Low-Level Program Verification under Cached Address Translation

Hira Taqdees Syeda

ORCID: 0000-0002-4923-3783

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales

May 2019

Thesis/Dissertation Sheet

Surname/Family Name	:	**Syeda**
Given Name/s	:	**Hira Taqdees**
Abbreviation for degree as give in the University calendar	:	**PhD**
Faculty	:	**Engineering**
School	:	**Computer Science**
Thesis Title	:	**Low-Level Program Verification under Cached Address Translation**

Abstract

Operating system (OS) kernels achieve isolation between user-level processes using multi-level page tables. The hardware-implemented translation lookaside buffer (TLB) caches page table walks, and therefore the TLB and its consistency with memory are security critical for OS kernels, including formally verified kernels such as seL4. If performance is paramount, this consistency can be subtle to achieve; yet, all major formally verified kernels currently leave the TLB as an assumption. They assume correct TLB management because faithfully modeling the hardware details of a TLB would significantly complicate the program logic used to verify the OS code. For instance, a simple memory read operation would now change the state of the program.

In this thesis, we present a formal model of the memory management unit (MMU) in the interactive proof assistant Isabelle/HOL for the ARMv7-A architecture which includes the TLB, its maintenance operations, and its derived properties. We integrate this specification into the Cambridge ARM model. We derive sufficient conditions for TLB consistency, and we abstract away the functional details of the MMU using data refinement for simpler reasoning about executions in the presence of cached address translation, including complete and partial walks.

Based on the verified abstraction of the MMU model of the ARMv7-A architecture, we present a logic in Isabelle/HOL for reasoning about low-level programs in the presence of cached address translation. We extract invariants and necessary conditions for correct TLB operation that mirror the informal reasoning of OS engineers. We show that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

This research removes the unnecessary TLB complexities from program reasoning, and provides a reasoning framework for validating TLB management in OS kernel verification.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

…………………………………………	…………………………………………	01/05/2019
Signature	Witness Signature	Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in their thesis in lieu of a Chapter if:

- The student contributed greater than 50% of the content in the publication and is the "primary author", ie. the student was responsible primarily for the planning, execution and preparation of the work for publication
- The student has approval to include the publication in their thesis in lieu of a Chapter from their supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis

Please indicate whether this thesis contains published material or not.

☐ *This thesis contains no publications, either published or submitted for publication*

☑ *Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement*

☐ *This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below*

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Name	**Signature**	**Date (dd/mm/yy)**
Hira Taqdees Syeda		**01/05/2019**

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ……………………………………………….............

Date 01/05/2019 ……………………………………………….............

COPYRIGHT STATEMENT

‘I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.
I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed ……………………………………………………..........................

Date 31/07/19 ……………………………………………………..........................

AUTHENTICITY STATEMENT

‘I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.’

Signed ……………………………………………………..........................

Date 31/07/19 ……………………………………………………..........................

Abstract

Operating system (OS) kernels achieve isolation between user-level processes using multi-level page tables. The hardware-implemented translation lookaside buffer (TLB) caches page table walks, and therefore the TLB and its consistency with memory are security critical for OS kernels, including formally verified kernels such as seL4. If performance is paramount, this consistency can be subtle to achieve; yet, all major formally verified kernels currently leave the TLB as an assumption. They assume correct TLB management because faithfully modeling the hardware details of a TLB would significantly complicate the program logic used to verify the OS code. For instance, a simple memory read operation would now change the state of the program.

In this thesis, we present a formal model of the memory management unit (MMU) in the interactive proof assistant Isabelle/HOL for the ARMv7-A architecture which includes the TLB, its maintenance operations, and its derived properties. We integrate this specification into the Cambridge ARM model. We derive sufficient conditions for TLB consistency, and we abstract away the functional details of the MMU using data refinement for simpler reasoning about executions in the presence of cached address translation, including complete and partial walks.

Based on the verified abstraction of the MMU model of the ARMv7-A architecture, we present a logic in Isabelle/HOL for reasoning about low-level programs in the presence of cached address translation. We extract invariants and necessary conditions for correct TLB operation that mirror the informal reasoning of OS engineers. We show that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

This research removes the unnecessary TLB complexities from program reasoning, and provides a reasoning framework for validating TLB management in OS kernel verification.

Publications

The work presented in this thesis has produced the following publications.

Journal Paper

- Hira Taqdees Syeda and Gerwin Klein: Formal Reasoning under Cached Address Translation. In: Journal of Automated Reasoning (JAR), Special Issue: ITP 2018.
  Invited Submission, Status: Submitted, under Review

Refereed Conference Papers

- Hira Taqdees Syeda and Gerwin Klein: Reasoning about Translation Lookaside Buffers. In: 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21). EPiC Series in Computing, vol 46, pages 490–508. Easy Chair.
  Published: May 4, 2017
- Hira Taqdees Syeda and Gerwin Klein: Program Verification in the Presence of Cached Address Translation. In: Interactive Theorem Proving (ITP 2018). Lecture Notes in Computer Science, vol 10895, pages 542-559. Springer, Cham.
  Published: July 4, 2018

In the loving memory of my father who taught me perseverance.

In the loving memory of my mother who taught how to smile through life.

I am eternally grateful.

Acknowledgements

Heartfelt thanks to my wonderful supervisor Gerwin Klein for almost everything related to my PhD life. His unwavering support, dedicated supervision, research vision, ethical work practices, calm optimism, perfect balance and joyful nature have made this PhD once in a lifetime experience for me.

Many thanks to all my friends and colleagues at Trustworthy Systems, especially to Miki Tanaka for keeping me motivated and listening to my worries, to Matthew Brecknell, Joel Beeren, Japheth Lim, Rafal Kolanski, Thomas Sewell, Daniel Matichuk and Xin Gao for making me Isabelle friendly and for helping me prove the initial results. Also, special thanks to Adrian Danis for explaining to me how a TLB works, Yutaka Nagashima for his cheerful company, Peter Höfner for impartially mentoring me, Christine Rizkallah for telling me the importance of summer schools, June Andronick for helping me out in presentation skills, and Gernot Heiser for taking care of the administration. Many thanks to Alejandro Gómez, Robert Sison, Siwei Zhuang, Peter Chubb, Sidney Amani, Corey Lewis, Carroll Morgan, Anna Lyons, Kent Mcleod, Qian Ge, Kofi Atuah, Amir Zarrabi, Johannes Åman, Ramana Kumar, Callum Bannister, Santiago Bautista, Pang Luo, Victor Duy and Brigitte Biscotto for their friendly company and many talks over coffee.

Many thanks to the dean of my residential college, Susan Bazzana for always being there when I needed her support. There is possibly no way to thank my amazing sister, Warda. I owe a great deal to her not only during the PhD phase but to life. Thanks for always being my support system, for loving me unconditionally, for countless giggles, for keeping me in reality and for being with me through every thick and thin of life. I strive to be like her. Thanks to my wonderful brother Uzman for always giving me the best of advice, for facilitating me in every way possible and for putting me ahead of his comfort. Thanks to my sweet sister Faria for always motivating me to do my best. Thanks to my sibling's significant others Kiran, Najam and Mahmood for bringing joy to my life. Thanks to my friends Sara, Muqaddas and Shuichi for many smiles and uplifting my spirits.

I would also like to thank my master's supervisor Osman Hasan who introduced me to the beautiful world of formal methods, and to my high-school teacher Saif-ur-Rehman for making me passionate about mathematics.

Contents

1 Introduction 1

1.1 Contributions . . . 5

1.2 Related Work . . . 6

1.3 Thesis Outline . . . 12

2 Notation 14

2.1 Isabelle . . . 15

2.2 HOL in Isabelle . . . 16

2.2.1 Types, Terms and Formulae . . . 16

2.2.2 Higher-Order Logic Operations . . . 17

2.2.3 Built-in Types used in this Thesis . . . 17

2.2.4 Function Update and the `Let` Construct . . . 19

2.3 Type Classes in Isabelle . . . 19

2.4 Record Types in Isabelle/HOL . . . 20

2.5 State Monads . . . 22

3 Virtual Memory in the ARMv7-A Architecture 23

3.1 Basic Concepts of Virtual Memory . . . 24

3.2 Virtual Memory System in the ARMv7-A Architecture . . . 26

3.2.1 Pages . . . 26

3.2.2 Page Tables . . . 27

3.2.3 Address Translation . . . 28

3.2.4 Translation Lookaside Buffer . . . 29

3.2.5 Caches . . . 33

3.3 OS Kernel Management of ARM's VMSA . . . 34

3.4 Summary and Remarks . . . 37

4 A Formal Model of the ARMv7-A MMU 39

4.1 Page Table Abstraction . . . 40

4.2 A Formal TLB Model for the ARMv7-style MMU . . . 43

4.3 From TLB to MMU Model . . . 46

4.3.1 Page Table Walk . . . 47

4.3.2 Address Translation . . . 49
4.3.3 Memory Operations . . . 50
4.3.4 Updating the Page Table Root Register . . . 50
4.3.5 Flush Operations . . . 50
4.4 MMU Abstraction . . . 51
4.4.1 Determinism . . . 53
4.4.2 Invariance . . . 56
4.4.3 Essence . . . 60
4.4.4 Joining the Refinement Levels . . . 65
4.5 Summary and Remarks . . . 67

5 A Formal Model of the ARMv7-A MMU with ASIDs 69

5.1 ARMv7-A MMU Model with ASIDs . . . 70
5.1.1 Page Table Walk . . . 73
5.1.2 Memory Operations . . . 74
5.1.3 MMU Operations . . . 75
5.2 MMU Abstraction . . . 77
5.2.1 The Deterministic MMU Model . . . 78
5.2.2 The Saturated MMU . . . 80
5.2.3 The Most Abstract MMU Model . . . 83
5.2.4 Joining the Refinement Levels . . . 92
5.3 Summary and Remarks . . . 93

6 A Formal Model of the ARMv7-A MMU with Two-Stage TLB 95

6.1 ARMv7-A MMU Model with TLB and PDC . . . 96
6.1.1 Page Table Walk . . . 100
6.1.2 Memory Operations . . . 102
6.1.3 MMU Operations . . . 104
6.2 MMU Abstraction . . . 106
6.2.1 The Saturated MMU Model . . . 107
6.2.2 The Most Abstract MMU Model . . . 112
6.2.3 Joining the Refinement Levels . . . 122
6.3 Summary and Remarks . . . 124

7 Program Logic in the Presence of Cached Address Translation 125

7.1 Program Logic . . . 126
7.1.1 Syntax . . . 127
7.1.2 Program State and Memory Model . . . 128
7.1.3 Semantic Operations . . . 130
7.1.4 Operational Semantics . . . 133
7.1.5 Hoare Logic . . . 134
7.2 Safe Set . . . 136
7.3 Summary and Remarks . . . 139

8 **Case Study** 141

8.1 MMU Layout - Formal Modeling 142

8.1.1 Kernel Data Structures 144

8.1.2 Assertions on MMU Layout 146

8.2 User Execution 150

8.3 Kernel Execution 151

8.4 Context Switch 153

8.5 Page Table Operations 154

8.6 Summary and Remarks 157

9 **Conclusions** 158

9.1 Summary of Novel Contributions 159

9.2 Proof Effort 160

9.3 Comments on the TLB Modeling 161

9.4 Future Research and Engineering Directions 162

9.5 Final Remarks 165

List of Figures

3.1 Basic Concepts of Virtual Memory 25
3.2 An Example of Paged Memory in the ARM Architecture 27
3.3 An Example of Two-Level Page Table in the ARM Architecture . . 28
3.4 Translation Flow for a Section 28
3.5 Translation Flow for a Small Page 29
3.6 Functional Role of the Translation Lookaside Buffer (TLB) 30
3.7 A Format for TLB Entries 31
3.8 An Example of a TLB Lookup Resulting in a Hit 31
3.9 TLB, PDC and Page Table Lookup 33
3.10 Cache Hierarchy in the ARMv7-A Architecture 34
3.11 An Example of a Virtual Address Space with Kernel Window . . . 35
3.12 OS Kernel Page Table Management 36

4.1 An Abstraction of an ARMv7-style TLB 43
4.2 ARMv7-style Memory Management Unit 46
4.3 Refinement Stack for MMU Models 53
4.4 Refinement between Nondeterministic and Deterministic Translation 55
4.5 ARMv7-style Memory Management Unit with Abstract TLB 61
4.6 Refinement between Nondeterministic and Abstract MMU 65
4.7 Refinement between Nondeterministic and Abstract Memory Operations 67
4.8 Refinement between Nondeterministic and Abstract MMU Operations 68

5.1 ARMv7-style Memory Management Unit with ASIDs 71
5.2 Visual Representation of Page Table Walk Function 73
5.3 Refinement Stack for MMU Models 78
5.4 ARMv7-style Memory Management Unit with Abstract MMU . . . 83
5.5 Refinement between Nondeterministic and Abstract MMU 92
5.6 Refinement between Nondeterministic and Abstract Memory Operations 93
5.7 Refinement between Nondeterministic and Abstract MMU Operations 94

6.1 Address Translation in the Presence of a TLB and PDC 97
6.2 PDC Inconsistency Example 98
6.3 ARMv7-A MMU with TLB and PDC 100
6.4 The Encoding of Two-Stage Page Table Walks 101
6.5 Refinement Stack for MMU Models 107
6.6 Hierarchical Saturation of PDC and TLB with Current Page Table 109
6.7 ARMv7-style Memory Management Unit with Abstract MMU . . . 113

6.8 Refinement between Nondeterministic and Abstract MMU 122
6.9 Refinement between Nondeterministic and Abstract Memory Operations 123
6.10 Refinement between Nondeterministic and Abstract MMU Operations 124

7.1 Syntax of the Heap based WHILE Language. 127
7.2 Abstracted TLB Memory Model 129
7.3 Semantics of Arithmetic and Boolean Expressions 133
7.4 Big-Step Semantics of Commands with Successful Memory Access . 134
7.5 Big-Step Semantics of Commands with Unsuccessful Memory Access 135
7.6 Hoare Logic Rules for Standard Commands 136
7.7 Hoare Logic Rules for Commands with TLB Effects 137

8.1 Virtual Address Space with Kernel Window 143
8.2 Page Table Layout by seL4-inspired Kernel 144
8.3 Roots Map Layout 145

List of Tables

2.1 Higher-Order Logic Standard Operations 17
2.2 Notation for Set Operations 17
2.3 Basic HOL Functions for List 18
2.4 N-bit Operators 18

3.1 TLBs in ARMv7-A Implementations 33

5.1 Read/Write Dependencies for Memory and MMU Operations 84

6.1 Read/Write Dependencies for Memory and MMU Operations 114
6.2 When does a Page Table Walk Change Produce an Inconsistency? . 116

CHAPTER

ONE

Introduction

The operating system (OS) kernel is the collection of low-level programs that communicate directly with the hardware and provide a high-level interface to user applications. The OS kernel is responsible for correctly managing software-visible hardware components such as caches and translation lookaside buffers (TLBs). The execution state of these hardware components determines the correct functionality of programs including the OS kernel itself. The interaction of low-level programs with the hardware is security critical, and if compromised can lead to leakage of confidential data. The Meltdown attack (Lipp et al., 2018) is an example of exploiting vulnerabilities in software-visible hardware such as the TLB which is a dedicated cache for address translation results. The Meltdown attack exploits the fact that permission bits of TLB entries are not checked during speculative execution on some platforms, and uses a cache side channel to thereby make kernel-only privileged TLB mappings readable to non-privileged user space.

This thesis addresses the functional formal verification of low-level programs. The main challenge in verifying low-level programs is memory virtualisation. In the most widely deployed architectures, virtual memory is hardware-implemented and software-managed. Address translation and its caching in the TLB impact the correct execution of otherwise functionally correct programs. For verifying programs in the presence of cached address translation, we pursue the following approach:

- model the software-visible TLB,
- soundly abstract away unnecessary hardware details,
- capture the essential functionality of the TLB required for correct program execution; and
- develop a program logic for verifying programs in the presence of these hardware effects.

The methods used for software verification depend on the software itself and the degree of assurance required. Simple programs, interacting indirectly with the hardware through intermediate software, can be tested in a simulated execution environment. This approach, however, is heuristic and provides incomplete verification. For critical software, such as an OS kernel or a low-level program, verification is sought via formally mathematical models of implementation and specifications. We can then formally verify and reason that the program implements its specifications. Formal analysis methods constitute a spectrum that contains fully automatic methods such as model checking for restricted logics, to interactive methods for more expressive logics such as theorem proving.

This thesis advances the verification of low-level programs: we construct a soundly abstracted memory model, and use this model to develop a logic for reasoning about programs in the presence of cached address translation. We opt for theorem proving as the verification method because we are aiming at high expressiveness and high assurance for the correctness of low-level programs.

This thesis is the intersection of the following fields:

Operating Systems: The operating system is a collection of software routines that manages the hardware. An operating system abstracts away the hardware details, and provides a high-level interface to execute multiple user programs sharing the same hardware by enforcing isolation and confidentiality between them; c.f. (Tanenbaum and Bos, 2014, Chapter 1). Further, the operating system is responsible for task scheduling, preventing unauthorized access to the hardware and protecting itself from potential corruption by user programs.

The operating system kernel is the part of the operating system that has privileged hardware access. The OS kernel configures the hardware and restricts direct hardware access of user programs and other higher-level OS components. The OS kernel is security critical, and any failure in the kernel compromises the functionality of the overall system. Therefore OS kernels are an excellent candidate for formal verification, which provides the highest levels of assurance for correct execution. This thesis addresses the correctness of memory management by the OS kernel, and develops a novel framework for formally verifying low-level programs.

Virtual Memory: Modern OS kernels together with sophisticated hardware implement the concept of virtual memory; thus providing task scheduling and execution of multiple applications at the same time. Virtual memory is a hardware-implemented abstraction over physical memory in which a process cannot access physical memory directly. Instead, a process accesses its virtual address space, which is then resolved to its physical address space by a hardware-supported translation mechanism controlled by the OS kernel.

Virtual memory can be implemented through paging or segmentation mechanisms. Most current virtual memory implementations use paging where the main memory is divided into small equal-sized frames; in contrast to the variable-sized partitions of the segmentation scheme. In addition to the protection of user programs from each other, virtual memory prevents unauthorised access to hardware and protects the operating system itself from corruption by user programs; c.f. (Stallings, 2008, Chapter 8). Any fault in virtual memory management can lead to a security breach, unauthorised communication between the processes and may compromise the overall functionality of the system. We provide more background on virtual memory in Chapter 3.

This thesis formlises the hardware aspects of virtual memory and then provides a platform for reasoning about the correctness of the virtual memory management by the operating system kernel.

Theorem Proving and Isabelle/HOL: Theorem proving is a technique where formal logical systems and automated reasoning tools are combined to prove mathematical theorems. The logical systems vary from less-expressive but fully automated propositional and first-order logic to the more expressive but semi automated second-order and higher-order logics (Harrison, 2009). Non-classical logics, such as intuitionistic logic, are also used for proving theorems. These logical systems are combined with reasoning tools to validate any given theorem using

axioms and derived inference rules of the formal system.

Proof assistants are software tools that allow systems to be expressed in a formal language and provide means for reasoning about these formalised systems in a logical calculus. Any theorem proved with in the proof assistant is sound with respect to the applied logical core. Proof assistants specialised in different logics are used to formally verify the functional correctness of the systems, ranging from software to hardware, biological and analog systems. The rigorous exercise of developing a mathematical model of a system and analysing it using mathematical reasoning increases the chances for catching subtle but critical design errors, that are often ignored by traditional techniques such as simulation and computer-algebra based software.

Our choice of theorem prover is the Isabelle proof assistant (Nipkow et al., 2002). It is a generic proof assistant, and has been instantiated to several object-logics such as first-order logic, higher-order logic, Zermelo-Fraenkel set theory etc. The most widespread instance of Isabelle is Isabelle/HOL, which provides a higher-order logic theorem proving environment that is more expressive and suitable for larger applications. Chapter 2 covers the basic notation and functions of Isabelle/HOL used in this thesis.

Logics for Low-level Program Verification: Hoare logic (Hoare, 1969) is a technique for reasoning about the correctness of imperative programs; c.f. (Nipkow and Klein, 2014, Chapter 12). Hoare logic has several variants for different verification environments, and it can successfully verify properties about low-level programs that manipulate data structures on the heap. The computational state models resources such as memory and registers. A set of inference rules for program statements determines how the execution of a piece of code changes the state of the computation. In this thesis, we develop a Hoare-style logic for reasoning about programs in the presence of cached address translation.

Data Refinement: Data refinement is a technique for program development and verification, and it relates an abstract data model to its concrete implementation model. Data refinement is used to prove that one program soundly implements another program (Morgan, 1994; Roever and Engelhardt, 2008). Typically, an abstract specification is transformed into a series of concrete implementations; with each refinement layer modeling the same behaviour as its predecessor but with more implementation details. We use data refinement in this thesis to develop a stack of abstraction layers of the memory model and its semantics for program execution. The advantage of this kind of refinement is that if we manage to prove safe execution behaviour of the abstraction, we will also have proved safe behaviour of all possible actual executions. We detail our refinement chains in Chapters 4, 5 and 6.

We now list the contributions of this thesis towards the fields mentioned above.

1.1 Contributions

This thesis claims the following novel contributions:

1. A Formal Model for the ARMv7-A Translation Lookaside Buffer (TLB):
We develop a formal operational model of the translation lookaside buffer (TLB) for the ARMv7-A architecture in Isabelle/HOL. This model includes the TLB's essential hardware features such as virtual and physical base addresses, address space identifiers (ASIDs) and global tags. We extend this TLB model for recent ARMv7-A implementations to formalise a two-stage TLB, with a separate page directory cache (PDC). This model captures the caching of partial and complete page table walks, and is based on the information provided by ARM architecture and implementation manuals (ARM, 2008, 2013).

2. A Formal Model for the ARMv7-A Memory Management Unit (MMU):
We integrate the formal TLB model with the page table abstraction by Kolanski (2011) to develop an MMU model for the ARMv7-A architecture in Isabelle/HOL. We integrate this MMU model with the formal instruction set semantics by Fox and Myreen (2010), and formalise instructions for memory operations, TLB maintenance and MMU register updates.

3. Abstraction of the MMU Model using Data Refinement:
We identify the inherent reasoning complexities the formal MMU model would entail for program reasoning, and use data refinement to develop a comprehensive refinement stack to reach an abstract yet sound MMU model. This abstract MMU model allows simpler reasoning and is well-behaved for standard use cases such as user-level programs and OS code under fixed address translation, and expressive enough to allow for the kind of optimisations OS developers need to achieve.

4. A Hoare-style Logic for Program Verification in the presence of Cached Address Translation:
We use the abstract MMU model as the memory model for defining the syntax and operational semantics of a heap-based language with TLB management primitives. Based on the operational semantics, we develop a Hoare-style logic for reasoning about programs in the presence of cached address translation. We also develop derived rules to facilitate simpler program reasoning for user-level code and specific classes of kernel code.

5. A Case Study for Reasoning about Low-Level Programs:
As a demonstrative case study, we formalise the MMU layout of a toy kernel inspired by the seL4 microkernel (Klein et al., 2009). We apply our logic to extract invariants and conditions necessary to reason about user-level and kernel-level executions, context switching and page table operations. The case study demonstrates that the program logic reduces to a standard logic for user-level reasoning,

reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

The models, theorems and logical results presented in this document are generated through Isabelle/HOL; hence they accurately describe our formalisation. The theory files are available online (Syeda, 2019).

1.2 Related Work

We now summarise the research relevant to the formal verification of operating systems and reasoning about virtual memory systems. Our focus remains on the modeling and reasoning about cached address translation.

Formal Verification of Operating Systems:
Operating systems are critical for the security, functionality and performance of computing systems. Given this importance, formal verification of operating systems has a long research history. Formal verification of operating systems is challenging: an operating system is a complex piece of software where several individual routines are interconnected to perform different tasks, continuously communicating with user programs and the underlying hardware. Nevertheless, there have been successful attempts to formally verify operating systems; these projects vary in their focus on specifications, implementation models, the degree of assurance and size of the OS kernel.

We summarise below the notable projects for OS kernel verification.

PSOS: The provably secure operating system (PSOS) is almost a decade long first attempt in the 1970s to design a general-purpose operating system with verifiable security properties (Neumann and Feiertag, 2003). PSOS is a layered architecture; with a total of 17 abstraction layers ranging from the application level to the source code. The design of PSOS is significantly complete, and its specifications are written in an assertion language called SPECIAL. Each abstraction layer and its interface is formally specified in SPECIAL. However, PSOS is not actually implemented; proofs for the code were not carried out. Nevertheless, PSOS has pioneered operating system verification, and it is the first to aim at applying formal methods for implementing operating systems.

UCLA Secure Linux: Effort to specify and verify the UCLA Secure Unix has been reported by Walker et al. (1980). The aim was to verify the OS kernel; hence this effort is a precursor of the modern operating system verification techniques. The kernel of UCLA Secure Unix is implemented in a subset of Pascal; the Pascal code is then read by a tool called XIVUS to perform verification. The code is then abstracted in several layers to prove refinement properties between them. Walker et al. report that about 90% of kernel specifications are complete and 20% of the code is verified. The kernel is slow in performance; the proof assumes the

correctness of compiler, hardware and verification tools. Walker et al. suggested in their report to wait for more machine aid and automation for the proof process, which is now the case after 40 years.

KIT: The kernel for isolated tasks (KIT) is a small kernel written for a uniprocessor with a simple von Neumann architecture; it was introduced by Bevier (1989). It is implemented in an artificial yet realistic assembler instruction set, and the verification is conducted in the Boyer-Moore theorem prover (Boyer and Moore, 1988). The proof method is refinement and correspondence between different levels of abstractions. KIT is significant because it is the first fully verified kernel against its specifications. However, it has a limited functionality: it does provide exception handling, single-word message passing and asynchronous I/O devices access, but stops short of shared memory or virtual memory, dynamic processes and services such as file systems.

Verisoft: The Verisoft project is an extensive proof effort started in 2003 and later evolved into the Verisoft XT project. The aim was to achieve the formal verification of a whole computer system from application software down to the gate level (Paul et al., 2010). In Verisoft, all steps including compiler correctness and instruction set model are formally verified. All artefacts of the Verisoft project are implemented and formally specified, the majority of them are verified in pen-and-paper proofs, and approximately 75% of the proofs are mechanised in Isabelle/HOL. The primary code verification technique used is a generic environment in Isabelle/HOL for the verification of sequential, imperative programs using Floyd-Hoare-style logic (Hoare, 1969). The verification approach is based on refinement and equivalence between different abstraction levels. The Verisoft project is an impressive effort demonstrating the possibility of an entire trustworthy base for computing systems, but the project focuses on implementation correctness only and did not provide high-level security policies or high performance implementations.

seL4: The seL4 microkernel by Klein et al. (2009, 2014) is a general-purpose OS microkernel belonging to the L4 microkernel family (Liedtke, 1996). The seL4 microkernel emphasises security, performance, and reliability of embedded systems, and the kernel provides a full functional correctness proof. The design and verification projects for the seL4 microkernel were started concurrently in 2005 and the first major results were concluded in 2009.

Later, Klein et al. (2014) report on the comprehensive formal verification of the kernel including a functional correctness proof of the kernel's C implementation, a proof of the correct implementation at the binary level for the C semantics, a verified IPC fastpath, verified access-control enforcement, a proof of information-flow noninterference, a sound worst-case execution time analysis of the binary, and an automatic initialiser for user-level systems. Current work includes the construction of secure systems on top of the kernel (Klein et al., 2018).

The main verification strategy is *refinement* between different abstraction lay-

ers. The abstraction layers include: an abstract specification of the kernel, an executable specification generated from Haskell (a functional programming language), the high-performance C implementation of seL4 and the refinement proofs are machine-checked in the Isabelle/HOL theorem prover, with the exception of binary correctness, which uses the HOL4 theorem prover (HOL) and SMT solvers.

The verification of seL4 makes explicit assumptions about the correctness of the hardware, the correctness of TLB and cache flushing operations as well as the correctness of machine interface functions implemented in inline assembly. The seL4 microkernel has been verified for 32-bit ARMv6, 32-bit ARMv7-A with hypervisor extensions and 64-bit x86 processors. In summary, seL4 is the first general-purpose microkernel with a code level machine-checked formal proof, that is also being used in real-world scenarios.

CeriKOS: Another prominent effort to verify operating system kernels is the certified kit operating system (CertiKOS) project by Gu et al. (2011, 2016). The CertiKOS architecture focuses on the specification and verification of *concurrent* OS kernels. The project uses a compositional approach for the kernel design, i.e. a kernel with modular and individually certified constituent components.

The CertiKOS framework supports mainly two implemented systems: the mC2 kernel and the mCertiKOS hypervisor. The mC2 kernel is a concurrent OS kernel with fine-grained locking and thread yield/sleep/wakeup primitives. It can also act as a hypervisor and runs on x86 multicore machines. The mCertiKOS base kernel is a simplified uniprocessor kernel designed for the 32-bit x86 architecture. The mCertiKOS-hyp kernel, an extension of the base kernel, is a realistic hypervisor kernel. These kernels are implemented in C and assembly language.

The CertiKOS architecture uses *contextual refinement* as the main reasoning technique for the concurrent systems. As concurrency involves interleaved module execution, a strong contextual refinement property for modular systems states that each implemented module will behave according to its specification under any context with any valid interleaving. The CertiKOS verification framework is a layered design, with each layer individually partitioned into functionally independent blocks and collectively refining the underlying concrete layer. The multicore hardware features are also abstracted using different machine layers, and the functional correctness property implies that all system calls and traps will run safely and terminate eventually. The refinement proofs are carried out in the Coq theorem prover (Coq).

Similar to seL4, the CeriKOS verification explicitly lists the verification of TLB management as a limitation. The TLB is not modeled in the machine model, hence any code related to TLB management or TLB shootdown can not be verified.

Reasoning about Virtual Memory:
We now summarise the related work for reasoning about virtual memory primitives. Our focus in the survey will remain on the TLB.

The TLB has the nice property that *if* managed correctly it has no functional effect on the behaviour of the program. For this reason, all large-scale formal OS kernel verifications so far have left correct TLB management as an assumption. This includes the OS kernel verification work in seL4 and CertiKOS, which as mentioned above do reason about page table structures, but omit the TLB.

Kolanski and Klein (2008) present *mapped separation logic*: an extension of separation logic to formally reason about page tables, virtual memory access, and shared memory in Isabelle/HOL. The core logic is independent of particular page table implementations, and they provide a case study for verifying OS level page table manipulating code. Kolanski and Klein (2009) further extend their separation logic framework for reasoning about low-level C code in the presence of virtual memory, and instantiate this framework to a formal model of ARMv6 page tables. However, their model does not include the TLB and does not address TLB caching, consistency and invalidation. We build directly on the abstract interface to page table encodings Kolanski et al. have developed, which makes our work independent of the precise page table format the architecture uses.

Alkassar et al. (2008) provide a correctness proof of a kernel page fault handler in Isabelle/HOL. They model interleaved executions of the page fault handler written in a high-level programing language, and combine sequential reasoning about the page fault handler with low-level concurrent machine model using simulation proofs. They successfully prove that the page fault handler establishes a plain memory abstraction for the user, swapping in pages from disk as required. However, this work does not include TLB modeling and its reasoning.

In further work, Alkassar et al. (2010) present the functional verification of a small hypervisor (they call it *baby hypervisor*) using VCC, an automatic verifier for concurrent C programs. The baby hypervisor uses a 32-bit RISC architecture with single-level address translation (without TLB), and other features such as interrupt handling, privilege levels etc. The verification technique includes modeling of software/hardware interaction and simulation proofs in a first-order logic setting. The verification results feature initialization of the guest partitions, a simple shadow page table algorithm for memory virtualization and verification of the simulation of the guest partitions.

Later, Alkassar et al. (2010) outline their initial work for formalising an x64-like TLB for verifying TLB virtualization in a hypervisor setting. They carry out the verification of a virtualization algorithm (implementing shadow page tables) in VCC. They model the x64-like TLB as a set of complete and partial page table walks, and provide semantics of the abstract TLB such as address translation and invalidation operations. Alkassar et al. (2012) further extend and conclude the verification of TLB virtualization, with address space identifiers (ASIDs). The machine state contains a (processor local) TLB tagged with ASIDs, as well as an ASID register providing the active ASID. This work counts as the first effort to formalise the TLB for a modern hardware MMU. However, the focus of this effort is to verify the TLB virtualization, and as such they formulate a TLB abstraction

function on the host configuration to formulate the virtualised TLB for every guest. We also model the TLB as a set of page table walks, but our aim is to develop a generic reasoning framework for verifying programs (both kernel-level and user-level) in the presence of a TLB in general.

In the continuation of the above work, Kovalev (2013) also provides a TLB model, in particular a model of the Intel x64 TLB including selected maintenance operations and partial walks. The verification setting proves the correspondence between virtual and real TLB entries. Kovalev (2013) states a reduction theorem for page table walks in ASID 0 for a specific hypervisor setup. However, while other parts of this development are mechanised, this reduction theorem is not. Additionally, the restriction to one ASID makes the model too conservative for usual OS code.

Barthe et al. (2012) formalise a virtualization model similar to Xen on ARM, featuring the TLB and cache. The model, formalised in the Coq proof assistant, provides abstract reasoning about cache-based side-channels. Barthe et al. model the TLB as a partial map from virtual addresses to machine addresses, and provide axiomatic semantics of hypervisor actions. They then use the model for reasoning about cache-based attacks and countermeasures, and prove flush-enforced isolation between guest operating systems upon context switch. They also reason about the transparency provided by the virtualization model to the guest operating system. However, the work stops short of a program logic and a proof that the abstraction of TLB is sound.

Daum et al. (2014) reason about user-level programs on top of seL4, including page tables, but not about the TLB. Specifically, they distinguish user programs from each other by restricting the memory accesses of each user program within its virtual memory. This effort contributes towards proving the separation and security properties of user programs on top of seL4 kernel, but they do not model the TLB.

Dam et al. (2013) present a formal verification of information flow security for a simple ARMv7-based separation kernel, called PROSPER, in the HOL4 theorem prover. They provide guarantees for secure partitioning of different executions using the PROSPER kernel. The top-level specification features explicit communication between executing partitions. For obtaining the verification models, they extend the formal model of the ARM instruction set architecture by Fox and Myreen (2010) with a simple MMU. The MMU supports section-based one-level page tables only, without address translation (only a page table prototype).

Khakpour et al. (2013) verify security properties of the ARMv7 instruction set architecture (ISA) for user mode executions, establishing the main requirement for the PROSPER kernel verification. In particular, they provide instruction level noninterference and integrity properties in the HOL4 theorem prover, building on the ARM ISA model. Their framework includes a simple MMU with single-level page tables with translation, but it does not take caches, TLB, timing or hardware

extensions into account.

Nemati et al. (2015b) verify the isolation properties of a hypervisor that uses direct paging on the ARMv7 architecture to virtualize the CPU memory subsystem. They develop a formal CPU model in HOL4 by extending the formal ARM ISA model by Fox and Myreen (2010) with an MMU. The MMU model provides address translation through two-level page tables, and the CPU state includes system MMU registers. Nemati et al. (2015a) extend this work further to design, implement and verify an MMU virtualization platform for the ARMv7-A architecture. They demonstrate their hypervisor capability of hosting Linux as an untrusted guest. The focus of their work is to virtualize the ARMv7 memory subsystem, they do not model the TLB.

Bolignano et al. (2016) provide concrete and abstract models of a specific hypervisor, and prove isolation properties based on the abstract model. Bolignano et al. establish invariants on the concrete model, and then deduce the abstract model of the hypervisor (with focus on the page tables) from these invariants. This work provides guarantees about the page table mechanism, but not the TLB.

Baumann et al. (2017) propose an approach for the compositional specification and verification of system-on-chip (SoC) level security properties in a virtualization context. The compositional specification enables abstraction: SoC components are modeled as a communicating automata with abstract specifications. Baumann et al. demonstrate this approach through a case study on abstract specification and verification of an ARMv8 hypervisor. The TLB is not modeled yet.

Lutsyk (2018) provides a paper-and-pencil correctness proof for the pipelined multi-core implementation of the MIPS-86 ISA, extended with nested translation. This work models the TLB as a set of walks at the *bare hardware level*. Lutsyk's most abstract model is similar to the functionality and abstraction level of our most concrete TLB model. This is a good indication that our bottom-level model is a useful interface for future hardware verification. Our work here is concerned with raising the level of abstraction and enabling reasoning about the TLB from a software perspective.

Achermann et al. (2018) present a methodology for formalising physical addresses interconnections for Systems-on-Chip (SoCs), and demonstrate their reasoning methodology by modeling the physical address space of the MIPS R4600 TLB. They develop a refinement stack for reasoning about the physical address interconnects of the TLB, and conclude that the manufacturer's TLB specifications are imprecise and hence prevent any proof of correct initialization. Our work, on the other hand, focuses on the functional correctness of TLB management for low-level program executions.

As a case study in Chapter 8, we reason about the OS context-switching code recommended by the ARM architecture to avoid the TLB flush. Our focus is on TLB management during context switching. However, we are not the only one

to report on the formalisation of context management. Ni et al. (2007) present the mechanised verification of x86 context management code in the Coq proof assistant. The verification includes modeling of function calls, byte-addressed memory, conditional flags and stack but not the TLB.

Summary: The presented survey concludes that the TLB formalism *does* exist in the literature, *but* with the focus on reasoning about concrete hypervisor settings. The OS verification attempts explicitly *assume* the correct TLB management. We are, therefore, the first to soundly model a stack of abstraction for the TLB and to develop a generic logic for reasoning about programs in the presence of TLB-address translation.

1.3 Thesis Outline

In Chapter 2 we summarise the syntax and notation of Isabelle/HOL used in this thesis. Isabelle is a generic proof assistant, and Isabelle/HOL is an instantiation of Isabelle for higher-order logic (HOL). Isabelle offers advanced functional programming features such as type classes and record types. We use these features to craft our formalisation with a generic interface that we later instantiate for specific models. Our MMU formalisation is grounded in the formal model of the ARM instruction set architecture (ISA) by Fox and Myreen (2010). This model uses a state monad to formalise the state transformations, we use the same state monad to define the semantics of our MMU operations.

Chapter 3 outlines the virtual memory system of the ARMv7-A architecture, with a focus on its translation lookaside buffer (TLB). Virtual memory is a hardware-implemented abstraction over the physical memory that is managed by the operating system (OS) kernel. It enables the execution of multiple applications at the same time while sharing the same physical memory. The TLB is a constituent component of the virtual memory system, and its correct functionality is critical for the correctness of the overall system. This thesis is concerned with the correct OS kernel management of ARMv7-A's TLB and also examines the TLB's effect on program execution in general. This chapter serves as the background for the MMU related concepts of this thesis.

