model

TLBs of the reduced model do not contain any walks in ASID 0, while having all the other walks copied from the underlying hardware layer:

$$\mathit{reduced\text{-}tlb}(\mathit{tlb} \in \mathit{Pid} \mapsto \mathit{Tlb}) \in \mathit{Pid} \mapsto \mathit{Tlb},$$
$$\mathit{reduced\text{-}tlb}(\mathit{tlb}) \stackrel{\text{def}}{=} \lambda i \in \mathit{Pid}, w \in \mathit{Walk} : w.\mathit{asid} \neq 0 \wedge \mathit{tlb}[i][w].$$

◄ **Definition 4.28** TLB abstraction

The hardware model with reduced TLBs is then defined in the following way:

$$\mathit{reduced\text{-}tlb\text{-}hw}(h \in \mathit{RedHardw}_{sb}) \in \mathit{RedHardw},$$
$$\mathit{reduced\text{-}tlb\text{-}hw}(h) \stackrel{\text{def}}{=} h[\mathit{tlb} \mapsto \mathit{reduced\text{-}tlb}(\mathit{tlb})].$$

◄ **Definition 4.29** Hardware reduction (TLBs)

Transitions of the fully reduced hardware are equivalent to the transitions of the SB-reduced model with the exception of the steps of processors running in hypervisor mode (i.e., with current ASID set to 0). The steps of these processors differ in the following way:

- all TLB steps except dropping of walks are empty (i.e., perform stuttering),
- core memory read, write and compare exchange operations are done directly to the virtual address and do not require an address translation,
- there are no core steps for triggering of the page fault (since it would never be triggered).

As an example, consider an updated core memory read step of a processor running in hypervisor mode.

label	$\mathit{core\text{-}memory\text{-}read}(i \in \mathit{Pid}, w \in \mathit{Walk})$
guard	$\mathit{asid}[i] = 0$, $\mathit{memreq}[i].\mathit{type} = \mathit{read}$, $\mathit{data} = \mathit{read}(\mathit{mm}, i, \mathit{memreq}[i].\mathit{va})$
effect	$\mathit{memres}'[i].\mathit{data} = \mathit{combine}(0^{64}, (\mathit{data}, \mathit{memreq}[i].\mathit{mask}))$ $\mathit{memres}'[i].\mathit{pf} = \mathit{no\text{-}page\text{-}fault}()$, $\mathit{memres}'[i].\mathit{vmexit} = 0$, $\mathit{memreq}'[i].\mathit{active} = 0$, $\mathit{memres}'[i].\mathit{ready} = 1$

◄ **Definition 4.30** Core memory read (reduced model)

Now we can prove a TLB reduction theorem.

Theorem 4.4 (TLB reduction). *Let all walks with ASID 0 be walks over the IMPTs and let the tlbres buffer contain no active page fault. Moreover, let reduction hold between states $h \in RedHardw_{sb}$ and $h_r \in RedHardw$. Then reduction is maintained after any step of the SB-reduced machine:*

$$\begin{aligned} & h \xrightarrow{a} h' \\ & \wedge\ \textit{inv-tlb-walks-impts}(h) \\ & \wedge\ h_r = \textit{reduced-tlb-hw}(h) \\ & \wedge\ h'_r = \textit{reduced-tlb-hw}(h') \\ & \implies h_r \xrightarrow{a} h'_r \\ & \qquad \wedge\ h'_r = \textit{reduced-tlb-hw}(h'). \end{aligned}$$

Proof. Steps of processors running in guest mode ($h.p[i].asid \neq 0$), except of removing walks in ASID 0, are equivalent to the steps of h. This is the case because TLB performs all steps (with the exception of walk removal) only with the walks in active ASID, which are simply copied from the original hardware to the reduced one.

Next, we consider steps of processors running in hypervisor mode ($h.p[i].asid = 0$). Note, that the step of triggering a page fault is not possible, because the predicate *tlb-fault-ready* will never hold under the invariant *inv-tlb-walks-impts*(h).

Case 1: $h \xrightarrow{a} h'$ is a core memory access (read, write, locked write, or compare exchange) from physical address pa on processor i, where

$$pa = w.pfn \circ h.p[i].memreq.va.off.$$

The reduced machine makes the same step, reading/writing the memory at the address $h.p[i].memreq.va$. From the invariant, we conclude that

$$pa = h.p[i].memreq.va.$$

Hence, the memory access is performed to the same address (returning the same result) and the theorem holds.

Case 2: $h \xrightarrow{a} h'$ is a TLB step of setting access and dirty bits. The write is performed directly to the main memory. From Invariant 4.25, we know that the PTE being written already has the access and dirty bits set. Hence, the memory is unchanged. This corresponds to an empty step of the reduced hardware.

Case 3: $h \xrightarrow{a} h'$ is a TLB step of removing a walk with ASID 0. This step corresponds to an empty step of the reduced hardware (since it doesn't contain any walks with ASID 0 anyway). Note, that if the step $h \xrightarrow{a} h'$ at the same time removes walks in different ASIDs including ASID 0, then the reduced machine will remove all the walks with the exception of the ones with ASID 0.

Case 4: $h \xrightarrow{a} h'$ is any other TLB step in ASID 0. This corresponds to an empty step of the reduced machine, and the theorem trivially holds.

All the other steps are performed identically on both machines. □

Note, that in general we don't necessarily need to have an identity mapping to make MMUs invisible. However, in order to have a reduced model, where MMUs of processors running in guest mode operate on the same abstract memory as the reduced processors do, we need the page tables of the reduced processors to be identity mapped[4]. Another advantage, which we get with the identity mapping is an easy way to obtain the allocated physical base address of data structures. For example, in our TLB virtualization algorithm (Chapter 9) we need to know the physical base address of shadow page tables to set them up correctly.

4.6 Putting It All Together

The reduced hardware configuration is obtained by applying the three reduction functions one after another:

$$reduced\text{-}hw(h \in Hardware) \in RedHardw$$
$$reduced\text{-}hw(h) \stackrel{\text{def}}{=} reduced\text{-}tlb\text{-}hw(reduced\text{-}sb\text{-}hw(reduced\text{-}ca\text{-}hw(h)))$$

◀ **Definition 4.31**
Hardware reduction

Now we unite the three reduction theorems presented in this chapter into one theorem.

Theorem 4.5 (Cache, SB, and TLB reduction). *Let the caches be consistent, the complete walks in TLBs and SBs have "write-back" memory types, and the ownership discipline hold. Moreover, let TLB reduction requirements be satisfied. Further, let hardware reduction hold between states $h \in Hardware$ and $h_r \in RedHardw$. Then reduction is maintained after any step of the*

[4]If we reduced MMUs under a mapping *hpa2spa*, which is not an identity mapping, we would have to consider different sets of addresses (i.e., *GuestAddr*, *SharedAddr*, etc.) for the reference model and for the reduced model. Moreover, non-reduced MMUs and processors running in guest mode would have to perform memory accesses under $hpa2spa^{-1}$ mapping rather than accessing the memory directly with the physical address.

The TLB ownership invariant (Invariant 4.36) in this case would change to

$$\begin{aligned}
&tlb[i][w] \wedge w.l \neq 0 \wedge w.asid \neq 0 \wedge w.asid = asid[i] \\
&\wedge bva \in qword2bytes(pte\text{-}addr(hpa2spa^{-1}(w.pfn), w.vpfn.px[w.l])) \\
&\implies bva \in SharedAddr \cup o[i], \\
&tlb[i][w] \wedge w.l = 0 \wedge w.asid \neq 0 \wedge w.asid = asid[i] \\
&\implies pfn2bytes(hpa2spa^{-1}(w.pfn)) \subseteq GuestAddr,
\end{aligned}$$

where *GuestAddr* and *SharedAddr* are sets of addresses defined for the reference model. Additionally, certain properties have to be enforced on *hpa2spa* mapping for the simulation proof to go through (e.g., injectivity).

reference machine:

$$
\begin{aligned}
& h \xrightarrow{a} h' \\
& \wedge\ \textit{inv-tlb-cacheable}(h), \\
& \wedge\ \textit{inv-sb-cacheable}(h.sb), \\
& \wedge\ \textit{inv-consistent-caches}(h.ca, h.mm), \\
& \wedge\ \textit{inv-ownership-discipline}(\textit{reduced-ca-hw}(h), o), \\
& \wedge\ \textit{inv-tlb-walks-impts}(\textit{reduced-sb-hw}(\textit{reduced-ca-hw}(h))) \\
& \wedge\ h_r = \textit{reduced-hw}(h) \\
& \implies\ h_r \xrightarrow{a} h_r' \\
& \qquad \wedge\ h_r' = \textit{reduced-hw}(h').
\end{aligned}
$$

Proof. The proof is done by applying Theorems 4.1, 4.3, 4.4 one after another. □

Our next goal is to make invariants needed for TLB, SB, and cache reduction inductive. Moreover, we want to define a programming discipline on the level of the reduced hardware model, which can be then used to transfer properties down to the reference model and to maintain the reduction invariants there. This will allow us to do the verification solely in the reduced model (e.g., in the program verifier) and still be sure that the simulation between the models holds.

All invariants needed for the reduction proofs can be divided into two groups:

- invariants talking about the part of the hardware state, which is fixed during initialization and remains constant afterwards. This includes the MTRR and PAT registers and content of the IMPTs,
- invariants talking about the part of the state, which is allowed to change during the code execution. This includes the ownership discipline and certain properties of page tables used for address translations when the guest is executed (i.e., properties of shadow page tables).

Establishing the properties of the first kind requires arguing about the hypervisor initialization code and the boot loader. Since we do not verify the hypervisor initialization, we simply require that these properties already hold at the time when we start execution of the machine. Maintaining these invariants afterwards requires that we never write to the part of the state they talk about.

The properties of the second kind, on the other hand, have to be explicitly maintained by the code we verify. And since we do the verification of this code w.r.t the reduced hardware model, we need to be able to transfer these properties from the reduced model downwards to the reference one, so that the preconditions of the reduction theorem are satisfied.

We start with defining the ownership discipline for the reduced model (we can not use the same discipline as for the reference model, because TLBs and SBs are partially invisible in the reduced model). Then we put everything together in a single top-level reduction theorem.

4.6.1 Ownership for Reduced Model

For the reduced model we split the ownership discipline into two parts: (i) properties for processors running in hypervisor mode and (ii) properties of processors running in guest mode.

Ownership for Hypervisor Mode

The requirement for disjoint ownership domains stays identical to the one for the reference model (Invariant 4.9).

The ownership requirements for read, compare-exchange, regular writes, and locked writes now only talk about the hardware state of processors in hypervisor mode. The property for steps of processors in guest mode follows from the TLB ownership invariant for the reduced model (Invariant 4.36), where we additionally require all complete walks to point to addresses from *GuestAddr*.

name	$\mathit{inv\text{-}owned\text{-}reads}_r(p \in \mathit{Pid} \mapsto \mathit{MemCore}, o \in \mathit{Ownership})$
property	$p[i].\mathit{asid} = 0 \wedge p[i].\mathit{memreq.type} = \mathit{read} \wedge \mathit{va} = p[i].\mathit{memreq.va}$ $\wedge\ \mathit{bva} \in \mathit{affected\text{-}byte\text{-}addr}(\mathit{va}, p[i].\mathit{memreq.mask})$ $\implies \mathit{bva} \notin \bigcup_{j \neq i} (o[j] \cup \mathit{StackAddr}_j)$

◀ **Invariant 4.32** Ownership for reads (reduced model)

name	$\mathit{inv\text{-}owned\text{-}writes}_r(p \in \mathit{Pid} \mapsto \mathit{MemCore}, o \in \mathit{Ownership})$
property	$p[i].\mathit{asid} = 0 \wedge p[i].\mathit{memreq.type} = \mathit{write}$ $\wedge\ \mathit{va} = p[i].\mathit{memreq.va}$ $\wedge\ \mathit{bva} \in \mathit{affected\text{-}byte\text{-}addr}(\mathit{va}, p[i].\mathit{memreq.mask})$ $\implies \mathit{bva} \in o[i] \cup \mathit{StackAddr}_i$

◀ **Invariant 4.33** Ownership for writes (reduced model)

name	$\mathit{inv\text{-}owned\text{-}atomic}_r(p \in \mathit{Pid} \mapsto \mathit{MemCore}, o \in \mathit{Ownership})$
property	$p[i].\mathit{asid} = 0 \wedge \mathit{va} = p[i].\mathit{memreq.va}$ $\wedge\ p[i].\mathit{memreq.type} \in \{\mathit{atomic\text{-}cmpxchng}, \mathit{locked\text{-}write}\}$ $\wedge\ \mathit{bva} \in \mathit{affected\text{-}byte\text{-}addr}(\mathit{va}, p[i].\mathit{memreq.mask})$ $\implies \mathit{bva} \in \mathit{SharedAddr} \cup \mathit{GuestAddr} \cup o[i] \cup \mathit{StackAddr}_i$

◀ **Invariant 4.34** Ownership for interlocked (reduced model)

Note, that Invariant 4.33 alone is not sufficient to maintain the property for owned stores (Invariant 4.14) in case of the ownership transfer (i.e., when the record o is being modified). Further we introduce a rule, which makes ownership transfer sound w.r.t to the ownership discipline.

Analogously to the ownership for the reference model we group all the properties (except the one for the TLB ownership) into a single invariant.

name	$\mathit{inv\text{-}ownership\text{-}discipline}_r(h \in \mathit{RedHardw}, o \in \mathit{Ownership})$
property	$\mathit{inv\text{-}disjoint\text{-}ownership\text{-}domains}_r(h.p, o)$, $\mathit{inv\text{-}owned\text{-}reads}_r(h.p, o)$, $\mathit{inv\text{-}owned\text{-}writes}_r(h.p, o)$, $\mathit{inv\text{-}owned\text{-}atomic\text{-}cmpxchng}_r(h.p, o)$

◀ **Invariant 4.35** Ownership discipline (reduced model)

Ownership for Guest Mode

For the ownership of PTEs pointed by incomplete walks in TLBs we strengthen Invariant 4.15 to talk about walks with $w.asid \neq 0$. We now require all PTEs to be located either in the shared writable memory or in the owned memory of the processor. For walks in ASID 0 the required ownership property follows from Invariant 4.25. Additionally, we require all complete walks with ASID other than zero to point to addresses from *GuestAddr*.

Invariant 4.36 ► TLB ownership (reduced model)

name	$\mathit{inv\text{-}tlb\text{-}ownership}_r(p \in \mathit{Pid} \mapsto \mathit{Core}, \mathit{tlb} \in \mathit{Pid} \mapsto \mathit{Tlb}, o \in \mathit{Ownership})$
property	$\mathit{tlb}[i][w] \wedge w.l \neq 0 \wedge w.asid \neq 0 \wedge w.asid = asid[i] \wedge bva \in \mathit{qword2bytes}(\mathit{pte\text{-}addr}(w.pfn, w.vpfn.px[w.l])) \implies bva \in \mathit{SharedAddr} \cup o[i]$, $\mathit{tlb}[i][w] \wedge w.l = 0 \wedge w.asid \neq 0 \wedge w.asid = asid[i] \implies \mathit{pfn2bytes}(w.pfn) \subseteq \mathit{GuestAddr}$

Note, that the property about the guest walks is stated only for the currently active ASID of the processor.

4.6.2 Ownership Transfer

Here we define an invariant which has to hold during ownership transfer from ownership setting o to o' in order for this transfer to be sound w.r.t to the ownership discipline and the hardware reduction. The ownership transfer can occur when the memory core makes a step from p to p'.

We allow the release of ownership on an address from some processor to occur only when the store buffer of this processor is empty. Since on the reduced machine store buffers might be invisible, we state the requirement on the processor core, rather than on the store buffer itself. In particular, we allow a processor to abandon ownership only when it performs a locked memory write[5].

Invariant 4.37 ► Ownership transfer (reduced model)

name	$\mathit{inv\text{-}safe\text{-}transfer}_r(p \in \mathit{Pid} \mapsto \mathit{MemCore}, p' \in \mathit{Pid} \mapsto \mathit{MemCore}, o \in \mathit{Ownership}, o' \in \mathit{Ownership})$
property	$bpa \in o[i] \wedge bpa \notin o'[i] \implies p[i].memres.ready = 0 \wedge p'[i].memres.ready = 1 \wedge p[i].memreq.type \in \{\mathit{atomic\text{-}cmpxchng}, \mathit{locked\text{-}write}\}$, $bpa \notin o[i] \wedge bpa \in o'[i] \implies bpa \in \mathit{PrivateAddr} \wedge bpa \notin \bigcup_{i \neq j} o'[j]$

4.6.3 Main Reduction Theorem

Safety for Reference Hardware

We group all properties of the reference model we need in order to prove reduction theorem and to maintain the properties after a step of the hardware

[5] In a C program this corresponds to acquiring/releasing of a lock, which obtains/abandons ownership of lock-protected data by a thread.

machine. We say that a hardware configuration which satisfies all reduction invariants is *safe*.

Invariant 4.38 Safe configuration (reference model)

name	*safe-conf*($h \in Hardware, o \in Ownership$)
property	*inv-valid-im-translations*(h), *inv-tlb-walks-impts*(*reduced-sb-hw*(*reduced-ca-hw*(h))), *mtrr-cacheable*(), *inv-cr3-cacheable*(h), *inv-tlb-cacheable*(h), *inv-sb-cacheable*($h.sb$), *inv-consistent-caches*($h.ca, h.mm$), *inv-ownership-discipline*(*reduced-ca-hw*(h), o),

A step of the hardware is safe when it starts and ends in a safe configuration. An execution sequence $h^0 \xrightarrow{\beta} h^n$, where $|\beta| = n$ and $n > 0$ is safe if it starts in a safe state and every step in this sequence also leads to a safe state[6]. The following predicate denotes that a sequence β from h^0 to h^n is safe starting with the ownership setting o and ending with the ownership setting o':

Definition 4.39 Safe sequence

$$\textit{safe-seq}(\beta, o, o') \stackrel{\text{def}}{=} \exists o^0, \ldots, o^n : o^0 = o \wedge o^n = o' \wedge \forall i \leq n : \textit{safe-conf}(h^i, o^i).$$

Note, that in the definition given above and in the upcoming definitions of functions which take as a parameter a sequence of hardware actions, we implicitly pass as another parameter a sequence of hardware configurations $h^0, h^1, \ldots, h^n$, produced by the sequence of actions. Later we use such functions only in the context where this sequence of configurations is well defined.

Since we don't explicitly fix the initial configuration of the reference machine h^0, we assume that any hardware configuration where the reduction invariant holds can be considered as initial one[7].

Safety for Reduced Hardware

Safety properties for the reduced model comprise properties for TLB walks with ASID other than zero (since we want to derive these properties in the program and then transfer them down to the reference model) and the programming discipline for the hypervisor program (i.e., for instructions executed in hypervisor mode). The programming discipline for hypervisor consists of the following requirements:

- the ownership discipline for the reduced model has to be maintained,
- if a write to the register *CR*3 is pending (as a result of a move to *CR*3 or a VMRUN), then the provided value of the register should have a "write-back" memory type,

[6] We don't require ownership transfer in execution of a reference hardware to be safe, but rather enforce this restriction on executions of a reduced model (see Definition 4.43).

[7] One can easily construct a trivial initial configuration, where the caches and TLBs are empty, no core request is pending, and identity mapped page tables are correctly initialized.

- if a VMRUN event was triggered, then the provided ASID should be different from 0,
- if after a pending write to $CR3$ the processor continues execution in ASID 0, then the *pfn* field of the new value of the register should point to the top-level IMPT.

Formally we write these requirements as follows.

Invariant 4.40 ► Hypervisor mode safety

name	$\mathit{safe\text{-}hyp\text{-}conf}_r(h \in \mathit{RedHardw}, o \in \mathit{Ownership})$
property	$\mathit{inv\text{-}ownership\text{-}discipline}_r(h, o)$, $h.p[i].\mathit{memreq.type} \in \{\mathit{mov2cr3}, \mathit{VMRUN}\}$ $\wedge\ h.p[i].\mathit{asid} = 0 \wedge h.p[i].\mathit{memreq.active}$ $\implies \mathit{root\text{-}pt\text{-}memtype}(h.p[i].\mathit{memreq.cr3in})$, $h.p[i].\mathit{memreq.type} = \mathit{VMRUN} \wedge h.p[i].\mathit{memreq.active}$ $\implies h.p[i].\mathit{memreq.asidin} \neq 0$, $h.p[i].\mathit{memreq.active} \wedge h.p[i].\mathit{memreq.type} \in \{\mathit{mov2cr3}\}$ $\wedge\ h.p[i].\mathit{asid} = 0$ $\implies h.p[i].\mathit{memreq.cr3in.pfn} = \mathit{IMPTRootBA}$

Safety for TLBs of the reduced model is stated as a separate predicate. We require all walks in TLBs with ASIDs other than 0 to have a "write-back" memory type and to point to PTEs which are located either in the shared or in the owned memory region. Moreover, all complete walks should point to an address from *GuestAddr*. Maintaining these properties requires arguing about the page tables, used for the translations when VMs are running.

Invariant 4.41 ► Safe TLBs (reduced model)

name	$\mathit{safe\text{-}tlbs}_r(h \in \mathit{RedHardw}, o \in \mathit{Ownership})$
property	$\mathit{inv\text{-}tlb\text{-}ownership}_r(h.p, h.tlb, o)$, $\forall i \in \mathit{Pid} : h.\mathit{asid}[i] \neq 0 \implies \mathit{cacheable\text{-}walks}(h.tlb[i], h.\mathit{asid}[i])$

Putting invariants 4.40 and 4.41 together we get the definition of a safe configuration of the reduced hardware machine.

Invariant 4.42 ► Safe configuration (reduced model)

name	$\mathit{safe\text{-}conf}_r(h \in \mathit{RedHardw}, o \in \mathit{Ownership})$
property	$\mathit{safe\text{-}hyp\text{-}conf}_r(h, o)$, $\mathit{safe\text{-}tlbs}_r(h, o)$

An execution sequence $h^0 \xrightarrow{\beta} h^n$ of a reduced machine, where $|\beta| = n$ and $n > 0$ is safe if it starts in a safe state and every step in this sequence also leads to a safe state. Moreover, if the ownership transfer occurs at some hardware step then the transfer also has to be safe. The following predicate denotes that a sequence β from h^0 to h^n is safe starting with the ownership setting o and ending with the ownership setting o':

Definition 4.43 ► Safe sequence (reduced machine)

$$\mathit{safe\text{-}seq}_r(\beta, o, o') \stackrel{\text{def}}{=} \exists o^0, \ldots, o^n : o^0 = o \wedge o^n = o' \wedge \forall i \leq n : \mathit{safe\text{-}conf}_r(h^i, o^i)$$
$$\wedge\ \forall i < n : \mathit{inv\text{-}safe\text{-}transfer}_r(h^i.p, h^{i+1}.p, o^i, o^{i+1}).$$

We also define a weaker predicate, which denotes that a given execution sequence is hypervisor-safe, i.e., only the safety of hypervisor steps and of the ownership transfer is guaranteed to hold:

Definition 4.44 Hypervisor-safe sequence

$$\begin{aligned} \textit{safe-hyp-seq}_r(\beta, o, o') \stackrel{\text{def}}{=} & \exists o^0, \ldots, o^n : o^0 = o \wedge o^n = o' \\ & \wedge \forall i \leq n : \textit{safe-hyp-conf}_r(h^i, o^i) \\ & \wedge \forall i < n : \textit{inv-safe-transfer}_r(h^i.p, h^{i+1}.po^i, o^{i+1}). \end{aligned}$$

Safety Transfer

Now we can prove a lemma, which ensures that if a step of the reduced machine is safe, then the same step of the reference machine is also safe.

Lemma 4.6 (Safety transfer). *Let safety requirements hold in state $h \in$ Hardware and $h \xrightarrow{a} h'$ be a step of the hardware machine. Further, let h_r, h'_r be respective states of the reduced hardware machine satisfying safety conditions. Then configuration h' is also safe:*

$$\begin{aligned} & h \xrightarrow{a} h' \\ & \wedge \textit{safe-conf}(h, o) \\ & \wedge h_r = \textit{reduced-hw}(h) \\ & \wedge h'_r = \textit{reduced-hw}(h') \\ & \wedge \textit{safe-conf}_r(h_r, o) \\ & \wedge \textit{safe-conf}_r(h'_r, o') \\ & \wedge \textit{inv-safe-transfer}_r(h_r.p, h'_r.p, o, o') \\ & \implies \textit{safe-conf}(h', o'). \end{aligned}$$

Proof. Unfolding *safe-conf* for the reference model we get the following statements to prove:

- *inv-valid-im-translations*(h'): from the ownership discipline it follows that no writes can be done to the range of addresses where identity mapped page tables are located, with the exception of MMU writes. All PTEs from the IMPTs are quad-word aligned. MMU is always writing to quad-word aligned entries, hence the only bits which could possibly be updated by MMU writes are A and D bits. Since these bits are always set in IMPTs, MMU writes have no effect and do not break the IMPT properties,
- *inv-tlb-walks-impts*(*reduced-sb-hw*(*reduced-ca-hw*(h'))): if a TLB is creating a new walk with ASID 0, then we know that the register $CR3$ points to the root IMPT. From *inv-valid-im-translations* we know that IMPTs have correct values and the property holds. If TLB is extending a walk with ASID 0, then the property follows from *inv-tlb-walks-impts*(*reduced-sb-hw*(*reduced-ca-hw*(h))), the definition of the memory abstraction from h to h_r, and the ownership discipline which guarantees that no SBs can have stores to IMPTs. If a move to $CR3$ register of a processor running in ASID 0 (or switching to ASID 0) is done, then from the programming discipline we know that the new $CR3$ register points to the root IMPT and the property holds,

- *inv-cr3-cacheable*(h'): from the programming discipline we know that the provided *CR*3 value always has a "write-back" memory type,
- *mtrr-cacheable*(): the property is always maintained since we never write MTRR registers,
- *inv-tlb-cacheable*(h'): If a new walk with ASID 0 is added to the TLB, then from *inv-valid-im-translations*, *inv-tlb-walks-impts* *inv-cr3-cacheable*, and *mtrr-cacheable* we know that it has a "write-back" memory type. The property for walks with ASID other than zero follows directly from *safe-conf*$_r(h'_r, o)$,
- *inv-sb-cacheable*$(h'.sb)$: if a new store is added to SB, then from *inv-tlb-cacheabe*(h') we know that this store has a cacheable memory type,
- *inv-consistent-caches*$(h'.ca, h'.mm)$: the property is shown by Lemma 4.2,
- *inv-ownership-discipline*(*reduced-ca-hw*$(h'), o')$: the parts of the invariant talking about ownership domains, memory reads, and memory compare-exchanges follow from the analogous properties of the ownership discipline for the reduced model and the safety of complete TLB walks (the fact that they point only to addresses from *GuestAddr*).
  If a new store is added to the store buffer, then from the ownership discipline for h we know that this store is done to an address either in the ownership domain of the processor (if it is running in hypervisor mode) or to the address from *GuestAddr* (if the processor is in guest mode). According to the invariant *inv-safe-transfer*$_r$ the processor could not give up the ownership of this address on a transition from h to h'. Hence, ownership for stores (Invariant 4.14) is maintained.
  If some TLB extends a walk with ASID 0, then we use *inv-tlb-walks-impts*(*reduced-sb-hw*(*reduced-ca-hw*(h))) to conclude that the *pfn* field of the new walk points to an address from the set *IMPTAddr*, which is disjoint from other ownership sets. Hence, ownership for PTEs is maintained. If a TLB adds a walk with ASID 0, the same property is ensured by *inv-valid-im-translations*. Ownership for PTEs pointed by walks with ASID other than 0 follows from the ownership discipline for the reduced model.
  If some processor gives up the ownership of an address during transition from h to h', then from *inv-safe-transfer*$_r$ we know that the SB of this processor is empty in h' and ownership for stores (Invariant 4.14) is maintained. If a processor acquires the ownership of an address which was not owned by anyone in configuration h, then no SBs of other processors can contain stores to this address. If a processor acquires the ownership of an address which was owned by another processor in h, then the store buffer of that processor has to be empty in h' and the ownership invariants are maintained.

□

Main Reduction Theorem

The purpose of the main reduction theorem (Theorem 4.7) is to guarantee that every trace of the reference model which starts in a safe state is also a trace

of the reduced model. To achieve this goal, we have to make sure that the following properties hold:

1. every safe sequence of steps of the reference model is also a sequence of steps of the reduced model (existence of a trace),
2. every unsafe sequence of steps of the reference model starting from a safe state is also a sequence of steps of the reduced model which leads to an unsafe state (soundness of reduction),
3. all sequences of steps of the reduced model are safe (this property has to be guaranteed by the compiler and the properties of the compiled program).

Formally we state the main reduction theorem in a slightly different way.

Theorem 4.7 (Main reduction theorem). *Let $h^0 \in Hardware$ be a safe initial hardware state and $h_r^0 \in RedHardw$ be a respective initial safe state of the reduced machine. Further, let every execution sequence of the reduced machine starting from h_r^0 be safe w.r.t to some ownership sequence. Then any execution sequence of the reference machine starting from h^0 is safe and is at the same time a (safe) sequence of the reduced machine:*

$$
\begin{aligned}
&\forall \beta, (h^0 \xrightarrow{\beta} h^n) : \\
&\quad \mathit{safe\text{-}conf}(h^0, o) \\
&\quad \wedge\, h_r^0 = \mathit{reduced\text{-}hw}(h^0) \\
&\quad \wedge\, (\forall \omega, h_r', (h_r^0 \xrightarrow{\omega} h_r') : \exists o' : \mathit{safe\text{-}seq}_r(\omega, o, o')) \\
&\quad \Longrightarrow \exists o' : \mathit{safe\text{-}seq}(\beta, o, o') \\
&\qquad\quad \wedge\, \exists (h_r^0 \xrightarrow{\beta} h_r^n) : h_r^n = \mathit{reduced\text{-}hw}(h^n).
\end{aligned}
$$

Proof. By induction on the step of the reference machine. Consider a step $h^i \xrightarrow{\beta_i} h^{i+1}$, where $h_r^i = \mathit{reduced\text{-}hw}(h^i)$ and $\mathit{safe\text{-}conf}(h^i, o^i)$ hold. Applying Theorem 4.5 we get $h_r^{i+1} = \mathit{reduced\text{-}hw}(h^{i+1})$, where $h_r^i \xrightarrow{\beta_i} h_r^{i+1}$. Now, with the assumption that all execution sequences of the reduced machine are safe, we get $\mathit{safe\text{-}conf}_r(h_r^{i+1}, o^{i+1})$. Finally, we apply Lemma 4.6 and get $\mathit{safe\text{-}conf}(h^{i+1}, o^{i+1})$. □

From Theorem 4.7 it follows, that if a certain property holds for all traces of the reduced machine (particularly we are interested in traces of the memory automaton of the reduced machine), then it also holds for all traces of the reference machine, under the assumption that both machines start executing from a safe configuration.

CHAPTER 5

Intermediate C (C-IL) Semantics

5.1 Sequential C-IL Semantics

5.2 Concurrent C-IL Semantics

5.3 C-IL Program Safety

5.4 Compiler Correctness

Despite the fact that informal specification for the C programming language first appeared more than 40 years ago there is still no agreement among computer scientists on what to consider the "formal C semantics". Moreover, the C programming language, as defined by ISO standards [ISO99], describes a whole class of semantics, which may differ depending on hardware architecture and compilers.

Since many high-level features of the C language (e.g., loops) are syntactic sugar and can be modeled with simpler C constructs (e.g., labels and gotos), we do not consider the complete C semantics. Instead, we present the semantics of the *C intermediate language* (C-IL), developed by Sabine Schmalz [SS12], which abstracts away some of the complex C constructs, while still being expressive enough for verification of low-level C code.

We present the operational semantics of the C-IL language and state a compiler correctness theorem, which establishes simulation between execution of the reduced hardware machine and the C-IL machine. Further, we define safety conditions on the C-IL level, which are necessary to derive the safety of the hardware execution of the reduced machine.

The formal C-IL semantics was designed with some specific low-level features, which made it possible (with a few extensions) to use it for the verification of the mixture of C and macro-assembly code, as well as for regular C verification [Sha12]. Another goal behind the development of the C-IL semantics was to use it as a basis for the paper-and-pencil soundness proof of the Microsoft's VCC [Mic12a] tool, which was used as the verification environment in the Verisoft XT project [The12]. However, due to the complexity of the VCC axiomatization system and the high-level memory model used there, the soundness proof of VCC still remains as future work even on paper.

Since one of the applications of the C-IL semantics is verification of low-level system code, such as hypervisors and OS kernels, which requires combination with high-level assembly languages, C-IL considers a byte-addressable memory, which includes the region allocated for the heap, and an abstract stack. Pointer arithmetic on global variables is fully supported, while on local variables it is restricted to calculating offsets inside local memories. Every memory access in C-IL includes dereferencing of the left-value, which is either a pointer to some part of the global memory, or is an offset in a local variable. Only assignments of primitive values (at most 64 bits) are supported.

Since the behavior of C in general depends on the underlying architecture and compiler, the C-IL semantics is parameterized with the information, obtained from the compiler. This information is necessary for expression evaluation and C-IL computations.

Note, that in the semantics presented in this chapter, we do not care much about the C syntax. We also do not model expressions with side effects, which again could be considered as syntactic sugar and implemented on top of the C-IL language.

5.1 Sequential C-IL Semantics

5.1.1 Types

Primitive types. The set of primitive types $\mathbb{T}_P$ consists of the signed and unsigned n-bit integers (usually we consider only sizes, which are multiple of four) and the type **void**:

Definition 5.1 ► Primitive types

$$\mathbb{T}_P \stackrel{\text{def}}{=} \{\textbf{void}\} \cup \{\mathbf{i}n, \mathbf{u}n \mid n \in \{8, 16, 32, 64\}\}.$$

Note, that we do not introduce an explicit type for boolean values, but rather use an integer type to model it.

Complex types. Let $\mathbb{T}_C$ denote the set of struct names. Then the set of C-IL types $\mathbb{T}$, including the subset of all pointer types $\mathbb{T}_{ptr} \subset \mathbb{T}$, is constructed in the following way:

Definition 5.2 ► C-IL types

- primitive types: $t \in \mathbb{T}_P \implies t \in \mathbb{T}$,
- struct types: $t_c \in \mathbb{T}_C \implies (\textbf{struct}\ t_c) \in \mathbb{T}$,
- (regular) pointer types: $t \in \mathbb{T} \implies \textbf{ptr}(t) \in \mathbb{T}_{ptr}$,
- array types: $t \in \mathbb{T}, n \in \mathbb{N} \implies \textbf{array}(t, n) \in \mathbb{T}_{ptr}$,
- function pointer types: $t \in \mathbb{T}, T \in \mathbb{T}^* \implies \textbf{fptr}(t, T) \in \mathbb{T}_{ptr}$.

Note, that the array type is also considered as a pointer type.

Qualified types. Regular C supports a number of type qualifiers, which are used as hints for a compiler on how to treat variables of these types. Such qualifiers can either give more or less freedom to a compiler when doing code optimization. For example, the **const** qualifier forbids writing to a variable and allows the compiler to do more optimizations relying on the fact that its value in never overwritten.

A **volatile** qualifier, on the other side, informs the compiler that the memory region might be accessed externally w.r.t the compiled program. The compiler in this case does not do reordering of memory accesses to such kind of variables. This concept is highly useful when arguing about compiler consistency for optimizing compilers. It is also widely used when doing memory mapped I/O and writing concurrent applications (especially for lock-free concurrent algorithms).

In order to define qualified types we introduce the set of type qualifiers $\mathbb{Q}$:

$$\mathbb{Q} \stackrel{\text{def}}{=} \{\textbf{volatile}, \textbf{const}\}.$$

◀ **Definition 5.3** Type Qualifiers

Now we inductively construct the set of qualified types $\mathbb{T}_\mathbb{Q}$ in exactly the same manner as we constructed the regular C-IL types. The set $\mathbb{T}_\mathbb{Q}$ contains the following qualified types:

◀ **Definition 5.4** Qualified C-IL types

- primitive types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_P \implies (q, t) \in \mathbb{T}_\mathbb{Q}$,
- struct types: $q \subseteq \mathbb{Q} \wedge t_c \in \mathbb{T}_C \implies (q, \textbf{struct}\ t_c) \in \mathbb{T}_\mathbb{Q}$,
- pointers: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_\mathbb{Q} \implies (q, \textbf{ptr}(t)) \in \mathbb{T}_\mathbb{Q}$,
- array types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_\mathbb{Q}, n \in \mathbb{N} \implies (q, \textbf{array}(t, n)) \in \mathbb{T}_\mathbb{Q}$,
- function pointer types: $q \subseteq \mathbb{Q} \wedge t \in \mathbb{T}_\mathbb{Q}, T \in \mathbb{T}_\mathbb{Q}^* \implies (q, \textbf{fptr}(t, T)) \in \mathbb{T}_\mathbb{Q}$.

Note, that the set of qualifiers $q \subseteq \mathbb{Q}$ might be empty, which allows as to trivially obtain a qualified type from unqualified one. Analogously, we can get unqualified type from the qualified one by simply dropping the qualifier away.

The function $qt2t(x \in \mathbb{T}_\mathbb{Q}) \in \mathbb{T}$ converts a qualified type to an unqualified one by throwing away type qualifiers:

$$qt2t(x) \stackrel{\text{def}}{=} \begin{cases} t & x = (q, t) \wedge t \in \mathbb{T}_P \\ \textbf{ptr}(qt2t(x')) & x = (q, \textbf{ptr}(x')) \\ \textbf{array}(qt2t(x'), n) & x = (q, \textbf{array}(x', n)) \\ \textbf{fptr}(qt2t(x'), \textbf{map}(qt2t, X)) & x = (q, \textbf{fptr}(x', X)) \\ \textbf{struct}\ t_c & x = (q, \textbf{struct}\ t_c). \end{cases}$$

◀ **Definition 5.5** Converting qualified type to unqualified

Type predicates. We define predicates, which check whether the provided type $t \in \mathbb{T}$ is a pointer type, an array type, or a function pointer type:

Definition 5.6 ► Pointer/array type predicates

$$isptr(t) \stackrel{\text{def}}{=} \exists t' : t = \mathbf{ptr}(t'),$$
$$isarray(t) \stackrel{\text{def}}{=} \exists t', n' : t = \mathbf{array}(t', n'),$$
$$isfptr(t) \stackrel{\text{def}}{=} \exists t', T : t = \mathbf{fptr}(t', T).$$

5.1.2 Values

Due to the fact that C-IL is designed to be used in conjunction with assembly and hardware models, most values are represented with bit or byte strings. The set of values *val* is defined as

Definition 5.7 ► C-IL values

$$val \stackrel{\text{def}}{=} val_{int} \cup val_{struct} \cup val_{ptr} \cup val_{lref} \cup val_{fptr} \cup val_{fun},$$

where each of the sets is defined in the following way:

- integer values - a value of an n-bit (unsigned or signed) integer is a bit string of the respective length:

$$n \in \{8, 16, 32, 64\} \wedge b \in \mathbb{B}^n \implies val(b, \mathbf{u}n) \in val_{int},$$
$$n \in \{8, 16, 32, 64\} \wedge b \in \mathbb{B}^n \implies val(b, \mathbf{i}n) \in val_{int},$$

- struct values - a value of a struct is represented by a sequence of byte strings:

$$t_c \in \mathbb{T}_C \wedge B \in (\mathbb{B}^8)^* \implies val(B, \mathbf{struct}\ t_c) \in val_{struct},$$

- global pointer and array values - a value $\mathbf{val}(b, t)$ of a pointer or an array consists of address b, and pointer type t:

$$b \in \mathbb{B}^{size_{ptr}} \wedge isptr(t) \vee isarray(t) \implies \mathbf{val}(b, t) \in val_{ptr},$$

  where $size_{ptr} \in \mathbb{N}$ is the size of the pointer (depends of the underlying architecture and for the x64 architecture we take $size_{ptr} = 64$),
- local pointer values (local references) - due to the stack abstraction, the values of pointers to local variables are represented by the name of the local variable v, the offset inside this variable o, the number of the local stack frame i, and the pointer type t:

$$v \in \mathbb{V}, o, i \in \mathbb{N}, t \in \mathbb{T}_{\mathbf{ptr}} \wedge isptr(t) \vee isarray(t) \implies \mathbf{lref}((v, o), i, t) \in val_{lref},$$

  where $\mathbb{V}$ denotes the set of variable names,
- function pointer values - a value $\mathbf{val}(b, t)$ of a function pointer consists of address b, where the compiled code of the function starts and the function pointer type t:

$$b \in \mathbb{B}^{size_{ptr}} \wedge isfptr(t) \implies \mathbf{val}(b, t) \in val_{fptr},$$

- (symbolic) function values - in C-IL for a function fn the function pointer may be undefined during expression evaluation (e.g., for inline

functions); to call such functions we use a symbolic value **fun**(fn):

$$fn \in \mathbb{F}_{name} \wedge isfptr(t) \implies \mathbf{fun}(fn, t) \in val_{fun},$$

where $\mathbb{F}_{name}$ is the set of function names.

5.1.3 Expressions and Statements

The sets of unary and binary operators $\mathbb{O}_1$ and $\mathbb{O}_2$ are defined in the following way:

◄ **Definition 5.8**
Unary and binary operators

$$\mathbb{O}_1 \subset \{\oplus \mid \oplus \in val \rightharpoonup val\}$$
$$\mathbb{O}_2 \subset \{\oplus \mid \oplus \in (val \times val) \rightharpoonup val\}$$
$$\mathbb{O}_1 \stackrel{\text{def}}{=} \{-, \sim, !\,\}$$
$$\mathbb{O}_2 \stackrel{\text{def}}{=} \{+, -, \star, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, |, \hat{}, \&\&, ||\}.$$

The set of C-IL expressions $\mathbb{E}$ is constructed recursively from the following sets of expressions:

◄ **Definition 5.9**
C-IL expressions

- 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: $e \in \mathbb{E} \wedge \oplus \in \mathbb{O}_1 \implies \oplus e \in \mathbb{E}$,
- binary operations: $e_1, e_2 \in \mathbb{E} \wedge \oplus \in \mathbb{O}_2 \implies (e_1 \oplus e_2) \in \mathbb{E}$,
- ternary operation: $e, e_1, e_2 \in \mathbb{E} \implies (e \; ? \; e_1 : e_2) \in \mathbb{E}$,
- 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: $t \in \mathbb{T}_\mathbb{Q} \implies sizeof(t) \in \mathbb{E}$,
- size of an expression: $e \in \mathbb{E} \implies sizeof(e) \in \mathbb{E}$,

where $\mathbb{F}$ denotes the set of field names.

In order to use standard syntax for the array access operation, we introduce the following notation:

$$a[i] \stackrel{\text{def}}{=} *(a + i).$$

Note, that C-IL supports only strictly typed expressions and implicit type casts have to be converted to explicit ones during the translation from C to C-IL.

The set $\mathbb{S}$ of C-IL statements consists of the following elements:

◄ **Definition 5.10**
C-IL statements
(part 1)

- assignment: $e_0, e_1 \in \mathbb{E} \implies (e_0 = e_1) \in \mathbb{S}$,
- goto: $l \in \mathbb{N} \implies (\mathbf{goto}\ l) \in \mathbb{S}$,
- if-not-goto: $l \in \mathbb{N}, e \in \mathbb{E} \implies (\mathbf{ifnot}\ e\ \mathbf{goto}\ l) \in \mathbb{S}$,
- function call: $e_0, e \in \mathbb{E}, E \in \mathbb{E}^* \implies (e_0 = \mathbf{call}\ e(E)) \in \mathbb{S}$,
- procedure call: $e \in \mathbb{E}, E \in \mathbb{E}^* \implies (\mathbf{call}\ e(E)) \in \mathbb{S}$,

- return from function and procedure: $e \in \mathbb{E} \implies (\textbf{return } e) \in \mathbb{S}$ and $\textbf{return} \in \mathbb{S}$.

In case of a function or procedure call, $E \in \mathbb{E}^*$ is the list of expressions passed to a function as function parameters.

Additionally, we include a number of special statements, which are abstractions of external assembly functions or inline assembly instructions used inside a hypervisor[1]. These statements include a compare-exchange operation and several statements used for hardware virtualization, which we later refer as *virtualization statements*. Execution of a virtualization statement does not have any effect on the C-IL memory/local stacks, except of increasing the program counter (which is also the case with assembly functions, implementing these statements in a real program). Later, when we extend the C-IL configuration with the hardware state (Chapter 7), we provide more meaningful semantics for these statements.

Additional C-IL statements are written in the following way:

Definition 5.11 ► C-IL statements (part 2)

- compare exchange:

$$e_0, e_1, e_2, e_3 \in \mathbb{E} \implies \textbf{cmpxchng}(e_0, e_1, e_2, e_3) \in \mathbb{S}.$$

  This is an abstraction of the respective compiler intrinsic if the compiler supports this operation, or an abstraction of the respective external assembly function, which performs a locked read-modify-write operation. Parameter e_0 is a return destination where the content of the memory has to be written, e_1 is a pointer to memory destination, e_2 holds the compared value, and e_3 contains the new value which has to be written to the memory destination if the comparison was successful,

- VMRUN instruction:

$$e_0, e_1, e_2 \in \mathbb{E} \implies \textbf{vmrun}(e_0, e_1, e_2) \in \mathbb{S}.$$

  This is an abstraction of the respective inline assembly instruction. Parameters e_0 and e_1 hold the values of the $CR3$ and ASID registers assigned to the guest by the hypervisor. Parameter e_2 holds a pointer to the struct, containing the data which has to be injected into *memreq* and *memres* buffers after a switch to guest mode has occurred. In hypervisor implementation all these values are not provided to VMRUN directly, but are rather taken from the architecture-specific control data structure (called *virtual monitor control block* or VMCB in the AMD x64 case [Adv11a, p. 373]). Here we don't want to stick to architecture specific code. Hence, we consider VMCB to be lying outside of the scope of our program and provide all VMRUN parameters explicitly when calling an abstract VMRUN statement,

- complete (all asids) TLB flush:

$$\textbf{completeflush} \in \mathbb{S}.$$

  For AMD x64 architecture this statement is an abstraction of setting the respective bit in the VMCB data structure and denoting that TLB has

[1] A complete TLB flush is an exceptional case, because it is implemented differently in AMD and Intel architectures and does not necessarily involve an external function call.

to be flushed at the next VMRUN call [Adv11a, p. 400]. In case of Intel x64 architecture the flush is done directly by executing an assembler instruction [Int11, p. 25-20]. Hence, for the Intel case **completeflush** statement is an abstraction of the respective external function. In order to model both AMD and Intel scenarios, we introduce a special auxiliary flag to our C-IL semantics, which denotes that TLB has to be flushed at the next VMRUN (see Section 5.1.4). This flag is set by the **completeflush** statement and is cleared by the next VMRUN. A compiler correctness theorem (Section 5.4.6) guarantees that if this bit is set, then the next VMRUN will be performed with the *complete-flush* bit being set in the *memreq* buffer of the processor executing the compiled code,

- INVLPGA instruction:

$$e_0, e_1 \in \mathbb{E} \implies \textbf{invlpga}(e_0, e_1) \in \mathbb{S}.$$

This is an abstraction of the respective external assembly function, which performs an address invalidation either in the ASID provided by the user, or in the currently active ASID (in the latter case parameter e_1 should evaluate to the current ASID value). Parameter e_0 holds the value of the invalidated virtual address and parameter e_1 holds the value of the ASID in which invalidation has to be done.

5.1.4 Configuration and Program

Configuration of a C-IL program consists of a byte-addressable global memory $\mathcal{M}$ and an abstract stack s, which is modelled as a list of C-IL frames. Additionally, we introduce an auxiliary $flush_{TLB}$ flag, which is an abstraction of the respective control bit, denoting that TLB has to be flushed at the next VMRUN execution:

$$conf_{C-IL} \stackrel{\text{def}}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, stack \in frame^*_{C-IL}, flush_{TLB} \in \mathbb{B}.],$$

◀ **Definition 5.12** C-IL configuration

where $\mathbb{B}_{gm} \subset \mathbb{B}^{size_{ptr}}$ is a set restricting the domain of the global memory. Note, that the global memory $\mathcal{M}$ is used only to store global and heap variables, and is not used to store local variables, which are stored in the abstract stack.

A single C-IL frame consists of a local memory, a return destination, a function name and a location:

$$frame_{C-IL} \stackrel{\text{def}}{=} [\mathcal{M}_{\mathcal{E}} \in \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*, rds \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}, f \in \mathbb{F}_{name}, loc \in \mathbb{N}],$$

◀ **Definition 5.13** C-IL frame

where $\mathcal{M}_{\mathcal{E}}$ maps variables names to their values, rds stores the pointer to the memory location where the return value has to be stored (if there is a return value), f is the name of the function/procedure which is executed in the given frame, and loc points to the next statement which has to be executed in the frame.

A program in C-IL consists of a function table $\mathcal{F}$, a set of global variables and their types $\mathcal{V}$, and the function T_F, which maps a struct type to the name of the struct and the list of its fields:

$$prog_{C-IL} \stackrel{\text{def}}{=} [\mathcal{F} \in \mathbb{F}_{name} \rightharpoonup fun_{C-IL}, \mathcal{V} \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*, T_F \in \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_{\mathbb{Q}})^*].$$

◀ **Definition 5.14** C-IL program

A single entry of a function table consists of the number of function parameters *npar*, a function body $\mathcal{P}$, and the set of local variables and their types $\mathcal{V}$:

Definition 5.15 ► Function table entry

$$fun_{C-IL} \stackrel{\text{def}}{=} [rettype \in \mathbb{T}_{\mathbb{Q}}, npar \in \mathbb{N}, \mathcal{P} \in \mathbb{S}^* \cup \{\mathbf{extern}\}, \mathcal{V} \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*],$$

where *retype* is the return value type of the function, $|fun_{C-IL}.\mathcal{V}| \geq npar$, and the first *npar* entries of $fun_{C-IL}.\mathcal{V}$ store the names and values of the function parameters.

If a function is not defined in the function table of the C-IL program, then it has to be marked with the keyword **extern**, which means that this function is an assembly/macro assembly function, and its execution is not governed by the C-IL semantics[2].

5.1.5 Context

In order to execute a C-IL program it is not enough to have the context and the program itself. For expression evaluation and C-IL transitions we need to get certain information from the compiler. This information, for instance, includes addresses of global variables in the memory, offsets of fields in structs, and sizes of struct types. The context $\theta \in context_{C-IL}$ provides the missing information from the compiler:

Definition 5.16 ► C-IL context

$$\begin{aligned} context_{C-IL} \stackrel{\text{def}}{=} [&alloc_{gvar} \in \mathbb{V} \rightharpoonup \mathbb{B}^{size_{ptr}}, \\ &\mathcal{F}_{addr} \in \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{size_{ptr}}, \\ &size_{struct} \in \mathbb{T}_C \rightharpoonup \mathbb{N}, \\ &size_t \in \mathbb{T}_P, \\ &offset \in \mathbb{T}_C \times \mathbb{F} \rightharpoonup \mathbb{N}, \\ &cast \in val \times \mathbb{T}_{\mathbb{Q}} \rightharpoonup val, \\ &endianness \in \{\mathbf{little}, \mathbf{big}\}] \end{aligned}$$

where $alloc_{gvar}$ maps the name of the global variable to its address, $\mathcal{F}_{addr}$ returns the address of the function for a given function name (undefined for inline and external functions), $size_{struct}$ maps a struct name to its size, *size_t* is the type of the value returned by the `sizeof` operator, *offset* returns the byte-offset of a given field in a struct, function *cast* does type casting of a given value to a given type, and *endianess* denotes the order in which bytes are stored in the memory.

Having the C-IL context we can now define a predicate, which checks whether a given function pointer corresponds to a given function name:

Definition 5.17 ► Is function

$$is\text{-}function(v \in val_{fptr}, f \in \mathbb{F}_{names} \theta \in context_{C-IL}) \mapsto \mathbb{B},$$

$$is\text{-}function(v, f, \theta) \stackrel{\text{def}}{=} v = \mathbf{val}(b, \mathbf{fptr}(t, T)) \wedge \theta.\mathcal{F}_{addr}(f) = b \vee v = \mathbf{fun}(f, \mathbf{fptr}(t, T))$$

[2] Currently we don't provide special treatment for external functions in our compiler correctness theorem. For more details on how one can treat external functions in C-IL semantics refer to [Sha12]

We also introduce the function, which calculates the size of a given type from the C-IL context and the type declaration:

$$size_{\partial}(t \in \mathbb{T}) \in \mathbb{N}.$$

The definition of this function is straightforward and we omit it here.

5.1.6 Memory Semantics

In operations with (global) memory accesses we have to deal with the fact that the memory is modelled as a flat bye-addressable mapping, while all C memory operations are typed. To perform conversions to and from byte strings we introduce the following functions:

Definition 5.18 Converting values to/from strings

$$
\begin{aligned}
&val2bytes_{\partial} \in val \rightharpoonup (\mathbb{B}^8)^*,\\
&bytes2val_{\partial} \in (\mathbb{B}^8)^* \times \mathbb{T} \rightharpoonup val,\\
&val2bytes_{\partial}(v) \stackrel{\text{def}}{=} \begin{cases} bytes(b) & v = \mathbf{val}(b,t) \land \partial.endianness = \mathbf{little}\\ bytes(\mathbf{rev}(b)) & v = \mathbf{val}(b,t) \land \partial.endianness = \mathbf{big}\\ \text{undefined} & \text{otherwise}\end{cases}\\
&bytes2val_{\partial}(B,t) \stackrel{\text{def}}{=} \begin{cases} \mathbf{val}(bits(B),t) & t \neq \mathbf{struct}\ t_C \land \partial.endianness = \mathbf{little}\\ \mathbf{val}(bits(\mathbf{rev}(B)),t) & t \neq \mathbf{struct}\ t_C \land \partial.endianness = \mathbf{big}\\ \text{undefined} & \text{otherwise.}\end{cases}
\end{aligned}
$$

The functions *bytes* and *bits* convert bit strings into byte strings and vice versa in an obvious way.

Now we define functions which read and write byte strings to the global memory of the C-IL machine. The first function reads a byte string of length s starting from address a:

Definition 5.19 Reading from the global memory

$$
\begin{aligned}
&read \in (\mathbb{B}_{gm} \mapsto \mathbb{B}^8) \times \mathbb{B}^{size_{ptr}} \times \mathbb{N} \rightharpoonup (\mathbb{B}^8)^*,\\
&read(\mathcal{M},a,s) \stackrel{\text{def}}{=} \begin{cases} read(\mathcal{M}, a + bin_{size_{ptr}}(1), s-1) \circ \mathcal{M}(a) & s > 0\\ \epsilon & \text{otherwise.}\end{cases}
\end{aligned}
$$

In case if $\exists b < s : (a + bin_{size_{ptr}}(b)) \notin \mathbb{B}_{gm}$, the function $read(\mathcal{M},a,s)$ is undefined.

Another function is used to write a provided byte string B to the global memory $\mathcal{M}$ starting at the address a:

Definition 5.20 Writing to the global memory

$$
\begin{aligned}
&write \in (\mathbb{B}_{gm} \mapsto \mathbb{B}^8) \times \mathbb{B}^{size_{ptr}} \times (\mathbb{B}^8)^* \rightharpoonup (\mathbb{B}_{gm} \mapsto \mathbb{B}^8),\\
&\forall x \in \mathbb{B}_{gm} : write(\mathcal{M},a,B)(x) \stackrel{\text{def}}{=} \begin{cases} B[\langle x\rangle - \langle a\rangle] & \langle x\rangle - \langle a\rangle \in [0 : |B|-1]\\ \mathcal{M}(x) & \text{otherwise.}\end{cases}
\end{aligned}
$$

In case if $\exists b < |B| : (a + bin_{size_{ptr}}(|B|)) \notin \mathbb{B}_{gm}$, the function $write(\mathcal{M},a,B)$ is undefined.

The following function reads a byte string of length s from local memory

$\mathcal{M}_\mathcal{E}$ for local variable v starting at offset o:

Definition 5.21 ► Reading from a local memory

$$read \in (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times \mathbb{N} \rightharpoonup (\mathbb{B}^8)^*,$$
$$read(\mathcal{M}_\mathcal{E}, v, o, s) \stackrel{\text{def}}{=} \mathcal{M}_\mathcal{E}(v)[o+s-1] \circ \ldots \circ \mathcal{M}_\mathcal{E}(v)[o].$$

In case if $s + o > |\mathcal{M}_{confE}(v)|$ or $v \notin dom(\mathcal{M}_\mathcal{E})$ the function $read(\mathcal{M}_\mathcal{E}, v, o, s)$ is undefined.

To write a byte string B to variable v of local memory $\mathcal{M}_\mathcal{E}$ starting at offset o we use the following function:

Definition 5.22 ► Writing to a local memory

$$write \in (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times \mathbb{B}^8 \rightharpoonup (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*),$$
$$\forall w \in \mathbb{V}, i < |\mathcal{M}_\mathcal{E}(w)| :$$
$$write(\mathcal{M}_\mathcal{E}, v, o, B)(w)[i] \stackrel{\text{def}}{=} \begin{cases} B[i-o] & w = v \wedge i \in [o : o+s-1] \\ \mathcal{M}_\mathcal{E}(w)[i] & \text{otherwise.} \end{cases}$$

In case if $s > |\mathcal{M}_{confE}(v)|$ or $v \notin dom(\mathcal{M}_\mathcal{E})$ the function $write(\mathcal{M}_\mathcal{E}, v, o, B)$ is undefined.

Now we are ready to define functions which read and write C-IL values to/from C-IL configurations. First, we define the function which performs a read using the provided pointer value and the provided C-IL configuration.

Definition 5.23 ► Reading from the C-IL configuration

$$read_\theta \in conf_{C-IL} \times val \rightharpoonup val,$$
$$read_\theta(c, x) \stackrel{\text{def}}{=}$$
$$\begin{cases} bytes2val_\theta(read(c.\mathcal{M}, a, size_\theta(t)), t) & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ bytes2val_\theta(read(c.stack[i].\mathcal{M}_\mathcal{E}, v, o, size_\theta(t)), t) & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ read_\theta(c, \mathbf{val}(a, \mathbf{ptr}(t))) & x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ read_\theta(c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t))) & x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The following function is used for writing C-IL value y at the memory pointed by pointer x in the given C-IL configuration c:

Definition 5.24 ► Writing to the C-IL configuration

$$write_\theta \in conf_{C-IL} \times val \times val \rightharpoonup conf_{C-IL},$$
$$write_\theta(c, x, y) \stackrel{\text{def}}{=}$$
$$\begin{cases} c[\mathcal{M} \mapsto write(c.\mathcal{M}, val2bytes_\theta(x), val2bytes_\theta(y))] & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ c' & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ write_\theta(c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ write_\theta(c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t)), y) & x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $c' = c[c'.stack[i].\mathcal{M}_\mathcal{E} \mapsto write(c.stack[i].\mathcal{M}_\mathcal{E}, v, o, val2bytes_\theta(y))]$.

Note, that due to the fact that local variables are accessed by their reference, rather than by an explicit address, we do not support storing of local pointers in the memory.

5.1.7 Expression Evaluation

An expression from the program $\pi \in prog_{C-IL}$ is evaluated in the configuration $c \in conf_{C-IL}$ with the context $\vartheta \in context_{C-IL}$ by the partial function

$$[\cdot]_c^{\pi,\vartheta} \in \mathbb{E} \rightharpoonup val.$$

Before we proceed with the formal definition of expression evaluation, we have to define a number of auxiliary functions calculating types of C-IL values, variables, and expressions. This functions are also used in Section 5.3 when we define a safe execution of a C-IL program.

Types of values. The following function extracts the (unqualified) type from a given C-IL value:

Definition 5.25 Type of a value

$$\tau(v \in val) \in \mathbb{T},$$
$$\tau(v) \overset{\text{def}}{=} \begin{cases} t & v = \mathbf{fun}(y, t) \\ t & v = \mathbf{val}(y, t) \\ t & v = \mathbf{lref}((v, o), i, t). \end{cases}$$

Types of functions. The function $\tau_{fun}^{\pi}(f)$ extracts the type information for the function f from the function table of the program:

Definition 5.26 Type of a function

$$\tau_{fun}^{\pi} \in \mathbb{F}_{name} \rightharpoonup \mathbb{T}_{\mathbb{Q}},$$
$$\tau_{fun}^{\pi}(f) \overset{\text{def}}{=} \begin{cases} (\emptyset, \mathbf{funptr}(\pi.\mathcal{F}(f).rettype, [t_0, \ldots, t_{npar-1}])) & f \in dom(\pi.\mathcal{F}) \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $npar = \pi.\mathcal{F}(f).npar$ and $t_i = \mathbf{snd}(\pi.\mathcal{F}(f).\mathcal{V}[i])$.

Types of declared variables/fields. The set of all variables extracted from the list of variable declarations $\mathcal{V}$ is obtained with the following function:

Definition 5.27 Declared variables

$$decl(\mathcal{V} \in \mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^* \to 2^{\mathbb{V}},$$
$$decl(\mathcal{V}) = \begin{cases} \{v\} \cup decl(\mathcal{V}') & \mathcal{V} = \mathcal{V}' \circ (v, t) \\ \emptyset & \mathcal{V} = \epsilon. \end{cases}$$

The following function calculates the qualified type of a given variable from a respective declaration list:

Definition 5.28 Type of a variable.

$$\tau_V \in \mathbb{V} \times (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*) \rightharpoonup \mathbb{T}_{\mathbb{Q}},$$
$$\tau_V(v, \mathcal{V}) = \begin{cases} t & \mathcal{V} = (v, t) \circ \mathcal{V}' \\ \tau_V(v, \mathcal{V}') & \mathcal{V} = (v', t) \circ \mathcal{V}' \wedge v' \neq v \\ \text{undefined} & \mathcal{V} = \epsilon, \end{cases}$$

Another function is used to calculate the qualified type of a given field from

a respective declaration list:

Definition 5.29 ► Type of a field.

$$\tau_F \in \mathbb{F} \times (\mathbb{F} \times \mathbb{T}_\mathbb{Q})^*) \rightharpoonup \mathbb{T}_\mathbb{Q},$$

$$\tau_F(f, T) = \begin{cases} t & T = (f, t) \circ T' \\ \tau_F(f, T') & T = (f', t) \circ T' \wedge f' \neq f \\ \text{undefined} & T = \epsilon \end{cases}$$

The set of variables declared for the top-most stack frame is obtained with the following shorthand:

$$c.\mathcal{V}_{top}(\pi) = \pi.\mathcal{F}(c.stack[|c.stack| - 1].f).\mathcal{V}.$$

Types of Expressions. Now we can define the function which returns a qualified type of a given expression e in the program π and the context ∂:

Definition 5.30 ► Type of an expression

$$\tau_E^{\pi,\partial}(e \in \mathbb{E}) \in \mathbb{T}_\mathbb{Q}.$$

We define this function by a case split on the type of an expression:

- constant: $x \in val \implies \tau_E^{\pi,\partial}(x) = (\emptyset, \tau(x))$,
- unary operator: $e \in \mathbb{E}, \ominus \in \mathbb{O}_1 \implies \tau_E^{\pi,\partial}(\ominus e) = \tau_E^{\pi,\partial}(e)$,
- binary operator: $e_0, e_1 \in \mathbb{E}, \oplus \in \mathbb{O}_2 \implies \tau_E^{\pi,\partial}(e_0 \oplus e_1) = \tau_E^{\pi,\partial}(e_0)$,
- ternary operator: $e, e_0, e_1 \in \mathbb{E} \implies \tau_E^{\pi,\partial}(e \;?\; e_0 : e_1) = \tau_E^{\pi,\partial}(e_0)$,
- type cast: $e \in \mathbb{E}, t \in \mathbb{T}_\mathbb{Q} \implies \tau_E^{\pi,\partial}((t)e) = t$,
- variable name:

$$v \in \mathbb{V} \implies \tau_E^{\pi,\partial}(v) = \begin{cases} \tau_V(v, c.\mathcal{V}_{top}(\pi)) & v \in decl(c.\mathcal{V}_{top}(\pi)) \\ \tau_V(v, \pi.\mathcal{V}) & v \notin decl(c.\mathcal{V}_{top}(\pi)) \wedge v \in decl(\pi.\mathcal{V}) \\ (\emptyset, \textbf{void}) & \text{otherwise,} \end{cases}$$

- function name: $fn \in \mathbb{F}_{name} \implies \tau_E^{\pi,\partial}(fn) = \tau_{fun}^{\pi}(fn)$,
- pointer dereference:

$$e \in \mathbb{E} \implies \tau_E^{\pi,\partial}(*e) = \begin{cases} t & \tau_E{}^{\pi,\partial}(e) = (q, \textbf{ptr}(t)) \\ t & \tau_E{}^{\pi,\partial}(e) = (q, \textbf{array}(t, n)) \\ (\emptyset, \textbf{void}) & otherwise, \end{cases}$$

- address of an expression: $e \in \mathbb{E} \implies$

$$\tau_E^{\pi,\partial}(\&e) = \begin{cases} \tau_E^{\pi,\partial}(e') & e = *e' \\ (\emptyset, \textbf{ptr}(\tau_E^{\pi,\partial}(v)) & e = v \\ (\emptyset, \textbf{ptr}(q' \cup q'', X)) & e = (e').f \wedge \tau_E^{\pi,\partial}(e') = (q', \textbf{struct}\ t_C) \\ & \wedge \tau_F(f, \pi.T_F(t_C)) = (q'', X) \\ (\emptyset, \textbf{void}) & otherwise, \end{cases}$$

- field access: $e \in \mathbb{E}, f \in \mathbb{F} \implies \tau_E^{\pi,\partial}(e.f) = \tau_E^{\pi,\partial}(* \&(e).f)$,
- size of a type: $t \in \mathbb{T}_\mathbb{Q} \implies \tau_E^{\pi,\partial}(\textbf{sizeof}(t)) = (\emptyset, \partial.size_t)$,
- size of an expression: $e \in \mathbb{E} \implies \tau_E^{\pi,\partial}(\textbf{sizeof}(e)) = (\emptyset, \partial.size_t)$.

Zero values. In order to distinguish zero values from non-zero ones, we introduce a predicate, which checks whether a value is considered to be zero:

$$\mathbf{zero}_\theta \in val \rightharpoonup \mathbb{B},$$
$$\mathbf{zero}_\theta(x) \overset{\text{def}}{=} \begin{cases} a = 0^{size_\theta(t)} & x = \mathbf{val}(a,t) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

◀ **Definition 5.31** Zero-value check

Expression evaluation. We define expression evaluation function

$$[\cdot]_c^{\pi,\theta} \in \mathbb{E} \rightharpoonup val,$$

◀ **Definition 5.32** Expression evaluation

by a case split on the type of the expression:

- constant: $x \in val \implies [x]_c^{\pi,\theta} = x$,
- unary operator: $e \in \mathbb{E}, \ominus \in \mathbb{O}_1 \implies [\ominus e]_c^{\pi,\theta} = \ominus[e]_c^{\pi,\theta}$,
- binary operator: $e_0, e_1 \in \mathbb{E}, \oplus \in \mathbb{O}_2 \implies [e_0 \oplus e_1]_c^{\pi,\theta} = [e_0]_c^{\pi,\theta} \oplus [e_1]_c^{\pi,\theta}$,
- ternary operator: $e, e_0, e_1 \in \mathbb{E} \implies$

$$[(e \; ? \; e_0 : e_1)]_c^{\pi,\theta} = \begin{cases} [e_0]_c^{\pi,\theta} & \neg\mathbf{zero}_\theta([e]_c^{\pi,\theta}) \\ [e_1]_c^{\pi,\theta} & \text{otherwise,} \end{cases}$$

- type cast: $e \in \mathbb{E}, t \in \mathbb{T}_\mathbb{Q} \implies [(t)e]_c^{\pi,\theta} = \theta.cast([e]_c^{\pi,\theta}, qt2t(t))$,
- function name: $fn \in \mathbb{F}_{name} \implies$

$$[fn]_c^{\pi,\theta} = \begin{cases} \mathbf{val}(\theta.\mathcal{F}_{adr}(fn), qt2t(\tau_{fun}^\pi(fn))) & fn \in dom(\pi.\mathcal{F}) \wedge fn \in dom(\theta.\mathcal{F}_{adr}) \\ \mathbf{fun}(fn, qt2t(\tau_{fun}^\pi(fn))) & fn \in dom(\pi.\mathcal{F}) \wedge fn \notin dom(\theta.\mathcal{F}_{adr}) \\ \text{undefined} & \text{otherwise,} \end{cases}$$

- pointer dereference: $e \in \mathbb{E} \implies$

$$[*e]_c^{\pi,\theta} = \begin{cases} read_\theta(c, [e]_c^{\pi,\theta}) & (\tau([e]_c^{\pi,\theta}) = \mathbf{ptr}(t) \wedge \neg isarray(t)) \\ & \vee \tau([e]_c^{\pi,\theta}) = \mathbf{array}(t,n)) \\ \mathbf{val}(a, \mathbf{array}(t,n)) & [e]_c^{\pi,\theta} = \mathbf{val}(a, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{lref}((v,o), i, \mathbf{array}(t,n)) & [e]_c^{\pi,\theta} = \mathbf{lref}((v,o), i, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \text{undefined} & \text{otherwise,} \end{cases}$$

- address of an expression: $e \in \mathbb{E} \implies$

$$[\&e]_c^{\pi,\theta} = \begin{cases} [e']_c^{\pi,\theta} & e = *e' \\ \mathbf{lref}((v,0), |c.stack| - 1, \mathbf{ptr}(t')) & e = v \wedge v \in decl(c.\mathcal{V}_{top}(\pi)) \\ \mathbf{val}(\theta.alloc_{gvar}(v), \mathbf{ptr}(t'')) & e = v \wedge v \notin decl(c.\mathcal{V}_{top}(\pi)) \\ & \wedge v \in decl(\pi.\mathcal{V}_G) \\ \sigma_\theta([\&e']_c^{\pi,\theta}, f) & e = (e').f \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $t' = qt2t(\tau_V(v, c.\mathcal{V}_{top}(\pi)))$ and $t'' = qt2t(\tau_V(v, \pi.\mathcal{V}_G))$. The function $\sigma_\theta^\pi \in val \times \mathbb{F} \rightharpoonup val$ is a field reference function and is used to calculate

the pointer or the local reference of a field in a variable. We omit giving the formal definition for this function here,

- variable name: $v \in \mathbb{V} \implies [v]_c^{\pi,\theta} = [*\&v]_c^{\pi,\theta}$,
- field access: $e \in \mathbb{E}, f \in \mathbb{F} \implies [(e).f]_c^{\pi,\theta} = [*\&(e).f]_c^{\pi,\theta}$,
- size of a type:

$$t \in \mathbb{T_Q} \implies [\mathbf{sizeof}(t)]_c^{\pi,\theta} = \mathbf{val}(bin_{size_\theta(\theta.size_t)}(size_\theta(qt2t(t))), \theta.size_t),$$

- size of an expression: $e \in \mathbb{E} \implies [\mathbf{sizeof}(e)]_c^{\pi,\theta} = [\mathbf{sizeof}(\tau([e]_c^{\pi,\theta}))]_c^{\pi,\theta}$.

5.1.8 Operational Semantics

Notation. First, we introduce a number of shorthands which make it easier to argue about components of the frame i of the C-IL configuration $c \in conf_{C-IL}$:

$$c.\mathcal{M}_i \overset{\text{def}}{=} c.stack[i].\mathcal{M} \quad c.rds_i \overset{\text{def}}{=} c.stack[i].rds,$$
$$c.f_i \overset{\text{def}}{=} c.stack[i].f \quad c.loc_i \overset{\text{def}}{=} c.stack[i].loc.$$

The index of the top most frame is computed by the function $top(c \in conf_{C-IL}) \in \mathbb{N}$, where

Definition 5.33 ► Top most stack frame

$$top(c) \overset{\text{def}}{=} |c.stack| - 1.$$

To identify the components of the top level stack frame we use the following notation for $x \in \{\mathcal{M}, rds, f, loc\}$:

$$c.x_{top} \overset{\text{def}}{=} c.x_{top(c)}.$$

Auxiliary functions. Now we start with introducing auxiliary functions which are used to calculate the C-IL configuration after execution of a single step. The function computing the next statement to be executed in a given C-IL configuration is defined using information from the top-most stack frame and from the function table of the program:

Definition 5.34 ► Next statement

$$stmt_{next}(c \in conf_{C-IL}, \pi \in prog_{C-IL}) \mapsto \mathbb{S},$$
$$stmt_{next}(c, \pi) \overset{\text{def}}{=} \pi.\mathcal{F}(c.f_{top}).\mathcal{P}[c.loc_{top}].$$

The function computing the C-IL configuration where the location counter is incremented by one is defined as follows:

Definition 5.35 ► Incrementing location counter

$$inc_{loc}(c \in conf_{C-IL}) \mapsto conf_{C-IL},$$
$$inc_{loc}(c) \overset{\text{def}}{=} c[loc_{top} \mapsto loc_{top} + 1].$$

Next, we define the function which removes the top-most frame from the C-IL configuration:

Definition 5.36 ► Removing top-most frame

$$drop_{frame}(c \in conf_{C-IL}) \mapsto conf_{C-IL},$$
$$drop_{frame}(c) \overset{\text{def}}{=} c[stack \mapsto \mathbf{tl}(stack)].$$

Another function is used to assign a given value to the location counter of the top most stack frame:

$$set_{loc}(c \in conf_{C-IL}, l \in \mathbb{N}) \mapsto conf_{C-IL},$$
$$set_{loc}(c, l) \stackrel{\text{def}}{=} c[loc_{top} \mapsto l].$$

◀ **Definition 5.37**
Setting location counter

The operational semantics of C-IL is defined by a case split on the type of the statement which has to be executed next in the given C-IL configuration.

Assignment. In case of an assignment operation, we store the result of the right-hand expression evaluation at the location identified by the left-hand expression, and increment the program counter.

$$\frac{stmt_{next}(c, \pi) = (e_0 = e_1)}{\pi, \partial \vdash c \rightarrow inc_{loc}(write_{\partial}(c, [\&e_0]_c^{\pi,\partial}, [e_1]_c^{\pi,\partial}))}$$

◀ **Definition 5.38**
Assignment

Goto. In case of a goto operation, we update the value of the current location counter of the top most stack frame with the provided value.

$$\frac{stmt_{next}(c, \pi) = \textbf{goto}\ l}{\pi, \partial \vdash c \rightarrow set_{loc}(c, l)}$$

◀ **Definition 5.39**
Goto

If-Not-Goto. This statement is used to model conditional jumps, which are used e.g., for implementing while- and for-loops on top of the C-IL semantics. The resulting C-IL configuration depends on the result of the conditional expression evaluation. Hence, we define two rules: one for the case when the expression is evaluated to zero (success), and another for the case when it is evaluated to a non-zero value (failure). As a result of the statement execution the location counter of the top-most stack frame is either set to the provided value (in case of success), or is incremented by one (in case of failure).

$$\frac{stmt_{next}(c, \pi) = \textbf{ifnot}\ e\ \textbf{goto}\ l \qquad \textbf{zero}_{\partial}([e]_c^{\pi,\partial})}{\pi, \partial \vdash c \rightarrow set_{loc}(c, l)}$$

◀ **Definition 5.40**
IfNotGoto (success)

$$\frac{stmt_{next}(c, \pi) = \textbf{ifnot}\ e\ \textbf{goto}\ l \qquad \neg\textbf{zero}_{\partial}([e]_c^{\pi,\partial})}{\pi, \partial \vdash c \rightarrow inc_{loc}(c)}$$

◀ **Definition 5.41**
IfNotGoto (failure)

Function call. In case the next statement is a call to a function or a procedure, we nondeterministically choose a new stack frame *frame*, which

satisfies the conditions of the following predicate:

Definition 5.42 ► New stack frame

$$call_{frame} \in conf_{C-IL} \times prog_{C-IL} \times context_{C-IL} \times \mathbb{F}_{name} \times \mathbb{E}^* \times frame_{C-IL} \mapsto \mathbb{B}$$

$$\begin{aligned} &call_{frame}(c, \pi, \partial, f, E, frame) \stackrel{\text{def}}{=} \\ &\quad \forall i \in [1 : npar - 1] : frame.\mathcal{M}_{\mathcal{E}}(v_i) = val2bytes_{\partial}([E[i]]_c^{\pi,\partial}) \\ &\quad \wedge \forall i \in [npar : |\mathcal{V}| - 1] : |frame.\mathcal{M}_{\mathcal{E}}(v_i)| = size_{\partial}(t_i) \\ &\quad \wedge frame.loc = 0 \\ &\quad \wedge frame.f = f \\ &\quad \wedge frame.rds = \begin{cases} [\&e_0]_c^{\pi,\partial} & stmt_{next}(c, \pi) = (e_0 = \textbf{call } e(E)) \\ \bot & stmt_{next}(c, \pi) = \textbf{call } e(E), \end{cases} \end{aligned}$$

where f is the name of the function, E is a list of expressions passed to the function as the function parameters, $\mathcal{V}$ is a set of local variables and their types ($\mathcal{V} = \pi.\mathcal{F}(f).\mathcal{V}$), (v_i, t_i) is the i-th variable declaration from $\mathcal{V}$, and $npar$ is the number of function parameters ($npar = \pi.\mathcal{F}(f).npar$). Note, that the initial content of the local variables (other than function parameters) is not fixed and can be chosen non-deterministically.

As a result of function call execution we push the new frame to the stack and increment the location counter.

Definition 5.43 ► Function call

$$\frac{\begin{array}{c} stmt_{next}(c, \pi) = \textbf{call } e(E) \vee stmt_{next}(c, \pi) = (e_0 = \textbf{call } e(E)) \\ is\text{-}function([e]_c^{\pi,\partial}, f) \quad \partial.\mathcal{F}(f).\mathcal{P} \neq \textbf{extern} \quad call_{frame}(c, f, \pi, \partial, E, frame_{new}) \end{array}}{\pi, \partial \vdash c \rightarrow inc_{loc}(c[stack \mapsto frame_{new} \circ c.stack])}$$

Function return. We define separate rules for return from a function (with a return destination) and for a return from a procedure (without a return destination). As a result of statement execution we drop the top-most frame and in case of return from a function write the result of the execution to the return destination.

Definition 5.44 ► Function return with result

$$\frac{stmt_{next}(c, \pi) = \textbf{return} \vee (stmt_{next}(c, \pi) = \textbf{return } e \wedge c.rds_{top} \neq \bot)}{\pi, \partial \vdash c \rightarrow write^{\partial}(drop_{frame}(c), c.rds_{top}, [e]_c^{\pi,\partial})}$$

Definition 5.45 ► Function return without result

$$\frac{stmt_{next}(c) = \textbf{return } e \qquad c.rds_{top} = \bot}{\pi, \partial \vdash c \rightarrow drop_{frame}(c)}$$

Compare-exchange. Additionally to a regular assignment we introduce a compare-exchange operation to the C-IL semantics. Since this operation has to be done atomically, we can model its impact on the C-IL memory. We distinguish between two cases: when compare operation succeeds and when

it fails.

$$\frac{stmt_{next}(c, \pi) = \textbf{cmpxchng}(rds, dest, cmp, exchng) \quad read_{\partial}(c, [dest]_c^{\pi,\partial}) = [cmp]_c^{\pi,\partial} \quad c' = write^{\partial}(c, [\&rds]_c^{\partial,\pi}, read^{\partial}(c, [dest]_c^{\pi,\partial}))}{\pi, \partial \vdash c \rightarrow inc_{loc}(write_{\partial}(c', [dest]_c^{\pi,\partial}, [exchng]_c^{\pi,\partial}))}$$

◀ **Definition 5.46**
Compare-exchange (success)

$$\frac{stmt_{next}(c, \pi) = \textbf{cmpxchng}(rds, dest, cmp, exchng) \quad read_{\partial}(c, [dest]_c^{\pi,\partial}) \neq [cmp]_c^{\pi,\partial} \quad c' = write^{\partial}(c, [\&rds]_c^{\pi,\partial}, read_{\partial}(c, [dest]_c^{\pi,\partial}))}{\pi, \partial \vdash c \rightarrow inc_{loc}(c')}$$

◀ **Definition 5.47**
Compare-exchange (failure)

Virtualization statements. The effect of execution of any of the virtualization statements is not visible on the C-IL level. The only result which we see is the increase of the location counter and modification of the $flush_{TLB}$ bit.

$$\frac{stmt_{next}(c, \pi) = \textbf{completeflush}}{\pi, \partial \vdash c \rightarrow inc_{loc}(c[flush_{TLB} \mapsto 1])}$$

◀ **Definition 5.48**
Complete flush step

$$\frac{stmt_{next}(c, \pi) = \textbf{vmrun}(e_0, e_1, e_2)}{\pi, \partial \vdash c \rightarrow inc_{loc}(c[flush_{TLB} \mapsto 0])}$$

◀ **Definition 5.49**
VMRUN step

$$\frac{stmt_{next}(c, \pi) = \textbf{invlpga}(e_0, e_1)}{\pi, \partial \vdash c \rightarrow inc_{loc}(c)}$$

◀ **Definition 5.50**
INVLPG step

The third parameter in the abstract VMRUN statement is a pointer to the data which, has to be injected into the *memres* buffer. Since the whole VMRUN statement in our semantics is just an abstraction, and on a real machine parameters of the injected request depend on the correct setting of the fields of the VMCB data structure, we do not want to define this data in full detail here. Nevertheless, later we want to identify that a request injected into the *memres* buffer is exactly the one defined by the third parameters of the VMRUN statement in C-IL. To do this, we introduce an uninterpreted function which takes this pointer and the current C configuration and returns an instance of the type *InjectData*:

$$inject\text{-}data^{\pi,\partial} \in conf_{C-IL} \times \mathbb{E} \rightharpoonup InjectData.$$

5.2 Concurrent C-IL Semantics

Let $Tid \subset \mathbb{N}$ be a set containing IDs of the C-IL threads. Then a configuration of the parallel C-IL semantics consists of a shared memory, an array of local

memory stacks, and an array of $flush_{TLB}$ flags:

Definition 5.51 ► Parallel C-IL configuration

$$conf_{CC-IL} \overset{\text{def}}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, stack \in Tid \mapsto frame^*_{C-IL}, flush_{TLB} \in Tid \mapsto \mathbb{B}].$$

The sequential configuration of a thread $t \in Tid$ is denoted by $c(t) := (c.\mathcal{M}, c.stack(t), c.flush_{TLB}(t))$ and the step of a thread t is denoted by

$$\pi, \theta \vdash c(t) \to c'(t).$$

A step of the concurrent C-IL semantics is a step of some thread operating on the shared memory and on its local stack.

Definition 5.52 ► Step of concurrent C-IL

$$\frac{\pi, \theta \vdash c(t) \to (\mathcal{M}', stack', flush'_{TLB}) \quad c' = (\mathcal{M}', c.stack[t \mapsto stack'], c.flush_{TLB}[t \mapsto flush'_{TLB}])}{\pi, \theta \vdash c \to c'}$$

To denote that transition of a concurrent configuration from c to c' involves only steps of a thread $t \in Tid$ (leaving local stacks of other threads unchanged) we write

$$\pi, \theta \vdash c \to_t c'.$$

To denote a non-empty sequence of steps (either for a particular thread or for the whole C-IL configuration) we use the symbol $\to^+$. For example, a sequence of steps of a thread $t \in Tid$ is denoted by

$$\pi, \theta \vdash c \to^+_t c'.$$

Analogously, we use $\to^*$ to denote a possibly empty sequence of C-IL steps.

Note, that a program in concurrent C-IL has the same format as in sequential C-IL (Definition 5.14).

5.3 C-IL Program Safety

Compiler correctness can be defined only if SB, cache, and TLB reduction holds. To ensure that reduction holds, we have to know that the program obeys a certain programming discipline. We call such a program *safe*. Further, the compiler has to guarantee that its output also satisfies certain rules (given that the program is safe), which results in the safe execution sequence of the reduced hardware machine (see Theorem 4.7).

In this section we define the programming discipline for C-IL, which is largely based on the programming discipline for the reduced hardware model defined in Section 4.6.3. We start with defining the ownership for C-IL.

5.3.1 C-IL Ownership

In the scope of this thesis we assume that any C-IL program has as many threads as the number of hardware processors executing this program i.e.,

$$Tid = Pid.$$

Moreover, a C-IL configuration has flat byte-addressable memory, which resembles the memory layout of the reduced hardware machine. Hence, ownership of C-IL addresses by a given thread should imply ownership of the same addresses by a hardware processor executing this thread.

An ownership set $o[i]$ from Section 4.3 includes addresses from the global memory of the C-IL program which are owned by thread i. Addresses from the part of the physical memory where the local stack of thread i is located are gathered in the address set $StackAddr_i$, which is also owned by the processor i. On the C-IL level we have to consider only ownership for the global memory, since all local variables are thread-local by default. As a result, we can use ownership setting $o \in Ownership$ from Section 4.3 to define safety of a C-IL program.

We say that a given C-IL state satisfies the ownership discipline when any expression and any statement, which could be evaluated/performed from this state satisfies this discipline. Expressions and statements which satisfy the ownership discipline and some additional restrictions on input parameters (e.g., for a VMRUN statement) are also called *safe*. Note, that operations on the stack (both reads and writes) are always safe. Further we proceed with defining ownership safety for C-IL expressions and statements[3].

5.3.2 Safe Expressions

A quick look at the C-IL expression evaluation (Section 5.1.7) tells us that only two kinds of expressions involve a read from the global memory. They are

- evaluation of a global variable and
- dereferencing of another expression.

In the C-IL semantics dereferencing a pointer may result in several (up to 8) byte-addresses being accessed in the memory. To define safety for memory reads and writes we first have to calculate all byte addresses, which belong to a given pointer. We call these addresses *support* of a pointer:

$$support^{\vartheta}(p \in val) \mapsto 2^{\mathbb{B}^{64}}$$

$$support^{\vartheta}(p) = \begin{cases} \{a + i \mid 0 \leq \langle i \rangle < size_{\vartheta}(t)\} & p = \mathbf{val}(a, \mathbf{ptr}(t)) \\ \emptyset & otherwise. \end{cases}$$

◀ **Definition 5.53** Support of a pointer

A read from pointer $p \in val$ in thread $k \in Tid$ is safe if it is a read from an owned, a shared, or a guest address:

$$safe\text{-}read^{\vartheta}(p \in val, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$safe\text{-}read^{\vartheta}(p, o, k) \stackrel{\text{def}}{=} (p = \mathbf{val}(a, \mathbf{ptr}(t)) \implies support^{\vartheta}(p) \subseteq o[k] \cup SharedAddr \cup GuestAddr.$$

◀ **Definition 5.54** Safe read

[3] In order to maintain ownership invariants on the C-IL level, one can for instance explicitly maintain ownership sets inside the ghost state of C-IL + ghost semantics [CMST09].

The following function determines whether a given expression $e \in \mathbb{E}$ from thread $k \in Tid$ is safe:

Definition 5.55 ► Safe expression

$$safe\text{-}expr^{\pi,\theta}(e \in \mathbb{E}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$safe\text{-}expr^{\pi,\theta}(e, c, o, k) \stackrel{\text{def}}{=} \begin{cases} safe\text{-}read^{\theta}([\&v]_c^{\pi,\theta}, o, k) & e = v \\ safe\text{-}read^{\theta}([e']_c^{\pi,\theta}, o, k) & e = *e' \\ 1 & \text{otherwise.} \end{cases}$$

5.3.3 Safe Statements

A given C-IL statement which does not involve a memory write and is not a VMRUN is safe if all its expressions are safe. A statement which involves a write to the global memory is safe if all its expressions are safe and the memory write is also safe.

We use the function $sub\text{-}expr^{\pi,\theta}(s, c)$ to extract the set of all sub-expressions of a statement $s \in \mathbb{S}$:

$$sub\text{-}expr^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}) \in 2^{\mathbb{E}}.$$

Definition of the function *sub-expr* is straightforward and follows from the rules for expression evaluation. We omit it here.

The following predicate denotes that all expressions extracted from a given statement are safe:

Definition 5.56 ► Safe expression

$$safe\text{-}exprs^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B},$$

$$safe\text{-}exprs^{\pi,\theta}(s, c, o, k) \stackrel{\text{def}}{=}$$
$$\forall e \in sub\text{-}expr^{\pi,\theta}(s, c) : safe\text{-}expr^{\pi,\theta}(e, c, o, k).$$

Memory writes in the C-IL semantics are performed as a result of an assignment operation, a compare-exchange operation, or a return from the function. Only one memory write per statement is allowed (only the left value can be updated). Moreover, C-IL supports only writes to primitive variables/fields. Hence, at most 64 bits are updated in a single C-IL statement.

To state safety on memory writes performed as a part of statement execution, we have to identify writes which will be compiled to locked operations (either compare exchange or a locked write) from regular writes. We do this distinction based on the type qualifier of the global variable being written. We assume that every memory write to a **volatile** global memory is compiled to a locked write and every execution of a compare-exchange statement (which is an abstraction of the respective compiler intrinsic/external assembly function) is compiled into an atomic compare exchange operation.

The following predicate denotes that a given statement $s \in \mathbb{S}$ involves a

write to a shared global variable:

Definition 5.57 Write to a shared memory

$$shared\text{-}write^{\pi,\theta}(c \in conf_{C-IL}, s \in \mathbb{S}) \in \mathbb{B},$$
$$shared\text{-}write^{\pi,\theta}(c, s) \stackrel{\text{def}}{=} (((s = (e_0 = e1)) \vee s = (e_0 = \textbf{call } e(E)))$$
$$\wedge\ [\&e_0]_c^{\pi,\theta} \neq \textbf{lref}((v, o), t) \wedge \textbf{volatile} \in \textbf{fst}(\tau_E{}^{\pi,\theta}(e_0)))$$
$$\vee\ (s = \textbf{cmpxchng}(rds, dest, cmp, exchng) \wedge \textbf{volatile} \in \textbf{fst}(\tau_E{}^{\pi,\theta}(dest))).$$

Analogously, we identify statements which involve a non-volatile global write:

Definition 5.58 Non-volatile write

$$normal\text{-}write^{\pi,\theta}(c \in conf_{C-IL}, s \in \mathbb{S}) \in \mathbb{B},$$
$$normal\text{-}write^{\pi,\theta}(c, s) \stackrel{\text{def}}{=} (((s = (e_0 = e1)) \vee s = (e_0 = \textbf{call } e(E)))$$
$$\wedge\ [\&e_0]_c^{\pi,\theta} \neq \textbf{lref}((v, o), t) \wedge \textbf{volatile} \notin \textbf{fst}(\tau_E{}^{\pi,\theta}(e_0)))$$
$$\vee\ (s = \textbf{cmpxchng}(rds, dest, cmp, exchng) \wedge \textbf{volatile} \notin \textbf{fst}(\tau_E{}^{\pi,\theta}(dest))).$$

A non-volatile global memory write to (64 bit) pointer $p \in val$ in thread $t \in Tid$ is safe iff the support of p is included into the ownership set of t:

Definition 5.59 Safe write

$$safe\text{-}write^{\pi,\theta}(p \in val, o \in Ownership, k \in Tid) \mapsto \mathbb{B},$$
$$safe\text{-}write^{\pi,\theta}(p, o, k) \stackrel{\text{def}}{=} (p = \textbf{val}(a, \textbf{ptr}(t)) \implies support^{\theta}(p) \subseteq o[k]).$$

A volatile (i.e., interlocked) write to a global variable is safe if it is performed either to a shared address or to a guest address, or to an address from the ownership set of t:

Definition 5.60 Safe locked write

$$safe\text{-}locked\text{-}write^{\pi,\theta}(p \in val, o \in Ownership, k \in Tid) \mapsto \mathbb{B},$$
$$safe\text{-}locked\text{-}write^{\pi,\theta}(p, o, k) \stackrel{\text{def}}{=} (p = \textbf{val}(a, \textbf{ptr}(t)) \implies$$
$$support^{\theta}(p) \subseteq SharedAddr \cup GuestAddr \cup o[k]).$$

In case of an assignment we have to distinguish between writes to a volatile variable from the writes to a non-volatile one:

Definition 5.61 Safe assignment

$$safe\text{-}assignment^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B},$$
$$safe\text{-}assignment^{\pi,\theta}(s, c, o, k) \stackrel{\text{def}}{=} s = (e_0 = e_1) \implies$$
$$safe\text{-}exprs^{\pi,\theta}(s, c, o, k)$$
$$\wedge\ (shared\text{-}write^{\pi,\theta}(c, s) \implies safe\text{-}locked\text{-}write^{\pi,\theta}([\&e_0]_c^{\pi,\theta}, o, k))$$
$$\wedge\ (normal\text{-}write^{\pi,\theta}(c, s) \implies safe\text{-}write^{\pi,\theta}([\&e_0]_c^{\pi,\theta}, o, k)).$$

Similar to the safety of an assignment we introduce the safety of a function

call:

Definition 5.62 ► Safe function call

$$safe\text{-}fcall^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$\begin{aligned} safe\text{-}fcall^{\pi,\theta}(s,c,o,k) &\stackrel{\text{def}}{=} s = (e_0 = \textbf{call } e(E)) \implies \\ & safe\text{-}exprs^{\pi,\theta}(s,c,o,k) \\ & \wedge (shared\text{-}write^{\pi,\theta}(c,s) \implies safe\text{-}locked\text{-}write([\&e_0]_c^{\pi,\theta}, o, k)) \\ & \wedge (normal\text{-}write^{\pi,\theta}(c,s) \implies safe\text{-}write([\&e_0]_c^{\pi,\theta}, o, k)). \end{aligned}$$

Another C-IL statement which might involve a write to the shared memory is an atomic compare exchange. We assume that all compare-exchange statements are compiled into respective hardware atomic compare-exchange operations. Hence, we don't distinguish between writes to volatile and non-volatile data in this case:

Definition 5.63 ► Safe compare-exchange

$$safe\text{-}cmpxchng^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$\begin{aligned} safe\text{-}cmpxchng^{\pi,\theta}(s,c,o,k) &\stackrel{\text{def}}{=} s = \textbf{cmpxchng}(rds, dest, cmp, exchng) \implies \\ & safe\text{-}exprs^{\pi,\theta}(s,c,o,k) \wedge safe\text{-}locked\text{-}write([dest]_c^{\pi,\theta}, o, k)). \end{aligned}$$

Additionally to ownership safety, we have to take care of the safety of C-IL steps which involve writing to the $CR3$ register (see Invariant 4.40). Since we do not support a move to $CR3$ instruction in our C-IL semantics, the only statement we have to deal with is a VMRUN:

Definition 5.64 ► Safe VMRUN

$$safe\text{-}vmrun^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$\begin{aligned} safe\text{-}vmrun^{\pi,\theta}(s,c,o,k) &\stackrel{\text{def}}{=} (s = \textbf{vmrun } e([asid, cr3, inject])) \implies \\ & safe\text{-}exprs^{\pi,\theta}(s,c,o,k) \wedge [asid]_c^{\pi,\theta} = \textbf{val}(a, u64) \wedge a \neq 0^{64} \\ & \wedge [cr3]_c^{\pi,\theta} = \textbf{val}(b, u64) \wedge \\ & \wedge root\text{-}pt\text{-}memtype(uint2cr3(\langle b \rangle)) = WB. \end{aligned}$$

The following predicate determines whether expression $s \in \mathbb{S}$ is safe in thread $k \in Tid$:

Definition 5.65 ► Safe statement

$$safe\text{-}stmt^{\pi,\theta}(s \in \mathbb{S}, c \in conf_{C-IL}, o \in Ownership, k \in Tid) \mapsto \mathbb{B}$$

$$safe\text{-}stmt^{\pi,\theta}(s,c,o,k) \stackrel{\text{def}}{=} \begin{cases} safe\text{-}assignment^{\pi,\theta}(s,c,o,k) & s = (e_0 = e1) \\ safe\text{-}fcall^{\pi,\theta}(s,c,o,k) & s = (e_0 = \textbf{call } e(E)) \\ safe\text{-}cmpxchng^{\pi,\theta}(s,c,o,k) & s = \textbf{cmpxchng}(E) \\ safe\text{-}vmrun^{\pi,\theta}(s,c,o,k) & s = \textbf{vmrun}(E) \\ safe\text{-}exprs^{\pi,\theta}(s,c,o,k) & \text{otherwise.} \end{cases}$$

Ownership transfer on the C-IL level has to comply with the same restrictions as on the reduced hardware level (Invariant 4.37). Release of the ownership has to be performed during a locked write or a compare-exchange. When a thread acquires the ownership of an address, then this address can

not be present in the ownership domain of any other thread.

Definition 5.66 Safe ownership transfer

$$\begin{aligned}&\mathit{safe\text{-}transfer}^{\pi,\partial}(c \in \mathit{conf}_{C-IL}, c' \in \mathit{conf}_{C-IL}, k \in \mathit{Tid},\\ &\qquad o \in \mathit{Ownership}, o' \in \mathit{Ownership}) \in \mathbb{B}\\ &\mathit{safe\text{-}transfer}^{\pi,\partial}(c, c', k, o, o') \stackrel{\text{def}}{=}\\ &\quad bpa \in o[i] \wedge bpa \notin o'[i] \implies i = k \wedge \mathit{shared\text{-}write}^{\pi,\partial}(c, \mathit{stmt}_{next}(c, \pi))\\ &\quad bpa \notin o[i] \wedge bpa \in o'[i] \implies bpa \in \mathit{PrivateAddr} \wedge bpa \notin \bigcup_{i \neq j} o'[j]\end{aligned}$$

5.3.4 Safe Execution

We say that a given step of a sequential C-IL configuration is *safe*, if the statement being executed and the ownership transfer being performed are safe:

Definition 5.67 Safe C-IL step

$$\begin{aligned}&\mathit{safe\text{-}step}^{\pi,\partial}_{C-IL}(c \in \mathit{conf}_{C-IL}, c' \in \mathit{conf}_{C-IL}, k \in \mathit{Tid},\\ &\qquad o \in \mathit{Ownership}, o' \in \mathit{Ownership}) \in \mathbb{B}\\ &\mathit{safe\text{-}step}^{\pi,\partial}_{C-IL}(c, c', k, o, o') \stackrel{\text{def}}{=} \mathit{safe\text{-}stmt}^{\pi,\partial}(\mathit{stmt}_{next}(c, \pi), c, o, k)\\ &\qquad \wedge\ \mathit{safe\text{-}transfer}(c, c', k, o, o').\end{aligned}$$

A local sequence of C-IL steps from configuration c to c' is safe if every step in this sequence is safe. The following predicate denotes that sequence $c \to^* c'$ is safe starting with the ownership setting o and ending with the ownership setting o':

Definition 5.68 Safe C-IL execution of a thread k

$$\begin{aligned}&\mathit{safe\text{-}local\text{-}seq}^{\pi,\partial}_{C-IL}(c \in \mathit{conf}_{C-IL}, c' \in \mathit{conf}_{C-IL}, k \in \mathit{Tid},\\ &\qquad o \in \mathit{Ownership}, o' \in \mathit{Ownership}) \in \mathbb{B}\\ &\mathit{safe\text{-}local\text{-}seq}^{\pi,\partial}_{C-IL}(c, c', k, o, o') \stackrel{\text{def}}{=} (c = c' \wedge o = o')\\ &\quad \vee (\forall c'' : \exists o'' : c \to c'' \implies \mathit{safe\text{-}step}^{\pi,\partial}_{C-IL}(c, c'', k, o, o'')\\ &\qquad \wedge\ \mathit{safe\text{-}local\text{-}seq}^{\pi,\partial}_{C-IL}(c'', c', k, o'', o'))\end{aligned}$$

Note, that *safe-local-seq* is well defined only if there exists an execution sequence from c to c' or $c = c'$.

5.4 Compiler Correctness

Compiler correctness is often stated in the form of a simulation relation between the code being compiled and the hardware instruction sequence, obtained as a result of compilation [LPP05, Lei08, BDL06, Ler09]. We call this relation *compiler consistency*. Normally, this simulation relation consists of a number of properties fixing memory layout of the compiled program, values of registers taking part in the program execution, and the stack layout. For non-optimizing compilers compiler consistency has to hold for every step of the compiled program (C-IL step in our case). With the presence of the compiler optimizations and code reordering consistency is relaxed to hold only at certain points in program execution, which we call *consistency points*.

In order to define compiler consistency and to state that it holds at consistency points, we first have to reorder the steps of a hardware execution. If we consider a regular (non-reordered) sequence of hardware steps, then at a given consistency point of some thread only consistency for this thread (and possibly for the shared memory) will be guaranteed to hold. Hence, to be able to state the compiler consistency for all threads at every consistency point, we introduce reordering of hardware steps into a so-called *consistency-block* schedule [Bau12].

In order for the reordering theorem to hold, one has to enforce certain requirements on the set of consistency points. More precisely, we have to make sure that between any two consistency points of a given thread there is no more than one access to a shared resource. A step performing such access we call an *I/O step*. In order to make sure that this restriction holds, we first define the set of *I/O points*, which denote hardware states directly before I/O steps. Further, we define the set of consistency points in such a way, that every I/O point is also a consistency points.

5.4.1 Hardware I/O Points

I/O points [DPS09] in the hardware execution sequence identify hardware states directly before and/or after an action of a given processor, which is visible for external environment (including other processors). For instance, access to a shared memory is such an action. Execution sequence of a given processor in between two I/O points is called *local*. We use the notion of I/O points to define an I/O-block schedule of a hardware execution sequence, where interleaving of steps of different processors can occur only at I/O points (Section 5.4.3).

In case of a hypervisor program running in parallel with guest threads, I/O points come in two flavors: I/O points of the hypervisor and I/O points of guests.

A *hypervisor I/O step* is a hardware step which involves an access to a global shared variable of a processor running in hypervisor mode. These steps are an atomic compare-exchange, a locked write, and a read from a shared/guest memory (guest memory is also considered to be a shared resource).

A *hypervisor I/O point* is a hardware configuration directly before a hypervisor I/O step or a hardware configuration before the first step of every processor (initial thread-local configuration after boot-loading and hypervisor initialization are complete). Note, that the end of the execution is not considered to be an I/O point because we assume (optimistically) that the hypervisor never terminates.

For a hardware execution fragment $h^0 \xrightarrow{\beta} h^n$, where $|\beta| = n$ and $n > 0$, we introduce a predicate, which denotes that the hardware configuration h^i is a hypervisor I/O point of processor k:

Definition 5.69 ► Hypervisor I/O point of processor k

$$\begin{aligned} \mathit{hyp\text{-}iopoint}_k(\beta, i) \stackrel{\text{def}}{=}\ & h^i.p[k].\mathit{asid} = 0 \\ & \wedge (\mathit{pid}(\beta_i) = k \wedge (\forall j < i : \mathit{pid}(\beta_j) \neq k) \\ & \quad \vee \mathit{affected\text{-}byte\text{-}addr}(\mathit{va}, \mathit{mask}) \subseteq \mathit{SharedAddr} \cup \mathit{GuestAddr} \\ & \quad\quad \wedge \beta_i \in \{\mathit{core\text{-}atomic\text{-}cmpxchng}(k, w), \mathit{core\text{-}locked\text{-}write}(k, w) \\ & \quad\quad\quad\quad \mathit{core\text{-}memory\text{-}read}(k, w)\})), \end{aligned}$$

where $va = h^i.p[k].memreq.va$ and $mask = h^i.p[k].memreq.mask$. Note, that in the definition given above and in the upcoming definitions of functions which take as a parameter a sequence of hardware actions, we implicitly pass as another parameter a sequence of hardware configurations $h^0, h^1, \ldots, h^n$, produced by the sequence of actions. Later we use such functions only in the context where this sequence of configurations is well defined.

For processors running in guest mode we consider all guest steps which involve an access to the main memory (including MMU reading/writing shared PTEs and SB committing stores) to be *guest I/O steps* and hardware configurations before such steps to be *guest I/O points*:

Definition 5.70 Guest I/O point of processor k

$$\begin{aligned} & guest\text{-}iopoint_k(\beta, i) \stackrel{\text{def}}{=} h^i.p[k].asid \neq 0 \\ & \quad \wedge (\beta_i \in \{extend\text{-}walk(k, w, r), set\text{-}access\text{-}dirty(k, w)\} \\ & \qquad \wedge\ qword2bytes(pte\text{-}addr(w.pfn, w.vpfn.px[w.l])) \subseteq SharedAddr \\ & \quad\ \vee \beta_i \in \{core\text{-}atomic\text{-}cmpxchng(k, w), core\text{-}locked\text{-}memory\text{-}write(k, w)\} \\ & \quad\ \vee \beta_i = core\text{-}memory\text{-}read(k, w) \\ & \qquad \wedge \neg pending\text{-}qword\text{-}store(sb^i[k], w.pfn \circ memreq^i[k].va.off) \\ & \quad\ \vee \beta_i = core\text{-}report\text{-}page\text{-}fault(k, w) \\ & \quad\ \vee \beta_i = commit\text{-}store(k)) \end{aligned}$$

Note, that the initial hardware configuration cannot be a guest I/O point, because we consider only those execution sequences which start when all processors are in hypervisor mode.

If we need to denote that configuration h^i is an I/O point regardless of its flavour or regardless of the processor which has performed an I/O step we use the following functions:

Definition 5.71 I/O point

$$\begin{aligned} iopoint_k(\beta, i) &\stackrel{\text{def}}{=} hyp\text{-}iopoint_k(\beta, i) \vee guest\text{-}iopoint_k(\beta, i) \\ iopoint(\beta, i) &\stackrel{\text{def}}{=} \exists k \in Pid : iopoint_k(\beta, i). \end{aligned}$$

5.4.2 Consistency Points

Another set of dedicated hardware states which we define, is the set of *hypervisor consistency points*. An optimizing compiler has to guarantee that the compiler consistency relation holds at every consistency point under an assumption that the program is executed alone on the hardware machine (i.e., there are no guest steps). Every hypervisor I/O point is also a consistency point.

The set of hypervisor consistency points consists of the following hardware configurations:

- any hardware configuration which is a hypervisor I/O point[4],
- hardware configuration before execution of a VMRUN instruction,

[4] Note, that the state before execution of an atomic compare-exchange is an I/O point only if the memory write is done to a shared memory region. Yet, one could also consider states before non-shared compare-exchanges to be consistency points. In order to do so, one has to add these states to sets of hardware and software consistency points.

- hardware configuration before the first step of a processor in hypervisor mode after a VMEXIT event (return from guest mode),
- hardware configuration before execution of an INVLPG statement.

For a hardware execution fragment $h^0 \xrightarrow{\beta} h^n$, where $|\beta| = n$ and $n > 0$, we introduce a predicate, which denotes that a hardware configuration h^i is a hypervisor consistency point of a processor k:

Definition 5.72 ► Hypervisor consistency point of processor k

$$\begin{aligned} \mathit{hyp\text{-}cpoint}_k(\beta, i) \stackrel{\text{def}}{=}\ & i < |\beta| \wedge \\ & (\mathit{hyp\text{-}iopoint}_k(\beta, i) \\ & \vee \beta_i \in \{\mathit{core\text{-}vmrun}(k) \vee \mathit{core\text{-}tlb\text{-}invlpga}(k)\} \\ & \vee \mathit{pid}(\beta_i) = k \wedge \exists j < i : \beta_j = \mathit{core\text{-}vmexit}(k, w) \\ & \qquad \wedge \forall m \in (j : i) : \mathit{pid}(\beta_m) \neq k). \end{aligned}$$

Note, that the set of consistency points could include more hardware states. For instance, one could also include a hardware state before execution of the first statement in every function into the set of consistency points. However, identifying these states in the execution of our hardware model is tedious, because we do not model instruction execution in detail. Since extension of the set of consistency points will not further affect any proofs presented in this thesis, we stick to the limited consistency set defined above.

Additionally to the set of hypervisor consistency points we introduce *guest consistency points*, which are used as auxiliary points in the compiler correctness proof. Note, that this set is defined solely from the hypervisor point of view and is (likely to be) different from the set of consistency points of a guest program running in a partition.

The set of guest consistency points has to include at least all guest I/O points. Yet, we define this set to include all states before execution of a guest step:

Definition 5.73 ► Guest consistency point of processor k

$$\mathit{guest\text{-}cpoint}_k(\beta, i) \stackrel{\text{def}}{=} i < |\beta| \wedge \mathit{pid}(\beta_i) = k \wedge h^i.p[k].\mathit{asid} \neq 0.$$

If we need to denote that configuration h^i is a consistency point regardless of its flavour or regardless of a processor ID we use the following functions:

Definition 5.74 ► Hardware consistency point

$$\begin{aligned} \mathit{cpoint}_k(\beta, i) &\stackrel{\text{def}}{=} \mathit{hyp\text{-}cpoint}_k(\beta, i) \vee \mathit{guest\text{-}cpoint}_k(\beta, i), \\ \mathit{cpoint}(\beta, i) &\stackrel{\text{def}}{=} \exists k \in \mathit{Pid} : \mathit{cpoint}_k(\beta, i). \end{aligned}$$

The following function returns the index of a hardware configuration at the next consistency point encountered in the execution sequence starting from configuration h^i (not including h^i itself). In case if no such point exists, the function returns $\perp$:

Definition 5.75 ► Next consistency point

$$\mathit{next\text{-}cpoint}(\beta, i) \stackrel{\text{def}}{=} \begin{cases} 0 & i = 0 \\ j & \mathit{cpoint}(\beta, j) \wedge \forall k \in [i+1, j-1] : \neg \mathit{cpoint}(\beta, k) \\ \perp & \forall k > i : \neg \mathit{cpoint}(\beta, k). \end{cases}$$

Given a point i in the hardware execution sequence $h^0 \xrightarrow{\beta} h^n$, where $|\beta| = n$, $i < n$, and $n > 0$, we want to identify, whether thread k will perform any steps in between h^i and h^n. If this is the case, then we call thread k a *running thread* in configuration h^i. In the compiler correctness theorem in Section 5.4.6 we require the consistency relation to hold at consistency points only for running threads. A thread which is not running, could have been interrupted before it has advanced to its own next consistency point. In this case, the compiler cannot guarantee local consistency for this thread. The following predicate denotes that a thread k is running in configuration h^i:

$$running\text{-}thread_k(\beta, i) \overset{\text{def}}{=} \exists j \in [i : n-1] : pid(\beta_j) = k.$$

◄ **Definition 5.76**
Running thread

5.4.3 Consistency-block Schedule

A consistency-block schedule is a hardware execution sequence, where steps of different processors can be interleaved only as *consistency blocks*. Every consistency block starts with a consistency point, which is followed by a number of local steps of the same thread.

The following predicate denotes that an execution sequence $h^0 \xrightarrow{\beta} h^n$, where $|\beta| = n$ and $n > 0$ is a consistency block schedule:

$$cosched(\beta) \overset{\text{def}}{=} \begin{cases} 1 & n = 1 \\ cosched(\beta[0 : n-2]) & cpoint(\beta, n-1) \\ cosched(\beta[0 : n-2]) & \\ \wedge pid(\beta_{n-1}) = pid(\beta_{n-2}) & \text{otherwise.} \end{cases}$$

◄ **Definition 5.77**
Consistency block schedule

Before we can state the reordering theorem, we first have to identify that two given hardware executions starting and ending in the same state are equal from the processor-local point of view, i.e., for a given processor execution traces of each of these sequences are the same. To state this equivalence formally, we introduce a function which extracts the sequence of local actions of a processor k from execution sequence β, where $|\beta| = n$ and $n > 0$:

$$local\text{-}seq(\beta, k) \overset{\text{def}}{=} \begin{cases} local\text{-}seq(\beta[0 : n-2], k) \circ \beta_{n-1} & pid(\beta_{n-1}) = k \\ local\text{-}seq(\beta[0 : n-2], k) & \text{otherwise.} \end{cases}$$

◄ **Definition 5.78**
Local sequence

We call hardware execution sequences $h^0 \xrightarrow{\beta} h^n$ and $h^0 \xrightarrow{\omega} h^n$ where $|\beta| = n$ and $n > 0$ *equivalent*, iff local sequences of actions of all processors are equal:

$$(\beta \equiv \omega) \overset{\text{def}}{=} |\beta| = |\omega| \wedge \forall i \in Pid : local\text{-}seq(\beta, i) = local\text{-}seq(\omega, i)$$

◄ **Definition 5.79**
Equivalent sequences

Now we can state the consistency-block reordering theorem.

Theorem 5.1 (Consistency-block reordering). *Let $h \xrightarrow{\beta} h'$ be an execution sequence of hardware machine $h \in RedHardw$, which starts in a safe state.*

Moreover, let all consistency block schedules which lead from h to h′ be safe. Then sequence $h \stackrel{\beta}{\to} h'$ *is also safe and there exists a consistency block schedule* $h \stackrel{\omega}{\to} h'$*, such that sequence of actions ω is equivalent to sequence of actions β:*

$$\begin{aligned}
&\forall \beta, (h \stackrel{\beta}{\to} h') : \\
&\quad \mathit{safe\text{-}conf}_r(h, o) \\
&\quad \wedge (\forall \gamma, (h \stackrel{\gamma}{\to} h') : \exists o' : \mathit{cosched}(\gamma) \implies \mathit{safe\text{-}seq}_r(\gamma, o, o')) \\
&\quad \implies \exists o'' : \mathit{safe\text{-}seq}_r(\beta, o, o'') \\
&\qquad\quad \wedge \exists \omega, (h \stackrel{\omega}{\to} h') : \mathit{cosched}(\omega) \wedge \omega \equiv \beta.
\end{aligned}$$

Proof. The proof of this great theorem for a general case of distributed communicating I/O automata was done by Christoph Baumann in [Bau12]. □

5.4.4 Consistency Relation

A compiler consistency relation normally consists of two parts: (i) control consistency and (ii) data consistency [LP08b].

Control consistency bounds values of program counters with memory addresses, where the executed code is located. Additionally, it fixes values of return addresses of all stack frames to point to the next instruction in the code after the function call corresponding to a given stack frame [Sha12].

Data consistency argues about the memory content of the hardware machine and consists of the following components:

- code consistency, which ensures that the program code is located at the dedicated memory region, disjoint from the memory where the program data is located (this assumes no self-modifying code),
- stack consistency, which argues about the memory region allocated for the program stack, which has to be disjoint from global data and code memory regions; this also includes register consistency, which fixes the values of registers used during execution of the code (e.g., stack pointer and base pointer) excluding the program counters, which are fixed by the control consistency relation,
- memory consistency, which talks about the global program variables, i.e., the content of the shared memory component of the C-IL memory.

We further divide all consistency properties into two groups:

- *global consistency*, which fixes the content of the hardware memory independently of the local processor state. This includes memory consistency and code consistency;
- *local consistency*, which fixes the local processor state and the content of the hardware memory region where the program stack is located. This includes stack consistency and control consistency.

Global Consistency

Now we can define the set of addresses which forms the global memory of the C-IL hypervisor program:

$$\mathbb{B}_{gm} = \{a \in \mathbb{B}^{64} : a \in PrivateAddr \cup SharedAddr \cup GuestAddr\}.$$

The set $\mathbb{B}_{gm}$ is statically fixed by the compiler and consists of physical addresses where global variables and the program heap are located. This set is disjoint from the physical addresses, where the program stack and the compiled program code reside (we do not support self-modifying code and pointers to local variables in the C-IL semantics). Additionally, we include the set of guest addresses to $\mathbb{B}_{gm}$ in order to allow the hypervisor program to read and write the guest memory.