In Chapter 4 we develop an operational model of the ARMv7-A memory management unit (MMU) including the TLB in Isabelle/HOL. The MMU of ARMv7-A consists of a TLB that caches page table walks from the main memory under ASIDs. While our aim in this research is to model the ARMv7-A TLB with ASIDs, in this chapter we focus on a TLB that caches entries *without* ASIDs. We develop a base MMU model for such a TLB and identify the inherent reasoning complexities even this simple TLB would entail. We then provide a series of refinements for the base MMU model to stepwise abstract away the hardware details and to reach at an abstract MMU model that captures the essential TLB

functionality and is easier to reason about.

Chapter 5 builds on the MMU model of Chapter 4 to formalise the MMU with the TLB caching page table entries under ASIDs and global tags. We again build a refinement stack to abstract away the hardware details and also explain how our refinement framework handles the added features.

In Chapter 6 we integrate the MMU model of Chapter 5 with a separate page directory cache (PDC) to develop a two-stage TLB model caching the partial and complete page table walks. We again build a refinement stack to abstract away the hardware details and explain how our refinement framework handles the additional PDC.

Chapter 7 uses the model from Chapter 6 to present a logic for reasoning about low-level programs in the presence of TLB address translation. We define the syntax and semantics of a heap based language with necessary instructions for TLB management, we then present the Hoare logic rules for the operational semantics. We also provide simplification rules for memory write to further facilitate the program reasoning in the presence of TLB effects.

In Chapter 8 we apply the logic of Chapter 7 to extract invariants and conditions necessary to reason about the user-level and kernel-level executions, context switching and page table operations. This case study shows that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

Chapter 9 concludes the thesis. We summarise the novel contributions towards verifying low-level programs in the presence of cached address translation. We then provide the proof effort of our modeling and reasoning framework. The chapter concludes with the future research and engineering directions.

CHAPTER

TWO

Notation

In this chapter, we summarise the syntax and notation of Isabelle/HOL used in this thesis. Isabelle is a generic proof assistant, and Isabelle/HOL is an instantiation of Isabelle for higher-order logic (HOL).

Isabelle offers advanced functional programming features such as type classes and record types. We use these features to craft our formalisation with a generic interface that we later instantiate for specific models. Our MMU formalisation is grounded in the formal model of the ARM instruction set architecture (ISA) by Fox and Myreen (2010). This model uses a state monad to formalise the state transformations, we use the same state monad to define the semantics of our MMU operations.

This chapter is organised as: we begin by providing the notation of built-in types and their functions upon which our formalisation is based. We then summarise the constituent features of type classes, record types and the state monad.

2.1 Isabelle

Isabelle has a polymorphic meta-logic in which one can formalise different logics. The meta-logic itself is an intuitionistic higher-order logic and has connectives for implication, quantification and equality. The implication expresses logical entailment and is denoted by $\sigma \Longrightarrow \varphi$, the quantification expresses generality in logical entities and is denoted by $\bigwedge \mathtt{x}.\ \varphi\ \mathtt{x}$, while the equality expresses equivalence and is denoted by $\sigma \equiv \varphi$. The meta-implication constitutes theorems with one to multiple premises and a compound conclusion. In this document, we represent theorems involving meta-implication with this logical notation:

$$\frac{\sigma_1 \qquad \sigma_2 \qquad \sigma_3 \qquad \sigma_n}{\phi_1 \qquad \phi_2 \qquad \phi_m}$$

For the purposes of this thesis, meta-quantifier is equivalent to the higher-order logic quantifier ($\forall$) and it is safe to assume that meta-equality is the same as normal equality (=).

Isabelle supports a polymorphic type system. Type variables are written 'a, 'b, etc. The notation $\mathtt{t}::\tau$ means that term t has the type τ. By convention, type names are usually lower case.

Functions in Isabelle are total and Isabelle denotes the space of total functions by $\upsilon \Rightarrow \tau$, where υ and τ are the arguments and return types. Application of a function f to arguments x and y is written as f x y, which yields two function applications resulting in functions at each stage, i.e. ((f x)y). Isabelle also provides lambada notation, e.g. λx. x = y, custom syntax for constants, and infix

operators, e.g. `a + b`.

2.2 HOL in Isabelle

We now cover standard features of the HOL instantiation of Isabelle. This instantiation also equips us with the features of functional programming for constructing proofs.

2.2.1 Types, Terms and Formulae

The Isabelle/HOL type system provides base types, type constructors, function types and type variables. Examples of the base types include `bool` (for truth values), `nat` (for natural numbers) and `int` (for integers). Types in Isabelle usually have one or more constructors, for example, type `bool` has two constructors `True` and `False`. By convention, constructor names start with a capital letter. Each constructor of a type is unique, for example, the constructors `True` and `False` of type `bool` are distinct.

In HOL, formulae are terms of the type `bool`. Logical connectives combine terms to make a formula, for example, equality with the type `bool ⇒ bool ⇒ bool` works as "if and only if" connective for any given two terms of type `bool`.

Isabelle/HOL enables the users to define their own polymorphic types with one or more constructors using the command **datatype**. For example, consider the type `mytype`:

```
datatype ('a, 'b) mytype  =  Constructor1
                          |  Constructor2 'a
                          |  Constructor3 'a ⇒ 'b ⇒ bool
```

This declares the data type `mytype` as a polymorphic type having three unique, disjoint and injective constructors $\texttt{Constructor}_1$, $\texttt{Constructor}_2$ and $\texttt{Constructor}_3$. The $\texttt{Constructor}_1$ has no arguments, while $\texttt{Constructor}_2$ and $\texttt{Constructor}_3$ have a single argument each of types `'a` and `'a ⇒ 'b ⇒ bool` respectively. Every new type comes with automatically derived structural induction and simplification rules for its constructors. Already defined types are abbreviated using **type_synonym**, for example:

```
type_synonym string = char list
```

2.2.2 Higher-Order Logic Operations

Isabelle/HOL formalises the standard operations of higher-order logic:

Operation	Type	Notation
Negation	`bool ⇒ bool`	`¬ P`
Conjunction	`bool ⇒ bool ⇒ bool`	`P ∧ Q`
Disjunction	`bool ⇒ bool ⇒ bool`	`P ∨ Q`
Implication	`bool ⇒ bool`	`P ⟹ Q`
Universal Quantification	`('a ⇒ bool) ⇒ bool`	`∀x. P x`
Existential Quantification	`('a ⇒ bool) ⇒ bool`	`∃x. P x`

Table 2.1: Higher-Order Logic Standard Operations

2.2.3 Built-in Types used in this Thesis

We now briefly present the notation of Isabelle/HOL's built-in types that we use in this thesis. We also provide the standard notation of the basic functions defined over these types.

Type `set`: In Isabelle/HOL, a set is formalised as the polymorphic type `'a set`, where `'a` is the type of the elements of the set. The empty and universal sets are represented by ∅ and `UNIV` respectively. Set enumeration has the form $\{e_1, e_2, \ldots, e_m\}$, and set comprehension has the form `{x | P x}`. Set operations come with the usual notation and are summarised in Table 2.2.

Operation	Notation	Operation	Notation
Membership	`x ∈ A`, `y ∉ A`	Subset	`A ⊆ B`, `A ⊂ B`
Union	`A ∪ B`	Intersection	`A ∩ B`
Infinitary Union	`⋃A`	Arbitrary Intersection	`⋂A`
Difference	`A - B`	Complement	`- A`

Table 2.2: Notation for Set Operations

The image of a function `f` over a set `A` is represented as ``f ` A`` and is defined as `{y | ∃x∈A. y = f x}`, infinitary union `⋃A` as `{x | ∃B∈A. x ∈ B}`, and arbitrary intersection `⋂A` as `{x | ∀B∈A. x ∈ B}`. `range f` is the set of values returned by function `f`, i.e. `range f = {y | ∃x. f x = y}`. The interval set between countable objects is represented as `{l..u}`, and is defined as `{x | l ≤ x} ∩ {x | x ≤ u}`. For example, `{(1::nat) .. 3}` represents the set `{(1::nat), 2, 3}`.

Type `list`: The type `'a list` of Isabelle/HOL formalises the list of elements of the type `'a`, with the two constructors `Nil` (`[]`) and `Cons` (`#`):

```
datatype 'a list  =  []  |  'a # ('a list)
```

[] represents an empty list, and # puts an element of the type 'a in front of a list of the type 'a list. Table 2.3 summarises the basic HOL functions over lists.

HOL Function	Purpose
hd	gets the first element,
tl	gets the rest,
length	calculates the list length,
set	constructs a set from the lists's elements,
map	applies a function to all list elements,
zip	creates a list of pairs from two lists, and
foldl	reduces a list to one element by applying a folding function from left to right.

Table 2.3: Basic HOL Functions for List

Type word: Isabelle's type system does not include dependent types, but can encode numerals and machine words of fixed length. The type 'n word represents a word with n bits, for example, 32 word and 64 word. We use the type 'n word to foramlise machine words. The notation of the basic bitwise operations is summarised in Table 2.4.

Operator	Notation	Operator	Notation
Bitwise Not	NOT n	Bitwise And	n AND m
Bitwise Or	n OR m	Shift Left	n << m
Shift Right	n >> m	Nth Bit as a Bool	m !! n

Table 2.4: N-bit Operators

The function UCAST('n → 'm) casts an n-bit word to another m-bit word. For example, a 12-bit word is converted to its 32-bit equivalent word as: ucast (w:: 12 word)::32 word. The function size returns the bit-length of a word in type nat. For example, size (5 :: 3 word) = 3. Finally, the function mask x returns an 'n word with the bottom x bits set to one, for example (mask 2) :: 4 word = 3 :: 4 word.

Type pair: An ordered pair of two types 'a and 'b is formalised as the type 'a × 'b. The term (a, b) represents an object of the pair type, where a and b are terms of types 'a and 'b. The first element of a pair is accessed with the function fst, the second with the function snd. Tuples are the right-associative nesting of pairs. For example, (a, b, c) has the type 'a × 'b × 'c, which is internally represented by 'a × ('b × 'c).

Type option: The type

```
datatype 'a option = None | Some 'a
```

adjoins a new element None to a type 'a. We use 'a option to model partial functions, writing ⌊a⌋ instead of Some a and 'a ⇀ 'b instead of 'a ⇒ 'b option. The

`Some` constructor has an underspecified inverse called `the`, satisfying the equation `the ⌊x⌋ = x`. The domain of a partial function is obtained by `dom m = {a | m a ≠ None}`.

With this we conclude presenting the Isabelle/HOL's built-in types used in this thesis.

2.2.4 Function Update and the `Let` Construct

Function update for a total function `f` is denoted as `f(x := y)` where `f::'a ⇒ 'b`, `x::'a` and `y::'b`. For partial functions, we write `f(x ↦ y)` representing `f (x:= Some y)`.

The `Let` construct improves readability by allowing us writing terms as:

```
let  y = f x ;
     z = g y
in   h y z
```

instead of: `h (f x) (g (f x))`.

We have summarised the essential features of Isabelle/HOL and their notation. While presenting our formalism later, we may repeat some of the concepts presented in this chapter for the reader's ease. We now proceed to explain advanced features of Isabelle: type class and record types. We also explain the constituent functions of the ARM ISA model's state monad.

2.3 Type Classes in Isabelle

Isabelle supports type classes: a type class is essentially a set of types with a common interface. The interface of a type class is categorised by class axioms and class functions. All types that are instances of a specific class must obey the axioms and provide the functions of the interface. The usual benefit of type classes is that they allow overloading, i.e. a constant may have multiple definitions at different types, for example, the operator `+` for types `nat`, `int`, `'n word`, etc. The axioms of a class also allow us to reason on the level of the class.

For example, we declare an example type class `ex_typ` for `'a list` that has an axiom `sum` and a function `tail_sum` as:

```
class ex_typ =
fixes sum :: 'a list ⇒ 'a list ⇒ 'a

definition tail_sum :: 'a
tail_sum l ≡ sum l (tl l)
```

We then simply instantiate the type class `ex_typ` to types `nat` and `int` for different semantics of the axiom `sum`:

```
instantiation nat :: ex_typ
begin
sum l l' ≡ hd l + hd l'
instance ..
end
```

```
instantiation int :: ex_typ
begin
sum l l' ≡ hd l + 1
instance ..
end
```

The `sum` is then accessed by type inference: a `sum` expression with `nat list` picks up the `sum` function for the `nat`, while the `int` picks up the `int` type instantiation. Both types `nat` and `int` provide the semantics of `tail_sum` function of type class `ex_typ`.

Type classes can be extended and organised in a hierarchy. For example, we can define a type `ex_typ_extended` as an extension of our example type class `ex_typ` with a `prod` parameter:

```
class ex_typ_extended = ex_typ +
fixes prod :: 'a list ⇒ 'a list ⇒ 'a
```

All the instantiations of the type class `ex_typ_extended` obey the parameters `prod` and `sum`.

We make use of Isabelle's type class support to build our MMU model, we also instantiate this generic MMU interface with states having different TLB abstractions. This setup helps us greatly in the modeling of our refinement framework.

2.4 Record Types in Isabelle/HOL

We now briefly explain Isabelle/HOL's record types, their notation and basic functions. The Cambridge ARM formalisation (Fox and Myreen, 2010) models the CPU state as a record type.

A record type of Isabelle/HOL is a collection of fields, with each field having a type which may by polymorphic. There is a selector and an update function for every record field. Isabelle/HOL also supports extensible records, i.e. new record

types can be defined by extending the existing record types (Naraschewski and Wenzel, 1998). We can also define custom operations over the record types.

We illustrate these concepts with an example. Suppose we want to model a state having a memory and a list of registers. For that we define a record type `ex_state`:

```
record  ex_state   =
        memory     ::  8 word ⇒ 32 word
        registers  ::  32 word list
```

The type `ex_state` is a record type with the fields: `memory :: 8 word ⇒ 32 word` and `registers:: 32 word list`. The selector and update functions for these fields have the same names. For example, if `s` has type `ex_state` then `memory s` denotes the value of the `memory` field of `s`, and `s⦇memory := id⦈` will update `memory` of `s` to be the identity function `id`.

Every record structure has an implicit but accessible pseudo-field, `more`, that keeps the extension as an explicit value. Its type is declared as completely polymorphic: `'a`. Our definition of `ex_state` above has generated two type abbreviations:

```
ex_state = ⦇memory :: 8 word ⇒ 32 word, registers :: 32 word list⦈
'a ex_state_scheme = ⦇memory :: 8 word ⇒ 32 word,
                      registers :: 32 word list, ... :: 'a⦈
```

The type `ex_state` is for fixed records having exactly the two fields `memory` and `registers`, while the polymorphic type `'a ex_state_scheme` comprises all possible extensions to these two fields. Now if we want to extend the original `ex_state` with a program counter register, we can either do:

```
record ex_state_extended =  ex_state_extended +
                            program_counter :: 32 word
```

or simply:

```
type_synonym ex_state_extended = 32 word ex_state_scheme
```

The built-in functions `extend` and `truncate` are used to extend and truncate the record types respectively:

- The function `extend` takes two arguments: a record to be extended and a record containing the new fields.
- The function `truncate` takes a record and returns a fixed record, removing any additional fields.

In our formalism, we extend the record type `state` of the ARM ISA model to introduce the TLB model. Our modeling framework makes extensive use of record types.

2.5 State Monads

The ARM ISA formalisation (Fox and Myreen, 2010) uses a state monad to model state transformers. A state monad generally encodes a purely functional model of computation with side effects. In the ARM model, the associated monad type for the result type `'r` and the state type `'s` is `'s ⇒ 'r × 's`. This type is abbreviated `('s, 'r) state_monad`, i.e. a function from the current state to the next state together with the computation result. A pure state transformer is typically denoted by the one-valued result type `unit`, i.e. `'s ⇒ unit × 's`. The two monad constructors `return` and `bind` are defined as follows:

```
return :: 'r ⇒ ('s, 'r) state_monad
return r ≡ λs. (r, s)

bind :: ('s,'a) state_monad ⇒ ('a ⇒ ('s,'b) state_monad) ⇒
        ('s,'b) state_monad
f ≫= g ≡ λs. let (r, s') = f s in g r s'
```

The constructor `return` simply injects the value `r` into the monad type, passing the state unchanged, while `bind` sequentially composes a computation `f`, and a computation `g` (a function from the return type of `f`). We occasionally write `bind f g` as `f ≫= g` and use the `do` syntax for longer computations.

```
f ≫= g ≡ do {   x ← f; g x }
```

For fetching and updating a particular parameter from the state, the ARM model uses the `read_state` and `update_state` functions (sometimes also called `gets` and `puts`):

```
read_state f ≡ λs. (f s, s)
update_state f ≡ λs. ((), f s)
```

We abbreviate multiple `read_state` calls into tuple notation, for example
`(a,b) ← read_state (f,g)`.

With this we conclude the chapter for notation: we have summarised the essential features of Isabelle/HOL and their notation. While presenting our formalism later, we may repeat some of the concepts presented in this chapter for the reader's convenience.

CHAPTER

THREE

Virtual Memory in the ARMv7-A Architecture

This chapter outlines the virtual memory system of the ARMv7-A architecture, with a focus on its translation lookaside buffer (TLB). Virtual memory is a hardware-implemented abstraction over the physical memory that is managed by the operating system (OS) kernel. It enables the execution of multiple applications at the same time while sharing the same physical memory. The TLB is a constituent component of the virtual memory system, and its correct functionality is critical for the correctness of the overall system. This thesis is concerned with the correct OS kernel management of ARMv7-A's TLB and also examines the TLB's effect on program execution in general. This chapter serves as the background for the MMU related concepts of this thesis.

This chapter is organised as follows: after highlighting basic concepts of virtual memory systems in general, we describe hardware features of the ARMv7-A virtual memory system in particular, including pages, page tables, TLB and caches. Finally, we summarise the role of an OS kernel in the management of the virtual memory system with specific focus on TLB management.

3.1 Basic Concepts of Virtual Memory

Virtual memory is at the basis of protected-mode operating systems; a virtual memory system provides:

- segmentation of the main memory,
- variable address space for applications,
- execution of multiple applications at the same time,
- isolation between user-level processes,
- authorized user-level accesses to the main memory,
- controlled and supervised memory sharing between applications,
- framework for kernel management of memory resources,
- localized memory management, and
- security mechanisms by associating access properties to memory locations through central memory management.

Different hardware architectures together with OS kernels implement a virtual memory system in various ways. However, at the core of all these implementations, the processes being executed cannot access main memory (RAM) directly through physical memory addresses. Whenever an active process has to operate on the main memory, whether it be reading or writing, it issues the operation with a *virtual* memory address. The memory management system then *resolves* that virtual address to the actual (*physical*) address.

The main memory itself often comes with a partitioning mechanism. Paging and

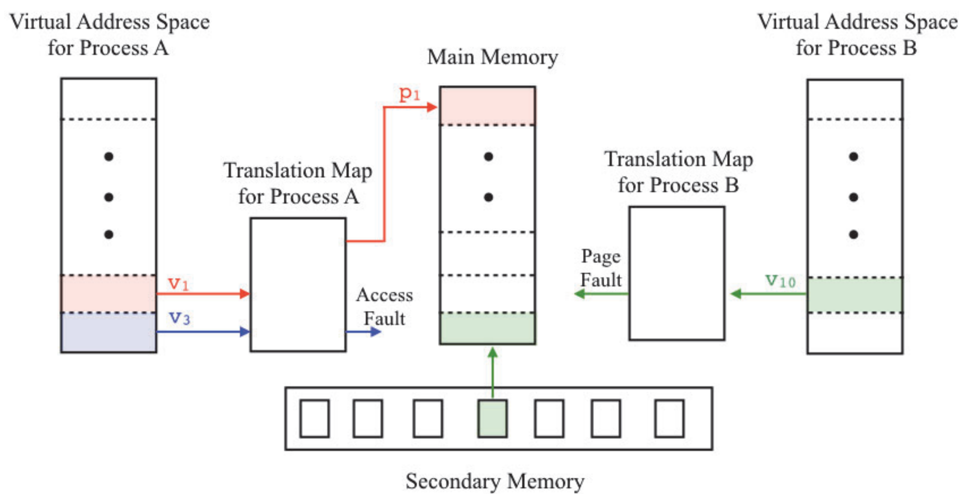

Figure 3.1: Basic Concepts of Virtual Memory

segmentation are two well-known such memory partitioning schemes. In paging, the hardware architecture divides the main memory into small fixed size frames. The OS kernel then divides each process into frame-sized chunks called pages. When a process is not active, its respective pages reside in the secondary memory. Whenever a process is required to be brought in for processing, the relevant set of its pages are loaded by the hardware into the available frames of main memory on demand. Whereas in the segmentation scheme, the main memory has variable sized segments and their size depends on the processes. While segmentation avoids internal fragmentation in the main memory, it is inefficient when it comes to the execution of multiple applications; c.f. (Stallings, 2008, Chapter 8). We omit any discussion of segmented memory architectures. The x86 architecture is segmented, but the segments are typically identical in size. ARM and PowerPC are prominent architectures (among many others) that use paged memory management.

Figure 3.1 gives an overview of a paged memory environment. The main memory has fixed size frames, and the processes `A` and `B` have their separate virtual address spaces. The virtual address space of a process is an abstraction of the memory that the process can access exclusively. A virtual address space is set up by the OS kernel by dividing the data of the process in frame-sized pages. The range of the virtual address space depends on the hardware's instruction set architecture. For example, the ARMv7-A architecture provides a flat address space of 2^{32} 8-bit bytes, covering 4GBytes; c.f. (ARM, 2008, Chapter A3).

Suppose that the process `A` needs to write a value to the memory location at address $\mathtt{v}_1$ of its virtual address space. The process `A` simply issues the memory write operation with the virtual address $\mathtt{v}_1$. The memory management system then *resolves* the virtual address $\mathtt{v}_1$ to the corresponding physical address $\mathtt{p}_1$ using the translation map for the process `A`. The translation map of a process, known as its page table, encodes the address translation from virtual to physical addresses. In most widely deployed architectures, the page tables are hardware implemented

data structures that reside in the main memory and are managed by the OS kernel. They also encode access permissions for the virtual addresses, for example, an exception would be raised when the process `A` tries to write at the read-only address v_3. When the process `B` accesses the unmapped virtual address v_{10}, an exception would be raised by the hardware. Depending on the exception handler, the OS kernel could then load the respective page into the main memory and update the page table for that process accordingly.

In summary, we have briefly highlighted the virtual memory system main concepts such as paging and address translation. We now describe the key features of ARMv7-A virtual memory system architecture (VMSA) that are relevant to this thesis.

3.2 Virtual Memory System in the ARMv7-A Architecture

The ARM architecture provides multiple address translation modes that differ in the number of translation levels and bit-length of virtual addresses. Without loss of generality for the treatment of TLBs we focus on one of these modes in this thesis — the TLB management in others is analogous. This mode is called *short-descriptor translation table format*.

3.2.1 Pages

As mentioned in the previous section, ARM is a paging architecture. The short-descriptor translation table format supports four types of pages and correspondingly four types of frames; c.f. (ARM, 2008, Chapter B3):

- **Small Page**: a 4KB block of memory,
- **Large Page**: a 64KB block of memory,
- **Section**: a 1MB block of memory, and
- **Supersection**: a 16MB block of memory.

Small and large pages provide fine granularity for mapping and accessing pages, and they are used effectively to map smaller applications, whereas sections and supersections allow mapping of a larger region of memory. An important feature of the ARM architecture's paging mechanism is that frames in main memory can overlap, i.e. a larger frame can contain smaller pages at the aligned locations. Figure 3.2 describes an example of a paged memory layout.

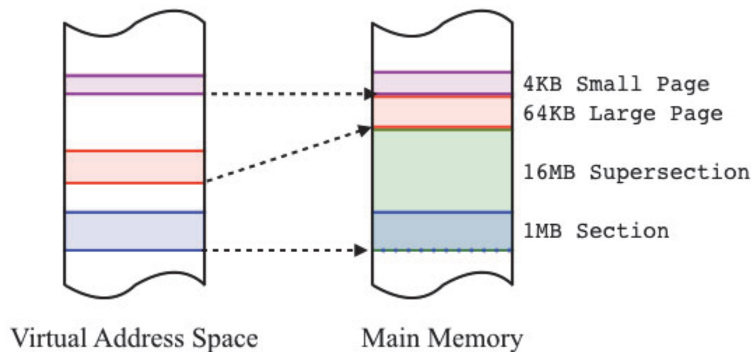

Figure 3.2: An Example of Paged Memory in the ARM Architecture

3.2.2 Page Tables

Page tables provide translation of virtual addresses to physical addresses. In the short-descriptor table format, two-level page tables are held in memory for processes:

First-level Page Table: holds first-level descriptors, also called page directory entries, that contain either
- the physical base address of the frame and translation properties for a section or a supersection; or
- translation properties and pointers to a second-level table for a large page or a small page.

Second-level Page Table: holds second-level descriptors, also called page table entries, that contain the base address and translation properties for a small page or a large page.

The first-level page table is located in the main memory by its root address. For a `32-bit` machine, the root of a page table is a `32-bit` physical address. Depending on the configuration of the VMSA, either register TTBR0 (an acronym for translation table base register) or register TTBR1 holds the root of the page table of the active process. Figure 3.3 shows an example of a two-level page table.

A descriptor of the page table is either:

- an invalid entry, or
- a pointer to the root of a next-level page table with translation properties, or
- a base entry that defines the base address of a memory page and its access properties, or
- a reserved format.

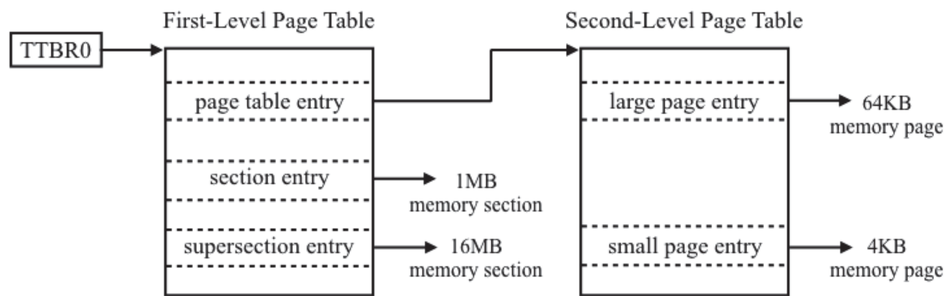

Figure 3.3: An Example of Two-Level Page Table in the ARM Architecture

3.2.3 Address Translation

We now explain how address translation works, using the two-level page table of the ARM architecture. Figure 3.4 represents the translation flow of a `32-bit` virtual address v_1 belonging to a section, i.e. a lookup that terminates at the first-level.

While resolving the virtual address v_1, the memory management unit (MMU) locates the respective page table entry by combining the TTBR0 register and the upper `12 bits` of the virtual address v_1. These bits provide the `12 bits` page table offset. In this case, the resultant page table entry is a section entry, and it contains the base address and permission bits of the section the virtual address v_1 belongs to. At this point, the MMU checks the permissions for the section against the active state of the processor. If it grants the access, the final physical address is formulated by combining the `12 bits` physical base address provided by the page table entry and the section offset provided by the lower `20 bits` of the virtual address v_1.

The translation flow for a supersection is analogous to that of a section: the difference is that now the first-level page table entry signifies the supersection it

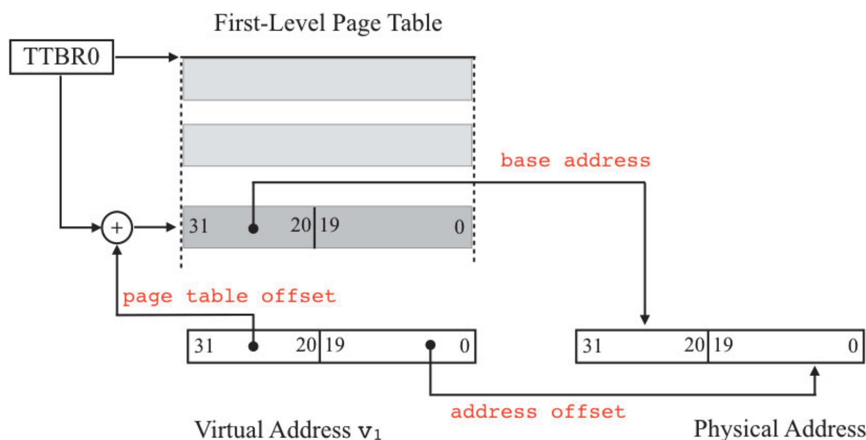

Figure 3.4: Translation Flow for a Section

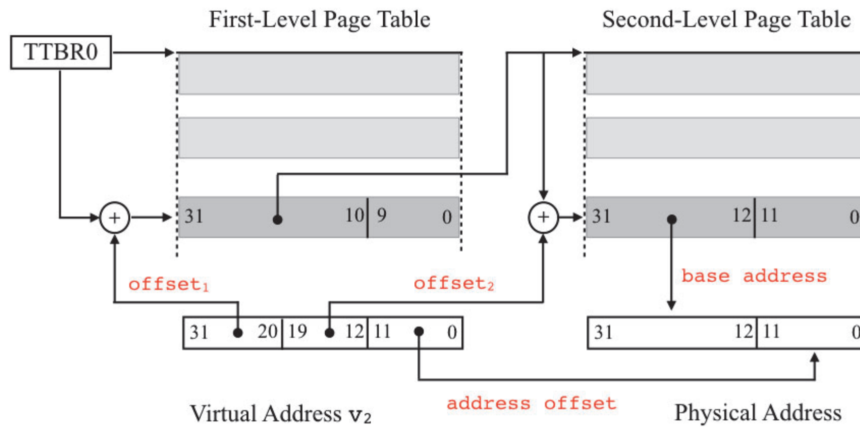

Figure 3.5: Translation Flow for a Small Page

belongs to, and the lower `24 bits` of the virtual address are supersection offset. Virtual addresses belonging to small or large pages undergo two-level page table translation. Figure 3.5 shows the translation flow of a `32-bit` virtual address $\texttt{v}_2$ belonging to a small page.

In this case, the first-level descriptor (found using the upper `12 bits` of $\texttt{v}_2$) provides a pointer to the second-level page table. The MMU then traverses the second level pages table using the next `8 bits` of the virtual address $\texttt{v}_2$ to locate the respective page table entry. The MMU also checks the permission and configuration settings at both levels of the page table. Finally, the base address is combined with the page offset to retrieve the final physical address. The translation flow for a large page is analogous to that of a small page, the ARM manual documents more details (ARM, 2008, Chapter B3).

3.2.4 Translation Lookaside Buffer

In a virtual memory system, a translation lookaside buffer (TLB) is a hardware cache for storing address translations (page table entries). Without further help, this central mechanism is slow: main memory is already significantly slower than the processor, and traversing a page table can cost up to three memory accesses. The TLB caches such lookups, and significantly reduces the number of such memory accesses.

Figure 3.6 shows the use of a TLB in the address translation process. While resolving a virtual address, the CPU first checks the TLB. If the respective page table entry is present in the TLB (a *hit*), the CPU directly generates the physical address avoiding the page table walk. If the respective page table entry is not present inside the TLB (a *Miss*), the CPU does the page table walk. If the respective page is mapped in the main memory, the CPU generates the physical address and also reloads the TLB for future use. Otherwise, the page fault handler of the OS kernel is called. It will usually map the respective page in the main

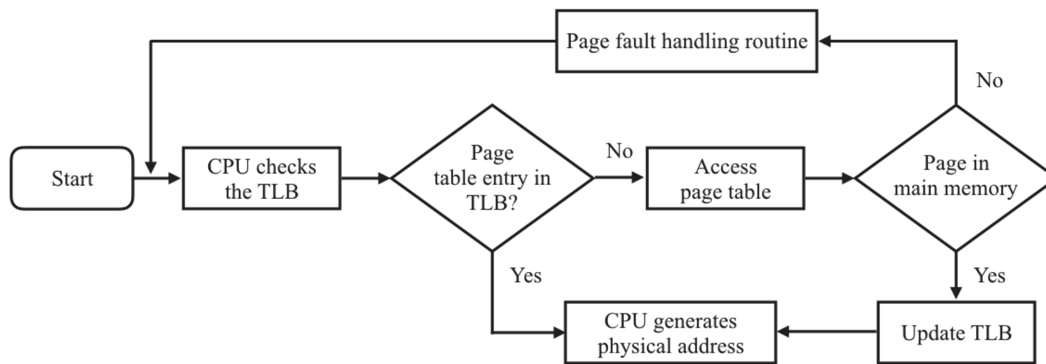

Figure 3.6: Functional Role of the Translation Lookaside Buffer (TLB)

memory, update the page table and the TLB with the page table entry, and finally generate the physical address. A TLB, in general, is either hardware-loaded or software-loaded. For a hardware-loaded TLB, the hardware does the page table walk and fills the TLB on a miss as explained above for the ARM architecture. In contrast, in a software-loaded TLB, the OS kernel is responsible for page table walk and loading the TLB.

The ARM architecture considers the TLB as an implementation technique; hence the manual (ARM, 2008) describes the TLB as a black box, i.e. by its external interface only. It does not specify its size, the replacement strategy, or exact internal state. The architecture only defines certain principles for the TLB that implementations must provide and that OS kernels can use for TLB management. We now summarise these features:

Process-Specific TLB Entries: In a multitasking environment, usually each process has its own page table; hence multiple page tables are present in the main memory at a time. The TLB being a cache of these page tables must implement some mechanism to distinguish between their entries. One approach can be that the TLB caches only entries for the *active* process. Subsequently, the TLB would be *flushed* during the context switch between processes. Flushing of the TLB at every context switch has an adverse effect on the overall speed of the system. Instead, the ARM architecture associates a process-specific tag with translation entries of the TLB. This tag is called an address space identifier (ASID). With ASIDs, translation entries from different processes co-exist in the TLB, and new entries can be cached without removing the previous mappings. The ARM architecture provides 8-bit ASIDs, which means, the TLB can cache entries for up to 256 processes; c.f. (ARM, 2008, Chapter B3).

Global TLB Entries: In a virtual memory implementation, the OS kernel can divide the main memory into global and non-global regions: any pages mapped in the global regions are accessible by all processes, whereas non-global regions have ASIDs associated to them. To enable this distinction, the ARM architecture associates an nG (non-Global) bit with every TLB entry:

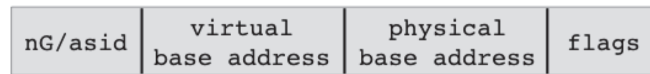

Figure 3.7: A Format for TLB Entries

- nG==0 implies the TLB entry is global, and
- nG==1 means the TLB entry is non-global (or ASID-specific).

TLB Matching: Although the ARM architecture does not provide any concrete details about the structure of a TLB entry, conceptually a TLB entry is of the form represented in Figure 3.7. A TLB entry essentially contains a physical base address of a memory page corresponding to its virtual base address and an ASID tag. It also encodes permissions for the respective memory region. The TLB itself is then a memory structure that is accessed by virtual addresses together with their ASIDs.

Figure 3.8 provides an overview of a TLB lookup resulting in a *hit*, i.e. a *unique match* of the given virtual address along with the active ASID with only one TLB entry. A TLB entry matches a virtual address when the entry's ASID equals the active ASID *and* the address's top `12 bits` for section or `20 bits` for small page match the virtual base address of the entry.

If a TLB does not contain any matching entry for a given virtual address, its TLB lookup results in a *Miss*. The ARM architecture provides only hardware-loaded TLBs, i.e. the processor does the page table walk on a TLB miss and it also reloads the TLB for future use. Architectures such as MIPS have a software-loaded TLB where the OS kernel is responsible for page table walk and loading the TLB in case of a miss (MIP, 2015).

Although ARM's TLB is hardware-loaded, it is not entirely invisible to software. The OS kernel is responsible for maintaining the consistency of the cached TLB entries with the page tables in the memory. Any update to the page table by

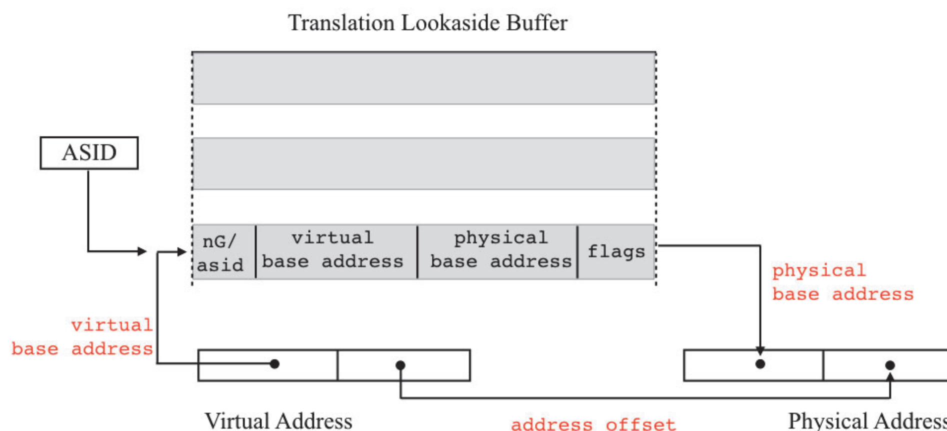

Figure 3.8: An Example of a TLB Lookup Resulting in a Hit

the OS kernel must be reflected in the TLB, and no more than one TLB entry should provide the address translation for a given virtual address and its ASID. An inconsistent TLB gives rise to unpredictable behaviour and may result in a crash of the overall virtual memory system.

TLB Maintenance Operations: The ARM architecture provides certain TLB maintenance operations to the OS kernel in order to *flush* entries from a TLB. The TLB can potentially store any page table entry that does not generate translation fault, therefore the OS kernel must maintain the TLB's consistency between updating mapped page table entries and accessing memory locations whose translation is determined by those entries. The general TLB maintenance operations provided by the ARM architecture are:

- invalidate all entries in the TLB,
- invalidate a single entry covering a range of virtual addresses for all ASIDs,
- invalidate a single entry covering a range of virtual addresses for a specified ASID, and
- invalidate all TLB entries matching a specified ASID.

Additionally, the OS kernel implements special TLB maintenance while updating the ASID and the page table root registers that we describe in Sect. 3.3.

TLB Lockdown: The ARM architecture has a concept of TLB lockdown, i.e. a *locked* entry is guaranteed to remain in the TLB unless explicitly flushed. We do not formalise locked TLB entries in the thesis, although our reasoning framework is easily applicable with minor modifications to an implementation supporting the TLB lockdown.

TLB Implementation Details: Different implementations of ARMv7-A are allowed to implement the TLB's architectural features differently. In an implementation, usually the TLB is split up into levels for increasing the speed of the lookup. For example, Cortex-A9 has two levels with three TLBs:

- **Instruction Micro-TLB**: 32 or 64 fully associative entries
- **Data Micro-TLB**: 32 fully associative entries
- **Unified Main-TLB**: 2-way associative $2\text{x}n^2$ entry TLB, where $n \in \{5, 6, 7, 8\}$

Micro-instruction and micro-data TLBs constitute the first-level of page table caching, and they provide a fully associative lookup in a single clock cycle. The main TLB makes the second-level, and it catches the misses from the micro TLBs. TLB maintenance operations of the ARM architecture maintain these three TLBs. Table 3.1 summarises TLB's implementations in other ARMv7-A's processors.