Since a C-IL configuration has flat byte-addressable memory, we define the memory consistency relation in a straightforward way, linking the content of the C-IL memory with the memory of the reduced hardware machine:

◄ **Definition 5.80**
Memory consistency

$$\begin{aligned}
&gm\text{-}consis(\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, h \in RedHardw) \in \mathbb{B},\\
&gm\text{-}consis(\mathcal{M}, h) \stackrel{\text{def}}{=} \forall a \in \mathbb{B}_{gm} : i = \langle a[0:2] \rangle \implies\\
&\quad \mathcal{M}[a] = byte_i(h.mm[a[52:3]]).
\end{aligned}$$

The code consistency argues about the read-only memory region where the compiled code of the program π is located. We define it as the following (uninterpreted) function:

$$code\text{-}consis(\pi \in prog_{C-IL}, mm \in ReadOnlyAddr \mapsto \mathbb{B}^{64}) \in \mathbb{B}.$$

Putting together memory consistency and code consistency we get the global part of the C-IL consistency relation:

◄ **Definition 5.81**
Global consistency

$$\begin{aligned}
&global\text{-}consis(\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, \pi \in prog_{C-IL}, h \in RedHardw) \in \mathbb{B}\\
&global\text{-}consis(\mathcal{M}, \pi, h) \stackrel{\text{def}}{=} code\text{-}consis(\pi, h.mm[ReadOnlyAddr])\\
&\qquad\qquad \wedge\ gm\text{-}consis(\mathcal{M}, h).
\end{aligned}$$

Local Consistency

The definition of the stack consistency largely depends on the compiler architecture and stack layout. It couples the current state of the stack of the C-IL configuration with the content of the hardware memory where the stack is located and with a certain state of hardware registers (callee/caller save registers, base pointer, and stack pointer). Since we do not model these registers explicitly, we use the uninterpreted state of our instruction automaton (Section 3.6) instead and define local stack consistency of processor i with the following function:

$$\begin{aligned}
&stack\text{-}consis_i(stack \in frame^*_{C-IL}, state \in InstrCoreState,\\
&\qquad mm \in StackAddr_i \mapsto \mathbb{B}^{64}) \in \mathbb{B}.
\end{aligned}$$

When switching to guest mode the hypervisor (i.e., a special assembly function inside the hypervisor) is responsible for saving values of callee/caller save registers, stack and base pointers and other registers participating in stack consistency relation. The same applies in case of a return from the guest execution, where a consistent configuration has to be restored.

When the guest code is executed on a processor, regular stack consistency [Sha12] for a hypervisor program will not hold. Yet, it does hold if one takes saved values of registers instead of the running ones. Hence, one can define hypervisor stack consistency in such a way, that it considers running registers when the hypervisor is executed (and after configuration is restored) and saved registers if a guest is running.

In this thesis we assume $\textit{stack-consis}_i$ to be defined in this way. Moreover, it cannot be broken by any guest step, if this step does not involve writing to the memory region where the stack is located (this also assumes that saving of registers is done to the same memory region):

Definition 5.82 ► Stack consistency stable

$$\begin{aligned}
&\textit{stack-consis-stable}_i(\textit{stack} \in \textit{frame}^*_{C-IL}, h \in \textit{RedHardw}) \in \mathbb{B},\\
&\textit{stack-consis-stable}_i(\textit{stack}, h) \stackrel{\text{def}}{=}\\
&\forall a : h \stackrel{a}{\to} h' \wedge \textit{pid}(a) = i \wedge h.p[i].\textit{asid} \neq 0\\
&\quad \wedge h.\textit{mm}[\textit{StackAddr}_i] = h'.\textit{mm}[\textit{StackAddr}_i]\\
&\quad \wedge \textit{stack-consis}_i(\textit{stack}, h.p[i].\textit{state}, h.\textit{mm}[\textit{StackAddr}_i])\\
&\quad \implies \textit{stack-consis}_i(\textit{stack}, h'.p[i].\textit{state}, h'.\textit{mm}[\textit{StackAddr}_i]).
\end{aligned}$$

A property of the same kind should also hold for control consistency, which also has to argue about saved register values in case the hypervisor is sleeping. We assume here that values of registers are either saved in the uninterpreted part of the core state or in the same memory region which is used for the local stack:

$$\begin{aligned}
\textit{control-consis}_i(&\textit{stack} \in \textit{frame}^*_{C-IL}, \textit{state} \in \textit{InstrCoreState},\\
&\textit{mm} \in \textit{StackAddr}_i \mapsto \mathbb{B}^{64}) \in \mathbb{B}.
\end{aligned}$$

In contrast to the stack consistency, registers fixed by the control consistency are saved automatically by the hardware which supports virtualization extensions. Hence, stability of control consistency under guest steps should always hold:

Definition 5.83 ► Control consistency stable

$$\begin{aligned}
&\textit{control-consis-stable}_i(\textit{stack} \in \textit{frame}^*_{C-IL}, h \in \textit{RedHardw}) \in \mathbb{B}\\
&\textit{control-consis-stable}_i(\textit{stack}, h) \stackrel{\text{def}}{=}\\
&\forall a : h \stackrel{a}{\to} h' \wedge \textit{pid}(a) = i \wedge h.p[i].\textit{asid} \neq 0\\
&\quad \wedge h.\textit{mm}[\textit{StackAddr}_i] = h'.\textit{mm}[\textit{StackAddr}_i]\\
&\quad \wedge \textit{control-consis}_i(\textit{stack}, h.p[i].\textit{state}, h.\textit{mm}[\textit{StackAddr}_i])\\
&\quad \implies \textit{control-consis}_i(\textit{stack}, h'.p[i].\textit{state}, h'.\textit{mm}[\textit{StackAddr}_i]).
\end{aligned}$$

Local consistency is obtained by putting together stack and control

consistencies:

$$local\text{-}consis_i(stack \in frame^*_{C-IL}, state \in InstrCoreState, mm \in StackAddr_i \in \mathbb{B}^{64}) \in \mathbb{B}$$
$$local\text{-}consis_i(stack, state, mm) \stackrel{\text{def}}{=} stack\text{-}consis_i(stack, state, mm) \wedge control\text{-}consis_i(stack, state, mm)$$

◄ **Definition 5.84**
Local consistency

Stability of the stack and control consistencies combined together gives us stability of the local consistency under guest steps. Further in this thesis we assume that the local consistency is defined in such a way, that its stability always holds.

name	*inv-local-consis-stable*()
property	$\forall h \in RedHardw, stack \in frame^*_{C-IL}, i \in Pid : stack\text{-}consis\text{-}stable_i(stack, h) \wedge control\text{-}consis\text{-}stable_i(stack, h)$

◄ **Invariant 5.85**
Local consistency stable

Putting together the global consistency and the local consistency of a given processor we obtain C-IL consistency relation for processor i:

$$consis_{C-IL}(c \in conf_{CC-IL}, \pi \in prog_{C-IL}, h \in RedHardw, i \in Pid) \in \mathbb{B}$$
$$consis_{C-IL}(c, \pi, h, i) \stackrel{\text{def}}{=} global\text{-}consis(c.\mathcal{M}, \pi, h) \wedge local\text{-}consis_i(c.stack[i], h.p[i].state, h.mm[StackAddr_i]).$$

◄ **Definition 5.86**
C-IL consistency

5.4.5 Software Consistency Points

In order to state compiler consistency in an inductive form (so that we can reuse it later for C-IL + HW consistency in Section 7.4), we need to be able to identify not only consistency points in the hardware execution sequence, but also respective consistency points in a C-IL program. The meaning of consistency points in a C-IL execution sequence is exactly the same as the meaning of hardware (hypervisor) consistency points.

The following predicate is used to identify an expression performing a read from a global shared variable.

$$shared\text{-}read^{\pi,\theta}(c \in conf_{C-IL}, e \in \mathbb{S}) \in \mathbb{B}$$
$$shared\text{-}read^{\pi,\theta}(c, e) \stackrel{\text{def}}{=} [e]^{\pi,\theta}_c = \mathbf{val}(a, \mathbf{ptr}(t)) \wedge \mathbf{volatile} \in \mathbf{fst}(\tau_E{}^{\pi,\theta}(e)).$$

◄ **Definition 5.87**
Read from a shared variable

The definition of a statement performing a write to shared data was given in Section 5.3.3 (Definition 5.57). Hence, we can now define a predicate which denotes that execution of a given statement requires an access (either a read or a write) to a shared global variable:

$$shared\text{-}stmt^{\pi,\theta}(c \in conf_{C-IL}, s \in \mathbb{S}) \stackrel{\text{def}}{=} \exists e \in sub\text{-}expr^{\pi,\theta}(s, c) : shared\text{-}read^{\pi,\theta}(c, e) \vee shared\text{-}write^{\pi,\theta}(c, s).$$

◄ **Definition 5.88**
Statement performing a shared memory access

A given state of local C-IL configuration $c \in conf_{C-IL}$ is a consistency point if the location counter points to the first statement to be executed in the program $\pi \in prog_{C-IL}$, if the next statement is a VMRUN or an INVLPG, if the next statement is the first one after a VMRUN, or if the next statement involves an access to the shared memory:

Definition 5.89 ► C-IL consistency point

$$\begin{aligned} cpoint_{C-IL}(c, \pi) \overset{\text{def}}{=} & (|c.stack| = 1 \land c.loc_{top} = 0) \\ & \lor stmt_{next}(c, \pi) = \textbf{vmrun}(E) \\ & \lor stmt_{next}(c, \pi) = \textbf{invlpg}(e) \\ & \lor c.loc_{top} > 0 \land \pi.\mathcal{F}(c.f_{top}).\mathcal{P}[c.loc_{top} - 1] = \textbf{vmrun}(E) \\ & \lor shared\text{-}stmt^{\pi,\theta}(c, stmt_{next}(c, \pi)). \end{aligned}$$

Note, that both states before and after execution of a VMRUN step are considered to be C-IL I/O points. In a respective hardware execution the state before a VMRUN will correspond to the hardware state before execution of a *core-vmrun* state and the state after VMRUN will correspond to the hardware state before the first step of the processor after a *core-vmexit* step.

5.4.6 Compiler Correctness Theorem

Intuitively, correct compiler has to guarantee, that for any consistency-block execution fragment $h^0, a_0, h^1, a_1, \ldots$ of the hardware machine $h \in RedHardw$ there exists an execution sequence $c^0, c^1, \ldots$ of a C-IL machine and a step function[5] $s \in \mathbb{N} \mapsto \mathbb{N}$, s.t. for every consistency point i the consistency relation holds between configurations h^i and $c^{s(i)}$.

Yet, in case of a hypervisor program this is not necessarily true, because hypervisor consistency might get broken after the first write to the guest memory by a processor running in guest mode (executing the guest code). Moreover, even if we exclude the guest memory from the C-IL memory component (which would make it impossible for hypervisor to access the guest memory), consistency still could get broken by guest MMUs setting A/D bits in shadow page tables, which are located in the hypervisor memory.

In this chapter we state compiler correctness in an iterative form, where we fix properties only for steps performed by processors running in the hypervisor mode. In Section 7.4.1 we introduce guest steps (including MMU steps) to the C-IL semantics and prove compiler consistency for all hardware steps (w.r.t to C-IL + HW semantics defined in Section 7.2).

Note, that though we call the following theorem "compiler correctness", it states more than just properties of the compiler. To prove such a theorem, one would also have to show that hypervisor state is saved and restored correctly. This would involve both arguing about hardware virtualization features (of the hardware instruction automaton which we don't define in this thesis) and about assembly code which saves and restores the hypervisor program stack. Additionally, our compiler correctness theorem defines the way how INVLPG and VMRUN abstractions, which we have introduced into the C-IL semantics, are compiled.

Theorem 5.2 (C-IL compiler correctness). *Let $\pi \in prog_{C-IL}$ be a C-IL program with context $\theta \in context_{C-IL}$. Further, let $h^0 \in RedHardw$ be the initial safe state*

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

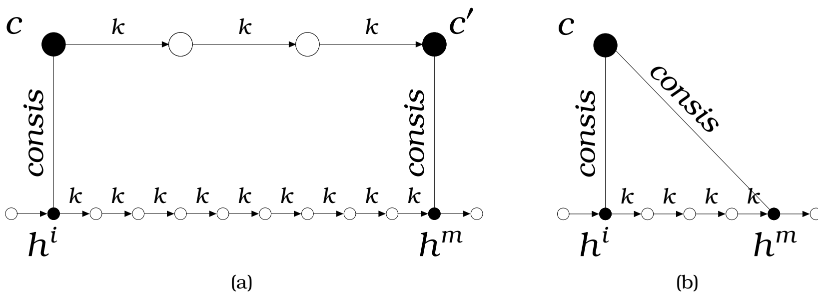

Figure 5.1: Compiler correctness for a C-IL hypervisor (induction step): (a) - case when one consistency point in hardware corresponds to one consistency point in C-IL, (b) - case when two consistency points in hardware correspond to the same consistency point in C-IL.

of the reduced hardware machine, where execution of the program starts and h^n be an arbitrary point in the execution sequence of the compiled program where $n > 0$. Let the consistency relation hold at the beginning of the execution[6]*:*

$$\forall k \in Pid : consis_{C-IL}(c^0, \pi, h^0, k).$$

Then for all block schedules starting from h^0 and ending in h^n the following property holds: if h^i is a safe hypervisor consistency point of processor k, $i < n$, and consistency for all running threads holds between state h^i and state c, where thread k is at the consistency point in c, then there exists configuration c' s.t.

- *either $c' = c$ (this is the case when one consistency point in C corresponds to several consistency points in hardware e.g., when a first statement in a thread or a first statement after VMRUN involves a volatile access) (Figure 5.1, b), or c' is a next consistency point of a thread k and is obtained from c by executing a number of steps of k: $\pi, \theta \vdash c \rightarrow^{+}_{k} c'$ (Figure 5.1, a);*
- *if m is the next hardware consistency point in the execution sequence then consistency for all running threads holds between states c' and h^m and sequence of hardware steps from h^i to h^m is hypervisor-safe if C-IL execution from c to c' is safe (note, that from the definition of a consistency point, there is always at least one running thread in configuration h^m);*
- *if β_i is a VMRUN step, then the next instruction to be executed in c is a VMRUN with the same inputs as the hardware VMRUN step has and C-IL execution from c to c' consists of exactly one step; otherwise (if β_i is not a VMRUN step and $c \neq c'$), the next instruction to be executed in c is not a VMRUN;*
- *if β_i is an INVLPG step, then the next instruction to be executed in c is an INVLPG with the same inputs as the hardware INLVPG step has; otherwise*

[6]The fact that consistency relation holds at the beginning of the hypervisor execution should be guaranteed by the bootloader. Bootloading on an x64 machine can not be performed in the long addressing mode and is left out of the scope of the thesis. We start our argumentation from the configuration h^0, which is a configuration after the bootloader finishes initialization and we assume that compiler consistency is already established at this point.

(if β_i is not an INLVPG step and $c \neq c'$), the next instruction to be executed in c is not an INVLPG;

- *if β_i is a VMRUN step and the flag $c.flush_{TLB}$ is set, then the complete-flush bit in the memreq buffer is set in the configuration h^i.*

Formally we state this as follows:

$$
\begin{aligned}
&\forall(h^0 \xrightarrow{\beta} h^n) : cosched(\beta)\\
&\quad\implies \forall i < n : \forall c, o : hyp\text{-}cpoint_k(\beta, i) \wedge cpoint_{C-IL}(c(k), \pi)\\
&\qquad \wedge\, safe\text{-}hyp\text{-}conf_r(h^i, o) \wedge m = next\text{-}cpoint(\beta, i)\\
&\qquad \wedge(\forall k' \in Pid : running\text{-}thread_{k'}(\beta, i) \implies\\
&\qquad\qquad consis_{C-IL}(c, \pi, h^i, k'))\\
&\qquad\quad \implies \exists c' : \pi, \theta \vdash c \to_k^* c'\\
&\qquad\qquad \wedge\, cpoint_{C-IL}(c'(k), \pi)\\
&\qquad\qquad \wedge\, (\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies\\
&\qquad\qquad\qquad consis_{C-IL}(c', \pi, h^m, k'))\\
&\qquad\qquad \wedge\, (\forall o' : safe\text{-}local\text{-}seq^{\pi,\theta}_{C-IL}(c(k), c'(k), k, o, o') \implies\\
&\qquad\qquad\qquad safe\text{-}hyp\text{-}seq_r(\beta[i : m-1], o, o'))\\
&\qquad\qquad \wedge\, (\beta_i = core\text{-}vmrun(k) \implies \pi, \theta \vdash c \to_k c'\\
&\qquad\qquad\qquad \wedge\, stmt_{next}(c(k), \pi) = \boldsymbol{vmrun}(e_0, e_1, e_2)\\
&\qquad\qquad\qquad \wedge\, [e_0]^{\pi,\theta}_c = \boldsymbol{val}(bin_{64}(h^i.memreq[k].asidin), u64)\\
&\qquad\qquad\qquad \wedge\, [e_1]^{\pi,\theta}_c = \boldsymbol{val}(bin_{64}(h^i.memreq[k].cr3in, u64)\\
&\qquad\qquad\qquad \wedge\, inject\text{-}data^{\pi,\theta}(c(k), e_2) = h^i.memreq[k].inject\text{-}data\\
&\qquad\qquad\qquad \wedge\, c.flush_{TLB} \implies h^i.memreq[k].complete\text{-}flush)\\
&\qquad\qquad \wedge\, (\beta_i \neq core\text{-}vmrun(k) \wedge c \neq c' \implies\\
&\qquad\qquad\qquad stmt_{next}(c(k), \pi) \neq \boldsymbol{vmrun}(E))\\
&\qquad\qquad \wedge\, (\beta_i = core\text{-}tlb\text{-}invlpga(k) \implies c \neq c'\\
&\qquad\qquad\qquad \wedge\, stmt_{next}(c(k), \pi) = \boldsymbol{invlpg}(e_0, e_1)\\
&\qquad\qquad\qquad \wedge\, [e_0]^{\pi,\theta}_c = \boldsymbol{val}(bin_{64}(h^i.memreq[k].va), u64))\\
&\qquad\qquad\qquad \wedge\, [e_1]^{\pi,\theta}_c = \boldsymbol{val}(bin_{64}(h^i.memreq[k].asid), u64))\\
&\qquad\qquad \wedge\, (\beta_i \neq core\text{-}tlb\text{-}invlpga(k) \wedge c \neq c' \implies\\
&\qquad\qquad\qquad stmt_{next}(c(k), \pi) \neq \boldsymbol{invlpg}(E)).
\end{aligned}
$$

Proof. Proof of this theorem does not fall into the scope of this thesis. A proof of a (simpler) compiler correctness theorem for a non-optimizing compiler of a Pascal-like language with C syntax was shown in [LP08b]. A proof of a correctness theorem for an optimizing compiler (with limited optimizations) of the C language was done in the CompCert verification project [Ler09]. □

To prove Theorem 5.2 one has to make sure that when the hardware machine advances from one consistency point to another, the C-IL machine must also advance from one consistent configuration to another consistent configuration (or stay unchanged, while remaining consistent). This implies the following restrictions on a C-IL program:

1. only one volatile access (i.e., an access to the shared portion of the memory) per compiled C-IL statement is allowed,
2. no volatile accesses are allowed in a VMRUN or INVLPG statements.

If a given C program does not satisfy these restrictions, then one can enforce them during the translation of the program from C to C-IL language done by a pre-processor of the compiler.

The situation when the hardware machine advances from one consistency point to another, but the C-IL machine stays unchanged (while remaining consistent with the hardware configuration) happens when the first statement in a thread or a first statement after VMRUN involves a volatile access or is an INVLPG. In this case, both the hardware state before the first step of the thread and the hardware state before a volatile access or an INVLPG step are hardware consistency points. And both these states must be consistent with a single C-IL configuration.

Note also, that the compiler is responsible for correct partitioning of the hardware memory into sets of addresses (i.e., *SharedAddr*, *ReadOnlyAddr*, *IMPTAddr*, etc.) introduced in Section 4.3.1 and in Section 4.5.1. Further, it has to ensure that the program code, local stacks, and global memory of the program are located in the designated memory regions and allocation addresses of all local variables of thread i are present in the ownership set $StackAddr_i$. Only under these conditions one can prove Theorem 5.2, particularly the part which ensures safe hardware execution sequence for the compiled code.

CHAPTER 6

C-IL + Ghost Semantics

6.1 Ghost Types and Values

6.2 Ghost Memory

6.3 Ghost Code

6.4 Configuration and Program

6.5 Memory and Operational Semantics

6.6 Simulation Theorem

Program verification often involves maintaining additional information about the program state. This information might include an abstract program specification or auxiliary data, necessary to prove that implementation behaves accordingly to its specification. To store and maintain this information we use an extension of the C-IL semantics with the *ghost state* as documented by Sabine Schmaltz in [Sch12a]. Ghost state consists of local and global ghost variables and ghost fields of implementation structures.

Ghost code comprises ghost statements and ghost parameters of functions. In order for a program extended with ghost state to simulate the original program, it has to fulfill certain properties. For instance, the ghost code should always terminate (it should not influence the control flow of the program) and there should be no information flow from ghost variables to implementation ones. Under these restrictions one can show simulation between execution of a regular C-IL machine and a C-IL + Ghost machine.

6.1 Ghost Types and Values

6.1.1 Ghost Types

Ghost variables can either be of a qualified non-ghost C-IL type $\mathbb{T}_\mathbb{Q}$ or of a special ghost type $\mathbb{T}_{G\mathbb{Q}}$. The set $\mathbb{T}_{G\mathbb{Q}}$ contains the following ghost types:

Definition 6.1 ► Ghost types

- mathematical (unbounded) integers:

$$q \subseteq \mathbb{Q} \implies (q, \textbf{math_int}) \in \mathbb{T}_{G\mathbb{Q}},$$

- mathematical maps (i.e., functions):

$$q \subseteq \mathbb{Q} \wedge t, t' \in \mathbb{T}_G \cup \mathbb{T} \implies (q, \textbf{map}(t, t')) \in \mathbb{T}_{G\mathbb{Q}},$$

- mathematical records:

$$q \subseteq \mathbb{Q} \wedge t_C \in \mathbb{T}_C \Rightarrow (q, \textbf{record}\ t_C) \in \mathbb{T}_{G\mathbb{Q}},$$

- state-snapshots of the C-IL machine:

$$q \subseteq \mathbb{Q} \Rightarrow (q, \textbf{state_t}) \in \mathbb{T}_{G\mathbb{Q}},$$

- pointers to variables of a ghost type:

$$q \subseteq \mathbb{Q} \wedge X \in \mathbb{T}_{G\mathbb{Q}} \Rightarrow (q, \textbf{ptr}(X)) \in \mathbb{T}_{G\mathbb{Q}},$$

- generic pointers:

$$\textbf{obj} \in \mathbb{T}_{G\mathbb{Q}},$$

- arrays over ghost types:

$$q \subseteq \mathbb{Q} \wedge X \in \mathbb{T}_{G\mathbb{Q}} \wedge n \in \mathbb{N} \Rightarrow (q, \textbf{array}(X, n)) \in \mathbb{T}_{G\mathbb{Q}}.$$

A value of the generic pointer type **obj** can hold arbitrary pointers including their type information. This, together with maps, can be used to formalize sets of pointers of an arbitrary type.

For C-IL + Ghost we extend the predicates defined on regular C-IL types to work for both C-IL and ghost types:

Definition 6.2 ► Pointer/array type predicates

$$\begin{aligned}
&\mathit{isptr}(t \in \mathbb{T}_\mathbb{Q} \cup \mathbb{T}_{G\mathbb{Q}}) \stackrel{\text{def}}{=} \exists t' : t = \textbf{ptr}(t'),\\
&\mathit{isarray}(t \in \mathbb{T}_\mathbb{Q} \cup \mathbb{T}_{G\mathbb{Q}}) \stackrel{\text{def}}{=} \exists t', n' : t = \textbf{array}(t', n'),\\
&\mathit{isfptr}(t \in \mathbb{T}_\mathbb{Q} \cup \mathbb{T}_{G\mathbb{Q}}) \stackrel{\text{def}}{=} \exists t', T : t = \textbf{fptr}(t', T).
\end{aligned}$$

6.1.2 Ghost Values

A value of a global ghost reference in C-IL + Ghost is represented by the following type:

Definition 6.3 ► Value of a global ghost reference

$$\begin{aligned}
&a \in (\mathbb{V} \cup \mathbb{N} \cup \mathit{val}_{\textbf{ptr}}) \times (\mathbb{N} \cup \mathbb{F})^* \wedge t \in \mathbb{T}_G \cup \mathbb{T} \wedge (\mathit{isptr}(t) \vee \mathit{isarray}(t))\\
&\implies \textbf{gref}(a, t) \in \mathit{val}_{\textbf{gref}}.
\end{aligned}$$

We consider the following kinds of ghost references, where $S \in (\mathbb{N} \cup \mathbb{F})^*$ is a finite sequence of subvariable selectors (a subvariable selector is either an array index or a struct field name) and $t \in \mathbb{T}_G \cup \mathbb{T}$ is a pointer or array type:

- $\mathbf{gref}((v, S), t)$ - a reference to a sub-variable of a global ghost variable $v \in \mathbb{V}$,
- $\mathbf{gref}((a, S), t)$ - a reference to a sub-variable of a ghost object allocated dynamically from the ghost memory at the address $a \in \mathbb{N}$,
- $\mathbf{gref}((x, S), t)$ - a reference to a ghost sub-variable of an implementation pointer value $x \in val_{ptr}$.

A reference to a local ghost variable is represented by the type:

$$a \in ((\mathbb{V} \times \mathbb{N}) \times \mathbb{T} \cup \mathbb{V} \times \mathbb{N}) \times (\mathbb{N} \cup \mathbb{F})^* \wedge t \in \mathbb{T}_G \cup \mathbb{T} \wedge (isptr(t) \vee isarray(t)) \implies \mathbf{lref}_G(a, t) \in val_{\mathbf{lref}_G}.$$

◀ **Definition 6.4**
Reference to a local ghost variable

The following kinds of local ghost variables are considered, where $S \in (\mathbb{N} \cup \mathbb{F})^*$ is a finite sequence of subvariable selectors and $t \in \mathbb{T}_G \cup \mathbb{T}$ is a pointer or array type:

- $\mathbf{lref}_G((((v, o), t'), i, S), t)$ - a reference to a ghost sub-variable of a local implementation variable $v \in \mathbb{V}$, where $o \in \mathbb{N}$ is an offset inside this variable and $i \in \mathbb{N}$ is the number of the stack frame. The local variable itself is identified by the corresponding local reference $\mathbf{lref}((v, o), i, t')$,
- $\mathbf{lref}_G((v, i, S), t)$ - a reference to a sub-variable of a local ghost variable $v \in \mathbb{V}$, where $i \in \mathbb{N}$ is a stack frame number.

A value of the generic pointer type, which includes both implementation and ghost global pointers is defined in the following way:

$$p \in val_{\mathbf{ptr}} \cup val_{\mathbf{gref}} \Rightarrow \mathbf{gval}(p, \mathbf{obj}) \in val_{\mathbf{obj}}.$$

◀ **Definition 6.5**
Value of a generic pointer type

A ghost variable of the generic pointer type can store either a pointer to the implementation memory or to the ghost memory.

A symbolic value of a ghost function is defined in the following way:

$$f \in \mathbb{F}_{name} \Rightarrow \mathbf{gfun}(f) \in val_{\mathbf{gfun}}.$$

◀ **Definition 6.6**
Value of a ghost function

Further, we provide values of special ghost types introduced in Definition 6.1:

- value of a mathematical integer:

$$i \in \mathbb{Z} \Rightarrow \mathbf{gval}(i, \mathbf{math_int}) \in val_{\mathbf{math_int}},$$

- value of a map:

$$\mathbf{map}(t', t) \in \mathbb{T}_G \wedge f \in (t2val_G(t') \mapsto t2val_G(t)) \Rightarrow \mathbf{gval}(f, \mathbf{map}(t', t)) \in val_{\mathbf{map}}.$$

  The function $t2val_G(t \in \mathbb{T} \cup \mathbb{T}_G) \in 2^{val \cup val_G}$ returns the set of all possible values of the type t and is defined in [Sch12a],

- value of a record:

$$t_C \in \mathbb{T}_C \wedge r \in (\mathbb{F} \rightharpoonup (val \cup val_G))^* \Rightarrow \mathbf{gval}(r, \mathbf{record}\ t_C) \in val_{\mathbf{record}},$$

- value of a state snapshot:

$$c \in conf_{C+G} \Rightarrow \mathbf{gval}(c, \mathbf{state_t}) \in val_{\mathbf{state_t}}.$$

Putting together inductive definitions given above we obtain the set of ghost values val_G:

Definition 6.7 ► Ghost values

$$val_G = val_{\mathbf{gref}} \cup val_{\mathbf{lref}_G} \cup val_{\mathbf{gfun}} \cup val_{\mathbf{math_int}} \cup val_{\mathbf{obj}} \cup val_{\mathbf{map}} \cup val_{\mathbf{record}} \cup val_{\mathbf{state_t}}.$$

For more information on ghost types and values as well as on arithmetic operations and functions defined on them refer to [Sch12a].

6.2 Ghost Memory

The global ghost memory in contrast to the regular memory does not have to support pointer arithmetic. Hence, we model it in a more abstract way:

Definition 6.8 ► Global ghost memory

$$\mathcal{M}_{\mathcal{G}} \in val_{\mathbf{ptr}} \cup \mathbb{N} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}.$$

The global ghost memory takes as an input an instance of one of the following types:

- a pointer of an implementation type (the value is defined only for ghost pointers of implementation type),
- the number of a ghost object on the ghost heap, or
- a global variable name (the value is defined only for ghost variables of implementation type).

The function $\mathcal{M}_{\mathcal{G}}$ returns a memory, which provides values not only for the variable itself, but also for all sub-variables of it. Hence, we call $val_{\mathcal{M}_G}$ the set of *structured ghost values*, which contains:

- non-ghost values, ghost values, and the undefined value:

$$val \cup val_G \cup \{\bot\} \subseteq val_{\mathcal{M}_G},$$

- struct and array values: $f \in (\mathbb{F} \cup \mathbb{N}) \mapsto val_{\mathcal{M}_G} \implies f \in val_{\mathcal{M}_G}$.

Ghost local variables as well as ghost sub-variables of local non-ghost variables are stored in local ghost memories of stack frames:

Definition 6.9 ► Local ghost memory

$$\mathcal{M}_{G\mathcal{E}} \in (\mathbb{V} \times \mathbb{N}) \times \mathbb{T} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}.$$

A local ghost memory takes as an input either a reference to a sub-variable of a non-ghost variable (described by the variable name, offset of the sub-variable and the type of the variable) or a name of a local ghost variable. As an output it provides the corresponding structured ghost value.

For reading and updating structured ghost values we introduce the following functions:

$$read_{val_{\mathcal{M}_G}} \in val_{\mathcal{M}_G} \times (\mathbb{N} \cup \mathbb{F})^* \mapsto val_{\mathcal{M}_G}$$
$$write_{val_{\mathcal{M}_G}} \in val_{\mathcal{M}_G} \times (\mathbb{N} \cup \mathbb{F})^* \times val_{\mathcal{M}_G} \mapsto val_{\mathcal{M}_G}.$$

For the formal definition of these functions refer to [Sch12a].

6.3 Ghost Code

We support the following types of instructions of the ghost code:

- ghost statements reading/writing ghost data and/or reading non-ghost data,
- allocation of ghost memory,
- ghost function calls of ghost functions,
- non-ghost function calls of functions extended with ghost parameters.

Expressions

The set of ghost expressions $\mathbb{E}_G$ is constructed from the set $\mathbb{E}$ by extending it to support both non-ghost and ghost types and values. Every non-ghost expression can also be a ghost expression, which operates either with implementation or ghost types and values (with the exception of some binary and unary operators which are not supported for mathematical integers). Additionally, we introduce a number of ghost expressions which are not supported by the non-ghost C-IL semantics:

- lambda expression: $t \in \mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}} \wedge v \in \mathbb{V} \wedge e \in \mathbb{E}_G \implies \textbf{lambda}(t\ v; e) \in \mathbb{E}_G$,
- record update: $e, e' \in \mathbb{E}_G \wedge f \in \mathbb{F} \implies e[f := e'] \in \mathbb{E}_G$,
- state-snapshot: $\textbf{current_state} \in \mathbb{E}_G$,
- expression in a state-snapshot: $e, e' \in \mathbb{E}_G \implies \textbf{at}(e, e') \in \mathbb{E}_G$,
- map access: $e, e' \in \mathbb{E}_G \implies e[e'] \in \mathbb{E}_G$.

For complete definition of $\mathbb{E}_G$ refer to [Sch12a].

Statements

To support function calls with ghost parameters we extend the set or non-ghost statements $\mathbb{S}$ and define a new set of annotated C-IL statements $\mathbb{S}'$ in the following way:

◀ **Definition 6.10**
Statements in C-IL + Ghost

- assignment: $e_0, e_1 \in \mathbb{E} \implies (e_0 = e_1) \in \mathbb{S}'$,
- goto: $l \in \mathbb{N} \implies (\textbf{goto}\ l) \in \mathbb{S}'$,
- if-not-goto: $l \in \mathbb{N}, e \in \mathbb{E} \implies (\textbf{ifnot}\ e\ \textbf{goto}\ l) \in \mathbb{S}'$,
- function call: $e_0, e \in \mathbb{E}, E \in \mathbb{E}^*, E' \in \mathbb{E}_G^* \implies (e_0 = \textbf{call}\ e(E, E')) \in \mathbb{S}'$,
- procedure call: $e \in \mathbb{E}, E \in \mathbb{E}^*, E' \in \mathbb{E}_G^* \implies (\textbf{call}\ e(E, E')) \in \mathbb{S}'$,
- return: $e \in \mathbb{E} \implies (\textbf{return}\ e) \in \mathbb{S}'$ and $\textbf{return} \in \mathbb{S}'$,
- compare exchange: $e_0, e_1, e_2, e_3 \in \mathbb{E} \implies \textbf{cmpxchng}(e_0, e_1, e_2, e_3) \in \mathbb{S}'$,
- VMRUN: $e_0, e_1 \in \mathbb{E} \implies \textbf{vmrun}(e_0, e_1, e_2) \in \mathbb{S}'$,

- complete TLB flush: **completeflush** $\in \mathbb{S}'$,
- INVLPGA: $e_0, e_1 \in \mathbb{E} \implies$ **invlpga**$(e_0, e_1) \in \mathbb{S}'$.

The set of ghost statements $\mathbb{S}_G$ is defined in the following way:

Definition 6.11 ► Ghost statements in C-IL + Ghost

- assignment: $e_0, e_1 \in \mathbb{E}_G \implies$ **ghost**$(e_0 = e_1) \in \mathbb{S}_G$,
- goto: $l \in \mathbb{N} \implies$ **ghost**(**goto** l) $\in \mathbb{S}_G$,
- if-not-goto: $l \in \mathbb{N}, e \in \mathbb{E}_G \implies$ **ghost**(**ifnot** e **goto** l) $\in \mathbb{S}_G$,
- function call: $e_0, e \in \mathbb{E}_G, E \in \mathbb{E}_G^*, \implies (e_0 =$ **ghost**(**call** $e(E))) \in \mathbb{S}_G$,
- procedure call: $e \in \mathbb{E}_G, E \in \mathbb{E}_G^*, \implies$ **ghost**(**call** $e(E)$) $\in \mathbb{S}_G$,
- return: $e \in \mathbb{E}_G \implies$ **ghost**(**return** e) $\in \mathbb{S}_G$ and **ghost**(**return**) $\in \mathbb{S}_G$,
- ghost allocation: $e \in \mathbb{E}_G \wedge t \in \mathbb{T}_G \implies$ **ghost**$(e = alloc(t)) \in \mathbb{S}_G$.

Note, that in regular C-IL we do not have a heap for memory allocation. The heap abstraction there should be implemented by the C-IL code performing explicit memory management. In contrast to that, we do consider an infinite heap for ghost objects in C-IL + Ghost. To manage the ghost heap we include an address of the first free location on the heap to the configuration of a C-IL + Ghost frame (see Section 6.4.1). By allocating a new variable, this address is always increased by one. As a result, in C-IL + Ghost we provide a ghost allocation statement, which allocates a ghost object of a given type on the heap. Since our ghost heap is infinite, we do not need to provide a deallocation statement.

6.4 Configuration and Program

6.4.1 Configuration

A stack frame of C-IL + Ghost consists of the same components as a stack frame of the regular C-IL (Section 5.1.4) plus the local ghost variable environment $\mathcal{M}_{\mathcal{E}G}$:

Definition 6.12 ► C-IL + Ghost frame

$$\begin{aligned} frame_{C+G} \stackrel{\text{def}}{=} [&\mathcal{M}_{\mathcal{E}} \in \mathbb{V} \mapsto (\mathbb{B}^8)^*, \mathcal{M}_{G\mathcal{E}} \in (\mathbb{V} \times \mathbb{N}) \times \mathbb{T} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G} \\ & rds \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup val_{\mathbf{gref}} \cup val_{\mathbf{lref}_G} \cup \{\bot\}, f \in \mathbb{F}_{name}, loc \in \mathbb{N}]. \end{aligned}$$

Sequential C-IL + Ghost configuration consists of the components for the non-ghost and ghost global memories, the local stack (which also includes ghost frames), the $flush_{TLB}$ bit, and the next free address on the ghost heap (i.e., a counter of the number of allocated ghost variables):

Definition 6.13 ► C-IL + Ghost configuration

$$\begin{aligned} conf_{C+G} \stackrel{\text{def}}{=} [&\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, \mathcal{M}_G \in val_{ptr} \cup \mathbb{N} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}, \\ & stack \in frame_{C+G}^*, flush_{TLB} \in \mathbb{B}, next\text{-}free_G \in \mathbb{N}]. \end{aligned}$$

Parallel C-IL + Ghost configuration is defined in a straightforward way:

Definition 6.14 ► Parallel C-IL + Ghost configuration

$$\begin{aligned} conf_{CC+G} \stackrel{\text{def}}{=} [&\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, \mathcal{M}_G \in val_{ptr} \cup \mathbb{N} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}, \\ & stack \in Tid \mapsto frame_{C+G}^*, flush_{TLB} \in Tid \mapsto \mathbb{B}, next\text{-}free_G \in Tid \mapsto \mathbb{N}]. \end{aligned}$$

The sequential configuration of a thread $t \in Tid$ is extracted from parallel configuration $c \in conf_{CC+G}$ by $c(t)$, where

$$c(t) := (c.\mathcal{M}, c.\mathcal{M}_G, c.stack(t), c.flush_{TLB}(t), c.next\text{-}free_G(t)).$$

6.4.2 Program and Context

A C-IL + Ghost program is defined in the following way

Definition 6.15 C-IL + Ghost program

$$\begin{aligned} prog_{C+G} \stackrel{\text{def}}{=} [&\mathcal{V} \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*, \mathcal{V}_G \in (\mathbb{V} \times (\mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}}))^*, \\ &T_F \in \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_{\mathbb{Q}})^*, T_{GF} \in \mathbb{T}_C \rightharpoonup (\mathbb{F} \times (\mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}}))^*, \\ &\mathcal{F} \in \mathbb{F}_{name} \rightharpoonup fun_{C+G}, \mathcal{F}_G \in \mathbb{F}_{name} \rightharpoonup gfun_{C+G}], \end{aligned}$$

where $\mathcal{V}$ is a list of global non-ghost variable declarations, $\mathcal{V}_G$ is a list of global ghost variable declarations, T_F is a type table for non-ghost fields of struct types, T_{GF} is is a type table for ghost fields of struct types, $\mathcal{F}$ is a function table for (annotated) non-ghost functions, and $\mathcal{F}_G$ is a function table for ghost functions.

A function table is defined as a partial function mapping function names $\mathbb{F}_{name}$ to function table entries. Note, that in a valid program domains of non-ghost and ghost function tables have to be disjoint.

A single entry in a non-ghost annotated function table is defined by the following type:

Definition 6.16 Annotated function table entry

$$\begin{aligned} fun_{C+G} \stackrel{\text{def}}{=} [&rettype \in \mathbb{T}_{\mathbb{Q}}, npar \in \mathbb{N}, ngpar \in \mathbb{N}, \\ &\mathcal{V} \in (\mathbb{V} \times \mathbb{T}_{\mathbb{Q}})^*, \mathcal{V}_G \in (\mathbb{V} \times (\mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}}))^*, \\ &\mathcal{P} \in (\mathbb{S} \cup \mathbb{S}_G)^* \cup \{\mathbf{extern}\}]. \end{aligned}$$

where *rettype* is a type of the function return value, *npar* is a number of function non-ghost parameters, *ngpar* is a number of function ghost parameters, $\mathcal{V}$ is a list of local variable declarations (including function parameters), $\mathcal{V}_G$ is a list of ghost local variable declarations (including ghost parameters), and $\mathcal{P}$ is a function body.

An entry in the ghost function table has the following type:

Definition 6.17 Ghost function table entry

$$\begin{aligned} gfun_{C+G} \stackrel{\text{def}}{=} [&rettype \in \mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}}, ngpar \in \mathbb{N}, \mathcal{V} \in (\mathbb{V} \times (\mathbb{T}_{\mathbb{Q}} \cup \mathbb{T}_{G\mathbb{Q}}))^*, \\ &\mathcal{P} \in (\mathbb{S} \cup \mathbb{S}_G)^* \cup \{\mathbf{extern}\}], \end{aligned}$$

Note, that all statements of a ghost function are treated as ghost statements in the operational semantics, even if they are not marked as ghost explicitly.

C-IL + Ghost semantics uses the same context $\theta \in context_{C-IL}$, as the original C-IL semantics does (Section 5.1.5).

6.5 Operational Semantics

Operational semantics of the sequential C-IL + Ghost is defined analogously to the C-IL semantics. Depending on whether the next statement to be executed is a ghost or a non-ghost one, the statement is executed either on the ghost or the non-ghost components of the configuration.

A single step of C-IL + Ghost is denoted in the same way as the step of the regular C-IL semantics:

$$\pi, \theta \vdash c \to c'.$$

In C-IL + Ghost we distinguish between ghost and implementation (i.e., non-ghost) steps. A ghost step of the program $\pi \in prog_{C+G}$ on configuration $c \in conf_{C+G}$ is denoted by

$$\pi, \theta \vdash c \xrightarrow{G} c'.$$

An implementation step is denoted by

$$\pi, \theta \vdash c \xrightarrow{I} c'.$$

The next statement to be executed is obtained with the function

$$stmt_{next}(c \in conf_{C+G}, \pi \in prog_{C+G}) \mapsto \mathbb{S} \cup \mathbb{S}_G.$$

Definition of this functions is identical to the one for the regular C-IL semantics (Definition 5.34).

What kind of step will be performed next depends on the next statement to be executed in the current configuration.

Definition 6.18 ►
Ghost step

$$\frac{stmt_{next}(c, \pi) \in \mathbb{S}_G \qquad \pi, \theta \vdash c \to c'}{\pi, \theta \vdash c \xrightarrow{G} c'}$$

Definition 6.19 ►
Implementation step

$$\frac{stmt_{next}(c, \pi) \in \mathbb{S}' \qquad \pi, \theta \vdash c \to c'}{\pi, \theta \vdash c \xrightarrow{I} c'}$$

,

In contrast to the freely interleaved scheduling of concurrent C-IL, the scheduling of concurrent C-IL + Ghost depends on the type of the statement which has to be executed next in a thread. The interleaving happens as before only between implementation steps, ghost steps can not interleave and are "attached" to the next implementation step.

A single step of the concurrent C-IL configuration $c \in conf_{CC+G}$ consists of execution of all ghost statements (if there are any) preceding an implementation statement and of this implementation statement itself.

Definition 6.20 ►
Concurrent C-IL + Ghost step

$$\frac{\begin{array}{c}\pi, \theta \vdash c(t) \xrightarrow{G}{}^{+} c'' \vee (c'' = c(t)) \\ \pi, \theta \vdash c'' \xrightarrow{I} (\mathcal{M}', \mathcal{M}'_{\mathcal{G}}, stack', flush'_{TLB}) \\ c' = (\mathcal{M}', \mathcal{M}'_{\mathcal{G}}, c.stack[t := stack'], flush'_{TLB})\end{array}}{\pi, \theta \vdash c \to c'}$$

For operational semantics of individual C-IL + Ghost statements and for details on expression evaluation consult [Sch12a].

6.6 Simulation Theorem

The execution result of a program annotated with ghost code should be the same, as the result of the same program without ghost annotations. First, we introduce two functions which extract a C-IL program from the given C-IL + Ghost program and a C-IL configuration from a given C-IL + Ghost configuration respectively:

$$cg2cil\text{-}prog(\pi \in prog_{C+G}) \in prog_{C-IL},$$
$$cg2cil(c \in conf_{C+G}, \pi \in prog_{C+G}) \mapsto conf_{C-IL}.$$

The function *cg2cil-prog* we leave undefined here. For the formal definition of this functions refer to [Sch12a]. The function *cg2cil* we define in the following way:

Definition 6.21 C-IL + Ghost to C-IL conversion

$$cg2cil(c, \pi).\mathcal{M} \stackrel{\text{def}}{=} c.\mathcal{M},$$
$$cg2cil(c, \pi).flush_{TLB} \stackrel{\text{def}}{=} c.flush_{TLB},$$
$$cg2cil(c, \pi).stack \stackrel{\text{def}}{=} cg2cil\text{-}stack(c.stack, \pi).$$

The function *cg2cil-stack* extracts the non-ghost part of the stack:

Definition 6.22 Extracting non-ghost part of the stack

$$cg2cil\text{-}stack(stack \in frame^*_{C+G}, \pi \in prog_{C+G}) \in frame^*_{C-IL},$$
$$cg2cil\text{-}stack(stack, \pi) \stackrel{\text{def}}{=} \begin{cases} \mathbf{map}(cg2cil\text{-}sf_\pi, stack[0 : si_G(\pi, stack) - 1]) & si_G(\pi, stack) \in \mathbb{N} \\ \mathbf{map}(cg2cil\text{-}sf_\pi, stack) & \text{otherwise.} \end{cases}$$

The function si_G returns the index from which the ghost part of the stack begins (i.e., starting from index $si_G(\pi, c)$ all frames in the stack of c are ghost frames):

Definition 6.23 Start of the ghost stack

$$si_G(\pi \in prog_{C+G}, stack \in frame^*_{C+G}) \in \mathbb{N} \cup \{\bot\},$$
$$si_G(\pi, c) \stackrel{\text{def}}{=} \begin{cases} \mathbf{min}\{i < |stack| \mid stack[i].f \in \pi.\mathcal{F}_G\} & \exists i : stack[i].f \in \pi.\mathcal{F}_G \\ \bot & \text{otherwise.} \end{cases}$$

One of the software conditions introduced later in this section guarantees that a ghost function never calls an implementation one. As a result, all stack frames starting from the index $si_G(\pi, c)$ must be ghost frames.

The function *cg2cil-sf* extracts the non-ghost part of a given C-IL + Ghost stack frame:

Definition 6.24 Extracting non-ghost part of the frame

$$cg2cil\text{-}sf_\pi(sf \in frame_{C+G}) \in frame_{C-IL},$$
$$cg2cil\text{-}sf_\pi(sf).\mathcal{M}_\mathcal{E} \stackrel{\text{def}}{=} sf.\mathcal{M}_\mathcal{E},$$
$$cg2cil\text{-}sf_\pi(sf).rds \stackrel{\text{def}}{=} \begin{cases} sf.rds & \notin val_G \\ \bot & \text{otherwise,} \end{cases}$$
$$cg2cil\text{-}sf_\pi(sf).f \stackrel{\text{def}}{=} .f,$$
$$cg2cil\text{-}sf_\pi(sf).loc \stackrel{\text{def}}{=}_{stmt} (\pi.\mathcal{F}(sf.f).\mathcal{P}, sf.loc).$$

The function $count_{stmt}(\mathcal{P} \in (\mathbb{S}' \cup \mathbb{S}_G)^*, loc \in \mathbb{N}) \in \mathbb{N}$ counts the number of non-ghost statements in list $\mathcal{P}$ up to location *loc*.

In order for the annotated program to behave in the same way as the annotated one does, we have to make sure that the annotated program respects a number of software conditions. Below we outline these software conditions informally:

- non-ghost expressions should not use the ghost component of the state i.e., all variables and functions occurring in a non-ghost expression of a program π_G should be also declared in $cg2cil\text{-}prog(\pi_G)$;
- left side (i.e., the address to be written) in a ghost assignment/allocation statement should evaluate to a ghost location;
- ghost code should never leave a ghost block i.e., there should be no jumps or calls from the ghost code to implementation code;
- return destinations of ghost functions should point to the ghost memory,
- ghost code should always terminate (should never "get stuck"),
- return from an implementation function or procedure must be non-ghost.

For the formal definitions of these software conditions consult [Sch12a].

We introduce a predicate which denotes that the next statement to be executed in a given configuration $c \in conf_{C+G}$ with a program $\pi \in prog_{C+G}$ satisfies all software conditions stated above :

$$ghost\text{-}safe\text{-}stmt^{\pi,\theta}_{C+G}(c \in conf_{C+G}) \in \mathbb{B}.$$

Further, we define a predicate on a C-IL program $\pi \in prog_{C+G}$ and a state $c \in conf_{C+G}$ which guarantees that execution of all ghost statements and the next implementation statement maintains ghost-safety:

Definition 6.25 ► Safety of statement execution

$$\begin{aligned} &ghost\text{-}safe\text{-}seq^{\pi,\theta}_{C+G}(c \in conf_{C+G}) \in \mathbb{B} \\ &ghost\text{-}safe\text{-}seq^{\pi,\theta}_{C+G}(c) \overset{\text{def}}{=} ghost\text{-}safe\text{-}stmt^{\pi,\theta}_{C+G}(c) \\ &\qquad \wedge (stmt_{next}(\pi, c) \in \mathbb{S}_G \wedge \pi, \theta \vdash c \to c' \implies ghost\text{-}safe\text{-}seq^{\pi,\theta}_{C+G}(c')). \end{aligned}$$

Now we state the simulation theorem between a step of the C-IL configuration and the respective sequence of steps of the C-IL + Ghost configuration.

Theorem 6.1 (C-IL + Ghost simulation (1 step)). *Let $\pi_G \in prog_{C+G}$ be an annotated program and $c \in conf_{C+G}$ be a C-IL + Ghost configuration. Further, let $\pi \in prog_{C-IL}$ and $\hat{c} \in conf_{C-IL}$ be a respective program and configuration of the regular C-IL. Then for every step of the C-IL configuration, there exists a respective sequence of steps of the ghost configuration, such that resulting*

configurations are equivalent w.r.t to the cg2cil function:

$$
\begin{aligned}
& \textit{ghost-safe-seq}^{\pi_G,\theta}_{C+G}(c) \\
& \wedge \pi = \textit{cg2cil-prog}(\pi_G) \\
& \wedge \hat{c} = \textit{cg2cil}(c, \pi_G) \\
& \wedge \pi, \theta \vdash \hat{c} \to \hat{c}' \\
& \implies \exists c', c'' : \pi_G, \theta \vdash c \xrightarrow{G}{}^{+} c'' \\
& \qquad\qquad \pi_G, \theta \vdash c'' \xrightarrow{I} c' \\
& \qquad\qquad \wedge \hat{c}' = \textit{cg2cil}(c', \pi_G).
\end{aligned}
$$

Proof. For the proof of this theorem refer to [Sch12a]. □

With the help of Theorem 6.1 one can additionally prove a simulation theorem between a concurrent C-IL + Ghost machine and a concurrent C-IL machine. In order to further apply compiler correctness theorem (Theorem 7.7) one has to show that safety of a regular C-IL program in the C-IL semantics follows from safety of the annotated program in the C-IL + Ghost semantics. In our verification proofs we do not directly use neither the regular C-IL semantics nor the C-IL + Ghost semantics, but rather work with their versions extended with the hardware component (Chapter 7). As a result, we state the properties mentioned above only for the C-IL + HW + Ghost semantics (Section 7.5).

CHAPTER 7

C-IL + HW Semantics

7.1 Configuration

7.2 Operational Semantics

7.3 C-IL + HW Program Safety

7.4 Simulation Theorem

7.5 C-IL + HW + Ghost Semantics

Hypervisor programs are normally written in a high-level language, such as C. At the same time, a hypervisor is running in parallel with guest code and in order to derive certain properties of the state of the hypervisor program we need to consider possible interaction with guests.

In this chapter we present an extension of the C-IL semantics with the hardware component, which mirrors the part of the host hardware executing guest code. We show that a regular C-IL hypervisor program running in parallel with the guest code behaves exactly the same way, as defined by our C-IL + Hardware (C-IL + HW) semantics. As a result, we can prove properties of such a program in a C program verifier by extending the program with the hardware component (and a "hardware thread" i.e., a C thread simulating the hardware) and verifying the combined program altogether. Further, we show that the hardware component of our C-IL + HW semantics simulates the memory automata of processors running in guest mode, which makes it possible to prove correct virtualization of these automata inside the C-IL + HW semantics. Finally, we combine two extensions of the C-IL semantics and obtain the C-IL + HW + Ghost semantics.

7.1 Configuration

To model guest steps on the C-IL level and to further prove virtualization of guest TLB and memory steps (Chapter 8), we extend the C-IL configuration with the component, which stores the configuration of the processor when it is running in guest mode. The part of the processor state visible on the C-IL level includes a TLB, an SB, the $CR3$ and ASID registers, as well as the memory result and request buffers:

Definition 7.1 ► Guest core state

$$core_c \stackrel{\text{def}}{=} [tlb \in Tlb, sb \in SB, CR3 \in RegCr3, asid \in \mathbb{N}, memreq \in MemReqMain, memres \in MemResMain].$$

A C-IL + HW configuration is obtained by extending the C-IL configuration with the hardware component:

Definition 7.2 ► C-IL + HW configuration

$$conf_{C+HW} \stackrel{\text{def}}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, stack \in frame_C^*, flush_{TLB} \in \mathbb{B}, p \in core_c].$$

A concurrent C-IL configuration is defined respectively:

Definition 7.3 ► Concurrent C-IL + HW configuration

$$conf_{CC+HW} \stackrel{\text{def}}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, stack \in Tid \mapsto frame_C^*, flush_{TLB} \in Tid \mapsto \mathbb{B}, p \in Tid \mapsto core_c].$$

Further, we define two functions which convert a given concurrent C-IL + HW configuration to a respective C-IL configuration and vice versa (extending a C-IL configuration with a given hardware component):

Definition 7.4 ► C-IL + HW conversion

$$chw2cil(c \in conf_{CC+HW}) \in conf_{CC-IL}$$
$$cil2chw(c \in conf_{CC-IL}, p_c \in Tid \mapsto core_c) \in conf_{CC+HW}$$
$$chw2cil(c) \stackrel{\text{def}}{=} conf_{CC-IL}[\mathcal{M} \mapsto c.\mathcal{M}, stack \mapsto c.stack, flush_{TLB} \mapsto c.flush_{TLB}]$$
$$cil2chw(c, p_c) \stackrel{\text{def}}{=} conf_{CC+HW}[\mathcal{M} \mapsto c.\mathcal{M}, stack \mapsto c.stack, flush_{TLB} \mapsto c.flush_{TLB}, p \mapsto p_c].$$

For an initial C-IL configuration one can obtain a respective initial C-IL + HW configuration (with an empty TLB component) with the help of the following function:

Definition 7.5 ► Initial C-IL + HW configuration

$$cil2chw_0(c \in conf_{CC-IL}) \in conf_{CC+HW}$$
$$cil2chw_0(c) \stackrel{\text{def}}{=} conf_{CC+HW}[\mathcal{M} \mapsto c.\mathcal{M}, stack \mapsto c.stack, flush_{TLB} \mapsto c.flush_{TLB}, p \mapsto core_c[tlb \mapsto empty\text{-}tlb(), CR3 \mapsto 0, asid \mapsto 0]].$$

Both C-IL and C-IL + HW semantics operate with the same set of expressions and the same set of rules for expression evaluation. As a result, we can state a simple lemma, which ensures equality of values of the expressions in both semantics.

Lemma 7.1 (Equality of expression evaluation). *Let $\hat{c} \in conf_{CC-IL}$ be a C-IL configuration and $c \in conf_{C+HW}$ be a C-IL + HW configuration, obtained from*

configuration $\hat{c}$ *and hardware component* $p_c \in Tid \mapsto core_c$. *Then expression evaluation of any expression results in the same value in both configurations.*

$$\forall k \in Tid : c = cil2chw(\hat{c}, p_c) \implies [e]^{\pi,\partial}_{c(k)} = [e]^{\pi,\partial}_{\hat{c}(k)}$$

Proof. The proof follows from the definition of *cil2chw* function and the fact that hardware components do not participate in expression evaluation. □

Analogous lemmas can be stated for the case of the opposite conversion (from C-IL + HW to regular C-IL) and for the values of all stack- and memory-dependent functions introduced in Chapter 5 (e.g., $stmt_{next}$, top_c, *global-consis*, *local-consis*, *safe-stmt*, $consis_{C-IL}$, or $cpoint_{C-IL}$). Definitions for all of these functions in the C-IL + HW (as well as in C-IL + Ghost and C-IL + HW + Ghost) semantics are completely identical to the ones for the regular C-IL semantics.

7.2 Operational Semantics

Operational semantics of C-IL + HW is obtained by interleaving an arbitrary step of a thread running in guest mode with implementation steps of a C-IL program. To distinguish between these steps we introduce labels to the semantics and denote a regular C-IL program step by $c \xrightarrow{cil} c'$ and a step of the hardware component by $c \xrightarrow{hw} c'$. For the steps of accepting a memory request and reporting a result of the memory access (see Section 7.2.1) we also provide parameters denoting the issued request $req \in MemReqMain$ or the reported result $res \in MemResMain$.

◂ **Definition 7.6** C-IL + HW step

$$\frac{\pi, \partial \vdash c \xrightarrow{cil} c' \lor \pi, \partial \vdash c \xrightarrow{hw} c' \lor \pi, \partial \vdash c \xrightarrow{hw(req)} c' \lor \pi, \partial \vdash c \xrightarrow{hw(res)} c'}{\pi, \partial \vdash c \rightarrow c'}$$

Operational semantics of individual hardware and C-IL steps is given in the following sections.

The sequential configuration of thread $t \in Tid$ is denoted by $c(t) := (c.\mathcal{M}, c.stack(t), c.p(t), c.flush_{TLB}(t))$ and a step of thread t is denoted by

$$\pi, \partial \vdash c(t) \rightarrow c'(t).$$

A step of the concurrent C-IL + HW semantics is a step of some thread operating on the shared memory, on its local stack, and on its local MMU component.

◂ **Definition 7.7** Step of concurrent C-IL + HW

$$\frac{\begin{array}{c}\pi, \partial \vdash c(t) \rightarrow (\mathcal{M}', stack', p', flush'_{TLB}) \\ c' = (\mathcal{M}', c.stack[t \mapsto stack', c.flush_{TLB}[t \mapsto flush'_{TLB}]], c.p[t \mapsto p'])\end{array}}{\pi, \partial \vdash c \rightarrow c'}$$

Analogously to regular C-IL we sometimes use the notation $\pi, \partial \vdash c \rightarrow_t c'$ to say that the step is performed by thread $t \in Tid$, leaving local configurations of other threads unchanged.

Additionally, in the concurrent C-IL + HW semantics we extend the labels of individual steps with the parameter identifying the thread performing this

step. I.e., the following notions are equivalent:

$$(\pi, \partial \vdash c \xrightarrow{cil(t)} c') \stackrel{\text{def}}{=} (\pi, \partial \vdash c \xrightarrow{cil}_t c')$$
$$(\pi, \partial \vdash c \xrightarrow{hw(t)} c') \stackrel{\text{def}}{=} (\pi, \partial \vdash c \xrightarrow{hw}_t c')$$
$$(\pi, \partial \vdash c \xrightarrow{hw(t,req)} c') \stackrel{\text{def}}{=} (\pi, \partial \vdash c \xrightarrow{hw(req)}_t c')$$
$$(\pi, \partial \vdash c \xrightarrow{hw(t,res)} c') \stackrel{\text{def}}{=} (\pi, \partial \vdash c \xrightarrow{hw(res)}_t c').$$

Given a step $\pi, \partial \vdash c \xrightarrow{a} c'$ the predicate $hw\text{-}step(a)$ denotes that this step is performed by the hardware component of the C-IL machine:

Definition 7.8 ► HW step of C-IL machine

$$hw\text{-}step(a) \stackrel{\text{def}}{=} a \in \{hw(t), hw(t, req), hw(t, res)\}.$$

Analogously to the regular C-IL semantics (Section 5.2), we use $\pi, \partial \vdash c \rightarrow^+ c'$ and $\pi, \partial \vdash c \rightarrow^* c'$ to denote that there exists a sequence of C-IL/HW steps starting in state c and ending in state c'. With $\rightarrow^+$ the sequence must be non-empty and with $\rightarrow^*$ it can be empty.

Given C-IL + HW states c and c', the expression $c \underset{\pi,\partial}{\xrightarrow{\beta}} c'$, where $|\beta| = n$ and $n > 0$, denotes execution sequence $c^0, \beta_0, c^1, \beta_1, \ldots, \beta_n, c^n$, where $c^0 = c$, $c^n = c'$ and every next C-IL + HW state is obtained from the previous one by performing the corresponding step from β:

$$\forall i < n : \pi, \partial \vdash c^i \xrightarrow{\beta_i} c^{i+1}.$$

7.2.1 C-IL Steps

All C-IL steps defined in Section 5.1.8, except VMRUN, TLB flush, and INVLPGA, have the same semantics in C-IL + HW with the exception that all of them can be performed only if $c.p.asid$ equals 0 (i.e., a processor executing the thread is running in hypervisor mode). Semantics of the MMU-related C-IL steps is given below.

Definition 7.9 ► C-IL VMRUN step

$$\frac{\begin{array}{c} stmt_{next}(c, \pi) = \mathbf{vmrun}(e_0, e_1, e_2) \qquad c.p.asid = 0 \\ tlb' = (c.flush_{TLB} = 1)\ ?\ empty\text{-}tlb()\ :\ c.p.tlb \\ [e_0]_c^{\pi,\partial} = \mathbf{val}(asid', u64) \qquad [e_1]_c^{\pi,\partial} = \mathbf{val}(cr3', u64) \\ idata = inject\text{-}data^{\pi,\partial}(c(k), e_2) \qquad memreq' = idata.req \\ memres' = c.p.memres[pf \mapsto idata.pf, ready \mapsto idata.ready, data \mapsto 0] \end{array}}{\begin{array}{l} \pi, \partial \vdash c \xrightarrow{cil} inc_{loc}(c'[p.tlb \mapsto tlb', p.asid \mapsto \langle asid' \rangle, p.CR3 \mapsto \langle cr3' \rangle, \\ \qquad p.memreq \mapsto memreq', p.memres \mapsto memres', flush_{TLB} \mapsto 0]) \end{array}}$$

Definition 7.10 ► C-IL complete TLB flush

$$\frac{stmt_{next}(c, \pi) = \mathbf{completeflush} \qquad c.p.asid = 0}{\pi, \partial \vdash c \xrightarrow{cil} inc_{loc}(c[flush_{TLB} \mapsto 1, p.tlb \mapsto empty\text{-}tlb()])}$$

◄ Definition 7.11 C-IL INVLPGA

$$\frac{stmt_{next}(c, \pi) = \mathbf{invlpga}(e_0, e_1) \qquad c.p.asid = 0 \qquad \mathbf{val}(va, u64) = [e_0]_c^{\pi,\theta} \qquad \mathbf{val}(asid, u64) = [e_1]_c^{\pi,\theta}}{\pi, \theta \vdash c \xrightarrow{cil} inc_{loc}(c[p.tlb \mapsto inval\text{-}tlb(c.p.tlb, \langle va \rangle.pfn, \langle asid \rangle))}$$

With definitions given in this section we are able to relate the result of a C-IL step performed in the C-IL semantics and the same step performed in the C-IL + HW semantics.

Lemma 7.2 (C-IL step transfer). *Let $\hat{c} \in conf_{CC-IL}$ be a C-IL configuration and $c \in conf_{CC+HW}$ be a C-IL + HW configuration, obtained from configuration $\hat{c}$ and hardware component $p_c \in Tid \mapsto core_c$. Further, let $\hat{c}$ perform a C-IL step from $\hat{c}$ to $\hat{c}'$. Then the C-IL + HW configuration also performs the same step. The resulting MMU state in configuration c' depends on whether this step involves execution of a certain virtualization statement or not.*

$$\begin{aligned}
&\pi, \theta \vdash \hat{c} \to_k \hat{c}' \\
&\wedge c = cil2chw(\hat{c}, p_c) \\
&\implies \pi, \theta \vdash c \xrightarrow{cil}_k c' \\
&\quad \wedge stmt_{next}(\hat{c}(k), \pi) \notin \{\boldsymbol{vmrun}(E), \boldsymbol{invlpga}(E), \boldsymbol{completeflush}\} \\
&\qquad \implies c' = cil2chw(\hat{c}', p_c) \\
&\quad \wedge stmt_{next}(\hat{c}(k), \pi) = \boldsymbol{vmrun}(e_0, e_1, e_2) \\
&\qquad \wedge [e_0]_c^{\pi,\theta} = \boldsymbol{val}(asid', u64) \wedge [e_1]_c^{\pi,\theta} = \boldsymbol{val}(cr3', u64) \\
&\qquad \wedge inject\text{-}data^{\pi,\theta}(c(k), e_2) = idata \\
&\qquad \implies c' = cil2chw(\hat{c}', p_c[k \mapsto p'_k]) \\
&\qquad\qquad \wedge p'_k = (p_c[k])[asid \mapsto \langle asid' \rangle, CR3 \mapsto \langle cr3' \rangle, \\
&\qquad\qquad\qquad tlb \mapsto (c.flush_{TLB}\ ?\ tlb\text{-}empty()\ :\ p_c[k].tlb), \\
&\qquad\qquad\qquad memreq = idata.req, \\
&\qquad\qquad\qquad memres.pf \mapsto idata.pf, memres.data \mapsto 0, \\
&\qquad\qquad\qquad memres.ready \mapsto idata.ready] \\
&\quad \wedge stmt_{next}(\hat{c}(k), \pi) = \boldsymbol{invlpga}(e_0, e_1) \\
&\qquad \wedge [e_0]_c^{\pi,\theta} = \boldsymbol{val}(va, u64) \wedge [e_1]_c^{\pi,\theta} = \boldsymbol{val}(asid, u64) \\
&\qquad \implies c' = cil2chw(\hat{c}', p_c[p_c[k].tlb \mapsto inval\text{-}tlb(c.p.tlb, \langle va \rangle.pfn, \langle asid \rangle)]) \\
&\quad \wedge stmt_{next}(\hat{c}(k), \pi) = \boldsymbol{completeflush} \\
&\qquad \wedge [e_0]_c^{\pi,\theta} = \boldsymbol{val}(va, u64) \wedge [e_1]_c^{\pi,\theta} = \boldsymbol{val}(asid, u64) \\
&\qquad \implies c' = cil2chw(\hat{c}', p_c[p_c[k].tlb \mapsto empty\text{-}tlb()])
\end{aligned}$$

Proof. By a case split on the type of the step $\hat{c} \to_k \hat{c}'$. For every case the proof follows from definitions and Lemma 7.1, which guarantees that the value of every expression in $c(k)$ is equal to the value of the same expression in $\hat{c}(k)$. □

7.2.2 Hardware Steps

I/O Steps

Since we do not add the instruction automaton of the core into C-IL + HW semantics and prove correct virtualization only for memory operations, we

allow any memory request to be raised non-deterministically at any time when the ASID of a thread not equals 0[1].

Definition 7.12 ► Accepting memory request

$$\frac{c.p.asid \neq 0 \qquad req \in MemReqMain}{\pi, \partial \vdash c \xrightarrow{hw(req)} c[p.memreq \mapsto req]}$$

Another non-deterministic step reflects the effect of a *core-send-mem-res* step on the hardware component of the C-IL + HW machine.

Definition 7.13 ► Reporting memory result

$$\frac{c.p.asid \neq 0 \qquad c.p.memres.ready = 1 \qquad res = c.p.memres}{\pi, \partial \vdash c \xrightarrow{hw(res)} c[p.memres.ready \mapsto 0]}$$

MMU Steps

For reading and writing of an abstract PTE from/to the C-IL memory we introduce the following functions [2]:

Definition 7.14 ► Reading PTE from C-IL memory

$$\begin{aligned}
&\textit{read-pte}_c(\mathcal{M} \in \mathbb{B}_{mem} \mapsto \mathbb{B}^8, pa \in \mathbb{B}^{qpa}) \in AbsPte,\\
&\textit{write-pte}_c(\mathcal{M} \in \mathbb{B}_{mem} \mapsto \mathbb{B}^8, pa \in \mathbb{B}^{qpa}, pte \in AbsPte) \in \mathcal{M},\\
&\textit{read-pte}_c(\mathcal{M}, pa) \overset{\text{def}}{=} \textit{abs-pte}(c.\mathcal{M}[pa : pa+3]),\\
&\textit{write-pte}_c(\mathcal{M}, pa, pte) \overset{\text{def}}{=} \lambda pa' \in \mathbb{B}_{mem} :\\
&\quad\begin{cases} byte_i(\textit{concrete-pte}(pte)) & i \in [0:7] \wedge pa' = (pa \circ 0^3) + i\\ \mathcal{M}[pa'] & \text{otherwise.}\end{cases}
\end{aligned}$$

A new top-level walk is added to the TLB with all rights enabled and with the PFN field set to the value of the $CR3$ register.

Definition 7.15 ► MMU create walk step

$$\frac{\begin{array}{c} c.p.asid \neq 0 w.l = 4 \qquad w.asid = c.p.asid\\ w.r = Rights[ex \mapsto 1, us \mapsto 1, rw \mapsto 1] \qquad w.mt = \textit{root-pt-memtype}(p.CR3)\\ w.pfn = c.p.CR3.pfn \qquad tlb' = c.p.tlb[w \mapsto true] \qquad c' = c[p.tlb \mapsto tlb'] \end{array}}{\pi, \partial \vdash c \xrightarrow{hw} c'}$$

During walk extension we read a PTE from the C-IL memory, calculate the new walk (it should be non-faulty) and add it to the TLB.

Definition 7.16 ► MMU extend walk step

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad c.p.tlb[w] = 1 \qquad w.asid = c.p.asid\\ pa = \textit{pte-addr}(w.pfn, w.vpfn.px[w.l])\\ pte = \textit{read-pte}_c(c.\mathcal{M}, pa) \qquad wext_{\surd}(w, pte, r) \qquad w' = wext(w, pte, r)\\ tlb' = c.p.tlb[w' \mapsto true] \qquad c' = c[p.tlb \mapsto tlb'] \end{array}}{\pi, \partial \vdash c \xrightarrow{hw} c'}$$

[1] To prove correctness of virtulization not only for memory accesses, but also for instruction execution one has to model instruction part of the core in detail and lift this model to the C-IL + HW semantics.

[2] Note, that if we reduced MMUs under a mapping *hpa2spa*, which is not an identity mapping (see Section 4.5), then MMU steps would perform accesses to the C-IL memory under $hpa2spa^{-1}$ mapping applied to the *w.pfn* field, rather than directly. Analogously, guest steps would have to update memory under $hpa2spa^{-1}$ applied to an address from the set *GuestAddr*.

MMU performs setting of access/dirty bits by writing the respective entries in the global C-IL memory.

Definition 7.17
MMU set A/D step

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad c.p.tlb[w] = 1 \qquad w.asid = c.p.asid \\ pa = pte\text{-}addr(w.pfn, w.vpfn.px[w.l]) \\ pte = read\text{-}pte_c(c.\mathcal{M}, pa) \qquad \neg complete(w) \qquad pte' = pte\text{-}set\text{-}ad\text{-}bits(pte, w) \\ \mathcal{M}' = write\text{-}pte(c.\mathcal{M}, pa, pte') \qquad c' = c[\mathcal{M} \mapsto \mathcal{M}'] \end{array}}{\pi, \vartheta \vdash c \xrightarrow{hw} c'}$$

Note, that we don't introduce walk removal to the C-IL + HW semantics because in the MMU consistency relation (Section 7.4.1) we require the hardware TLB to be a subset of the software TLB, rather than to be equal to it. The walks from the software TLB can be removed only by executing a complete flush or an INVLPG statement.

Core and SB Steps

A VMEXIT step in the C-IL + HW semantics sets the ASID of the thread to zero. Hence, no further MMU/guest steps can occur after VMEXIT and until the next VMRUN is executed.

Definition 7.18
Guest VMEXIT step

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad is\text{-}empty(c.p.sb) \qquad c.p.memres.ready = 0 \\ c.p.memreq.type \in \{mov2cr3, invlpg\text{-}asid, vmexit\} \cup MemAcc \\ c.p.memreq.type \notin MemAcc \implies c.p.memreq.active = 1 \wedge tlb' = tlb \\ c.p.memreq.type \in MemAcc \implies c.p.memreq.pf\text{-}flush\text{-}req = 1 \\ \wedge\ c.p.memreq.active = 0 \\ \wedge\ tlb' = pf\text{-}inval\text{-}tlb(c.p.tlb, c.p.memreq.pa.vpfn, c.p.asid) \end{array}}{\pi, \vartheta \vdash c \xrightarrow{hw} c[p.asid \mapsto 0, p.tlb \mapsto tlb']}$$

Note, that we do not update the buffer $c.p.memreq$ in case of a VMEXIT event. Hence, when the hypervisor code is executed, this buffer contains the parameters of the last memory access which was issued in guest mode before the VMEXIT event has occurred.

For masked updates of the C-IL memory during guest memory writes we use the following functions:

Definition 7.19
C-IL guest memory update

$$masked\text{-}update_c(\mathcal{M} \in \mathbb{B}_{mem} \mapsto \mathbb{B}^8, pa \in \mathbb{B}^{qpa}, data \in \mathbb{B}^{64}, mask \in \mathbb{B}^8) \in \mathcal{M}$$

$$masked\text{-}update_c(\mathcal{M}, pa, data, mask) \stackrel{\text{def}}{=} \lambda pa' \in \mathbb{B}_{mem} : \begin{cases} byte_i(combine(c.\mathcal{M}[pa : pa+3], \\ \qquad (data, mask)) & i \in [0:7] \wedge pa' = (pa \circ 0^3) + i \\ \mathcal{M}[pa'] & \text{otherwise.} \end{cases}$$

Below we define all remaining steps of the hardware component of a thread, which resemble the respective steps of the reduced hardware machine. In Definitions 7.20 - 7.27 we apply the hardware semantics from Chapter 3 to

the hardware component of the C-IL + HW machine.

Definition 7.20 ► Guest memory write

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad tlb\text{-}transl\text{-}ready(c.p.memreq, c.p.asid, c.p.tlb, w) \\ c.p.memreq.type = write \qquad pa = w.pfn \circ c.p.memreq.va.off \\ data = c.p.memreq.data \qquad mask = c.p.memreq.mask \\ store = Store[pa \mapsto pa, data \mapsto data, mt \mapsto w.mt, mask \mapsto mask] \\ sb' = write(c.p.sb, store) \\ memres' = p.memres[ready \mapsto 1, pf \mapsto no\text{-}page\text{-}fault(), data \mapsto 0] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[p.sb \mapsto sb', p.memreq.active \mapsto 0, p.memres \mapsto memres']}$$

Definition 7.21 ► Guest memory read

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad tlb\text{-}transl\text{-}ready(c.p.memreq, c.p.asid, c.p.tlb, w) \\ c.p.memreq.type = read \qquad pa = w.pfn \circ c.p.memreq.va.off \\ data' = combine(c.\mathcal{M}[pa : pa+3], forward(c.p.sb, pa)) \\ memres' = c.p.memres[ready \mapsto 1, data \mapsto data, \\ \qquad pf \mapsto no\text{-}page\text{-}fault(), data \mapsto data'] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[p.sb \mapsto sb', p.memreq.active \mapsto 0, p.memres \mapsto memres']}$$

Definition 7.22 ► Guest locked memory write

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad tlb\text{-}transl\text{-}ready(c.p.memreq, c.p.asid, c.p.tlb, w) \\ c.p.memreq.type = locked\text{-}write \qquad pa = w.pfn \circ c.p.memreq.va.off \\ data = c.p.memreq.data \qquad mask = c.p.memreq.mask \\ \mathcal{M}' = masked\text{-}update_c(\mathcal{M}, pa, data, mask) \\ memres' = p.memres[ready \mapsto 1, pf \mapsto no\text{-}page\text{-}fault(), data \mapsto 0] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[\mathcal{M} \mapsto \mathcal{M}', p.memreq.active \mapsto 0, p.memres \mapsto memres']}$$

Definition 7.23 ► Guest atomic compare-exchange

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad tlb\text{-}transl\text{-}ready(c.p.memreq, c.p.asid, c.p.tlb, w) \\ c.p.memreq.type = atomic\text{-}cmpxchng \qquad pa = w.pfn \circ c.p.memreq.va.off \\ mask = c.p.memreq.mask \qquad cmp\text{-}data = c.p.memreq.cmp\text{-}data \\ store\text{-}data = \begin{cases} c.p.memreq.data & meq(c.\mathcal{M}[pa : pa+3], cmp\text{-}data, mask) \\ c.\mathcal{M}[pa : pa+3] & \text{otherwise} \end{cases} \\ \mathcal{M}' = masked\text{-}update_c(\mathcal{M}, pa, store\text{-}data, mask) \\ memres' = c.p.memres[ready \mapsto 1, data \mapsto c.\mathcal{M}[pa : pa+3], \\ \qquad pf \mapsto no\text{-}page\text{-}fault()] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[\mathcal{M} \mapsto \mathcal{M}', p.memreq.active \mapsto 0, p.memres \mapsto memres']}$$

Definition 7.24 Guest triggering PF (stage 1)

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad c.p.memreq.active = 1 \\ c.p.memreq.type \in MemAcc \qquad c.p.memres.ready = 0 \\ pa = pte\text{-}addr(w.pfn, w.vpfn.px[w.l]) \qquad pte = read\text{-}pte_c(c.\mathcal{M}, pa) \\ tlb\text{-}fault\text{-}ready(c.p.memreq, c.p.asid, tlb[i], pte, w) \\ memres' = c.p.memreq[ready \mapsto 1, pf.va \mapsto c.p.memreq.va, \\ pf.r \mapsto c.p.memreq.r, pf.fault \mapsto 1, data \mapsto 0 \\ pf.fault\text{-}code \mapsto page\text{-}fault\text{-}code(req.r, pte.p, pte.v)] \\ memreq' = c.p.memreq[active \mapsto 0, pf\text{-}flush\text{-}req \mapsto 1] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[p.memres \mapsto memres', p.memreq \mapsto memreq']}$$

Definition 7.25 Guest SB commit store

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad 0 < |c.p.sb.buffer| \qquad store = c.p.sb.buffer[0] \\ store \neq SFENCE \qquad pa = store.pa \qquad mask \mapsto store.mask \\ \mathcal{M}' = masked\text{-}update_c(\mathcal{M}, pa, store.data, mask) \\ sb' = c.p.sb[buffer \mapsto \mathbf{tl}(c.p.sb.buffer)] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[\mathcal{M} \mapsto \mathcal{M}', c.p.sb \mapsto sb']}$$

Definition 7.26 Guest SB reorder stores

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad j < |c.p.sb.buffer| - 1 \\ c.p.sb.buffer[j] \neq SFENCE \qquad c.p.sb.buffer[j+1] \neq SFENCE \\ c.p.sb.buffer[j].pa \neq c.p.sb.buffer[j+1].pa \\ c.p.sb.buffer[j].WC \vee c.p.sb.buffer[j+1].WC \\ buffer' = c.p.sb.buffer[j \mapsto c.p.sb.buffer[j+1], (j+1) \mapsto c.p.sb.buffer[j]] \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[\mathcal{M} \mapsto \mathcal{M}', c.p.sb \mapsto c.p.sb[buffer \mapsto buffer']}$$

Definition 7.27 Guest SB drop SFENCE

$$\frac{\begin{array}{c} c.p.asid \neq 0 \qquad 0 < |c.p.sb.buffer| \qquad c.p.sb.buffer[0] = SFENCE \\ buffer' = \mathbf{tl}(c.p.sb.buffer) \end{array}}{\pi, \theta \vdash c \xrightarrow{hw} c[\mathcal{M} \mapsto \mathcal{M}', c.p.sb \mapsto c.p.sb[buffer \mapsto buffer']}$$

7.2.3 C-IL + HW I/O Traces

With a closer look at the operational semantics introduced in this chapter one can also interpret C-IL + HW machine as a classical I/O automaton performing internal and external actions. The set of external actions in this case consists of only two actions:

- accepting a memory request (Definition 7.12), which is the only input action and,
- completing a memory operation (Definition 7.13), which is the only output action.

These external actions correspond to the same kind of actions performed by the memory automaton of the hardware machine (Section 3.2.2). As a result, we can show correspondence between a trace (i.e., the sequence of external actions) of the reduced hardware machine and a respective trace of the C-IL + HW machine. This correspondence of traces later allows us to prove virtualization properties about requests/replies of the memory automata running in virtualization mode solely on software level (e.g., in a C verifier) and then transfer the properties down to the hardware level (see Chapter 8).

Given an execution sequence $c^0 \overset{\beta}{\underset{\pi,\vartheta}{\rightarrow}} c^n$ of the concurrent C-IL + HW machine, where $\beta = n$ and $n > 0$, we use the following function to extract a trace of external actions:

Definition 7.28 ► C-IL + HW I/O Trace

$$hw\text{-}trace(\beta) \overset{\text{def}}{=} \begin{cases} \beta_0 & \beta_0 \in ext(\text{C-IL+HW}) \wedge |\beta| = 1 \\ \beta_0 \circ hw\text{-}trace(\mathbf{tl}(\beta)) & \beta_0 \in ext(\text{C-IL+HW}) \wedge |\beta| > 1 \\ hw\text{-}trace(\mathbf{tl}(\beta)) & \text{otherwise.} \end{cases}$$

The set $ext(\text{C-IL+HW})$ consists of all possible labels of input and output actions of the C-IL + HW semantics:

$$ext(\text{C-IL+HW}) \overset{\text{def}}{=} \{hw(i, req), hw(i, res) \mid \\ i \in Pid, req \in MemReqMain, res \in MemResMain\}.$$

Analogously, we extract the set of external actions performed in guest mode from the execution sequence $h^0 \overset{\beta}{\rightarrow} h^n$ of the reduced hardware machine:

Definition 7.29 ► Guest hardware trace

$$guest\text{-}trace(\beta) \overset{\text{def}}{=} \begin{cases} \beta_0 & \beta_0 \in ext(RedHardw) \wedge |\beta| = 1 \\ & \wedge h^0.p[pid(\beta_0)].asid \neq 0 \\ \beta_0 \circ guest\text{-}trace(\mathbf{tl}(\beta)) & \beta_0 \in ext(RedHardw) \wedge |\beta| > 1 \\ & \wedge h^0.p[pid(\beta_0)].asid \neq 0 \\ guest\text{-}trace(\mathbf{tl}(\beta)) & \text{otherwise.} \end{cases}$$

The set $ext(RedHardw)$ consists of labels of input and output actions of the memory automaton of the reduced hardware machine:

$$ext(RedHardw) \overset{\text{def}}{=} \{core\text{-}issue\text{-}mem\text{-}req(i, req), core\text{-}send\text{-}mem\text{-}res(i, res) \mid \\ i \in Pid, req \in MemReqMain, res \in MemResMain\}.$$

Now we can talk about the equivalence of guest I/O traces on the hardware and on the C-IL level. Given a guest hardware trace β and a C-IL + HW I/O trace ω we define their equivalence in the following way:

Definition 7.30 ► Equivalent hardware and C-IL I/O traces

$$(\beta \equiv \omega) \overset{\text{def}}{=} |\beta| = |\omega| \wedge \forall i < |\beta| : \\ (\omega_i = core\text{-}issue\text{-}mem\text{-}req(i, req) \implies \beta_i = hw(i, req.main)) \\ \wedge (\omega_i = core\text{-}send\text{-}mem\text{-}res(i, res) \implies \beta_i = hw(i, res.main)).$$

7.3 C-IL + HW Program Safety

For the C-IL + HW semantics we extend the program safety definition from Section 5.3.4 to include the TLB safety properties from Invariant 4.41.

The safety for guest TLBs in the C-IL + HW semantics is defined as follows.

◀ **Invariant 7.31**
Safe TLBs
(C-IL + HW semantics)

name	$safe\text{-}tlbs_c(p \in core_c, o \in Ownership, k \in Pid)$
property	$p.tlb[w] \wedge w.l \neq 0 \wedge p.asid \neq 0 \wedge w.asid = p.asid$ $\wedge\; bva \in qword2bytes(pte\text{-}addr(w.pfn, w.vpfn.px[w.l]))$ $\implies bva \in SharedAddr \cup o[k]$, $p.tlb[w] \wedge w.l = 0 \wedge p.asid \neq 0 \wedge w.asid = p.asid$ $\implies pfn2bytes(w.pfn) \subseteq GuestAddr$, $p.asid \neq 0 \implies cacheable\text{-}walks(h.tlb[i], p.asid)$

Note, that we state the TLB safety property only for threads executing MMU steps (i.e., with $p[i].asid \neq 0$).

Safety of a given state of the C-IL + HW semantics is then stated as follows:

◀ **Definition 7.32**
Safe configuration
(C-IL + HW semantics)

$$safe\text{-}conf^{\pi,\theta}_{C+HW}(c \in conf_{C+HW}, o \in Ownership, k \in Tid) \in \mathbb{B},$$
$$safe\text{-}conf^{\pi,\theta}_{C+HW}(c, o, k) \stackrel{\text{def}}{=} safe\text{-}stmt^{\pi,\theta}(stmt_{next}(c, \pi), c, o, k) \wedge safe\text{-}tlbs_c(c.p[k], o, k).$$

An execution sequence $c \xrightarrow[\pi,\theta]{\beta} c'$ of the concurrent C-IL + HW machine $c \in conf_{CC+HW}$ starting with the ownership setting $o \in Ownership$ is safe if every state in this sequence is safe and the ownership transfer is safe:

◀ **Definition 7.33**
Safe sequence
(C-IL + HW semantics)

$$safe\text{-}seq^{\pi,\theta}_{CC+HW}(\beta, o) \in \mathbb{B},$$
$$\begin{aligned} safe\text{-}seq^{\pi,\theta}_{CC+HW}(\beta, o) \stackrel{\text{def}}{=}\; & \forall t \in Tid : safe\text{-}conf^{\pi,\theta}_{C+HW}(c(t), o, t) \\ & \wedge (|\beta| > 0 \wedge \pi, \theta \vdash c \xrightarrow{\beta_0} c' \wedge \beta_0 = cil(k) \\ & \quad \implies \exists o' : safe\text{-}seq^{\pi,\theta}_{CC+HW}(\mathbf{tl}(\beta), o') \\ & \qquad \wedge\; safe\text{-}transfer^{\pi,\theta}(c(k), c'(k), k, o, o')) \\ & \wedge (|\beta| > 0 \wedge \pi, \theta \vdash c \xrightarrow{\beta_0} c' \wedge hw\text{-}step(\beta_0) \\ & \quad \implies safe\text{-}seq^{\pi,\theta}_{CC+HW}(\mathbf{tl}(\beta), o)). \end{aligned}$$

Note, that we do not allow the ownership transfer to occur during steps of the hardware component.

A program $\pi \in prog_{C-IL}$ with the initial configuration $c \in conf_{CC+HW}$ is safe if every possible execution sequence of π is safe:

◀ **Definition 7.34**
Safe program
(C-IL + HW semantics)

$$safe\text{-}prog^{\pi,\theta}_{CC+HW}(c \in conf_{CC+HW}, o \in Ownership) \in \mathbb{B},$$
$$safe\text{-}prog^{\pi,\theta}_{CC+HW}(c, o) \stackrel{\text{def}}{=} \forall c', \beta : \pi, \theta \vdash c \xrightarrow{\beta} c' \implies safe\text{-}seq^{\pi,\theta}_{CC+HW}(\beta, o).$$

The following lemma establishes safety of the local C-IL execution sequence extracted from a C-IL + HW execution.

Lemma 7.3 (C-IL local sequence safe). *Let $\pi \in prog_{C-IL}$ be a safe C-IL + HW program. Further, let c' be a state obtained from c by execution of a number of C-IL steps of thread k. Then (i) local sequence of thread k is safe w.r.t the C-IL semantics and (ii) C-IL + HW program safety is maintained in state c':*

$$\begin{aligned}&safe\text{-}prog^{\pi,\theta}_{CC+HW}(c, o) \wedge \pi, \theta \vdash c \xrightarrow{cil}{}^{+}_{k} c' \wedge \hat{c} = chw2cil(c) \wedge \hat{c}' = chw2cil(c')\\ &\implies \exists o' : safe\text{-}local\text{-}seq^{\pi,\theta}_{C-IL}(\hat{c}(k), \hat{c}'(k), k, o, o') \wedge safe\text{-}prog^{\pi,\theta}_{CC+HW}(c', o')\end{aligned}$$

Proof. Follows from Definition 7.34 and the definition of the safe sequence of C-IL steps (Definition 5.68). □

Another lemma states transitivity of program safety for a single step of the hardware component.

Lemma 7.4 (Safe C-IL + HW program transitive (HW step)). *Let $\pi \in prog_{C-IL}$ be a safe program w.r.t to state $c \in conf_{C+HW}$ and ownership $o \in Ownership$. Further, let c' be a state obtained from c with a single step of the hardware component. Then π is also safe w.r.t to c' and o.*

$$safe\text{-}prog^{\pi,\theta}_{CC+HW}(c, o) \wedge \pi, \theta \vdash c \xrightarrow{a} c' \wedge hw\text{-}step(a) \implies safe\text{-}prog^{\pi,\theta}_{CC+HW}(c', o)$$

Proof. Follows from Definition 7.34. □

7.4 Simulation Theorem

7.4.1 HW Consistency

We introduce an additional consistency relation, which couples the state of hardware processor i with the respective components of the C-IL + HW semantics:

Definition 7.35 ► HW consistency

$$hw\text{-}consis(c \in conf_{CC+HW}, h \in RedHardw, i \in Pid) \in \mathbb{B}$$

$$\begin{aligned}hw\text{-}consis(c, h, i) \stackrel{\text{def}}{=}\ & h.asid[i] = c.p[i].asid\\ \wedge\ & h.asid[i] = 0 \implies (h.tlb[i] \subseteq c.p[i].tlb \vee c.flush_{TLB}[i])\\ & \qquad\qquad\qquad \wedge is\text{-}empty(c.p[i].sb)\\ \wedge\ & h.asid[i] \neq 0 \implies h.tlb[i] \subseteq c.p[i].tlb\\ & \qquad\qquad\qquad \wedge h.memreq[i].main = c.p[i].memreq\\ & \qquad\qquad\qquad \wedge h.memres[i].main = c.p[i].memres\\ & \qquad\qquad\qquad \wedge h.sb[i] = c.p[i].sb\\ & \qquad\qquad\qquad \wedge h.CR3[i] = c.p[i].CR3.\end{aligned}$$

For buffers *memreq* and *memres* from the hardware configuration we couple only the part which is relevant for the guest execution. For the TLB component we require the hardware TLB to be the subset of $c.p[i].tlb$, rather than to be equal to it. This allows us to leave the walks in the software TLB component, when the hardware TLB drops them non-deterministically. The walks from $c.p[i].tlb$ are removed only when an INVLPG or a complete flush is requested by the user. Note also, that we disable the coupling invariant for the TLB when the bit $c.flush_{TLB}[i]$ is set and the processor is running in hypervisor mode. The bit $c.flush_{TLB}[i]$ guarantees that at the next VMRUN the hardware TLB flush

will be performed, which will enable the coupling for the TLB component once again.

Putting together regular C-IL consistency and hardware consistency we get the main consistency relation we aim at:

$$consis_{CC+HW}(c, \pi, h, i) \stackrel{\text{def}}{=} hw\text{-}consis(c, h, i) \wedge global\text{-}consis(c.\mathcal{M}, \pi, h) \\ \wedge\ local\text{-}consis_i(c.stack[i], h.p[i].state, h.mm[StackAddr_i]).$$

◀ **Definition 7.36**
C-IL + HW consistency

The following lemma states that consistency is maintained after any step of a processor running in guest mode.

Lemma 7.5 (Safe and consistent guest step). *Let hardware $h \in RedHardw$ perform a step of processor i running in guest mode and resulting in state h' Let $c \in conf_{CC+HW}$ be a configuration of safe program $\pi \in prog_{C-IL}$ and consistency for thread i hold between c and h, where h is a valid hardware state. Then configuration h' is also safe and there exists configuration c' s.t. consistency for thread i holds between h' and c', c' either equals to c or $\pi, \partial \vdash c \xrightarrow{hw} c'$, and traces of the C step and the hardware step are equivalent. Moreover, consistency for all other threads holds in c', if it holds in c:*

$$\begin{aligned}
& safe\text{-}prog^{\pi,\partial}_{CC+HW}(c, o) \\
& \wedge\ h \xrightarrow{a} h' \\
& \wedge\ pid(a) = i \\
& \wedge\ safe\text{-}conf_r(h, o) \\
& \wedge\ consis_{CC+HW}(c, \pi, h, i) \\
& \wedge\ h.p[i].asid \neq 0 \\
& \implies \exists c' : consis_{CC+HW}(c', \pi, h', i) \\
& \qquad\qquad \wedge\ safe\text{-}conf_r(h', o) \\
& \qquad\qquad \wedge\ (\pi, \partial \vdash c \xrightarrow{b} c' \wedge hw\text{-}step(b) \wedge pid(b) = i \\
& \qquad\qquad\qquad\quad \wedge\ guest\text{-}trace(a) \equiv hw\text{-}trace(b) \\
& \qquad\qquad\qquad \vee\ c = c' \wedge guest\text{-}trace(a) = \{\}) \\
& \qquad\qquad \wedge\ (\forall t \in Pid : consis_{CC+HW}(c, \pi, h, t) \implies consis_{CC+HW}(c', \pi, h', t)).
\end{aligned}$$

Proof. By a case split on the type of the hardware step from h to h'.

Case 1: $h \xrightarrow{a} h'$ is an internal step of the instruction automaton. In this case the part of the hardware state visible on the C-IL + HW level is unchanged, as well as the state of the physical memory. Hence, $hw\text{-}consis(c, h', i)$ and $global\text{-}consis(c.\mathcal{M}, \pi, h')$ hold, and we choose $c' = c$. Assuming that local C-IL consistency is stable under guest steps which don't write the local memory of a thread (see Section 5.4.4), we get

$$\forall t \in Pid : consis_{CC+HW}(c, \pi, h, t) \implies consis_{CC+HW}(c, \pi, h', t).$$

The safety for h' is trivially maintained from the fact that the TLB state and the state of processors running in hypervisor mode is unchanged.

Case 2: $h \xrightarrow{a} h'$ is not a TLB step, but performs a write to the main memory at the quad-word address pa. From $hw\text{-}consis(c, h, i)$, we know that

the hardware TLB is a subset of the software TLB component and the content of the buffers is the same on hardware and software levels. Hence, we can perform the same step with the C-IL machine, performing a locked memory write, a compare-exchange, or an SB commit store step:

$$\pi, \partial \vdash c \xrightarrow{hw} c'.$$

Since we write the same data to the hardware memory and to the C-IL memory, hardware consistency relation is maintained. Hardware memory regions where the compiled code and the local stacks are located are left unchanged. Hence, code and stack consistency are also maintained (assuming stability of local consistency under guest steps). The state of TLBs is unchanged, as well as the state of processors running in hypervisor mode. This implies $\mathit{safe\text{-}conf}_r(h', o)$.

Case 3: $h \overset{a}{\to} h'$ is an MMU step of adding a new walk with the ASID other than 0 to the TLB. The C-IL configuration performs the same kind of an MMU step, adding a new walk to $c.p[i].tlb$ (even if this walk was already present there before) and producing configuration c' s.t.

$$\pi, \partial \vdash c \xrightarrow{hw} c'.$$

From hardware consistency, we know that $h.CR3[i] = c.p[i].CR3$. Hence, all parameters of the newly added walk in the hardware configuration and in C are the same and equality of TLB states is maintained after the step. The hardware memory is unchanged. Hence, code and stack consistency are also maintained (assuming stability of local consistency under guest steps). From $\mathit{safe\text{-}prog}^{\pi,\partial}_{CC+HW}(c, o)$ applying Lemma 7.4, we get $\mathit{safe\text{-}prog}^{\pi,\partial}_{CC+HW}(c', o)$, which implies

$$\forall i : \mathit{safe\text{-}tlbs}_c(c'.p[i], o, i).$$

The state of processors running in hypervisor mode is unchanged. Hence, $\mathit{safe\text{-}conf}_r(h', o)$ also holds.

Case 4: $h \overset{a}{\to} h'$ is an MMU step of extending walk w with ASID other than 0. The C-IL configuration performs the same kind of an MMU step, extending the walk w and producing configuration c' s.t.

$$\pi, \partial \vdash c \xrightarrow{hw} c'.$$

From $\mathit{safe\text{-}conf}_r(h, o)$ we know that the walk w points to a PTE located at the address from $\mathit{SharedAddr}$. The content of the shared memory in the hardware machine and in the C-IL machine is the same (according to $\mathit{global\text{-}consis}(c.\mathcal{M}, \pi, h)$). Hence, all parameters of the extended walk in the hardware configuration and in the C-IL configuration are the same and equality of TLB states is maintained after the step. All the other arguments are identical to Case 3.

Case 5: $h \overset{a}{\to} h'$ is an MMU step of setting A/D bits in a PTE pointed by a walk w. The proof for this case is completely analogous to Cases 2 and 4.

Case 6: $h \stackrel{a}{\to} h'$ is an MMU step of removing an arbitrary number of walks from the TLB. The C-IL configuration does not perform any steps. Consistency relation can not be broken and there is nothing to show.

Case 7: $h \stackrel{a}{\to} h'$ is a VMEXIT step on processor i. The C-IL configuration also performs a VMEXIT step producing configuration c' s.t.

$$\pi, \theta \vdash c \stackrel{hw}{\longrightarrow} c'.$$

The ASID of processor i in state h' equals 0. Hence, hardware consistency holds for h' and c' (it couples register values only for the case when $h.asid[i] = 0$). Other parts of the consistency relation follow from the fact that memory and TLB content is unchanged both in the hardware and software configurations.

Case 8: $h \stackrel{a}{\to} h'$ is a step of issuing a memory request $req \in MemReq$ to the memory automaton of processor i. The C-IL machine also perform the step of accepting a memory request:

$$\pi, \theta \vdash c \stackrel{b}{\to} c' \wedge b = hw(i, req.main),$$

which ensures that traces of a and b are equal. Consistency relation for the memory request buffer is maintained and other consistency relations can not be broken.

Case 9: $h \stackrel{a}{\to} h'$ is a step of sending a memory result $res \in MemRes$ from the memory automaton of processor i. The C-IL machine also perform the step of sending a memory result:

$$\pi, \theta \vdash c \stackrel{b}{\to} c' \wedge b = hw(i, res.main),$$

which ensures that traces of a and b are equal (the fact that the C-IL machine can perform such a step follows from the memory result buffer consistency between h and c). The consistency relation for the memory result buffer is maintained and other consistency relations can not be broken.

Case 10: $h \stackrel{a}{\to} h'$ is any other processor step. The C-IL configuration performs the same kind of a step and the proof is analogous to previous cases.

□

Another lemma guarantees that hardware consistency is maintained after a VMRUN step, if parameters of C-IL VMRUN statement are the same as parameters of the hardware *core-vmrun* step.

Lemma 7.6 (Consistent VMRUN). *Let consistency hold between $c \in conf_{CC+HW}$ and $h \in RedHardw$. Further, let hardware h perform a VMRUN step on processor $k \in Pid$ and configuration c perform a respective VMRUN step of*

thread k. Then hardware consistency also holds between h' and c'.

$$
\begin{aligned}
& h \xrightarrow{a} h' \\
& \wedge\ pid(a) = k \\
& \wedge\ hw\text{-}consis(h, c, k) \\
& \wedge\ h.asid[k].asid = 0 \\
& \wedge\ h'.asid[k].asid \neq 0 \\
& \wedge\ stmt_{next}(c(k), \pi) = \boldsymbol{vmrun}(e_0, e_1, e_2) \\
& \wedge\ [e_0]^{\pi,\theta}_{c(k)} = \boldsymbol{val}(bin_{64}(h.memreq[k].asidin), u64) \\
& \wedge\ [e_1]^{\pi,\theta}_{c(k)} = \boldsymbol{val}(bin_{64}(h.memreq[k].cr3in), u64) \\
& \wedge\ inject\text{-}data^{\pi,\theta}(c(k), e_2) = h.memreq[k].inject\text{-}data \\
& \wedge\ c.flush_{TLB} \implies h.memreq[k].complete\text{-}flush \\
& \wedge\ \pi, \theta \vdash c \xrightarrow{cil}_k c' \\
& \implies hw\text{-}consis(h', c', k)
\end{aligned}
$$

Proof. Follows from definitions. We omit it here due to its simplicity. □

7.4.2 C-IL + HW Simulation

Below we state a simulation theorem, analogous to the compiler correctness for regular C-IL (Theorem 5.2). This theorem guarantees, that on a machine where a C-IL (hypervisor) program is executed in parallel with the guest code, the result of the execution is consistent with the state of the C-IL + HW machine executing the same program.

Theorem 7.7 (C-IL + HW simulation). *Let $\pi \in prog_{C-IL}$ be a safe C-IL program with initial C-IL configuration $\hat{c}^0 \in conf_{CC-IL}$ and initial C-IL + HW configuration $c^0 \in conf_{CC+HW}$, where $c^0 = cil2chw_0(\hat{c}^0)$, and all threads in c^0 are at C-IL consistency points (this is the case when the location counter of every thread points to the first statement of a thread). Let $h^0 \in RedHardw$ be an initial safe state of the reduced hardware machine which is consistent with $\hat{c}^0$, and h^n be an arbitrary point in the execution sequence of the compiled program where $n > 0$. Let all processors in h^0 be in hypervisor mode. Then for all block schedules starting from h^0 and ending in h^n there exists a step function $s \in \mathbb{N} \mapsto \mathbb{N}$ and an execution sequence $c^0, c^1, c^2, \ldots, c^{s(n)}$ s.t. for all consistency points h^i consistency relation holds between states h^i and $c^{s(i)}$ for all running threads, execution from h^0 to h^n is safe, and the trace of the hardware component of this sequence is equivalent to the guest trace of the hardware sequence*

$h^0, h^1, \ldots, h^n$.

$$
\begin{aligned}
& h^0 \xrightarrow{\beta} h^n \\
& \wedge \forall k' \in Pid : cpoint_{C-IL}(c(k'), \pi) \\
& \wedge \forall k' \in Pid : h^0.asid[k'] = 0 \\
& \wedge \forall k' \in Pid : consis_{C-IL}(\hat{c}^0, \pi, h^0, k') \\
& \wedge cosched(\beta) \\
& \wedge c^0 = cil2chw_0(\hat{c}^0) \\
& \wedge safe\text{-}prog^{\pi,\theta}_{CC+HW}(c^0, o) \\
& \wedge safe\text{-}conf_r(h^0, o) \\
& \wedge \forall k' \in Pid : h^0.asid[k'] = 0 \wedge h^0.tlb[k'] = empty\text{-}tlb() \\
& \implies \exists s, o' : \forall i < n : cpoint(\beta, i) \implies \\
& \qquad (\forall k' \in Pid : running\text{-}thread_{k'}(\beta, i) \implies \\
& \qquad\qquad consis_{CC+HW}(h^i, \pi, c^{s(i)}, k')) \\
& \qquad \wedge safe\text{-}seq_r(\beta, o, o') \\
& \qquad \wedge (s(n) = 0 \implies guest\text{-}trace(\beta) = \{\}) \\
& \qquad \wedge (s(n) \neq 0 \implies \exists \omega, (c^0 \underset{\pi,\theta}{\xrightarrow{\omega}} c^{s(n)}) : guest\text{-}trace(\beta) \equiv hw\text{-}trace(\omega))
\end{aligned}
$$

Proof. The proof is done by induction on i. For the base case we have $cpoint(\beta, 0)$ from the definition of a hardware consistency point. From the preconditions of the theorem we get

$$\forall k' \in Pid : consis_{C-IL}(\hat{c}^0, \pi, h^0, k').$$

From $c^0 = cil2chw_0(\hat{c}^0)$ and the fact that all TLBs in configuration h^0 are empty we get

$$\forall k' \in Pid : consis_{CC+HW}(c^0, \pi, h^0, k').$$

Hence, we choose $s(0) = 0$.

For the induction step we assume that consistency for all running threads holds between a state h^i and $c^{s(i)}$, where $i < n$, state h^i is safe ($safe\text{-}conf_r(h^i, o)$), and C-IL + HW program safety holds starting from state $c^{s(i)}$:

$$safe\text{-}prog^{\pi,\theta}_{CC+HW}(c^{s(i)}, o).$$

Moreover, we assume that all threads in configuration $c^{s(i)}$ are at consistency points:

$$\forall k' \in Pid : cpoint_{C-IL}(\hat{c}^{s(i)}(k'), \pi).$$

Let m be the next consistency point in the hardware execution sequence:

$$m = next\text{-}cpoint(\beta, i).$$

If such point doesn't exists, then i is the last consistency point in the execution sequence and there is nothing to show. Further, let h^i be a consistency point of processor k: $cpoint_k(\beta, i)$. We have to show that (i) there exists some number $y \geq s(i)$ s.t.

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{CC+HW}(h^m, \pi, c^y, k')$$

holds, all threads in c^y are at C-IL consistency points, and

$$\pi, \partial \vdash c^{s(i)} \xrightarrow{\omega} c^y \wedge \textit{guest-trace}(\beta[i : m-1]) = \textit{hw-trace}(\omega)$$
$$\vee\ y = s(i) \wedge \textit{guest-trace}(\beta[i : m-1]) = \{\},$$

and (ii) there exists o' s.t. execution from h^i to h^m is safe and C-IL + HW program safety holds for c^y

$$\textit{safe-seq}_r(\beta[i : m-1], o, o') \wedge \textit{safe-prog}^{\pi,\partial}_{CC+HW}(c^y, o').$$

A safe hardware sequence also gives us safety of the final step in the sequence, which is a part of our induction hypothesis ($\textit{safe-conf}_r(h^m, o')$). We proceed with a case split on the type of the consistency point h^i:

Case 1: h^i is a hypervisor consistency point and step β_i is not a *core-vmrun* step and not a *core-tlb-invlpga* step. First, we extract C-IL configuration $\hat{c}$ from configuration $c^{s(i)}$:

$$\hat{c} = \textit{chw2cil}(c^{s(i)}),$$

and observe that regular C-IL consistency holds between $\hat{c}$ and h^i:

$$\forall k' \in \textit{Pid} : \textit{consis}_{C-IL}(\hat{c}, \pi, h^i, k').$$

Applying regular C-IL compiler correctness (Theorem 5.2) we find configuration $\hat{c}'$, where

$$\forall k' \in \textit{Pid} : \textit{running-thread}_{k'}(\beta, m) \implies \textit{consis}_{C-IL}(\hat{c}', \pi, h^m, k'),$$
$$\pi, \partial \vdash \hat{c} \to_k^* \hat{c}' \wedge \textit{cpoint}_{C-IL}(\hat{c}'(k), \pi).$$

It follows that thread k is at consistency point in $\hat{c}'$. All the other threads do not perform any steps in between. Hence, they also stay at consistency points (as assumed by our induction hypothesis) and we get

$$\forall k' \in \textit{Pid} : \textit{cpoint}_{C-IL}(\hat{c}'(k), \pi).$$

The guest trace of the hardware machine (i.e., the trace of the memory automaton in guest mode) from configuration h^i to h^m is empty. Hence, the hardware trace of the C-IL machine should also be empty. We further split cases on whether a C-IL machine performs any steps or not:

Case 1.1: the C-IL machine performs a number of steps from $\hat{c}$ to $\hat{c}'$:

$$\pi, \partial \vdash \hat{c} \to_k^+ \hat{c}'.$$

We apply Lemma 7.2 and get C-IL + HW configuration $c^y = \textit{cil2chw}(\hat{c}', p')$, where $\pi, \partial \vdash c^{s(i)} \xrightarrow{cil}_k^+ c^y$ and the hardware component is unchanged ($p' = c^{s(i)}.p$) or the complete TLB flush has been performed in between $c^{s(i)}$ and c^y. In the latter case we have

$$p' = c^{s(i)}.p[c^{s(i)}.p[k].tlb \mapsto \textit{empty-tlb}()] \wedge c^y.p.[k].\textit{flush}_{TLB}.$$

Since c^y is an MMU-extension of configuration $\hat{c}'$, we know

that regular consistency holds for c^y and all threads are at consistency points:

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{C-IL}(c^y, \pi, h^m, k'),$$
$$\forall k' \in Pid : cpoint_{C-IL}(c^y(k'), \pi).$$

From the fact that h^i is a hypervisor consistency point of processor k and β is a block schedule, we know that all hardware steps in between h^i and h^m are performed by processor k, which is running in hypervisor mode. Hence, TLBs between h^i and h^m do not add new walks (TLB of processor k may only remove walks). Moreover, since β_i is not a VMRUN, we know that ASIDs of all processors also stay unchanged between h^i and h^m and execution from $c^{s(i)}$ to c^y does not involve any VMRUN steps (this follows from Theorem 5.2 and definition of *cil2chw* function). The only change to the hardware component which could happen between $c^{s(i)}$ and c^y is removal of walks from the TLB by executing a **completeflush** step, which could not possibly break the hardware consistency relation (because $flush_{TLB}$ flag is always set in this case). Hence, applying induction hypothesis we get

$$\forall k' \in Pid : hw\text{-}consis(c^{s(i)}, \pi, h^i, k) \implies hw\text{-}consis(c^y, \pi, h^m, k').$$

From the definition of a running thread it follows that the set of running threads can not increase from h^i to h^m. Hence, we get

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{CC+HW}(h^m, \pi, c^y, k')$$

Using program safety $safe\text{-}prog^{\pi,\partial}_{CC+HW}(c^{s(i)}, o)$ (induction hypothesis) and applying Lemma 7.3 we find ownership setting o', s.t.

$$safe\text{-}local\text{-}seq^{\pi,\partial}_{C-IL}(\hat{c}(k), \hat{c}'(k), k, o, o') \wedge safe\text{-}prog^{\pi,\partial}_{CC+HW}(c^y, o').$$

Given safety of the local sequence in C-IL, we further apply Theorem 5.2 and get hypervisor safety of hardware steps between h^i and h^m:

$$safe\text{-}hyp\text{-}seq_r(\beta[i : m-1], o, o').$$

Further, from safety of hypervisor steps we get the safety of ownership transfer for every step in between h^i and h^m. Since all these steps are performed by processor k, ownership domains of other threads do not decrease. Hence, TLB safety for other processors is maintained in

every step. For TLB safety of processor k there is nothing to show, because it is running in hypervisor mode. That gives us

$$safe\text{-}seq_r(\beta[i:m-1], o, o').$$

From the fact that all steps between $c^{s(i)}$ and c^y are performed by a thread with ASID 0, we know that the hardware trace of this sequence is empty, which concludes the proof for this case.

Case 1.2: the C-IL machine does not perform any steps from $\hat{c}$ to $\hat{c}'$. This situation happens when one consistency point in C corresponds to multiple consistency points in hardware (e.g., when the first statement in a thread or the first statement after the return following a VMRUN involves an access to the shared memory). For this case configuration c^y is equal to c' and we don't change the ownership setting and take $o' = o$. The part of the consistency relation for the hardware component of a thread trivially folds. Hypervisor safety for hardware states between h^i and h^m, and the consistency relation for the software part of the C-IL + HW machine follows from Theorem 5.2. TLB safety follows from the fact that ownership setting o is not changing between those states and TLBs are not adding any new walks.

Case 2: h^i is a hypervisor consistency point and step β_i is a *core-vmrun* step. In this case we proceed in the same way as in Case 1, obtaining configuration $\hat{c} = chw2cil(c^{s(i)})$ and observing that $consis_{C-IL}(\hat{c}, \pi, h^i)$ holds. Analogously to Case 1 we apply Theorem 5.2 and find configuration $\hat{c}'$, where

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{C-IL}(\hat{c}', \pi, h^m, k'),$$
$$\pi, \theta \vdash \hat{c} \rightarrow_k \hat{c}' \wedge cpoint_{C-IL}(\hat{c}'(k), \pi).$$

From Theorem 5.2 we also know that the first statement to be executed by thread k in configuration $\hat{c}$ is a VMRUN statement with the same parameters as the hardware *core-vmrun* step has. Further, we apply Lemma 7.2 and get C-IL + HW configuration c^y s.t.

$$\begin{aligned}
c^y &= cil2chw(\hat{c}', c^{s(i)}.p[k \mapsto p'_k]),\\
p'_k &= (c^{s(i)}.p[k])[asid \mapsto h^i.memreq[k].asidin,\\
&\qquad CR3 \mapsto h^i.memreq[k].cr3in,\\
&\qquad tlb \mapsto (c^{s(i)}.flush_{TLB}\ ?\ tlb\text{-}empty()\ :\ c^{s(i)}.p[k].tlb)\\
&\qquad memreq = h^i.memreq[k].inject\text{-}data.req,\\
&\qquad memres.pf \mapsto h^i.memreq[k].inject\text{-}data.pf,\\
&\qquad memres.data \mapsto 0,\\
&\qquad memres.ready \mapsto h^i.memreq[k].inject\text{-}data.ready]).
\end{aligned}$$

Since c^y is an MMU-extension of configuration c', we know that regular consistency holds for c^y and all threads are at consistency

points:

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{C-IL}(c^y, \pi, h^m, k'),$$
$$\forall k' \in Pid : cpoint_{C-IL}(c^y(k'), \pi).$$

Next, we observe that in the consistency block schedule the next statement after VMRUN is either a step of the same processor in guest mode (i.e., first guest step after VMRUN) or a step of another processor, starting from a consistency point. Hence, from the definition of the set of consistency points the next state after VMRUN is also a consistency point and $m = i + 1$. Observing that transition from $c^{s(i)}$ to c^y involves only one C-IL step (hence, $y = s(i) + 1$) and applying Lemma 7.6, we get hardware consistency for thread k between h^m and c^y:

$$hw\text{-}consis(c^y, \pi, h^m, k).$$

Since the state of other processors is not changed, we know that the hardware consistency for these processors is maintained. This gives us

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{CC+HW}(h^m, \pi, c^y, k').$$

Further, from the program safety $safe\text{-}prog^{\pi,\partial}_{CC+HW}(c^{s(i)}, o)$ (induction hypothesis), and applying Lemma 7.3 we find ownership setting o', s.t.

$$safe\text{-}local\text{-}seq^{\pi,\partial}_{C-IL}(\hat{c}(k), \hat{c}'(k), k, o, o')$$
$$\wedge\ safe\text{-}prog^{\pi,\partial}_{CC+HW}(c^y, o').$$

Given safety of the local sequence in C-IL, we further apply Theorem 5.2 and get hypervisor safety of the hardware step between h^i and h^{i+1}:

$$safe\text{-}hyp\text{-}seq_r(\beta[i], o, o').$$

Ownership domains of all threads do not decrease on transition from o to o'. Hence, TLB safety of processors other than k is maintained. TLB safety for processor k follows from $safe\text{-}prog^{\pi,\partial}_{CC+HW}(c^y, o')$ and the fact that hardware consistency for processor k holds between states c^y and h^m. This gives us

$$safe\text{-}seq_r(\beta[i : m-1], o, o').$$

Further, we observe that both the guest trace of the hardware machine and the trace of the hardware component of the C-IL machine are empty, because both the hardware and the software VMRUN do not contribute to these traces, and conclude the proof for this case.