Additionally, higher implementations of ARMv7-A such as Cortex-A15 have dedicated caches to store intermediate levels of page table entries; c.f. (ARM, 2013, Chapter 5). These intermediate TLBs are called page directory caches (PDCs).

Processor Cores	Instruction TLB	Data TLB	Unified TLB
Cortex-A5	10 entries	10 entries	128 entries, 2-way set-associative
Cortex-A7	10 entries	10 entries	256 entries, 2-way set-associative
Cortex-A15	32 entries	32 entries	512 entries, 4-way set-associative
Cortex-A17	10 entries	10 entries	256 entries, 2-way set-associative

Table 3.1: TLBs in ARMv7-A Implementations

For a two-level page table, this setting gives us two stages of TLB caching. The first stage that we refer to as simply TLB, caches entries that provide end-to-end address translations, i.e. results of complete page table lookups, while the second stage (PDC) caches the results of partial page table lookups – up to the first-level traversal of the page table only. Figure 3.9 gives an overview of the page table lookup and respective information cached in the two-stage TLB for a virtual address mapped to a small page of the memory. Here, the TLB caches the physical base address of the small page the virtual address resolves to, while the PDC stores the pointer to the second-level page table. The ARM TLB maintenance operations work on both TLB and PDC.

With this, we have covered the aspects of ARMv7-A TLB hardware features necessary for this thesis. The main features not covered are related multiprocessor effects on TLB maintenance operations. For more details, please refer to the reference manual (ARM, 2008).

3.2.5 Caches

This thesis focuses on modeling the TLB and examining its effect on program verification and does not investigate the effects of caches in the process. However, for completeness, we briefly explain the role of caches in ARMv7-A VMSA.

The purpose of caches in any memory system is well-known: providing data to

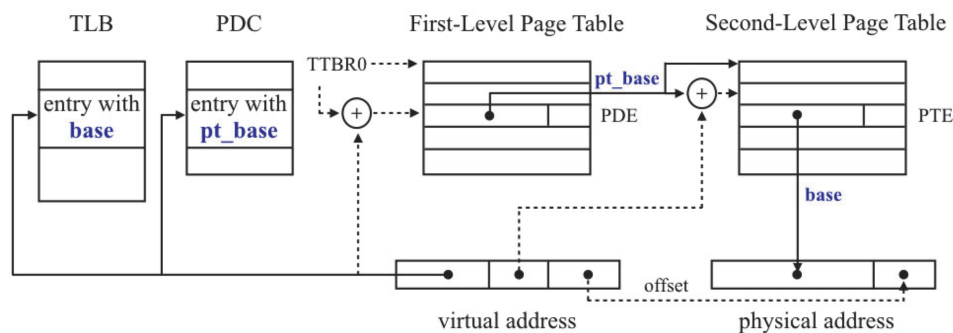

Figure 3.9: TLB, PDC and Page Table Lookup

the processor at a higher speed and avoiding traversal of larger memory structures for data operations. Cache implementation and organisation is a large field with several features and mechanisms; from the point of view of virtual memory, however, the dominant feature is the placement of a cache. In the cache hierarchy of a memory system, any cache present before the address translation is virtually-addressed (also called virtually-indexed), and the caches placed after are physically-addressed (also called physically-indexed). A TLB itself is a virtually-addressed cache dedicated to storing page table walks. The cache placement also determines the responsibilities of the OS kernel for its management.

The ARMv7-A architecture provides cache levels; each core in the implementation has its dedicated physically-addressed L1 (level-1) cache, whereas the L2 cache is shared among the cores. An implementation is free to choose separate data- and instruction-L1 caches, similar to that of the TLB. Figure 3.10 shows an example of ARM's cache structure for a uni-core processor. The architecture provides the required cache maintenance operations for both levels. More details are available in the manual; c.f. (ARM, 2008, Chapter B3).

In summary, we have detailed the hardware aspects of ARMv7-A virtual memory system in this section and our focus has remained on the TLB and its constituent features. We now explain responsibilities of the OS kernel towards the ARM virtual memory system.

3.3 OS Kernel Management of ARM's VMSA

The ARMv7-A architecture provides advanced hardware features for virtual memory; however, it leaves the setup of virtual address space, page table management and TLB and cache maintenance for the OS kernel. In this section, we explain the virtual address space setup for processes, the page table management and the TLB maintenance of a toy OS kernel that is inspired by the seL4 microkernel (Klein et al., 2009). These configurations also apply to all major protected-mode OS kernels, e.g. Linux, Windows, MacOS, as well as most microkernels for the ARMv7-A architecture. We omit the discussion of cache maintenance as our focus in this

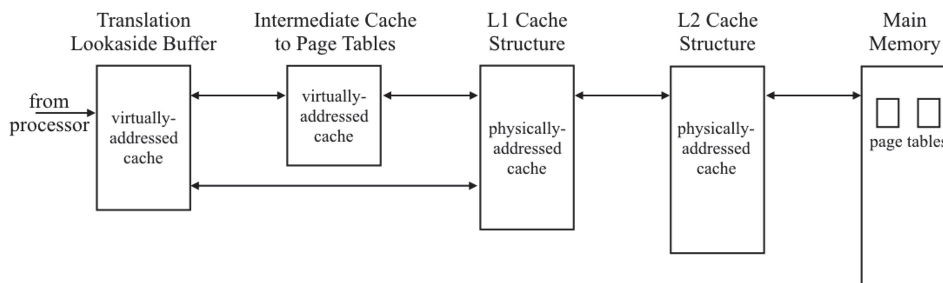

Figure 3.10: Cache Hierarchy in the ARMv7-A Architecture

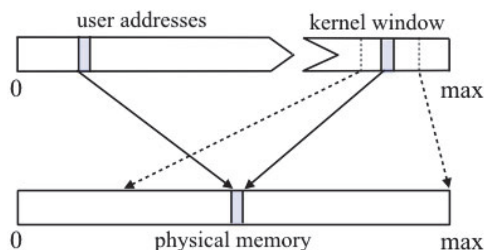

Figure 3.11: An Example of a Virtual Address Space with Kernel Window

thesis is on TLB maintenance. This section also serves as the background for our case studies in Chapter 7.

Virtual Address Space Setup: Our toy OS kernel sets up the virtual address space for every user-level process. The main assumption in setting up an address space is that the toy OS kernel manages page tables and the TLB, and prevents users from accessing them (as well as other kernel data structures) directly. This means that our kernel maintains a set of user-level page tables, potentially shared between multiple users; the set might even be a singleton for single-address-space systems. Given this setting, there are multiple ways to achieve separation between user-accessible memory and kernel memory. For instance, the kernel could switch to its own page table and make sure that none of the user-level page tables contain mappings to the physical addresses that store kernel data structures. For our toy OS kernel, we choose a slightly more interesting and popular setting. To avoid switching page tables for entering the kernel, the kernel maintains a so-called kernel window in every virtual address space: a set of virtual addresses, typically at the top end of the virtual address space, that are unavailable to the user, and instead maintain kernel mappings with permissions set so that they are only active in kernel mode of the processor[1]. Linux, for instance, uses this scheme, and in a 32-bit address space, which would span 4GB of memory, e.g. only 3.5GB may actually be addressable in user mode. The top 512KB implement the kernel window. Figure 3.11 shows an example for a virtual address space maintained by our toy OS kernel.

Page Table Management: The toy OS kernel maintains two-level page tables for every user process. As required by the ARM architecture, the first level stores mappings for memory sections and supersections, as well as pointers to the second-level page tables. Since the kernel window is present in the virtual address spaces of all processes, the kernel makes the page table mappings for these addresses constant. Since the mappings are constant and their translation function is statically known, the corresponding page table entries are constant too. That means each user-level page table that the kernel maintains has a number of known entries which, for each user, reside at the same position in the page table encoding.

[1]This is the technique attacked by Meltdown (Lipp et al., 2018). Since hardware manufacturers are promising to fix this major flaw, we present the more interesting setting instead of the less complex and slower scenario with a separate kernel address space.

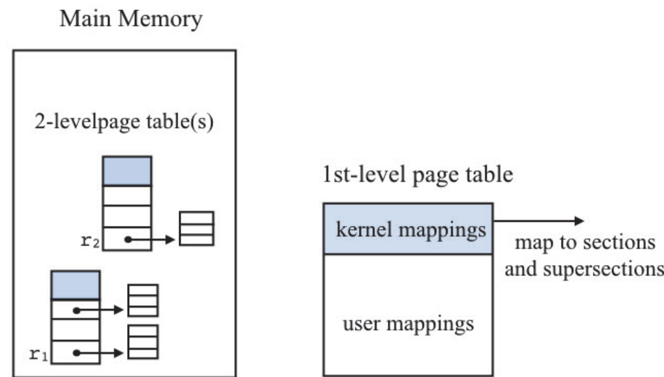

Figure 3.12: OS Kernel Page Table Management

Figure 3.12 describes this layout.

In summary, the OS kernel manages page tables for the user processes and also for itself, e.g. by adding, removing, or changing mappings, by keeping a page table structure per user process, and by maintaining invariants, such as never giving a user access to kernel-private data structures, ensuring that certain mappings are always present, or ensuring non-overlapping mappings between different page tables if so desired.

TLB Management: Since a TLB caches address translation, page table operations by the OS kernel may leave the TLB out of date for the page table in memory, and the OS kernel must flush (invalidate) the TLB before that lack of synchronisation can affect program execution. Since flushing the TLB is expensive, OS kernel designers work hard to delay and minimise flushes and to make them as specific as possible using ASIDs for invalidating only specific sets of entries. The requirement of TLB flushing after a page table operation depends on the page table operation itself: if the operation is to *remap* or *unmap* a page, then the OS kernel must invalidate the respective TLB entries, as a TLB potentially caches any mapped page table entry. Mapping a page, on the other hand, does not require TLB maintenance as the TLB in ARMv7-A does not store translation fault entries. If TLB management is done correctly, the TLB has no effect other than speeding up execution. If it is done incorrectly, machine execution leads to unpredictable behaviour and wrong memory contents are read/written, or unexpected memory access faults might occur.

In addition to the page table operations, an OS kernel has to take special care during a context switch between processes to maintain TLB consistency. In ARM's VMSA, two separate registers hold values for the active ASID and page table root, and these two registers cannot be updated atomically during a context switch. This lack of atomicity is a problem because any speculative memory access by the processor might contaminate the TLB in these two ways:

- the *old* ASID being associated with entries from the *new* page table, or

- the *new* ASID being associated with entries from the *old* page table.

The ARM architecture leaves TLB synchronisation after updating ASID and page table root registers to the OS kernel, and it recommends specific code sequences for the OS kernel to avoid TLB contamination. We provide two of these here that are widely used.

```
Using a reserved ASID to synchronize TLB:
update ASID register to reserved ASID 0
ISB
update page table root register
ISB
update ASID register to new ASID
```

```
Using a page table with only global mappings to synchronize TLB:
update page table root register to the global-only mappings
ISB
update ASID register to new ASID
ISB
update page table root register to new root
```

We prove the correctness of the first sequence in the case study Chapter 8.

In summary, this section has outlined the responsibilities of an OS kernel to manage the virtual memory system of ARMv7-A architecture.

3.4 Summary and Remarks

In this chapter, we have outlined the background information required to understand the virtual memory aspects addressed in this thesis. The chapter includes a brief overview of ARMv7-A VMSA's features such as pages, page tables and most importantly the TLB. We have also described the TLB maintenance requirements for the OS kernel in the VMSA. The main message of this chapter is that a TLB is security-critical in a virtual memory system because a poorly managed TLB will lead to

- compromise in process isolation,
- wrong memory operations, and
- unpredictable behavior.

An alternative to the virtual memory system of ARMv7-A, the ARMv7-R architecture provides another memory protection mechanism, called protected memory system architecture (PMSA). The PMSA scheme uses control registers in a memory protection unit (MPU) instead of page tables to achieve memory protection; it does not have the non-deterministic behaviour introduced by potential TLB

misses. Though PMSA is easier to implement and to manage than VMSA, its memory control is less fine-grained as compared to that of pages in the VMSA. PMSA is not a point of interest in this thesis.

The focus of the thesis is TLB reasoning for the ARMv7-A architecture, but Chapter 9 will give an outlook of how the reasoning technique might apply to other architectures such as x86, RISC-V and MIPS.

CHAPTER

FOUR

A Formal Model of the ARMv7-A MMU

In this chapter, we develop an operational model of the ARMv7-A memory management unit (MMU) including the TLB in Isabelle/HOL. As outlined in the previous chapter, the MMU of ARMv7-A consists of a TLB that caches page table walks from the main memory under ASIDs. While our aim in this research is to model the ARMv7-A TLB with ASIDs, in this chapter we focus on a TLB that caches entries *without* ASIDs. We develop a base MMU model for such a TLB and identify the inherent reasoning complexities even this simple TLB would entail. We then provide a series of refinements for the base MMU model to stepwise abstract away the hardware details and to reach at an abstract MMU model that captures the essential TLB functionality and is easier to reason about.

In the next two chapters, we then build on the MMU model of this chapter to introduce ASIDs and global TLB entries, and eventually a two-stage TLB that is implemented in the more recent implementations of the ARMv7-A architecture. For page table operations, we reuse Kolanski's existing ARMv6 page table model (Kolanski and Klein, 2009), update it to ARMv7-A and integrate it with the TLB formalisation that we build up in this and the next chapters to form the MMU models.

We begin this chapter by summarising our memory layout and Kolanski's page table model. We then develop a generic formal model of an ARMv7-style TLB and instantiate this model for a TLB without ASIDs. Next we present our base MMU model including page table interface for TLB reloading, address translation, memory operations, updating page table root register and TLB maintenance operations. We then identify reasoning complexities for this base MMU model and provide a series of refinements to abstract the hardware details. In the end we join refinement levels to prove the soundness of our abstraction and conclude the chapter.

This chapter is based on the published work (Syeda and Klein, 2017).

4.1 Page Table Abstraction

As described in the previous chapter (Sect. 3.2), the ARMv7-A architecture provides multiple address translation modes that differ in the number of translation levels and bit-length of virtual addresses. Without loss of generality for the treatment of TLBs we focus on one of these modes here — the others are analogous. This mode provides four sizes of pages (small, large, section, and supersection; c.f. (ARM, 2008, Chapter B3)) and a two-level page table structure.

In this section we provide the page table abstraction that we use for MMU operations. For page table operations, we reuse Kolanski's existing ARMv6 page table model (Kolanski and Klein, 2009), update it to ARMv7-A and integrate it with

our TLB formalisation.

Addresses: Kolanski's model differentiates between virtual and physical address by type, and we continue in that tradition. He defines addresses `addr_t` as:

```
datatype ('a, 'p) addr_t = Addr 'a
```

where `'a` is the address size (e.g. `32 word`) and `'p` is a tag which can be `physical` or `virtual`. For modeling the addresses of an ARMv7-style machine, we specialise `addr_t` as:

```
vaddr = (32 word,virtual) addr_t    paddr = (32 word,physical) addr_t
```

We use `addr_val (Addr a) = a` to extract the address.

Main Memory: We model main memory as the partial function

$$\texttt{heap :: paddr} \rightharpoonup \texttt{8 word}$$

to express that it works on physical addresses and that not all physical address might be backed by memory in the machine. If a computation accesses non-existing memory, an exception should be raised.

Page Table Root: In paging, location of the first-level of a page table is determined by its root address, we model the root of the page table with type `paddr`.

Page Directory Entry: In our page table abstraction, an entry of the first level of a page table is referred to as page directory entry and it is of the type `pde`:

```
datatype pde  =  InvalidPDE
              |  ReservedPDE
              |  PageTablePDE paddr
              |  SectionPDE paddr arm_perm_bits
```

A `pde` is either an `InvalidPDE`, a `ReservedPDE`, a `PageTablePDE` with the pointer to the start of second level of the page table, or a `SectionPDE` with the base address and access permission bits of the section it belongs to. `arm_perm_bits` simply encodes the access permission bits of an ARMv7-A translation context.

Without loss of generality for the treatment of TLBs, we omit formalising super-sections and their page directory entries.

Page Table Entry: An entry of the second level of the page table is called a page table entry and formalised as:

```
datatype pte = InvalidPTE
             | SmallPagePTE paddr arm_perm_bits
```

It is either an `InvalidPTE`, or holds the base physical address of a small page along with its permission bits. Again for the treatment of TLBs, we omit formalising large pages and their page table entries, as any results based on the model having small page table entries will be applicable to the model having large page table entries.

Page Table Lookup: We reuse lookup functions from Kolanski's page table model to obtain page directory entries and page table entries for given virtual addresses. We provide these functions here with a brief explanation.

For any virtual address `va`, the function `get_pde` encodes its respective page directory entry from the page table rooted at the physical address `rt` in the memory `mem` (for `mask` and bit shifts operations, please refer to Sect. 2.2.3 in the chapter of notation):

```
get_pde :: heap ⇒ paddr ⇒ vaddr ⇒ pde option
get_pde mem rt va ≡
let pd_idx_offset = (addr_val va >> 20) && mask 12 << 2
in decode_heap_pde mem (rt + pd_idx_offset)
```

Based on the translation flow explained in Sect. 3.2.3, the function `get_pde` first calculates the offset for page table traversal using the page table root `rt` and the virtual address `va`, and then decodes the corresponding machine word into a page directory entry using the function `decode_heap_pde` (definition not shown here).

Similar to function `get_pde`, the function `get_pte` returns a page table entry `pte` given any virtual address `va` and a pointer to the page table `pt_base`.

```
get_pte mem pt_base va ≡
let pt_idx_offset = (addr_val va >> 12) && mask 8 << 2
in decode_heap_pte mem (pt_base + pt_idx_offset)
```

In this thesis, we use functions `get_pde` and `get_pte` to derive TLB entries from page tables present in the main memory. With this we conclude the interface functions of our page table abstraction and now explain our generic ARMv7-style TLB model.

4.2 A Formal TLB Model for the ARMv7-style MMU

In this section we introduce our formal TLB model for ARMv7-style MMU and its constituent operations. As mentioned in the previous chapter (Sect. 3.2.4), the ARM architecture manual (ARM, 2008) describes the TLB as a black box, i.e. by its external interface only. It does not specify the replacement strategy or its exact internal state. We use the same approach and base our abstraction directly on the architecture manual: we specify a TLB model and its lookup operation. Together with Kolanski's ARM page table model, this will then form the basis for specifying the semantics of memory operations and TLB maintenance operations that we eventually want to reason about.

The state of a TLB is straightforward: it is merely a set of TLB entries, where a TLB entry consists of an ASID, a virtual base address, a physical base address, and a set of flags for access control and other page attributes. Figure 4.1 gives a visual representation. Corresponding to the four page sizes of the architecture, there are four different sizes of TLB entries. In this thesis we restrict ourselves to small pages and sections, and therefore we have two types of TLB entries, one with 20-bit base addresses for small pages and one with 12-bits for sections. Formally:

ASID	VBA	PBA	flags
ASID	VBA	PBA	flags
⋮	⋮	⋮	⋮
ASID	VBA	PBA	flags

```
where
VBA =
Virtual Base Address
PBA =
Physical Base Address
ASID =
Address Space Identifier
```

Figure 4.1: An Abstraction of an ARMv7-style TLB

```
type_synonym 'a tlb = 'a tlb_entry set

datatype 'a tlb_entry =
            EntrySmall ('a option) (20 word) (20 word) flags
          | EntrySection ('a option) (12 word) (12 word) flags
```

We have specified the ASID field with a polymorphic type `'a option`, so that we can instantiate our TLB model for different caching layouts. For example in this chapter where we model a simple TLB without ASIDs, we simply instantiate `'a tlb_entry` with the `unit` type. Later in Chapter 5, we instantiate the ASID field with a type `asid` to formalise TLB with ASIDs and global entries, where a TLB entry with `None` ASID field represents a global translation entry and with `Some` ASID a process-specific translation entry.

The field `flags` in the type `'a tlb_entry` is a record type for access permission bits of TLB entries. The ARMv7-A architecture manual specifies the following permission bits that are encoded in the TLB entries during page table walks:

```
record  flags     =
        nG        ::  1 word  -- non-global bit
        perm_APX  ::  1 word  -- access permission bit 2
        perm_AP   ::  2 word  -- access permission bits 1 and 0
        perm_XN   ::  1 word  -- execute-never bit
```

The `nG` bit of a TLB entry represents whether this entry provides address translation globally or for a process under an ASID. For a TLB without ASIDs we ignore the `nG` bit in lookup and reload operations. We utilise the `nG` bit in our TLB models with ASIDs in Chapter 5. The rest of the TLB flags are access permission bits.

TLB Lookup: With the TLB state formalised, we now describe its lookup. For any given 32-bit virtual address, a TLB lookup finds the corresponding TLB entry. A lookup can have three kinds of results:

```
datatype 'e lookup_type = Miss | Incon | Hit 'e
```

These results are: either there is no corresponding entry and the TLB needs to be refilled (`Miss`), or there is more than one matching entry and the TLB is inconsistent (`Incon`), or there is exactly one correct result (`Hit`). We have kept the `lookup_type` polymorphic for a generic TLB model.

For TLB lookup, we specify the virtual address range covered by a TLB entry wrapped up in a type class `entry_op`:

```
class entry_op =
fixes range_of :: 'e ⇒ vaddr set
```

A TLB entry matches a virtual address `va` when `range_of` that TLB entry includes the virtual address `va`. The lookup operation is then defined as:

```
entry_set :: 'e set ⇒ vaddr ⇒ 'e set
entry_set t va ≡ {e ∈ t | va ∈ range_of e}

lookup :: (vaddr ⇒ 'e set) ⇒ vaddr ⇒ 'e lookup_type
lookup eset va ≡
if eset va = ∅ then Miss
else if ∃x. eset va = {x} then Hit (the_elem (eset va)) else Incon
```

where `the_elem {x} = x`. The function `entry_set` simply collects entries of the given TLB matching the virtual address `va`. The `lookup` function expects a matched entry set and performs a cardinality check to determine the resultant lookup value. For notation of set operations, please refer to Sect. 2.2.3 of the notation chapter.

It is worthwhile to discuss our choice of polymorphism for the parameter `range_of`: we define it for any type `'e` instead of an `'a tlb_entry` because eventually we will reason about a two-stage TLB framework having a separate page directory cache (PDC). This choice allows us to conveniently instantiate class `entry_op` for PDC entries in Chapter 6. For the TLB we instantiate the class `entry_op` for all possible instantiations of `'a tlb_entry`, since a virtual address range of a TLB entry is independent of its ASID field:

```
range_of :: 'a tlb_entry ⇒ vaddr set
range_of e ≡
case e of
EntrySmall a vba pba fl ⇒
  Addr ` {base_addr vba..base_addr vba + (2^12 - 1)}
| EntrySection a vba pba fl ⇒
    Addr ` {base_addr vba..base_addr vba + (2^20 - 1)}
```

Where

```
base_addr v ≡ UCAST('a → 32) v << 32 - size v
```

The `range_of` a small TLB entry is the set of all virtual addresses with their most-significant 20-bits equal to the virtual base address of that entry. For a section TLB entry, this range is for the most-significant 12-bits. The function `base_addr` converts the 12-bit or 20-bit base address to the respective 32-bit address using bit shifts operations `<<`. The function `size` returns the bit-length of a word in type `nat`.

Physical Address from a TLB Lookup: Any result `Hit e` for a given `vaddr va` can be translated directly into a `paddr pa` by replacing the most-significant bits of `va` with the 12-bit or 20-bit physical base address stored in `e`. The function `va_to_pa` performs this operation (for details about the notation and functions over the type `word`, please refer to Sect. 2.2.3 in the chapter of notation):

```
va_to_pa va (EntrySmall a vba pba fl) =
Addr ((UCAST(20 → 32) pba << 12) || addr_val va && mask 12)
```

```
va_to_pa va (EntrySection a vba pba fl) =
Addr ((UCAST(12 → 32) pba << 20) || addr_val va && mask 20)
```

With this we conclude our generic TLB formal model for ARMv7-A architecture.

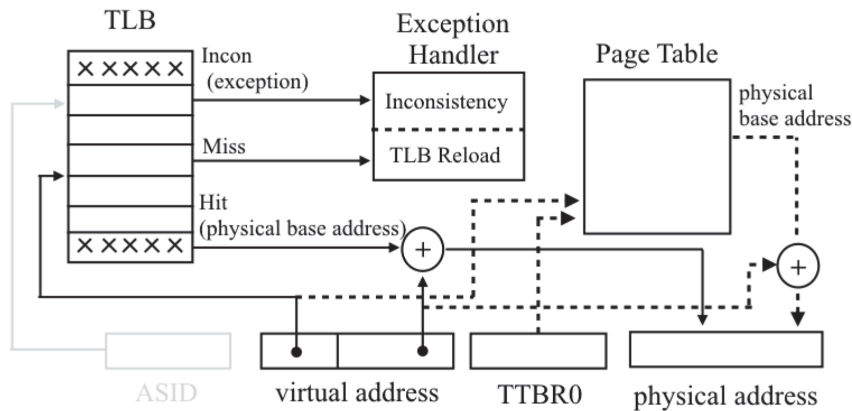

Figure 4.2: ARMv7-style Memory Management Unit

4.3 From TLB to MMU Model

We now present a formal MMU model based on the ARM architecture manual (ARM, 2008) integrated with the instruction set architecture (ISA) semantics by Fox and Myreen (Fox and Myreen, 2010). This MMU model consists of a simple TLB that caches entries from the page table present in main memory without ASIDs.

The purpose of this model is to highlight complexities even a simple TLB would entail for program reasoning. It also introduces key concepts of our modeling and refinement framework for the advanced MMU models of the next chapters. This model also signifies how our reasoning methodology can potentially be scaled to different TLB layouts.

The ARM ISA model is very detailed and extensively validated, but it assumes a flat, total function `MEM :: 32 word` $\Rightarrow$ `8 word` without address translation as its model for memory. We keep `MEM` as the basic model for physical memory, but generalise it to the partial function `MEM :: paddr` $\rightharpoonup$ `8 word` to express that it works on physical addresses and that not all physical address might be backed by memory in the machine. We then change all read and write instructions that access main memory to not access physical memory directly, but to go through the TLB and address translation first. The existing Cambridge ARM model conveniently provides a narrow interface to memory with the functions `mem_write` and `mem_read` that all other memory accesses go through, so we concentrate our work there.

Since our plan for this research is to provide a series of MMU models that differ in the TLB abstraction, making them simpler and easier to reason about as we progress, we design the interface between the rest of the ARM model and the MMU as a type class `mmu` in Isabelle that we can instantiate. Separate instances will give us separate models between which we then can prove refinement theorems.

Figure 4.2 gives an overview of our MMU model. To formalise this picture, we extend the original `state` record of the Cambridge ARM model with an additional hardware register: the page table root register `TTBR0`. As explained in the previous section, for a TLB without ASIDs we instantiate `'a tlb` as:

```
type_synonym TLB = unit tlb
```

Next we use Isabelle's extensible records (Naraschewski and Wenzel, 1998) to extend `state` with the type `TLB` which will contain the TLB hardware state. The TLB lookup operation for a virtual address `va` is simply:

```
abbreviation tlb_lookup t va = lookup (entry_set t) va
```

The main interface for the rest of the ARM model to the MMU is wrapped up in the type class `mmu`:

```
class mmu =
mmu_translate :: vaddr ⇒ 'a state_scheme ⇒ paddr × 'a state_scheme
mmu_read :: vaddr × nat  ⇒ 'a state_scheme ⇒ bl × 'a state_scheme
mmu_write ::
     bl × vaddr × nat ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
update_TTBR0 :: paddr ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
flush :: flush_type ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
```

Where `'a state_scheme` are the potential extensions of the existing record type `state`. The interface for the values being read and written in the ARM model is via `bl` = `bool list` instead of machine words directly, which we keep here, and the `nat` parameter indicates how many bytes to read/write, e.g. one byte, a word, a double word, etc. The OS kernel may update the page table root register during context switches which requires TLB maintenance, therefore we introduce an `update_TTBR0` instruction. The `flush` instruction, as the name describes, is specified to invalidate TLB entries for maintaining coherency.

We now explain the instantiation of each of the parameters of type class `mmu` for our MMU model. These functions also constitute the base model of our refinement chain in Sect. 4.4. We begin by presenting the interface between TLB and page table present in the main memory.

4.3.1 Page Table Walk

As explained in Sect. 3.2.4, in the ARMv7-A architecture the processor walks the page table after a TLB miss and it also reloads the TLB with the respective translation entry. We formalise such a page table walk from Kolanski's page table model (Kolanski and Klein, 2009) using his functions `get_pde` and `get_pte` to find

the respective page table entries and to encode the result in the corresponding `'a tlb_entry` format. Formally:

```
pt_walk :: 'a ⇒ heap ⇒ paddr ⇒ vaddr ⇒ 'a tlb_entry option
pt_walk asid mem root va ≡
case get_pde mem root va of None ⇒ None
| ⌊PageTablePDE p⌋ ⇒
    case get_pte mem p va of None ⇒ None | ⌊InvalidPTE⌋ ⇒ None
    | ⌊SmallPagePTE bpa perms⌋ ⇒ ⌊to_sml_entry bpa perms va asid⌋
| ⌊SectionPDE bpa perms⌋ ⇒ ⌊to_sec_entry bpa perms va asid⌋
| ⌊_⌋ ⇒ None
```

After reading the entry from the page table present in the memory `mem` at the location `root` for the given virtual address `va`, the function `pt_walk` returns an `'a tlb_entry option`: `None` in the case of an invalid page table entry and `Some` TLB entry for a valid page table entry. The functions `to_sml_entry` and `to_sec_entry` convert base physical address and permission bits stored in the page table entries to the TLB entry format. They also encode the base virtual address from the virtual address `va` and and determine the ASID field for the resultant TLB entry. They are defined as (for bit shifts operations, please refer to Sect. 2.2.3):

```
to_sec_entry bpa perms va asid ≡
EntrySection (tag_conv asid (to_flgs perms))
 (UCAST(32 → 12) (addr_val va >> 20))
 (word_extract 31 20 (addr_val bpa)) (to_flgs perms)
```

```
to_sml_entry bpa perms va asid ≡
EntrySmall (tag_conv asid (to_flgs perms))
 (UCAST(32 → 20) (addr_val va >> 12))
 (word_extract 31 12 (addr_val bpa)) (to_flgs perms)
```

We have defined these conversion functions with a generic ASID interface, so that we can instantiate them to different MMU layouts. The function `to_flgs` converts the permission bits of the given page table entry to the respective TLB flags, and the function `word_extract` extracts the specified number of bits from the given n-bits word. The function `tag_conv` determines the ASID for the TLB entry. The ASID field of a TLB entry is determined as: if the `nG` bit of the respective page table entry is global, the ASID field of the TLB entry is `None`, otherwise the given ASID is assigned to the TLB entry.

As the function `pt_walk` is defined for `'a tlb_entry`, we access it for the MMU model of Figure 4.2 as

```
pt_walk () mem root va
```

receiving an entry with type `unit tlb_entry option`. The tag conversion for the `unit` case is automatically instantiated as:

$$\texttt{tag_conv ut perms} \equiv \lfloor\texttt{()}\rfloor$$

where the parameter `ut` is of type `unit`.

4.3.2 Address Translation

The address translation for memory operations is defined as:

```
mmu_translate va = do {
  update_state (λs. s(|TLB := TLB s - tlb_evict s|));
  (mem, ttbr0, tlb) ← read_state (MEM, TTBR0, TLB);
  case tlb_lookup tlb va of
  Miss ⇒
    let entry = pt_walk () mem ttbr0 va
    in if fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|TLB := TLB s ∪ {the entry}|));
             return (va_to_pa va (the entry))
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

The function `mmu_translate` first evicts an underspecified set of entries from the TLB. This models the fact that the architecture does not define the replacement strategy and the programmer must assume that any entry could be evicted at any time.[1] Since the rest of the Cambridge ARM model is deterministic, we use an oracle function `tlb_evict` here instead of true nondeterminism.

The next step in `mmu_translate` after reading out the hardware state is to do a TLB lookup for the virtual address `va` to be translated. If the result of that lookup is `Incon`, the machine raises an unrecoverable exception and halts, expressing the fact that in normal operation, this state should never be encountered.

If the result is `Hit entry`, we translate `entry` to the corresponding physical address `pa` using the function `va_to_pa` and return that address. A full formalisation would at this point additionally check flags and access rights and generate the appropriate exception information where needed.

If the result is `Miss`, we perform a page table walk using the function `pt_walk` starting from the root address `TTBR0`. If the result of the page table walk is a page `fault` i.e. `fault entry` ≡ `(entry = None)`, we raise this fault, which will cause the machine to jump to the appropriate exception handler. If the result of the walk

[1] ARM also provides locked down entries that will not be evicted automatically. These could be modelled easily here by excluding them from the eviction set.

is a particular mapping entry `e`, we perform a TLB reload by adding this entry to the TLB, and execute address translation as in the `Hit` case.

4.3.3 Memory Operations

Reusing the original functions `mem_write` and `mem_read` from the ARM model for physical memory, the instances for memory operation of the MMU model are straightforward:

```
mmu_write (val, va, sz) = do {
  pa ← mmu_translate va;
  when_no_exc mem_write (val, pa, sz)
}

when_no_exc f = do {
  exception ← read_state exception;
  if exception = NoException then f else return ()
}

mmu_read (va, sz) = do { pa ← mmu_translate va; mem_read (pa, sz) }
```

Both, `mmu_write` and `mmu_read`, first perform address translation, and then their original purpose, but using translated addresses instead. In case of an exception in `mmu_translate`, the write function does nothing to give the translation exception precedence, while the pure read function can continue, because it does not change the state.

4.3.4 Updating the Page Table Root Register

In this base model, `update_TTBR0` instruction merely does what its name describes, TLB eviction is not entailed with updating the page table root register:

```
update_TTBR0 r = update_state (λs. s⦇TTBR0 := r⦈)
```

4.3.5 Flush Operations

As described in section Sect. 3.2.4, the ARM architecture provides TLB maintenance operations to the OS kernel for flushing the entire TLB and also for invalidating (evicting) outdated entries either by ASID or by virtual addresses or by virtual addresses globally for all ASIDs. We formalise these maintenance

operations, and since our TLB model does not yet support ASIDs in this chapter we model the required flush operations as:

```
datatype flush_type = FlushTLB | Flushvarange (vaddr set)
```

The instantiation for this base model is:

```
flush f ≡
case f of FlushTLB ⇒ update_state (λs. s(|TLB := ∅|))
| Flushvarange vset ⇒
    update_state (λs. s(|TLB := flush_vset (TLB s) vset|))
```

`FlushTLB` simply makes the TLB set empty, whereas `Flushvarange` flushes the entries matching the given set of virtual addresses, i.e.,

```
flush_vset t vset = t - (⋃v∈vset {e ∈ t | v ∈ range_of e})
```

With this we conclude presenting the base MMU model. By redirecting all other memory-related functions in the ARM model to go through the interface of class `mmu` and by introducing instructions for TLB maintenance operations, we arrive at a full operational model that supports address translation and TLB caching without ASIDs. The purpose of this research is not to provide a fully detailed formalisation that is validated to comprehensively conform with existing hardware, but to present the main ideas on how to simplify reasoning in the presence of a TLB of ARMv7-A architecture. Despite this focus, we have validated the model by executing a number of instructions in Isabelle/HOL, manually checking consistency with the expected behaviour. A full formalisation would need a more extensive test suite in the spirit of Fox and Myreen (Fox and Myreen, 2010).

In summary, we have so far extended the Cambridge ARM model by: a change of memory model to admit the notion of unmapped memory, the introduction of an MMU including the TLB and page table lookup mechanisms, the extension with maintenance operations, and an adjustment of the subsequent memory operations to include the address translation layer.

4.4 MMU Abstraction

The MMU model of Sect. 4.3 gives us the ground truth of how hardware operates, and thereby the foundation for a logic for programs under the TLB without ASIDs, but even this simple model is hard to reason about directly. From Sect. 4.3, we identify that a TLB introduces:

- nondeterminism through unspecified entry replacement strategy,

- potential state change caused by any mapped memory access, including reads,
- potential (internally) inconsistent TLB state from multiple conflicting entries, and
- potential (external) inconsistency between page table and TLB.

The latter two are states the program must avoid. The first two introduce unnecessary complexity: a program that is otherwise deterministic should not require reasoning about nondeterminism, and a correctly operated TLB framework should not complicate reasoning about memory reads nor memory writes that are unrelated to page tables.

In this section, we show how we can construct a model that avoids the additional complexity and produces sufficient conditions for safe execution. In particular, we build a series of formal abstractions of the concrete MMU model of Sect. 4.3 that are increasingly easier to reason about, but preserve functionality and the optimisation opportunities OS developers must be able to exploit. We verify these step-wise abstractions by refinement theorems.

Our refinement stack is shown in Figure 4.3 and it consists of three levels. In the first refinement, we abstract our base MMU model to remove the eviction of TLB entries, thus eliminating the nondeterminism for any logic built for this abstract model. Next we abstract the deterministic TLB to cache the mapped state of the active page table completely, giving us an MMU model with a saturated TLB and hence eliminating state change on memory reads. Finally we abstract the TLB to an extent that no actual TLB lookup is required: TLB inconsistencies are tracked using a set of virtual addresses, while MMU operations are performed using the page tables present in main memory. For each level up in the refinement stack we prove that its abstraction preserves a refinement relation and is sound with respect to its immediate concrete MMU model. We then join refinement levels in order to show the soundness of the most abstract model with respect to the base model of Sect. 4.3. We argue that the most abstract model is sound for program verification by the usual data refinement idea, that if we manage to prove that a program execute safely with the most abstract model, it will execute safely with the concrete TLB model.

The main burden on the proof engineer that we cannot hope to eliminate completely in general will be to show that the TLB is currently in a consistent state for the address to be accessed. We formalise consistency for a virtual address as:

```
consistent mem root tlb va =
(tlb_lookup tlb va = Hit (the (pt_walk () mem root va)) ∧
 no_fault (pt_walk () mem root va) ∨
 tlb_lookup tlb va = Miss)
```

This condition combines internal consistency of the TLB (no `Incon` results permitted), with external consistency, i.e. synchronicity with the current state of the

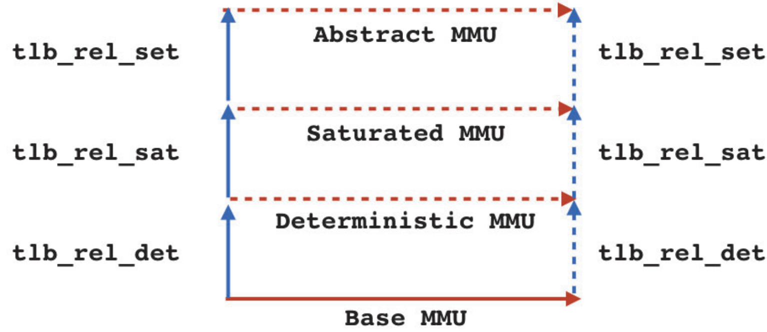

Figure 4.3: Refinement Stack for MMU Models

mapped page table for this particular address. The condition for `Hit` ensures that a TLB does not store translation entries for unmapped virtual addresses (`no_fault`). We will see in the most abstract model that, while not eliminated, the condition can be greatly simplified.

4.4.1 Determinism

With this in mind, we observe as the first step in our abstraction chain that a TLB with fewer entries is always more consistent, and in this sense safer, than one with more entries. Formally, lookup results naturally form an order with `Miss` being the bottom element, and `Incon` the top:

$$\texttt{l} \leq \texttt{l'} \equiv \texttt{l = Miss} \vee \texttt{l' = l} \vee \texttt{l' = Incon}$$

We can prove monotonicity

Lemma 1. $\texttt{t} \subseteq \texttt{t'} \Longrightarrow \texttt{tlb_lookup t v} \leq \texttt{tlb_lookup t' v}$

Proof. By case distinction and unfolding the definitions. □

We can use this in the abstraction chain by making the abstraction less safe, i.e. more inconsistent, with the standard refinement idea that if we manage to prove safe behaviour of the abstraction, we will also have proved safe behaviour of all possible actual executions.

This means, we can use our observation above by noting that, instead of a TLB that nondeterministically evicts entries, we can use a TLB that *never* evicts entries, unless explicitly instructed. If we can prove a program safe with this larger TLB, it will also be safe with the smaller TLB. We can prove this fact by instantiating `mmu_translate` for a deterministic version in which the TLB does not evict entries and then proving refinement. We name this instantiation `mmu_translate_det` and define it as:

```
mmu_translate_det va ≡ do {
  (mem, ttbr0, tlb) ← read_state (MEM, TTBR0, TLB);
  case tlb_lookup tlb va of
  Miss ⇒
    let entry = pt_walk () mem ttbr0 va
    in if fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s(|TLB := TLB s ∪ {the entry}|));
              return (va_to_pa va (the entry))
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

For `mmu_translate_det`, the only difference to `mmu_translate` from Sect. 4.3.2 is the missing `tlb_evict` line. We now present the refinement theorem between `mmu_translate` and `mmu_translate_det`.