Case 3: h^i is a hypervisor consistency point and step β_i is a *core-tlb-invlpga* step. From Theorem 5.2 it follows that the next step to be executed in $c^{s(i)}$ is **invlpga**(e_0, e_1) statement with the ASID parameter being equal

to the one of the hardware step:

$$[e_0]^{\pi,\theta}_{c(k)} = \mathbf{val}(bin_{64}(h^i.memreq[k].va), u64)$$
$$[e_1]^{\pi,\theta}_{c(k)} = \mathbf{val}(bin_{64}(h^i.memreq[k].asid), u64).$$

From the semantics of hardware and software INVLPG steps it follows that hardware consistency is maintained after these steps are performed in both machines:

$$hw\text{-}consis(c^{s(i)+1}, \pi, h^{i+1}, k).$$

From the definition of a block schedule and Theorem 5.2 it follows that no new walks are added to the hardware TLB until the next consistency point, and the state of the software TLB remains unchanged until then. Hence, we get

$$hw\text{-}consis(c^y, \pi, h^m, k),$$

where configuration $c^{s(y)}$ is constructed using Theorem 5.2 and Lemma 7.2 (analogously to the previous case). Since the state of other processors is not changed, we know that the hardware consistency for these processors is maintained. This gives us

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{CC+HW}(h^m, \pi, c^y, k').$$

Arguments for the safety of a hardware execution between h^i and h^m, as well as for the safety of the program starting from state c^y, are identical to those used in Case 1.

Case 4: h^i is a guest consistency point. From the definition of consistency-block schedule and our choice of consistency points it follows that $h^m = h^{i+1}$ and β_i is a guest step (possibly being a VMEXIT step). We apply Lemmas 7.5 and 7.4 and obtain configuration c^y, where $y >= s(i)$ and consistency holds between states h^m and c^y:

$$\forall k' \in Pid : running\text{-}thread_{k'}(\beta, m) \implies consis_{CC+HW}(h^m, \pi, c^y, k'),$$

sequence of hardware steps between h^i and h^m is safe

$$safe\text{-}seq_r(\beta[i : m-1], o, o),$$

program safety is maintained for c^y

$$safe\text{-}prog^{\pi,\theta}_{CC+HW}(c^y, o),$$

and traces of hardware and software execution sequences are equivalent. Location counters of all threads are not changed from $c^{s(i)}$ to c^y, which means that all threads remain at consistency points in c^y.

□

Note, that in Theorem 7.7 we show that compiler consistency holds for all hardware consistency points. Yet, with a simple extension to the theorem one can show that compiler consistency also holds for all software C-IL consistency points. To do that, one has to keep track of all software consistency points

and to show that we always advance to the next consistency point during the induction step. To show this, one has to strengthen Theorem 5.2 to require configuration c' to be the next software consistency point of a given thread (so that we don't skip any C-IL consistency points in between c and c').

7.5 C-IL + HW + Ghost Semantics

Putting together the C-IL + Ghost semantics from Chapter 6 and the C-IL + HW semantics introduced in this chapter we obtain the C-IL + HW + Ghost semantics.

Configuration of sequential C-IL + HW + Ghost is obtained by extending the configuration from Definition 6.13 with the hardware component:

$$conf_{C+HW+G} \stackrel{\text{def}}{=} [\mathcal{M} \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, \mathcal{M}_G \in val_{ptr} \cup \mathbb{N} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}, stack \in frame^*_{C+G}, flush_{TLB} \in \mathbb{B}, p \in core_c, next\text{-}free_G \in \mathbb{N}].$$

◀ **Definition 7.37** C-IL + HW + Ghost configuration

Concurrent C-IL + HW + Ghost configurations are constructed respectively:

$$conf_{CC+HW+G} \stackrel{\text{def}}{=} [confM \in \mathbb{B}_{gm} \mapsto \mathbb{B}^8, \mathcal{M}_G \in val_{ptr} \cup \mathbb{N} \cup \mathbb{V} \mapsto val_{\mathcal{M}_G}, stack \in Tid \mapsto frame^*_{C+G}, flush_{TLB} \in Tid \mapsto \mathbb{B}, next\text{-}free_G \in Tid \mapsto \mathbb{N} p \in Tid \mapsto core_c].$$

◀ **Definition 7.38** Concurrent configuration (C-IL + HW + Ghost)

A software step of the C-IL + HW + Ghost semantics now involves execution of a number of ghost statements (if there are any) followed by a single implementation statement.

$$\frac{\pi, \theta \vdash c \rightarrow^*_G c'' \qquad \pi, \theta \vdash c'' \rightarrow_I c'}{\pi, \theta \vdash c \xrightarrow{cil} c'}$$

◀ **Definition 7.39** Software step of C-IL + HW + Ghost

A step of the sequential C-IL + HW + Ghost semantics involves either execution of a software step or execution of a single step of the hardware component.

$$\frac{\pi, \theta \vdash c \xrightarrow{cil} c' \vee \pi, \theta \vdash c \xrightarrow{hw} c' \vee \pi, \theta \vdash c \xrightarrow{hw(req)} c' \vee \pi, \theta \vdash c \xrightarrow{hw(res)} c'}{\pi, \theta \vdash c \rightarrow c'}$$

◀ **Definition 7.40** C-IL + HW + Ghost step

A step of the concurrent C-IL + HW + Ghost semantics is a step of some thread operating on the shared memory, shared ghost memory, on its local stack, and on its local MMU component.

$$\frac{\begin{array}{c}\pi, \theta \vdash c(t) \rightarrow (\mathcal{M}', \mathcal{M}'_G, stack', flush'_{TLB}, p') \\ c' = (\mathcal{M}', \mathcal{M}'_G, c.stack[t \mapsto stack', c.flush_{TLB}[t \mapsto flush'_{TLB}]], c.p[t \mapsto p'])\end{array}}{\pi, \theta \vdash c \rightarrow c'}$$

◀ **Definition 7.41** Step of concurrent C-IL + HW + Ghost

We introduce two functions, which have as an input a C-IL + HW + Ghost configuration and return a corresponding C-IL + HW or a C-IL + Ghost

configuration by throwing away part of the state:

Definition 7.42 ► C-IL + HW + Ghost conversion

$$chwg2chw(c \in conf_{C+HW+G}, \pi \in prog_{C+G}) \in conf_{C+HW},$$
$$chwg2cg(c \in conf_{C+HW+G}) \in conf_{C+G},$$
$$chwg2chw(c, \pi).\{\mathcal{M}, p, flush_{TLB}\} \stackrel{\text{def}}{=} c.\{\mathcal{M}, p, flush_{TLB}\},$$
$$chwg2chw(c, \pi).stack \stackrel{\text{def}}{=} cg2cil\text{-}stack(c.stack, \pi),$$
$$chwg2cg(c) \stackrel{\text{def}}{=} conf_{C+G}[\mathcal{M} \mapsto c.\mathcal{M}, \mathcal{M}_G \mapsto c.\mathcal{M}_G, stack \mapsto c.stack$$
$$flush_{TLB} \mapsto c.flush_{TLB}, next\text{-}free_G \mapsto c.next\text{-}free_G].$$

Analogously, we define the same functions for a concurrent C-IL + HW + Ghost configuration:

Definition 7.43 ► Concurrent C-IL + HW + Ghost conversion

$$chwg2chw(c \in conf_{CC+HW+G}, \pi \in prog_{C+G}) \in conf_{CC+HW},$$
$$chwg2cg(c \in conf_{CC+HW+G}) \in conf_{CC+G},$$
$$chwg2chw(c, \pi).\{\mathcal{M}, p, flush_{TLB}\} \stackrel{\text{def}}{=} c.\{\mathcal{M}, p, flush_{TLB}\},$$
$$chwg2chw(c, \pi).stack[t] \stackrel{\text{def}}{=} cg2cil\text{-}stack(c.stack[t], \pi),$$
$$chwg2cg(c) \stackrel{\text{def}}{=} conf_{CC+G}[\mathcal{M} \mapsto c.\mathcal{M}, \mathcal{M}_G \mapsto c.\mathcal{M}_G,$$
$$stack \mapsto c.stack, flush_{TLB} \mapsto c.flush_{TLB},$$
$$next\text{-}free_G \mapsto c.next\text{-}free_G].$$

Further in this section we have two goals: (i) show (forward) simulation between the C-IL + HW semantics and between the C-IL + HW + Ghost semantics and (ii) transfer program safety from the C-IL + HW + Ghost level to the C-IL + HW level (so that we could satisfy preconditions of Theorem 7.7 afterwards).

To achieve these goals we first define program safety for a C-IL + HW + Ghost machine.

A state of the C-IL + HW + Ghost machine is safe if it satisfies both the safety from C-IL + HW and the safety of the ghost code. Moreover, in the definition of the safety of a given state of C-IL + HW + Ghost machine, we consider safety of the next non-ghost statement to be executed as well as the safety of all ghost statements preceding the next implementation statement:

Definition 7.44 ► Safe configuration (C-IL + HW + Ghost)

$$safe\text{-}conf^{\pi,\partial}_{C+HW+G}(c \in conf_{C+HW+G}, o \in Ownership, k \in Tid) \in \mathbb{B}$$
$$safe\text{-}conf^{\pi,\partial}_{C+HW+G}(c, o, k) \stackrel{\text{def}}{=} safe\text{-}tlbs_c(c.p, o, k)$$
$$\wedge\ ghost\text{-}safe\text{-}stmt^{\pi,\partial}_{C+G}(chwg2cg(c, \pi))$$
$$\wedge\ (stmt_{next}(c, \pi) \notin \mathbb{S}_G \implies safe\text{-}stmt^{\pi,\partial}(stmt_{next}(c, \pi), c, o, k))$$
$$\wedge\ (stmt_{next}(c, \pi) \in \mathbb{S}_G \wedge \pi, \partial \vdash c \to c' \implies safe\text{-}conf^{\pi,\partial}_{C+HW+G}(c', o, k)).$$

Annotated program $\pi \in prog_{C+G}$ with initial configuration $c \in conf_{CC+HW+G}$ is safe if every possible state of the execution of π and the ownership transfer

are safe:

Definition 7.45 Safe program (C-IL + HW + Ghost)

$$
\begin{aligned}
&\mathit{safe\text{-}prog}^{\pi,\theta}_{CC+HW+G}(c \in \mathit{conf}_{CC+HW+G}, o \in \mathit{Ownership}) \in \mathbb{B}\\
&\mathit{safe\text{-}prog}^{\pi,\theta}_{CC+HW+G}(c, o) \overset{\text{def}}{=} \forall i \in \mathit{Tid} : \mathit{safe\text{-}conf}^{\pi,\theta}_{C+HW+G}(c(i), o, i)\\
&\quad \wedge \forall c' : \pi, \theta \vdash c \xrightarrow{cil}_k c' \implies \exists o' : \mathit{safe\text{-}prog}^{\pi,\theta}_{CC+HW+G}(c', o')\\
&\qquad \wedge \mathit{safe\text{-}transfer}^{\pi,\theta}(\mathit{chwg2chw}(c(k), \pi), \mathit{chwg2cg}(c'(k), \pi), k, o, o')\\
&\quad \wedge \forall c' : \pi, \theta \vdash c \xrightarrow{a} c' \wedge \mathit{hw\text{-}step}(a) \implies \mathit{safe\text{-}prog}^{\pi,\theta}_{CC+HW+G}(c', o).
\end{aligned}
$$

Note, that the ownership transfer is allowed to occur only when the machine performs an implementation software step. Note also, that the function *chwg2chw* converts a C + HW + Ghost configuration to a respective C + HW configuration, which also involves setting the location counter in the stack to a next non-ghost statement.

Before we proceed to the simulation theorem for the C-IL + HW + Ghost machine, we first prove a lemma which derives safety of a C-IL + HW step from the safety of the annotated program inside the C-IL + HW + Ghost semantics.

Lemma 7.8 (Safe C-IL + HW + Ghost step). *Let $\pi_G \in \mathit{prog}_{C+G}$ be a safe annotated program and $c \in \mathit{conf}_{CC+HW+G}$ be a concurrent C-IL + HW + Ghost configuration. Further, let $\pi \in \mathit{prog}_{C-IL}$ and $\hat{c} \in \mathit{conf}_{CC+HW}$ be the corresponding program and configuration of C-IL + HW. Then every possible step of the C-IL + HW machine is safe:*

$$
\begin{aligned}
&\mathit{safe\text{-}prog}^{\pi_G,\theta}_{CC+HW+G}(c, o)\\
&\wedge \hat{c} = \mathit{chwg2chw}(c, \pi_G)\\
&\wedge \pi = \mathit{cg2cil\text{-}prog}(\pi_G)\\
&\wedge \pi, \theta \vdash \hat{c} \xrightarrow{a} \hat{c}'\\
&\implies \mathit{safe\text{-}seq}^{\pi,\theta}_{CC+HW}(a, o).
\end{aligned}
$$

Proof. Unfolding definition of $\mathit{safe\text{-}seq}_{CC+HW}$ we get the properties we have to show:

$$
\begin{aligned}
&\forall t \in \mathit{Tid} : \mathit{safe\text{-}conf}^{\pi,\theta}_{C+HW}(\hat{c}(t), o, t),\\
&\forall c' : \pi, \theta \vdash \hat{c} \xrightarrow{a} \hat{c}' \wedge a = \mathit{cil}(k) \implies\\
&\qquad \exists o' : \mathit{safe\text{-}transfer}^{\pi,\theta}(\hat{c}(k), \hat{c}'(k), k, o, o')\\
&\qquad\qquad \wedge \forall t \in \mathit{Tid} : \mathit{safe\text{-}conf}^{\pi,\theta}_{C+HW}(\hat{c}'(t), o', t),\\
&\forall c' : \pi, \theta \vdash \hat{c} \xrightarrow{a} \hat{c}' \wedge \mathit{hw\text{-}step}(a) \implies\\
&\qquad \forall t \in \mathit{Tid} : \mathit{safe\text{-}conf}^{\pi,\theta}_{C+HW}(\hat{c}'(t), o, t).
\end{aligned}
$$

First, we unfold definitions $\mathit{safe\text{-}prog}_{CC+HW+G}$ and $\mathit{safe\text{-}conf}_{C+HW+G}$ and get the safety of every non-ghost statement to be executed next in every thread and the safety of a hardware component in state c. From the definition of the *chwg2chw* function is follows that the next statement to be executed in every thread in $\hat{c}$ is the same one as the next non-ghost statement in c. Moreover, hardware components in c and $\hat{c}$ are the same. This gives us

$$\forall t \in \mathit{Tid} : \mathit{safe\text{-}conf}^{\pi,\theta}_{C+HW}(\hat{c}(t), o, t).$$

Further, we perform a case split on the type of a step performed by the C-IL + HW machine:

Case 1: a step from $\hat{c}$ to $\hat{c}'$ is a step of the hardware component of thread k: From *safe-prog*$_{CC+HW+G}$ we know that safety is maintained after every step of the hardware component:

$$\forall c' : \pi, \theta \vdash c \xrightarrow{a} c' \wedge hw\text{-}step(a) \implies safe\text{-}prog^{\pi_G,\theta}_{CC+HW+G}(c', o).$$

Since memory and hardware components of c and $\hat{c}$ are equal, both machines can perform the same hardware steps resulting in configurations c' and $\hat{c}'$, such that $\hat{c}' = chwg2chw(c', \pi_G)$. Hence, we can now use $safe\text{-}prog^{\pi_G,\theta}_{CC+HW+G}(c', o)$ to get

$$\forall t \in Tid : safe\text{-}conf^{\pi,\theta}_{C+HW}(\hat{c}'(t), o, t).$$

Case 2: a step from $\hat{c}$ to $\hat{c}'$ is a software step of thread k. Applying Theorem 6.1 we can find a respective sequence of steps of the C-IL + HW + Ghost machine, such that it executes a number of ghost steps of thread k and one implementation step of k, and results in configuration c', such that $\hat{c}' = chwg2chw(c', \pi_G)$. From *safe-prog*$_{CC+HW+G}$ we get ownership setting o' s.t.

$$\begin{aligned}&safe\text{-}prog^{\pi_G,\theta}_{CC+HW+G}(c', o')\\ &\wedge safe\text{-}transfer^{\pi_G,\theta}(chwg2chw(c(k), \pi_G), chwg2cg(c'(k), \pi_G), k, o, o').\end{aligned}$$

Further, we observe that $\hat{c}(k) = chwg2chw(c(k), \pi_G)$ and $\hat{c}'(k) = chwg2cg(c'(k), \pi_G)$. This gives us

$$safe\text{-}transfer^{\pi,\theta}(\hat{c}(k), \hat{c}'(k), k, o, o').$$

Unfolding $safe\text{-}prog_{CC+HW+G}(c', o')$ we get the safety of every statement to be executed next in every thread and the safety of the hardware component in c'. From the definition of function *chwg2chw* it follows that the next statement to be executed in every thread in $\hat{c}'$ is the same one as the next non-ghost statement in c', and hardware components in c' and $\hat{c}'$ are equal. This concludes the proof.

□

Now we can prove a simulation theorem between a C-IL + HW + Ghost machine and a C-IL + HW machine.

Theorem 7.9 (C-IL + HW + Ghost simulation). *Let $\pi_G \in prog_{C+G}$ be a safe annotated program and $c \in conf_{CC+HW+G}$ be a concurrent C-IL + HW + Ghost configuration. Further, let $\pi \in prog_{C-IL}$ and $\hat{c} \in conf_{CC+HW}$ be the corresponding program and configuration of C-IL + HW. Then for every sequence of steps of the C-IL + HW machine, there exists a sequence of steps of the C-IL + HW + Ghost machine, such that resulting configurations correspond w.r.t the chwg2chw function and hardware traces of both execution sequences are the*

same. Moreover, execution sequence of the C-IL + HW machine is safe.

$$
\begin{aligned}
&\forall \beta, (\hat{c} \xrightarrow[\pi,\partial]{\beta} \hat{c}') : \\
&\quad \textit{safe-prog}^{\pi_G,\partial}_{CC+HW+G}(c, o) \\
&\quad \wedge\ \hat{c} = \textit{chwg2chw}(c, \pi_G) \\
&\quad \wedge\ \pi = \textit{cg2cil-prog}(\pi_G) \\
&\quad \implies \exists c', \omega, (c \xrightarrow[\pi_G,\partial]{\omega} c') : \hat{c}' = \textit{chwg2chw}(c', \pi_G) \\
&\qquad\qquad\qquad\qquad \wedge\ \textit{safe-seq}^{\pi,\partial}_{CC+HW}(\beta, o) \\
&\qquad\qquad\qquad\qquad \wedge\ \textit{hw-trace}(\beta) = \textit{hw-trace}(\omega)
\end{aligned}
$$

Proof. By induction on steps of the C-IL + HW machine and a case split on the type of a step.

Case 1: if machine $\hat{c}$ performs a step of the hardware component, then machine c performs the same kind of a step, maintaining *chwg2chw* abstraction and equality of traces.

Case 2: if machine $\hat{c}$ performs a software step of thread k, then we apply Theorem 6.1, and machine c performs a sequence of steps of k (which consists of a number of ghost steps and a single implementation step) resulting in configuration c', s.t.

$$\hat{c'} = \textit{chwg2chw}(c', \pi_G).$$

The trace property is maintained since all software steps are considered to be internal and do not affect the hardware trace in any way.

For every step we obtain the safety of the C-IL + HW execution sequence with the help of Lemma 7.8 and the inductive nature of $\textit{safe-prog}_{CC+HW+G}$ (i.e., for every c' reachable from c there exists o' such that the program safety is maintained). □

With the help of Theorem 7.9 we can derive safety of a C-IL program in the C-IL + HW semantics from the safety of a C-IL + Ghost program inside the C-IL + HW + Ghost semantics. We need this property to discharge the respective precondition of Theorem 7.7.

Lemma 7.10 (Safety of C-IL + HW program.). *Let $\pi_G \in prog_{C+G}$ be an annotated program and $c \in conf_{CC+HW+G}$ be a concurrent C-IL + HW + Ghost configuration. Further, let $\pi \in prog_{C-IL}$ and $\hat{c} \in conf_{CC+HW}$ be the corresponding program and configuration of C-IL + HW. Let the annotated program π_G be safe w.r.t state c and ownership setting $o \in Ownership$. Then the program π is also safe w.r.t state $\hat{c}$ and ownership setting o:*

$$
\begin{aligned}
&\textit{safe-prog}^{\pi_G,\partial}_{CC+HW+G}(c, o) \\
&\wedge\ \hat{c} = \textit{chwg2chw}(c, \pi_G) \\
&\wedge\ \pi = \textit{cg2cil-prog}(\pi_G) \\
&\implies \textit{safe-prog}^{\pi,\partial}_{CC+HW}(\hat{c}, o).
\end{aligned}
$$

Proof. Follows directly from Theorem 7.9. □

CHAPTER 8

TLB Virtualization

8.1 Specification and Implementation Models

8.2 VM Configuration

8.3 Coupling Invariant

8.4 Simulation

8.5 Emulating Machine With Caches

The purpose of a hypervisor is to provide to several virtual machines (VMs), each running its own operating system, an illusion that every VM is running alone on a physical machine, even though different machines might try to configure their page tables to use the same physical addresses. To provide this illusion, the hypervisor provides an additional level of address translation. It does this either with the help of the hardware support (if the hardware supports nested paging) or by maintaining a separate set of page tables, called Shadow Page Tables (SPTs), for each VM. These SPTs are the tables actually used by the hardware for address translation, but are kept invisible to the VMs. The SPT algorithm guarantees, that the virtual TLB, provided to the guest by the hardware TLB together with the intercept handlers, behaves according to the hardware specification and provides appropriate translations to the guest. In this chapter we provide the specification model for the VMs and give the correctness criteria for TLB virtualization. Then we define the coupling invariant for the VMs, including the VTLB, and prove correctness of simulation for hardware steps performed in virtualization mode.

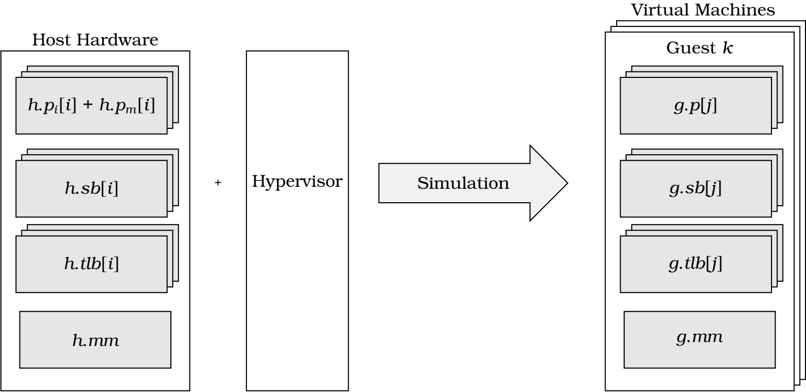

Figure 8.1: Hypervisor virtualization correctness.

Correcntess of hypervisor virtualization is normally expressed via a simulation proof between the host hardware machine executing a hypervisor program and a guest virtual machine abstracted from the hypervisor/host hardware configuration [AP08]. In this thesis we don't aim at the full hypervisor correctness, but only show correct virtualization of guest memory accesses (including TLB operations).

We do this by showing forward simulation (Section 2.3) between memory automata (Section 3.5) of the host hardware machine and an abstraction of the memory automata of guest VMs (Figure 8.1). As a result, we show that for any sequence of host hardware steps there exists a respective sequence of steps of guest virtual machines, such that the guest memory trace[1] of the host hardware equals to the memory trace of the VMs[2].

Our main virtualization correctness property is stated between the host hardware machine (executing a hypervisor) and an abstract guest configuration. Nevertheless, we want to use a C program verifier for performing all proofs which involve arguing about the hypervisor code. Hence, our correctness proof consists of two parts:

1. we define a coupling invariant between the hypervisor configuration in C-IL + HW + Ghost semantics and the abstract guest VMs. We show that for any sequence of steps of the C-IL machine there exists a valid sequence of steps of the abstract guest VMs such that the coupling invariant is maintained afterwards and traces of executions are equal (i.e., we show forward simulation between the C-IL machine executing hypervisor code and the abstract guest VMs) (Figure 8.2);
2. we observe that the compiler consistency theorem (Theorem 7.7) together with the C-IL + HW + Ghost simulation theorem (Theorem 7.9) guarantees that for every hardware execution sequence of a host hardware machine there exists an execution sequence of a C-IL + HW +

[1] A guest memory trace is a sequence of inputs/outputs of memory automata running in guest mode (see Section 7.2.3).

[2] In this and the following chapters the notion of a "guest" and a "VM" can be considered equivalent. Nevertheless, we mostly use "VM" when talking about an abstract machine provided by the hypervisor, and "guest" when talking about the user code executed in this machine.

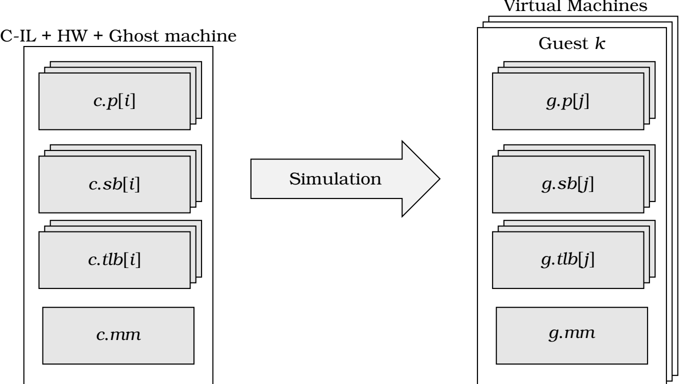

Figure 8.2: Hypervisor virtualization correctness on C-IL + HW + Ghost level.

Ghost machine, such that consistency holds between consistency points and guest traces of both machines are equal. Using (1), we transfer our simulation property down to the hardware level, showing equality of memory traces of the host hardware and the guest VMs.

In this and the following chapter we aim at proving the first part of the correctness theorem. The second part is obtained by a simple combination of two theorems and we omit stating it explicitly here.

8.1 Specification and Implementation Models

8.1.1 Host Hardware Model

The host hardware machine h is modelled as an instance of the reduced hardware machine *RedHardw* (Definition 4.2), where caches are completely invisible, while SBs and TLBs are visible only on the processors running in guest mode (Figure 8.3).

A hypervisor configuration running atop of the host hardware machine is modelled via an instance of the C-IL + HW + Ghost machine, referred simply as a “C-IL machine” later in this chapter.

Throughout this chapter we have to argue about the values of certain global variables of a hypervisor program in a given C-IL configuration. For this reason we introduce a number of abstraction functions, which extract these values. We leave these functions undefined in this chapter and instantiate them in the next chapter, where we consider a particular implementation of the SPT algorithm. Yet, when proving Theorem 8.3 we have to know that values of these abstractions are left unchanged when the C-IL machine performs a step of the hardware component. Hence, we assume here that all these abstractions are located in the hypervisor memory (i.e., not in the memory allocated to the guest) and that they do not alias with each other.

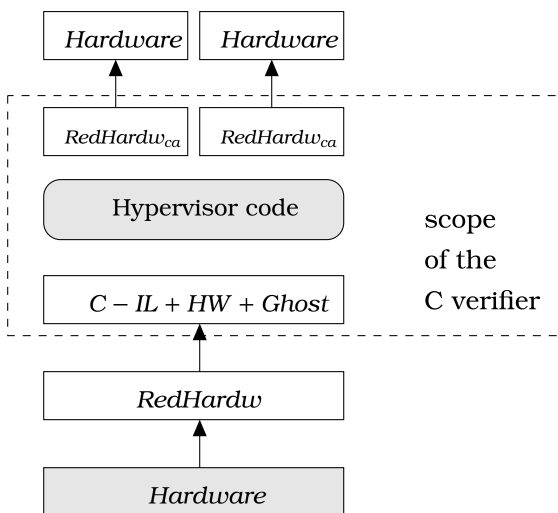

Figure 8.3: Semantics stack for hypervisor verification.

8.1.2 Guest Virtual Machines

For guest virtual machines emulated by the hypervisor we model only the part responsible for memory accesses (i.e., the memory-controlling part of the core together with the physical memory, TLBs, and store buffers). We do not include caches to the guest configuration. Since we do not model devices and consider memory accesses to have no side-affect, one can later prove an easy theorem showing simulation between a cache-reduced VM model and a full hardware model introduced in Chapter 3 (a machine without caches is simulated by a machine with caches).

The number of VMs emulated by the hypervisor is expressed with the set of guest IDs *Gid*. We assume that every guest machine has the same number of (virtual) processors as the host machine has (identified by the set *Pid*).

The state of VMs is modelled as a map from a guest ID to an instance of the reduced hardware configuration of the memory automaton:

$$VmHardw \stackrel{\text{def}}{=} Gid \mapsto RedMemHardw.$$

Since we don't model the instruction part of the core, steps *core-issue-mem-req* and *core-send-mem-res* are considered to be input and output actions respectively.

Transitions of a single guest VM form a subset of transitions of the cache-reduced machines (Section 4.2) under the following restrictions:

- the ASID register is not used in transitions; for guest machines it is considered to be always equal 0,
- there are no VMRUN and VMEXIT steps; if the memory core of a guest processor gets a VMEXIT or a VMRUN request then it will set bit *memreq.active* to 0 and the request will never get served (i.e., getting a VMRUN or a VMEXIT request is equivalent to getting an inactive request),
- step *core-issue-mem-req* takes as an input an instance of type

MemReqMain rather than *MemReq*,

- step *core-send-mem-req* provides as an output an instance of type *MemResMain* rather than *MemRes*,
- step *core-invlpga* performs invalidation in the current ASID (i.e., in ASID 0), which makes TLB tags invisible for the guest.

Every step of machine $g \in VmHardw$ is additionally parametrised with the ID of the VM performing a step, e.g.,

$$\begin{aligned} &\textit{core-issue-mem-req}(i \in Gid, j \in Pid, req \in MemReqMain), \\ &\textit{core-send-mem-res}(i \in Gid, j \in Pid, res \in MemResMain). \end{aligned}$$

If VM i is making a step, then all components of all other VMs remain unchanged.

To denote that transition a from state g to g' is a part of the transition relation of VMs we write $g \xrightarrow{a} g'$.

Given states g and g', the expression $g \xrightarrow{\beta} g'$, where $|\beta| = n$ and $n > 0$, denotes execution sequence $g^0, \beta_0, g^1, \beta_1, \ldots, \beta_n, g^n$, where $g^0 = g$, $g^n = g'$ and every next hardware state is obtained from the previous one by performing the corresponding step from β:

$$\forall i < n : g^i \xrightarrow{\beta_i} g^{i+1}.$$

Analogously to the function $pid(a)$ we introduce the function $gid(a)$, which extracts the ID of the VM which is performing step a:

$$gid(a) = i \overset{\text{def}}{=} (a \text{ is a step of VM } i).$$

◄ **Definition 8.1**
Step of VM i

8.1.3 Equality of Traces

A memory trace of VM execution sequence $g \xrightarrow{\beta} g'$ is obtained by extracting labels of all external actions of VM's memory automata:

$$\textit{vm-trace}(\beta) \overset{\text{def}}{=} \begin{cases} \beta_0 & \beta_0 \in ext(VmHardw) \land |\beta| = 1 \\ \beta_0 \circ \textit{vm-trace}(\mathbf{tl}(\beta)) & \beta_0 \in ext(VmHardw) \land |\beta| \neq 1 \\ \textit{vm-trace}(\mathbf{tl}(\beta)) & \text{otherwise} \end{cases}$$

◄ **Definition 8.2**
VM memory trace

The set $ext(VmHardw)$ contains the labels of all possible external actions of *VmHardw*.

$$\begin{aligned} &ext(VmHardw) \overset{\text{def}}{=} \\ &\quad \{\textit{core-issue-mem-req}(i, j, req), \textit{core-send-mem-res}(i, j, res) \mid \\ &\quad i \in Gid, j \in Pid, req \in MemReqMain, res \in MemResMain\}. \end{aligned}$$

In order to be able to identify a particular virtual machine and a virtual processor which is currently being executed on a given host processor we

introduce function

$$hp2vp_c(i \in Pid) \in (Gid, Pid),$$

which we leave undefined for now and define it in Section 8.2.2 using hypervisor configuration $c \in conf_{CC+HW+G}$.

We extend the C-IL hardware trace definition from Section 7.2.3 to collect IDs of virtual processors which are being executed at the time when external actions occur. Let $c^0 \overset{\beta}{\underset{\pi,\theta}{\rightarrow}} c^n$ be an execution sequence of the C-IL machine. Then we define the extended C-IL +HW I/O trace in the following way:

Definition 8.3 ► C-IL + HW I/O Trace (extended with VP IDs)

$$hw\text{-}id\text{-}trace(\beta) \overset{\text{def}}{=} \begin{cases} (\beta_0, hp2vp_{c^0}(pid(\beta_0))) & \beta_0 \in ext(\text{C-IL+HW}) \wedge |\beta| = 1 \\ (\beta_0, hp2vp_{c^0}(pid(\beta_0)) \circ hw\text{-}id\text{-}trace(\mathbf{tl}(\beta))) & \beta_0 \in ext(\text{C-IL+HW}) \wedge |\beta| > 1 \\ hw\text{-}id\text{-}trace(\mathbf{tl}(\beta)) & \text{otherwise} \end{cases}$$

Now we can specify our correctness criteria for virtualization of memory accesses between a given C-IL execution trace and a respective execution trace of the guest VMs.

Given an execution sequence of the C-IL machine $c \overset{\beta}{\underset{\pi,\theta}{\rightarrow}} c'$ and an execution sequence of the guest VMs $g \overset{\omega}{\rightarrow} g'$ we say that guest memory traces of these sequences are equal if the following property holds:

Definition 8.4 ► Equal VM memory traces

$$\begin{aligned} traces\text{-}eq(\beta, \omega) \overset{\text{def}}{=} & |hw\text{-}id\text{-}trace(\beta)| = |vm\text{-}trace(\omega)| \wedge \forall i < |vm\text{-}trace(\omega)| : \\ & hw\text{-}id\text{-}trace(\beta)[i] = (hw(l, req), (j, k)) \\ & \implies vm\text{-}trace(\omega)[i] = core\text{-}issue\text{-}mem\text{-}req(j, k, req) \\ & \wedge\ hw\text{-}id\text{-}trace(\beta)[i] = (hw(l, res), (j, k)) \\ & \implies vm\text{-}trace(\omega)[i] = core\text{-}send\text{-}mem\text{-}res(j, k, res). \end{aligned}$$

8.1.4 VM Simulation

We state correct virtualization of memory actions in the form of the following theorem.

Theorem 8.1 **(Correct virtualization).** *Let $c \in conf_{CCC+HW+G}$ be the initial hypervisor configuration and $g \in VmHardw$ be the initial configuration of guest VMs. Then for any sequence of C-IL steps starting from c there exists a sequence of VM steps starting from g, such that traces of C-IL and VM executions are equal.*

$$\forall \beta, (c \overset{\beta}{\underset{\pi,\theta}{\rightarrow}} c') : (\exists \omega, (g \overset{\omega}{\rightarrow} g') : traces\text{-}eq(\beta, \omega)) \vee hw\text{-}trace(\beta) = \{\})$$

To prove Theorem 8.1 we do the following:

- define the coupling invariant between C-IL states and states of the abstract VMs,
- verify hypervisor initialization phase, which ends in a state c'', s.t. it is reachable from c, the coupling invariant holds between c'' and g, and the

guest trace from c to c'' is empty. In this thesis we do not argue about initialization of data structures and consider a starting thread where the data structures are already initialized and the coupling invariant initially holds[3],

- show that for every C step there exists a valid sequence of guest steps, s.t. the coupling invariant is preserved and traces of this step and of the hardware steps are equal. This further includes two cases:
  1. if a step is done by a thread running in hypervisor mode and the coupling invariant holds before the step, then the coupling invariant also holds after the step and the trace of the respective sequence of guest steps is empty. The sketch of the proof of this property for intercept handlers of the SPT algorithm is given in the next chapter and the theorem which argues about the correctness of VMRUN is stated in Section 8.4.2,
  2. if a step is performed by the hardware component of a thread and the coupling invariant holds before the step, then the coupling invariant also holds after the step and the trace of the respective sequence of guest steps is equal to the trace of the C-IL step. The proof of this property is done in Theorem 8.3.

The definition of the coupling invariant $\mathit{inv\text{-}coupling}(c, g)$ is given in Section 8.3. In Section 8.4.1 we prove the property stated above for hardware steps of the C-IL configuration and in the next chapter we present an implementation of a simple SPT algorithm and sketch a proof for software steps. In Chapter 10 we talk about proving this property for the implementation of the SPT algorithm in VCC (for both hardware and software steps).

8.2 VM Configuration

In order to specify the correct behaviour of the VM, we first need to say how VMs are abstracted from the host hardware. For this we need to introduce data, specific to a guest partition, running in a given VM.

We use abstract data type *VmConfig* to model partition specific data, and data type *VpConfig* to keep the data specific to a given virtual processor of the partition. In the VM configuration we store the array of VP configurations and (ghost) map $gpa2hpa$, which is an abstraction of the guest physical to host physical address translation:

Definition 8.5 Partition configuration

$$\begin{aligned} \mathit{VmConfig} \stackrel{\text{def}}{=} [&\mathit{vp} \in \mathit{Pid} \mapsto \mathit{VpConfig}, \\ &\mathit{gpa2hpa} \in \mathbb{B}^{pfn} \mapsto \mathbb{B}^{pfn} \cup \{\bot\}], \end{aligned}$$

In case if guest physical to host physical translation is undefined, the map $gpa2hpa$ returns $\bot$.

In the concrete hypervisor implementation, the guest configuration is maintained in the data structures of the hypervisor. To obtain the abstract configuration of VM i from a given C-IL machine $c \in \mathit{conf}_{CC+HW+G}$, we use the

[3] In our VCC proofs we have verified initialization of the data structures of the SPT algorithm, to make sure that the coupling invariant can be initially established.

following function:

$$guest_c(i \in Gid) \in VmConfig.$$

The configuration of the virtual processor is defined in the following way:

Definition 8.6 ► Vp configuration

$$\begin{aligned} VpConfig \stackrel{\text{def}}{=} [&hpid \in Pid, \\ &gwo \in \mathbb{B}^{pfn}, \\ &iwo \in \mathbb{N}_{spt\text{-}cnt}, \\ &asid \in \mathbb{N}, \\ &asid_{gen} \in \mathbb{N}, \\ &walks \in Walk \mapsto \mathbb{B}] \end{aligned}$$

The field $guest_c(i).vp[j].hpid$ denotes the index of the host hardware processor, which executes VP j of guest partition i. The field *gwo* contains the *guest walk origin* of the given VP i.e., the guest physical base address of the top level guest page table. The field *iwo*, contains the *index walk origin* of the VP i.e., the index of the top-level SPT, allocated to this VP (see Section 8.2.3). The fields *asid* and $asid_{gen}$ denote the current ASID and ASID generation of the VP (see Section 8.2.1). The ghost set *walks* is an auxiliary set, which is used to store all walks of the VP possibly residing in the host TLB. As a result, this set is a translated version of the VTLB of this VP.

We also use a shorter notation to identify the configuration of VP j of guest i in a given C-IL configuration $c \in conf_{CC+HW+G}$:

$$vp_c(i,j) \stackrel{\text{def}}{=} guest_c(i).vp[j].$$

To guarantee that hypervisor maps guest memories to memory portions disjoint from each other and from the memory where hypervisor data is located we state the following invariant:

Invariant 8.7 ► Disjoint guest memories

name	$inv\text{-}gpa2hpa\text{-}disjoint(c \in conf_{CC+HW+G})$
property	$guest_c(i).gpa2hpa(gpfn) = pfn \implies pfn \in GuestAddr,$ $guest_c(i).gpa2hpa(gpfn_1) = pfn \wedge guest_c(j).gpa2hpa(gpfn_2) = pfn$ $\implies j = i \wedge gpfn_1 = gpfn_2$

8.2.1 ASIDs and ASID generations.

Every set of SPTs is used for performing translations in a separate address space and is identified by the ASID, allocated to this address space. The hardware support for multiple address spaces and the presence of the tagged TLB on the host hardware allows us to keep walks from different address spaces present in the host TLB at the same time.

From the point of view of the guest running in the VM, its VPs may either have different address spaces or may be run in a shared address space. This depends on whether they use one or different sets of guest page tables for address translations. Different implementations of SPT algorithms may either support (up to a certain extent) sharing of SPTs by the VPs, or may allocate to every VP a separate ASID and maintain a separate set of SPTs, even if these VPs share one set of GPTs. Sharing of SPTs makes arguing about the correctness

of the virtualization significantly harder and is not considered in the frame of this thesis. Hence, we run every VP in a separate address space (allocating a separate set of page tables for every VP) and assign a unique ASID identifier to every virtual processor,

The ASID, currently assigned to VP j of VM i is stored in the field

$$vp_c(i,j).asid$$

of the partition configuration. The hypervisor may allocate a new (fresh) ASID to the VP in case the VP performs a TLB flush (this behaviour is called TLB *lazy flushing* algorithm).

Since the number of the tags supported by the x64 hardware TLB is limited to 256, we introduce another counter, which denotes the *generation* of every ASID, allocated for the VP. We store the generation of the ASID assigned to the VP in variable

$$vp_c(i,j).asid_{gen}.$$

The TLB lazy flushing algorithm utilizes TLB tags to reduce the number of TLB flushes while handling intercepts. When the hypervisor gets a request for a TLB flush from the VP (e.g., by intercepting a $mov2cr3$ request), it does not perform a real flush of the hardware TLB, but rather allocates a new ASID to this VP. Translations cached with the old ASID remain sitting in the TLB, but are never used again, because we guarantee that no VP will get this old ASID again.

The only time when we have to perform a real TLB flush is when we run out of free ASIDs on a host processor. After a (complete) flush, all ASIDs become once again available for use. All the VPs assigned to this host processor now have to obtain a new ASID. To keep track of what ASIDs are still available on a given host processor and whether an ASID of some VP was allocated before or after the last complete TLB flush, we assign each host processor with its own ASID generation and a counter of maximal ASIDs. We store this information in a special data structure, called processor local storage. Every time when we allocate a new ASID for some VP we increase the counter of maximal ASIDs and assign the ASID generation of the host processor to the VP. When we run out of free ASIDs we perform a flush, reset the maximal ASID counter and increase the ASID generation of the host processor. When a VP is scheduled to run, we check whether it has the same ASID generation as the host processor does. If this is not the case, then we allocate a new ASID to this VP.

8.2.2 Processor Local Storage

In order to implement a TLB lazy flushing algorithm one has to keep track of the current ASID generation and the maximal currently allocated ASID of every host TLB in the system. We call the data structure used for storing this data a *processor local storage* or PLS.

We model a PLS with the abstract data type *PLS*:

$$PLS \stackrel{\text{def}}{=} [asid_{gen} \in \mathbb{N}, asid_{max} \in \mathbb{N}, walks \in Walk \mapsto \mathbb{B}].$$

◄ **Definition 8.8** Processor local storage

The ghost set *walks* is used to store all walks which could be possibly residing in the host TLB and is obtained as a union of sets $vp_c(i,j).walks$ of all VPs with

valid ASIDs (valid ASIDs are introduced further in this section). As a result, this set acts as an overapproximation of the hardware TLB. For details on the way how this set is defined refer to Section 8.3.5 and Figure 8.5.

The following function is used to extract PLS of host processor i from the hypervisor configuration $c \in conf_{CC+HW+G}$:

$$pls_c(i \in Pid) \in PLS.$$

Identifying the running VP. We introduce the function $hp2vp_c(i)$, providing the ID of the virtual processor currently running on the host processor $h.p[i]$ or returning $\perp$ if the host processor is running in hypervisor mode. The virtual processor is identified by a pair of a guest ID and of the processor ID. Since the ASID together with the ASID generation are unique for every VP in the system, we are able to use these tags to uniquely identify the VP currently running on the host processor. The values of the tags are taken from the hypervisor C-IL configuration $c \in conf_{CC+HW+G}$:

Definition 8.9 ► Running VP

$$hp2vp_c(i \in Pid) \in (Gid \times Pid) \cup \{\perp\}$$

$$hp2vp_c(i) \stackrel{\text{def}}{=} \begin{cases} (j,k) & c.p[i].asid \neq 0 \wedge vp_c(j,k).asid = c.p[i].asid \\ & \wedge vp_c(j,k).asid_{gen} = pls_c(i).asid_{gen} \wedge vp_c(j,k).hpid = i \\ \perp & \text{otherwise.} \end{cases}$$

Note, that $hp2vp_c$ is well defined only when pairs of ASIDs together with ASID generations of all VPs scheduled to run on a given hardware processor are different (when this processor is running in guest mode). We state this property in the following invariant.

Invariant 8.10 ► Distinct ASIDs

name	$inv\text{-}distinct\text{-}asids(c \in conf_{CC+HW+G})$
property	$c.p[i].asid \neq 0 \wedge i = vp_c(j_1,k_1).hpid = vp_c(j_2,k_2).hpid$ $\wedge\ vp_c(j_1,k_1).asid = vp_c(j_2,k_2).asid$ $\wedge\ vp_c(j_1,k_1).asid_{gen} = vp_c(j_2,k_2).asid_{gen}$ $\implies j_1 = j_2 \wedge k_1 = k_2$

Valid ASIDs. If a given ASID could be scheduled to run on a host hardware processor without a flush, we call it *valid*. An ASID is valid on host processor i, iff there exists a VP with this ASID, which is scheduled to run on host processor i, and which has the same ASID generation as the host processor does:

Definition 8.11 ► Valid ASIDs

$$valid\text{-}asid_c(i \in Pid, asid \in \mathbb{N}) \in \mathbb{B}$$

$$valid\text{-}asid_c(i, asid) \stackrel{\text{def}}{=} \exists k,j : vp_c(k,j).hpid = i \wedge vp_c(k,j).asid = asid \wedge vp_c(k,j).asid_{gen} = pls_c(i).asid_{gen}.$$

We maintain an invariant, which guarantees that all valid ASIDs are less or equal than the maximal ASID, stored in the PLS. When allocating a fresh ASID to the VP, we use this invariant to make sure that invariant *inv-distinct-asids* is maintained

name	$inv\text{-}valid\text{-}asids\text{-}range(c \in conf_{CC+HW+G})$
property	$valid\text{-}asid_c(i, asid) \implies asid \leq pls_c(i).asid_{max}$, $vp_c(j,k).asid_{gen} \leq pls_c(vp_c(j,k).hpid).asid_{gen}$

◄ **Invariant 8.12** Valid ASIDs range

8.2.3 Shadow Page Tables

Every shadow page table (as well as a regular page table) consists of exactly 512 page table entries. The number of the allocated SPTs may either be fixed during initialization, or may be controlled dynamically by the hypervisor. We aim at a simple version of the SPT algorithm, and therefore choose a fixed number of SPTs. The set $\mathbb{N}_{spt\text{-}cnt}$ contains indices of all SPTs allocated by the hypervisor.

The following function is used to obtain the address of a given SPT in the global memory of the C-IL machine:

$$idx2hpa_c(i \in \mathbb{N}_{spt\text{-}cnt}) \in \mathbb{B}^{pfn}.$$

Note, that the address of an SPT has to be page-aligned.

Another function extracts an abstract SPT with index i from the hypervisor configuration $c \in conf_{CC+HW+G}$:

◄ **Definition 8.13** SPT abstraction

$$spt_c(i \in \mathbb{N}_{spt\text{-}cnt}) \in Pt$$
$$\forall px \in \mathbb{B}^9 : spt_c(i)[\langle px \rangle] \stackrel{\text{def}}{=}$$
$$abs\text{-}pte(c.\mathcal{M}[(idx2hpa_c(i) \circ px \circ 0^3) : (idx2hpa_c(i) \circ px \circ 1^3)]).$$

The following predicate denotes that a page table entry px of the SPT with index i points to or "walks to" (i.e., contains the address of) the SPT with index j:

◄ **Definition 8.14** SPT link

$$walks\text{-}to_c(i \in \mathbb{N}_{spt\text{-}cnt}, j \in \mathbb{N}_{spt\text{-}cnt}, px \in \mathbb{N}) \in \mathbb{B}$$
$$walks\text{-}to_c(i, j, px) \stackrel{\text{def}}{=} spt_c(i)[px].pfn = idx2hpa_c(j).$$

8.2.4 SPT Properties

Every shadow page table has a number of properties, which are used for defining the coupling relation for the VTLB and showing correctness of the algorithm. We store the auxiliary page table data in the Page Table Info (PTI) object.

◄ **Definition 8.15** Page Table Info

$$PTI \stackrel{\text{def}}{=} [used \in \mathbb{B}, vpid \in (Gid \times Pid), l \in \mathbb{N}, re \in \mathbb{B},$$
$$prefix \in \mathbb{B}^{vpfn}, r \in Rights].$$

The fields of PTI record $pti \in PTI$ have the following meaning:

- $pti.used$: the flag, which denotes whether the associated SPT is assigned to some VP or is free otherwise;

- *pti.vpid*: if flag *used* is set, returns the pair of indices (i, j) identifying the VP to which the SPT belongs (giving ID of the guest and of the VP itself);
- *pti.l*: the level of the associated SPT in the SPT tree;
- *pti.re*: the flag, denoting whether the SPT is *reachable* by the hardware TLB, i.e., the hardware TLB could fetch an entry from this SPT for the extension. Note, that this does not necessarily mean that the SPT is linked in to the current SPT tree (i.e., is reachable from the top-level SPT). If an SPT algorithm does not perform a hardware TLB invalidation after detaching a shadow subtree, then the detached SPTs could still be reachable by the HTLB, and thus cannot be reused for shadowing other GPTs. Yet, in our simple version of the SPT algorithm presented in Chapter 9 SPT is reachable by the HTLB only if it is linked into the current SPT tree of the VP;
- *pti.prefix*: the prefix of the associated SPT i.e., the virtual address range for the addresses of the walks that might use this SPT during address translation;
- *pti.r*: accumulated rights from the top-level SPT to the associated SPT.

We obtain the PTI record of a given SPT from the hypervisor configuration $c \in conf_{CC+HW+G}$ with the help of the following function:

$$pti_c(i \in \mathbb{N}_{spt\text{-}cnt}) \mapsto PTI.$$

Note, that some of the fields of the PTI (e.g., *pti.re*, *pti.prefix*, *pti.r*) might not be used for implementation of the SPT virtualization in the hypervisor. Yet, they have to be maintained as ghost values for specification and verification needs.

8.3 Coupling Invariant

The coupling invariant for the virtual hardware establishes the relation between the components of the hypervisor configuration $c \in conf_{CC+HW+G}$ and the state of the guest virtual hardware $g \in VmHardw$.

8.3.1 Memory Coupling

The main memory of the virtual machine is coupled with the guest portion of the C-IL memory of the hypervisor configuration c.

Invariant 8.16 ► Memory coupling.

name	$inv\text{-}mm\text{-}coupling(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$guest_c(i).gpa2hpa[gpfn] = pfn \implies \forall px \in \mathbb{B}^9 :$ $g[i].mm[gpfn \circ px] = c.\mathcal{M}[(gpfn \circ px \circ 0^3) : (gpfn \circ px \circ 1^3)]$

Note, that the map $guest_c(i).gpa2hpa$ operates with page frame numbers (52 bits long), the guest memory $g[i].mm$ is quadword addressable (61 bit addresses), and the memory $c.\mathcal{M}$ is byte addressable (64 bit addresses). As a result, in Invariant 8.16 we have to perform conversion of page frame numbers to quadword and byte addresses.

8.3.2 SB Coupling

In contrast to the main memory, which has to have meaningful values for all VMs at the same time, the buffers (e.g., store buffer and memory result/request buffers) of a given VP need to be coupled with the host configuration only when this VP is running on some host processor.

For the store buffer coupling, we apply the function *gpa2hpa* to physical addresses of all stores in the queue:

Invariant 8.17 SB coupling.

name	$inv\text{-}sb\text{-}coupling(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$hp2vp_c(i) = (j,k) \implies \lvert g[j].sb[k].buffer\rvert = \lvert c.p[i].sb.buffer\rvert$ $\wedge \forall l < \lvert g[j].p[k].sb.buffer\rvert, store = g[j].p[k].sb.buffer[l]:$ $(store \neq SFENCE \implies c.p[i].sb.buffer[l] = store[pa \mapsto hpa])$ $\wedge (store = SFENCE \implies c.p[i].sb.buffer[l] = store),$ $hp2vp_c(i) \neq (j,k) \wedge vp_c(j,k).hpid = i$ $\implies is\text{-}empty(g[j].sb[k]),$

where $hpa = guest_c(j).gpa2hpa[store.pa.pfn] \circ store.pa.px$.

Note, that Invariant 8.17 guarantees, that when a VP is not running on a host machine its store buffer is always empty.

8.3.3 Memory Core Coupling

Since we do not support virtualization features for the guest hardware, the values of the $CR3_{hyp}$ register is never used in the execution of the guest virtual machine and does not need to be coupled with the host machine.

Register $CR3$ is fully virtualized by the hypervisor. When the guest performs an instruction writing to $CR3$, this instruction is intercepted and the provided value is stored in the variable $vp_c(j,k).gwo$. At the same time, the *pfn* field of the host hardware $CR3$ contains the base address of the top-level SPT, allocated to the currently running VP. The type of the memory where the top-level SPT is located is required to be "write-back". Additionally, we require that the valid bit is always set in the $CR3$ registers of the VP.

Invariant 8.18 $CR3$ coupling.

name	$inv\text{-}cr3\text{-}coupling(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$g[j].p[k].CR3.pfn = vp_c(j,k).gwo,$ $g[j].p[k].CR3.valid = 1,$ $hp2vp_c(i) = (j,k)$ $\implies c.p[i].CR3.pfn = idx2hpa(vp_c(j,k).iwo)$ $\wedge root\text{-}pt\text{-}memtype(c.p[i].CR3) = WB$

Memory request/result buffers are coupled with the respective parts of the hypervisor configuration inside the C-IL semantics. The coupling for the *memres* buffer is straightforward: the *ready* bits are always said to be equal, while the other bits are equal only in case if they are meaningful i.e., when the *ready* bit is set.

The *active* bit of the *memreq* buffer of the VP is set if the *memreq* buffer of the host processor contains an active request, which is not a VMEXIT (VMEXIT requests are not simulated by the virtual machine at all). The *memreq* buffer of the host processor contains an active request if the *active* bit is set or

the *pf-flush-req* bit is set. The latter occurs if the faulty walk was found in the TLB, but the TLB invalidation, which has to be done in case of a page fault, has not been performed yet (see Section 3.5.1). When a host processor running in guest mode performs the first stage of page fault signalling and sets the *pf-flush-req* bit, the corresponding virtual processor remains in the same state as it was before. Later, when VMEXIT occurs, we simulate both the first and the second stages of the page fault processing for this VP in the PF intercept handler.

All the other fields of the *memreq* buffer are coupled only in case if the host processor contains an active request and only if their values are meaningful w.r.t the type of the pending request. The following predicate states conditional equality of memory request/result buffers:

Definition 8.19 ► Request/result buffers conditional equality

$$memreq\text{-}eq(req \in MemReqMain, req_g \in MemReqMain) \in \mathbb{B},$$
$$memres\text{-}eq(res \in MemResMain, res_g \in MemResMain) \in \mathbb{B},$$
$$\begin{aligned} memreq\text{-}eq(req, req_g) &\stackrel{\text{def}}{=} \\ &((req.active \lor req.pf\text{-}flush\text{-}req) \land req.type \neq VMEXIT \iff req_g.acitve) \\ &\land req_g.active \implies (req.type = req_g.type \\ &\quad \land (req.type \in MemAcc \implies req.\{va, r, mask\} = req_g.\{va, r, mask\}) \\ &\quad \land (req.type \in MemAcc \setminus \{read\} \implies req.data = req_g.data) \\ &\quad \land (req.type = atomic\text{-}cmpxchng \implies \\ &\qquad req.cmp\text{-}data = req_g.cmp\text{-}data)), \end{aligned}$$
$$\begin{aligned} memres\text{-}eq(res, res_g) &\stackrel{\text{def}}{=} res.ready = res_g.ready \\ &\land res.ready \implies res = res_g. \end{aligned}$$

Note, that in case when the host processor gets a VMEXIT request, the corresponding running VP does not have an active request at all.

The coupling invariant for the memory request/result buffers is then stated as follows.

Invariant 8.20 ► Memory request/result buffers coupling

name	$inv\text{-}core\text{-}buffers\text{-}coupling(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$hp2vp_c(i) = (j, k) \implies memreq\text{-}eq(c.p[i].memreq, g[j].p[k].memreq) \land memreq\text{-}eq(c.p[i].memres, g[j].p[k].memres) \land g[j].p[k].memreq.pf\text{-}flush\text{-}req = 0$

Note, that when the host processor is running in guest mode we require the *memreq.pf-flush-req* flag of the running VP to be always 0. This means, that we never simulate page-fault triggering steps when the VP is running, but rather perform this simulation while executing the code of the PF intercept handler of the hypervisor. As a result, when the host processor running in guest mode performs the first stage of the page-fault triggering, the respective virtual processor makes an empty step. Later, after the host processor performs a VMEXIT step, we execute a PF intercept handler and simulate both the first and the second stages of the PF triggering. Then we execute VMRUN and inject the proper data to the *memreq* and *memres* buffers of the host processor, which correspond to the state of the virtual hardware after we performed the simulation.

8.3.4 VTLB Coupling

Though we also could define the VTLB coupling relation solely between the MMU component of a C-IL thread (as we did with the SB and the memory result/request buffers), we decided to choose a more complex form of the VTLB coupling invariant. Our VTLB coupling relation consists of a number of invariants, relating the walks in the host TLB with the walks defined by SPTs and the walks in the VTLB, as well as stating properties of the valid ASIDs and ASID generations. The reason for this decision was to allow more modular verification of the SPT algorithm in a C verifier (see Chapter 10).

The coupling of walks in the host hardware TLB with the walks in the virtual TLB is done with the help of the function *gpa2hpa* applied to the base address field of a walk in the VTLB.

Since the translation of a given virtual address could be done by any of the complete walks, which has at least the same rights as the translation request, we want the VTLB to have walks with the maximal possible rights. So we do not strictly fix the rights of the walks in the VTLB to be equal to the rights of the walks in HTLB, but rather allow the VTLB to store more general walks.

We define the set of VTLB walks, that could be possibly used to justify a hardware walk w under the *gpa2hpa* function of VM i:

$$hw2gw_c(w \in Walk, j \in Gid) \in Walks \mapsto \mathbb{B}$$

$$hw2gw_c(w,j) \stackrel{\text{def}}{=} \nexists gw \in Walk : w.r \leq gw.r \wedge guest_c(j).gpa2hpa(gw.pfn) = w.pfn.$$

◄ **Definition 8.21**
Host walk to guest walk translation

Now we can define the crucial property, coupling all complete walks in the host TLB with the respective walks in the virtual TLBs (Figure 8.4). Note, that though we state this property here, we do not maintain it as an invariant over the hypervisor program, but rather use other VTLB coupling invariants (defined later in this section) to derive this one.

name	*inv-htlb-complete-walks*$(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$hp2vp_c(i) = (j,k) \wedge w \in c.p[i].tlb \wedge w.asid = c.p[i].asid \wedge w.l = 0 \implies \exists gw \in hw2gw_c(w,j) : gw \in g[j].tlb[k]$

◄ **Invariant 8.22**
Complete walks in HTLB

To maintain Invariant 8.22 after a step of the machine, we have to argue about the host TLB walks not only in the currently running ASID, but in all ASIDs which could possibly be scheduled to run without a preceding TLB flush. In the next section we introduce a number of auxiliary invariants, which are used to derive Invariant 8.22.

8.3.5 Auxiliary VTLB Invariants

We use the set $pls_c(i).walks$ to store all walks in valid ASIDs which could possibly be added to the host TLB since the last TLB flush (see Figure 8.5 for relations between the host TLB, the virtual TLBs, the SPTs, and the auxiliary sets of walks stored in $plc_c(i)$ and in $vp_c(j,k)$).

The following invariant couples valid walks in the host TLB with the walks from the set $pls_c(i).walks$. Additionally, it guarantees that all walks in the TLB have ASIDs less than the maximal ASID stored in the PLS. We need this

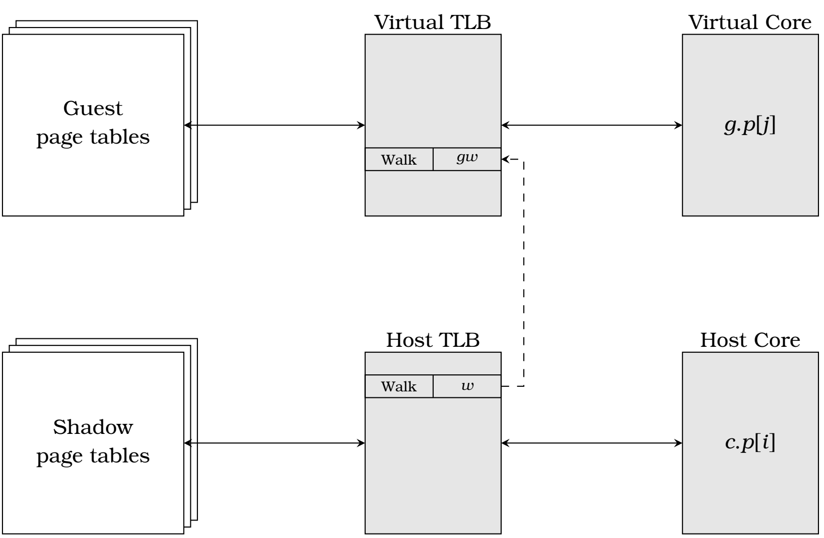

Figure 8.4: Coupling of complete walks in the host TLB.

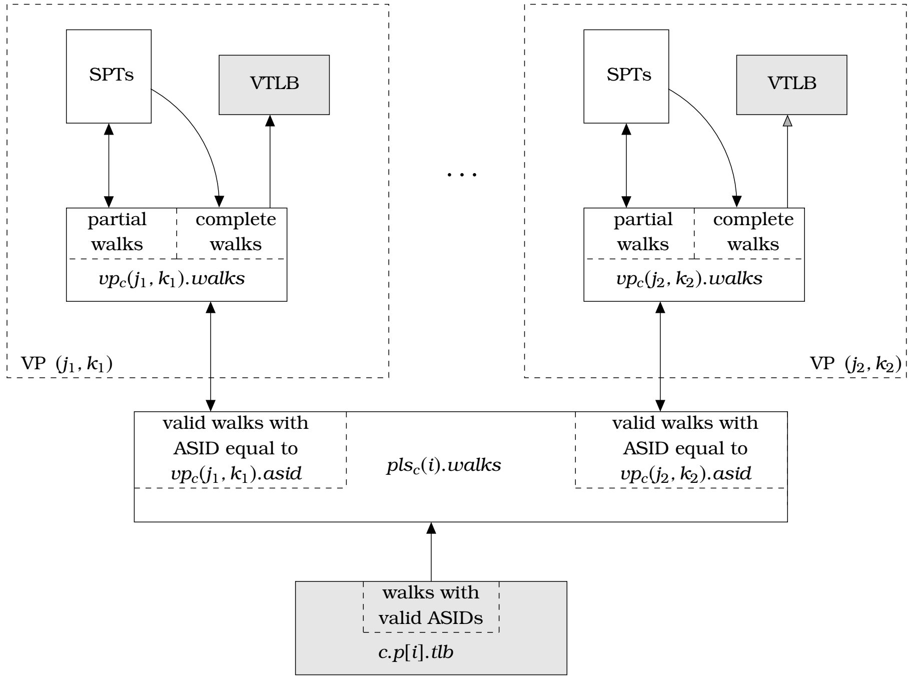

Figure 8.5: Coupling of walks in the host/virtual TLB.

property when verifying TLB lazy flushing (Section 9.4) to make sure that the hardware TLB does not have any walks in the newly allocated ASID.

Invariant 8.23 Walks in HTLB.

name	$\text{inv-htlb-walks}(c \in conf_{CC+HW+G})$
property	$w \in c.p[i].tlb \wedge \text{valid-asid}_c(i, w.asid) \implies w \in pls_c(i).walks,$ $w \in c.p[i].tlb \implies w.asid \leq pls_c(i).asid_{max}$

Another invariant couples the content of $pls_c(i).walks$ with the content of $vp_c(j,k).walks$ from the VPs assigned to hardware processor i.

Invariant 8.24 Walks in PLS.

name	$\text{inv-pls-walks}(c \in conf_{CC+HW+G})$
property	$w \in vp_c(j,k).walks \wedge \text{valid-asid}(i, w.asid) \wedge vp_c(j,k).hpid = i \implies w \in pls_c(i).walks,$ $w \in pls_c(i).walks \implies \exists j,k : vp_c(j,k).hpid = i \wedge w \in vp_c(j,k).walks \wedge vp_c(j,k).asid_{gen} = pls_c(i).asid_{gen} \wedge vp_c(j,k).asid = pls_c(i).asid$

The virtual TLB contains translated (w.r.t to the function $gpa2hpa$) versions of complete walks from the set $vp_c(j,k).walks$.

Invariant 8.25 Walks in VTLB.

name	$\text{inv-vtlb-walks}(c \in conf_{CC+HW+G}, g \in VmHardw)$
property	$w \in vp_c(j,k).walks \wedge w.l = 0 \implies \exists gw : gw \in hw2gw_c(w,j) \wedge gw \in g[j].tlb[k]$

Additionally, to maintain Invariant 8.22, we need to know that the host processors operate only in valid ASIDs. This means, that every AISD, which is run on the host hardware processor in guest mode, is currently allocated to some VP.

Invariant 8.26 Running ASID.

name	$\text{inv-running-asids}(c \in conf_{CC+HW+G})$
property	$c.p[i].asid \neq 0 \implies \text{valid-asid}_c(i, c.p[i].asid)$

Now we can use the auxiliary invariants introduced above to derive Invariant 8.22.

Lemma 8.2 (Complete walks in HTLB). *Let $c \in conf_{CC+HW+G}$ be a hypervisor configuration and g be the state of the abstract VMs s.t. Invariant 8.25 holds between c and g. Moreover, let all auxiliary VTLB invariants hold in state c. Then Invariant 8.22 also holds between c and g.*

$$
\begin{aligned}
&\text{inv-htlb-walks}(c) \\
&\wedge \text{inv-pls-walks}(c) \\
&\wedge \text{inv-vtlb-walks}(c, g) \\
&\wedge \text{inv-running-asids}(c) \\
&\wedge \text{inv-distinct-asids}(c) \\
&\implies \text{inv-htlb-complete-walks}(c, g)
\end{aligned}
$$

Proof. Follows from the invariant definitions and the definition of the function $hp2vp_c$. □

The last thing we need to do, in order to make VTLB coupling inductive, is to define the content of sets $vp_c(j,k).walks$ and $pls_c(i).walks$ in such a way, that Invariant 8.23 holds after a step of the host TLB. More precisely, we need to be sure that the host TLB adds only the walks, which are already present in $vp_c(j,k).walks$ and $pls_c(i).walks$. We fix this using the properties of SPTs, collected and maintained in the data structures of the hypervisor.

The choice, which walks are allowed to be added to the host TLB is determined by a particular implementation of the SPT algorithm. Hence, invariants fixing the content of $vp_c(j,k).walks$ are implementation specific. We define them w.r.t a simple SPT algorithm, which we present in the next chapter (Chapter 9).

8.3.6 Reachable Walks

Using the auxiliary data maintained in the PTI data structure we are able to specify the set of (complete and partial) walks belonging to a given VP.

Partial Walks

The set of the partial walks of a given VP is defined by the set of reachable SPTs. To construct a (partial) walk, "sitting" on some SPT, we need

- the level of the walk to be equal to the level of the SPT,
- access rights of the walk to be less or equal to the accumulated rights of the SPT,
- the PFN field of the walk to contain the base address of the SPT,
- the ASID of the walk to be equal to the current ASID of the VP,
- the memory type of the walk to be equal to the type of the memory where the SPT is located; as soon as we maintain invariants which guarantee that all reachable SPTs are located in the "write-back" memory (*inv-cr*3*-coupling* and *inv-memory-types*, which is defined later in this section), we can simply set the memory type of the walk to WB,
- the top-most page indices (up to the level of the SPT) of the virtual PFN of the walk have to be equal to the corresponding indices of the prefix of the SPT.

To compare the top-most indices of two virtual PFNs we use the following operator:

$$op(=_l)(vpfn_1 \in \mathbb{B}^{vpfn}, vpfn_2 \in \mathbb{B}^{vpfn}) \in \mathbb{B}$$
$$(vpfn_1 =_l vpfn_2) \stackrel{\text{def}}{=} \forall i \in [l+1:4] : (vpfn_1.px[i] = vpfn_2.px[i]).$$

The set of all partial walks of VP (j,k) sitting on the reachable SPTs is defined in the following way:

Definition 8.27 ► Partial walks through a reachable SPT

$$rwalks_c(j \in Gid, k \in Pid) \in Walk \mapsto \mathbb{B}$$
$$\begin{aligned} rwalks_c(j,k) \stackrel{\text{def}}{=} \lambda w \in Walk : \exists i \in \mathbb{N}_{spt\text{-}cnt} : \; & pti_c(i).re \wedge pti_c(i).vpid = (j,k) \wedge w.r \le pti_c(i).r \\ & \wedge w.pfn = idx2hpa_c(i) \wedge w.l = pti_c(i).l \wedge w.mt = WB \\ & \wedge w.vpfn =_{w.l} pti_c(i).prefix \wedge w.asid = vp_c(j,k).asid. \end{aligned}$$

The following invariant relates partial walks from set $vp_c(j,k).walks$ with walks over reachable SPTs of VP (j,k):

name	$inv\text{-}partial\text{-}walks(c \in conf_{CC+HW+G})$
property	$w \in rwalks_c(j,k) \implies w \in vp_c(j,k).walks,$ $w \in vp_c(j,k).walks \wedge w.l \neq 0 \implies w \in rwalks_c(j,k).$

Invariant 8.28 Partial reachable walks

The next two invariants are used to maintain Invariant 8.23 when the host TLB creates a new walk or performs a walk extension. The first one ensures that the top-level SPT is always reachable, and that it has the same initial parameters, as the top-level walk does.

name	$inv\text{-}reachable\text{-}root(c \in conf_{CC+HW+G})$
property	$pti_c(vp_c(j,k).iwo).re,$ $pti_c(vp_c(j,k).iwo).vpid = (j,k),$ $pti_c(vp_c(j,k).iwo).r = [ex \mapsto 1, rw \mapsto 1, us \mapsto 1],$ $pti_c(vp_c(j,k).iwo).l = 4$

Invariant 8.29 Reachable root

The second guarantees, that all reachable non-terminal SPTs point only to other reachable SPTs, and that the parameters of SPTs are accumulated correctly when going down the SPT tree. Additionally, we require every reachable SPT to be linked to exactly one SPTE of another reachable SPT. We use this property when we detach a subtree and mark SPTs "unreachable".

name	$inv\text{-}reachable\text{-}child(c \in conf_{CC+HW+G})$
property	$pti_c(n).re \wedge spt_c(n)[px].p$ $\implies \exists m \in \mathbb{N}_{spt\text{-}cnt} : walks\text{-}to_c(n, m, px)$ $pti_c(m).re \wedge pti_c(m).vpid = pti_c(n).vpid$ $\wedge\ pti_c(m).l = pti_c(n).l - 1$ $\wedge\ pti_c(m).r = (pti_c(n).r \wedge spt_c(n)[px].r)$ $\wedge\ pti_c(m).prefix =_{pti_c(n).l} pti_c(n).prefix$ $\wedge\ pti_c(m).prefix.px[pti_c(n).l] = bin_9(px)$ $pti_c(n).re \implies \exists! m \in \mathbb{N}_{spt\text{-}cnt}, px \in \mathbb{N}_{512} :$ $spt_c(m)[px].p \wedge walks\text{-}to_c(m, n, px) \wedge pti_c(m).re$

Invariant 8.30 Reachable child

We maintain an invariant for the fields of SPTEs, which define the memory type. We require all SPTEs to point to the memory with the "write-back" type.

name	$inv\text{-}memory\text{-}types(c \in conf_{CC+HW+G})$
property	$pti_c(n).re \wedge spt_c(n)[px].p$ $\implies mt\text{-}combine(pat\text{-}mt(spt_c(n)[px].pat\text{-}idx),$ $mtrr\text{-}mt(spt_c(n)[px].pfn)) = WB$

Invariant 8.31 Memory types

Complete Walks

A straightforward way to identify the complete walks in sets $pls_c(i).walks$ and $vp_c(j,k).walks$ is to argue about all terminal shadow PTEs that could have possibly been walked by the host TLB since the last flush [ACH+10]. The task

however is cumbersome: a single SPT could be reused for shadowing different GPTs without a complete flush of the host TLB. In this case the host TLB could have walked some shadow PTE twice - before and after it was reused for a new shadowing. In our approach we only keep track of the terminal shadow PTEs belonging to reachable SPTs, which is enough to justify the new walks added to the HTLB w.r.t the VTLB. Additionally, we make sure that the VTLB (and sets $pls_c(i).walks$ and $vp_c(j, k).walks$) drops only walks which are no longer present in the HTLB.

A complete walk through a (terminal) shadow PTE has the following properties:

- the level of the walk is equal 0,
- access rights of the walk are less or equal to the accumulated rights of the SPT and the access rights of the PTE,
- the PFN field of the walk is equal to the PFN field of the PTE,
- the ASID of the walk is equal to the current ASID of the VP,
- the memory type of the walk is equal to the memory type of the memory page, pointed by the PTE; as soon as invariant *inv-memory-types* guarantees that all PTEs point to "write-back" memory, we can simply set the memory type of the walk to *WB*,
- the top-most page indices (up to level 1) of the virtual PFN of the walk are equal to the corresponding indices of the prefix of the SPT,
- page index 0 of the virtual PFN of the walk is equal to the index of the PTE in the page table.

Formally the set of complete reachable walks of VP (j, k) is defined in the following way:

Definition 8.32 ► Complete walks through a reachable SPT

$$cwalks_c(j \in Gid, k \in Pid) \in Walk \mapsto \mathbb{B}$$

$$\begin{aligned} cwalks_c(j, k) \stackrel{\text{def}}{=} \{\!\!\{ w \in Walk : \exists i \in \mathbb{N}_{spt\text{-}cnt} : \\ & pti_c(i).re \wedge pti_c(i).vpid = (j, k) \\ & \wedge w.r \leq (pti_c(i).r \wedge spte.r) \\ & \wedge spte.p \wedge w.l = 0 \\ & \wedge w.pfn = spte.pfn \wedge w.mt = WB \\ & \wedge w.vpfn =_1 pti_c(i).prefix \wedge w.asid = vp_c(j, k).asid, \end{aligned}$$

where $spte = spt_c(i)[w.vpfn.px[1]]$.

The following invariant relates the set of complete reachable walks of VP (j, k) with complete walks from set $vp_c(j, k).walks$.

Invariant 8.33 ► Complete reachable walks

name	$inv\text{-}complete\text{-}walks(c \in conf_{CC+HW+G})$
property	$w \in cwalks_c(j, k) \implies w \in vp_c(j, k).walks,$ $w \in vp_c(j, k).walks \wedge w.l = 0 \implies w.asid = vp_c(j, k).asid$

Note, that in contrast to Invariant 8.28, we don't require all walks from $vp_c(j, k).walks$ to be included into the set $cwalks_c(j, k)$. We only require them to have the same ASID as the current ASID of the VP (together with *inv-partial-walks* this guarantees that the set $vp_c(j, k).walks$ contains only walks in the current ASID of the VP). This is sufficient, because the hardware TLB never uses complete walks for further walk extension and for fetching

PTEs. Hence, the complete walks through a terminal SPT could remain in HTLB even after this SPT is freed or reused for further shadowing.

Note also, that safety of TLBs introduced in Section 5.3 can be derived from our invariants, if we additionally require all SPTs assigned to a VP to be either shared or owned by a thread, when this thread is running in guest mode in the ASID of the VP (which we do in our VCC proofs).

8.4 Simulation

8.4.1 Simulation for Hardware C-IL Steps

The inductive version of the TLB coupling invariant includes all the VTLB invariants defined in the previous section.

Invariant 8.34 TLB coupling

name	$\mathit{inv\text{-}tlb\text{-}coupling}(c \in \mathit{conf}_{CC+HW+G}, g \in \mathit{VmHardw})$
property	$\mathit{inv\text{-}htl\text{-}walks}(c)$, $\mathit{inv\text{-}pls\text{-}walks}(c)$, $\mathit{inv\text{-}vtlb\text{-}walks}(c, g)$, $\mathit{inv\text{-}running\text{-}asids}(c)$, $\mathit{inv\text{-}distinct\text{-}asids}(c)$, $\mathit{inv\text{-}valid\text{-}asids\text{-}range}(c)$, $\mathit{inv\text{-}reachable\text{-}root}(c)$, $\mathit{inv\text{-}reachable\text{-}child}(c)$, $\mathit{inv\text{-}memory\text{-}types}(c)$, $\mathit{inv\text{-}partial\text{-}walks}(c)$, $\mathit{inv\text{-}complete\text{-}walks}(c)$

The VM coupling invariant includes the coupling for the main memory, for the buffers, and for the TLB.

Invariant 8.35 VM coupling

name	$\mathit{inv\text{-}coupling}(c \in \mathit{conf}_{CC+HW+G}, g \in \mathit{VmHardw})$
property	$\mathit{inv\text{-}gpa2hpa\text{-}gisjoint}(c)$, $\mathit{inv\text{-}mm\text{-}coupling}(c, g)$, $\mathit{inv\text{-}cr3\text{-}coupling}(c, g)$, $\mathit{inv\text{-}sb\text{-}coupling}(c, g)$, $\mathit{inv\text{-}core\text{-}buffers\text{-}coupling}(c, g)$, $\mathit{inv\text{-}tlb\text{-}coupling}(c, g)$

Now we are almost ready to prove correctness of virtualization for steps of the hardware component of the C-IL configuration. The only invariant which is missing is the one which guarantees that all C-IL abstractions defined in this chapter are not located in the guest memory and do not alias with the SPTs (and different SPTEs do not alias with each other). We call this invariant

$$\mathit{non\text{-}aliasing\text{-}abstractions}(c \in \mathit{conf}_{CC+HW+G}).$$

The formal definition of this invariant is straightforward and boring, and we do not give it here.

The following theorem argues about correctness of virtualization for steps of the hardware component of the C-IL configuration.

Theorem 8.3 (Virtualization of hardware steps). *Let $c \in conf_{CCC+HW+G}$ and c' be pre- and post-states of the concurrent C-IL machine performing a step of the hardware component. Let $g \in VmHardw$ be the state of guest VMs s.t. the coupling invariant between c and g holds. Then there exists abstract VM configuration g', s.t. the transition from g to g' is valid, the coupling invariant is maintained between c' and g', and traces of C-IL and VM executions are equal:*

$$\begin{aligned}
&\pi, \vartheta \vdash c \xrightarrow{a} c' \\
&\wedge\ hw\text{-}step(a) \\
&\wedge\ inv\text{-}coupling(c, g) \\
&\wedge\ non\text{-}aliasing\text{-}abstractions(c) \\
&\implies \exists \beta, g \xrightarrow{\beta} g' : inv\text{-}coupling(c', g') \\
&\qquad\qquad \wedge\ (traces\text{-}eq(a, \beta) \vee g = g' \wedge hw\text{-}trace(a) = \{\}).
\end{aligned}$$

Proof. To show that the theorem holds, we first have to find the ID of the VP currently being executed on the host processor. Invariant *inv-running-asid* ensures that the ASID of the host processor making a step is valid:

$$valid\text{-}asid_c(i, c.p[i].asid).$$

Unfolding the definition of a valid ASID we can find virtual processor (j, k) s.t.

$$\begin{aligned}
&vp_c(j, k).hpid = i \\
&\wedge\ vp_c(j, k).asid = c.p[i].asid \\
&\wedge\ vp_c(j, k).asid_{gen} = pls_c(i).asid_{gen}.
\end{aligned}$$

From uniqueness of valid ASIDs, established by *inv-distinct-asids*, we conclude that the function $hp2vp_c(i)$ is well defined and returns the pair (j, k):

$$hp2vp_c(i) = (j, k).$$

Now, we perform a case split on the type of the step performed by the C-IL machine.

Case 1: a step from c to c' is a regular memory read (Definition 7.21) performed with complete walk $w \in c.p[i].tlb$. The content of the C-IL memory and the value of the $CR3$ register are not changed on the transition from c to c'. Hence, all abstractions defined on the C-IL memory (e.g., $guest_c$, spt_c, etc.) have the same values in c and c'.
Guest machine $g[j]$ is performing the same kind of a step of virtual processor $p[k]$, while the other guest machines (and guest processors) remain unchanged. Applying Lemma 8.2 we get *inv-htlb-complete-walks*(c, g) and use it to find guest walk gw, which corresponds to the host walk w s.t.

$$gw \in hw2gw_c(w, j) \wedge gw \in g[j].tlb[k].$$

From the coupling invariant we know that the memory request buffers of $c.p[i]$ and $g[j].p[k]$ have the same parameters of the read request. On the host machine the read is done from the address $hpa = w.pfn \circ c.p[i].memreq.va.off$, while the virtual guest machine performs the read from the address $pa = gw.pfn \circ g[j].p[k].memreq.va.off$. Unfolding $hw2gw_c$, we get $hpa = guest_c(j).gpa2hpa(pa.pfn) \circ pa.px$. From the coupling invariant for the physical memory and the store buffers, we get that the result of the read operation is the same on both machines and the core buffers coupling is maintained:

$$inv\text{-}core\text{-}buffers\text{-}coupling(c', g').$$

The other parts of the coupling invariant are trivially maintained.

Case 2: a step from c to c' is a regular memory write step (Definition 7.20) performed with complete walk $w \in c.p[i].tlb$. In this case a new store is added to the SB of thread i. Analogously to the previous case we find guest walk gw s.t.

$$gw \in hw2gw_c(w,j) \wedge gw \in g[j].tlb[k].$$

Guest processor $g[j].p[k]$ performs the same kind of a step, using walk gw to add a new store to the SB. Hence, SB and core buffers coupling holds after the step. All the other arguments in this case are identical to the ones from the previous case.

Case 3: a step from c to c' is a locked memory write step (Definition 7.22) performed with complete walk $w \in c.p[i].tlb$. From *inv-htlb-walks* we know that the walk w is also present in in set $pls_c(i).walks$. From *inv-vtlb-walks* it follows that there exists guest walk gw s.t.

$$gw \in hw2gw_c(w,j) \wedge gw \in g[j].tlb[k].$$

Unfolding $hw2gw_c$, we get $w.pfn = guest_c(j).gpa2hpa(w.gpfn)$. Hence, the memory write is performed to the portion of the memory, allocated to guest j at address $pa = w.pfn \circ c.p[i].memreq.va.off$. The value being written is taken from buffer *memreq*:

$$data = c.p[i].memreq.data,$$
$$mask = c.p[i].memreq.mask.$$

The result of the memory write operation to the address pa is

$$c'.\mathcal{M} = masked\text{-}update_c(\mathcal{M}, pa, data, mask).$$

The virtual guest processor $g[j].p[k]$ performs the same kind of a step using guest walk gw s.t.

$$gw \in hw2gw_c(w,j) \wedge gw \in g[j].tlb[k].$$

The virtual memory of the abstract VM is updated at the address $gpa = gw.pfn \circ g[j].p[i].memreq.va.off$, using the data $g[j].p[i].memreq.data$ and the mask $g[j].p[i].memreq.mask$ ($inv\text{-}core\text{-}buffers\text{-}coupling(c, g)$ guarantees that these values are equal to the ones in $c.p[i].memreq$).

Hence,

$$g'[j].mm[gpa] = combine(g[j].mm[gpa], (data, mask)).$$

From $pa.pfn = guest_c(j).gpa2hpa(gpa.pfn)$ applying memory coupling *inv-mm-coupling*(c, g) and unfolding *masked-update*$_c$, we get

$$\begin{aligned} g[j].mm[gpa] &= c.\mathcal{M}[pa \circ 0^3 : pa \circ 1^3], \\ g'[j].mm[gpa] &= combine(c.\mathcal{M}[pa \circ 0^3 : pa \circ 1^3], (data, mask)) \\ &= c'.\mathcal{M}[pa \circ 0^3 : pa \circ 1^3]. \end{aligned}$$

Hence, the memory coupling for the address *pa* holds. From injectivity and disjointness of the *gpa*2*hpa* maps (Invariant 8.7) we get the memory coupling for all guest machines:

$$inv\text{-}mm\text{-}coupling(c', g').$$

The core buffers coupling also holds, because the machines perform the same kind of a step starting with consistent configurations. Further, we observe that all abstractions defined on the C-IL memory (e.g., $guest_c$, spt_c, etc.) have equal values in c and c'. Hence, all other invariants are maintained between c' and g', which concludes the proof for this case.

Note, that here we rely on the fact that all abstractions defined on the C-IL memory are not located in the guest memory. Hence, a guest memory write does not affect the values of these abstractions.

Case 4: a step from c to c' is an atomic compare-exchange step (Definition 7.23). The proof for this case is completely analogous to the previous case.

Case 5: a step from c to c' is a commit store step (Definition 7.25) performed by an SB of thread i to the address *pa*. The guest machine performs the same kind of a step, updating the main memory at the address *gpa*, where

$$pa = guest_c(j).gpa2hpa(gpa).$$

Since both machines commit a store from the start of the queue, SB coupling is maintained between c' and g'. The further proof for this case is analogous to the case of a locked memory write, with the only difference being that memory result/request buffers remain unchanged in c' and g'.

Case 6: a step from c to c' is any other SB step (reorder store or drop store fence). Guest processor $g[j].p[k]$ performs the same step maintaining the SB coupling invariant. The C-IL memory remains unchanged and the values of C-IL abstraction functions are the same in c and c'. Hence, the coupling invariant holds between c' and g'.

Case 7: a step from c to c' is a triggering page fault step of thread i. In this case the guest virtual machine does not perform any steps. The flag *pf-flush-req* is set to 1 in the buffer $c'.p.memreq$, while the flag *active* is set to 0. Coupling for the memory request buffer is maintained, because all the other fields of the $c.p.memreq$ are unchanged and the flag *pf-flush-req* is said to be always zero in $g[j].p[k]$, when it is being

executed. Coupling for the memory result buffer follows from the fact that the *ready* bit is low in c and c' (this is a requirement for the step to occur). All the other parts of the coupling invariant are trivially maintained.

Case 8: a step from c to c' is a VMEXIT step of thread i. In this case the ASID of thread i is changed to 0. Hence the function $hp2vp_{c'}(i)$ will return $\bot$, and there is nothing to show for the core buffers coupling. All the other coupling invariants are trivially maintained between c' and g (the guest does not perform any steps in this case).

Case 9: a step from c to c' is an input step of accepting a memory request *req* in thread i (Definition 7.12). Guest processor $g[j].p[k]$ accepts the same memory request, performing a step $g \xrightarrow{b} g'$, where

$$b = \textit{core-issue-memreq}(j, k, req).$$

This allows us to conclude the equality of traces:

$$\textit{traces-eq}(a, b).$$

In case the request is not a VMEXIT, both machines will have the same state of the *memreq* buffer and the coupling invariant for the memory request buffer will be maintained. If the request is a VMEXIT, then the abstract VP will set the *memreq.active* bit to 0, and the conditional equality of memory request buffers will also hold. All the other coupling invariants are trivially maintained.

Case 10: a step from c to c' is an output step of reporting memory result *res* in thread i (Definition 7.13). From the coupling invariant we know that memory result buffers are consistent. Hence, guest processor $g[j].p[k]$ can perform the same kind of a step, outputing the same result *res*. This allows us to conclude equality of traces. The coupling invariants are trivially maintained.

Case 11: a step from c to c' is an MMU step of adding a new top-level walk w to $c.p[i].tlb$. Memory of the C-IL machine remains unchanged. Hence, all abstractions defined on the C-IL memory ($pls_c(i)$, $guest_c(i)$, etc.) have the same values in c and in c'.
The only invariant which might get broken by this step is *inv-htlb-walks*. The second part of this invariant follows from *inv-running-asids*, *inv-valid-asids-range*, and the fact that we can add walks only in the currently active ASID. It remains to show that the newly added walk is already present in the set $pls_{c'}(i).walks$:

$$w \in pls_{c'}(i).walks.$$

From the semantics of the create walk step (Definition 7.15) we get the following parameters of the newly added walk:

$$\begin{aligned} & w.l = 4 \wedge w.r = Rights[ex \mapsto 1, us \mapsto 1, rw \mapsto 1] \\ & \wedge w.pfn = c.p[i].CR3.pfn \wedge w.asid = c.p[i].asid \\ & \wedge w.mt = \textit{root-pt-memtype}(c.p[i].CR3). \end{aligned}$$

From invariant *inv-CR3-coupling*, we know that the *pfn* field of the

host $CR3$ register contains the allocated address of the top-level SPT and the memory type of the walks is WB:

$$\begin{aligned} w.mt &= WB, \\ w.pfn &= c.p[i].CR3.pfn \\ &= idx2hpa(vp[j][k].iwo). \end{aligned}$$

Invariant *inv-reachable-root* guarantees that the SPT with index $iwo = vp_c(j,k).iwo$ is reachable and has the following parameters:

$$\begin{aligned} &pti_c(iwo).vpid = (j,k), \\ &pti_c(iwo).r = [ex \mapsto 1, rw \mapsto 1, us \mapsto 1], \\ &pti_c(iwo).l = 4. \end{aligned}$$

Constructing the set of reachable walks for the page table with index iwo we conclude that $w \in rwalks_c(j,k)$. Applying invariants *inv-partial-walks* and *inv-pls-walks* we get

$$w \in pls_{c'}(vp_{c'}(j,k).hpid).walks,$$

which concludes the proof for this case.

Case 12: a step from c to c' is an MMU step of extending partial walk w from $c.p[i].tlb$ and adding the obtained new walk w' to $c.p[i].tlb$. Memory of the C-IL machine remains unchanged. Hence, all abstractions defined on the C-IL memory ($pls_c(i)$, $guest_c(i)$, etc.) have the same values in c and in c'. Analogously to the previous case, the only invariant which might get broken is *inv-htlb-walks*. Hence, we have to show that

$$w' \in pls_{c'}(i).walks.$$

From the semantics of the extend walk step (Definition 7.15) we get

$$\begin{aligned} &pte = read\text{-}pte(c.\mathcal{M}, w.pfn, w.vpfn.px[w.l]) \\ &\wedge wext_{\surd}(w, pte, r) \\ &\wedge w' = wext(w, pte, r), \end{aligned}$$

where pte is a page table entry used for a walk extension, and w' is a newly added walk. Moreover, the asid of w equals to $c.p[i].asid$. From *inv-running-asids* it follows that $w.asid$ is valid. Invariant *inv-htlb-walks* guarantees that $w \in pls_c(i).walks$. Invariant *inv-pls-walks* gives us

$$\begin{aligned} \exists j', k' : vp_c(j',k').hpid = i \wedge w \in vp_c(j',k').walks \\ \wedge vp_c(j',k').asid_{gen} = pls_c(i).asid_{gen} \\ \wedge vp_c(j',k').asid = pls_c(i).asid. \end{aligned}$$

From the uniqueness of valid ASIDs (*inv-distinct-asids*) it follows that only one VP can have a valid ASID at a time. Hence, we get $j' = j$ and $k' = k$. Using invariant *inv-partial-walks* we conclude

$$w \in rwalks_c(j,k).$$

Unfolding definition $rwalks_c$, we obtain ID $n \in \mathbb{N}_{spt\text{-}cnt}$ of the SPT,

pointed by the field $w.pfn$:

$$pti_c(n).re \wedge w.r \leq pti_c(n).r \wedge w.pfn = idx2hpa_c(n) \\ \wedge w.l = pti_c(n).l \wedge w.vpfn =_{w.l} pti_c(n).prefix.$$

Unfolding definitions $wext$ and $wext_{\surd}$, we obtain the parameters of the newly added walk w':

$$w'.l = w.l - 1 \wedge w'.pfn = pte.pfn \wedge w'.r \leq w.r \\ \wedge w'.r \leq pte.r \wedge w'.vpfn = w.vpf \\ \wedge w.mt = mt\text{-}combine(pat\text{-}mt(pte.pat\text{-}idx), mtrr\text{-}mt(pte.pfn)).$$

Invariant *inv-memory-types* guarantees that all PTEs point to a "write-back" memory:

$$w.mt = WB.$$

Further, we need to consider two sub-cases.

Case 12.1: if the level of SPT n is greater than 1, then w' is a partial walk. Applying invariant *inv-reachable-child*, we get that *pte* points to some other reachable SPT with index m, with the following properties:

$$pti_c(m).l = pti_c(n).l - 1 \wedge pti_c(m).re \\ \wedge pti_c(m).r = pti_c(n).l \wedge pte.r \\ \wedge pti_c(m).prefix =_{pti_c(n).l} pti_c(n).prefix \\ \wedge pti_c(m).prefix.px[pti_c(n).l] = bin_9(px).$$

Constructing the set of reachable walks for the page table with index m we get

$$w' \in rwalks_{c'}(j, k),$$

which, together with *inv-partial-walks* and *inv-pls-walks*, concludes the proof for this case,

Case 12.2: if the level of SPT n equals 1, then w' is a complete walk. Constructing the set of complete walks $cwalks_c(j, k)$ over the page table with index n we get

$$w' \in cwalks_c(j, k),$$

and, applying invariant *inv-complete-walks* we conclude the proof for this case.

Case 13: a step from c to c' is an MMU step of setting A/D bits in a PTE pointed by walk $w \in c.p[i].tlb$. From the semantics of the set A/D step (Definition 7.17) we get

$$pte = read\text{-}pte(c.\mathcal{M}, w.pfn, w.vpfn.px[w.l]) \\ \wedge pte' = pte\text{-}set\text{-}ad\text{-}bits(pte, w) \\ \wedge c'.\mathcal{M} = write\text{-}pte(c.\mathcal{M}, pte\text{-}addr(w.pfn, w.vpfn.px[w.l]), pte').$$

Using invariants *inv-htlb-walks*, *inv-pls-walks*, and *inv-partial-walks* we conclude that walk w belongs to the set of reachable walks of VP

(j, k):

$$w \in rwalks_c(j, k).$$

Unfolding definition $rwalks_c$, we obtain ID $n \in \mathbb{N}_{spt\text{-}cnt}$ of the SPT, pointed to by the field $w.pfn$. The only abstraction which gets changed during the transition from c to c' is $spt_c(n)$. All other abstractions defined on the C-IL memory ($pls_c(i)$, $guest_c(i)$, etc.) have the same values in c and in c'. Further, we observe that setting of A/D bits in a given PTE can not break any invariant introduced in this chapter. Hence, the coupling invariant is maintained in c'.
Note, that here we rely on the fact that all abstractions defined on the C-IL memory do not alias and that the update of a single shadow PTE does not affect values of other abstractions.

□

Note, that we could extend Theorem 8.3 with the postcondition, saying that only SPTs assigned to a running VP can be modified by the hardware component of a thread. We need this statement to make sure that the hardware component does not break the data structures of VPs which are sleeping or which are running on other processors. Yet, stating this framing property formally would require us to introduce ownership on objects, which we don't have in our semantics thus far. Nevertheless, when verifying the hypervisor code in VCC (together with the steps of hardware components of the threads), we do state these framing conditions by identifying the sets of objects which can be modified in a step of the hardware component.

Moreover, one can observe that some of the coupling invariants are local to a single VP (as e.g., *inv-complete-walks*, *inv-partial-walks*, *inv-reachable-root*, *inv-reachable-child*, *inv-pls-walks*, and *inv-vtlb-walks*) and do not have to hold all the time, but are strictly required to hold only when this VP is being executed on a host processor. As a result, the statement of Theorem 8.3 can be weakened to talk only about the processor-local coupling invariants of the running VP and about the "global" part of the coupling invariant (as e.g., memory coupling and *inv-htlb-walks*). When verifying the algorithm in VCC we use this modular approach (see Section 10.4). Nevertheless, in the proof sketch of the SPT algorithm presented in this thesis we stick to the formalism introduced in this chapter and maintain all parts of the coupling invariant after every step of the C machine.

8.4.2 Correctness of VMRUN

The following lemma states simulation of an empty guest step for the execution of a VMRUN statement. In this lemma we require VMRUN to have appropriate parameters of the injected memory request/result. When implementing a concrete hypervisor one has to argue that the abstraction of the VMRUN statement always gets these (appropriate) parameters. To prove this, one has to argue about the following parts of the hypervisor program:

- first, one has to make sure that the intercept handling mechanism chooses an appropriate intercept handler, providing it with the appropriate parameters obtained after a VMEXIT event (e.g., a page

faulting address and faulting access rights in case of a page fault intercept). From this, one concludes that the preconditions on the state of the guest virtual hardware required by the chosen intercept handler are satisfied;

- second, one uses the correctness of the chosen intercept handler, which (possibly) simulates a number of guest steps producing a configuration of the virtual machine with certain parameters (specified by the postcondition of the intercept handler);
- finally, a correct VMRUN mechanism ensures that the memory request injected to the guest matches the state of the guest virtual machine produced by the intercept handler (e.g., if a page fault is injected, it has to be justified by the postconditions of the page fault intercept handler).

Lemma 8.4 (Correct virtualization of VMRUN). *Let $c \in conf_{CCC+HW+G}$ and c' be pre- and post-states of the concurrent C-IL machine performing a VMRUN step of thread i. Let g be the state of guest VMs s.t. the coupling invariant between c and g holds. Further, let the parameters of the VMRUN statement be in-sync with the state of the abstract VP scheduled to be run on processor i. Then the coupling invariants also holds between g and c'.*

$$
\begin{aligned}
&\wedge\ \mathit{inv\text{-}coupling}(c, g)\\
&\wedge\ c.p[i].\mathit{asid} = 0\\
&\wedge\ \pi, \theta \vdash c \rightarrow_i c'\\
&\wedge\ \mathit{stmt}_{next}(c(i), \pi) = \boldsymbol{vmrun}(e_0, e_1, e_2)\\
&\wedge\ \mathit{hp2vp}_{c'}(i) = (j, k)\\
&\wedge\ \mathit{inject\text{-}data} = \mathit{inject\text{-}data}^{\pi,\theta}(c(k), e_2)\\
&\wedge\ \mathit{memreq\text{-}eq}(\mathit{inject\text{-}data.memreq}, g[j].p[k].\mathit{memreq})\\
&\wedge \mathit{memres} = \mathit{MemResMain}[\mathit{ready} \mapsto \mathit{inject\text{-}data.ready},\\
&\qquad\qquad\qquad\qquad \mathit{pf} \mapsto \mathit{inject\text{-}data.pf}, \mathit{data} \mapsto 0]\\
&\wedge\ \mathit{memres\text{-}eq}(\mathit{memres}, g[j].p[k].\mathit{memres})\\
&\wedge\ [e_1]_c^{\pi,\theta} = \boldsymbol{val}(\mathit{cr3in}, \mathit{u64})\\
&\wedge\ \langle \mathit{cr3in} \rangle.\mathit{pfn} = \mathit{idx2hpa}_c(\mathit{vp}_c(j, k).\mathit{iwo})\\
&\wedge\ \mathit{root\text{-}pt\text{-}memtype}(\langle \mathit{cr3in} \rangle) = \mathit{WB}\\
&\wedge\ \mathit{is\text{-}empty}(g[j].p[k].\mathit{sb})\\
&\implies \mathit{inv\text{-}coupling}(c', g)
\end{aligned}
$$

Proof. Store buffer coupling holds after the step, because both the store buffer of the abstract VP and of the host processor are empty. Coupling of the memory result and request buffers holds, because we inject to the host processor the same values of the memory request and result, as the abstract VP has. Analogously, *inv-cr3-coupling* is maintained, because the new value of the *CR3* register has the proper value of the *pfn* field and the "write-back" memory type. Since we require the function *hp2vp* after the step to return ID of the VP (and not ⊥), invariant *inv-running-asids* holds in c'. All the other parts of the coupling invariant are trivially maintained, because we do not update the memory of the C-IL machine in the VMRUN step. □

Note, that when we enter the hypervisor after a VMEXIT event, the VP which was executed on the processor before the VMEXIT has occurred always has

the *memres.ready* bit set to 0. Hence, during VMRUN we can inject the active result to the *memres* buffer only if some steps of the VP have been simulated by the hypervisor and the resulting VP state has this bit set to 1. Since we never simulate memory read/compare-exchange operations in the hypervisor, the field *memres.data* of the VP will always be equal 0, if *memres.ready* equals 1 (for all steps, except memory read/compare-exchange we set *memres.data* to 0).

The memory request buffer of the abstract VP, on the other hand, after VMEXIT may either contain no request (if VMEXIT was requested from the instruction automaton) or contain an active request. In the first case we cannot simulate any steps of the memory core of the abstract VP (though we could possibly simulate steps of the TLB or of the SB). Hence, when executing next VMRUN step the state of the memory request buffer of the VP will be unchanged and we have to inject an inactive memory request to the *memreq* buffer of the host processor.

In case the VP has an active memory request at VMEXIT, we again have two options. One of them is to leave the state of the *memreq* buffer unchanged (no steps of the memory core simulated) and to inject the same type of the request to the *memreq* buffer of the host processor at VMRUN. The memory request, which caused a VMEXIT, will be repeated then. We do this for instance in case when we detect a spurious page fault in SPTs, which we fix in the PF intercept handler. The other option is to simulate steps of the memory core of the VP, and to inject at VMRUN the resulting state. After VMRUN, the guest will have an illusion that the intercepted memory access has successfully been served. For instance, we do this when we handle INVLPG and move to $CR3$ intercepts, or when we detect a page fault in GPTs in the PF intercept handler.

CHAPTER 9

Shadow Page Table Algorithm

9.1 Types and Data Structures

9.2 Software Walks

9.3 Basic Functions on Page Tables

9.4 TLB Lazy Flushing

9.5 Intercept Handlers

The SPT algorithm virtualizes intercepted page faults and TLB controlling instructions of the guest, maintaining the invariants of the virtual TLB defined in Chapter 8. In this chapter we present a C implementation of the basic "Virtual TLB" algorithm described in [Int11, Chapter 28] and [HP10]. Additionally, we provide the most crucial portions of the ghost code necessary for maintaining coupling invariants from Chapter 8 and sketch the most crucial arguments showing that the code maintains these invariants. The code presented in this chapter was formally verified in Microsoft's VCC verifier (Chapter 10).

Most realizations of the SPT algorithms share the general TLB virtualization approach. Nevertheless, they differ a lot in details and optimizations. These optimizations for instance include sharing of SPTs between different processors and selective write-protection of GPTs from guest edits to keep them in sync with their SPTs (so that they don't have to be flushed on a guest address-space switch) [SHW+08, Phi06]. We consider the simplest version of the SPT algorithm, without sharing and without write-protection of GPTs.

In this (and the following) chapter we use the regular C syntax when talking about hypervisor program variables, program types, and program code. Conversion from the standard C syntax to the C-IL syntax from Chapter 5 is straightforward, except for the loops which are not present in the C-IL semantics. A C program with loops has first to be translated to a C-IL program, where all loops are converted into IF-NOT-GOTO statements.

To distinguish ghost variables and code from implementation variables and code, we use the keyword **ghost**. For instance, the following statement represents an assignment to a ghost variable x:

```
1 _(ghost x = 10)
```

The value of variable x at the beginning of the function execution, or at the beginning of a loop (if used inside the loop body) is denoted by

```
1 \old(x)
```

A map from an integer to an integer is stated as

```
1 _(ghost int m[int];)
```

Assingment of a lambda expression to a map is written in the following way:

```
1 _(ghost m = \lambda int a; a+1)
```

An update of field f of record r with value x is stated as

```
1 _(ghost r = r[f := x])
```

In this Chapter we also give comments on our VCC annotations and proofs. These comments should be considered in the context of the next chapter, but since they are related to the code presented in this chapter, we leave them here in blocks of this kind.

9.1 Types and Data Structures

9.1.1 Constants and Types

We fix the number of virtual processors in a guest by the constant VP_CNT and the number of processors in a host hardware machine by the constant PROC_CNT. The number of guest partitions is fixed by the constant GUEST_CNT.

We use the type uint for unsigned integers 32-bit long and the type uint64 for unsigned integers 64-bit long. For physical/virtual page frame numbers and PTEs we use dedicated types Ppfn, Vpfn, and Pte respectively. All these types are shorthands for uint64. For ASIDs we use a dedicated type ASID (which is a shorthand for an 8-bit unsigned integer) and for ASID generations we use the type ASIDGen, which is again a shorthand for 64-bit integers. For IDs of both hardware and virtual processors we use the type Pid and for guest IDs we use the type Gid.

For abstract memory types we use ghost type MemType, which is implemented as an enum of all possible memory types. For abstract access permissions we use a ghost type Rights, which is implemented as a boolean map (i.e., a set) of write, execute, and privilege permissions. Abstract walks

```
typedef struct _Vp {
  ASID asid; // current asid of the VP
  ASIDGen asid_generation; // ASID generation of the VP
  Ppfn gwo; // guest walk origin (points to top−level GPT)
  uint iwo; // index walk origin (index of the top−level SPT)
  Pid id; // ID of the VP
  Guest *guest; // back−link to the guest, to which VP belongs
  Pid pidx; // index of the processor on which this VP is scheduled to run
  _(ghost bool walks[AbsWalk];)
} Vp;
typedef struct _Guest {
  Vp vp[VP_CNT]; // array of VPs
  _(ghost Pid id;) // ID of the guest
  _(ghost Ppfn gpa2hpa[Ppfn];) // address map of the guest
} Guest;
typedef struct _Gm {
  Guest guests[GUEST_CNT]; // array of Guests
 } Gm;
```

Listing 9.1: VM Configuration.

are modelled by the ghost type AbsWalk, which is defined analogously to Definition 3.45.

9.1.2 VM Configuration

A configuration of a VP (Definition 8.6) is stored in an instance of the data type Vp (Listing 9.1). A single VP contains its current ASID, its ASID generation, guest and index walk origins, an identifier, a back-link pointer to the partition configuration, an index of the hardware processor on which this VP is scheduled to run, and the (ghost) set of walks belonging to this VP.

Configuration of a guest partition (Definition 8.5) is stored in an instance of the data type Guest (Listing 9.1). A single guest configuration contains an array of VPs, which belong to this guest, a ghost identifier, and a ghost $gpa2hpa$ map.

A guest manager is implemented by the data type Gm (Listing 9.1) and contains the array of partition configurations.

In VCC annotations the VP data structure owns all SPTs and PTIs assigned to the VP.

9.1.3 Processor Local Storage

A PLS (Section 8.2.2) is implemented with the following data type.

```
typedef struct _Pls {
  ASID max_asid; // maximal ASID in use
  ASIDGen asid_generation; // ASID generation
  _(ghost bool walks[AbsWalk];)
} Pls;
```

A pointer to the PLS of a given processor is always stored in a dedicated hardware register. For instance, it can be stored in one of the segment registers

```
typedef struct _Spt {
  volatile Pte e[512];
} Spt;
typedef struct _Gpt {
  volatile Pte e[512];
} Gpt;
typedef struct _Pti {
  uint l; // level of SPT
  _(ghost Rights r) // accumulated rights
  _(ghost Vpfn prefix) // virtual prefix of a corresponding GPT
  _(ghost bool used;) // used flag
  _(ghost bool re;) // reachable flag
  _(ghost Gid gid;) // identifier of the guest
  _(ghost Pid vpid;) // identifier of the VP
} Pti;
typedef struct _Am {
  SpinLock free_spt_lock; // lock for free SPTs
  Pti PTI[SPT_CNT]; // array of PTIs
  bool free_spt[SPT_CNT]; // list of free SPTs
  Spt SPT[SPT_CNT]; // array of SPTs
} Am;
```

Listing 9.2: Page tables, PTIs, and the address manager.

if segmentation is disabled on the host machine (which is normally the case when paging is used).

To obtain the pointer to the PLS we use the following function.

```
Pls* get_pls();
```

In our VCC verification the set of all possible walks residing in a hardware TLB is located not in the PLS, but is the special ghost data structure which we call "hardware interface" (Section 10.2.2). The hardware interface is used to keep the invariants which relate the HW state of a C thread (which is modelled as a ghost object in VCC[a]) with the data structures of the SPT algorithm, as well as the data necessary for maintaining these invariants.

---

[a]For reasons why we use the ghost state to model the hardware component of a thread refer to Section 10.2

9.1.4 Page Tables

A single shadow page table (as well as a single guest page table) contains 512 volatile PTEs, where every PTE is a 64-bit integer[1] (Listing 9.2) . Note, that since in our algorithm we do not support sharing of SPTs, it is not strictly necessary to make them volatile. Yet, we developed our algorithm with the goal to further add sharing of SPTs (which remains as future work) and decided to stick with volatile SPTs to make this change easier in the future (this mainly refers to VCC annotations, which treat volatile and regular fields differently).

[1]Normally a PTE is implemented as a 64-bit union. Yet, at the time when we did our VCC proofs unions were considered as separate objects in VCC and arguing about plain 64 bit integers was much more efficient. To perform updates on single fields of a PTE we use macros, which resemble updates of respective fields in the union.

```
typedef struct _Walk {
  Ppfn pfn; // page frame number
  uint level; // level of the walk
  Walk_state state; // state of the walk
  Vpfn vpfn; // virtual PFN
  bool ex; // execute bit
  bool us; // privilege bit
  bool rw; // write bit
} Walk;
typedef enum Walk_state_ {
  WS_PROGRESS = 0, // walk in progress
  WS_COMPLETE = 1, // walk successfully completed
  WS_FAULT_NP = 2, // non−present page fault occurred
  WS_FAULT_RSV = 3, // reserved (``valid'') bit violation occured
  WS_FAULT_PVL = 4, // permission check failed
} Walk_state;
```

Listing 9.3: Software walks

A single PTI data structure (Section 8.2.4) is implemented with the data type `Pti` (Listing 9.2).

We introduce different types for GPTs and SPTs because in VCC we annotate them with different invariants (see Section 10.2.3 and Section 10.3). We don't put the fields `used`, `re`, `gid` *and* `vpid` *to the PTI data structure. Instead of this, we maintain maps of indices of used and reachable SPTs in the VP configuration (an instance of the data type* `Vp`*). Additionally, in the* `Guest` *data structure we maintain invariants over maps from different VPs which guarantee their disjointness (i.e., no single SPT can be marked as used or reachable in multiple VPs). Since approach with maps is counterintuitive and was implemented only to make technical work with invariants easier in VCC, we stick here to the formalism introduced in Chapter 8 and leave the fields mentioned above present in the PTI data structure.*

All SPTs and associated PTIs are stored in the data structure of type `Am` (Listing 9.2). This data structure contains a lock on free SPTs, which has to be acquired by a thread in order to allocate or deallocate an SPT. Bit-array `free_spt` is used to denote which SPTs are still remaining free in the system and can be used to shadow a GPT.

9.2 Software Walks

So far in this thesis we have talked only about abstract walks, which we used to store a state of the hardware address translation. Yet, in the SPT algorithm we also have to talk about the walks over page tables, which are performed by software. We call these walk *software walks*. A single software walk is implemented by the data type `Walk` (Listing 9.3).

All fields of a software walk have the same meaning as the fields of an abstract walk introduced in Section 3.4.1. The only difference is the `state` field, which is used here instead of a page fault flag of an abstract walk. The state of a walk does not only give information whether this walk is faulty or

not, but also identifies a particular type of the page fault. Additionally, the state of a walk distinguishes between complete and partial walks.

To initialize the walk with given parameters and to extend a walk over a given PTE we use the following functions:

```
1 Walk initwalk(Ppfn wo, Vpfn vpfn, bool ex, bool us, bool rw);
2 Walk wextf(Pte pte, Walk walk);
```

Implementation of these functions is straightforward and matches definitions introduced in Section 3.4.3.

9.3 Basic Functions on Page Tables

In this section we provide the implementation for a number of functions which are later used in intercept handlers of the SPT algorithm.

9.3.1 Creating an SPT

To set all entries of SPT *i* with zero values we use the following function:

```
1 void init_SPT(Am *am, uint i);
```

Another function is used to return the index of the first free SPT from the respective list in the address manager[2]:

```
1 uint find_free_spt(Am *am);
```

For acquiring and releasing the lock from the address manager we use the following functions[3]:

```
1 void SpinLockAcquire(SpinLock *l);
2 void SpinLockRelease(SpinLock *l);
```

Function `createshadow` (Listing 9.4) is used to find a free SPT and initialized it with given parameters. Note, that initially we set flag *re* in the PTI to zero, denoting that a fresh SPT is not yet linked to the SPT tree and therefore no walks over this SPT could be present in the hardware TLB.

9.3.2 Shadowing a GPT

Function `compspte` (Listing 9.5) is used to construct an SPTE, which shadows a given GPTE. For the case of a non-terminal SPTE this also includes finding and initializing a free SPT which will be pointed to by a newly constructed SPTE.

As an input this function takes pointers to the address manager and to the VP configuration, a GPTE to be shadowed, an index of the SPT which will hold

[2]Currently we assume that there is always at least one free SPT available. To weaken this assumption one has to implement a more sophisticated approach in management of free/shared SPTs. For instance, one can allocate SPTs dynamically from the heap memory of the hypervisor and limit the number of SPTs which can be allocated to a given VP to make sure that every VP will get its own portion of the heap memory reserved for SPTs. Further, if the number of SPTs allocated to a single VP exceeds the limit, one has to find some SPTs for reclaiming (i.e., detaching and freeing). In our algorithm we do reclaiming only at the time when we detach a subtree in the PF intercept handler, but a similar reclaiming strategy can be applied to an arbitrary SPT of a given VP.