Theorem 1. *Assuming that two states* `s` *and* `t` *have the refinement relationship*

```
tlb_rel_det s t ≡ truncate s = truncate t ∧ TLB s ⊆ TLB t
```

where the notation `truncate s` *means all fields of the extensible* `state` *record without the* `TLB` *extension. That is, the states* `s` *and* `t` *differ only in the contents of the TLB, and the TLB of* `s` *contains fewer entries. If the TLB of* `t` *is consistent w.r.t. lookups in* `va`*, then the address translation of a virtual address* `va` *performed using* `mmu_translate` *in* `s` *is the same as the one performed by* `mmu_translate_det` *in* `t`*. Moreover, the resultant final states retain the relationship* `tlb_rel_det` *and the TLBs remain consistent w.r.t.* `va`*. Figure 4.4 depicts this theorem as a diagram. Formally:*

```
mmu_translate va s = (pa, s')     mmu_translate_det va t = (pa', t')
                consistent t va      tlb_rel_det s t
--------------------------------------------------------------------
           pa' = pa ∧ consistent t' va ∧ tlb_rel_det s' t'
```

Proof. We observe that the abstract TLB in state `t` is consistent for `va`, that is, a lookup for `va` will either produce `Miss` or `Hit`. Given the subset relationship and Lemma 1, we get that either both TLBs produce the same `Hit e`, or both walk the page table (with the same result, since the states only differ in TLB content), or that `t` produces a `Hit`, but `s` walks the page table. Since `t` is consistent for `va`, the result of the walk has to agree with the `Hit`. □

The definitions of the memory write and read operations remain unchanged compared to the base model, but they now pick up the new `mmu_translate_det` instance of the `mmu` class. The refinement between `mmu_write` and `mmu_write_det` is presented below.

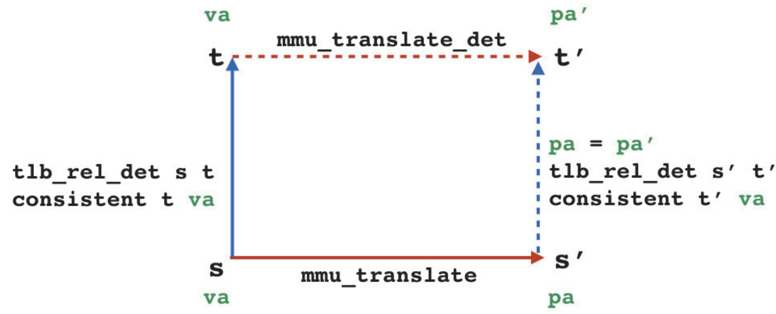

Figure 4.4: Refinement between Nondeterministic and Deterministic Translation

Theorem 2. *Assuming that two states* `s` *and* `t` *have the refinement relationship* `tlb_rel_det` *as defined for Theorem 1, if the TLB of* `t` *is consistent w.r.t. lookups in* `va`*, then a memory write of value* `val` *to the virtual address* `va` *using* `mmu_write` *is same as one performed by* `mmu_write_det` *in* `t`*. Moreover, the resultant final states retain the relationship* `tlb_rel_det`*. Formally:*

```
      mmu_write (val, va, sz) s = ((), s')
    mmu_write_det (val, va, sz) t = ((), t')
     consistent t va      tlb_rel_det s t
    ------------------------------------------
              tlb_rel_det s' t'
```

Proof. Since Theorem 1 says that `mmu_translate_det` and `mmu_translate` return the same results, memory write is trivially equal to the base model. □

It is important to note here that consistency of the TLB cannot be preserved in Theorem 2, since memory writes can change the page table. This means, re-establishing consistency on writes will be an obligation on the proof engineer, not an automatic invariant that is provided by the model. This mirrors the reasoning OS developers do mentally. Consistency is established either by reasoning that the write was not to a page table, by using appropriate invalidation instructions, or by reasoning that if a page table was changed, the change was unrelated to the address that is about to be accessed.

Refinement between deterministic and nondeterministic memory read functions is straightforward:

Theorem 3. *Memory reads preserve* `tlb_rel_det` *and TLB consistency.*

```
         mmu_read (va, sz) s = (val, s')
       mmu_read_det (va, sz) t = (val', t')
        consistent t va      tlb_rel_det s t
  --------------------------------------------------
  val = val' ∧ consistent t' va ∧ tlb_rel_det s' t'
```

Proof. Since Theorem 1 says that `mmu_translate_det` and `mmu_translate` return the same results, memory read is trivially equal to the base model. □

The instructions `update_TTBR0_det` and `flush_det` have identical instantiations to their nondeterministic version, since TLB is not evicted in these instructions. We have proved their trivial refinement for completeness and omit presenting these theorems here.

This MMU model removes nondeterminism from the base model and is sound for executions in which the larger TLB never triggers an inconsistency. We will now present our second abstract model based on this deterministic MMU.

4.4.2 Invariance

As the next step, we eliminate TLB state change for memory reads. We note that the presence of an inconsistent entry in the TLB is not dangerous yet, only *using* the inconsistent entry is. This means, for every memory transaction and MMU operation we can add to the TLB *all* the mapped entries that the current page table produces. As we have seen in the previous step, this is sound because we add more entries that are consistent with the page table, and inconsistency with older entries is not dangerous yet. This gives us a TLB that is always saturated with entries for the mapped virtual addresses. On read operations, the state will not change, because the set of page table results before and after reading is the same. On write operations outside the page table we have the same — only on writes to the page table we will get a state change in the TLB, which is what we should expect. We saturate the TLB with the mapped page table entries after updating the page table root register and flush operations as well. This way, we fully capture the mapped state of the active page table after all MMU operations.

Formally, we instantiate `mmu_translate` of type class `mmu` in this model such that the TLB always remains saturated i.e. whenever it accesses memory it reloads the mapped page table to the TLB. We name the operation `mmu_translate_sat`:

```
mmu_translate_sat va = do {
  tlb_refill;
  tlb ← read_state TLB;
  case tlb_lookup tlb va of Miss ⇒ raise PAGE_FAULT
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}

tlb_refill = do {
  (mem, ttbr0, tlb) ← read_state (MEM, TTBR0, TLB);
  let mapped_ptable = ran (pt_walk () mem ttbr0);
  update_state (λs. s(|TLB := tlb ∪ mapped_ptable|))
}
```

The call to `tlb_refill` at the beginning of `mmu_translate_sat` achieves the satu-

ration mentioned above by adding the `ran` of the `pt_walk` function to the TLB, i.e. `ran m = {b | ∃a. m a = ⌊b⌋}`. In this saturated TLB `mmu_translate_sat` then performs a standard lookup. `Incon` results still lead to the same exception as before. `Miss` results represents a page fault, and `Hit` results are the same as in the previous models.

The saturation of the TLB with the mapped page table implies that we have reduced external and internal inconsistency conditions to one condition: our saturated TLBs always encode the full state of the mapped page table.

We now show refinement between `mmu_translate_det` of the last section with this `mmu_translate_sat`.

Theorem 4. *With the refinement relation*
`tlb_rel_sat s t ≡ truncate s = truncate t ∧ TLB s ⊆ TLB t ∧ saturated t`
where
`saturated t ≡ ran (pt_walk () (MEM t) (TTBR0 t)) ⊆ TLB t`

We get data refinement between the deterministic `mmu_translate_det` *and* `mmu_translate_sat`*:*

```
       mmu_translate_det va s = (pa, s’)
       mmu_translate_sat va t = (pa’, t’)
      consistent t va     tlb_rel_sat s t
---------------------------------------------------
pa’ = pa ∧ consistent t’ va ∧ tlb_rel_sat s’ t’
```

This means, we still get the same address translation results, and preserve consistency, as well as the refinement relation, including saturation.

Proof. Essentially the same argument as before, observing that entries stemming from `pt_walk` cannot make a `va`-consistent entry inconsistent. □

For this MMU model, we also change the memory operations to preserve saturation. The new instantiations for saturated TLBs are `mmu_write_sat` and `mmu_read_sat`:

```
mmu_write_sat (val, va, sz) = do {
  pa ← mmu_translate_sat va;
  when_no_exc do { mem_write (val, pa, sz); tlb_refill }
}
```

```
mmu_read_sat (va, sz) = do {
  pa ← mmu_translate_sat va;
  mem_read (pa, sz)
```

```
}
```

In `mmu_write_sat`, the TLB is refilled after the write operation to maintain saturation as this write could have been to a page table present in the memory. Similarly in `mmu_read_sat`, this saturation is being achieved implicitly through `mmu_translate_sat` function in the start, as reading from the memory does not affect the state of page table.

Theorem 5. *Memory writes preserve the TLB refinement relation* `tlb_rel_sat`*, including saturation.*

```
mmu_write_det (val, va, sz) s = ((), s')
mmu_write_sat (val, va, sz) t = ((), t')
consistent t va     tlb_rel_sat s t
-------------------------------------------
tlb_rel_sat s' t'
```

Proof. Follows directly from the refinement result on `mmu_translate_sat`. □

Theorem 6. *Memory reads preserve the TLB refinement relation* `tlb_rel_sat`*, including TLBs consistency and saturation.*

```
mmu_read_det (va, sz) s = (val, s')
mmu_read_sat (va, sz) t = (val', t')
consistent t va     tlb_rel_sat s t
-------------------------------------------------
val = val' ∧ consistent t' va ∧ tlb_rel_sat s' t'
```

Proof. Follows directly from the refinement result on `mmu_translate_sat`. □

Similarly, updating the page table root and flush functions now need to include a global TLB refill after their operations to preserve saturation. We saturate the TLB after the flush operations so that the TLB always capture the mapped state of the page table. Their instantiations are presented below:

```
update_TTBR0_sat r =
do { update_state (λs. s(|TTBR0 := r|)); tlb_refill }


flush_sat f = do {
  tlb ← read_state TLB;
  (case f of FlushTLB ⇒ update_state (λs. s(|TLB := ∅|))
   | Flushvarange vset ⇒
       update_state (λs. s(|TLB := flush_vset tlb vset|)));
  tlb_refill
}
```

These instantiations preserve `tlb_rel_sat` refinement relation.

Theorem 7. *Updating the page table root register preserves the refinement relation between saturated and deterministic states.*

```
            update_TTBR0_det r s = ((), s')
update_TTBR0_sat r t = ((), t')     tlb_rel_sat s t
───────────────────────────────────────────────────
                  tlb_rel_sat s' t'
```

Proof. Follows directly from the definitions of `update_TTBR0_det` and `update_TTBR0_sat`. □

Theorem 8. *TLB flush operations preserve the refinement relation between saturated and deterministic states.*

```
          flush_det f s = ((), s')
flush_sat f t = ((), t')     tlb_rel_sat s t
────────────────────────────────────────────
             tlb_rel_sat s' t'
```

Proof. Follows directly from the definitions of `flush_det` and `flush_sat`. □

With this we complete the presentation of the second abstract MMU model in our refinement chain. Before concluding this section, we comment on the reductions and reasoning simplifications for memory operations based on this saturated MMU model.

Simplification Lemmas: For memory *reads*, as planned, the TLB state remains unchanged in this saturated model, eliminating one of the major difficulties in reasoning about the TLB.

Lemma 2. *In saturated states, memory reads do not change the TLB.*

```
mmu_read_sat (va, sz) s = (val, t)     saturated s
──────────────────────────────────────────────────
                 TLB t = TLB s
```

Proof. By observing that memory reads do not change the state and that a saturated TLB already contains all current mapped page table entries. □

A simple optimisation to this model would be to not update the TLB for *every* memory write, but only for writes to the current page table structure or the page table root register. This immediately produces a reduction result: if the current page table structure is not writeable, and if the execution mode is unprivileged, i.e. the page table root register cannot be changed, then we know that no memory transaction will change the saturated TLB state, and we can therefore reason about a much simpler model without TLB and with fixed address translation. This is what user-level execution expects: users should not need to worry about the presence or absence of a TLB. The following theorem encapsulates the conditions for this reduction.

Lemma 3. *Memory writes that do not change the page table content leave the saturated TLB constant, preserving consistency and saturation.*

```
                   mmu_write_sat (val, va, sz) s = ((), s')
∀va. pt_walk () (MEM s) (TTBR0 s) va = pt_walk () (MEM s') (TTBR0 s') va
                  consistent s v      saturated s
─────────────────────────────────────────────────────────────────────────
              TLB s' = TLB s ∧ consistent s' v ∧ saturated s'
```

Proof. The condition that all `pt_walk` outcomes remain the same after the memory write directly implies that the `range (pt_walk () mem ttbr0)` term in `tlb_refill` remains the same, and since the TLB is already saturated, the TLB refill has no effect. □

In summary, reasoning about the TLB has become much more tractable in this model. Inconsistency is reduced to internal inconsistency only, and nondeterminism as well as unnecessary state change are removed. For a program logic on top of this model it would suffice to guarantee the absence of inconsistencies, and to treat page faults the same way a program logic for standard address translation would, e.g. as in Kolanski's work (Kolanski, 2011).

4.4.3 Essence

This leads us to the last refinement step, where we abstract the saturated TLB to the extent that no actual TLB `lookup` is required: the functionality of the TLB of Figure 4.2 can be captured completely by only keeping record of those virtual addresses that are inconsistent in the TLB with the current page table. It is then enough to perform address translation using the page table only. Figure 4.5 presents an overview of the ARMv7-A MMU with our abstract TLB.

For this last abstraction, we extend the record type `state` not with `tlb`, but with `incon_set` of type `vaddr set` and instantiate `mmu_translate` of type class `mmu` to:

```
mmu_translate_set va ≡
do {
  (mem, ttbr0, incon_set) ← read_state (MEM, TTBR0, incon_set);
  if va ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk () mem ttbr0 va
       in if fault entry then raise PAGE_FAULT
          else return (va_to_pa va (the entry))
}
```

Note that this address translation contains no state change at all any more, apart from potentially raising exceptions.

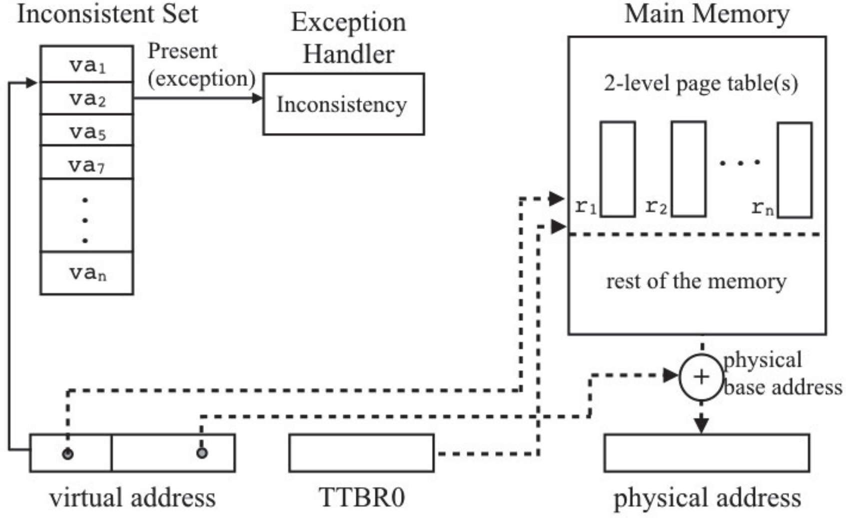

Figure 4.5: ARMv7-style Memory Management Unit with Abstract TLB

With this definition, we get the following theorem.

Theorem 9. *Let* `tlb_rel_set` *denote the refinement relation*

```
tlb_rel_set s t ≡
truncate s = truncate t ∧ incon_addrs s ⊆ incon_set t ∧ saturated s
```

where

```
incon_addrs s ≡
{va | tlb_lookup (TLB s) va = Incon} ∪
{va | ∃x. tlb_lookup (TLB s) va = Hit x ∧
          fault (pt_walk () (MEM s) (TTBR0 s) va)}
```

constructs the set of TLB-inconsistent addresses and false hits in the saturated TLB. False hits are those unmapped virtual addresses with resepct to the current page table that produces a Hit *instead of a* Miss *in the TLB lookup. The functions* `mmu_translate_sat` *and* `mmu_translate_set` *preserve this relation and yield the same result. Formally:*

$$\frac{\texttt{mmu_translate_sat va s = (pa, s')} \quad \texttt{mmu_translate_set va t = (pa', t')} \quad \texttt{va} \notin \texttt{incon_set t} \quad \texttt{tlb_rel_set s t}}{\texttt{pa = pa'} \wedge \texttt{va} \notin \texttt{incon_set t'} \wedge \texttt{tlb_rel_set s' t'}}$$

Proof. According to the refinement relation, the `incon_set` tracks the inconsistent entries in the saturated TLB. We are, therefore, in the `else` branch of the function `mmu_translate_set` and in the `Hit` case of `mmu_translate_sat`. The results must agree, because `saturated` says that the `Hit` results represent precisely the walks we perform in `mmu_translate_set`. □

As in the previous step, the memory access instantiations have to change. For `mmu_write_set`, we must figure out which new addresses might have become inconsistent. We do this by comparing the page tables before and after the physical write operation. For `mmu_read_set`, the definition is similar to the base-level model, we only use the new `mmu_translate_set` instance.

```
mmu_write_set (val, va, sz) = do {
  (mem, ttbr0, incon_vaddrs) ← read_state (MEM, TTBR0, incon_set);
  pa ← mmu_translate_set va;
  when_no_exc do {
    mem_write (val, pa, sz);
    mem' ← read_state MEM;
    let pt_comp =
      ptable_comp (pt_walk () mem ttbr0) (pt_walk () mem' ttbr0);
    update_state (λs. s(|incon_set := incon_vaddrs ∪ pt_comp|))
  }
}
```

```
ptable_comp wlk wlk' ≡
{va | no_fault (wlk va) ∧ no_fault (wlk' va) ∧ wlk va ≠ wlk' va ∨
      no_fault (wlk va) ∧ fault (wlk' va)}
```

```
mmu_read_set (va, sz) = do {
  pa ← mmu_translate_set va;
  mem_read (pa, sz)
}
```

To figure out the new TLB-inconsistencies as the result of a memory write, we compare the results of page table walks before and after the write operation using the `ptable_comp` function. Two scenarios might add inconsistent entries: changing an existing mapping (first disjunct of `ptable_comp`), or removing an existing mapping (second disjunct). Note that a single heap write can affect multiple mappings at once, for instance when it changes the pointer to an entire page table level. It is the effect of this comparison that OS engineers reason about informally when they compute which addresses need to be flushed from the TLB.

For `mmu_translate_set` and `mmu_read_set` it is now obvious in this model that the entire state remains constant if there is no translation exception, and, with Theorem 9 also that memory reads return the correct result. Moreover, `mmu_write_set` also preserves `tlb_rel_set` with the saturated `mmu_write_sat` and we provide its refinement below.

Theorem 10. *Memory writes for TLB-consistent addresses preserve the refinement relation* `tlb_rel_set`.

$$\frac{\texttt{mmu_write_sat (val, va, sz) s = ((), s')} \quad \texttt{mmu_write_set (val, va, sz) t = ((), t')} \quad \texttt{va} \notin \texttt{incon_set t} \quad \texttt{tlb_rel_set s t}}{\texttt{tlb_rel_set s' t'}}$$

Proof. First, we observe that, with Theorem 9, the TLB lookup produces the same result on each abstraction level, and therefore the two physical write operations produce the same memory state. Second, we need to establish that `incon_set` correctly tracks which entries in the saturated TLB have become inconsistent. These are the mapped entries with those addresses for which `pt_walk` now yields a different result, which is precisely what `ptable_comp` computes. □

Refinement between `mmu_read_set` and `mmu_read_sat` is presented below.

Theorem 11. *Memory reads for TLB-consistent addresses preserve the refinement relation* `tlb_rel_set`.

$$\frac{\texttt{mmu_read_sat (va, sz) s = (val, s')} \quad \texttt{mmu_read_set (va, sz) t = (val', t')} \quad \texttt{va} \notin \texttt{incon_set t} \quad \texttt{tlb_rel_set s t}}{\texttt{val = val'} \wedge \texttt{va} \notin \texttt{incon_set t'} \wedge \texttt{tlb_rel_set s' t'}}$$

Proof. By noting that the state of TLB and the incon set remain constant in `mmu_read_sat` and `mmu_read_set`, and their memory read results are equal. □

For this abstract MMU model that keeps track of inconsistent virtual addresses with the current page table, updating the page table root register has an effect on the incon set similar to changing the state of the current page table with a memory write operation. In `update_TTBR0_set`, we use the `ptable_comp` function with walks from two page tables to compute the resultant inconsistent virtual addresses.

```
update_TTBR0_set r = do {
  (mem, ttbr0, incon_set) ← read_state (MEM, TTBR0, incon_set);
  let ptable_comp =
    ptable_comp (pt_walk () mem ttbr0) (pt_walk () mem r);
  update_state
   (λs. s⦇TTBR0 := r⦈⦇incon_set := incon_set ∪ ptable_comp⦈)
}
```

For flush operations, we simply remove the given virtual addresses from the incon set.

```
flush_set f ≡
```

```
case f of FlushTLB ⇒ update_state (λs. s(|incon_set := ∅|))
| Flushvarange vset ⇒ do {
      incon_set ← read_state incon_set;
      update_state (λs. s(|incon_set := incon_set - vset|))
    }
```

`update_TTBR0_set` and `flush_set` preserve the refinement relation with their respective saturated instantiations.

Theorem 12. *Updating the page table root register preserves the refinement relation* `tlb_rel_set`*.*

$$\frac{\begin{array}{c}\texttt{update_TTBR0_sat r s = ((), s')}\\ \texttt{update_TTBR0_set r t = ((), t')} \qquad \texttt{tlb_rel_set s t}\end{array}}{\texttt{tlb_rel_set s' t'}}$$

Proof. By establishing that `incon_set` correctly tracks the inconsistent entries using `ptable_comp` of the respective saturated TLB after updating the page table root. □

Theorem 13. *Flush instruction preserve the refinement relation* `tlb_rel_set`*.*

$$\frac{\begin{array}{c}\texttt{flush_sat f s = ((), s')}\\ \texttt{flush_set f t = ((), t')} \qquad \texttt{tlb_rel_set s t}\end{array}}{\texttt{tlb_rel_set s' t'}}$$

Proof. By unfolding definitions and basic set reasoning. □

We now conclude our most abstract MMU model and its refinement. In this model the TLB modeling has been reduced to merely a consistency check for virtual addresses (using `incon_set`), there is no actual TLB lookup, and after every MMU operation the resultant inconsistencies are detected and loaded into the `incon_set`. A program reasoning framework taking this MMU model as its memory model would avoid the complexities of TLB reasoning that are identified in the beginning of this section. In our program logic and case study for program verification (Chapter 7 and Chapter 8), we show that for unprivileged user-level code we can reduce to a model without TLB and with fixed address translation. For privileged OS-level code, address translation is usually fixed for the OS code itself and all locations it accesses. In this case, the TLB will always return these fixed mappings, and cannot become inconsistent since they never change. That means, if we prove that each OS memory access remains within a safe set of addresses and that the page table mappings for this set never change, execution is safe and does not need to reason about the TLB. For seL4 for instance, this property is already proved as part of its reasoning about page tables without the TLB.

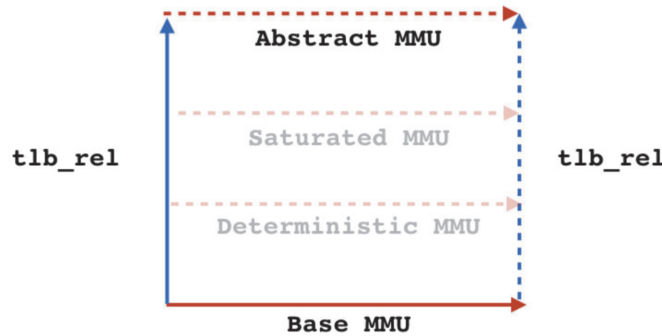

Figure 4.6: Refinement between Nondeterministic and Abstract MMU

4.4.4 Joining the Refinement Levels

In this section, we join the refinement levels of Figure 4.3 to show that our most abstract model is sound with respect to the base model. The resultant refinement is between the abstract and the base model through the deterministic and saturated MMU models as shown in Figure 4.6. The refinement relation `tlb_rel` is:

```
tlb_rel r t ≡
∃s s'. tlb_rel_det r s ∧ tlb_rel_sat s s' ∧ tlb_rel_set s' t
```

Where the state `r` has the nondeterministic TLB, the state `s` has the deterministic TLB, the state `s'` has the saturated TLB and the state `t` has incon set. The functions `tlb_rel_det`, `tlb_rel_sat` and `tlb_rel_set` are the refinement relations provided in Sect. 4.4.1, Sect. 4.4.2 and Sect. 4.4.3 respectively.

Since we have explained refinement of every MMU operation for each level of Figure 4.3 in the previous sections, we now group MMU operations and supply their refinement collectively. We use the same approach while explaining refinement of MMU models with ASIDs and two-stage TLBs in Chapter 5 and Chapter 6. We group MMU operations as:

```
datatype mem_op_typ  =  translate (vaddr)
                     |  read (vaddr × nat)
                     |  write (bool list × vaddr × nat)


datatype mmu_op_typ  =  root_update (paddr)
                     |  tlb_flush (flush_type)
```

We have grouped memory operations separately from MMU maintenance because memory operations require TLB consistency for the given virtual address for successful evaluation and subsequent refinement, and MMU operation do not require TLB consistency. We encode the results of memory operations as:

```
datatype mem_res_typ  =  PA (paddr)
                      |  BL (bool list)
                      |  UT (unit)
```

The mem_res_typ represents the result of memory operations: a physical address paddr for address translation, a machine word in the form of bool list for memory read and unit for memory write. The result value of MMU operations is simply unit. The evaluation functions for memory and MMU operations in a state extended with either of the TLB interfaces is then:

```
mem_op :: mem_op_typ ⇒ 'a state_scheme ⇒ res_typ × 'a state_scheme
mem_op (translate va) s =
(PA (fst (mmu_translate va s)), snd (mmu_translate va s))
mem_op (read (va, sz)) s =
(BL (fst (mmu_read (va, sz) s)), snd (mmu_read (va, sz) s))
mem_op (write (bl, va, sz)) s =
(UT (fst (mmu_write (bl, va, sz) s)), snd (mmu_write (bl, va, sz) s))
```

```
mmu_op :: mmu_op_typ ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
mmu_op (root_update rt) s = update_TTBR0 rt s
mmu_op (tlb_flush f) s = flush f s
```

Note that these evaluation functions are polymorphic with 'a state_scheme, and this polymorphism enables us to have a generic interface for different MMU models: we simply access these functions with a nondeterministic TLB state and name these accesses as mem_op_nondet and mmu_op_nondet. Similarly, mem_op_set and mmu_op_set represent the evaluation for the abstract TLB model having the incon_set.

We then have two refinement theorems between the nondeterministic and the absstract MMU models as present below and also shown in Figure 4.7 and Figure 4.8.

Theorem 14. *Refinement between nondeterministic and abstract memory operations.*

$$\frac{\texttt{mem_op_nondet f r = (res, r')} \quad \texttt{mem_op_set f t = (res', t')} \quad \texttt{consistent_set f t} \quad \texttt{tlb_rel r t}}{\texttt{res = res'} \wedge \texttt{tlb_rel r' t'}}$$

where mem_op_nondet *and* mem_op_set *simply evaluate the memory operation* f *of type* mem_op_typ *for the given states giving the respective results* res *and* res' *and reaching the evaluated states. The* consistent_set *ensures that the abstract state* t *is TLB-consistent: the given virtual address is not an element of* incon_set *of state* t.

Proof. By case analysis on the function f and using the respective refinement theorems of Sect. 4.4.1, Sect. 4.4.2 and Sect. 4.4.3 and observing that for each

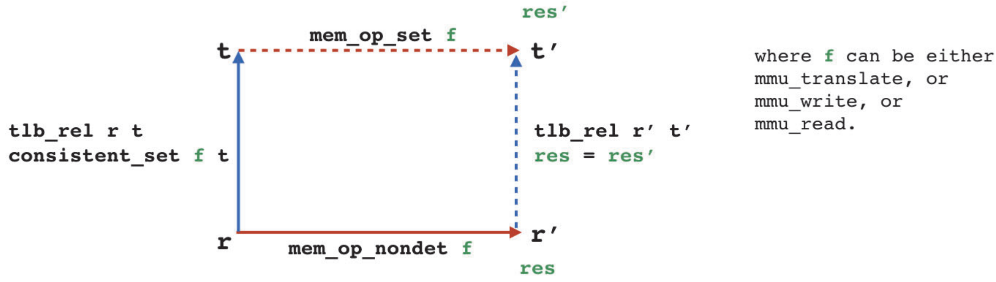

Figure 4.7: Refinement between Nondeterministic and Abstract Memory Operations

level, the address consistency condition on this level implies address consistency on the level below. □

Theorem 15. *Refinement between nondeterministic and abstract MMU maintenance operations.*

$$\frac{\texttt{mmu_op_nondet f r = ((), r')} \quad \texttt{mmu_op_set f t = ((), t')} \quad \texttt{tlb_rel r t}}{\texttt{tlb_rel r' t'}}$$

where `mmu_op_nondet` *and* `mmu_op_set` *simply evaluates the MMU operation* `f` *of type* `mmu_op_typ` *for the given states reaching the evaluated states.*

Proof. By case analysis on the function `f` and using the respective refinement theorems of Sect. 4.4.1, Sect. 4.4.2 and Sect. 4.4.3. □

With this we conclude presenting the refinement stack.

4.5 Summary and Remarks

In this chapter, we have presented our formal page table interface and a generic TLB model for the ARMv7-A architecture. We have then instantiated our TLB model to a simple TLB caching translation entries without ASIDs and have provided an MMU model with memory and maintenance operations integrated with the Cambridge ARM ISA model. We have then identified the reasoning complexities entailed by the TLB and have presented our refinement framework in detail that avoids these additional complexities and abstracts away unnecessary hardware details. We have concluded this chapter by presenting the soundness of our abstraction with respect to the base model.

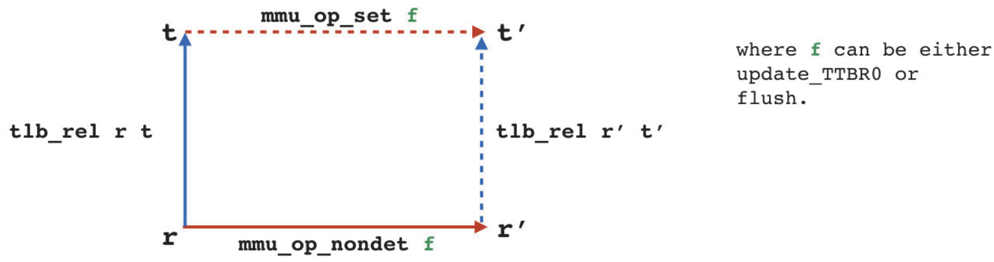

Figure 4.8: Refinement between Nondeterministic and Abstract MMU Operations

The main message of the refinement chain available online (Syeda, 2019) and presented in this chapter is that a program logic on top of this model only has to keep track of and check for inconsistent TLB entries, and that TLB entries can only be made inconsistent with changes to the page table and TTBR0. TLB invalidation can be selective and can be deferred until we can no longer prove from other sources that we only access consistent mappings. In essence, the refinement chain in this chapter hides the low-level hardware TLB reasoning and provides a much simpler interface to the proof engineer.

In Chapter 5 and Chapter 6, we build on the MMU model of this chapter to formalise advanced MMU features such as ASIDs, global TLB entries and two-stage TLB.

CHAPTER

FIVE

A Formal Model of the ARMv7-A MMU with ASIDs

In the previous chapter, we have presented an operational model of the ARMv7-A memory management unit (MMU) without ASIDs. In this chapter, we build on that model to formalise the MMU with the TLB caching page table entries under ASIDs and global tags. We again build a refinement stack to abstract away the hardware details and we also explain how our refinement framework handles the added features. In Chapter 6, we extend the MMU model of this chapter with a separate page directory cache (PDC) to develop a two-stage TLB model for more recent implementations of the ARMv7-A architecture.

This chapter is organised as: we instantiate the generic TLB model presented in the previous chapter (Sect. 4.2) to the TLB caching entries under ASIDs and global tags. We then provide the base MMU model including address translation, memory operations, TLB maintenance operations as well register update instructions affecting the state of the TLB. We then provide a series of refinements to abstract away the hardware details of this base MMU model. For each refinement level, we first explain the model and then provide the refinement theorems collectively as we presented in Sect. 4.4.4. In the end we join the refinement levels and conclude the chapter.

This chapter is based on the published work (Syeda and Klein, 2017) and the submitted work (Syeda and Klein, 2019).

5.1 ARMv7-A MMU Model with ASIDs

We now present a formal MMU model with ASIDs for the ARMv7-A architecture (ARM, 2008) and integrate it with the instruction set architecture (ISA) semantics by Fox and Myreen (Fox and Myreen, 2010). This MMU model includes a TLB that caches entries under ASIDs from the page table present in the main memory. As mentioned in the virtual memory chapter (Sect. 3.2.4), the ARMv7-A architecture associates a process-specific tag called address space identifier (ASID) with translation entries of the TLB. The ASIDs enable the TLB to cache translation entries from different processes. The architecture provides 8-bit ASIDs, which means the TLB can cache entries for up to 256 processes; c.f. (ARM, 2008, Chapter B3). The architecture also supports global page table entries: a page table entry marked as global is cached in the TLB providing address translation for all processes. On a TLB miss, the processor does the page table walk, checks the global bit of the page table entry to determine the `nG` bit (non-global bit) for the respective TLB entry. We model ASIDs and global attributes for our formal ARMv7-A MMU specification, and also develop a refinement stack to abstract the hardware details. This model is then used in the next chapter to formalise the MMU with the two-stage TLB as implemented in Cortex-A15.

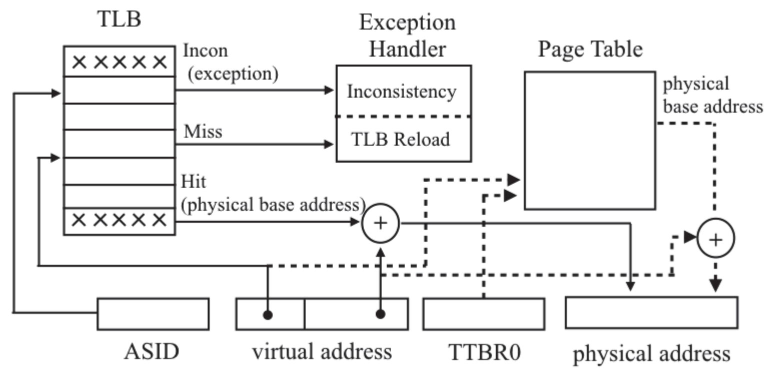

Figure 5.1: ARMv7-style Memory Management Unit with ASIDs

Figure 5.1 gives an overview of the ARMv7-A MMU with ASIDs. This MMU is similar to the MMU without ASIDs (Figure 4.2), but now the TLB stores entries with ASIDs and global tags. To formalise the simpler MMU of Figure 4.2, we had extended the original `state` record type of the Cambridge ARM model with the page table root register `TTBR0`. For the TLB with ASIDs and global entries, we now additionally introduce the type `asid` as:

type_synonym `asid = 8 word`

and instantiate `'a` for the type `'a tlb`, introduced in Sect. 4.2, as:

type_synonym `TLB = asid tlb`

An ASID-specific TLB entry has an ASID field `Some asid`, while `None` represents a global TLB entry. The `range_of` a TLB entry remains independent of its ASID field, and we use the same instantiation provided as in Sect. 4.2. However, TLB lookup now has to change to a lookup under an ASID. We model the TLB lookup under an ASID by finding the matched entry set of the TLB for that ASID:

```
asid_entry_set :: asid tlb_entry set ⇒ vaddr ⇒ asid tlb_entry set
asid_entry_set t a va ≡
{e ∈ entry_set t va | asid_of e = None ∨ asid_of e = ⌊a⌋}
```

abbreviation `tlb_lookup t a va = lookup (asid_entry_set t a) va`

Where `lookup` is the same function as presented in Sect. 4.2 on Page 44. Given a virtual address `va` and an ASID `a`, the function `asid_entry_set` is a filter for the matched `entry_set`: an entry matching the virtual address `va` has to be either global or under the same ASID `a` in order to provide translation for the virtual address `va`. The function `asid_of` simply returns the ASID field of a TLB entry. We can also classify a given TLB within its `global_entries` and `non_global_entries`:

```
global_entries :: asid tlb_entry set ⇒ asid tlb_entry set
global_entries t = {e ∈ t | asid_of e = None}
```

```
non_global_entries :: asid tlb_entry set ⇒ asid tlb_entry set
non_global_entries t = {e ∈ t | ∃a. asid_of e = ⌊a⌋}
```

Next we use Isabelle's extensible records (Naraschewski and Wenzel, 1998) to extend the record type `state` with the type `asid × TLB` which will contain the active ASID register and the TLB hardware state. This MMU model inherits all operations of the base model without ASIDs (presented in Sect. 4.3). These operations include address translation, memory read and write, updating the page table root register and TLB maintenance instructions. The state of the TLB caching entries under ASIDs is also affected when the OS kernel changes the ASID register, therefore for this MMU we add instruction for updating the ASID register. We also formalise the TLB flush operations under ASIDs.