[3]On annotation and verification of acquiring/releasing a lock in VCC refer to [HL09].

```
1  uint createshadow(Am *am, Vp *vp, uint l _(ghost Vpfn prefix, Rights r)
       )
2  {
3    uint u;
4    SpinLockAcquire(&am->free_spt_lock);
5    u = find_free_spt(am);
6    am->free_spt[u] = 0;
7    am->PTI[u].l = l;
8    _(ghost am->PTI[u].vpid = vp->id)
9    _(ghost am->PTI[u].gid = vp->guest->id)
10   _(ghost am->PTI[u].used = 1)
11   _(ghost am->PTI[u].re = 0)
12   _(ghost am->PTI[u].prefix = prefix)
13   _(ghost am->PTI[u].r = (l == 4 ? ALL_INITIAL_RIGHTS : r))
14   init_SPT(am, u);
15   SpinLockRelease(&am->free_spt_lock);
16   return u;
17 }
```

Listing 9.4: Allocating an SPT.

```
1  Pte compspte(Am *am, Vp *vp, Pte gpte, uint idx _(ghost uint px))
2  {
3    Pte spte;
4    uint u;
5    Ppfn ppfn;
6    _(ghost Rights r)
7    _(ghost Vpfn prefix)
8
9    spte = SET_WB_PAT_MEMTYPE(gpte);
10   if (am->PTI[idx].l > 1) {
11     _(ghost r = ACCUM_RIGHTS(am->PTI[idx].r, READ_PTE_RW(gpte),
           READ_PTE_US(gpte), READ_PTE_EX(gpte)))
12     _(ghost prefix = am->PTI[idx].prefix + (px << (am->PTI[idx].l - 1)))
13     u = createshadow(am, vp, am->PTI[idx].l - 1 _(ghost prefix, r));
14     ppfn = &am->SPT[u];
15     spte = WRITE_PTE_PFN(spte, ppfn);
16   } else {
17     ppfn = compute_gpa2hpa(READ_PTE_PFN(gpte), vp->guest);
18     spte = WRITE_PTE_PFN(spte, ppfn);
19   }
20   return spte;
21 }
```

Listing 9.5: Computing an SPTE from a GPTE.

a constructed SPTE, and a page index in this SPT where the new SPTE will be located (the latter two parameters we need for proper initialization of a newly allocated SPT and for distinguishing a terminal SPTE from a non-terminal one).

Note, that in case the returned SPTE is not terminal, the new SPT pointed to by this SPTE satisfies the conditions of *inv-reachable-child*. If the returned SPTE is terminal, then its PFN field is obtained by applying map *gpa2hpa* to the PFN field of the shadowed GPT. Implementation of function `compute_gpa2hpa` depends on the way how map *gpa2hpa* is defined in the implementation. For instance, one way to define this map is by the means of a separate set of page tables, called *host page tables*. In this thesis we leave a particular implementation of *gpa2hpa* out of the scope and therefore leave function `compute_gpa2hpa` undefined, assuming that its return value complies with the ghost map *gpa2hpa* stored in the guest configuration.

Note also, that function `compspte` has to guarantee that the newly constructed PTE points to the memory with the "write-back" memory type. This is necessary to maintain invariant *inv-memory-types*, after we write the new SPTE to the SPT tree. Moreover, we have to additionally restrict the value of MTRR registers (see Section 3.3.1) to return a WB memory type for any PFN allocated to the guest and for a base address of any SPT (we get a *WB* memory type only if both the PAT and the MTRR memory types are *WB* [Adv11a, 199]).

In our VCC verification so far we haven't argued about memory types of the walks at all (we considered them to be already invisible). We also don't prove that invariant inv-memory-types is maintained. Extending our VCC proofs to argue about memory types is considered as one of the directions of the future work.

9.3.3 Walking SPTs

As the result of page table walking we return the set of PTEs fetched during the walking process and the level where the walking has stopped. If we return a result with `level` equals 0, then the walking was successful and fetched PTEs do not contain a page fault. Otherwise, if level is greater than zero, then `pte[level]` contains a page-faulty PTE.

```
1 typedef struct _Walkres {
2  Pte pte[5];
3  uint level;
4 } Walkres;
```

To find the index of the SPT from a given base address of the SPT we use the following function:

```
1 uint SPTa2i(Am_t *am, Ppfn ba);
```

Another function is used to calculate the page index of the next PTE to be fetched from a given virtual PFN and the level of the PTE:

```
1 uint compute_idx(Vpfn vpfn, uint level)
2 {
3  return (vpfn >> ((level - 1) * 9)) & 0x1FF;
4 }
```

```
1  Walkres walkshadow(Am *am, Vp *vp, Vpfn vpfn, bool ex, bool us, bool rw
       )
2  {
3    Walkres res;
4    Walk ws[5];
5    Ppfn wo;
6    uint idx;
7    uint px;
8    bool fault;
9    fault = 0;
10   wo = &am->SPT[vp->iwo];
11   ws[4] = initwalk(wo, vpfn, ex, us, rw);
12   res.level = 4;
13   while (res.level > 0 && !fault) {
14     idx = SPTa2i(am, ws[res.level].pfn);
15     px = compute_idx(vpfn, res.level);
16     res.pte[res.level] = am->SPT[idx].e[px];
17       ws[res.level - 1] = wextf(res.pte[res.level], ws[res.level]);
18       if (ws[res.level - 1].state < WS_FAULT_NP) {
19         res.level = res.level - 1;
20       } else {
21         fault = 1;
22       }
23   }
24   return res;
25 }
```

Listing 9.6: Walking shadow page tables.

Walking of SPTs of a given VP is performed by function `walkshadow` (Listing 9.6), which takes as an input pointers to the address manager and to the VP configuration, the virtual PFN to be translated, and the set of access permissions for a translation.

We start with initializing a top-level walk using the index walk origin of a given VP. Then we start fetching PTEs and performing walk extensions until we either get a page fault or complete the translation.

At the beginning of the function we use invariant inv-reachable-root (Invariant 8.29) to get the properties of the top-level SPT. Further, we use invariant inv-reachable-child (Invariant 8.30) to find properties of other SPTs used during walking. These invariants guarantee that we only fetch PTEs which are owned by our VP and hence do not change during walking. As a result, when the function returns a set of fetched PTEs we know that they are still present in the SPT tree of the VP.

9.3.4 Walking GPTs

When walking GPTs (Listing 9.7), the code of the hypervisor plays the role of a virtual MMU. Hence, all operations performed with GPTs have to be simulated by a VTLB. The main problem here is the setting of access and dirty bits in GPTs, which has to be atomic. The x64 architecture does not provide an instruction performing a generic atomic read-modify-write (not to be confused with an atomic compare-exchange, which is provided by the x64

ISA). To overcome this restriction and to perform an atomic GPTE update we execute a loop (line 19), where we do the following

- we fetch the GPTE to a local variable (line 20),
- we check whether the fetched entry can be used for a walk extension (line 21). If this is not the case (line 26), we do not update the GPTE and exit the loop. If the fetched GPTE can be used for a walk extension we proceed to the next step,
- we try to perform an interlocked compare-exchange operation, where we check whether the entry in the memory is still the same one as was fetched in the beginning of the loop (line 22). If this is the case, then the value written to the memory by the compare-exchange instruction is the fetched GPTE with A/D bits set. The compare-exchange returns 1 and we exit the loop. If the compare-exchange fails, the update of the memory is not performed (since it would not be atomic) and we continue to the next loop iteration.

Another difference from walking of SPTs is that the fetched GPTE could point to the memory region which is out of range of the guest memory. Hence, we have to perform an additional check to ensure that PFN field of the fetched GPTE is in the allocated range of the guest memory (line 30). In this case we set the level of the walk result to special value `GM_VIOLATION`, which is greater than 4 (the maximal possible level of a successful/faulty walk).

Function `walkguest` plays a crucial role in the verification of the page fault intercept handler, because there we have to simulate the most crucial steps of the abstract virtual hardware:

- when we initialize a software walk (line 11) we simulate the step of creating a walk in the VTLB of the abstract VP;
- when we successfully set A/D bits in a GPT, we first simulate the step of setting of A/D bits by the VTLB and then simulate a walk extension. Note, that we have to do simulation of both steps at the same time as we update the GPT (line 22), because when we later write the updated value to the `res.pte` array (line 28), the GPTE in the memory could already be changed by other players and the simulation would not be possible anymore.
  We always set the access bit for present GPTEs which we fetch. The dirty bit is set only for terminal entries, when `rw` is on and all fetched GPTEs have the write permission enabled (when making a choice whether to set the dirty bit or not in line 23 we check only the last GPTE; if the write permission is not enabled in any of the previously fetched GPTEs, the walk extension (line 33) will result in a page fault and we will not be able to reach the last loop iteration).
  Note, that we always add walks to the VTLB with the maximal possible rights defined by the fetched GPTEs (independently on the input access permissions). The only exception is the last loop iteration, when we add a complete walk to the VTLB. In case bit `rw` equals 0 and all fetched GPTEs are writable (i.e., have bit *pte.rw* set to 1), we restrict the complete walk in the VLTB to contain only non-writable walks through this PTE. This is due to the fact that we don't set the dirty bit in the terminal GPTE in this case. Later, in the PF handler, we mark the SPTE, which shadows this

```
1  Walkres walkguest(Am_t *am, Guest *guest, Vpfn vpfn, Ppfn gwo, bool ex,
       bool us, bool rw)
2  {
3   Walkres res;
4   Walk ws[5];
5   Ppfn pfn;
6   Pte old_pte;
7   Gpt *gpt;
8   bool fault;
9   uint px;
10  bool cmp_result;
11  ws[4] = initwalk(gwo, vpfn, ex, us, rw);
12  res.level = 4;
13  fault = 0;
14  while (res.level > 0 && !fault && res.level != GM_VIOLATION) {
15    pfn = compute_gpa2hpa(ws.[res.level].pfn, guest);
16    gpt = (Gpt *)(pfn<<12);
17    px = compute_idx(vpfn, res.level);
18    cmp_result = 0;
19    while (!cmp_result) {
20      old_pte = gpt->e[px];
21       if (can_wextend(old_pte, rw, ex, us, res.level)) {
22         cmp_result = (old_pte == asm_cmpxchg(&gpt->e[px], old_pte,
23           (res.level == 1 && rw && READ_PTE_RW(old_pte))
24           ? SET_PTE_AD(old_pte): SET_PTE_A(old_pte)));
25       } else
26           cmp_result = 1;
27    }
28    res.pte[res.level] = (res.level == 1 && rw && READ_PTE_RW(old_pte))
29      ? SET_PTE_AD(old_pte): SET_PTE_A(old_pte);
30    if(READ_PTE_PFN(res.pte[res.level]) > MAX_GPFN) {
31      res.level = GM_VIOLATION;
32    } else {
33      ws[res.level - 1] = wextf(res.pte[res.level], ws[res.level]);
34      if (ws[res.level - 1].state < WS_FAULT_NP) {
35        res.level = res.level - 1;
36      } else {
37        fault = 1;
38      }
39    }
40  }
41  return res;
42 }
```

Listing 9.7: Walking guest page tables.

terminal GPTE as non-writable, even though the GPTE itself is marked as writable. This mechanism allows us to later intercept the first write access through this SPTE and propagate the dirty bit to the GPTE.

- when we fetch a GPTE from the memory (line 20) we simulate a page fault triggering step (*core-trigger-page-fault*) if the fetched GPTE can be used for page fault signalling. Note, that we have to simulate this step immediately at the time when we read GPTE from the memory, because later it could be overwritten by other players and the simulation would not be possible anymore.

In VCC we store the configuration of abstract VMs in the ghost state (see Section 10.2.3). Updates of the state of abstract VMs is performed by the ghost code. We update the state of the virtual hardware in the same atomic block where we access a GPT. To be able to do simulation on every iteration of the top-level while-loop we maintain loop invariants on the current state of the VTLB, which guarantee that the VTLB contains a walk of the same level as the remaining number of loop iterations. In the code snippets presented in this chapter we do not show updates of the virtual hardware. An example of such an update is shown in the next chapter (Section 10.5). We also do not present invariants, assertions, and function contracts from our VCC-annotated sources.

9.3.5 Comparing GPTEs and SPTEs

Function `notinsync` (Listing 9.8) takes as an input the results of guest and shadow walking (`gws` and `sws` respectively) and compares PTEs contained in these results. Additionally it takes integer `min_level`, which denotes the level up to which the comparison has to be done. As a result of comparison it returns the level of the SPTE, which is not-in-sync with the respective GPTE. If all the entries are in-sync, then the function returns `min_level`.

9.3.6 Reclaiming SPTs

Function `reclaim_spt` (Listing 9.9) takes as an input a pointer to the address manager, a pointer to the VP configuration, and the index of an SPT, which is going to be reclaimed. This SPT must be owned by the VP configuration.

The function first recursively reclaims all SPTs attached to the provided SPT and then marks this SPT free.

As a precondition the function requires that all SPTs in the reclaimed subtree have the *re* bit set to 0, meaning that the host TLB is not sitting on any of these tables. This precondition allows to maintain invariant *inv-reachable-child* after we do the reclaiming.

9.4 TLB Lazy Flushing

Implementation of the TLB lazy flushing algorithm, which we described in Section 8.2.1, consists of two functions: `vp_flush_tlb` is called every time when a VP requests a TLB flush and `vp_pre_run` is invoked every time when a VP is prepared to be run on a hardware processor.

```
1  uint notinsync(Am_t *am, Guest *guest, Walkres sws, Walkres gws, uint
       min_level)
2  {
3   uint level;
4   bool terminal;
5   level = 4;
6   terminal = !((bool)(level - 1);
7   while (level > min_level)
8   {
9     if (READ_PTE_A(gpte) != READ_PTE_A(spte) ||
10      READ_PTE_D(gpte) != READ_PTE_D(spte) ||
11      READ_PTE_EX(gpte) != READ_PTE_EX(spte) ||
12      READ_PTE_RW(gpte) != READ_PTE_RW(spte) ||
13      READ_PTE_US(gpte) != READ_PTE_US(spte) ||
14      READ_PTE_P(gpte) != READ_PTE_P(spte) ||
15      (terminal && READ_PTE_PFN(spte) !=
16       compute_gpa2hpa(READ_PTE_PFN(gpte), guest))){
17       break;
18      }
19      level --;
20   }
21   return level;
22 }
```

Listing 9.8: Comparing GPTEs and SPTEs.

```
1  void reclaim_spt(Am *am, Vp *vp, uint idx)
2  {
3   Ppfn child_pfn;
4   uint child_id;
5   uint pxi;
6   if (am->PTI[idx].l > 1)
7   {
8     for(pxi = 0; pxi < 512; pxi++)
9     {
10      if (READ_PTE_P(am->SPT[idx].e[pxi])) {
11         child_pfn = READ_PTE_PFN(am->SPT[idx].e[pxi]);
12      }
13      child_id = SPTa2i(am, child_pfn);
14      reclaim_spt(am, vp, child_id);
15    }
16   }
17   SpinLockAcquire(&am->free_spt_lock);
18   am->free_spt[idx] = 1;
19   _(ghost am->PTI[idx].used = 0;)
20   SpinLockRelease(&am->free_spt_lock);
21 }
```

Listing 9.9: Reclaiming SPTs.

Function `vp_flush_tlb` (Listing 9.10) tries to find the first free ASID and to allocate it to the VP. We distinguish two cases:

- the current maximal ASID is less than 255 (line 16). In this case there is still at least one free ASID available and we allocate it to the VP. The set `vp->walks` is updated respectively (line 18) to change the ASID of the walks of the VP to the newly allocated one. This allows us to maintain invariants *inv-partial-walks*, *inv-complete-walks*, and *inv-vtlb-walks* after the ASID of the VP is changed. In case the ASID of our VP was valid before the step, i.e., if the ASID generation of the VP was equal to the ASID generation of the host processor (line 20), we also update the set `pls->walks` to include all the walks of our VP with the newly allocated ASID and to remove walks with the old ASID (line 21). This is necessary to maintain invariant *inv-pls-walks* after we change the ASID of the VP. In case if the ASID of the VP was not valid before the step, we make it valid by updating the ASID generation of the VP (line 28). At that point we also have to update the set `pls->walks` to include the walks of our VP (line 26), which is necessary for maintaining *inv-pls-walks*. Note, that in this case we don't need to remove old walks of the VP from `pls->walks`, because the ASID of the VP was previously invalid.
  All ASIDs of other VPs which were valid before the function call, remain valid. The only ASID which becomes invalid (if it was valid before) is the old ASID of our VP. Invariant *inv-htlb-walks* guarantees that the host TLB does not contain any walks in the newly allocated ASID. Since the set `pls->walks` keeps all walks with ASIDs other than the old ASID of our VP, invariant *inv-htlb-walks* is maintained.
- the current maximal ASID equals 255 (line 6). In this case all available ASIDs have been already allocated and we perform a complete TLB flush[4]. After the flush we increase the ASID generation of the host processor (line 9), which makes ASIDs of all VPs assigned to this processor invalid (invariant *inv-valid-asids-range* guarantees that the ASID generation of a VP is less or equal to the ASID generation of the host processor). At the same time, we have to empty the set `pls->walks` (line 8) in order to maintain invariant *inv-pls-walks*. Further, we allocate ASID 1 to the VP (line 13) and update the set `vp->walks` to change the ASID of the walks of the VP to the newly allocated one (line 11). This allows us to maintain invariants *inv-partial-walks*, *inv-complete-walks*, and *inv-vtlb-walks* after the ASID of the VP is changed. In the end we make the ASID of our VP valid by updating the ASID generation of the VP (line 15). ASIDs of all other VPs stay invalid, and we update the set `pls->walks` to include only the walks of our VP. Hence, invariants *inv-partial-walks* and *inv-complete-walks* are maintained. Invariant *inv-htlb-walks* follows from the fact that the TLB is flushed and contains no walks at the time when we start updating our PLS[5].

[4]For the formal semantics of a complete TLB flush see Section 7.2.1.

[5]Note, that we currently assume that the ASID generation in the PLS (which is stored as a 64 bit unsigned integer) never overloads. To weaken this assumption one has to specifically handle the situation when all ASID generations are depleted. In this case the ASID generation has to be set to 0 and all VPs assigned to the current core have to be explicitly checked to make sure that

```
1  void vp_flush_tlb(Am_t *am, Vp *vp)
2  {
3   Pls *pls;
4   pls = get_pls();
5   cpu_max_asid = pls->max_asid;
6   if (cpu_max_asid == 255) {
7     complete_tlb_flush(vp);
8     _(ghost pls->walks = \lambda AbsWalk w; 0)
9     pls->asid_generation++;
10    pls->max_asid = 1;
11    _(ghost vp->walks = \lambda AbsWalk w; w.asid = 1 &&
12      vp->walks[w / {.asid = vp->asid}])
13    vp->asid = 1;
14    _(ghost pls->walks = vp->walks)
15    vp->asid_generation = pls->asid_generation;
16  } else {
17    pls->max_asid++;
18    _(ghost vp->walks = \lambda AbsWalk w; w.asid = pls->max_asid &&
19      vp->walks[w / {.asid = vp->asid}])
20    _(ghost if (vp->asid_generation == pls->asid_generation) {
21      pls->walks = \lambda AbsWalk w; vp->walks[w] ||
22        (pls->walks[w] && (w.asid != vp->asid));
23    })
24    vp->asid = pls->max_asid;
25    _(ghost if (vp->asid_generation != pls->asid_generation) {
26      pls->walks = \lambda AbsWalk w; vp->walks[w] || pls->walks[w];
27    })
28    vp->asid_generation = pls->asid_generation;
29  }
30 }
31 void vp_pre_run(Am_t *am, Vp *vp)
32 {
33  Pls *pls;
34  pls = get_pls();
35  if (pls->asid_generation != vp->asid_generation)
36    vp_flush_tlb(am, vp);
37 }
```

Listing 9.10: TLB lazy flushing.

As a postcondition of the function we know that the ASID of the provided VP is valid and that the hardware TLB does not contain any walks in that ASID. Further, we use this knowledge for simulating a complete VTLB flush in the MOVE TO $CR3$ intercept handler (Section 9.5.2).

If we increase the ASID generation of the host processor, all ASIDs of other VPs become invalid. To make sure that we don't schedule to run a VP with the ASID being invalid (and to maintain invariant *inv-running-asid*), we introduce function `vp_pre_run` (Listing 9.10). This function is called every time some VP is prepared to be scheduled to run. It checks whether the ASID of this VP is valid (by comparing ASID generations of the VP and of the host processor) and calls function `vp_flush_tlb` if it is not valid. After this, we can be sure that the VP has a valid ASID.

their ASIDs are marked invalid.

9.5 Intercept Handlers

When we continue execution of the hypervisor after VMEXIT, we have to select an appropriate intercept handler. This selection is done depending on the state of a number of control registers, which are getting the parameters of the intercept when it occurs. The state of these registers has to be in sync with the state of the abstract VP. For instance, if we decide to choose a PF handler, then the *memreq* buffer of the abstract VP should contain a request for the memory access (the request which caused the intercept) and parameters of this access should match the parameters passed into the PF handler. Moreover, the SB of the abstract VP should be empty after VMEXIT, which follows from the coupling invariant and from the fact that VMEXIT requires the SB of the host processor to be empty. Since we do not explicitly model the control registers used for storing the parameters of the intercept (they are located in the instruction part of the core which we leave undefined), we do not formally prove the correctness of the intercept dispatching process, but only verify individual intercept handlers. Nevertheless, when verifying intercept handlers we assume that dispatching is done correctly and the state of the abstract VP corresponds to the chosen handler.

All intercept handlers take as an input parameter pointer `vp` to the VP configuration. From this configuration we obtain ID (j, k) of the abstract VP which is associated with this VP configuration:

$$(j, k) = (\texttt{vp->guest->id}, \texttt{vp->id}).$$

Let $c \in conf_{CC+HW+G}$ be the state of the C-IL + HW + Ghost machine before the first statement of the intercept handler is executed and $g \in VmHardw$ be the state of the abstract VMs, where $inv\text{-}coupling(c, g)$ holds. Then the state of the abstract VP, which is associated with the provided VP configuration can be obtained as

$$(g[j].p[k], g[j].tlb[k], g[j].sb[k]).$$

As a precondition to every intercept handler we require that the VP configuration, as well as all SPTs and PTIs assigned to this VP are owned by a thread (i.e., are thread-local). The PLS is also considered to be owned by a thread, since only one thread can run on a host processor at a time. We also require the abstract VP configuration to be owned by a thread, meaning that no other threads can perform updates of this abstract VP. To make sure that no two threads get the ownership of the same abstract VP, we maintain an invariant stating uniqueness of VP and guest identifiers. We also observe that the steps of the hardware component of a thread (Theorem 8.3) update only the state of the running abstract VP. When we are executing an intercept handler, we know that no VPs assigned to our hardware processor are running. Hence, we can be sure that the state of the abstract VP stays unchanged in between the steps of the handler.

We require the coupling invariant to hold at the beginning of the function and show that it is maintained after every step of the function, independently of the scheduling.

```
1  void invlpg_intercept(Am_t *am, Vp *vp, Vpfn vpfn, uint off _(ghost
       bool page_fault))
2  {
3   Walkres sws;
4   uint pxi, idx;
5   _(ghost Pls* pls)
6   _(ghost pls = get_pls())
7   asm_invlpga(vp->asid, ((vpfn << 12) + off));
8   sws = walkshadow(am, vp, vpfn, 0, 0, 0);
9   if (sws.level == 0) {
10    pxi = compute_idx(vpfn, 1);
11    idx = SPTa2i(am, READ_PTE_PFN(sws.pte[2]));
12    am->SPT[idx].e[pxi] = RESET_PTE_P(am->SPT[idx].e[pxi]);
13  }
14  _(ghost vp->walks = \lambda AbsWalk w; vp->walks[w] && (w.l != 0 || w
        .vpfn != vpfn))
15  _(ghost if (vp->asid_generation == pls->asid_generation) {
16    pls->walks = \lambda AbsWalk w; pls->walks[w] &&
17      (w.l != 0 || w.vpfn != vpfn || w.asid != vp->asid);
18  })
19 }
```

Listing 9.11: INVLPG intercept handler.

In VCC we store the configuration of an abstract VP in the ghost state. A thread, executing an intercept handler owns the state of the abstract VP which has been intercepted. In the beginning of the handler we require the state of the VP to match the handler and its parameters (i.e., there should be an active memory request, its type and parameters should correspond to the type and parameters of the handler, and SB should be empty). Further, when executing the body of the handler we simulate steps of the VP and update its state appropriately. As a postcondition of the handler, we know that the state of the abstract VP is in sync with the return result of the handler (e.g., if we require a page fault to be propagated to the guest, then we have already simulated steps core-prepare-page-fault and core-trigger-page-fault).

9.5.1 INVLPG Handler

We use function `invlpg_intercept` (Listing 9.11) to both handle the INVLPG intercept and to perform the TLB invalidation in case when we propagate a page fault to the guest (see Section 9.5.3).

Function `invlpg_intercept` takes as an input a pointer to the address manager, a pointer to the VP configuration, and the address being invalidated, which consists of the virtual PFN `vpfn` and of the page offset (in bytes) `off`. Additionally we provide the flag `page_fault`, which we use to distinguish whether the function is handling an INVLPG intercept or is called from the PF handler.

Let (j, k) be the ID of the abstract VP which is associated with the provided VP configuration, $c \in conf_{CC+HW+G}$ be the state of the C-IL + HW + Ghost machine before the first statement of the handler is executed and $g \in VmHardw$ be the state of the abstract VMs, where $inv\text{-}coupling(c, g)$ holds.

Then as a precondition to the handler we require the abstract VP to be in the following state:

$$\begin{aligned}
\langle g[j].p[k].memreq.va\rangle &= (\texttt{vpfn} << 0^{12}) + \texttt{off},\\
\texttt{page_fault} = 0 &\implies g[j].p[k].memreq.type = INVLPG\\
&\qquad \wedge g[j].p[k].memreq.active,\\
\texttt{page_fault} = 1 &\implies g[j].p[k].memreq.type \in MemAcc\\
&\qquad \wedge g[j].p[k].memreq.pf\text{-}flush\text{-}req\\
&\qquad \wedge \neg g[j].p[k].memreq.active.
\end{aligned}$$

As a postcondition we ensure that either a *core-invlpga* or a *core-trigger-page-fault* step is performed i.e., the ready bit in the *memres* buffer is set, the *active* bit in the *memreq* buffer is lowered and the other fields of the memory request and result buffers are left unchanged or get a default "zero" value.

In function `invlpg_intercept` we first perform the hardware INVLPG in the ASID of the VP[6] (line 7) to flush the translations from the host TLB. Further, we walk down the SPT tree for the invalidated address (line 8). When reaching a terminal SPTE, we mark it non-present (line 12) and update the set `vp->walks` (line 14) to remove all complete walks with the invalidated virtual PFN. In case the ASID of the VP is valid, we also have to update the set `pls->walks` in the same manner (line 16), so that invariant *inv-pls-walks* is maintained. Invariant *inv-htlb-walks* is maintained, because at that moment the host TLB is already invalidated and does not contain translations with the provided virtual PFN.

Note, that if we don't reach a terminal PTE, this means that SPTs of the VP, do not contain valid translations for the invalidated address. Yet, sets `vp->walks` and `pls->walks` still could contain such translations, remaining there from some outdated state of SPTs. Hence, in this case we also have to update these sets.

After we remove the invalidated translations from `vp->walks`, we simulate the respective walk removal from the VTLB. After the last statement of the function is executed, we either simulate a *core-invlpga* step of the VP or step *core-trigger-page-fault*, depending on whether the function is used for handling an INVLPG intercept or is called from the PF handler. Both of the steps are possible, because we know that at the end of the function the VTLB does not contain any walks in the invalidated address.

9.5.2 MOVE TO $CR3$ Handler

Function `mov2cr3_intercept` (Listing 9.12) takes as an input a pointer to the address manager, a pointer to the VP configuration, and the physical page frame number *pfn* of the new top-level GPT.

Let (j, k) be the ID of the abstract VP which is associated with the provided VP configuration, $c \in conf_{CC+HW+G}$ be the state of the C-IL + HW + Ghost machine before the first statement of the handler is executed and $g \in VmHardw$ be the state of the abstract VMs, where $inv\text{-}coupling(c, g)$ holds. Then as a precondition to the handler we require the abstract VP to be in the

[6] For the formal semantics of the hardware INVLPGA see Section 7.2.1.

```
1  int mov2cr3_intercept(Am_t *am, Vp *vp, Ppfn gpfn)
2  {
3   uint detached_idx;
4   if(gpfn > MAX_GPFN) {
5    return RESULT_GM_RANGE_VIOLATION;
6   }
7   vp_flush_tlb(am, vp);
8   for(uint pxi = 0; pxi < 512; pxi++) {
9    if(READ_PTE_P(am->SPT[vp->iwo].e[pxi])) {
10     _(ghost mark_unreachable_subtree(am, vp, vp->iwo, pxi))
11     am->SPT[vp->iwo].e[pxi] = RESET_PTE_P(am->SPT[vp->iwo].e[pxi]);
12     detached_idx = SPTa2i(am, READ_PTE_PFN(am->SPT[vp->iwo].e[pxi]));
13     reclaim_spt(am, vp, detached_idx);
14    }
15   }
16   _(ghost vp->walks = \lambda AbsWalk w; vp->walks[w] && w.l != 0)
17   _(ghost pls->walks = \lambda AbsWalk w; pls->walks[w] &&
18    (w.l != 0 || w.asid != vp->asid))
19   vp->gwo = gpfn;
20   return RESULT_CONTINUE;
21 }
```

Listing 9.12: Move to $CR3$ intercept handler.

following state:

$$g[j].p[k].memreq.type = mov2cr3 \wedge g[j].p[k].memreq.active$$
$$\wedge \langle g[j].p[k].memreq.cr3in\rangle.pfn = \texttt{gpfn} \wedge g[j].p[k].memreq.cr3in.valid.$$

As a postcondition we guarantee that a *core-mov2cr3* step is performed i.e., the ready bit in the *memres* buffer is set, the *active* bit in the *memreq* buffer is lowered, and the other fields of the memory request and result buffers are left unchanged or are getting a default “zero” value.

First, we check whether the provided *gpfn* value fits in the range of allocated guest addresses (line 4) and continue only if it does. Next, we perform a TLB lazy flush by calling function `vp_flush_tlb` (line 7). After that we know that the hardware TLB does not contain any walks in the ASID of our VP. Further, we go through all entries of the top-level SPT, mark them not present (line 11), and reclaim all SPTs pointed by these entries (line 13). At the same time when resetting the present bit of an SPTE we reset the *re* bit in all SPTs in the detached subtrees and remove all walks (complete and incomplete ones) through this subtree from sets `vp->walks` and `pls->walks` (necessary to maintain invariant *inv-partial-walks* after the step). This is done by the ghost function `mark_unreachable_subtree` (line 10), which we leave undefined here (for the body of this function consult the sources).

When we reset the *re* bit of some SPT, we have to maintain invariant *inv-reachable-child*. The first part of this invariant guarantees that all reachable SPTEs point only to reachable SPTs. To maintain this property after the step, we use the second part of *inv-reachable-child*, which guarantees the uniqueness of a link in the SPT tree. Hence, the detached SPT tree is linked only to one SPTE, where we reset the present bit (line 11). As a result, marking this subtree unreachable does not break the first part of *inv-reachable-child*, because no reachable and present SPTE can point to any of the detached

tables. Analogously, we maintain the second part of *inv-reachable-child*.

After we have marked all entries in the top-level SPT as not present, we remove all the (possibly remaining) complete walks form the set `vp->walks` (line 16). Invariant *inv-complete-walks* is maintained, because the set of complete reachable walks though the top-level SPT is empty (since this SPT has only non-present entries). At the same time we simulate the step of removal of all walks from the VTLB. Since we know that the ASID of our VP is valid (this is guaranteed by the function `vp_flush_tlb`), we have to respectively update the set `pls->walks` (line 17) to maintain invariant *inv-pls-walks*. Invariant *inv-htlb-walks* is maintained, because the host TLB does not contain any walks in the current ASID of the VP.

Finally, after we have an empty VTLB, we set the new value for the guest walk origin in the VP configuration (line 19) and simulate step *core-mov2cr3*. Invariant *inv-cr3-coupling* is maintained, because we write the same value to the `gwo` field and to the *CR*3 register of the abstract VP.

9.5.3 PF Handler

Function `pf_intercept` (Listing 9.13) takes as an input a pointer to the address manager, a pointer to the VP configuration, the page fault address, which consists of the virtual PFN `vpfn` and of the page offset (in bytes) `off`, and access permissions `ex`, `us`, and `rw`. The function returns a result of the type `Walkres`.

Let (j, k) be the ID of the abstract VP which is associated with the provided VP configuration, $c \in conf_{CC+HW+G}$ be the state of the C-IL + HW + Ghost machine before the first statement of the handler is executed, and $g \in VmHardw$ be the state of the abstract VMs, where $inv\text{-}coupling(c, g)$ holds. Then as a precondition to the handler we require the abstract VP to be in the following state:

$$\begin{aligned}
&g[j].p[k].memreq.type \in MemAcc \wedge g[j].p[k].memreq.active \\
&\wedge \langle g[j].p[k].memreq.va \rangle = (\texttt{vpfn} << 12) + \texttt{off} \\
&\wedge g[j].p[k].memreq.r = Rights[ex \mapsto \texttt{ex}, us \mapsto \texttt{us}, rw \mapsto \texttt{rw}] \\
&\wedge tlb\text{-}invalidated\text{-}pf(g[j].tlb[k], g[j].p[k].memreq[i].va.vpfn, \texttt{vp->asid}).
\end{aligned}$$

Note, that we require the host TLB to contain no walks with the faulty virtual PFN in the ASID of the provided VP. This property follows from the fact that the TLB is invalidated in case of a VMEXIT event caused by a page fault (Section 3.5.1).

In the postcondition of the function we distinguish three cases depending on the field *res.level*, where *res* is the result returned by the function:

- if *res.level* = `GM_VIOLATION` (line 9), this means that we have encountered a (present) GPTE, which has the PFN field not fitting into the range of allocated guest addresses defined by the *gpa2hpa* function; in this case we do not simulate any steps of the virtual hardware and simply return,
- if *res.level* ≠ 0 and *res.level* ≠ `GM_VIOLATION` (line 12), it means that we have encountered a page fault while walking GPTs. In this case we simulate steps *core-prepare-page-fault* and *core-trigger-page-fault*. As a result, we ensure that the *active* and *pf-flush-req* bits in the

```
1  Walkres pf_intercept(Am_t *am, Vp *vp, Vpfn vpfn, uint off, bool ex,
       bool us, bool rw)
2  {
3   Walkres gws, sws;
4   uint z, i, idx, pxi;
5   Pte nspte[5];
6   uint detached_idx;
7   gws = walkguest(am, vp->guest, vpfn, vp->gwo, ex, us, rw);
8   if (gws.level == GM_VIOLATION)
9     return gws; // guest memory range violation
10  if (gws.level > 0) {
11    invlpg_intercept(am, vp, vpfn, off _(ghost 1));
12    return gws; // propagate a PF
13  } else {
14     sws = walkshadow(am, vp, vpfn, ex, us, rw);
15     z = notinsync(am, vp->guest, sws, gws, sws.level);
16     if (z == 0)
17       return gws; // repeat guest instruction
18     i = z;
19     detached_idx = SPT_CNT;
20     while (i > 0) {
21       if (i == z)
22         idx = SPTa2i(am, sws.pfn[i]);
23       else
24         idx = SPTa2i(am, READ_PTE_PFN(nspte[i + 1]));
25       pxi = compute_idx(vpfn, i);
26       nspte[i] = compspte(am, vp, idx, gws.pte[i], pxi);
27       if (READ_PTE_P(am->SPT[idx].e[pxi]) && i > 1) {
28         detached_idx = SPTa2i(am, READ_PTE_PFN(am->SPT[idx].e[pxi]));
29       }
30       if (i == 1 && !rw && !READ_PTE_D(gws.pte[1]) && RW_SET(gws)) {
31         _(ghost vp->walks = \lambda Walk w;
32           vp->walks[w] && (w.l != 0 || w.vpfn != vpfn) ||
33           WALK_THROUGH_PTE(w, RESET_PTE_RW(nspte[i]), am, idx, pxi))
34         _(ghost if (vp->asid_generation == pls->asid_generation) {
35           pls->walks = \lambda Walk w; vp->walks[w] ||
36             pls->walks[w] && (w.l != 0 || w.vpfn != vpfn ||
37               w.asid != vp->asid);
38         })
39         am->SPT[idx].e[pxi] = RESET_PTE_RW(nspte[i]);
40       } else {
41         _(ghost if (i > 1) {
42           mark_unreachable_subtree(am, vp, idx, pxi);
43           mark_reachable(am, vp, idx, pxi, nspte[i]);
44         } else {
45           vp->walks = \lambda Walk w;
46             vp->walks[w] && (w.l != 0 || w.vpfn != vpfn) ||
47             WALK_THROUGH_PTE(w, nspte[i], am, idx, pxi);
48           if (vp->asid_generation == pls->asid_generation) {
49             pls->walks = \lambda Walk w; vp->walks[w] ||
50             pls->walks[w] && (w.l != 0 || w.vpfn != vpfn ||
51               w.asid != vp->asid);
52           }
53         })
54         am->SPT[idx].e[pxi] = nspte[i];
55       }
56       i--;
57     }
58    if (detached_idx < SPT_CNT)
59      reclaim_spt(am, vp, detached_idx);
60    return gws; // repeat guest instruction
61  }
62 }
```

Listing 9.13: PF intercept handler.

memreq buffer are low, the *ready* bit in the *memres* buffer is high, and *memres.pf* contains the page fault information defined by the faulty GPTE *res.pte*[*res.level*]. The other fields of the *memreq* buffer in this case are left unchanged. After we return from the handler the page fault information contained in the entry *res.pte*[*res.level*] has to be injected into the VP,

- if *res.level* = 0 (line 17 or 56), then GPTs do not have a page fault for the provided parameters of the access. This means, that the PF intercept was caused by a faulty shadow PTE which is out-of-sync with the respective GPTE (this also includes the case, when the present bit in the SPTE is not set). In this case we fix the problem by allocating new SPTEs and making them in sync with the previously fetched GPTEs. We do not simulate any steps of the virtual hardware rather than VTLB steps of adding/dropping walks and setting of A/D bits (this simulation is done while walking GPTs). As a result, we ensure that the state of the *memreq* and *memres* buffers is unchanged and the request for a memory access is still pending. After we return from the handler the guest memory accessing instruction has to be repeated once again.

First, we execute function `walkguest` (line 7), which walks down the GPTs, simulates VTLB steps, and returns the result `gws` of the guest walk. Second, we check whether violation of the guest memory range was encountered (line 8) and return from the function in case it it was found.

Further, we distinguish cases when `gws.level` is greater than zero (which denotes that a PF was found in GPTs) and when it is equal to zero. In the first case postconditions of the `walkguest` function ensure that step *core-prepare-page-fault* was already performed and the abstract VP has the flag *memreq.pf-flush-req* set to 1 and the flag *memreq.active* set to 0. We then execute the function `invlpg_intercept` (line 11), which invalidates the faulty address and simulates step *core-trigger-page-fault*, and exit the function returning `gws`. Postconditions of the `walkguest` function also guarantee that the PF information from *memres.pf* is in sync with the PF information from `gws` and postconditions of function `invlpg_intercept` guarantee that *memres.pf* is not overwritten.

In case no page fault was found while walking GPTs we execute function `walkshadow` (line 14) and write the result of the shadow walk to `sws`. Next, we execute function `notinsync` (line 15), which compares `gws` with `sws` and finds the first entry in `sws` which is out-of-sync with the entry from `gws`. If no such entry is found (line 17), this means that SPTs do not contain a page fault and we simply return[7]. Otherwise we store the level of the first SPT which is out-of-sync in variable `i` (line 18) and execute a loop, where we compute new SPTEs and write them into the SPT tree (line 20). In the loop we first find the index of the SPT to which we will write the new SPTE (lines 22 and 24). Then we find the page index of this SPTE (line 25). Next, we execute function `compspte`, which computes a new SPTE from a given GPTE and if this GPTE is not a terminal one, then it also allocates a fresh SPT linked to the newly computed SPTE (line 26).

[7] In our algorithm this situation is actually impossible, because SPTs of our VP could not change since the time when the page fault intercept has happened. As a result, SPTs must contain a page fault. Yet, with the shared version of the SPT algorithm this situation is possible.

Further, we check whether the SPTE which is going to be overwritten points to another SPT (line 27). If it is the case, then we put the index of the detached SPT to the variable `detached_idx` (line 28). Note, that the situation when the present bit is set in the entry which we overwrite can happen only in the first iteration of the loop. On latter iterations we always overwrite an entry in the newly allocated SPT, where all entries are zeroed (this is guaranteed by function `compspte`).

Next, we distinguish two cases (line 30). In the first case we take care of the write protection for further dirty bit propagation to the guest. In this case we are in the last loop iteration, the `rw` bit is not set, all fetched GPTEs have a write permission enabled, and the dirty bit in the terminal GPTE is not set (function `walkguest` for this case does not set a dirty bit, because the intercepted request is a read request, see Section 9.3.4). We mark the terminal SPTE non-writable, which guarantees that the first write access through this SPTE will be intercepted (line 39). All complete walks though the newly written SPTE are added to the set `vp->walks` (line 31)[8]. This guarantees that invariant *inv-complete-walks* is maintained. In case the ASID of our VP is valid, we also have to update the set `pls->walks` respectively (line 34) to maintain invariant *inv-pls-walks*. Invariant *inv-htlb-walks* is maintained, since we know that the hardware TLB does not contain any walks with the faulty virtual PFN.

In the second case we simply overwrite the old SPTE with the newly computed one (line 54). If we are overwriting a non-terminal SPTE (line 41), we first call the ghost function `mark_unreachable_subtree` (line 42), which resets the *re* bit in all detached SPTs and removes all the walks through these SPTs from `vp->walks` and `pls->walks`. Analogously, to the MOVE to $CR3$ handler (see Section 9.5.2), when we mark detached SPTs unreachable we rely on the uniqueness of a “parent” in the SPT tree, which is guaranteed by *inv-reachable-child*. After that we call the ghost function `mark_reachable` (line 43), which sets the *re* bit for the newly attached SPTE and adds the walks through this SPTE to sets `vp->walks` and `pls->walks` (for the body of this function consult the sources). In case we are overwriting a terminal SPTE, we update the set `vp->walks` to remove old complete walks with the faulty virtual PFN and to include the walks over the new SPTE (line 45). If our VP has a valid ASID, then we also update the set `pls->walks` respectively (line 50). Invariant *inv-htlb-walks* is maintained, since we know that the hardware TLB does not contain any walks with the faulty virtual PFN.

In all cases invariant *inv-vtlb-walks* holds, because the virtual TLB already contains all walks, which we are adding to `vp->walks` (this is ensured by function `walkguest`). Invariants *inv-complete-walks* and *inv-partial-walks* are maintained, because we always remove the walks over the detached SPTs and add the walks over the newly attached ones from/to the set `vp->walks`.

Finally, we perform reclaiming of the detached subtree (line 59). Since detaching of a subtree can happen only once (on the first loop iteration), we can be sure that `detached_idx` is never overwritten after it is set. Hence, all detached SPTs are reclaimed afterwards.

[8]For the way how to construct the set of complete walks for a given SPTE see Definition 8.32.

In VCC we perform all updates of `vp->walks` *and* `pls->walks` *in dedicated ghost functions, rather than directly in the page fault handler. Moreover, we split the PF handler (as well as other implementation functions) into a number of blocks, to make verification easier for VCC. Updates of the ghost state in VCC we normally perform after the update of an implementation field, rather than before that (as we do in this chapter). This does not break the soundness of verification, because scheduling of the ghost code (before or after the implementation statement) can be considered benign. Note also, that a number of invariants local to a single VP (as e.g., inv-complete-walks, inv-partial-walks, inv-reachable-root, inv-reachable-child, and inv-vtlb-walks) do not have to hold all the time, but are strictly required to hold only when this VP is being executed on a host processor (see Section 10.4). Hence, in VCC we sometimes disable these invariants (by unwrapping the VP configuration) in the middle of the function and show that they are again maintained in the end of the function (i.e., before we execute a VMRUN statement). Yet, in the proof sketch presented in this chapter all the invariants are maintained after every implementation step of the C machine, with the ghost code being executed before the implementation statement.*

CHAPTER 10

Verification of the SPT Algorithm in VCC

10.1 The Verifying C Compiler

10.2 Modelling Hardware

10.3 Shadow Page Tables

10.4 Virtualization Correctness

10.5 Virtual Hardware Simulation

10.6 Hardware Thread

10.7 Verification Statistics

VCC [Mic12a] is a verifier for concurrent C code [CDH+09] which is being developed at Microsoft and which was used as a proof tool in the Verisoft XT project, aimed at the formal verification of industrial software including the Microsoft's hypervisor Hyper-V [Mic12b, LS09]. VCC supports adding annotations to the C code of a program, which includes pre- and postconditions, loop and type invariants, and ghost code. The features of VCC and its focus on verification of concurrent code make VCC an ideal instrument for implementation of the C-IL + HW + Ghost semantics and for verification of hypervisor code. We used VCC as the tool for the formal verification of our TLB virtualization algorithm.

In this chapter we give an overview of VCC and discuss the key aspects of verification, which include modelling of the hardware component of a thread introduced in Chapter 7, modelling of the virtual hardware state in VCC, and simulation of steps of the virtual hardware. Code snippets which we present in this chapter contain only a part of the annotations, which is necessary for understanding these crucial aspects.

The formal verification work has started in the frame of the Verisoft XT project [The12] and was completed before the formal C-IL + Hardware semantics was developed. As a result, the hardware component of a thread and the model for the guest virtual machines which we used in VCC verification is simpler than the one introduced in Chapter 7 (e.g., we haven't modelled SBs and memory request/result buffers there). Nevertheless, we believe that these differences do not produce any additional obligations on the hypervisor code itself and only reduce the number of unintercepted hardware steps, for which we can show correct virtualization in VCC (in VCC we have only proven correct virtualization for all MMU steps, namely walk creation, walk extension, and setting of A/D bits). The paper-and-pencil proof for all of these steps, including the ones which were not proven in VCC, is given in Theorem 8.3. We plan to adapt our formal VCC proofs[1] so that they adhere to the paper-and-pencil verification presented in this thesis as a part of the future work.

10.1 The Verifying C Compiler

VCC first translates an annotated C program into BoogiePL [DRL05], an intermediate language for verification. A BoogiePL program is further translated to logical formulas using the Boogie verification condition generator [ByECD+06]. These logical formulas are then passed to an automated SMT solver Z3 [dMB08] to check their validity.

VCC provides a number of features which are central to our methodology:

- it provides the ghost state, similar to the ghost state of the C-IL + Ghost semantics introduced in Chapter 5; moreover, we use the VCC ghost state to store the state of the hardware component of a thread from C-IL + HW semantics[2],
- it provides two-state object invariants (i.e., invariants that not only talk about the state of the object, but also constrain its transitions), which we use to express the transition system of the virtual hardware,
- it provides the ghost code, which we use for maintaining auxiliary information necessary for the TLB virtualization proof (Chapter 8); additionally it allows us to update the abstract state of the virtual hardware preserving the coupling invariant,
- it supports verification of programs with fine-grained concurrency which makes it possible to model atomic steps of the hardware component of a thread.

Formalization and documentation of VCC semantics, as well as the formal soundness proof of VCC still remains as future work. Yet, we believe that there should exist a simulation proof between a program executed and verified in VCC semantics and the same program executed in C-IL + Ghost or C-IL + HW + Ghost semantics (depending on whether the hardware component of a thread is modelled or not).

[1]The annotated sources of the verified SPT algorithm (including the models introduced in this chapter) can be found at http://www-wjp.cs.uni-saarland.de/publikationen/sources/kov12_sources.zip.

[2]For reasons why we use the ghost state to model the hardware component of a thread refer to Section 10.2.

10.1.1 Memory Model

The memory state of type-safe languages like *C#* and Java is defined as a collection of typed objects, which can not overlap. Moreover, one object can not contain another object as a member, it can have only a pointer to another objects. As a result, aliasing can occur only through two pointers pointing to the same object. This typed memory model allows a convenient logical representation of a program state, which is then defined as a mapping from an object and its fields to a value.

In contrast to that, the C-IL semantics from Chapter 5 as well as the regular C [ISO99] considers a flat untyped byte-addressable memory model. "Objects" in C can overlap arbitrarily (w.r.t to the object alignment) and there is no strict distinction between objects and their fields. A pointer is allowed to point to a field of a struct, and any struct can contain another struct as a member. Hence, two objects in C are disjoint only if they occupy disjoint memory regions. The whole concepts of types and objects in C is used to merely give a way of interpreting a chunk of memory, rather than to provide a self-contained abstraction.

When doing program verification it is much more efficient to work with typed memory objects, rather than with the flat untyped memory. For instance, it makes framing axioms much simpler by ensuring that if a single object gets updated, the other objects stay unchanged. For this reason VCC considers a typed object oriented memory model on top of the flat C memory [CMTS09, BM11]. Pointers to structs are interpreted as pointers to (implicitly) non-overlapping objects with disjoint fields. The set of "valid" typed pointers to "real" objects is maintained by VCC in the ghost state of the program. For every memory update VCC finds a respective "real" object, which has to be modified. If it cannot find such an object, then verification fails.

The difference of the VCC memory model compared to *C#* or Java memory model is that a struct is allowed to be a member of another struct. Hence, it two objects overlap, then one has to be a member of the other.

Soundness and completeness of the typed memory model on top of the flat C memory model was shown in [CMTS09].

10.1.2 Objects, Invariants, and Ownership

An object in VCC is an instance of a structured type. In each state each object is classified as *open* or *closed* and has a unique *owner*. In contrast to the ownership model used in this thesis (see Section 5.3.1), VCC has an object-oriented ownership model[3]. Any object can be owned by a thread, owned by another object, or can be not owned by anyone. Only threads can own open objects, and only closed objects can own other objects.

VCC allows each object to be annotated with 1-state and 2-state invariants. The first ones define a property which has to hold in every state of the program. The second ones specify how the state of an object may change (in a single atomic transition). The invariant of an object is only required to hold in transitions that begin or end with the object being closed.

[3]We believe that the object-oriented ownership can be translated into byte-wise ownership, like the one used in this thesis or alike.

VCC uses a modular approach to verification [CMST10]: when verifying the code performing a memory write, it checks only invariants of the object being updated. A transition which maintains invariants of all objects which are modified is called *legal*. VCC allows object invariants to mention arbitrary parts of the state. To make modular verification under this condition sound VCC performs an *admissibility* check for all invariants of all objects: an object invariant is said to be *admissible* iff it is preserved by all legal transitions that do not update the object itself. The admissibility check is performed once for every type definition and does not require looking at the program code (besides the program type definitions being checked).

Fields of an object which do not have the **volatile** type qualifier are considered to be sequential. All volatile fields are considered to be shared. The value of a sequential field can change only when the object is open and is owned by a thread. The value of a volatile field can change anytime, if this transition satisfies the 2-state invariant of the object. A thread can either write a field of an owned, open object or can write a shared field of an object which is known to be closed (only if an object is closed its volatile fields are really "shared"; volatile fields of an open object are treated the same way as regular sequential fields).

A thread is allowed to open (i.e. *unwrap*) a closed object which it owns. After performing desired updates of the unwrapped object the thread can *wrap* the object back to the closed state. Invariants of this object are checked only at the time when the object gets wrapped. As a result, 2-state invariants are meaningless for the fields of sequential objects.

Nevertheless, sometimes we want to make sure that sequential fields of objects change in a certain way when the object gets closed (i.e., specify pre- and postconditions on a state before the object was open and on a state when the object gets closed). To solve this problem we can add a ghost volatile copy of sequential data in the object and add a coupling invariant for this data, which has to hold when the object is closed. This volatile copy of the data has to reside in the part of the object which always stays closed. VCC supports splitting of a given object into parts called *groups*, which can be treated as different objects. Now we can state 2-state invariants on the volatile copy of the data located in a closed group inside the object. These invariants restrict the updates of the volatile copy of the data. When the object gets closed, the coupling invariant between the volatile and the sequential data guarantees that the changes to the sequential data were done with the same restrictions, as apply to the volatile copy of the data.

10.1.3 Claims

Object invariants are required to hold only when an object is closed. When a thread owns an object it can guarantee that it stays closed. Yet, shared objects are often not owned by a thread trying to access it. This is, for instance, a typical case when dealing with synchronisation objects like locks [HL09]. To capture information on closed, shared objects, VCC provides so-called *claims*.

A claim is a ghost object that stores a reference to its claimed object and has the invariant that this object will stay closed as long as the claim is not destroyed. To guarantee that the claimed object really stays closed, VCC adds a ghost field to it that counts the number of currently active claims on that

object. Every object has an invariant which prevents it from opening the object if this counter is greater than zero.

In addition to guaranteeing that an object stays closed, a claim may state certain properties of the system state. More precisely, a property stated by a claim has to hold initially when the claim is created and has to be stable under changes to other objects (in this sense it doesn't differ from admissibility or regular objects). For example, a claim may state a property which holds initially and which is guaranteed to be maintained by a 2-state invariant of the claimed object.

10.1.4 Atomic Updates

If a thread can not open the data being accessed (i.e., it is either not owned by a thread or has a claim counter greater than zero), the only way to perform this access is inside an *atomic block*. Any atomic block represents a single transition of the state. VCC uses atomic blocks to distinguish places in a thread execution where other threads may interfere. This allows VCC to perform sequential verification of code in between atomic blocks and to consider other threads only in the beginning of these blocks.

Each atomic block requires a claim to the updated object as well as the object itself to be passed as a ghost parameter[4]. This is necessary to guarantee that the updated object will not be opened by other threads interfering with the one being verified.

In the beginning of the atomic block VCC havocs information about the shared state and about the sequential part of the state which is not owned by a verified thread, over-approximating the interference of an arbitrary number of steps of other threads. At the end of the atomic block VCC checks whether the invariants of the updated objects are maintained. All knowledge required to perform this check is derived from the sequential state, which includes the claimed properties of any claim passed to an atomic block.

An atomic block may contain any arbitrary number of ghost statements (including ghost updates), but only one implementation statement. Moreover, this statement has to be consistent with atomic operations provided by the underlying architecture (i.e., it has to compile into an instruction performing an atomic memory update). To comply with our program safety for C-IL semantics (see Section 5.3.3) one has to additionally guarantee that all memory updates inside atomic blocks (i.e., all updates of volatile data) are compiled into locked writes or atomic compare-exchanges instructions[5]. To weaken this restriction one can consider a more sophisticated SB reduction theorem [CS10], which requires less flushing of SB and which uses less memory fences to guarantee sequential consistency of the shared memory. Nevertheless, VCC currently does not perform a check for memory fences to be inserted correctly though there are plans for extending it with this feature. Currently, one either has to require all volatile updates to be compiled into SB-flushing instructions, or one has to perform a manual check for store fences inserted correctly w.r.t the chosen SB reduction strategy.

[4]Atomic blocks can actually take an arbitrary number of claims and objects, depending on the number of objects being updated inside the block.

[5]Another option is to insert memory fences (draining the store buffer) after every update of the volatile data, though we didn't consider this option in our store buffer reduction proof.

An implementation update being executed in the atomic block can be either a regular write to a volatile field or involve an execution of a compiler intrinsic or an external assembly function (e.g., an atomic compare-exchange operation). Since the choice of available compiler intrinsics is platform specific, VCC does not have support for built-in intrinsics. Hence, we have to manually specify the effect of such an intrinsic/assembly function in VCC by writing a ghost body of the function, which performs the same update of the C state as the intrinsic does (for the semantics of an atomic compare exchange operation refer to Section 5.1.8).

Note, that memory accesses with atomic blocks can also be performed in the ghost code. Ghost atomic blocks obviously do not require any SB flushing policy and do not enforce any restrictions on the compiler, but are used for verification of programs which have shared ghost state (e.g., used for hardware modelling).

10.1.5 Approvals

Admissibility checks force restrictions on the part of the state which can be fixed in an object invariant. The check succeeds only if a given invariant is stable under legal updates of all other objects. As a result, we cannot simply write an invariant which talks about the state of another object, if this object is not owned by the current one. To solve this problem, we have to add an additional invariant to the observable object, which would explicitly require to check the invariant of the observer if the object gets changed. We call such an invariant *approval* and say that the observer *approves* the observable object.

Approval acts as a technique for semantic subclassing of concurrency in VCC. An object which is not approved by anyone corresponds to a "closed" object (w.r.t concurrency), meaning that clients cannot strengthen invariants of this object. An object which is approved by a client allows the client to effectively strengthen its invariants to the extent allowed by the approval (i.e., restrict only those fields which are approved).

Approvals are very helpful in the design of a concurrent algorithm. For instance, one can use approvals to make sure that a given object behaves in a certain way described by an abstract specification data type (e.g., the abstract hardware model).

Note, that approval is a 2-state invariant and it works only when the object being approved stays closed (otherwise its invariants are not checked at the update, but are only checked at the time when the object gets closed). Hence, approvals cannot be stated for sequential fields. In order to overcome this problem (e.g., if we want to restrict values of sequential fields of an observable object), we can add a ghost volatile copy (residing in the closed group inside the object) of sequential data in the object and add a coupling invariant for this data, which has to hold when the object is closed. Now we can add approvals on the volatile copy of the data. The observer in this case can restrict the values of sequential fields of the client object, when this object is in a closed state. When verifying our SPT algorithm we often use this trick to state properties over multiple objects (e.g., when we state disjointness of ASIDs and ASID generations of different VPs).

Approval of a volatile field of an object by a thread that owns the object has the effect of making the field sequential from the standpoint of the owning

thread, with the exception that it must still update the field in atomic blocks. This is helpful e.g., in the approval scenario for sequential fields described in the previous paragraph: the volatile copy of the data has to be approved by the owning thread in order for the coupling invariant between volatile and sequential data to hold at the time when the object gets closed.

10.1.6 Scheduling

The C-IL + Ghost and the C-IL + HW + Ghost semantics introduced in chapters 5 and 7 consider fine-grained scheduling (modular ghost steps), where threads may interleave in between any implementation steps. VCC, however, considers an I/O-block (coarse-grained) scheduling, which switches a thread only when it is about to execute an atomic block (i.e., accessing shared data). Yet, one can prove a theorem justifying this approach analogously to the hardware reordering theorem from Section 5.4.3. More precisely, for any fine-grained execution of a safe program there should exists coarse grained execution resulting in the same state. It follows, that any program which can go wrong under an arbitrary scheduler, can also go wrong under an I/O-block scheduler. The proof of such a theorem for a simplified language can be found in [CMST09].