This means, we extend the interface of type class `mmu`, presented in Sect. 4.3, as:

```
class mmu_extended = mmu +
fixes update_ASID :: 8 word ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
fixes flush_with_ASID ::
     asid_flush_type ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
```

Where `'a state_scheme` are the potential extensions of the existing record type `state`. The type class `mmu_extended` inherits all the parameters of its parent class `mmu`. We will explain `asid_flush_type` in the MMU operations of the base model in Sect. 5.1.3.

For presenting the MMU model and then later the refinement stack, we group the parameters of the type class `mmu_extended` similar to as before:

```
datatype mem_op_typ  =  translate (vaddr)
                     |  read (vaddr × nat)
                     |  write (bool list × vaddr × nat)
```

```
datatype mmu_op_typ  =  root_update (paddr)
                     |  asid_update (asid)
                     |  flush_addr (flush_type)
                     |  flush_asid (asid_flush_type)
```

Note that the type `mmu_op_typ` now contains additional ASID operations.

We evaluate the operations of the type class `mmu_extended` in an `'a state_scheme` reaching the output state as:

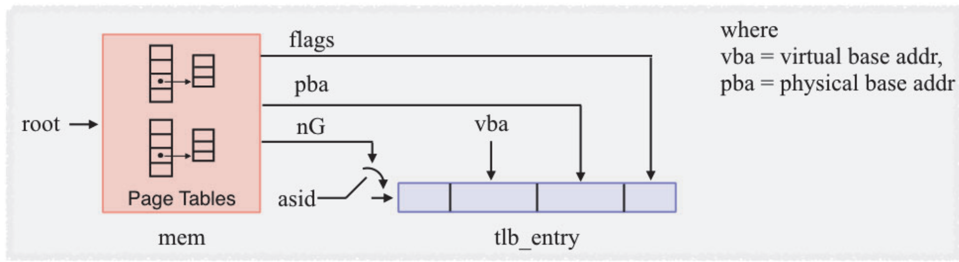

Figure 5.2: Visual Representation of Page Table Walk Function

```
mem_op :: mem_op_typ ⇒ 'a state_scheme ⇒ res_typ × 'a state_scheme
mmu_op :: mmu_op_typ ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
```

Note that these evaluation functions are polymorphic with `'a state_scheme`, and this polymorphism will enable us to have a generic interface for different MMU models later in our refinement stack. For an example of a similar evaluation function, please refer to the refinement section of the previous chapter (Sect. 4.4.4).

We now explain the instantiation of each of the parameters of the type class `mmu_extended` for our MMU model, these functions will also constitute the base model of our refinement chain in Sect. 5.2.

5.1.1 Page Table Walk

We have explained our generic page table interface for encoding the TLB entries in the previous chapter (Sect. 4.1). For the MMU model with ASIDs, we access this interface for a virtual address `va` under an ASID tag `asid` to retrieve the TLB entry from the page table located at the root address `root` in the memory `mem` as:

```
pt_walk asid mem root va
```

This access provides us with a TLB entry of the type `asid tlb_entry option`.

The Figure 5.2 gives an overview of our page table interface. The function `pt_walk` uses a function called `tag_conv` to determine the ASID field of the resultant TLB entry:

```
tag_conv asid perms ≡ if nG perms = 0 then None else ⌊asid⌋
```

This function checks the `nG` (non-global) bit of the `arm_perm_bits` of a page table entry to determine the value of the ASID field.

5.1.2 Memory Operations

We now explain memory operations including address translation, memory read and write for our MMU model. In the end of this section, we wrap them in the interface type `mem_op_typ`.

Address Translation: The address translation for memory operations is:

```
mmu_translate va = do {
  update_state (λs. s(|TLB := TLB s - tlb_evict s|));
  (mem, ttbr0, asid, tlb) ← read_state (MEM, TTBR0, ASID, TLB);
  case tlb_lookup tlb asid va of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s(|TLB := TLB s ∪ {the entry}|));
              return (va_to_pa va (the entry))
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

The strength of our modeling is that the above function is structurally similar to the address translation function of the base model of Chapter 4, it now takes the current ASID into account for TLB lookup and reloading. As before, the first step in the function `mmu_translate` is to evict an underspecified set of entries from the TLB.

The next step in the `mmu_translate` function after reading out the hardware state is to do a TLB lookup for the virtual address `va` to be translated under the active `ASID`. If the result of that lookup is `Incon`, the machine raises an unrecoverable exception and halts, expressing the fact that in normal operation, this state should never be encountered.

If the result is `Hit entry`, we translate `e` to the corresponding physical address `pa` using the function `va_to_pa` and return that address.

If the result is `Miss`, we perform a page table walk using the function `pt_walk` starting from the root address `TTBR0` under the active `ASID`. If the result of the page table walk is a page `fault`, we raise this fault. If the result of the walk is a particular mapping entry `entry`, we perform a TLB reload by adding this entry to the TLB, and execute address translation as in the `Hit` case.

Memory Write and Read: Again, the memory operations are structurally similar to that of the base model of Chapter 4, they are now carried out under the current ASID. We reuse the original functions `mem_write` and `mem_read` from the

ARM model for physical memory to define the instances for memory operations:

```
mmu_write (val, va, sz) = do {
  pa ← mmu_translate va;
  when_no_exc mem_write (val, pa, sz)
}
```

```
mmu_read (va, sz) = do {
  pa ← mmu_translate va;
  mem_read (pa, sz)
}
```

Similar to the base model of Chapter 4, `mmu_write` and `mmu_read` first perform address translation, and then their original purpose, but using translated addresses instead. In case of an exception in `mmu_translate`, the write function does nothing to give the translation exception precedence, while the pure read function can continue, because it does not change the state.

Evaluation: We instantiate the function `mem_op` for the state having the `asid` and the nondeterministic TLB, and call this instantiation `mem_op_nondet`. The generic interface of the function `mem_op` and the type class `mmu_extended` then picks up the definitions for `translate`, `read` and `write` presented above (Page 72).

5.1.3 MMU Operations

We now explain MMU operations including updating the page table root and ASID registers and flush operations. In the end of this section, we wrap them in the type `mmu_op_typ` for their evaluation.

Updating the Page Table Root Register: In this base model, the instruction `update_TTBR0` merely does what its name describes. TLB eviction is not entailed by updating the page table root register:

```
update_TTBR0 r = update_state (λs. s⦇TTBR0 := r⦈)
```

Updating the ASID Register: The `update_ASID` instruction merely does what its name describes. Again, TLB eviction is not entailed by updating the ASID register:

```
update_ASID a = update_state (λs. s⦇ASID := a⦈)
```

Flush Operations: As described in the chapter of virtual memory (Sect. 3.2.4), the ARM architecture provides TLB maintenance operations to the OS kernel for

flushing the entire TLB and also for invalidating (evicting) outdated entries either by ASID or by virtual addresses or by virtual addresses globally for all the ASIDs. We formalise these TLB maintenance operations as:

```
datatype flush_type  =  FlushTLB  |  Flushvarange (vaddr set)
```

```
datatype asid_flush_type  =  FlushASID (asid)
                          |  FlushASIDvarange (asid) (vaddr set)
```

The instantiations of flush operations for this base model are:

```
flush f ≡
case f of FlushTLB ⇒ update_state (λs. s⦇TLB := ∅⦈)
| Flushvarange vset ⇒
    update_state (λs. s⦇TLB := flush_vset (TLB s) vset⦈)
```

```
flush_with_ASID f ≡
case f of
FlushASID a ⇒ update_state (λs. s⦇TLB := flush_asid (TLB s) a⦈)
| FlushASIDvarange a vset ⇒
    update_state (λs. s⦇TLB := flush_asid_vset (TLB s) a vset⦈)
```

`FlushTLB` simply makes the TLB set empty, whereas `Flushvarange` flushes the entries matching the given set of virtual addresses, i.e.,

$$\texttt{flush_vset t vset = t - } (\bigcup_{\texttt{v}\in\texttt{vset}} \{\texttt{e} \in \texttt{t} \mid \texttt{v} \in \texttt{range_of e}\})$$

`FlushASID` flushes all the entries under the given ASID:

$$\texttt{flush_asid t a = t - } \{\texttt{e} \in \texttt{t} \mid \texttt{asid_of e = } \lfloor \texttt{a} \rfloor\}$$

And, `FlushASIDvarange` flushes the entries for the given set of virtual addresses under the given ASID:

$$\texttt{flush_asid_vset t a vset =}$$
$$\texttt{t - } (\bigcup_{\texttt{v}\in\texttt{vset}} \{\texttt{e} \in \texttt{t} \mid \texttt{v} \in \texttt{range_of e} \wedge \texttt{asid_of e = } \lfloor \texttt{a} \rfloor\})$$

Evaluation: We instantiate the function `mmu_op` for the state having the `asid` and the nondeterministic TLB, and call this instantiation `mmu_op_nondet`. The generic interface of the function `mmu_op` and the type class `mmu_extended` then picks up the above presented definitions.

With this we conclude presenting the base MMU model.

5.2 MMU Abstraction

As in Chapter 4, the base MMU model presented above gives us the ground truth of how the hardware operates, and thereby the foundation for a logic for programs under the TLB with ASIDs and global translation entries, but as in Chapter 4 this model is hard to reason about directly. It inherits the reasoning complexities of a simple MMU model caching TLBs without ASIDs, and it involves reasoning about ASIDs and global tags.

In this section, we show how we construct a model that avoids the additional complexity and produces sufficient conditions for TLB-safe execution. Similar to the refinement stack of Sect. 4.4, we build a series of formal abstractions of the base MMU model that are increasingly easier to reason about, but preserve functionality and the optimisation opportunities OS developers must be able to exploit. We verify these step-wise abstractions by refinement theorems.

Our refinement stack is shown in Figure 5.3 and it consists of three levels. The first two levels are similar to the refinement stack of the previous chapter: first we remove the TLB eviction and then saturate the TLB with the mapped state of the page table, but this time, under the active ASID. This gives us an MMU model with the saturated TLB and hence eliminating the TLB-state change on memory reads. For the last level, the abstraction technique is similar to the abstract model of the previous chapter, but the models are not the same. This abstraction still keeps track of the inconsistent virtual addresses under the active ASID, but to soundly model the effect of the `update_ASID` instruction without requiring unnecessary flushes, this new model keeps track of a conservative estimate of what the TLB might remember from the time an ASID was last active. This model also caters for global inconsistencies while updating the page table either by memory write or page table root update. The strength of our abstract MMU model is its simplicity: we still abstract away the TLB lookup and keep the record of inconsistent virtual addresses, and the actual address translation is carried out by the page table in the memory.

For each level up in the refinement stack of Figure 5.3 we prove that its abstraction preserves a refinement relation and is sound with respect to its immediate concrete MMU model. We then join refinement levels in order to show the soundness of the abstract model with the base model of Sect. 4.3.

As in Chapter 4, the main burden on the proof engineer that we cannot hope to eliminate completely in general will be to show that the TLB is currently in a consistent state for the address to be accessed under the active ASID. We formalise consistency for a virtual address in the base and saturated models as: (note that the definition now mentions the current ASID).

```
consistent mem root asid tlb va =
(tlb_lookup tlb asid va = Hit (the (pt_walk asid mem root va)) ∧
```

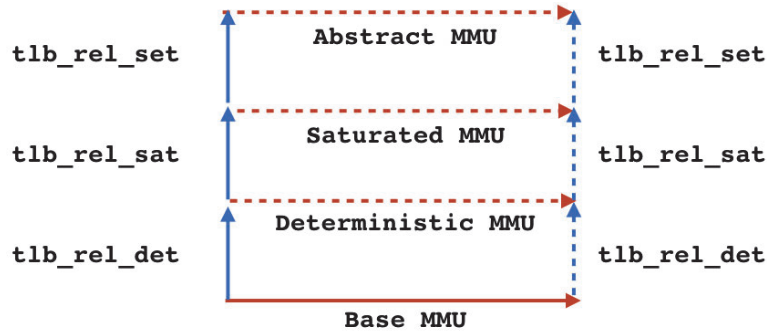

Figure 5.3: Refinement Stack for MMU Models

```
no_fault (pt_walk asid mem root va) ∨
tlb_lookup tlb asid va = Miss)
```

We now provide our MMU models and their refinement.

5.2.1 The Deterministic MMU Model

In this abstraction, we remove non-determinism from the MMU model of Sect. 5.1 by eliminating the `tlb_evict` function. We then prove refinement of each operation of the type class `mmu_extended` with the base MMU model. In addition to the parameters of the type class `mmu`, the type class `mmu_extended` includes the instructions for updating the ASID register and for flushing the TLB under ASIDs.

Memory Operations: The deterministic address translation `mmu_translate_det` is simply:

```
mmu_translate_det va ≡ do {
  (mem, ttbr0, asid, tlb) ← read_state (MEM, TTBR0, ASID, TLB);
  case tlb_lookup tlb asid va of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|TLB := TLB s ∪ {the entry}|));
             return (va_to_pa va (the entry))
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

For the function `mmu_translate_det`, the only difference to the nondeterministic `mmu_translate` is the missing `tlb_evict` line in the start.

The definitions of the memory write and read operations remain unchanged compared to the base model, but they now pick up the new `mmu_translate_det` instance of the `mmu_extended` class. Similar to the evaluation of the base model functions, we instantiate the function `mem_op` for the state having the deterministic TLB, and call this instantiation `mem_op_det`.

MMU Operations: MMU operations including the page table root and ASID update and flush instructions remain the same as the base model, since they do not involve TLB eviction. We wrap up their evaluation for the state with the deterministic TLB in the function `mmu_op_det`.

Refinement: We now present the refinement theorems between the nondeterministic and the deterministic MMU model. The refinement relation is:

```
tlb_rel_det s t ≡
truncate s = truncate t ∧ ASID s = ASID t ∧ TLB s ⊆ TLB t
```

Where the notation `truncate s` means all fields of the extensible `state` record without the `ASID` and `TLB` extension. The relation states the states `s` and `t` differ only in the contents of the TLB, and the TLB of `s` contains fewer entries than the TLB of `t`.

Theorem 16. *The nondeterministic and deterministic memory operations preserve the refinement relation given the consistency of the deterministic TLB for the virtual address.*

$$\frac{\texttt{mem_op_nondet f s = (res, s')} \quad \texttt{mem_op_det f t = (res', t')} \quad \texttt{consistent_det f t} \quad \texttt{tlb_rel_det s t}}{\texttt{res' = res} \wedge \texttt{tlb_rel_det s' t'}}$$

Where `consistent_det` *ensures that memory operation is for a consistent virtual address with respect to the deterministic TLB.*

Proof. Essentially by using operational definitions and the refinement relation. □

Theorem 17. *The nondeterministic and deterministic MMU operations preserve the refinement relation.*

$$\frac{\texttt{mmu_op_nondet f s = ((), s')} \quad \texttt{mmu_op_det f t = ((), t')} \quad \texttt{tlb_rel_det s t}}{\texttt{tlb_rel_det s' t'}}$$

Proof. Trivially true through the operational definitions and the refinement relation. □

With this we conclude our deterministic MMU model and its refinement with the nondeterministic MMU model.

5.2.2 The Saturated MMU

We now present the second model of our refinement chain, where we saturate the TLB with the mapped state of the page table under the active ASID after every MMU operation. This abstraction eliminates the TLB state change for memory reads and for memory writes outside of the page table.

Memory Operations: Formally, we instantiate the parameter `mmu_translate` of the type class `mmu_extended` in this model such that the TLB always remains saturated under the active ASID i.e. whenever it accesses memory it reloads the mapped page table to the TLB. We name the operation `mmu_translate_sat`:

```
mmu_translate_sat va = do {
  tlb_refill;
  (asid, tlb) ← read_state (ASID, TLB);
  case tlb_lookup tlb asid va of Miss ⇒ raise PAGE_FAULT
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}

tlb_refill = do {
  (mem, ttbr0, asid, tlb) ← read_state (MEM, TTBR0, ASID, TLB);
  let mapped_ptable = ran (pt_walk asid mem ttbr0);
  update_state (λs. s(|TLB := tlb ∪ mapped_ptable|))
}
```

The call to `tlb_refill` at the beginning of `mmu_translate_sat` achieves the saturation mentioned above by adding the `ran` of the `pt_walk` function to the TLB under the active ASID, i.e. `ran m = {b | ∃a. m a = ⌊b⌋}`. In this saturated TLB `mmu_translate_sat` then performs a standard lookup under the active ASID. `Incon` results still leads to the same exception as before. `Miss` results represent a page fault, and `Hit` results are the same as in the previous models.

For this abstraction level, we also change the memory operations to preserve saturation as in Chapter 4. The new instantiations for the saturated TLBs are `mmu_write_sat` and `mmu_read_sat`:

```
mmu_write_sat (val, va, sz) = do {
  pa ← mmu_translate_sat va;
  when_no_exc do { mem_write (val, pa, sz); tlb_refill }
}


mmu_read_sat (va, sz) = do {
  pa ← mmu_translate_sat va;
  mem_read (pa, sz)
```

```
}
```

As in Chapter 4, in `mmu_write_sat` the TLB is refilled after the write operation to maintain saturation as this write could have been to a page table present in the memory. In `mmu_read_sat` this saturation is being achieved implicitly through the function `mmu_translate_sat` in the start, as reading from the memory does not affect the state of page table.

We wrap up the evaluation of functions `mmu_translate_sat`, `mmu_write_sat` and `mmu_read_sat` in the function `mem_op`, and call it `mem_op_sat`.

MMU Operations: Similar to the memory operations, we saturate the TLB after updating the page table root and ASID register with the mapped state of the page table under the active ASID. We also saturate the TLB after the flush operations. We do that so that the TLB always captures the mapped state of the page table. The instantiations of MMU operations for this model are presented below.

```
update_TTBR0_sat r =
do { update_state (λs. s⦇TTBR0 := r⦈); tlb_refill }
```

```
update_ASID_sat a =
do { update_state (λs. s⦇ASID := a⦈); tlb_refill }
```

The instantiations for the flush operations are:

```
flush_sat f = do {
  (case f of FlushTLB ⇒ update_state (λs. s⦇TLB := ∅⦈)
   | Flushvarange vset ⇒
       update_state (λs. s⦇TLB := flush_vset (TLB s) vset⦈));
  tlb_refill
}
```

```
flush_with_ASID_sat f = do {
  (case f of
   FlushASID a ⇒ update_state (λs. s⦇TLB := flush_asid (TLB s) a⦈)
   | FlushASIDvarange a vset ⇒
       update_state (λs. s⦇TLB := flush_asid_vset (TLB s) a vset⦈));
  tlb_refill
}
```

We wrap up the evaluation of the functions `update_TTBR0_sat`, `update_ASID_sat`, `flush_sat` and `flush_with_ASID_sat` in the function `mem_op_sat`.

Refinement: We now present the refinement between the deterministic and saturated MMU models. The refinement relation is:

```
tlb_rel_sat s t ≡
truncate s = truncate t ∧
ASID s = ASID t ∧ TLB s ⊆ TLB t ∧ saturated t
```

Where
`saturated t ≡ ran (pt_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ TLB t`

The relation states the states `s` and `t` differ only in the contents of their TLBs, the `TLB s` has fewer entries than the `TLB t` and the abstract state `t` is TLB-saturated with the mapped state of the page table under the active ASID.

Theorem 18. *The deterministic and saturated memory operations preserve the refinement relation given the consistency of the saturated TLB for the virtual address.*

$$\frac{\texttt{mem_op_det f s = (res, s')} \quad \texttt{mem_op_sat f t = (res', t')} \quad \texttt{consistent_sat f t} \quad \texttt{tlb_rel_sat s t}}{\texttt{res' = res} \wedge \texttt{tlb_rel_sat s' t'}}$$

Where `consistent_sat` *ensures that memory operation is for a consistent virtual address with respect to the saturated TLB.*

Proof. We observe that the deterministic TLB of state `s` is consistent for the virtual address `va` given the TLB-subset relationship and the consistency of the saturated TLB of state `t`. The lookup for `va` in both states `t` and `s` will produce either a `Miss` or a `Hit`. When the saturated-TLB of state `t` produces a `Miss` (implies a page table fault), the deterministic TLB of state `s` also has a `Miss` and the memory operation completes the translation for the address `va` through page table walk to eventually encounter a page table fault. In case of a `Hit` with an entry in the saturated-TLB of state `t`, the TLB of `s` either agrees on the same entry with a `Hit` or performs a page table walk. □

Theorem 19. *The deterministic and saturated MMU operations preserve the refinement relation.*

$$\frac{\texttt{mmu_op_det f s = ((), s')} \quad \texttt{mmu_op_sat f t = ((), t')} \quad \texttt{tlb_rel_sat s t}}{\texttt{tlb_rel_sat s' t'}}$$

Proof. By using operational definitions and basic set reasoning. □

With this we conclude our saturated MMU model and its refinement with the deterministic MMU.

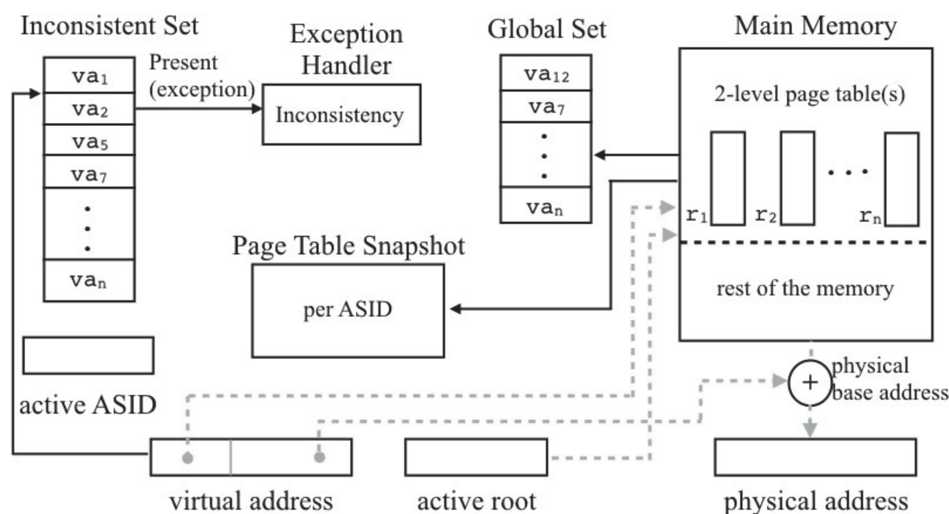

Figure 5.4: ARMv7-style Memory Management Unit with Abstract MMU

5.2.3 The Most Abstract MMU Model

We now present the last and the most abstract model of our refinement chain for the TLB with ASIDs. This model abstracts the TLB lookup completely and the functionality of the TLB is soundly modeled using a set of TLB-inconsistent virtual addresses. In this MMU, we do not extend the state with `asid tlb_entry set` to model the TLB, rather the TLB is modeled using these three components:

```
incon_set :: vaddr set
global_set :: vaddr set
snapshot :: asid ⇒ vaddr set × (vaddr ⇒ asid tlb_entry option)
```

Before presenting the details of this abstraction, we would like to briefly revisit the relation of the TLB with the page tables: *the TLB caches entries from multiple page tables present in the main memory under different ASIDs.* Now that we aim to completely abstract the TLB and to capture its caching functionality using inconsistent virtual addresses with the page table(s), we would have to remember a conservative estimate of what the TLB might remember from the time an ASID was last active. Essentially this is, for each ASID, a snapshot of the current page table state when that ASID was last active modulo all addresses that were inconsistent at that time. This estimate then enables us to keep track of the inconsistencies for different ASIDs and also to detect new inconsistencies while switching between the ASIDs.

Figure 5.4 gives an overview of the ARMv7-A MMU with our most abstract TLB interface. This model might appear complicated, as it has three components and the `snapshot` contains an actual page table state encoded in the TLB entry format. However, this model indeed provides a simpler TLB interface for reasoning.

For better understanding, we summarise the read/write dependencies of the most abstract model for memory and MMU operations in Table 5.1 before explaining the model itself.

Operation	Utilise from Abs. MMU	Update to Abs. MMU
Address Translation	`incon_set`	Nothing
Memory Read	`incon_set`	Nothing
Memory Write	`incon_set`	`incon_set`, `global_set`
Root Update	Nothing	`incon_set`, `global_set`
ASID Update	`incon_set`, `global_set`, `snapshot`	`incon_set`, `snapshot`
Flush Operations	Nothing	`incon_set`, `global_set`, `snapshot`

Table 5.1: Read/Write Dependencies for Memory and MMU Operations

The address translation and memory operations which constitute the majority of program instructions use only the `incon_set` as their TLB interface. The `snapshot` is used only to detect inconsistent addresses while switching the ASID register. The ASID register is only updated on a context switch between processes, and we will see later in this section that `snapshot` is used there for the page table comparison: there is no actual TLB lookup involved.

We now explain our abstract TLB model. The `incon_set` keeps track of the inconsistent virtual addresses for the current state. This includes the inconsistent virtual addresses for the active ASID as well as the globally mapped inconsistent virtual addresses. The address translation and memory operations are carried out under the active ASID, hence they utilise only the `incon_set`. The MMU model reloads the `incon_set` with the resultant inconsistent addresses each time the ASID register is updated *or* the current mapped page table is changed. The current mapped page table can be changed either through a memory write or by updating the page table root register. The TLB flush operations for virtual addresses and the active ASID simply make this set smaller.

The `global_set` consists of all globally mapped virtual addresses irrespective of their consistency. The contents of the `global_set` do not have to strictly equal the globally mapped virtual addresses of the current state, i.e., we keep reloading `global_set` after every write to the page table and after updating the page table root register. The purpose of the `global_set` is to then detect the global inconsistencies while switching ASIDs. Its contents are evicted using TLB flush operations.

The TLB-`snapshot` of the page table state modulo the inconsistent virtual addresses for every ASID is modeled as a pair type of `vaddr set` and the map type `vaddr` $\Rightarrow$ `asid tlb_entry option`. The `vaddr set` of a snapshot for an ASID `a`

contains the inconsistent virtual addresses under the ASID `a`, and the page table state holds the mapped non-global entries. A `Some` TLB entry in the page table state represents a mapped non-global virtual address, and the `None` encodes both global and unmapped virtual addresses. We explain in this section how our model uses `snapshot` to express the effects of updating the ASID register. As expected, the flush operations under ASIDs are able to alter the content of the `snapshot`.

We now explain the operations of our most abstract MMU model for ASIDs and also provide the refinement theorems between the saturated and the most abstract MMU model.

Memory Operations: For translating a virtual address, we simply check its TLB-consistency using the `incon_set` and subsequently carry out the translation from the page table itself.

```
mmu_translate_set va ≡ do {
  (mem, ttbr0, asid, incon_set) ←
    read_state (MEM, TTBR0, ASID, incon_set);
  if va ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if fault entry then raise PAGE_FAULT
         else return (va_to_pa va (the entry))
}
```

Note that we have used only `incon_set` for checking the consistency of the virtual address `va` as it is being resolved under the active ASID.

For `mmu_write_set`, we must figure out which new addresses might have become inconsistent. We do this by comparing the page tables before and after the physical write operation.

```
mmu_write_set (va, va, sz) = do {
  (mem, ttbr0, asid, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  paddr ← mmu_translate va;
  when_no_exc do {
    mem_write (va, paddr, sz);
    mem' ← read_state MEM;
    let incon_set_new =
      ptable_comp (pt_walk asid mem ttbr0) (pt_walk asid mem' ttbr0);
    let global_set_new = global_varange asid mem' ttbr0;
    update_incon_set (incon_set ∪ incon_set_new);
    update_global_set (global_set ∪ global_set_new)
  }
}
```

Where

```
ptable_comp wlk wlk' ≡
{va | no_fault (wlk va) ∧ no_fault (wlk' va) ∧ wlk va ≠ wlk' va ∨
      no_fault (wlk va) ∧ fault (wlk' va)}
```

```
global_varange a m rt ≡
⋃e∈global_entries (ran (pt_walk a m rt)) range_of e
```

The first step in the memory write is to resolve the given virtual address. On a successful translation we simply write to the physical memory. To figure out the potential TLB-inconsistencies as the result of this memory write, we compare the results of page table walks before and after the write operation using the function `ptable_comp`. Two scenarios might add inconsistent entries: changing an existing mapping (first disjunct of `ptable_comp`), or removing an existing mapping (second disjunct). Note that the `ptable_comp` function automatically detects the change of non-global to global entries and vice versa, because the result of `pt_walk` changes when an entry becomes global. This enables us to solely rely on the `incon_set` for memory operations while determining the consistency of global addresses. The function `update_incon_set` in the function `mmu_write_set` simply updates the `incon_set` of the state with the given argument.

We also reload the `global_set` with the address range of new global mappings after the memory write. This helps us to soundly model the inconsistencies while switching ASID as we will see later in this section. The function `global_varange` simply computes the `range_of` the `global_entries` (the TLB entries with `None` ASID tag) from the page table starting at the location `rt` in the memory `m`. The function `update_global_set` in the function `mmu_write_set` updates the `global_set` of the state with the given argument.

For the read operation `mmu_read_set`, the definition is similar to the base MMU model, we only use the new `mmu_translate_set` instance.

```
mmu_read_set (va, sz) = do {
  pa ← mmu_translate_set va;
  mem_read (pa, sz)
}
```

We wrap up the evaluation of functions `mmu_translate_set`, `mmu_write_set` and `mmu_read_set` in the function `mem_op`, and call it `mem_op_set`.

MMU Operations: The effect that updating the page table root register has on our most abstract TLB model is similar to changing the state of the current page table through the memory write operation. In the following function `update_TTBR0_set`, we use the `ptable_comp` function with walks from the two page tables to compute the resultant inconsistent virtual addresses. We also reload the `global_set` with the updated global virtual addresses.

```
update_TTBR0_set r = do {
  (mem, ttbr0, asid, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  update_state (λs. s(|TTBR0 := r|));
  let incon_set_new =
    ptable_comp (pt_walk asid mem ttbr0) (pt_walk asid mem r);
  let global_set_new = global_varange asid mem r;
  update_incon_set (incon_set ∪ incon_set_new);
  update_global_set (global_set ∪ global_set_new)
}
```

We now explain our instruction for updating the ASID register. Until now, we have only used `incon_set` from our abstract model, while only reloading the `global_set`. The ASID update will manipulate all three components of the abstract TLB model.

The `snapshot` stores the inconsistencies and the page table state for an ASID when it was last active, i.e. the type of `snapshot` is `asid ⇒ vaddr set × (vaddr ⇒ asid tlb_entry option)`. Now that we want to switch from one ASID to another, we need to first store the inconsistencies for the active ASID to its `snapshot` so that we can preserve and later retrieve the inconsistencies when we switch back to this ASID. We will also have to retrieve the inconsistencies for the new ASID, so that we can have a sound execution with the new ASID. For this, we compute the inconsistencies from the stored `snapshot` of the new ASID and its comparison from the page table since the page table state would likely have changed through memory writes and updating the page table root register. The global inconsistencies play an interesting role: a globally mapped inconsistent addresses is inconsistent for all ASIDs. Therefore, we are required to propagate the global inconsistencies to the new ASID, we use `global_set` for this purpose. We summarise these concepts in the following three steps:

1. For updating the ASID register, we start by storing the `incon_set` and the page table state to the `snapshot` for the ASID we are *switching away from*,
2. next we update the *active* ASID to the *new* ASID,
3. and finally we compute the new `incon_set` for the *new* ASID by combining the global inconsistencies, the stored `incon_set` in its snapshot and by comparing the page table state stored for the *new* ASID and the active page table.

The `update_ASID` instruction is then instantiated as:

```
update_ASID_set a = do {
  (mem, ttbr0, asid, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  snapshot ← read_state_iset snapshot;
```

```
  let new_snp = snapshot(asid := (incon_set, pt_walk asid mem ttbr0));
  update_snapshot new_snp;
  update_state (λs. s(|ASID := a|));
  let global_incon = incon_set ∩ global_set;
     incon_set_snap = fst (new_snp a);
     ptcomp_snap =
       ptable_comp (snd (new_snp a)) (pt_walk a mem ttbr0)
  in update_incon_set (global_incon ∪ incon_set_snap ∪ ptcomp_snap)
}
```

In the above definition, we first store the `incon_set` and the page table state of the active ASID to the `snapshot`. The function `update_snapshot` updates the `snapshot` of the state with the given argument. Next we update the ASID register to the ASID `a`. For finding the `incon_set` for the ASID `a`, we

- find the globally inconsistent virtual addresses by intersecting the `incon_set` and the `global_set`,
- retrieve the stored inconsistent virtual addresses from the `snapshot` of the ASID `a`, and finally
- compare the stored page table state and the active page table using the function `ptable_comp`.

We simply update the `incon_set` with these sets of inconsistent virtual addresses using the `update_incon_set` function and the execution can continue with the new ASID under its `incon_set`. It should be noted that the order of updating the `snapshot` and then finding the `incon_set` is important: we always update the `snapshot` for the previous ASID *before* updating to the new one, since the ASID `a` could have been identical to the active ASID.

We now proceed to explaining the flush operations for our most abstract model. The flush operations remove the relevant virtual addresses from the `incon_set` and the `global_set`, and also unmap them from `snapshot` depending on the nature of the flush instruction. We trivially saturate the `global_set` with the mapped global addresses of the current page table after the flush instructions for virtual addresses; this helps us proving the refinement later. The instantiation `flush_set` for flushing either the TLB or a range of virtual addresses (irrespective of ASIDs) is:

```
flush_set f = do {
  (mem, ttbr0, asid, incon_set, global_set, snapshot) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set, snapshot);
  case f of FlushTLB ⇒ do {
              update_incon_set ∅;
              update_global_set (global_varange asid mem ttbr0);
              update_snapshot (λa. (∅, empty))
            }
  | Flushvarange vset ⇒ do {
        update_incon_set (incon_set - vset);
        update_global_set
```

```
         (global_set - vset ∪ global_varange asid mem ttbr0);
        update_snapshot
         (λa. (fst (snapshot a) - vset,
               λv. if v ∈ vset then None else snd (snapshot a) v))
      }
}
```

FlushTLB empties the incon_set and the inconsistent addresses of snapshot for all ASIDs, unmaps the stored page table state, and reloads the global_set with the consistent globally mapped virtual addresses of the current state. Similary, Flushvarange removes the given set of virtual addresses from the incon_set and from the inconsistent addresses of snapshot for all ASIDs, unmaps them from the stored page table state of all the ASIDs, and removes them from the global_set before making the global_set saturated with the globally mapped virtual addresses of the current state.

The flush operation for invalidating virtual addresses under ASIDs is instantiated as:

```
flush_with_ASID_set f = do {
  (mem, ttbr0, asid, incon_set, global_set, snapshot) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set, snapshot);
  case f of
  FlushASID a ⇒
    if a = asid then update_incon_set (incon_set ∩ global_set)
    else update_snapshot (snapshot(a := (∅, empty)))
  | FlushASIDvarange a vset ⇒
      if a = asid
      then update_incon_set (incon_set - (vset - global_set))
      else let iset = fst (snapshot a); pt = snd (snapshot a)
          in update_snapshot
              (λa'. if a' = a
                    then (iset - vset,
                          λv. if v ∈ vset then None else pt v)
                    else snapshot a')
}
```

For the active ASID, FlushASID removes the ASID-specific addresses from the incon_set. While for an inactive ASID as its argument, FlushASID unmaps the ASID's snapshot. The FlushASIDvarange repeats the same process for the given set of virtual addresses under an ASID.

We instantiate the evaluation of functions update_TTBR0_set, update_ASID_set, flush_set and flush_with_ASID_set for the function mmu_op, and call this function mmu_op_set.

Refinement We now present the refinement theorems between the saturated and the most abstract models.

The refinement relation should provide a lookup order between the abstract TLB (`incon_set`, `global_set` and `snapshot`) and the saturated TLB. Such a refinement relation is:

```
tlb_rel_set s t ≡
truncate s = truncate t ∧
ASID s = ASID t ∧
saturated s ∧
incon_addrs s ⊆ incon_set t ∧
(⋃e∈global_entries (TLB s) range_of e) ⊆ global_set t ∧
(∀a v. a ≠ ASID s ⟶
       tlb_lookup (non_global_entries (TLB s)) a v
       ≤ tlb_lookup_from (snapshot t) a v)
```

Where the function `incon_addrs` constructs the set of inconsistent addresses under the *active* ASID in the saturated TLB of state `s`:

```
incon_addrs s ≡
{va | tlb_lookup (TLB s) (ASID s) va = Incon} ∪
{va | ∃e. tlb_lookup (TLB s) (ASID s) va = Hit e ∧
          fault (pt_walk (ASID s) (MEM s) (TTBR0 s) va)}
```

This subset relation between the `incon_addrs` of state `s` and the `incon_set` of state `t` is analogous to the subset assumption of our earlier refinement between the saturated and the non-deterministic MMU model. This subset relation provides us with the TLB lookup order, hence guarantees about safe executions. We impose a similar lookup order for the global addresses cached in the saturated TLB of state `s` and between the `global_set` of state `t`.

The last conjunct of the refinement relation `tlb_rel_set` provides us with a similar order for the *inactive* ASIDs. The TLB of state `s` is not saturated for these ASIDs; therefore we assert that the stored `snapshot` of the abstract state `t` covers all possible ASID-specific executions (represented by the `tlb_lookup` on `non_global_entries`) of the underlying TLB of state `s`. Formally:

```
tlb_lookup_from snp a va ≡
let iset = fst (snp a); pt = snd (snp a)
in if va ∈ iset then Incon
   else case pt va of None ⇒ Miss
        | ⌊entry⌋ ⇒ if asid_of entry = None then Miss else Hit entry
```

The function `tlb_lookup_from` estimates a TLB lookup for the given ASID `a` and virtual address `va` from the snaphot `snp`. The resultant lookup is `Incon` if the address `va` is in the inconsistent set of the snaphot `snp`, otherwise faults and global addresses are encoded to `Miss`, and an ASID-specific entries to `Hit`.

We now present the refinement theorems.

Theorem 20. *The saturated and abstract memory operations preserve the refinement relation given the consistency of the most abstract TLB for the virtual address.*

```
                           mem_op_sat f s = (res, s')
 mem_op_set f t = (res', t')     consistent_set f t     tlb_rel_set s t
─────────────────────────────────────────────────────────────────────────
                        res' = res ∧ tlb_rel_set s' t'
```

Where `consistent_set` *ensures that memory operation is for a consistent virtual address i.e. the virtual address is not an element of* `incon_set` *of state* `t`*.*

Proof. We first explain how the refinement works for address translation. According to the refinement relation, the `incon_set` of state `t` tracks the inconsistent virtual addresses for the active ASID in the saturated TLB of state `s`. We are therefore in the `else` branch of `mmu_translate_set`, and in either the `Hit` or the `Miss` branch of `mmu_translate_sat`. In both these cases, the results must agree because `saturated` and `tlb_rel_set` say that the `Hit` and `Miss` results represent precisely the walks we perform in `mmu_translate_set`.

For memory reads, the refinement relation after the address translation holds trivially. Memory writes are interesting: we validate that the function `ptable_comp` correctly tracks the resultant inconsistencies for memory writes, we also verify that the `global_set` reload of the abstract model is sound and that its subset relation still holds with the global entries of the saturated state.

Note that the last conjunct of `tlb_rel_set` is trivially true in this refinement, because the memory operations are being carried out under the active ASID. □

Theorem 21. *The saturated and abstract MMU operations preserve the refinement relation* `tlb_rel_set`*.*

```
              mmu_op_sat f s = ((), s')
 mmu_op_set f t = ((), t')     tlb_rel_set s t
──────────────────────────────────────────────
              tlb_rel_set s' t'
```

Proof. The proof strategy for the `update_TTBR0` instantiations is similar to that of memory writes, as the `ptable_comp` comparison is inherently the same. The proof for flush instructions includes set reasoning about preserving the refinement relation after the respective `incon_set`, `global_set` and `snapshot` updates.

The refinement proof for the `update_ASID` instruction is interesting: overall we are required to establish that the `incon_set` correctly models the inconsistencies of the saturated TLB after updating the ASID, the `global_set` holds the subset relation and the `snapshot` provides the lookup order for the all the inactive ASIDs. The

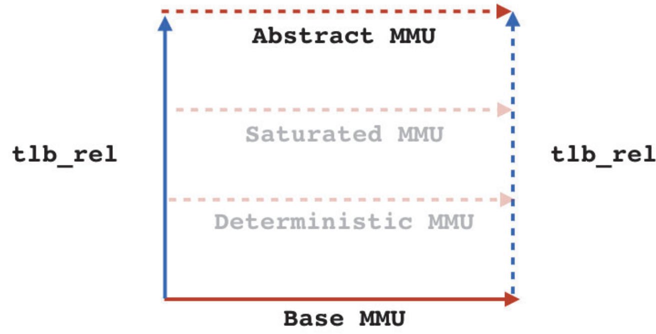

Figure 5.5: Refinement between Nondeterministic and Abstract MMU

main observation that makes the proof possible is that the saturated TLB can be classified into its global and non-global entries. We can then reason about the disjoint lookup order for the non-global entries. The global entries are reasoned about by the subset relation of the global entries and `global_set`. The `snapshot` order is preserved by proving that the `ptable_comp` function correctly detects the inconsistencies. □

With this we conclude our most abstract MMU model and its refinement with the saturated MMU model.

5.2.4 Joining the Refinement Levels

In this section, we join the refinement levels of Figure 5.1 to show that our most abstract model is sound with respect to the base model. The resultant refinement is between the abstract and the base model through the deterministic and saturated MMU models as shown in Figure 5.5. The refinement relation `tlb_rel` is:

```
tlb_rel r t ≡
∃s s'. tlb_rel_det r s ∧ tlb_rel_sat s s' ∧ tlb_rel_set s' t
```

Where the state `r` has the nondeterministic TLB, the state `s` has the deterministic TLB, the state `s'` has the saturated TLB and the state `t` has the most abstract TLB. The functions `tlb_rel_det`, `tlb_rel_sat` and `tlb_rel_set` are the refinement relations provided in Sect. 5.2.1, Sect. 5.2.2 and Sect. 5.2.3 respectively. We then have two refinement theorems, presented below and also shown in Figure 5.6 and Figure 5.7.

Theorem 22. *Refinement between the nondeterministic and the abstract memory operations.*

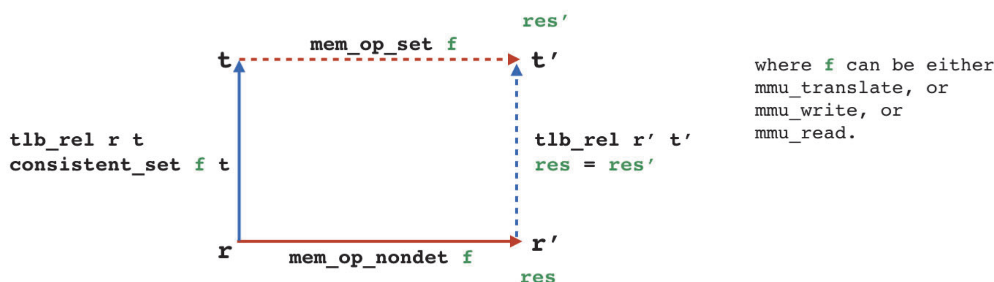

Figure 5.6: Refinement between Nondeterministic and Abstract Memory Operations

$$\frac{\texttt{mem_op_nondet f r = (res, r')} \qquad \texttt{mem_op_set f t = (res', t')} \qquad \texttt{consistent_set f t} \qquad \texttt{tlb_rel r t}}{\texttt{res = res'} \wedge \texttt{tlb_rel r' t'}}$$

where `consistent_set` *ensures that the abstract state* `t` *is TLB-consistent, i.e. the given virtual address is not an element of the* `incon_set` *of state* `t`.

Proof. By case analysis on the function `f` and using the respective refinement theorems of Sect. 5.2.1, Sect. 5.2.2 and Sect. 5.2.3, and observing that for each level, the address consistency condition on this level implies address consistency on the level below. □

Theorem 23. *Refinement between nondeterministic and abstract MMU maintenance operations.*

$$\frac{\texttt{mmu_op_nondet f r = ((), r')} \qquad \texttt{mmu_op_set f t = ((), t')} \qquad \texttt{tlb_rel r t}}{\texttt{tlb_rel r' t'}}$$

Proof. By case analysis on the function `f` and using the respective refinement theorems of Sect. 5.2.1, Sect. 5.2.2 and Sect. 5.2.3. □

With this we conclude presenting the refinement stack.

5.3 Summary and Remarks

In this chapter, we have built on the MMU model of Chapter 4 to introduce ASIDs and global tags for the ARMv7-style TLB. We have identified the additional reasoning complexities these features entail, have developed a refinement stack

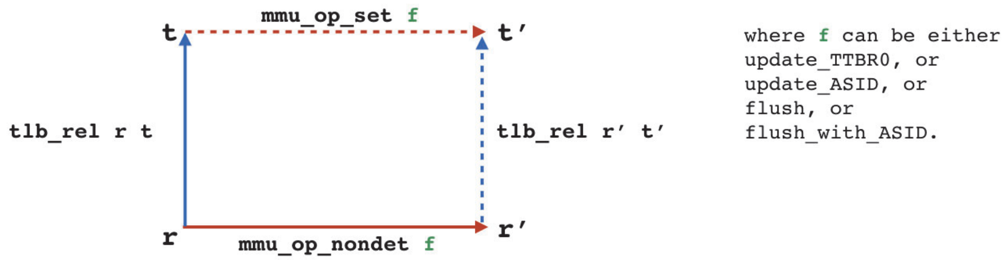

Figure 5.7: Refinement between Nondeterministic and Abstract MMU Operations

that abstracts away the hardware details and have reached at an abstract model of a TLB with ASIDs and global entries that is easier to reason about.

The abstract model has three components for modeling the TLB: a set of inconsistent virtual addresses for the active ASID-specific and globally mapped virtual addresses, a set of globally mapped virtual addresses and a snapshot of the page table state modulo the inconsistent addresses for all ASIDs. The main message of the refinement chain presented in this chapter is that any logic taking this abstract model as its memory interface would avoid the reasoning complexities of the actual hardware state of the TLB. This model is sound to reason about for the implementations of ARMv7-A architecture that do not have the separate PDC to cache the partial walks. The model and refinement chain of this chapter is available online in the form of Isabelle theories (Syeda, 2019).

In the next chapter (Chapter 6), we build on the MMU model of this chapter to formalise a two-stage TLB, and again abstract away the hardware details.

CHAPTER

SIX

A Formal Model of the ARMv7-A MMU with Two-Stage TLB

In the previous chapter, we have presented an operational model of the ARMv7-A memory management unit (MMU) where the TLB caches page table entries under ASIDs and global tags. In this chapter, we integrate that model with a separate page directory cache (PDC) to develop a two-stage TLB model caching partial and complete page table walks. We again build a refinement stack to abstract away the hardware details and also explain how our refinement framework handles the additional PDC. In Chapter 7, we use the most abstract model of this chapter as the memory model of a program logic for reasoning about programs in the presence of cached address translation.

This chapter is organised as: we instantiate the generic TLB model presented in Sect. 4.2 to a PDC caching partial walks from two-level page tables. We then provide the base MMU model including address translation, memory operations, TLB maintenance operations as well register update instructions affecting the state of the two-stage TLB. Next we provide a series of refinements to abstract away the hardware details of this base MMU model. For each refinement level, we first explain the model and then provide the refinement theorems collectively. In the end we join the refinement levels and conclude the chapter.

This chapter is based on the published work (Syeda and Klein, 2017) and the submitted work (Syeda and Klein, 2019).

6.1 ARMv7-A MMU Model with TLB and PDC

We now present a formal MMU model for the ARMv7-A architecture (ARM, 2008) that includes a two-stage TLB for caching partial and complete page table walks. As mentioned in the virtual memory chapter (Sect. 3.2.4), the recent implementations of ARMv7-A implement a two-stage TLB that collectively caches entries and the machinery required for resolving address translations using the page table from main memory when needed. The first stage caches entries that provide end-to-end address translations, i.e. results of complete page table lookups, we simply call this stage “the TLB”. The second stage, called the page directory cache (PDC), caches the results of partial page table lookups – up to the first-level traversal of the page table only. For a two-level page table this means that the PDC stores translation entries for sections and supersections, and pointers to the second level page table containing small and large page translation entries.

While resolving virtual addresses, the processor checks the PDC on every TLB miss before consulting the page table present in the main memory. If the processor finds a cached PDC entry, it simply completes the translation either by a page table walk (for small and large pages) or by resolving the virtual address directly (for sections and supersections). Figure 6.1 gives an overview of the information cached

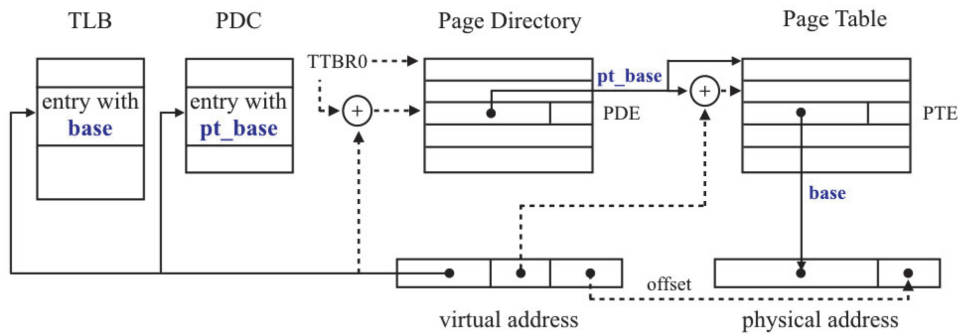

Figure 6.1: Address Translation in the Presence of a TLB and PDC

in TLB and PDC for a virtual address mapped to a small page of the memory. We formalise the PDC and its effect on address translation and subsequently the MMU operations. Similar to the previous chapter, we develop a refinement stack for the base MMU model, but now with TLB and PDC. In the next chapter, we use the most abstract model we develop here as the memory model for the logic we use to reason about programs in the presence of cached address translation.

Why consistency of the PDC is important to establish: The TLB model presented in the previous chapter (Chapter 5) has a single TLB with ASIDs and global tags. This model is sound for the ARMv7-A implementations that do not have the PDC. However, in the presence of the PDC, the model fails to detect some inconsistent scenarios. We explain a specific example below.

Example

In the MMU model of the previous chapter, the TLB entries contain end-to-end physical base addresses. This leaves a hole in the model for hardware caching partial page table walks in the PDC. Consider a page directory entry `E` that points to a page table `A`. Now if we do the following (Figure 6.2 gives an overview):

1. make a copy `B` of the page table `A`, then
2. update `E` to point to `B`, then
3. change an entry of page table `A`.

These steps do not affect the final result of the page table translation, so the `pt_walk` output remains the same. However, hardware with a two-stage TLB may be caching `E` in the PDC, and then a memory access in the corresponding area may still cause lookups in the page table `A`, which has changed, resulting in the inconsistency.

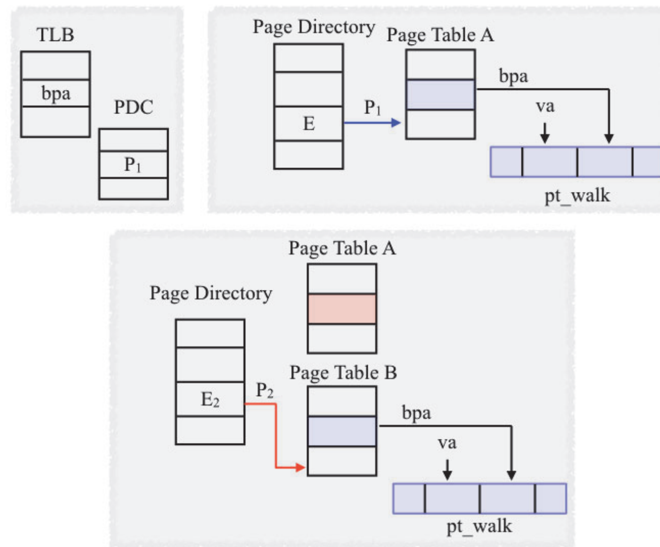

Figure 6.2: PDC Inconsistency Example

To overcome this problem, it should suffice to record the addresses of the intermediate page tables in the `'a tlb_entry` type. Alternatively, the state could contain separate TLBs for each level of the virtual to physical address translation. We opt for the second approach as it is closer to the actual hardware and introduce a separate PDC in the MMU model that caches the results of first-level page table walks.

We formalise the PDC similarly to how we have formalised the TLB in Sect. 4.2: *The PDC is a set of PDC entries.* As we model only two types of page table entries in our work, the PDC entries specify either 32-bit base addresses for sections or 32-bit pointers to the second level page tables.

```
type_synonym pdc = pdc_entry set

datatype pdc_entry =
          PDE_Table (asid) (12 word) (32 word)
        | PDE_Section (asid option) (12 word) (32 word) flags
```

Where the type `asid` is an abbreviation for `8 word`. We always associate an ASID with `PDE_Table` since the hardware does not provide global tables (only global pages). The constructor `PDE_Section` can have an ASID or a global tag depending on the `nG` bit of its `flags`.

PDC lookup With the PDC state formalised, we now describe its lookup. For any given 32-bit virtual address, a PDC lookup finds the corresponding PDC entry. The lookup of the PDC is modeled using the type `'e lookup_type` from Sect. 4.2.

In order to find the virtual address range of a PDC entry, we simply instantiate

the `range_of` parameter of type class `entry_op` (presented in Sect. 4.2, Page 44) for `pdc_entry` as:

```
range_of :: pdc_entry ⇒ vaddr set
range_of e ≡
Addr ` {base_addr (vba_of e)..base_addr (vba_of e) + (2^20 - 1)}

base_addr v ≡ UCAST('a → 32) v << 32 - size v
```

Where `vba_of e` returns the `12 word` virtual base address stored in `e`. For notation of operations over the types `set` and `word`, please refer to Sect. 2.2.3 of the notation chapter. The PDC lookup is then defined using the functions `lookup` and `entry_set` from Sect. 4.2 as:

```
pdc_entry_set :: pdc ⇒ vaddr ⇒ pdc
pdc_entry_set p a va ≡
{e ∈ entry_set p va | asid_of e = None ∨ asid_of e = ⌊a⌋}
```

abbreviation `pdc_lookup p a va = lookup (pdc_entry_set p a) va`

Given a virtual address `va` and an ASID `a`, the function `pdc_entry_set` is a filter for the matched `entry_set`: an entry matching the virtual address `va` has to be either global or under the same ASID `a` in order to provide translation for the virtual address `va`. The function `asid_of` returns the ASID field of a TLB entry.

Similar to the TLB, we can also partition the PDC into its `global_entries` (entries with `None` ASID) and `non_global_entries` (entries with `Some` ASIDs). The `PDE_Table` entries are always `non_global_entries`, whereas the `PDE_Section` entries can be global or non-global entries.

From PDC Entry to TLB Entry: The aim of a PDC lookup is to find the respective PDC entry and decode it into the form of a TLB entry, so that we can at least partially reuse our existing definitions for the PDC. We write a function `pdc_to_tlb` to convert a PDC entry to the corresponding TLB entry. For a `PDE_Table` entry, that holds a pointer to the page table, we simply complete the page table walk for a virtual address `va` from the memory `mem`:

```
pdc_to_tlb (PDE_Table a vba pba) mem va =
(case get_pte mem (Addr pba) va of None ⇒ None | ⌊InvalidPTE⌋ ⇒ None
 | ⌊SmallPagePTE p' perms⌋ ⇒ ⌊to_sml_entry p' perms va a⌋)
```

In the above definition, the function `get_pte` finds the page table entry and the function `to_sml_entry` simply encodes information into a small TLB entry. From a `PDE_Section`, that already provides the base physical address of the section it belongs to, we directly encode an `asid tlb_entry`:

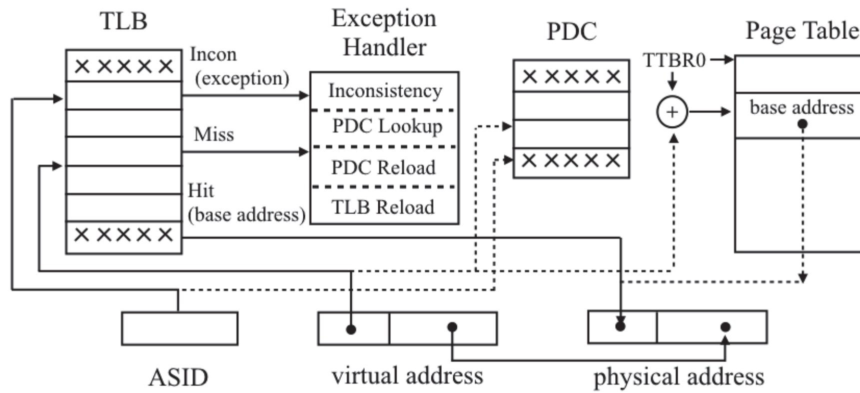

Figure 6.3: ARMv7-A MMU with TLB and PDC

```
pdc_to_tlb (PDE_Section a vba pba fl) mem va =
⌊EntrySection a (UCAST(32 → 12) (addr_val va >> 20))
  (UCAST(32 → 12) (pba >> 20)) fl⌋
```

Figure 6.3 gives an overview of the ARMv7-A MMU with the TLB and the PDC. To formalise it, we use Isabelle's extensible records (Naraschewski and Wenzel, 1998) to extend the record type `state` with the type `asid` × `TLB` × `pdc` which will contain the active ASID register and the hardware state of TLB and PDC. We then instantiate this extended state for type class `mmu_extended` of Sect. 5.1. This gives us the base MMU model that has both the TLB and the PDC with instructions for address translation, memory read and write, updating the page table root and ASID registers and TLB maintenance operations. We group these operations into the types `mem_op_typ` and `mmu_op_typ` (defined previously on Page 72) for presenting the MMU models and their refinement.

We now present the base MMU model.

6.1.1 Page Table Walk

As explained in the example of the previous section (Page 97), we are required to update our page table walk interface to soundly model the presence of the PDC. For a two-stage TLB reload from the two-level page table, we classify page table walks as:

```
datatype pt_walk_typ  =  Fault
                      |  Partial_Walk (pdc_entry)
                      |  Full_Walk (asid tlb_entry) (pdc_entry)
```

For a virtual address `va` a page table walk resulting in a `Fault` represents an un-

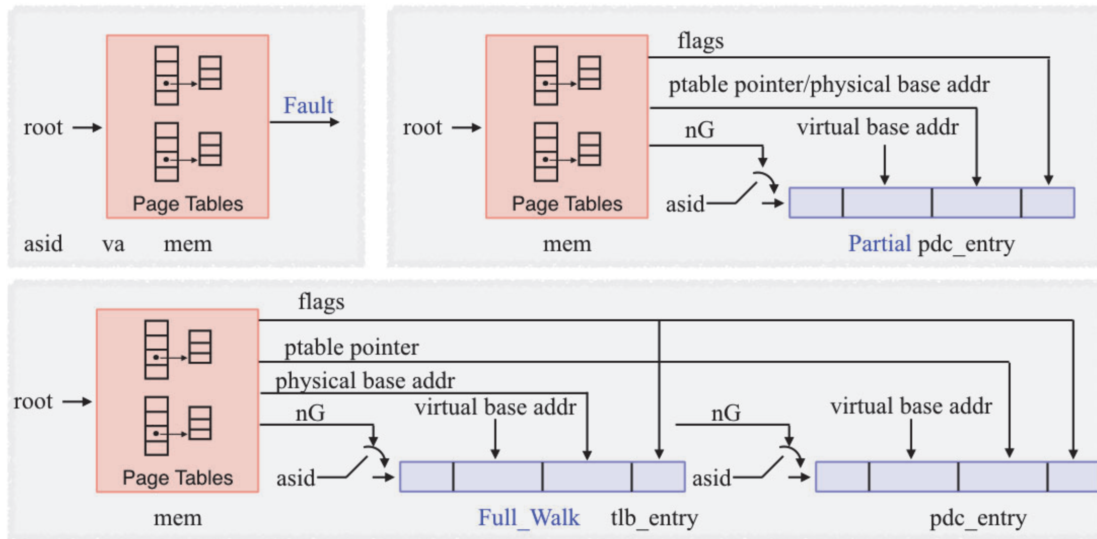

Figure 6.4: The Encoding of Two-Stage Page Table Walks

mapped virtual address, the `Partial_Walk` with a PDC entry means that either the virtual address `va` belongs to a mapped section, or its translation can be continued using the page table presents at the specified location in memory. Whereas the `Full_Walk` with TLB and PDC entries represents that the virtual address `va` is fully mapped: it either belongs to a section, or to a mapped small page of the memory. Note that a `Partial_Walk` with a `PDE_Table` entry does not ensure that a `Full_Walk` exists.

Figure 6.4 gives an overview how the two-stage page table walk is encoded. We then define `ptd_walk` (page table-directory walk) as:

```
ptd_walk :: asid ⇒ heap ⇒ paddr ⇒ vaddr ⇒ pt_walk_typ
ptd_walk a mem rt va ≡
case pdc_walk a mem rt va of None ⇒ Fault
| ⌊pde⌋ ⇒
    case pdc_to_tlb pde mem va of None ⇒ Partial_Walk pde
    | ⌊tlbentry⌋ ⇒ Full_Walk tlbentry pde
```

For a given virtual address `va`, the function `ptd_walk` walks the first level of the page table present at the root `rt` in memory `mem` under the ASID `a`. This first level walk is formalised using the function `pdc_walk`, which we explain in the next paragraph. For an invalid first level entry we simply return `Fault`, while we convert a valid page table entry to either `Partial_Walk` or `Full_Walk` using the function `pdc_to_tlb`.

The function `pdc_walk` decodes the first level page table entry into a PDC entry (for the notation of operations for the type `word`, please refer to Sect. 2.2.3 in the notation chapter):

```
pdc_walk :: asid ⇒ heap ⇒ paddr ⇒ vaddr ⇒ pdc_entry option
pdc_walk a mem rt va ≡
case get_pde mem rt va of None ⇒ None
| ⌊PageTablePDE p⌋ ⇒
    ⌊PDE_Table a (UCAST(32 → 12) (addr_val va >> 20)) (addr_val p)⌋
| ⌊SectionPDE bpa perms⌋ ⇒ ⌊to_sec_pde bpa perms a va⌋ | ⌊_⌋ ⇒ None
```

Where the function `to_sec_pde` encodes base physical address and permission bits for the virtual address `va` under the ASID `a` to a section TLB entry.

6.1.2 Memory Operations

We now explain memory operations including address translation, memory read and write for our MMU model.

Address Translation: The address translation for memory operations is defined as:

```
mmu_translate va = do {
  update_state (λs. s(|TLB_PDC := TLB_PDC s - tlb_evict s|));
  (asid, tlb, pdc) ← read_state (ASID, TLB_PDC);
  case tlb_lookup tlb asid va of
  Miss ⇒ pdc_lookup_reload_translation va
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

As in previous chapters, the function `mmu_translate` first evicts an underspecified set of entries from TLB and PDC. The next step in `mmu_translate` after reading out the hardware state is to do a TLB lookup for the virtual address `va` to be translated under the current ASID. The TLB-`Miss` case is different from the base model of single-stage TLB, but the `Incon` and `Hit` cases are similar as in previous chapters. We now explain them. If the result of that lookup is `Incon`, the machine raises an unrecoverable exception and halts, expressing the fact that in normal operation, this state should never be encountered. If the result is `Hit entry`, we translate this TLB `entry` to the corresponding physical address `pa` using the function `va_to_pa` and return that address.

If the result of the TLB lookup is `Miss`, using the `pdc_lookup_reload_translation` function we perform the PDC lookup, the potential TLB reload and the address translation. It is defined as:

```
pdc_lookup_reload_translation va = do {
  (asid, tlb, pdc) ← read_state (ASID, TLB_PDC);
  case pdc_lookup pdc asid va of Miss ⇒ translation_tlb_pdc_reload va
```

```
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit pde ⇒ tlb_reload_trans pde va
}
```

In the above function, if the result of PDC lookup is `Incon` the machine raises an unrecoverable exception and halts. If the PDC lookup results in `Hit pde`, we use the function `tlb_reload_trans` to complete the partial translation for address `va` from the page directory entry `pde` and store the result in the TLB.

```
tlb_reload_trans pde va = do {
  (mem, ttbr0, asid) ← read_state (MEM, TTBR0, ASID);
  let entry = pdc_to_tlb pde mem va;
  if fault entry then raise PAGE_FAULT
  else do {
        update_state
         (λs. s⦇TLB_PDC := TLB_PDC s ∪ ({the entry}, ∅)⦈);
        return (va_to_pa va (the entry))
      }
}
```

Going back to the `pdc_lookup_reload_translation` function, if the result of the PDC lookup is `Miss`, we use the function `translation_tlb_pdc_reload` to perform the full address translation from the page table using the function `ptd_walk`, and potentially reload both PDC and TLB:

```
translation_tlb_pdc_reload va = do {
  (asid, mem, ttbr0) ← read_state (ASID, MEM, TTBR0);
  let walk = ptd_walk asid mem ttbr0 va;
  case walk of Fault ⇒ raise PAGE_FAULT
  | Partial_Walk pde ⇒ do {
        update_state (λs. s⦇TLB_PDC := TLB_PDC s ∪ (∅, {pde})⦈);
        raise PAGE_FAULT
      }
  | Full_Walk entry pde ⇒ do {
        update_state (λs. s⦇TLB_PDC := TLB_PDC s ∪ ({entry}, {pde})⦈);
        return (va_to_pa va entry)
      }
}
```

In the above function, if the result of the `ptd_walk` is the `Fault`, we raise this fault which will cause the machine to jump to the appropriate exception handler, and if we get the `Partial_Walk` we reload the PDC and again raise the page table fault. If the result is `Full_Walk` with the TLB and PDC entries we simply add them to the TLB and PDC respectively, and execute address translation as in the `Hit` case.

Memory Write and Read: Reusing the original physical memory functions `mem_write` and `mem_read` from the ARM model, the instances for the memory operations of the MMU model are straightforward:

```
mmu_write (val, va, sz) = do {
  pa ← mmu_translate va;
  when_no_exc mem_write (val, pa, sz)
}
```

```
mmu_read (va, sz) = do {
  pa ← mmu_translate va;
  mem_read (pa, sz)
}
```

The definitions are unchanged from previous chapters, but now use the new `mmu_translate` function of this chapter.

Evaluation: We instantiate the function `mem_op` for the state with `asid` and the nondeterministic TLB and PDC, and call this instantiation `mem_op_nondet`. The generic interface of the function `mem_op` and the type class `mmu_extended` then picks up the definitions for `translate`, `read` and `write` presented above (Page 72).

6.1.3 MMU Operations

We now explain MMU operations including updating the page table root and ASID registers and flush operations.

Updating the Page Table Root Register: In this base model, the instruction `update_TTBR0` merely does what its name describes, and as in the base models of the previous chapter, this instruction does not entail TLB eviction:

```
update_TTBR0 r = update_state (λs. s⦇TTBR0 := r⦈)
```

Updating the ASID Register: The `update_ASID` instruction merely does what its name describes, and similar to the `update_TTBR0` instruction, it does not entail TLB eviction:

```
update_ASID a = update_state (λs. s⦇ASID := a⦈)
```

Flush Operations: The flush instructions operate on both the TLB and the PDC. We have defined the flush types `flush_type` and `asid_flush_type` in the previous chapter (Sect. 5.1.3, Page 76). The instantiation of these flush instruction for this base model is:

```
flush f ≡
case f of FlushTLB ⇒ update_state (λs. s(|TLB_PDC := (∅, ∅)|))
| Flushvarange vset ⇒
    update_state (λs. s(|TLB_PDC := flush_vset (TLB_PDC s) vset|))
```

```
flush_with_ASID f ≡
case f of
FlushASID a ⇒
  update_state (λs. s(|TLB_PDC := flush_asid (TLB_PDC s) a|))
| FlushASIDvarange a vset ⇒
    update_state
     (λs. s(|TLB_PDC := flush_asid_vset (TLB_PDC s) a vset|))
```

The `FlushTLB` instruction makes the TLB and PDC sets empty, whereas the `Flushvarange` instruction takes a pair `(TLB × pdc)` and flushes the entries matching the given set of virtual addresses, i.e.,

```
flush_vset tp vset =
(let tlb = fst tp; pdc = snd tp
 in (tlb - (⋃v∈vset {e ∈ tlb | v ∈ range_of e}),
     pdc - (⋃v∈vset {e ∈ pdc | v ∈ range_of e})))
```

`FlushASID` flushes all entries under the given ASID:

```
flush_asid tp a =
(let tlb = fst tp; pdc = snd tp
 in (tlb - {e ∈ tlb | asid_of e = ⌊a⌋},
     pdc - {e ∈ pdc | asid_of e = ⌊a⌋}))
```

Finally, `FlushASIDvarange` flushes the entries for the given set of virtual addresses under the given ASID:

```
flush_asid_vset t a vset =
(let tlb = fst t; pdc = snd t
 in (tlb - (⋃v∈vset {e ∈ tlb | v ∈ range_of e ∧ asid_of e = ⌊a⌋}),
     pdc - (⋃v∈vset {e ∈ pdc | v ∈ range_of e ∧ asid_of e = ⌊a⌋})))
```

Evaluation: We instantiate the function `mmu_op` for the state with `asid` and the nondeterministic TLB and PDC, and call this instantiation `mmu_op_nondet`. The generic interface of the function `mmu_op` and the type class `mmu_extended` then picks up the above presented definitions.

6.2 MMU Abstraction

Similar to the base model of the previous chapters, the MMU model of the previous section gives us the ground truth of how hardware operates, and thereby the foundation for a logic for programs under the two-stage TLB with ASIDs and global translation entries, but this model is hard to reason about directly. It inherits the reasoning complexities of an MMU model with a single-stage TLB as identified in Sect. 5.2. It also involves an additional PDC lookup potentially on every memory transaction, as demonstrated by the address translation function `mmu_translate` of the previous section.

In this section, we show how we construct a model that avoids the PDC and TLB lookups and produces sufficient conditions for safe execution. We have already observed from the abstraction chains of the single-stage MMU models (Sect. 4.4 and Sect. 5.2) that a TLB lookup forms an order with `Miss` being the bottom element and `Incon` the top. Since we have modeled the PDC with the same `'e lookup_type`, a PDC lookup forms the same order and we can prove monotonicity.

Lemma 4. `t` $\subseteq$ `t'` $\Longrightarrow$ `pdc_lookup t a v` $\leq$ `pdc_lookup t' a v`

Proof. By case distinction and unfolding the definitions. □

Similar to the refinement stack of the single-stage TLB model (Sect. 5.2), we build a series of formal abstractions of the concrete two-stage TLB model that are increasingly easier to reason about, but preserve functionality and the optimisation opportunities OS developers must be able to exploit.

Our refinement stack is shown in Figure 6.5 and this time it consists of two levels. In the first level, we hierarchically saturate the PDC and the TLB with the mapped state of page table, this eliminates nondeterminism, state change for the memory reads and the PDC lookup from the model. Next we abstract the PDC and the TLB completely and reach at the most abstract MMU model similar to the abstract model of single-stage TLB (Sect. 5.2.3). The highlight of our most abstract two-stage TLB model is that it has the same three components as the single-stage most abstract model: the `incon_set`, the `global_set` and the `snapshot`. The only difference between these models is the updated page table interface with `pt_walk_typ` and now a more conservative page table comparison function.

As before, for each level up in the refinement stack of Figure 6.5 we prove that its abstraction preserves a refinement relation and is sound with respect to its immediate concrete MMU model. We then join these refinement levels in order to prove the soundness of the most abstract model with the base model.

As in the previous models we need to be able to express what it means that the TLB and the PDC are currently in a consistent state for an address to be accessed.

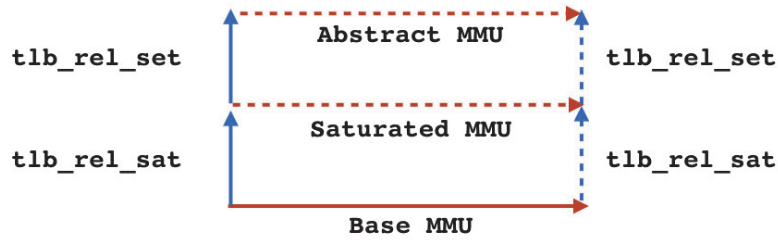

Figure 6.5: Refinement Stack for MMU Models

We formalise consistency of TLB and PDC for a virtual address in the base and saturated model as:

```
consistent mem root tlb pdc asid va ≡
tlb_consistent mem root tlb asid va ∧
pdc_consistent mem root pdc asid va
```

Where

```
tlb_consistent mem root tlb asid va ≡
tlb_lookup tlb asid va = Hit (the (pt_walk asid mem root va)) ∧
no_fault (pt_walk asid mem root va) ∨
tlb_lookup tlb asid va = Miss


pdc_consistent mem root pdc asid va ≡
pdc_lookup pdc asid va = Hit (the (pdc_walk asid mem root va)) ∧
no_fault (pdc_walk asid mem root va) ∨
pdc_lookup pdc asid va = Miss
```

Similar to the consistency condition of previous chapters, the above condition combines internal consistency of the TLB and the PDC (no `Incon` results permitted), with external consistency, i.e. synchronicity with the current state of the mapped page table for this particular address. The condition for `Hit` ensures that TLB and PDC do not store translation entries for unmapped virtual addresses (`no_fault`). We will again see in the most abstract model that the condition can be greatly simplified.

We now provide our MMU models and their refinement.

6.2.1 The Saturated MMU Model

In this abstraction, we remove nondeterminism and state change on memory reads by removing the TLB's and the PDC's eviction and also by saturating them hierarchically with the mapped page table entries after every memory and MMU

operation. We leave out a separate refinement step for only removing eviction because the TLB and PDC evictions can influence each other, and it is therefore easier to merge eviction abstraction with saturation. We observe that this saturation enables us to also eliminate the potential PDC lookup while resolving the virtual addresses: we first saturate the PDC with the mapped page table entries of the current state, and then from this saturated PDC we carry out page table walks and saturate the TLB. This way the TLB contains all the mapped entries for the current page table through the saturated PDC, and a TLB `Miss` implies a page table fault. Figure 6.6 gives an overview. The critical aspect of our saturation framework is that we propagate the inconsistencies of the PDC for the active state entirely to the TLB, this is required to capture the complete information stored in the PDC in the TLB.

Memory Operations: We instantiate the parameter `mmu_translate` of the type class `mmu_extended` for the deterministic and saturated two-stage TLB and name it `mmu_translate_sat`:

```
mmu_translate_sat va = do {
  pdc_tlb_refill;
  (asid, tlb, pdc) ← read_state (ASID, TLB_PDC);
  case tlb_lookup tlb asid va of Miss ⇒ raise PAGE_FAULT
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}

pdc_tlb_refill = do {
  (m, rt, a) ← read_state (MEM, TTBR0, ASID);
  let pdes = ran (pdc_walk a m rt);
     entries = these (⋃v to_tlb pdes a m rt v)
  in update_state (λs. s(|TLB_PDC := TLB_PDC s ∪ (entries, pdes)|))
}
```

The call to `pdc_tlb_refill` at the beginning of `mmu_translate_sat` achieves the saturation mentioned above by adding all mapped page directory entries and then all mapped TLB entries for the current state and ASID. Here, `ran f = {b | ∃a. f a = ⌊b⌋}` and `these f' ≡ the ` (f' ∩ {x | x ≠ None})`. Next we do the TLB lookup for the given virtual address `va`: `Incon` still leads to the same exception as before. `Miss` implies a page fault, since the TLB is saturated with the mapped entries, and `Hit` gives us the respective physical address.

The function `to_tlb` in the function `pdc_tlb_refill` plays an important role to soundly capture the information cached in the PDC: it takes a PDC and essentially returns a set of TLB entries for a given virtual address and an ASID through the PDC lookup and the page table walk from the memory:

```
to_tlb pdc asid mem rt va =
(case pdc_lookup pdc asid va of Miss ⇒ {pt_walk asid mem rt va}
```

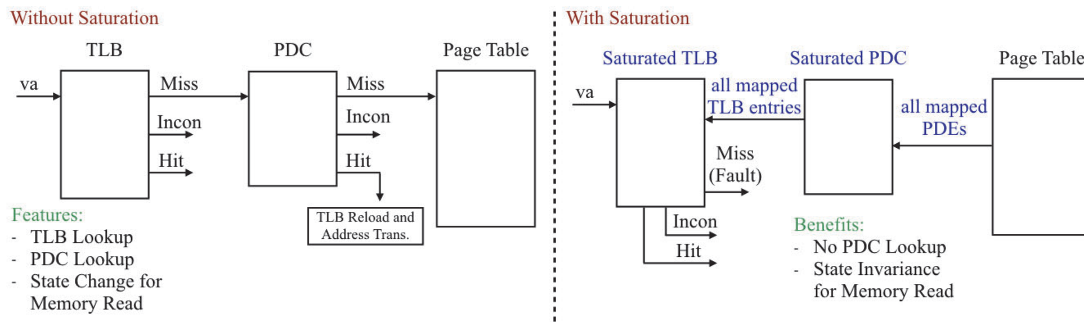

Figure 6.6: Hierarchical Saturation of PDC and TLB with Current Page Table

```
| Incon ⇒ (λx. pdc_to_tlb x mem va) ‘ pdc_entry_set pdc asid va
| Hit pde ⇒ {pdc_to_tlb pde mem va})
```

For a virtual address `va` and an ASID `a`, we first do the PDC lookup in the above function. If the result of the PDC lookup is `Miss`, we simply return the `pt_walk` as a singleton set containing either a `Some` TLB entry or `None` (in case of a page table fault). If the PDC lookup results in `Hit`, we complete the page table walk using the function `pdc_to_tlb`. The `Incon` PDC lookup is interesting: we take all matching PDC entries (using `pdc_entry_set`) and return a set of respective TLB entries. This way the inconsistency in the PDC is preserved in the TLB.

For this MMU model, we also change the memory operations to preserve saturation. The new instantiations for the saturated PDC and TLB are `mmu_write_sat` and `mmu_read_sat`:

```
mmu_write_sat (val, va, sz) = do {
  pa ← mmu_translate_sat va;
  when_no_exc do { mem_write (val, pa, sz); pdc_tlb_refill }
}
```

```
mmu_read_sat (va, sz) = do {
  pa ← mmu_translate_sat va;
  mem_read (pa, sz)
}
```

The function `mmu_write_sat` refills the two-stage TLB after the write operation to maintain saturation, as this write could have been to a page table present in the memory. The function `mmu_read_sat` achieves this saturation implicitly in the start through the `mmu_translate_sat` function, as reading from the memory does not affect the state of page table.

We wrap up the evaluation of functions `mmu_translate_sat`, `mmu_write_sat` and

`mmu_read_sat` in the function `mem_op`, and call this function `mem_op_sat`.

MMU Operations: Similar to the memory operations, we saturate the two-stage TLB after updating the page table root and ASID register with the mapped state of the page table under the active ASID. We also saturate the two-stage TLB after the flush operations. We saturate the two-stage TLB after the flush operations so that we can maintain the invariant that it always captures the mapped state of the page table. The instantiations of MMU operations for this model are presented below.