10.2 Modelling Hardware

To verify the program where software steps are interleaved with the steps of the hardware component of a thread (which we later also refer as the host hardware state), we extend the program code with the (ghost) hardware thread, which non-deterministically updates the state of the host hardware and the memory of the program (w.r.t to the allowed hardware transitions defined in Section 7.2.2). We locate the host hardware state in the ghost memory, but we do allow limited information flow between some of its fields (e.g., registers and TLB) and the memory of the concrete program. This is done for lack of a dedicated hybrid type capturing implementation state other than the main memory. As a result, the ghost code implementing the hardware thread does not comply with the safety requirements for the ghost code stated in Section 6.6, and is treated as "hybrid" code, modelling the hardware actions of the C-IL + HW semantics. In case if data flow between ghost and implementation state occurs, VCC throws a warning rather than an error. By examining these warning we can ensure that the data flow occurs only to/from the "hybrid" state and that the real ghost code satisfies all the restrictions.

The hardware transition relation of both the host and the virtual hardware is formulated as a 2-state invariant of the corresponding hardware data structure.

The state of the virtual hardware (excluding the memory) is also located in the ghost memory. The memory of the VM is abstracted from the portions of the C memory allocated to the machine w.r.t the function *gpa2hpa*. To ensure that every update of the virtual memory is justified by the transition relation of the VM, we model the memory of the VM as volatile data approved by the virtual hardware.

Updates of the virtual hardware, simulating steps of the VM, are performed by the ghost code in atomic blocks, guaranteeing that the transition relation and the coupling invariant are maintained by every update. When a step of the virtual hardware involves accessing the implementation memory (e.g., fetching of a GPTE by the *#PF* handler), the update to the virtual configuration is done in the same atomic block as the memory access. This allows to simulate a step of the VM on the virtual memory abstracted from the implementation memory.

10.2.1 Locating Invariants

The correctness (coupling) invariants from Section 8.3 are specified as 1-state invariants over data structures of the hypervisor and over the simulated virtual hardware. More precisely, invariants specific to a single virtual processor are included in the invariant of the implementation data structure of type `Vp` and invariants establishing properties over the VPs altogether (as e.g., invariant *distinct_asids*) are specified in data structures of types `Guest` and `Gm`.

Properties of the overall system which have to be maintained by software and hardware steps are specified in the so called *hardware interface*. For instance, it specifies for each host processor a map $walks[i]$ (Section 8.3.5), which contains all walks possibly residing in the HTLB of that processor, and states invariant *inv-htlb-walks* (Invariant 8.23). The hardware interface is purely ghost, since it is only used for specification rather than to implement concrete data structures or hardware components. To check that the invariants of the hardware interface are maintained by all possible hardware transitions, we have to explicitly invoke each of them in the hardware thread. For more information on how we partition the coupling invariants between different data structures see Section 10.4.

On Figure 10.1 we give a top-level overview on the ownership and approval scenario for implementation and specification data structures, which we used for verification of the SPT algorithm. An arrow from an object to another object means that the second object is owned or approved by the first one. Filled objects represent the volatile (i.e., shared) part of the state, which is not thread-approved (thread approval makes the volatile data to behave like sequential data).

10.2.2 Host Hardware

Host Processor

The state of a single host processor[6] is modeled using the struct type `Processor` (Listing 10.1).

All fields of a host processor are approved by the owning thread, which makes them sequential in their nature, though still allowing only atomic updates to be performed on these fields. (We make these sequential fields volatile to allow them being controlled by the 2-state transition invariant of the hardware.) A host processor is always owned by a thread which runs on

[6]The state of a processor includes the state of all processor local components (i.e., memory core, TLB, and SB). As mentioned in the beginning of the chapter, our hardware model in VCC is slightly simpler then the one used in the paper-and-pencil proofs in this thesis (SB and memory request/result buffers are currently missing).

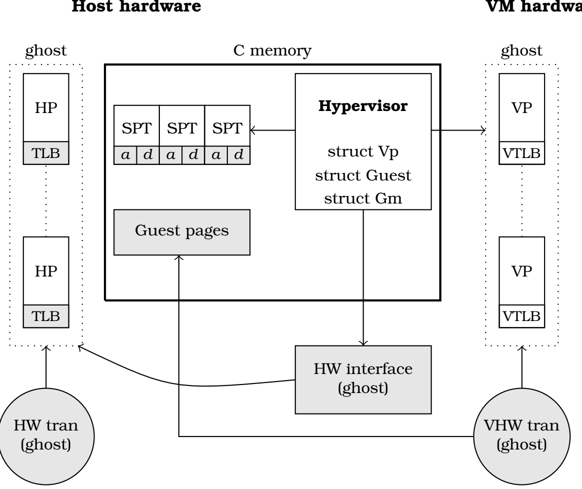

Figure 10.1: Approval and ownership scenario for the SPT algorithm.

it (either by a hypervisor thread or by a hardware thread). This is sound, because the hardware component of a thread can not perform steps on its own if a processor is running in the hypervisor mode (see Section 7.2). At the VMRUN statements the ownership of a processor should be passed from a hypervisor thread to a hardware thread.

Additionally to thread approval, all fields of the host processor are approved by the hardware interface, which couples the state of the hardware with the

```
1  _(ghost typedef struct _Processor {
2   Pid i; // Processor id
3   volatile Asid asid; // processor ASID
4   volatile Ppfn CR3; // Pfn field of the CR3 register
5   volatile bool tlb[AbsWalk]; // TLB (a map of walks)
6   Hardware *h; // pointer to the hardware container
7   Hwinterface *hwi; // pointer to HW interface
8   _(invariant \approves(h, tlb, CR3, asid)) // approval by hardware
9   _(invariant \approves(hwi, tlb, CR3, asid)) // approval by HWI
10  _(invariant \approves(\this->\owner, tlb, asid, CR3)) // thread approval
11  _(invaraint \on_unwrap(\this, \false) // remains always closed
12 } Processor;)
13 _(ghost typedef struct _Hardware {
14  Processor p[HP_CNT]; // array of processors
15  volatile Pid i; // index of acting processor
16  volatile Action act; // type of action
17  volatile AbsWalk w; // TLB walk for the action
18  _(invariant \forall Pid i; i < HP_CNT ==>
19    p.h == \this && p.h.\closed) // back link for approvals
20  _(invariant proc_unch(p) ||
21     act == TLB_SET_AD && tlb_setad(p, i, w, old(read_pte(w))) ||
22     act == CORE_INVLPGA && core_invlpga(p, i) || ... ) // transition relation
23  _(invaraint \on_unwrap(\this, \false) // remains always closed
24 } Hardware;)
```

Listing 10.1: Host hardware configuration.

hypervisor data structures.

Host Hardware Transition Relation

The (ghost) data structure `Hardware` (Listing 10.1) encapsulates all processors and defines via 2-state invariants all valid transitions of the hardware component of a thread.

The parameters of the next hardware transition to be performed are specified by variables i, act, and w, where i identifies the acting processor, act the action type, and w the walk targeted by the action in case of a TLB transition. In the hardware thread these variables allow us to explicitly go over all possible transitions of the hardware component.

Note, that restricting transitions of host processors by the transition relation stated in the `Hardware` data structure is not strictly necessary, because all possible hardware steps are explicitly performed in the hardware thread. Yet, having a hardware transition relation specified as a 2-state invariant makes it easier to make sure that we have the desired semantics of these steps.

We locate the configuration of the host hardware as a ghost field of the guest manager (Section 9.1.2). Additionally, in the guest manager we maintain a map `hp2vp` from the ID of the host processor and its ASID to a VP configuration, which has the same active ASID and is assigned to this host processor.

```
1 typedef struct _Gm {
2   ...
3   _(ghost Hardware h) // host hardware component
4   _(ghost volatile Vp *hp2vp[Pid][ASID]) // pointers to assgined VPs
5   _(invariant \mine(h)) // ownership of hardware
6   _(invaraint \on_unwrap(\this, \false) // remains always closed
7  } Gm;
```

The map `hp2vp` is well defined only for valid ASIDs. For host processors running in virtualization mode this mapping implements the function $hp2vp_c$ from Section 8.2.2.

Hardware Interface

The hardware interface (Listing 10.2) is a container for properties which relate the state of the hardware component of a thread with the hypervisor data structures.

The hardware interface stores the following information:

- a collection of maps of reachable walks for all processors[7],
- the set of valid ASIDs for every processor. This set is defined according to Definition 8.11,
- the set of valid values of the CR3 registers for all processors. For every VP scheduled to run on a given hardware processor with some ASID this map returns the address of the top-level SPT,
- pointers to PLSes of all host processors. These pointers are used to bound the current ASID of a given processor with the maximal ASID

[7] In our paper-and-pencil verification we store individual maps for every processor in the PLS data structure. Here we collect all these maps in the hardware interface.

```
_(ghost typedef struct _Hwinterface {
  volatile bool walks[Pid][AbsWalk]; // overapproximation of HTLB
  volatile bool valid_asid[Pid][Asid]; // map of valid ASIDs
  volatile Ppfn gCR3[Pid][ASID]; // CR3 registers used by VPs
  Pls *pls[Pid]; // map of pointers to PLSes
  Processor *p[HP_CNT]; // pointers to processors
  \object gm; // pointer to top-level data strucure of the hypervisor
  _(invariant \approves(gm, rwalks, valid, gCR3)) // approval by GM
  _(invariant \forall Pid i; i < HP_CNT ==>
    p->hwi == \this && p->\closed) // back link for approvals
  // fixing the content of host TLBs
  _(invariant \forall AbsWalk w; Pid i; i < HP_CNT && p[i]->tlb[w] &&
      valid_asid[i][w.asid] ==> rwalks[i][w])
  // running ASID has to be valid
  _(invariant \forall Pid i; i < HP_CNT ==> !h.p[i].asid ||
      valid[i][h.p[i].asid])
  _(invariant \forall AbsWalk w; Pid i; i < HP_CNT ==>
      (p[i]->tlb[w] ==> w.asid <= pls[i]->max_asid))
  // value of CR3 registers in guest mode
 _(invariant \forall Pid i; i < HP_CNT ==> (p[i]->asid != 0 ==>
    p[i]->CR3 == gCR3[i][p[i]->asid]))
 _(invariant \on_unwrap(\this, \false))
} Hwinterface;)
```

Listing 10.2: Hardware interface.

allocated to this processor in the respective PLS, which is necessary for verification of the TLB lazy flushing mechanism (see Section 9.4).

The top-level data structure which approves all fields of the hardware interface in our case is a guest manager (Section 9.1.2), where we also locate the hardware interface itself as a ghost field.

```
typedef struct _Gm {
  ...
  _(ghost Hwinterface hwi;) // pointer to HWI
  _(invariant \mine(hwi) && hwi->gm == \this) // ownership of HWI
 } Gm;
```

10.2.3 Virtual Hardware

Virtual Processor

The state of a single abstract VP is modelled by a struct type `VProcessor` (Listing 10.3), similar to the state of a host processor. The difference to the host processor model is that in the abstract VP configuration we don't need to have an ASID register, and we don't have approval by a hardware interface.

For every abstract virtual processor (which is an instance of type `VProcessor`) there is a corresponding implementation data structure of type `Vp`, which stores the configuration of this processor and owns the state of the abstract virtual processor. When a hypervisor thread is running on some host processor it always owns configurations of all VPs assigned to this processor. Hence, it also owns all abstract states of these VPs and can modify them the way it likes (w.r.t to the transition relation of the virtual hardware), so that the coupling invariant is always maintained. At the VMRUN statement

```
1  _(ghost typedef struct _VProcessor {
2    Pid i; // Processor id
3    volatile Ppfn CR3; // Pfn field of the CR3 register
4    volatile bool tlb[AbsWalk]; // TLB (a map of walks)
5    Hardware *h; // pointer to hardware container
6    _(invariant \approves(h, tlb, CR3) // approval by hardware
7    _(invariant \approves(\this->\owner, tlb, CR3)) // thread approval
8    _(invariant \on_onwrap(\this, \false)) // remains always closed
9  } VProcessor;)
10 _(ghost typedef struct _VHardware {
11   VProcessor p[VP_CNT]; // map of virtual processors
12   Ppfn gpa2hpa[Ppfn]; // memory translation
13   volatile Pid i; // index of acting processor
14   volatile Action act; // type of action
15   volatile AbsWalk w; // TLB walk for the action
16   _(invariant \forall Pid i; i < VP_CNT ==>
17     p.h == \this && p.h.\closed) // back link for approvals
18   _(invariant \forall Ppfn a; gpa2hpa[a] ==>
19       ((Gpt *)page(gpa2hpa, a))->h == \this &&
20       ((Gpt *)page(gpa2hpa, a))->\closed) // back link for approvals
21   _(invariant p_unch(p) && m_unch(abs_m(gpa2hpa)) ||
22       act == TLB_SET_AD && tlb_setad(p, i, w, old(read_pte(w,gpa2hpa)))
23         && m_upd(abs_m(gpa2hpa), w) ||
24       act == CORE_INVLPGA && core_invlpga(p, i)
25         && m_unch(abs_m(gpa2hpa)) || ... ) // transition relation
26   _(invariant \on_onwrap(\this, \false)) // remains always closed
27 } VHardware;)
```

Listing 10.3: Virtual hardware configuration.

the ownership of these VPs has to be transfered to the hardware thread, analogously to the ownership of the host processor state.

Virtual Hardware Transition Relation

Analogously to the host hardware model, we introduce a data structure `VHardware` which approves individual transitions of every virtual processor of a given VM (Listing 10.3).

Meta variables i, act, and w are now used to choose a certain step we want to simulate at some point in the hypervisor execution or in the execution of a hardware thread, if the hardware step involves simulation of a VM step (see Section 8.4.1).

The main difference of the virtual hardware w.r.t the host hardware is the treatment of memory. For the host hardware (which is a model of the hardware component of a C thread introduced in Chapter 7), we do not need to explicitly state framing for the memory. This is due to the fact that all (memory-writing) steps of the host hardware are modelled explicitly in the hardware thread, and there we guarantee that these steps perform only valid memory updates of the C memory according to the C-IL + HW semantics.

In contrast to that, the memory of the VM is abstracted from a dedicated portion of the C memory. Since this memory region can be also updated by the hypervisor, we have to make sure that these memory updates comply with the semantics of the virtual hardware machines. For this we require the virtual hardware to approve all memory pages allocated to this VM. We obtain

those pages using the map *gpa2hpa* translating guest physical addresses to host physical addresses (which is a copy of the *gpa2hpa* map from the guest configuration). An arbitrary memory page of the VM is modeled as a GPT consisting of guest PTEs (GPTEs).

```
1 typedef struct _Gpt {
2   volatile Pte e[512]; // array of PTEs
3   _(ghost VHardware *h;) // pointer to the virtual hardware
4   _(invariant \approves(h, e)) // approval by virtual hardware
5   _(invariant \on_onwrap(\this, \false)) // remains always closed
6 } Gpt;
```

We locate the virtual hardware configuration as a ghost field of the partition configuration (Section 9.1.2).

```
1 typedef struct _Guest {
2   ...
3   _(ghost VHardware g;) // pointer to the virtual hardware
4   _(invariant \mine(g)) // ownership of the virtual hardware
5   _(invariant \on_onwrap(\this, \false)) // remains always closed
6 } Guest;
```

10.3 Shadow Page Table

Shadow page tables are implemented as structs, consisting of 512 volatile unsigned integers 64-bit long (Section 9.1.4). In our algorithm sharing of SPTs is not supported and every SPT is owned by the VP to which it is assigned, meaning that only this VP can modify this SPT. At the same time, we further plan to extend our verification for a version of the SPT algorithm with sharing. In that algorithm a single SPT which is owned by some VP, can be also written by host TLBs of other processors. Hence, a hypervisor thread operating with this VP might race with other TLBs when accessing this SPT. To handle this situation, we introduce a ghost copy for every SPTE and make this copy approved by the VP, which owns the SPT. Further, we add an invariant saying that all the bits of the original SPTE, except A and D bits, are always equal to the same bits of the ghost copy of this SPTE. As a result, we can be sure that TLBs would never modify any bits of SPTEs, except A and D bits.

```
1 typedef struct _Spt {
2   volatile Pte e[512]; // array of PTEs
3   _(ghost volatile Pte ge[uint];) // owner-approved copies of PTEs
4   _(invariant \approves(\this->\owner, ge)) // owner approval
5   _(invariant sptes_eq_except_a_and_d(e, ge)) // relation between two copies of PTEs
6   _(invariant \on_onwrap(\this, \false)) // remains always closed
7 } Spt;
```

10.4 Virtualization Correctness

The way how we stated and proved virtualization correctness in Chapter 8 and Chapter 9 is nice for a paper-and-pencil proof, but is not the best one for a modular C verifier. In VCC after a step of the machine we want to check as few invariants as possible and do not want to explicitly maintain all parts of the coupling invariant after every step of the machine. Hence, we apply the following modifications to our verification approach:

1. we observe, that in our main correctness theorem (Theorem 8.1), the coupling invariant is not mentioned at all. It is used not as a part of the correctness criteria, but rather as an auxiliary invariant which is necessary to derive the main property (i.e., equality of traces). Hence, the coupling invariant is not strictly required to hold all the time,
2. we split all invariants from Chapter 8 into two sets: one set which gathers "global" properties over all VPs (this includes e.g., *inv-memory-coupling* and *inv-distinct-asids*) and another one which fixes "local" properties of VPs (this includes local version of *inv-complete-walks*, *inv-partial-walks*, *inv-reachable-root*, *inv-reachable-child*, and *inv-vtlb-walks*),
3. we also observe, that for correct simulation of the steps of the hardware component (Theorem 8.3) we don't need to require all parts of the coupling invariant, but only need the "global" invariants, and the "local" ones for the running VP. Moreover, since these steps can possibly modify only SPTs belonging to the running VP, "local" invariants of other VPs can not be broken,
4. when verifying an intercept handler, we require as a precondition that the "global" and the "local" part of the coupling invariant for the handled VP holds. We show that for all possible scheduling, the "global" part of the invariant is maintained after every C step and the "local" part is maintained in the end of the handler (though it could possibly break in the middle of the function).
5. we show that all updates of the state of the virtual hardware (either by the hardware component or by an intercept handler) comply with the transition relation (i.e., form the sequence of valid guest steps),
6. we show that if "local" invariant of some VP holds, and the configuration of this VP, as well as all SPTs and PTIs allocated to this VP, remain unchanged in a machine step, then this "local" invariant also holds after the step (this is done by the VCC admissibility check).

Finally, we can be sure that (i) the "global" part of the coupling invariant holds for all steps of the machine (including the steps of the hardware component), (ii) the "local" invariant of a handled VP holds in the end of the intercept handler, and (iii) this local invariant is maintained afterwards if no other thread is modifying its state, SPTs, and PTIs (which we guarantee to be true, because of the ownership of these data structures by a thread).

To make sure that the "global" properties from the coupling invariant always hold, we put them into data structures which are always closed (i.e., the hardware interface and the guest configuration). To be able to update ghost fields mentioned in these invariants, we make them volatile. For the implementation fields, which are sequential but are still mentioned in the invariants (as e.g., ASID field of the VP configuration) we introduce a ghost volatile copy, which we put in the closed object (see Section 10.1.2). We further make sure that every time when we use a sequential copy of this field its value is equal to the value of the volatile copy, which guarantees that invariant is preserved.

The "local" properties of a VP are stored in the VP configuration. VP owns all SPTs and PTIs assigned to it, as well as the state of the respective abstract VP. As a precondition to an intercept handler we require the VP configuration to be

wrapped, which means that it is owned by a thread and its “local” invariants hold. This implicitly gives the thread ownership of all objects owned by the VP itself. As a postcondition we guarantee that the VP is wrapped back and its state corresponds to the action performed by a handler (see Section 9.5 for the state of the abstract VP after the handler is executed).

When emulating the steps of the hardware component of a thread (Theorem 8.3) we require all VPs (including the running one) assigned to the host processor to be wrapped[8]. The ownership of these VPs has to be passed to the hardware thread from the hypervisor thread at the execution of the VMRUN statement. At the VMEXIT step the ownership of the VP has to be passed back to the hypervisor thread.

When we perform an update of the virtual hardware, we have to make sure that it is valid w.r.t to the transition relation. For this we make the state of the virtual hardware volatile, keep it in a closed object, and make it approved by another object, which contains the hardware transition relation (see Section 10.2.3). This guarantees that we simulate only valid steps of the virtual hardware (Section 10.5).

VCC admissibility check guarantees that the “local” invariant of a VP is stable under updates of other object. Note, that to state “local” invariants mentioning the set of host walks (`pls->walks` from Chapter 8 and `hwi->rwalks` from this Chapter), which is shared between different VPs, we have to make this set volatile and to add 2-state invariants restricting its transitions. For instance, we add an invariant which guarantees that any thread can remove walks from this set only if it owns some VP configuration belonging to the same host processor. Moreover, it can remove the walks only in the current ASID of this VP configuration. From the uniqueness of ASIDs (invariant *inv-distinct-asids*), we always know that no other VP can have the same valid ASID as our VP does. Hence, all possible updates of the set of walks by other threads would not break the invariant of our VP.

10.5 Virtual Hardware Simulation

As an example of a guest-memory accessing operation we consider the setting of A/D bits and performing a walk extension in the inner loop of the `walkguest` function (line 19 Listing 9.7). The version of this loop with (simplified) VCC annotations is given in Listing 10.4.

The simulation is done in the same atomic block, where the writing of the GPTE occurs and only if the compare-exchange operation was successful. As parameters to the atomic block we pass the following objects:

- GPT `gpt` on which we operate,
- pointer to abstract configuration `p` of the VP which is used to justify the access,
- virtual hardware configuration `guest->g` which contains the transition relation of the VM,

[8] It would be sufficient to require ownership only for the running VP. Nevertheless, requiring ownership of all VPs belonging to a given host processor is also sound.

```
1  ...
2  while (!cmp_result)
3    _(writes vp)
4    _(invariant \thread_local(vp) && \claims(gc, guest->g) && ...)
5    _(invariant guest->g.p[vp->id].tlb[WALK_PTES(vp->gwo, vpfn, res)])
6    ...
7  {
8    _(atomic gpt){
9      old_pte = gpt->e[px]; //fetching GPTE
10     ... // here simulate PF step
11   }
12   _(unwrap vp) // opening thread−local object
13   _(ghost VProcessor *p = &guest->g.p[vp->id];) // pointer to the abstract VP
14   _(atomic gpt, p, guest->g, gc){ // setting A and D bits
15     if (can_wextend(old_pte, rw, ex, us, res.level)) {
16       cmp_result = (old_pte == asm_cmpxchg(&gpt->e[px], old_pte,
17         (res.level == 1 && rw && READ_PTE_RW(old_pte))
18         ? SET_PTE_AD(old_pte): SET_PTE_A(old_pte)));
19       _(ghost if (cmp_result) { // fixing step parameters
20         guest->g.i = vp->id; // setting ID of the abstract VP
21         guest->g.act = TLB_SET_AD_WEXT; // choosing the type of a step
22         guest->g.w = WALK_PTES(vp->gwo, vpfn, res); // walk that will be extended
23         Pte pte = (res.level == 1 && rw && READ_PTE_RW(old_pte))
24           ? SET_PTE_AD(old_pte): SET_PTE_A(old_pte); // PTE for a walk extension
25         Rights r = ACCUM_RIGHTS(guest->g.w.r, old_pte); // permissions
26         if (res.level == 1 && !rw && !READ_PTE_D(res.pte[1]) &&
27           RW_SET(res.level))
28           r[RIGHT_WRITE] = 0; // restricting writes for dirty bit propagation
29         AbsWalk new_walk = WEXT(guest->g.w, pte, r); // extended walk
30         guest->g.p[vp->id].tlb[new_walk] = 1; // adding new walk to VTLB
31       })
32     } else // don’t do update if the entry is not present
33       cmp_result = 1;
34   }
35   _(wrap vp) // closing thread−local object
36 }
37 ...
```

Listing 10.4: Simulating step of the virtual hardware.

- a claim on the virtual hardware which guarantees that VM configuration is closed (it is not thread-local because one container is shared between all VPs belonging to a given VM).

To show simulation, we have to choose an appropriate action by writing the (volatile) meta-variables of the VM configuration. First, we choose the acting VP by writing its ID to the field `guest->g.i` (line 20). Then we choose the action of setting A/D bits and extending a walk (line 21). Note, that since we are not able to simulate two different actions in one atomic block, we had to extend our hardware transition relation with a step which performs both setting of A/D bits and walk extension in a single transition.

Further, we assign the walk we are going to extend to the field `guest->g.walk` (line 22). In order for the simulation to succeed, this walk has to be already present in the virtual TLB (i.e., has to be added to the VTLB on one of the previous iterations of the outer loop or has to be added to the VTLB before the loop execution has started). To ensure this we maintain loop invariants (on both inner and outer loops), which guarantee that the walk

constructed through the GPTEs already fetched to `res` is present in the VTLB (line 5).

After this, we calculate the value of the PTE which we are going to use for the extension (line 23). It should be equal to the value of the GPTE in the memory after the compare-exchange operation succeeds. Additionally, we calculate the permissions for the new walk. To do this, we first calculate the maximal possible rights through all fetched GPTEs, including the one fetched on this loop iteration (line 25)[9]. Further, we decide whether we need to restrict the write permission of the walk (line 28) for further dirty bit propagation (see Section 9.5.3). Finally, using the walk `guest->g.w`, the PTE, and the permissions for the extension we can calculate the new walk, which we add to the VTLB (line 30).

The invariants of the virtual hardware are checked automatically at the end of the atomic block, ensuring that a selected hardware step is performed accordingly to the transition relation. Since in this atomic block we operate only with the state of a single VP, VCC doesn't need to check the invariants of other VPs. The invariants of the hardware interface also are untouched here, because the set of the reachable walks remains unchanged.

10.6 Hardware Thread

The hardware thread consists of a number of ghost functions each performing a single step of the host hardware (see Section 7.2.2) in a single atomic block. Depending on the type of a hardware step we either have to respectively perform a step of the virtual hardware or to show that the coupling invariant is maintained without changing the state of the VMs (see Theorem 8.3).

For a step which does require an update of the virtual hardware configuration (i.e., all steps which are not TLB steps), the running VP performs exactly the same kind of a step as the host hardware does and the proof is trivial. Yet, the most complicated steps verification-wise are the ones where the state of the VMs remains unchanged. These steps include walk creation, walk extension, and setting of A/D bits by the host TLB.

As an example, we consider a step of setting A/D bits (Listing 10.5). The ghost function `mmu_step_setad` takes as an input pointer to the guest manager `gm`, the ID `hp_idx` of the host processor making a step, the walk which will be used for the step, and claim `gc` which guarantees that the guest manager always stays closed. This implies that the hardware container `gm->h` and the hardware interface `gm->hwi`, which are owned by the guest manager, also stay closed.

As a precondition, we require the claim to be valid, `hp_idx` to be less then the number of processors in the system, and all VP configurations assigned to this host processor to be wrapped (i.e., closed and owned by a thread). The ownership of VPs guarantees that no other thread at the same time will modify their configuration, SPTs, and PTIs.

As parameters to the atomic block (line 10) we pass the following objects:

[9]Note, that the "fetching" of a GPTE actually occurs at the time when the compare-exchange operation succeeds. Later in the function we add the written value of the GPTE to the array `res.pte` (line 28 in Listing 9.7).

```
1  _(ghost void mmu_step_setad(Gm *Gm, \claim gc, uint hp_idx, AbsWalk w)
2    _(requires \claims(gc, Gm->\closed) && \wrapped(gc))
3    _(requires hp_idx < HP_CNT)
4    _(requires \forall uint i, j; i < VP_CNT && j < GUEST_CNT ==>
5      (gm->guests[i].vp[j].hp_idx == hp_idx ==>
6        \wrapped(&gm->guests[i].vp[j])))
7  {
8    Processor *hp = gm->h.p[hp_idx]; // pointer to a host processor
9
10 _(ghost_atomic gm, gc, hp, (Spt *)(w.pfn << 12)) {
11     Spt *spt = (Spt *)(w.pfn << 12);
12     uint px = compute_idx(w.vpfn, w.l);
13     _(assume hp->tlb[w] && w.l != 0 && w.asid == hp->asid
14         && hp->asid > 0 && spt->e[px].p) // assuming guard
15     Vp* vp = gm->hp2vp[hp->id][hp->asid]; // getting the running VP
16     _(assert \inv(vp)) // asserting invariant of the running VP
17     _(begin_update) // start of update in the block
18     spt->e[px] = (w.l == 1 && w.r[rw] && spt->e[px].rw)
19         ? SET_AD(spt->e[px]) : SET_A(spt->e[px]); // performing a write
20     gm->h.id = hp_idx; // setting ID of the acting host processor
21     gm->h.act = TLB_SET_AD; // choosing the type of a step
22     gm->h.w = w; // choosing a walk for the step
23 }
```

Listing 10.5: Step of the (host) hardware component.

- a pointer to the guest manager and a claim, which guarantees that the guest manager stays closed,
- a pointer to the abstract configuration of the host processor performing a step, and
- a pointer to the SPT which we are going to modify.

Inside the atomic block we start with getting the pointer to the SPT (line 11) and the page index of the updated PTE (line 12). Then we assume a guard for the step (line 13). Next, we find the VP which is currently running on this host processor (line 15) and assert invariant of that VP (line 16). Then we use keyword `begin_update`, which tells VCC at which place to actually start the atomic action by havocing information over the shared (volatile) state. Then we perform a write to the SPT (line 19). Finally, we fix the parameters of the step by setting the meta variables of the hardware configuration (lines 20-22).

With the help of the invariant of the running VP and the hardware interface (particularly, *inv-pls-walks*, *inv-partial-walks*, and *inv-complete-walks*) VCC derives that the memory write is performed to an SPT owned by that VP. The write to the memory goes through, since the updated SPTE is volatile and we update only A and D bits from it (which means that the invariant of the SPT, which links SPTE with its owner-approved copy is maintained). All invariants of the updated and approving objects, including the invariants of the updated SPT, of the hardware container, and of the hardware interface, are checked by VCC automatically at the end of the atomic block.

CHAPTER 11

Summary and Future Work

Up to our knowledge, this thesis presents the first functional verification of a TLB virtualization algorithm, as well as the first verification of any kind against a realistic model of a modern hardware MMU. We have presented a formal model stack starting from an abstract hardware model up to the integrated semantics of C-IL + HW + Ghost, providing a framework for functional verification of the hypervisor code running in parallel with the guest code. Though we have applied this framework only to prove TLB virtualization, it can also be generalized for verification of the complete virtualization layer of the hypervisor, by instantiating the instruction automaton of the hardware machine and adding it to the hardware component of the C-IL + HW semantics (see below for details). We have implemented our framework inside an automatic C code verifier and have used it for the verification of a simple SPT algorithm, written in C.

Our implementation of the SPT algorithm contains ca. 700 lines of C code (including initialization of data structures which is not presented in this thesis) and ca. 4K lines of the annotations which include function contracts, loop invariants, data invariants, ghost code, and (proof) assertions. Roughly a third of annotations comprise function and block contracts and another third is ghost code for maintaining ghost fields, showing simulation, and running the hardware thread. The overall proof time is ca. 18 hours on one core of 2GHz Intel Core 2 Duo machine[1].

Finally, we outline the possible directions of future work.

[1] In our verification we used the second version of VCC from July 2011. Since then many changes have been made to VCC, which dramatically improved its performance. The major change was a new memory model introduced with the third version of VCC [BM11]. Certain technical adaptations have to be made to our VCC annotations to make the proofs run through with recent versions of VCC and making these adaptations remains a part of the planned future work. From our experience with the third version of VCC we believe that after the necessary adaptations are made we can improve the verification time by an order of magnitude and can decrease the annotation overhead roughly by half (mainly by decreasing the number of assertions in the code acting as "verification hints" for VCC).

- In the model stack presented in this thesis we use a simple ownership strategy, where we require all writes to shared data to be performed with atomic interlocked instructions. Cohen and Schirmer in [CS10] prove a store buffer reduction theorem for a much more elaborate ownership strategy, aimed at showing absence of triangular data races. Though they consider a quite general hardware model, its instantiation with our model is tedious because of the presence of MMUs as separate actors, which operate directly on the main memory, bypassing SBs. Hence, to replace the simple ownership discipline in our framework with the Cohen-Schirmer's ownership, one has to adapt the models from [CS10] to include MMUs, and to modify the proofs respectively. This work is currently in progress at the chair of Prof. Paul in Saarland University.
- In Section 3.3.2 we introduce an abstract MOESI protocol. One might prove, that our model simulates the concrete implementation of the shared memory with the MOESI cache-coherency protocol implemented in [Pau11].
- The TLB model presented in this thesis is lacking some widely-used features, such as support for large and global pages. One might extend our hardware model to support these features. Extension of the SPT algorithm is also needed in this case, because one has to virtualize global and large pages of the guest correctly.
- Currently in our work we do not consider memory-mapped devices. As a result, we do not model memory writes with side effects and can set the type of the whole guest memory to "write-back". One might add support for memory mapped I/O to our model. In this case the caching policy for the virtualized memory has to be changed either to mirror the caching policy of the guest, or to split the memory region into two disjoint portions, one with a write-back type and another with an uncacheable memory type. With the first solution one would have to make caches visible in the hardware machine running in the guest mode, as well as to add caches to guest VMs.
- Another restriction which we have in our hardware model, is the absence of interrupts. A possible way to integrate interrupts to our model is to reorder the steps of interrupt handlers to consistency points, just as we do with the guest steps. This requires proving another reduction theorem in the style of [Bau12]. The work on the interrupt handling in hypervisor verification in currently in progress at the chair of Prof. Paul in Saarland University.
- For the hypervisor's own translations we are currently considering only identity-mapped page tables. One might generalize our theorems to be applied for other mappings. Throughout the thesis we have given some hints on how to do that (see Section 4.5.3 and Section 7.2.2).
- The formal framework, presented in this thesis, does not consider the kernel layer of the hypervisor. Probably the most complicated part of the kernel layer verification is the proof of a thread switch mechanism. To integrate the results from this thesis with the correctness of the kernel layer, one has to show that the kernel layer provides an abstraction of the hardware machine, which we use in the bottom of our model stack. The work on this problem is currently in progress at the chair of Prof.

Paul in Saarland University.

- The hardware model which we used in our VCC proofs is slightly simpler than the one introduced in this thesis. Particularly, we haven't argued about memory request/result buffers and SBs. We believe, that these differences do not produce any additional obligations on the hypervisor code itself and only reduce the number of unintercepted hardware steps, for which we show correct virtualization in VCC (the paper-and-pencil proof for all of these steps, including the ones which were not performed in VCC, is given in Theorem 8.3). We plan to adapt our formal VCC proofs so that they adhere to the paper-and-pencil verification presented in this thesis as a part of the future work.
- When creating a new SPT, we currently assume that there is always at least one free SPT available. To weaken this assumption one has to implement a more sophisticated approach in management of free/used SPTs. For instance, one can allocate SPTs dynamically from the heap memory of the hypervisor and limit the number of SPTs which can be allocated to a given VP to make sure that every VP will get its own portion of the heap memory reserved for SPTs. Further, if the number of SPTs allocated to a single VP exceeds the limit, one has to find some SPTs for reclaiming (i.e., detaching and freeing). In our algorithm we do reclaiming only at the time when we detach a subtree in the PF intercept handler, but a similar reclaiming strategy can be applied to an arbitrary SPT of a given VP.
- Using our verification framework, one might verify more complicated versions of the SPT algorithm, for instance the version with sharing of SPTs, pre-fetching, and selective-write protection of GPTs.

To complete verification of the virtualization layer of the hypervisor using our framework, one has to do the following:

- instantiate the instruction automaton of the hardware machine with the x64 ISA specification in the style of [Deg11],
- lift the part of the instruction automaton responsible for the guest execution to the hardware component of the C-IL + HW machine,
- extend the consistency relation for C-IL + HW to couple the newly added part of the state,
- for the abstract VM configuration use the automaton with both memory and instruction parts, instead of just the memory automaton which is used now,
- show simulation not only for memory actions, but for all hardware steps in the guest mode.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 2–13, New York, NY, USA, 2006. ACM.

[ACH+10] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul. Verifying shadow page table algorithms. In *Formal Methods in Computer Aided Design (FMCAD) 2010*, pages 267–270, Lugano, Switzerland, 2010. IEEE.

[ACKP12] Eyad Alkassar, Ernie Cohen, Mikhail Kovalev, and Wolfgang Paul. Verification of TLB virtualization implemented in C. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 209–224. Springer Berlin / Heidelberg, 2012.

[Adv08] AMD-V nested paging. White paper, July 2008.

[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.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[AHL+09] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. In *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*. Springer, 2009.

[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.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, May 1991.

[Alk09] Eyad Alkassar. *OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, University of Saarland, 2009.

[AP08] E. Alkassar and W. Paul. On the verification of a “baby” hypervisor for a RISC machine; draft 0. `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur/ws0607/layouts/hypervisor.pdf`, 2008.

[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] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE’10, pages 71–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[AS07] David Aspinall and Jaroslav Sevcik. Formalising Java’s data race free guarantee. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 22–37. Springer Berlin / Heidelberg, 2007.

[Bau12] Christoph Baumann. Reordering and simulation in concurrent systems. Technical report, Saarland University, Saarbrücken, 2012.

[BBCL11] Gilles Barthe, Gustavo Betarte, Juan Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 231–245. Springer Berlin / Heidelberg, 2011.

[BCHP05] Richard Bornat, Cristiano Calcagno, Peter W. O Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and MartÃ-n Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer Berlin / Heidelberg, 2006.

[Bev89a] William R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Software Eng.*, pages 1382–1396, 1989.

[Bev89b] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5:519–530, 1989.

[BHMY89a] William R. Bevier, Warren A. Hunt, J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, 1989.

[BHMY89b] W.R. Bevier, W.A. Hunt, J. Strother Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[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.

[BLD+10] Chang Bae, John R. Lange, Peter A. Dinda, Chang Bae, John R. Lange, and Peter A. Dinda. Comparing approaches to virtualized page translation in modern vmms. Technical Report NWU-EECS-10-07, Northwestern Univirsity (Electrical Engineering and Computer Science Department), April 2010.

[BLD11] Chang S. Bae, John R. Lange, and Peter A. Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 255–264, New York, NY, USA, 2011. ACM.

[BM11] Sascha Böhme and MichałMoskal. Fat pointers, skinny annotations: A heap model for modular C verification, 2011. Draft.

[ByECD+06] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[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.

[CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009.

[CMST10] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010.

[CMTS09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci.*, 254:85–103, October 2009.

[CPS13] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of multi core hypervisor verification. In Peter van Emde Boas et al., editor, *Invited paper. To appear on SOFSEM 2013, Theory and Practice of Computer Science*, LNCS. Springer, 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)*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418, Edinburgh, UK, July 2010. Springer.

[CVJ+12] Sagar Chaki, Amit Vasudevan, Limin Jia, Jonathan M. McCune, and Anupam Datta. Design, development and automated verification of an integrity-protected hypervisor. Technical Report CMU-CyLab-12-017, CMU CyLab, July 2012.

[Day10] Robert Day. Hardware virtualization puts a new spin on secure systems. White paper, 2010.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, University of Saarland, 2011.

[dMB08] Leonardo Mendonca de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340, 2008.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms - Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2009.

[DRL05] Robert Deline, K. Rustan, and M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (MSR), March 2005.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[FSGD09] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *Journal of Automated Reasoning*, 42:301–347, 2009.

[GWF10] Thomas Gaska, Brian Werner, and David Flagg. Applying virtualization to avionics systems - The integration challenges. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5.E.1–1–5.E.1–19. IEEE, October 2010.

[HKV98] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models. Part I: Definitions and comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998.

[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.

[HN09] Kenneth Hess and Amy Newman. *Practical Virtualization Solutions: Virtualization from the Trenches*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[HP07] M. A. Hillebrand and W. J. Paul. On the architecture of system verification environments. In *Haifa Verification Conference 2007, October 23-25, 2007, Haifa, Israel*, LNCS. Springer, 2007.

[HP10] Mark Hillebrand and Wolfgang Paul. Walking in the shadows: summary of Ernie Cohen's talks on page table virtualization (draft), February 2010. obtained from the authors.

[IdRT08] T. In der Rieden and A. Tsyban. CVM - A verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[Int07] Intel Corporation. *TLBs, Paging-Structure Caches, and Their Invalidation*, April 2007.

[Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A and 3B)*, May 2011.

[Int12] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, May 2012.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kiv07] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS'07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[LABS12] Xavier Leroy, W. Appel, Andrew, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Rapport de recherche RR-7987, INRIA, June 2012.

[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.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[Lie95] Jochen Liedtke. On Âμ-kernel construction. In *Symposium on Operating System Principles*. ACM, 1995.

[LP08a] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[LP08b] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08).*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 23–40. Elsevier Science B. V., 2008.

[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.

[LS09] D. Leinenbach and T. Santen. Verifying the microsoft Hyper-V hypervisor with VCC. In *16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LV92] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0032002.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121:214–233, 1995.

[LW11] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In Alex Groce and Madanlal Musuvathi, editors, *Model Checking Software*, volume 6823 of *Lecture Notes in Computer Science*, pages 144–160. Springer Berlin / Heidelberg, 2011.

[Mic12a] Microsoft Corp. VCC: A C Verifier. `http://vcc.codeplex.com`, 2012.

[Mic12b] Microsoft Corp. Windows Server 2008 R2 – virtualization with Hyper-V. `http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-v.aspx`, 2012.

[NSL+06] G Neiger, A Santoni, F Leung, D Rodgers, and R Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–178, 2006.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs*, pages 189–206, 2007.

[O'H04] Peter O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer Berlin / Heidelberg, 2004.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin / Heidelberg, 2009.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In Theo Dâ™Hondt, editor, *ECOOP 2010 â“ Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer Berlin / Heidelberg, 2010.

[Pau11] Wolfgang J. Paul. Multicore system architecture: Lecture notes WS09/10. `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur/ss11/layouts/multicore_notes.pdf`, 2011.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, May 2007.

[Phi06] Robert Phillips. The design of the XI shadow mechanism. `http://old-list-archives.xen.org/archives/html/xen-devel/2006-06/pdfz2XOX8IfNY.pdf`, 2006.

[Rid07] Tom Ridge. Operational reasoning for concurrent Caml programs and weak memory models. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer Berlin / Heidelberg, 2007.

[SAGG+93] Jorgen F. Sogaard-Andersen, Stephen J. Garl, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In *In Proceedings of the 5th international conference on Computer aided verification, Elounda, Greece, volume 697 of LNCS*, pages 305–319. Springer Verlag, 1993.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *Logic for Programming, AI, and Reasoning, volume 3452 of LNAI*, pages 398–414. Springer, 2005.

[Sch12a] Sabine Schmaltz. C-IL with ghost state semantics. `http://www-wjp.cs.uni-saarland.de/publikationen/SchmaltzC-IL+Ghost.pdf`, July 2012.

[Sch12b] Sabine Schmaltz. *Towards Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C.*

PhD thesis, Saarland University, Computer Science Department, 2012. To appear.

[Sha12] Andrey Shadrin. *Mixed Low- and High Level Programming Languages Semantics. Automated Verification of a Small Hypervisor: Putting It All Together. (DRAFT)*. PhD thesis, Saarland University, Saarbrücken, 2012.

[SHW+08] John Te-Jui SHEU, Matthew D. HENDEL, Landy WANG, Ernest S. COHEN, Rene Antonio VEGA, and Sharvil A. NANAVATI. Reduction of operational costs of virtual TLBs. Patent Application, 06 2008. US 2008/0134174 A1.

[SI94] CORPORATE SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *SIGARCH Comput. Archit. News*, 14(2):414–423, May 1986.

[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, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg, 2012.

[SSN+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. *SIGPLAN Not.*, 44(1):379–391, January 2009.

[Sta86] Eugene W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56, 1986.

[Tew07] Hendrik Tews. Formal methods in the Robin project : Specification and verification of the Nova microhypervisor. In *Proceedings of the IFM 2007 C/C++ Verification Workshop (Oxford, UK, July 2007)*. Nijmegen : Radboud University Nijmegen, 2007.

[The12] The Verisoft XT Consortium. The Verisoft XT Project. `http://www.verisoftxt.de`, 2012.

[TWV+08] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro–hypervisor verification. Technical Report ICIS–R08012, Radboud University Nijmegen, May 2008.

[Ver08] Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`, 2008.

[VMQ+10] Amit Vasudevan, Jonathan M. McCune, Ning Qu, Leendert van Doorn, and Adrian Perrig. Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (Trust 2010)*, June 2010.

[VMw07] Understanding full virtualization, paravirtualization, and hardware assist. White paper, 2007.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[WZW+11] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective hardware/software memory virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 217–226, New York, NY, USA, 2011. ACM.

Index

AbsPte, 44
CD, 32
Cache, 33
GuestAddr, 67
Hardware, 25
IMPTAddr, 75
InstrCore, 58
InstrHw, 25
MemAcc, 28
MemCore, 50
MemHw, 24
MemReq, 29, 57
MemReqMain, 29
MemRes, 30
MemResMain, 30
MemType, 32
Memory, 31
Ownership, 67
PLS, 173
PTI, 175
PfData, 49
Pid, 25, 107
PrivateAddr, 67
Pt, 42
ReadOnlyAddr, 66
RedHardw, 61
$RedHardw_{ca}$, 62
$RedHardw_{sb}$, 62
RedMemHardw, 61
*RegCr*3, 51
Rights, 41
SB, 38
SBItem, 38
SharedAddr, 66
StackAddr, 67
Store, 38
Tid, 105, 107
Tlb, 44
TlbReq, 28
UC, 32
VmConfig, 171
VpConfig, 172
WB, 32
WC, 32
WP, 32
WT, 32
Walk, 41
$[e]_c^{\pi,\partial}$, 99, 101
$g^0 \xrightarrow{\beta} g^n$, 169
$g \xrightarrow{a} g'$, 169
$h^0 \xrightarrow{\beta} h^n$, 15
$\beta \equiv \omega$, 146
$c \xrightarrow[\pi,\partial]{\beta} c'$, 140
$\mathbb{T}_C$, 90
$\mathbb{T}_P$, 90
$\mathcal{M}_{\mathcal{G}}$, 128
$\mathcal{M}_{G\mathcal{E}}$, 128
$\pi, \partial \vdash c \rightarrow c'$, 106, 139, 159
$\pi, \partial \vdash c \rightarrow^* c'$, 106
$\pi, \partial \vdash c \rightarrow_t c'$, 106
$\pi, \partial \vdash c \rightarrow_t^+ c'$, 106
$\pi, \partial \vdash c \rightarrow^+ c'$, 106
$\pi, \partial \vdash c \xrightarrow{cil} c'$, 139, 159
$\pi, \partial \vdash c \xrightarrow{cil}_t c'$, 140
$\pi, \partial \vdash c \xrightarrow{cil(t)} c'$, 140
$\pi, \partial \vdash c \xrightarrow{G} c'$, 132
$\pi, \partial \vdash c \xrightarrow{hw} c'$, 139
$\pi, \partial \vdash c \xrightarrow{hw}_t c'$, 140
$\pi, \partial \vdash c \xrightarrow{hw(t)} c'$, 140
$\pi, \partial \vdash c \xrightarrow{I} c'$, 132
$\pi, \partial \vdash c(t) \rightarrow c'(t)$, 106, 139
$\mathbb{B}^{pfn}$, 27
$\mathbb{B}^{vpfn}$, 27

$\mathbb{B}^{bpa}$, 27
$\mathbb{B}^{bva}$, 27
$\mathbb{B}^{qpa}$, 27
$\mathbb{B}^{qva}$, 27
$\mathbb{B}_{gm}$, 117
$\mathbb{E}$, 93
$\mathbb{E}_G$, 129
$\mathbb{F}_{name}$, 93
$\mathbb{N}_{spt\text{-}cnt}$, 175
$\mathbb{O}_1$, 93
$\mathbb{O}_2$, 93
$\mathbb{S}$, 93
$\mathbb{S}'$, 129
$\mathbb{S}_G$, 130
$\mathbb{T}_{G\mathbb{Q}}$, 126
$\mathbb{T}_{\mathbb{Q}}$, 91
σ^{π}_{∂}, 101
ι, 99
$\iota^{\pi,\partial}_{E}$, 100
ι_F, 100
ι_V, 99
ι^{π}_{fun}, 99
zero$_{\partial}$, 101
abs-pte, 43
affected-byte-addr, 68
$byte_i$, 31
$bytes2val_{\partial}$, 97
$c(t)$, 106
$c.\mathcal{M}_i$, 102
$c.\mathcal{V}_{top}$, 100
$c.f_i$, 102
$c.loc_i$, 102
$c.rds_i$, 102
cacheable, 32
$call_{frame}$, 104
cg2cil, 133
cg2cil-prog, 133
cg2cil-sf$_{\pi}$, 133
cg2cil-stack, 133
chwg2cg, 160
chwg2chw, 160
cil, 138
cil2chw, 138
cil2chw$_0$, 138
code-consis, 117
combine, 31
commit-store, 41
complete, 42
concrete-pte, 44
$conf_{C+G}$, 130
$conf_{C+HW+G}$, 159
$conf_{C+HW}$, 138
$conf_{C-IL}$, 95
$conf_{CC+G}$, 130
$conf_{CC+HW+G}$, 159
$conf_{CC+HW}$, 138
$conf_{CC-IL}$, 106
consis, 119
$consis_{CC+HW}$, 149
$context_{C-IL}$, 96
control-consis-stable$_i$, 118
control-consis$_i$, 118
core-atomic-cmpxchng, 53
core-instr-step, 58
core-issue-mem-req, 51, 58
core-locked-memory-write, 54
core-memory-read, 52
core-memory-write, 52
core-mov2cr3, 56
core-prepare-page-fault, 54
core-send-mem-res, 51, 58
core-tlb-invlpga, 55
core-trigger-page-fault, 55
core-vmexit, 56
core-vmrun, 57
$core_c$, 138
cosched, 115
$count_{stmt}$, 134
cpoint, 114
$cpoint_{C-IL}$, 120
$cpoint_k$, 114
cr3-2-uint, 51
create-walk, 45
$cwalks_c$, 184
decl, 99
drop-walks, 47
drop-line, 37
drop-sfence, 41
$drop_{frame}$, 102
empty-sb, 40
empty-tlb, 50
ext(*RedHardw*), 146
ext(C-IL+HW), 146
extend-walk, 46
fetch-line-from-ca, 36
fetch-line-from-mm, 36
forward, 40
$frame_{C+G}$, 130
$frame_{C-IL}$, 95
fun_{C+G}, 131

fun$_{C-IL}$, 96
gfun$_{C+G}$, 131
ghost-safe-seq$^{\pi,\partial}_{C+G}$, 134
ghost-safe-stmt$^{\pi,\partial}_{C+G}$, 134
gid, 169
global-consis, 117
gm-consis, 117
guest-cpoint$_k$, 114
guest-iopoint$_k(\beta, i)$, 113
guest-trace, 146
guest$_c$, 172
hp2vp$_c$, 170, 174
hw2gw$_c$, 179
hw-consis, 148
hw-id-trace, 170
hw-step(a), 140
hw-trace, 146
hyp-cpoint$_k$, 114
hyp-iopoint$_k$, 112
idx2hpa$_c$, 175
impt-in-IMPTAddr, 75
inc$_{loc}$, 102
inject-data$^{\pi,\partial}$, 105
inv-complete-walks, 184
inv-core-buffers-coupling, 178
inv-coupling, 185
inv-cr3-cacheable, 77
inv-cr3-coupling, 177
inv-disjoint-ownership-domains, 68
inv-distinct-asids, 174
inv-htlb-complete-walks, 179
inv-htlb-walks, 181
inv-local-consis-stable, 119
inv-memory-types, 183
inv-mm-coupling, 176
inv-owned-atomic, 69
inv-owned-atomic$_r$, 81
inv-owned-reads, 68
inv-owned-reads$_r$, 81
inv-owned-stores, 69
inv-owned-writes, 68
inv-owned-writes$_r$, 81
inv-ownership-discipline, 70
inv-ownership-discipline$_r$, 81
inv-ownership-transfer$_r$, 82
inv-partial-walks, 183
inv-pls-walks, 181
inv-reachable-child, 183
inv-reachable-root, 183
inv-running-asids, 181
inv-sb-cacheable, 63
inv-sb-coupling, 177
inv-tlb-cacheable, 63
inv-tlb-coupling, 185
inv-tlb-ownership, 69
inv-tlb-ownership$_r$, 82
inv-tlb-walks-impts, 76
inv-valid-asids-range, 175
inv-valid-im-translations, 76
inv-vtlb-walks, 181
inval-tlb, 50
iopoint$_{\beta,k}$, 113
iopoint$_\beta$, 113
is-empty, 40
is-function, 96
isarray, 92, 126
isfptr, 92, 126
isptr, 92, 126
local-consis$_i$, 119
local-seq, 115
make-exclusive, 37
masked-update$_c$, 143
memreq-eq, 178
memres-eq, 178
mt-combine, 33
mtrr-cacheable, 76
mtrr-mt, 33
next-cpoint, 114
next-instr-state, 58
no-page-fault, 49
non-aliasing-abstractions, 185
normal-write$^{\pi,\partial}$, 109
page-fault, 46
pass-ownership, 37
pat-mt, 33
pending-byte-store, 39
pending-qword-store, 39
pending-store, 39
pfn2bytes, 27
pfn2qwords, 27
ph-inval-tlb, 50
pid, 25
pls$_c$, 174
prog$_{C+G}$, 131
prog$_{C-IL}$, 95
pte-addr, 43
pte-read, 43
pte-set-ad-bits, 47
pte-write, 43

pti_c, 176
$qt2t$, 91
$qword2bytes$, 27
$read$, 32, 34, 97, 98
$read\text{-}pte_c$, 142
$read_{\surd}$, 34
$read_{val_{\mathcal{M}_G}}$, 129
$reduced\text{-}sb$, 71
$reduced\text{-}sb\text{-}mm$, 70
$reduced\text{-}ca\text{-}hw$, 62
$reduced\text{-}ca\text{-}mm$, 62
$reduced\text{-}hw$, 79
$reduced\text{-}sb\text{-}hw$, 71
$reduced\text{-}tlb$, 77
$reduced\text{-}tlb\text{-}hw$, 77
$reorder\text{-}store$, 40
$root\text{-}pt\text{-}memtype$, 51
$running\text{-}thread_k$, 115
$rwalks_c$, 182
$safe\text{-}assignment^{\pi,\partial}$, 109
$safe\text{-}cmpxchng^{\pi,\partial}$, 110
$safe\text{-}conf$, 83
$safe\text{-}conf_r$, 84
$safe\text{-}conf^{\pi,\partial}_{C+HW+G}$, 160
$safe\text{-}conf^{\pi,\partial}_{C+HW}$, 147
$safe\text{-}exp^{\pi,\partial}$, 108
$safe\text{-}exps^{\pi,\partial}$, 108
$safe\text{-}fcall^{\pi,\partial}$, 110
$safe\text{-}hyp\text{-}conf_r$, 84
$safe\text{-}hyp\text{-}seq_r$, 85
$safe\text{-}local\text{-}seq^{\pi,\partial}_{C-IL}$, 111
$safe\text{-}locked\text{-}write^{\pi,\partial}$, 109
$safe\text{-}prog^{\pi,\partial}_{CC+HW+G}$, 161
$safe\text{-}prog^{\pi,\partial}_{CC+HW}$, 147
$safe\text{-}read$, 107
$safe\text{-}seq$, 83
$safe\text{-}seq^{\pi,\partial}_{CC+HW}$, 147
$safe\text{-}seq_r$, 84
$safe\text{-}step^{\pi,\partial}_{C-IL}$, 111
$safe\text{-}stmt^{\pi,\partial}$, 110
$safe\text{-}tlbs_c$, 147
$safe\text{-}tlbs_r$, 84
$safe\text{-}transfer^{\pi,\partial}$, 111
$safe\text{-}vmrun^{\pi,\partial}$, 110
$safe\text{-}write^{\pi,\partial}$, 109
$sb\text{-}cnt$, 38
$sb\text{-}data$, 39
$sb\text{-}memtype$, 39
$set\text{-}access\text{-}dirty$, 47
set_{loc}, 103
$shared\text{-}read^{\pi,\partial}$, 119
$shared\text{-}stmt^{\pi,\partial}$, 119
$shared\text{-}write^{\pi,\partial}$, 109
si_G, 133
$size_{\partial}$, 97
spt_c, 175
$stack\text{-}consis\text{-}stable_i$, 118
$stack\text{-}consis_i$, 117
$stmt_{next}$, 102, 132
$sub\text{-}expr^{\pi,\partial}$, 108
$tlb\text{-}empty\text{-}asid$, 49
$tlb\text{-}fault\text{-}ready$, 49
$tlb\text{-}invalidated$, 49
$tlb\text{-}invalidated\text{-}pf$, 49
$tlb\text{-}memtype$, 50
$tlb\text{-}transl\text{-}ready$, 48
top, 102
$traces\text{-}eq$, 170
$uint2cr3$, 51
val, 92
$val2bytes_{\partial}$, 97
val_G, 128
$valid\text{-}asid_c$, 174
$valid\text{-}im\text{-}transl\text{-}step$, 76
$vm\text{-}trace$, 169
vp_c, 172
$walks\text{-}to_c$, 175
$wext$, 45
$wext_{\surd}$, 46
$write$, 32, 35, 40, 97, 98
$write\text{-}pte_c$, 142
$write_{\surd}$, 35
$write_{val_{\mathcal{M}_G}}$, 129
$writeback\text{-}line\text{-}to\text{-}mm$, 37