```
update_TTBR0_sat r = do {
  update_state (λs. s(|TTBR0 := r|));
  pdc_tlb_refill
}
```

```
update_ASID_sat a = do {
  update_state (λs. s(|ASID := a|));
  pdc_tlb_refill
}
```

The instantiations for flush operations are:

```
flush_sat f = do {
  (case f of FlushTLB ⇒ update_state (λs. s(|TLB_PDC := (∅, ∅)|))
   | Flushvarange vset ⇒
       update_state (λs. s(|TLB_PDC := flush_vset (TLB_PDC s) vset|)));
  pdc_tlb_refill
}
```

```
flush_with_ASID_sat f = do {
  (case f of
   FlushASID a ⇒
     update_state (λs. s(|TLB_PDC := flush_asid (TLB_PDC s) a|))
   | FlushASIDvarange a vset ⇒
       update_state
        (λs. s(|TLB_PDC := flush_asid_vset (TLB_PDC s) a vset|)));
  pdc_tlb_refill
}
```

We wrap up the evaluation of the functions `update_TTBR0_sat`, `update_ASID_sat`, `flush_sat` and `flush_with_ASID_sat` in the function `mem_op_sat`.

Refinement: We now present the refinement between the deterministic and the saturated MMU models. The refinement relation is:

```
tlb_rel_sat s t ≡
let tlb_nondet = fst (TLB_PDC s); pdc_nondet = snd (TLB_PDC s);
   tlb_sat = fst (TLB_PDC t); pdc_sat = snd (TLB_PDC t)
in truncate s = truncate t ∧
  ASID s = ASID t ∧
  tlb_nondet ⊆ tlb_sat ∧ pdc_nondet ⊆ pdc_sat ∧ saturated t
```

This relation demands that the states `s` and `t` differ only in the contents of their TLBs and PDCs. The nondeterministic TLB of state `s` has fewer entries than the saturated TLB of state `t`, and the nondeterministic PDC of state `s` has fewer entries than the saturated PDC of state `t`. These subset relations provide the lookup order between the states `s` and `t`. The `saturated` assertion represents that both TLB and PDC of state `t` are saturated with the mapped page table under the active ASID:

```
saturated t ≡ tlb_saturated t ∧ pdc_saturated t
```

Where

```
tlb_saturated t ≡
let tlb_sat = fst (TLB_PDC t)
in ran (pt_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ tlb_sat
pdc_saturated t ≡
let pdc_sat = snd (TLB_PDC t)
in ran (pdc_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ pdc_sat
```

Theorem 24. *The nondeterministic and saturated memory operations preserve the refinement relation* `tlb_rel_sat` *given the consistency of the two-stage saturated TLB for the virtual address.*

```
                    mem_op_nondet f s = (res, s')
 mem_op_sat f t = (res', t')     consistent_sat f t     tlb_rel_sat s t
-------------------------------------------------------------------------
                    res' = res ∧ tlb_rel_sat s' t'
```

Where `consistent_sat` *ensures that memory operation is for a consistent virtual address with respect to the two-stage saturated TLB including the PDC.*

Proof. For address translation, we observe that the TLB and the PDC of state `s` are consistent given the subset relationship, and the fact that the TLB and the PDC of state `t` are consistent. The lookup for `va` in both the states `t` and `s` will either produce `Miss` or `Hit`. When the saturated-TLB of state `t` produces `Miss` (implies a page table fault), the nondeterministic TLB of state `s` also produces a `Miss` and we will be in the PDC lookup case. The nondeterministic PDC of state `s` then has to conform with the saturated-PDC of state `t`: producing either `Miss` or `Hit` with a consistent page directory entry and completing the translation for the

address `va` through page table walk to eventually encounter the page table fault. In the case of `Hit` with an entry in the saturated-TLB of state `t`, the TLB of `s` either agrees on the same entry with `Hit`, or performs a consistent PDC lookup and page table walk. The refinement for memory read and write follows directly from the refinement of the address translation. □

Theorem 25. *The nondeterministic and saturated MMU operations preserve the refinement relation* `tlb_rel_sat`*.*

```
          mmu_op_nondet f s = ((), s')
mmu_op_sat f t = ((), t')     tlb_rel_sat s t
-----------------------------------------------
              tlb_rel_sat s' t'
```

Proof. By using operational definitions and basic set reasoning. □

With this we conclude our saturated MMU model and its refinement with the nondeterministic MMU.

6.2.2 The Most Abstract MMU Model

We now present the last and most abstract model of our refinement chain. Similar to the abstract model of the single-stage TLB (Sect. 5.2.3), we abstract the two-stage TLB completely and soundly model its functionality using the record of inconsistent virtual addresses. As expected and required, this abstract model is conservative in detecting inconsistencies for the caching of each level of the page table walks. In this MMU, we do not extend the state with `TLB × pdc`, rather the two-stage TLB is modeled using these three components:

```
incon_set :: vaddr set
global_set :: vaddr set
snapshot :: asid ⇒ vaddr set × (vaddr ⇒ pt_walk_typ)
```

For motivation of `snapshot`, please refer to the starting paragraphs of the most abstract model of the single-stage TLB (Page 83). The snapshot now stores the `pt_walk_typ` instead of `asid tlb_entry option`. We will go into more detail on this later in this section. Figure 6.7 gives an overview of the ARMv7-A MMU with our most abstract TLB interface. The read/write dependencies of the model are similar to the abstract model for the single-stage TLB, and we summarise it in Table 6.1 for the reader's convenience.

As in the single-stage case, the address translation and memory operations which constitute the majority of program instructions use only the `incon_set` as their TLB interface. The `snapshot` is used only to detect the inconsistent addresses

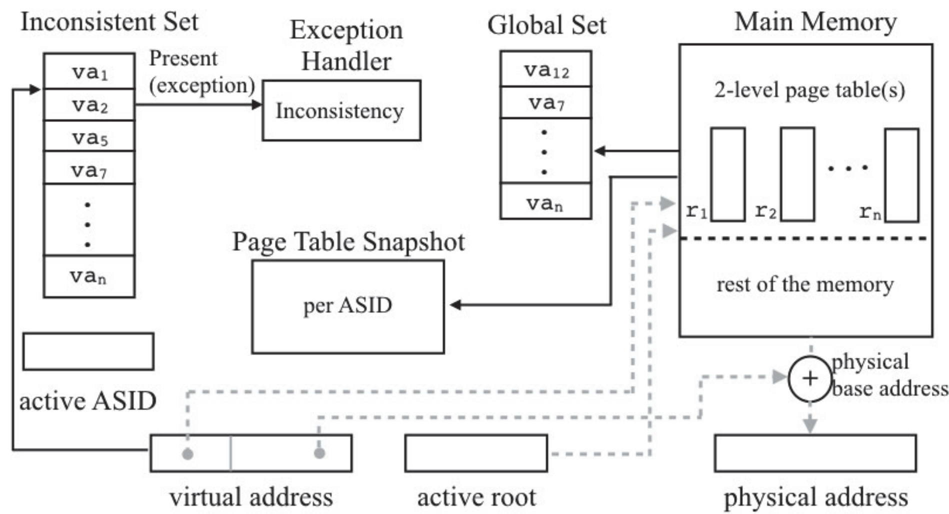

Figure 6.7: ARMv7-style Memory Management Unit with Abstract MMU

while switching the ASID register. The ASID register is only updated on a context switch between processes, and we will see later in this section that `snapshot` is used there for the page table comparison: there is no actual TLB lookup involved.

We now explain our abstract TLB model. The `incon_set` and the `global_set` are identical to the previous abstract model: the `incon_set` stores inconsistent virtual addresses that are either global or under the active ASID, the `global_set` consists of all the globally mapped virtual addresses irrespective of their consistency. The `snapshot` now holds the state of two-stage page table walks `pt_walk_typ` for all inactive ASIDs. This implies that this model is more conservative in detecting inconsistencies than the single-stage TLB, and this is exactly what we should expect. The TLB-`snapshot` of the page table state modulo the inconsistent virtual addresses for every ASID is modeled as the pair type of `vaddr set` and the map type `vaddr ⇒ pt_walk_typ`. The `vaddr set` of a snapshot an ASID `a` contains the inconsistent virtual addresses under the ASID `a`, and the page table state holds the mapped non-global entries. A `None` in the page table state represents either global or unmapped virtual addresses. As expected, the flush operations under ASIDs are able to alter the content of the `snapshot`.

We now explain the operations of the most abstract MMU model on the two-stage TLB and also provide the refinement theorems between the saturated and the most abstract MMU models.

Memory Operations: For translating a virtual address, we simply check its consistency for the two-stage TLB using the `incon_set` and subsequently carry out the translation from the page table itself.

```
mmu_translate_set va ≡ do {
  (mem, ttbr0, asid) ← read_state (MEM, TTBR0, ASID);
```

Operation	Utilise from Abs. MMU	Update from Abs. MMU
Address Translation	`incon_set`	Nothing
Memory Read	`incon_set`	Nothing
Memory Write	`incon_set`	`incon_set`, `global_set`
Root Update	Nothing	`incon_set`, `global_set`
ASID Update	`incon_set`, `global_set`, `snapshot`	`incon_set`, `snapshot`
Flush Operations	Nothing	`incon_set`, `global_set`, `snapshot`

Table 6.1: Read/Write Dependencies for Memory and MMU Operations

```
  incon_set ← read_state_iset incon_set;
  if va ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if fault entry then raise PAGE_FAULT
        else return (va_to_pa va (the entry))
}
```

Note that for address translation, we have used the function `pt_walk` which gives us end-to-end address translation. This is another highlight of our most abstract model: since we carry out the address translation from the actual page table itself, we are not required to do the conservative two-stage page table walk. From the abstract two-stage TLB model, we have utilised only `incon_set` for checking the consistency of the virtual address `va` as it is being resolved under the active ASID. The reader is encouraged to compare the complexity of the nondeterministic address translation function of Sect. 6.1.2 to this simple yet sound address translation function.

We now proceed to the memory write operation. For its instantiated function `mmu_write_set`, we must figure out which new addresses might have become inconsistent for the two-stage TLB. For keeping track of such inconsistencies stemming from two different page table walks, we introduce a *less or equal* relation between `pt_walk_typ` walks as:

```
walk ≼ walk' ≡
walk = Fault ∨
walk = walk' ∨
(∃pde. walk = Partial_Walk pde ∧
       (∃tlbentry. walk' = Full_Walk tlbentry pde))
```

In the above relation, the first two disjuncts imply that a `Fault` walk is always smaller then a `Partial_Walk` and a `Full_Walk`. The third disjunct states that a `walk` is *less* than the another `walk'` if `walk` is a `Partial_Walk` with a `pde`, and `walk'` is a `Full_Walk` with the same `pde` and a `tlbentry`.

Using the *less or equal* relation for page table walks, we define the `ptable_comp` function that compares two page tables and returns a set of all the virtual addresses that are two-stage TLB-*inconsistent*:

```
ptable_comp walk walk' ≡ {va | ¬ walk va ⪯ walk' va}
```

This comparison gives us the set of unmapped and remapped virtual addresses, as well as virtual addresses that result in the same end-to-end translation from the both `walk` and `walk'` but have different page directory entries. The later category covers an important aspect of the two stages of page table caching in the concrete MMU model of Sect. 6.1.

The memory write instantiation `mmu_write_set` for this abstracted model is:

```
mmu_write_set (val, va, sz) = do {
  (m, rt, a, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  paddr ← mmu_translate_set va;
  when_no_exc do {
    mem_write (val, paddr, sz);
    m' ← read_state MEM;
    let incon_set_new =
      ptable_comp (ptd_walk a m rt) (ptd_walk a m' rt);
    let global_set_new = global_varange a m' rt;
    update_incon_set (incon_set ∪ incon_set_new);
    update_global_set (global_set ∪ global_set_new)
  }
}
```

Where

```
global_varange a m rt ≡
⋃ e∈global_entries (ran (pt_walk a m rt)) range_of e
```

The first step in the memory write is to resolve the given virtual address. On the successful translation we simply write to the physical memory. To figure out the potential TLB-inconsistencies as the result of this memory write, we compare the results of the two-stage page table walks (`ptd_walk`) before and after the write operation using the function `ptable_comp`. We enumerate all cases of this page table comparison in Table 6.2 for the reader's convenience. The function `update_incon_set` in the function `mmu_write_set` updates the `incon_set` of the state with the given argument.

We also reload the `global_set` with the address range of new global mappings after the memory write. This helps us to soundly model the inconsistencies while switching ASID as we will see later in this section. The function `global_varange` simply computes the `range_of` the `global_entries` (the TLB entries with `None`

before va	after va	Check
Partial_Walk pde	Fault	True
Partial_Walk pde	Partial_Walk pde’	pde ≠ pde’
Partial_Walk pde	Full_Walk e pde’	pde ≠ pde’
Full_Walk e pde	Fault	True
Full_Walk e pde	Full_Walk e pde’	pde ≠ pde’
Full_Walk e pde	Full_Walk e’ pde	e ≠ e’
Full_Walk e pde	Partial_Walk pde’	True
Full_Walk e pde	Full_Walk e’ pde’	pde ≠ pde’ ; e ≠ e’

where `before = ptd_walk a m rt` and `after = ptd_walk a m’ rt`

Table 6.2: When does a Page Table Walk Change Produce an Inconsistency?

ASID tag) from the page table starting at the location `rt` in the memory `m`. Note that for global entries, we use the end-to-end page table walk function (`pt_walk`) instead of the two-stage `ptd_walk`, because global entries are inherently determined by the final address translation. This will also make reasoning simpler later in the logic. The function `update_global_set` in the function `mmu_write_set` updates the `global_set` of the state with the given argument.

For `mmu_read_set`, the definition is similar to the base-level model, we only use the new `mmu_translate_set` instance.

```
mmu_read_set (va, sz) = do {
  pa ← mmu_translate_set va;
  mem_read (pa, sz)
}
```

We wrap up the evaluation of functions `mmu_translate_set`, `mmu_write_set` and `mmu_read_set` in the function `mem_op`, and call this function `mem_op_set`.

MMU Operations: We now explain MMU operations for our most abstact MMU model.

As in the most abstract model of the single-stage TLB, updating the page table root register has an effect on the most abstract two-stage TLB similar to changing the state of the current page table through the memory write operation. In the instantiation `update_TTBR0_set`, we use the page table comparison `ptable_comp` function with walks from two page tables, in order to compute the resultant inconsistent virtual addresses. We also reload the `global_set` with the updated global virtual addresses.

```
update_TTBR0_set r = do {
  (mem, ttbr0, asid, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  update_state (λs. s(|TTBR0 := r|));
```

```
  let incon_set_new =
    ptable_comp (ptd_walk asid mem ttbr0) (ptd_walk asid mem r);
  let global_set_new = global_varange asid mem r;
  update_incon_set (incon_set ∪ incon_set_new);
  update_global_set (global_set ∪ global_set_new)
}
```

We now explain the instruction for updating the ASID register. Until now, we have only used the `incon_set` from our most abstract model, while only reloading the `global_set`. Now, as in the single-stage TLB, we will manipulate all three components of the most abstract TLB model. The instantiation `update_ASID_set` for the two-stage TLB is almost identical to the respective instantiation for a single-stage abstract TLB model that we presented in the previous chapter (Sect. 5.2.3, Page 87). This similarity is another salient feature of our refinement framework.

The updateASID instruction is instantiated as:

```
update_ASID_set a = do {
  (mem, ttbr0, asid, incon_set, global_set) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set);
  snapshot ← read_state_iset snapshot;
  let new_snp = snapshot
    (asid := (incon_set, ptd_walk asid mem ttbr0));
  update_snapshot new_snp;
  update_state (λs. s(|ASID := a|));
  let global_incon = incon_set ∩ global_set;
     incon_set_snap = fst (new_snp a);
     ptcomp_snap =
       ptable_comp (snd (new_snp a)) (ptd_walk a mem ttbr0)
  in update_incon_set (global_incon ∪ incon_set_snap ∪ ptcomp_snap)
}
```

We proceed as in the single-stage case, only with `ptd_walk` instead of `pt_walk`: we first store the `incon_set` and the two-level page table state (`ptd_walk`) of the active ASID to the `snapshot`. The function `update_snapshot` updates the `snapshot` of the state with the given argument. Next we update the ASID register to the ASID `a`. For finding the `incon_set` for the ASID `a`, we

- mask the globally inconsistent virtual addresses by intersecting the `incon_set` and the `global_set`,
- retrieve the stored inconsistent virtual addresses from the `snapshot` of ASID `a`, and finally
- compare the stored two-level page table state and the active page table using the `ptable_comp` function.

We simply update the `incon_set` with these set of inconsistent virtual addresses using the `update_incon_set` function and the execution can continue with the new ASID under its `incon_set`.

We now proceed to explaining the flush operations for our most abstract model. Depending on the nature of the flush instruction, the flush operations remove the relevant virtual addresses from the `incon_set` and the `global_set`, and unmap them from `snapshot`. To unmap an address from the `snapshot` means to make its respective `snapshot` the `Fault`, since flush instructions operate both on the PDC and the TLB. We trivially saturate the `global_set` with the mapped global addresses of the current page table after the flush instructions for virtual addresses; this helps us proving the refinement later. The instantiation `flush_set` for flushing either the TLB or a range of virtual addresses (irrespective of ASIDs) is defined as:

```
flush_set f = do {
  (m, rt, a, iset, gset, snp) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set, snapshot);
  case f of FlushTLB ⇒ do {
               update_incon_set ∅;
               update_global_set (global_varange a m rt);
               update_snapshot (λa. (∅, λv. Fault))
              }
  | Flushvarange vset ⇒ do {
        update_incon_set (iset - vset);
        update_global_set (gset - vset ∪ global_varange a m rt);
        update_snapshot
         (λa. (fst (snp a) - vset,
               λv. if v ∈ vset then Fault else snd (snp a) v))
      }
}
```

`FlushTLB` empties the `incon_set` and inconsistent addresses of `snapshot` for all ASIDs, unmaps the stored page table state to make it `Fault`, and reloads the `global_set` with the consistent globally mapped virtual addresses of the current state. Similary, `Flushvarange` removes the given set of virtual addresses from the `incon_set` and from the inconsistent addresses of `snapshot` for all ASIDs, unmaps them from the stored page table state of all ASIDs, and removes them from the `global_set` before making the `global_set` saturated with the globally mapped virtual addresses of the current state.

The flush operation for the two-stage TLB is instantiated as:

```
flush_with_ASID_set f = do {
  (m, rt, asid, iset, gset, snp) ←
    read_state (MEM, TTBR0, ASID, incon_set, global_set, snapshot);
  case f of
  FlushASID a ⇒
    if a = asid then update_incon_set (iset ∩ gset)
    else update_snapshot (snp(a := (∅, λv. Fault)))
  | FlushASIDvarange a vset ⇒
```

```
      if a = asid then update_incon_set (iset - (vset - gset))
      else let iset = fst (snp a); pt = snd (snp a)
          in update_snapshot
              (λa'. if a' = a
                    then (iset - vset,
                          λv. if v ∈ vset then Fault else pt v)
                    else snp a')
}
```

For the active ASID, FlushASID simply removes the ASID-specific addresses from the incon_set. While for an inactive ASID as its argument, the FlushASID unmaps the ASID's snapshot. The FlushASIDvarange repeats the same process for the given set of virtual addresses under an ASID.

We instantiate the evaluation of functions update_TTBR0_set, update_ASID_set, flush_set and flush_with_ASID_set for the function mmu_op, and call this function mmu_op_set.

Refinement: We now present the refinement theorems between the saturated and the most abstract models.

The refinement relation should provide a lookup order between the abstract TLB model (incon_set, global_set and snapshot) and the saturated TLB and PDC. Such a refinement relation is:

```
tlb_rel_set s t ≡
let tlb_sat = fst (TLB_PDC s); pdc_sat = snd (TLB_PDC s)
in truncate s = truncate t ∧
   ASID s = ASID t ∧
   saturated s ∧
   incon_addrs s ⊆ incon_set t ∧
   global_range s ⊆ global_set t ∧
   (∀a v. a ≠ ASID s ⟶
          tlb_lookup (non_global_entries tlb_sat) a v
          ≤ tlb_lookup_from (snapshot t) a v ∧
          pdc_lookup (non_global_entries pdc_sat) a v
          ≤ pdc_lookup_from (snapshot t) a v)
```

Where the function incon_addrs constructs the set of inconsistent addresses under the *active* ASID in the saturated TLB and PDC of state s:

```
incon_addrs s ≡ tlb_incon_addrs s ∪ pdc_incon_addrs s
```

Where

```
tlb_incon_addrs s ≡
let tlb = fst (TLB_PDC s); a = ASID s; m = MEM s; rt = TTBR0 s
in {va | tlb_lookup tlb a va = Incon} ∪
```

```
  {va | ∃e. tlb_lookup tlb a va = Hit e ∧ fault (pt_walk a m rt va)}
```

```
pdc_incon_addrs s ≡
let pdc = snd (TLB_PDC s); a = ASID s; m = MEM s; rt = TTBR0 s
in {va | pdc_lookup pdc a va = Incon} ∪
  {va | ∃e. pdc_lookup pdc a va = Hit e ∧ fault (pdc_walk a m rt va)}
```

This subset relation between the `incon_addrs` of state `s` and the `incon_set` of state `t` is analogous to the subset assumption of our earlier refinement between the saturated and the non-deterministic MMU model. This subset relation provides us with the TLB's and the PDC's lookup order, hence guarantees about safe execution. We impose a similar lookup order for the global addresses cached in the saturated TLB and PDC of state `s` and between the `global_set` of state `t`. The function `global_range` (provided below) computes the global addresses of the saturated TLB and PDC, and we assert that these addresses are in the subset relation with the `global_set` of state `t`. The `global_range` for the state `s` is computed as:

```
global_range s ≡ tlb_global_range s ∪ pdc_global_range s
```

Where

```
  tlb_global_range s ≡
  let tlb = fst (TLB_PDC s) in ⋃e∈global_entries tlb range_of e
```

```
  pdc_global_range s ≡
  let pdc = snd (TLB_PDC s) in ⋃e∈global_entries pdc range_of e
```

The last conjunct of the refinement relation `tlb_rel_set` provides us with a similar order for the *inactive* ASIDs. The TLB and the PDC of sate `s` are not saturated for these ASIDs; therefore we assert that the stored `snapshot` of the abstract state `t` covers all possible ASID-specific executions (represented by the `tlb_lookup` on `non_global_entries`) of the TLB and the PDC of state `s`. Formally:

```
tlb_lookup_from snp a va ≡
let iset = fst (snp a); pt = snd (snp a)
in if va ∈ iset then Incon
  else case pt va of
      Full_Walk te pe ⇒ if asid_of te = None then Miss else Hit te
      | _ ⇒ Miss
```

```
pdc_lookup_from snp a va ≡
let iset = fst (snp a); pt = snd (snp a)
in if va ∈ iset then Incon
```

```
else case pt va of Fault ⇒ Miss
    | Partial_Walk pe ⇒ if asid_of pe = None then Miss else Hit pe
    | Full_Walk te pe ⇒
        if asid_of te = None ∧ asid_of pe = None then Miss
        else Hit pe
```

The function `tlb_lookup_from` estimates a TLB lookup for the given ASID `a` and virtual address `va` from the snaphot `snp`: the resultant lookup is `Incon` if the address `va` is in the inconsistent set of the snaphot `snp`, otherwise `Faults` and global addresses are encoded to `Miss`, and an the ASID-specific entries to `Hit`. Similarly, the function `pdc_lookup_from` estimates PDC lookup for the given ASID `a` and virtual address `va` from the snaphot `snp`: the resultant lookup is `Incon` if the address `va` is in the inconsistent set of the snaphot `snp`, otherwise `Faults` and global addresses are encoded to `Miss`, global `Partial_Walks` to `Miss`, non-global `Partial_Walks` to `Hit`, global `Full_Walks` to `Miss`, and non-global `Full_Walks` to `Hit`.

We now present the refinement theorems.

Theorem 26. *The saturated and abstract memory operations preserve the refinement relation given the consistency of the most abstract TLB for the virtual address.*

$$\frac{\texttt{mem_op_sat f s = (res, s')} \quad \texttt{mem_op_set f t = (res', t')} \quad \texttt{consistent_set f t} \quad \texttt{tlb_rel_set s t}}{\texttt{res' = res} \land \texttt{tlb_rel_set s' t'}}$$

Where `consistent_set` *ensures that the memory operation is for a consistent virtual address i.e. the virtual address is not an element of the* `incon_set` *of state* `t`*.*

Proof. We first explain how the refinement works for address translation. According to the refinement relation, the `incon_set` of state `t` tracks the inconsistent virtual addresses for the active ASID in the saturated TLB of state `s`. We are therefore in the `else` branch of `mmu_translate_set`, and in either the `Hit` or the `Miss` branch of `mmu_translate_sat`. In both these cases, the results must agree because `saturated` and `tlb_rel_set` say that the `Hit` and `Miss` results represent precisely the walks we perform in `mmu_translate_set`.

As in the single-stage case, memory reads preserve the refinement relation after the address translation in a straight forward way. Memory writes are again similar to the single-stage case, but we now need to observe that the new definition of `ptable_comp` with two-stage walks correctly captures all possibilities for creating inconsistencies. □

Theorem 27. *The saturated and abstract MMU operations preserve the refinement relation.*

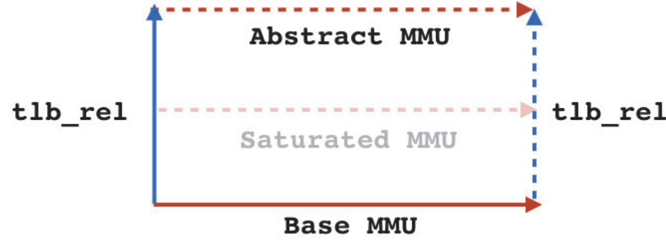

Figure 6.8: Refinement between Nondeterministic and Abstract MMU

```
                 mmu_op_sat f s = ((), s')
 mmu_op_set f t = ((), t')      tlb_rel_set s t
 ───────────────────────────────────────────────
                   tlb_rel_set s' t'
```

Proof. As before, the proof strategy for the `update_TTBR0` instantiations is similar to that of memory writes, as the `ptable_comp` comparison is inherently the same. The proof for flush instructions includes the set reasoning about preserving the refinement relation after the respective `incon_set`, `global_set` and `snapshot` updates.

The refinement proof for the `update_ASID` instruction follows the same structure as the single-stage case. Overall we are required to establish that the `incon_set` correctly models the inconsistencies of the saturated TLB and PDC after updating the ASID, the `global_set` holds the subset relation and the `snapshot` provides the lookup order for the all the inactive ASIDs. The main observation that makes the proof possible is that the saturated TLB and PDC can be partitioned into its global and non-global entries, and then we can reason about the disjoint lookup order for the non-global entries. The global entries are reasoned about by the subset relation of the global entries and `global_set`. The `snapshot` order is preserved by proving that the `ptable_comp` function correctly detects the inconsistencies. □

With this we conclude our most abstract MMU model and its refinement with the saturated MMU.

6.2.3 Joining the Refinement Levels

In this section, we join the refinement levels of Figure 6.5 to show that our most abstract model is sound with respect to the base model. The resultant refinement is between the most abstract and the base model through the saturated MMU model as shown in Figure 6.8. The refinement relation `tlb_rel` is:

```
tlb_rel r t ≡ ∃s. tlb_rel_sat r s ∧ tlb_rel_set s t
```

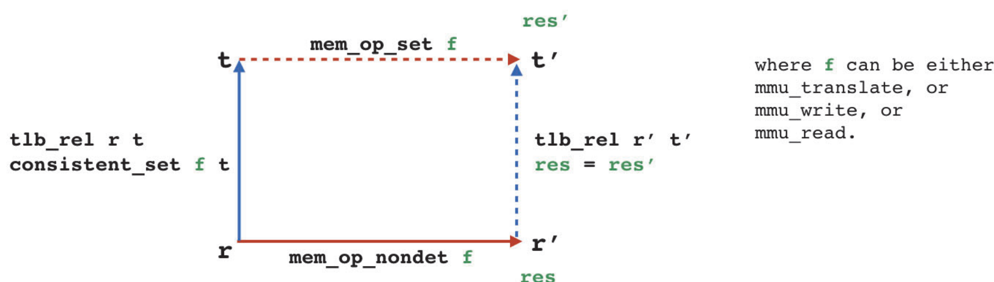

Figure 6.9: Refinement between Nondeterministic and Abstract Memory Operations

Where the state `r` has the nondeterministic two-stage TLB, the state `s` has the saturated two-stage TLB and the state `t` has the most abstract TLB.

The functions `tlb_rel_sat` and `tlb_rel_set` are the refinement relations provided in Sect. 6.2.1 and Sect. 6.2.2 respectively. We then have two refinement theorems, presented below and also shown in Figure 6.9 and Figure 6.10.

Theorem 28. *Refinement between nondeterministic and abstract memory operations.*

$$\frac{\texttt{mem_op_nondet f r = (res, r')} \quad \texttt{mem_op_set f t = (res', t')} \quad \texttt{consistent_set f t} \quad \texttt{tlb_rel r t}}{\texttt{res = res'} \wedge \texttt{tlb_rel r' t'}}$$

where `consistent_set` *ensures that the abstract state* `t` *is TLB-consistent: the given virtual address is not an element of the* `incon_set` *of state* `t`.

Proof. By case analysis on the function `f` and using the respective refinement theorems of Sect. 6.2.1 and Sect. 6.2.2, and proving that consistency of a virtual address on the most abstract model implies its consistency on the base model. □

Theorem 29. *Refinement between nondeterministic and abstract MMU maintenance operations.*

$$\frac{\texttt{mmu_op_nondet f r = ((), r')} \quad \texttt{mmu_op_set f t = ((), t')} \quad \texttt{tlb_rel r t}}{\texttt{tlb_rel r' t'}}$$

Proof. By case analysis on the function `f` and using the respective refinement theorems of Sect. 6.2.1 and Sect. 6.2.2. □

With this we conclude presenting the refinement stack.

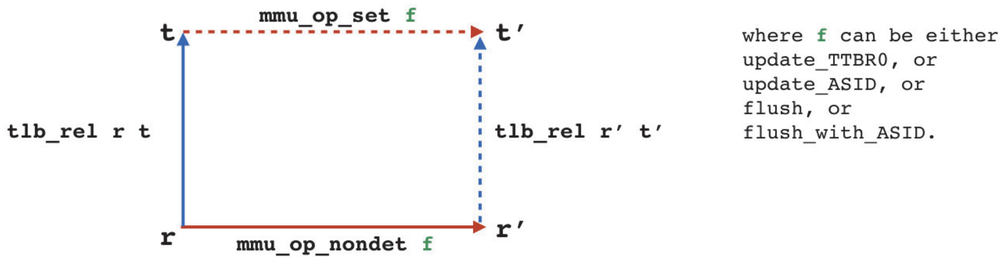

Figure 6.10: Refinement between Nondeterministic and Abstract MMU Operations

6.3 Summary and Remarks

In this chapter, we have built on the MMU model of Chapter 5 to introduce a separate page directory cache (PDC) resulting in a two-stage TLB. We have identified the additional reasoning complexities this two-stage TLB entails and have developed a refinement stack that abstracts away the hardware details and that reaches at an abstract model of ARMv7-A MMU that is easier to reason about.

The abstract model has three components for modeling the two-stage TLB: a set of inconsistent virtual addresses for the active ASID-specific and globally mapped virtual addresses, a set of globally mapped virtual addresses and a snapshot of the two-stage page table state modulo the inconsistent addresses for all ASIDs. The main message of the refinement chain presented in this chapter is that any logic taking this abstract model as its memory interface would avoid the reasoning complexities of the actual hardware state of the TLB. This model is sound to reason about for the implementations of the ARMv7-A architecture that caches partial page table walks under ASIDs and global tags. The model and refinement chain of this chapter is available online in the form of Isabelle theories (Syeda, 2019).

In the next chapter, we use the most abstract model of this chapter as the memory model of a program logic for reasoning about programs in the presence of cached address translation.

CHAPTER

SEVEN

Program Logic in the Presence of Cached Address Translation

Operating system (OS) kernels achieve isolation between user-level processes using multi-level page tables and translation lookaside buffers (TLBs). Controlling the TLB correctly is a fundamental security property. We present a logic for reasoning about low-level programs in the presence of TLB address translation. For its memory model, we use the sound abstraction of the ARM7-A MMU presented in the previous chapter. In the next chapter, we apply the rules of this logic to extract invariants and necessary conditions for correct program execution at the TLB level.

This chapter is organised as: we define the syntax and semantics of a heap based language with the instructions necessary for TLB management, we then present the Hoare logic rules on top of the operational semantics. We also provide simplification rules for memory write to further facilitate program reasoning in the presence of TLB effects.

This chapter is based on the published work (Syeda and Klein, 2018) and the submitted work (Syeda and Klein, 2019).

7.1 Program Logic

We present a program logic in Isabelle/HOL (Nipkow et al., 2002) for verifying programs in the presence of an ARMv7-style memory management unit, consisting of multi-level page tables and a two-stage translation lookaside buffer for caching page table walks. This logic builds on the work of Chapter 6, a machine model with a sound abstraction of the ARMv7-style two-stage TLB with ASIDs and global tags. While program logics for reasoning in the presence of address translation exist (Kolanski and Klein, 2009), reasoning in the presence of a TLB has so far remained hard, and is left as an assumption in all large-scale operating system (OS) kernel verification projects such as seL4 (Klein et al., 2014) and CertiKOS (Gu et al., 2011).

The OS kernel manages page table structures, e.g. by adding, removing, or changing mappings, by keeping a page table structure per user process, and by maintaining invariants on them (details then in the next chapter). Since the TLB caches address translation, each of these operations may leave the TLB out of date w.r.t. the page table in memory, and the OS kernel must flush the TLB before that lack of synchronisation can affect program execution. Since flushing the TLB is expensive, OS kernel designers work hard to delay and minimise flushes and to make them as specific as possible, e.g. using ASIDs or global entries. If this management is done correctly, the TLB has no effect other than speeding up execution. If it is done incorrectly, machine execution will diverge from the semantics usual program logics assume, e.g. wrong memory contents will be read/written, or unexpected memory access faults might occur. The logic we demonstrate here will enable proof engineers to reason about such effects.

```
datatype aexp =
    Const val
  | UnOp (val ⇒ val) aexp
  | BinOp (val ⇒ val ⇒ val) aexp aexp
  | HeapLookup aexp

datatype bexp =
    BConst bool
  | BComp (val ⇒ val ⇒ bool) aexp aexp
  | BBinOp (bool ⇒ bool ⇒ bool) bexp bexp
  | BNot bexp

datatype mode_t = Kernel | User

datatype com =
    SKIP
  | aexp := aexp
  | com ;; com
  | IF bexp THEN com ELSE com
  | WHILE bexp DO com
  | Flush flush_type
  | UpdateRoot aexp
  | UpdateASID asid
  | SetMode mode_t

type_synonym asid = 8 word
type_synonym val = 32 word

datatype flush_type = flushTLB      | flushVR (val set)
                    | flushASID asid | flushASIDVR asid (val set)
```

Figure 7.1: Syntax of the Heap based WHILE Language.

We demonstrate our logic by implementing it for a small deeply-embedded imperative language, which contains memory operations and TLB maintenance instructions, as well as a distinction between privileged kernel mode and unprivileged user mode. We then define semantics of the language with the abstract MMU model of the previous chapter as its memory model. Based on the semantics, we derive Hoare logic rules and prove their soundness. The logic is generic and can easily be adapted to, for instance, the shallow embedding the seL4 specifications use, or the more deeply embedded C semantics of the same project. It should also transfer readily to other settings such as the lower levels of CertiKOS in Coq.

We now present the syntax of our heap-based language.

7.1.1 Syntax

We define the syntax of a simple Turing-complete heap language with TLB management primitives. Figure 7.1 shows the Isabelle data types for the abstract syntax of the language.

Control structures are the standard `SKIP`, `IF`, `WHILE` and assignment, where assignment expects the left-hand side to evaluate to a heap address. In addition, we have specific privileged commands for flushing the TLB, updating the current page table root, the current ASID, and the processor mode. `Flush` operation has a number of variants: invalidate all entries, invalidate by virtual address or by virtual address/ASID pair, and invalidate an entire ASID.

For simplicity, there are no local variables in this language, only the global heap. We identify values and pointers and admit arbitrary HOL functions for compari-

son, binary, and unary arithmetic expressions.

We now present the program state and the memory model for defining the semantics of our language.

7.1.2 Program State and Memory Model

In the previous chapter we have developed an abstract and sound MMU model that keeps track of the TLB-inconsistent addresses and uses direct page table access for address translation. We use the same model here, but there is a break in logic: the model of the previous chapter is at the ISA level, the program logic we present here is for a higher-level language with explicit memory access, intended for languages such as C.

In the absence of a formal compiler correctness statement, there is one main difference to consider and justify in making this jump: the high-level language makes fewer memory accesses visible than the low-level machine model. In particular, a compiler from a higher-level language to the previous binary-level machine model will implement a stack for local variables, will have a memory area for global variables, and a memory area for the code itself. These memory accesses are under address translation and might be relevant for TLB reasoning. We will initially develop a model that simply ignores this issue and then come back to it in Sect. 7.2. We will see that for kernel-level code, we have to assume that these memory areas (code, stack, globals) are statically known and that the compiler will not generate additional memory accesses outside the program heap and these memory areas. This is a reasonable assumption — otherwise kernel code could never be sure that privileged memory areas such as memory-mapped devices are not randomly overwritten by compiler-generated accesses. We will then have to prove that we never remove or change active mappings for these areas (adding new mappings for e.g. the stack would be fine). For user-level code, we will see that the issue becomes irrelevant.

The state of our language model has the following components (Figure 7.2 gives an overview):

- the heap (physical memory),
- the set of inconsistent virtual addresses (global and under the active ASID),
- the set of globally mapped virtual addresses,
- the active page table root,
- the active ASID,
- page table snapshot which represents the last known ASID-specific page table state modulo inconsistencies for all inactive ASIDs, and
- the processor mode.

The first of these is for traditional heap manipulation, the rest for keeping track

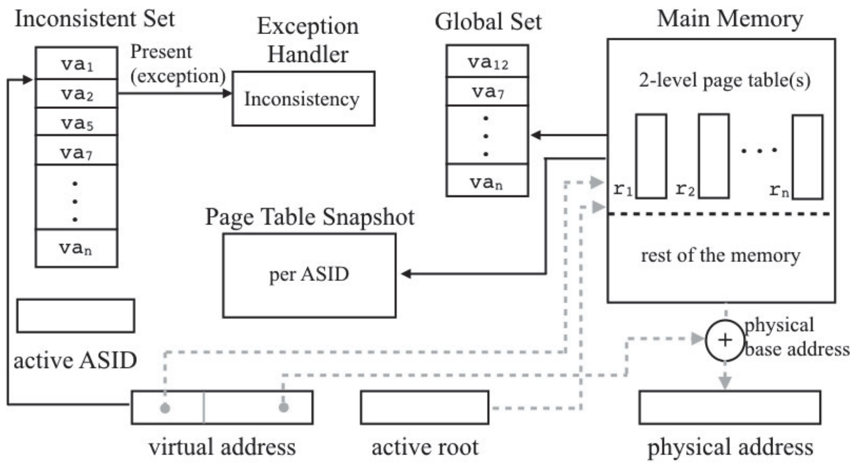

Figure 7.2: Abstracted TLB Memory Model

of the abstract TLB interface. This state model is the same as the TLB-relevant machine state of the previous chapter on the ISA level. We use the types `vaddr` and `paddr` for virtual and physical addresses from Kolanski's page table interface (Kolanski and Klein, 2009). To simplify the language, we make it operate exclusively on type `val = 32 word`. The program state is modeled as the record type `p_state` with fields:

- `heap :: paddr ⇀ val`,
- `iset :: iset`,
- `gset :: gset`,
- `pt_snpshot :: ptable_snapshot`,
- `root :: paddr`,
- `asid :: asid`, and
- `mode :: mode_t`.

Where

type_synonym `iset = vaddr set`
type_synonym `gset = vaddr set`
type_synonym `ptable_snapshot = asid ⇒ vaddr set × (vaddr ⇒ pt_walk_typ)`

The type `iset` is the set of TLB-inconsistent virtual addresses that are either mapped globally or under the active ASID, and the type `gset` is the set of globally mapped virtual addresses. `ptable_snapshot` is the same type for the page table snapshot as introduced in the abstract MMU model of the previous chapter (Sect. 6.2.2). We summarise it here; the motivation for page table snapshot can be found in the abstract memory model at Page 83. For each ASID, we keep a snapshot of the ASID-specific current page table state when that ASID was last active modulo all addresses that were inconsistent at that time. The `ptable_snapshot` is the map from `asid` to the pair of the inconsistent `vaddr set` and the page table

state `vaddr ⇒ pt_walk_typ`. The `pt_walk_typ` is the same type as introduced for the two-level page table walk in the previous chapter (Sect. 6.1.1).

We use the type `mode_t` to decode access control information in the page table, that is, some mappings might be accessible in kernel mode only and lead to a page fault otherwise.

We now proceed to the semantic operations that are used for defining the big-step semantics of our language.

7.1.3 Semantic Operations

We interpret the values `val` of the language as virtual addresses, which means memory read and write first undergo address translation. To decode page tables, we reuse Kolanski's existing ARM page table formalisation (Kolanski and Klein, 2009), extended with this access control behaviour for the machine mode as mentioned above. Our interface to this formalisation is the function `pt_lookup`, which takes a heap, a page table root, and the current mode, and yields a partial function from virtual address to physical address. The function `pt_lookup` uses the same constituent functions as that of the `pt_walk` function of Sect. 4.3, but it returns a physical address instead of a TLB entry. Using the function `pt_lookup`, we can formalise address translation, read, and write under a TLB.

Adding a TLB to address translation only adds a check that the virtual address is not part of the `iset`:

```
phy_ad :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇀ paddr
phy_ad IS hp rt m va ≡ if va ∉ IS then pt_lookup hp rt m va else None
```

The memory read and write functions are then simply:

```
read :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇀ val
read IS hp rt m va ≡ phy_ad IS hp rt m va ▷ load_value hp
```

```
write :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇒ val ⇀ heap
write IS hp rt m va v ≡
case phy_ad IS hp rt m va of None ⇒ None | ⌊y⌋ ⇒ ⌊hp(y ↦ v)⌋
```

where `x ▷ g ≡ case x of None ⇒ None | ⌊y⌋ ⇒ g y`. Both functions first perform address translation, then access the physical heap. The `read` operation returns `None` when the translation failed, and the `write` returns a new heap if successful and `None` otherwise.

The effect of a write operation extends further than the heap. If the operation has modified the active page table, we may have to add new inconsistent addresses

to the TLB `iset`, and new globally mapped addresses to the `gset`. For the `iset` reload, we compare the page table before and after:

```
ptable_comp wlk wlk' ≡ {va | ¬ wlk va ≼ wlk' va}
incon_comp a hp hp' rt rt' =
ptable_comp (ptd_walk a hp rt) (ptd_walk a hp' rt')
```

where `a` is the current ASID. The function `ptable_comp` compares the walks of two page tables for inconsistencies and is same as the page table comparison function defined in the previous chapter. More details about *less or equal* relation between walks are in Sect. 6.2.2 on page 114, with enumerated cases in Table 6.2. For the function `incon_comp`, we compare the results of page table walks in a heap `hp` from a root `rt` with walks in a different, updated heap `hp'` and potentially different root `rt'`. For heap writes, the root will be the same, and for root updates, the heaps will be the same. Note that a single heap write can affect multiple mappings at once, for instance when it changes the pointer to an entire page table level. It is the effect of this comparison that OS engineers reason about informally when they compute which addresses need to be flushed from the TLB. We will show examples in the case study (Chapter 8).

As in the ISA-level model, for the global set reload after the memory write, we add the resultant virtual address range of the global entries of mapped page table after the write as follows.

```
global_vaddrs a hp rt ≡
let mapped_ptable = ran (pt_walk a hp rt);
   global_ptable = global_entries mapped_ptable
in ⋃x∈global_ptable range_of x
```

The effect of a write is then

```
heap_iset_gset_upd_s (pp ↦ v) ≡
let hp = heap s; hp' = hp(pp ↦ v); rt = root s; a = asid s
in s⦇heap := hp', iset := iset s ∪ incon_comp a hp hp' rt rt,
     gset := gset s ∪ global_vaddrs a hp' rt⦈
```

and the effect of a page table root update is

```
root_iset_gset_upd_s rt' ≡
let rt = root s; hp = heap s; a = asid s
in s⦇root := rt', iset := iset s ∪ incon_comp a hp hp rt rt',
    gset := gset s ∪ global_vaddrs a hp rt'⦈
```

For changing the current ASID, we make use of the page table snapshots to determine which addresses have become inconsistent since that ASID was last active.

There are two steps involved: first we store the snapshot for the ASID we are switching away from, and second we compute the inconsistent addresses for the new ASID from its snapshot, `iset` and the active page table (more details in the previous chapter at Page 117). The snapshot update is formalised as:

```
new_snp s ≡
let a = asid s; hp = heap s; rt = root s
in (pt_snpshot s)(a := (iset s, λv. ptd_walk a hp rt v))
```

Taking a snapshot is taking the `iset` and `ptd_walks` in the current state, marking everything in the `iset` to the inconsistent addresses of the snapshot, and all unmapped entries as `Fault`, and then storing that function under the current ASID in `new_snp`.

Determining the `iset` for the new ASID `a` compares the entries in the snapshot for the ASID `a` with the current `ptd_walk`. We use `new_snp s` instead of `pt_snpshot s`, because the ASID `a` could also be the current ASID. We also preserve the global inconsistencies in the process, intersecting `iset` with the `gset`.

```
snp_incon a s ≡
let snp = new_snp s; iset = iset s; gset = gset s; hp = heap s;
   rt = root s; snp_incon = fst (snp a); glb_incon = iset ∩ gset;
   pt_incon = ptable_comp (snd (snp a)) (ptd_walk a hp rt)
in snp_incon ∪ glb_incon ∪ pt_incon
```

The `UpdateASID` command then executes as:

```
asid_iset_snp_upd_s a ≡
s(|asid := a, iset := snp_incon a s, pt_snpshot := new_snp s|)
```

The final set of semantic effects are flush operations. The functions

`flush_iset :: flush_type ⇒ iset ⇒ asid ⇒ iset` and
`flush_gset :: flush_type ⇒ gset ⇒ asid ⇒ iset` and
`flush_snpshot :: flush_type ⇒ pt_snpshot ⇒ asid ⇒ pt_snpshot`

simply remove the relevant entries from the `iset`, `gset` and set them to `Fault` in the `pt_snpshot` depending on the specific flush instruction. The identical definitions of flush operations for the abstract MMU model have been presented in the previous chapter, in Sect. 6.2.2 at Page 118. The effect of the flush instruction on the state is:

```
iset_gset_snp_upd_s f ≡
let is = iset s; gs = gset s; snp = pt_snpshot s; a = asid s;
   hp = heap s; rt = root s
```

```
⟦Const c⟧ s = ⌊c⌋
⟦UnOp f e⟧ s = case ⟦e⟧ s of None ⇒ None | ⌊v⌋ ⇒ ⌊f v⌋
⟦BinOp f e₁ e₂⟧ s = case (⟦e₁⟧ s, ⟦e₂⟧ s) of
                    (⌊v₁⌋, ⌊v₂⌋) ⇒ ⌊f v₁ v₂⌋ | _ None
⟦HeapLookup vp⟧ s = case ⟦vp⟧ s of None ⇒ None
                    | ⌊v⌋ ⇒ read (iset s) (heap s) (root s) (mode s) (Addr v)

⟦BConst b⟧_b s = ⌊b⌋
⟦BComp f e₁ e₂⟧_b s = case (⟦e₁⟧ s, ⟦e₂⟧ s) of
                    (⌊v₁⌋, ⌊v₂⌋) ⇒ ⌊f v₁ v₂⌋ | _ None
⟦BBinOp f b₁ b₂⟧_b s = case (⟦b₁⟧_b s, ⟦b₂⟧_b s) of
                   (⌊v₁⌋, ⌊v₂⌋) ⇒ ⌊f v₁ v₂⌋ | _ None
⟦BNot b⟧_b s = (case ⟦b⟧_b s of None ⇒ None | ⌊v⌋ ⇒ ⌊¬ v⌋)
```

Figure 7.3: Semantics of Arithmetic and Boolean Expressions

```
in s⦇iset := flush_iset f is gs a, gset := flush_gset f gs a hp rt,
   pt_snpshot := flush_snpshot f snp a⦈
```

With this we conclude presenting the semantics operations and proceed to the big-step operational semantics of our language.

7.1.4 Operational Semantics

The semantics of arithmetic and Boolean expressions, ⟦A⟧ s and ⟦B⟧$_b$ s, are partial functions from program state to val and bool, respectively. They are shown in Figure 7.3. All of these are straightforward: HeapLookup is the memory read in the current state after the successful evaluation to a valid virtual pointer.

Figure 7.4 and Figure 7.5 show a big-step operational semantics of commands in the language. We write (c, s) ⇒ ⌊s'⌋ if command c, started in s, evaluates to s'. As usual, the semantics is the smallest relation satisfying the rules of Figure 7.4 and Figure 7.5. We model memory access failure explicitly by writing (c,s) ⇒ None for executions that fail. Figure 7.4 presents the semantic rules for successful memory access and Figure 7.5 summarizes the cases of memory access failure. These relations use the semantics operations of the previous section (Sect. 7.1.3) to model state effects.

The semantics could be made slightly more precise by distinguishing between situations that must always be avoided, such as using inconsistent TLB entries, and page faults, which can be recoverable by executing a page fault handler. In kernel-level code, page faults are usually unwanted as modelled here, in user-level code they will usually be recovered from by a page fault handler. We omit the distinction here for simplicity. It could easily be added by including a jump to a page fault exception handler if desired.

$$\texttt{(SKIP, s)} \Rightarrow \lfloor \texttt{s} \rfloor \qquad \frac{\texttt{(c}_1\texttt{, s}_1\texttt{)} \Rightarrow \lfloor \texttt{s}_2 \rfloor \qquad \texttt{(c}_2\texttt{, s}_2\texttt{)} \Rightarrow \texttt{s}_3}{\texttt{(c}_1\texttt{;; c}_2\texttt{, s}_1\texttt{)} \Rightarrow \texttt{s}_3}$$

$$\frac{[\![\texttt{lval}]\!]\ \texttt{s = } \lfloor \texttt{vp} \rfloor \qquad [\![\texttt{rval}]\!]\ \texttt{s = } \lfloor \texttt{v} \rfloor \qquad \texttt{vp} \notin \mathcal{IC}\ \texttt{s} \qquad \texttt{Addr vp} \hookrightarrow_s \texttt{pp}}{\texttt{(lval ::= rval, s)} \Rightarrow \lfloor \texttt{heap_iset_gset_upd}_s\ \texttt{(pp} \mapsto \texttt{v)} \rfloor}$$

$$\frac{\langle \texttt{b} \rangle\ \texttt{s} \qquad \texttt{(c}_1\texttt{, s)} \Rightarrow \texttt{t}}{\texttt{(IF b THEN c}_1\texttt{ ELSE c}_2\texttt{, s)} \Rightarrow \texttt{t}} \qquad \frac{\neg\langle \texttt{b} \rangle\ \texttt{s} \qquad \texttt{(c}_2\texttt{, s)} \Rightarrow \texttt{t}}{\texttt{(IF b THEN c}_1\texttt{ ELSE c}_2\texttt{, s)} \Rightarrow \texttt{t}}$$

$$\frac{\langle \texttt{b} \rangle\ \texttt{s} \qquad \texttt{(c, s)} \Rightarrow \lfloor \texttt{s''} \rfloor \qquad \texttt{(WHILE b DO c, s'')} \Rightarrow \texttt{s'}}{\texttt{(WHILE b DO c, s)} \Rightarrow \texttt{s'}}$$

$$\frac{\neg\langle \texttt{b} \rangle\ \texttt{s}}{\texttt{(WHILE b DO c, s)} \Rightarrow \lfloor \texttt{s} \rfloor} \qquad \frac{\texttt{mode s = Kernel}}{\texttt{(Flush f, s)} \Rightarrow \lfloor \texttt{iset_gset_snp_upd}_s\ \texttt{f} \rfloor}$$

$$\frac{\texttt{mode s = Kernel} \qquad [\![\texttt{rte}]\!]\ \texttt{s = } \lfloor \texttt{rt} \rfloor}{\texttt{(UpdateRoot rte, s)} \Rightarrow \lfloor \texttt{root_iset_gset_upd}_s\ \texttt{Addr rt} \rfloor}$$

$$\frac{\texttt{mode s = Kernel} \qquad \texttt{snp = new_snp s} \qquad \texttt{is = snp_incon a s}}{\texttt{(UpdateASID a, s)} \Rightarrow \lfloor \texttt{s}(\!|\texttt{asid := a, iset := is, pt_snpshot := snp}|\!) \rfloor}$$

$$\frac{\texttt{mode s = Kernel}}{\texttt{(SetMode m, s)} \Rightarrow \lfloor \texttt{s}(\!|\texttt{mode := m}|\!) \rfloor}$$

Figure 7.4: Big-Step Semantics of Commands with Successful Memory Access

The assignment rule in the second row of Figure 7.4 requires that both the arithmetic expressions for the left and right hand side evaluate without failure. The left hand side is taken as a virtual pointer, the right hand side as the value being assigned. The assignment succeeds if the virtual address `vp` is *consistent* in the current state: $\texttt{vp} \notin \mathcal{IC}\ \texttt{s}$ represents $\texttt{Addr vp} \notin \texttt{iset s}$ and *mapped*: $\texttt{Addr vp} \hookrightarrow_s \texttt{pp}$ means

$$\texttt{phy_ad (iset s) (heap s) (root s) (mode s) (Addr vp) = } \lfloor \texttt{pp} \rfloor$$

Updating the page table root and ASID registers require the privileged (`Kernel`) mode for successful operation.

7.1.5 Hoare Logic

We now proceed to the logic rules of these semantics. Having shown the semantics, we can now proceed to defining Hoare triples. Validity is the usual:

$$\{\!|\texttt{P}|\!\}\ \texttt{c}\ \{\!|\texttt{Q}|\!\} \equiv \forall \texttt{s s'. (c, s)} \Rightarrow \texttt{s'} \wedge \texttt{P s} \longrightarrow (\exists \texttt{r. s' = } \lfloor \texttt{r} \rfloor \wedge \texttt{Q r})$$

$$\frac{[\![\texttt{lval}]\!]\ \texttt{s} = \lfloor\texttt{vp}\rfloor \qquad [\![\texttt{rval}]\!]\ \texttt{s} = \lfloor\texttt{v}\rfloor \qquad \texttt{vp} \in \mathcal{IC}\ \texttt{s} \lor \texttt{Addr vp} \hookrightarrow_s \texttt{None}}{(\texttt{lval ::= rval, s}) \Rightarrow \texttt{None}}$$

$$\frac{[\![\texttt{lval}]\!]\ \texttt{s} = \texttt{None} \lor [\![\texttt{rval}]\!]\ \texttt{s} = \texttt{None}}{(\texttt{lval ::= rval, s}) \Rightarrow \texttt{None}} \qquad \frac{(\texttt{c}_1\texttt{, s}_1) \Rightarrow \texttt{None}}{(\texttt{c}_1\texttt{;; c}_2\texttt{, s}_1) \Rightarrow \texttt{None}}$$

$$\frac{[\![\texttt{b}]\!]_b\ \texttt{s} = \texttt{None}}{(\texttt{IF b THEN c}_1\ \texttt{ELSE c}_2\texttt{, s}) \Rightarrow \texttt{None}} \qquad \frac{\texttt{mode s = User}}{(\texttt{Flush f, s}) \Rightarrow \texttt{None}}$$

$$\frac{\langle\texttt{b}\rangle\ \texttt{s} \qquad (\texttt{c, s}) \Rightarrow \texttt{None}}{(\texttt{WHILE b DO c, s}) \Rightarrow \texttt{None}} \qquad \frac{[\![\texttt{b}]\!]_b\ \texttt{s} = \texttt{None}}{(\texttt{WHILE b DO c, s}) \Rightarrow \texttt{None}}$$

$$\frac{\texttt{mode s = User} \lor [\![\texttt{rt}]\!]\ \texttt{s} = \texttt{None}}{(\texttt{UpdateRoot rt, s}) \Rightarrow \texttt{None}} \qquad \frac{\texttt{mode s = User}}{(\texttt{UpdateASID a, s}) \Rightarrow \texttt{None}}$$

$$\frac{\texttt{mode s = User}}{(\texttt{SetMode m, s}) \Rightarrow \texttt{None}}$$

Figure 7.5: Big-Step Semantics of Commands with Unsuccessful Memory Access

Instead of defining a separate syntactic Hoare calculus, we directly derive Hoare rules from validity as theorems in Isabelle/HOL. Figure 7.6 and Figure 7.7 show the resulting rules. Figure 7.6 summarizes the rules for traditional commands such as `SKIP`, `WHILE`, etc. and Figure 7.7 gives the rules for the commands that interact with the cached address translation layer. We note that the traditional rules are completely standard. As expected, cached address translation had no direct influence there. We write $\langle\langle\texttt{b}\rangle\rangle\ \texttt{s}$ to denote that $[\![\texttt{b}]\!]_b\ \texttt{s} \neq \texttt{None}$: the precondition in the `IF` and `WHILE` case must be strong enough to guarantee failure free evaluation of the condition `b`. The rules in Figure 7.7 are in weakest-precondition form. They have a generic postcondition `P` and the weakest precondition that will establish `P`. We will now explain them.

The assignment rule requires that the expressions `l` and `r` evaluate without failure. The assignment succeeds if the virtual address `vp` is *consistent* in the current state ($\texttt{vp} \notin \mathcal{IC}\ \texttt{s}$) and `vp` is *mapped* ($\texttt{Addr vp} \hookrightarrow_s \texttt{pp}$). The effect of the assignment is the heap, `iset` and `gset` update `heap_iset_gset_upd` as we described in the semantic operations (Sect. 7.1.3).

The rule for the command `UpdateRoot`, only available in kernel mode, updates the current page table root to the value of the expression `rte`. The effect is modelled by `root_iset_gset_upd` defined in the semantic operations (Sect. 7.1.3).

The `UpdateASID` command, also only available in kernel mode, sets the new ASID `a`, increases the `iset` using `snp_incon`, and records a page table snapshot for the old ASID using `new_snp`.

Finally, `Flush` is the instruction that the makes the `iset` smaller, and removes mappings in the snapshots of inactive ASIDs, using `iset_gset_snp_upd` from the

semantic operations (Sect. 7.1.3).

With this we conclude presenting the logic rules, we use these rules along with the simplification theorems of the next section to verify programs in Chapter 8.

7.2 Safe Set

This section identifies a key concept that simplifies reasoning for code that does not directly modify page tables, and even for code that does modify page tables safely.

As outlined in the previous section, the assignment rule involves reasoning about: a) consistency of the ASID and virtual address pair in the current state b) valid address translation, and c) potential update of the inconsistent set (`iset`) and the global set (`gset`). The functions `phy_ad` and `ptable_comp` explicitly mention page table walks, which means reasoning for every memory write has to be aware of them: the proof engineer has to discharge the page table obligations even if the memory write has nothing to do with page tables. Even though the assignment rule is phrased in a weakest-precondition style, the resulting verification conditions will accumulate quickly.

Our aim in developing a program logic that models low-level details of virtual memory management is to verify the correctness of OS kernel code, including code that manipulates the virtual memory layer, but at the same time also to create a framework that allows us to easily show the absence of TLB inconsistencies in other kernel code as well as user code.

While the set of potentially inconsistent addresses might be large, the only commands that add new elements to this set are assignments to page tables and changing the page table root and ASID registers. This section focuses on assignments, which is an extremely frequent operation, whereas updating the page table root and ASID registers only happens in context switching code.

$$\{\!|P|\!\}\ \texttt{SKIP}\ \{\!|P|\!\} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \qquad P' \longrightarrow P}{\{\!|P'|\!\}\ c\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \wedge \langle b\rangle|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|P \wedge \neg\langle b\rangle|\!\}\ c_2\ \{\!|Q|\!\}}{\{\!|P \wedge \langle\langle b\rangle\rangle|\!\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \wedge \langle b\rangle|\!\}\ c\ \{\!|P|\!\} \qquad P \longrightarrow \langle\langle b\rangle\rangle}{\{\!|P|\!\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\!|P \wedge \neg\langle b\rangle|\!\}} \qquad \frac{\{\!|P|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|Q|\!\}\ c_2\ \{\!|R|\!\}}{\{\!|P|\!\}\ c_1;;\ c_2\ \{\!|R|\!\}}$$

Figure 7.6: Hoare Logic Rules for Standard Commands

$$\{\!|\lambda \mathtt{s}.\ [\![\mathtt{l}]\!]\ \mathtt{s} = \lfloor \mathtt{vp} \rfloor \wedge [\![\mathtt{r}]\!]\ \mathtt{s} = \lfloor \mathtt{v} \rfloor \wedge \mathtt{vp} \notin \mathcal{IC}\ \mathtt{s} \wedge \mathtt{Addr\ vp} \hookrightarrow_s \mathtt{pp} \wedge \mathtt{P\ (heap_iset_gset_upd}_s\ \mathtt{(pp} \mapsto \mathtt{v))}|\!\}\ \mathtt{l ::= r}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge [\![\mathtt{rte}]\!]\ \mathtt{s} = \lfloor \mathtt{rt} \rfloor \wedge \mathtt{P\ root_iset_gset_upd}_s\ \mathtt{Addr\ rt}|\!\}\ \mathtt{UpdateRoot\ rte}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \exists \mathtt{snp\ il.\ mode\ s = Kernel} \wedge \mathtt{snp = new_snp\ s} \wedge \mathtt{il = snp_incon\ a\ s} \wedge \mathtt{P\ s(|asid := a,\ iset := il,\ pt_snpshot := snp|)}|\!\}\ \mathtt{UpdateASID\ a}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge \mathtt{P\ iset_gset_snp_upd}_s\ \mathtt{f}|\!\}\ \mathtt{Flush\ f}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge \mathtt{P\ (s(|mode := flg|))}|\!\}\ \mathtt{SetMode\ flg}\ \{\!|\mathtt{P}|\!\}$$

Figure 7.7: Hoare Logic Rules for Commands with TLB Effects

The key insight is that OS engineers do not reason about a constantly changing set of inconsistent addresses when they write kernel code, but instead approach the problem from the other direction. Given a set of virtual addresses we know is consistent and safe to write to, the only way this set can become unsafe is when we change page table mappings that are responsible for translating the addresses in this set. All other mappings can change arbitrarily, as long as we stay within that set.

To formalise this notion, we re-use another function from Kolanski's page table interface (Kolanski and Klein, 2009) `ptable_trace` (for bit shift notation, please refer to Sect. 2.2.3):

```
ptable_trace h rt vp ≡
let vp_val = addr_val vp; pd_idx_offset = vaddr_pd_index vp_val << 2;
   pt_idx_offset = vaddr_pt_index vp_val << 2;
   pd_touched = {rt + pd_idx_offset};
   pt_touched = λpt_base. {pt_base + pt_idx_offset}
in case decode_pde (the (h (rt + pd_idx_offset))) of
   PageTablePDE pt_base ⇒ pd_touched ∪ pt_touched pt_base
   | _ ⇒ pd_touched
```

This function takes a heap, a root, and a virtual address `va`, and returns the set of physical addresses that are used in the page table walk for `va`. The function `decode_pde` (definition not shown here) decodes the corresponding machine word in to a page directory entry.

The page table interface has the property that memory writes outside this set will not change the outcome of the walk for `va`. Generalising this notion to a set of virtual addresses, we can define

$$\texttt{ptrace_set V s = } (\bigcup_{\texttt{x}\in\texttt{V}} \texttt{ ptable_trace (heap s) (root s) x})$$

The `ptrace_set V` gives us the set of physical addresses that encode the translation for the virtual addresses in `V`. Using this set, we can define when a set of virtual address is a *safe set*, that is, a static set of addresses that we can write to in the current state without making that same set unsafe:

$$\texttt{safe_set V s} \equiv \forall \texttt{va}\in\texttt{V. va} \in \mathcal{C}\ \texttt{s} \wedge (\exists \texttt{p. va} \hookrightarrow_s \texttt{p} \wedge \texttt{p} \notin \texttt{ptrace_set V s})$$

where $\mathcal{C}\ \texttt{s} \equiv \{\texttt{va} \mid \texttt{va} \notin \texttt{iset s}\}$. In words, a set `V` is a safe set in state `s` iff all addresses $\texttt{va} \in \texttt{V}$ are consistent in the current state under the current ASID, if they map to a physical physical address `p`, and if that address is not part of the page table encoding for any of the addresses in `V`.

Our first observation is that once a set `V` is a safe set, assignments within it can no longer make it unsafe, and the safe set property will remain invariant:

Theorem 30. *Any write to the safe set will preserve the safe set. Formally:*

$$\{\!|\lambda\texttt{s. safe_set V s} \wedge$$
$$(\exists\texttt{vp v.}\ [\![\texttt{lval}]\!]\ \texttt{s} = \lfloor\texttt{vp}\rfloor \wedge [\![\texttt{rval}]\!]\ \texttt{s} = \lfloor\texttt{v}\rfloor \wedge \texttt{Addr vp} \in \texttt{V})|\!\}$$
$$\texttt{lval ::= rval}\ \{\!|\lambda\texttt{s. safe_set V s}|\!\}$$

Proof. We apply the previous weakest-precondition rule for assignment and reason that a write to a mapped virtual address `vp` from the set `V` does not resolve to the page table trace of set `V`, and therefore will not change any page table entries for the set `V`. Hence none of the addresses in `V` will be added to the inconsistent set in the `incon_comp` update, and if they were consistent before, they will be consistent afterwards. While we might have changed other mappings, the trace of the mappings for `V` has not changed, and so all conditions of `safe_set` are still satisfied. □

The next theorem shows that it is sufficient to check that the address is part of the safe set to reason about the heap effect of the memory write. Since we already know that the safe set will remain invariant, we can ignore how the inconsistent set develops during execution as long we only operate within the safe set. When switching modes, we will still be interested in at least parts of the inconsistent set, for instance in the fact that we have only invalidated virtual addresses for the current ASID, but not for any other ASID. To enable this kind of reasoning, we leave the information in the rule, but note that the only reference to the inconsistent set is inside the definition of `safe_set`, which we already have shown invariant.

Theorem 31. *In the assignment rule, it is sufficient to check the static safe set instead of the dynamic inconsistent set $\mathcal{IC}$.*

```
{|λs. (∃vp v. ⟦V⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧ Addr vp ∈ V ∧
      Q (heap_iset_gset_upd_s (the_phy_ad vp s ↦ v))) ∧ safe_set V s|}
V ::= rval {|Q|}
```

where
`the_phy_ad vp s ≡ the (pt_lookup (heap s) (root s) (mode s) (Addr vp))`

Proof. Follows directly from the definition of `safe_set` and the existing assignment rule. □

For the parts of the code that are not interested in TLB effects, i.e. outside context switching and page table manipulations, this rule enables proof engineers to treat the code as if no TLB was present as long as they can show that each memory access is within a statically known safe memory region. The majority of OS and user-level code satisfies this condition. The rule still mentions address translation — reducing reasoning under address translation to traditional Hoare logic reasoning is orthogonal and, for instance, solved in Kolanski's separation logic framework (Kolanski, 2011), or in simpler instances by maintaining locally injective constant memory mappings, which then behave like standard memory.

The reduction to checking a static set of addresses also give us the justification that compilers do not introduce additional complexity into reasoning under the TLB, they merely give us additional addresses that need to be part of this safe set, e.g. the area of virtual memory that contains code, stack, and global variables should be part of the set that we show safe once at the beginning of program execution.

This section has presented the main tool for reducing TLB reasoning to a simpler, static setting. The next chapter use this rule to show how the program logic behaves in the scenarios that are common in OS kernel code.

7.3 Summary and Remarks

In this chapter, we have presented a Hoare-style logic for verifying programs in the presence of TLB-address translation. We have provided the syntax and semantics of a generic heap based language that takes the abstract ARMv7-A MMU model of Chapter 6 into account, and have derived the soundness of their Hoare logic rules. We have also provided reduction rules for memory write operations.

The strength of the logic is its simplicity, which took multiple iterations to achieve, finding a balance between abstraction soundness, not too complex reasoning, and not too much conservatism for allowing optimisations and idioms used in real OS code, resulting in a program logic that feels familiar to proof engineers.

In the next chapter, we use the logic to prove reduction theorems that mirror the informal reasoning OS engineers perform when they write kernel code. It also allows us to drop into a simpler setting when we reason about code that does not affect virtual memory mappings.

CHAPTER

EIGHT

Case Study

In the previous chapter, we have presented a logic for verifying programs in the presence of cached address translation. In this chapter, we apply this logic to extract invariants and conditions necessary for reasoning about user-level and kernel-level executions, context switching and page table operations. This case study shows that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

This chapter is organised as: we present the formal MMU layout of a toy kernel inspired by the seL4 microkernel, we then present reductions theorems for standard user-level code, kernel-level code without TLB or page table manipulations, context-switching, and page table manipulations.

This chapter is based on the published work (Syeda and Klein, 2018) and the submitted work (Syeda and Klein, 2019).

8.1 MMU Layout - Formal Modeling

In this chapter, we apply the program logic and its reduction theorem presented in the previous chapter to reason about program fragments in multiple scenarios. The aim is not to verify specific OS kernel code or user-level code, but rather to demonstrate how the logic behaves in the settings one might expect to use it in. These are: kernel-level code without TLB or page table manipulations, standard user-level code, context-switching, as well as a representative example for a page table manipulation.

The case study uses the seL4 microkernel as inspiration to distill out code sequences for a toy kernel that manages page tables and the TLB, and prevents users from accessing these, as well as other kernel data structures, directly. It maintains a set of page tables, typically one per user, potentially shared. This setting applies to all major protected-mode OS kernels, e.g. Linux, Windows, MacOS, as well as most microkernels, e.g. the L4 family, including seL4, etc. While simplified, the case study aims to be realistic in demonstrating popular techniques for avoiding TLB flushes, such as ASIDs.

There are multiple ways to achieve separation between user-accessible memory and kernel memory. For instance, the kernel could switch to its own page table and make sure that none of the user-level page tables contain mappings to the physical addresses that store kernel data structures. For our example, we choose a slightly more interesting and popular setting. To avoid switching page tables for entering the kernel, the kernel maintains a so-called kernel window. [1] The kernel

[1] This is the technique attacked by Meltdown (Lipp et al., 2018). Since hardware manufacturers are promising to fix this flaw, we present the more interesting setting instead of the less

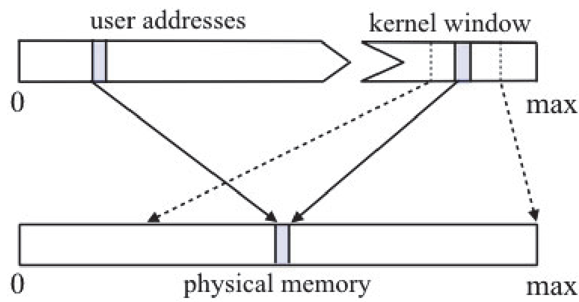

Figure 8.1: Virtual Address Space with Kernel Window

window is a set of virtual addresses, unavailable to the user, backed by kernel mappings with permissions that make them available only in kernel mode. Linux, for instance, uses this scheme, and in a 32-bit address space, which would span 4GB of memory, e.g. only 3.5GB may actually addressable in user mode. The top 512KB implement the kernel window. Figure 8.1 shows an example for a virtual address space maintained by our toy OS kernel.

As is customary, the mappings for this kernel window are constant and global, and typically implement just a very simple offset to transform a virtual into a physical address, although any injective function would work. Since the mappings are constant and their translation function is statically known, the corresponding page table entries are constant too. That means, each user-level page table that the kernel maintains has a number of known entries which, for each user, reside at the same position in the page table encoding. Thinking back to Sect. 7.2, this gives us a very simple candidate for the safe set that we can use for reasoning about standard kernel code: all addresses mapped by the kernel window minus the addresses that are used to encode the kernel mappings in any of the potentially active page table data structures.

Figure 8.2 gives an overview of the page table layout maintained by our toy kernel. We have used two-level ARMv7-A page tables and have chosen very simple concrete encodings, fixing a specific layout. The reasoning ideas below do not depend on this encoding, they just represent a simple instance of the general setting. The page granularities used in our example will be *section*: 1MB and *pages*: 4KB blocks of physical memory. The first-level page table contains either mappings for sections, or pointers to the second-level page table, while the second-level page table has mappings for pages. The kernel window is mapped in the high area of the first-level of every page table. Page tables are stored in the memory locations mapped by kernel window, and at any time, more than one such page table will usually be present, e.g. one for each user. The kernel maintains specific data structures to maintain page table layout, we explain these data structures and their formalisation below.

complex and slower scenario with a separate kernel address space.

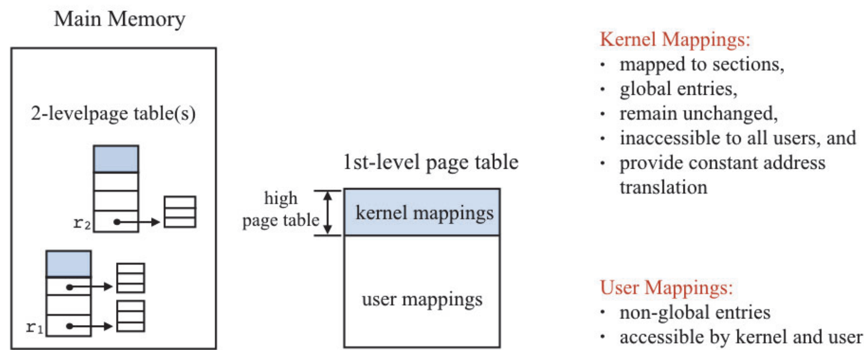

Figure 8.2: Page Table Layout by seL4-inspired Kernel

8.1.1 Kernel Data Structures

We now explain the formal modeling of kernel features and data structures essential to maintain the above mentioned MMU setting.

The Kernel Physical Memory: We specify kernel physical memory as the set of consecutive physical addresses between a lower and an upper bound:

```
kernel_lower :: paddr
kernel_upper :: paddr
kernel_phy_mem = {kernel_lower..kernel_upper}
```

Where `{l..u}` represents the set of consecutive enumerations between `l` and `u`.

The Kernel Window: The kernel window of the address space is mapped by the high part of the page table. For a page table starting at the root `rt`, we specify its region mapping the kernel window as:

```
k_window_lower :: paddr
k_window_upper :: paddr
high_ptable rt = {rt + k_window_lower..rt + k_window_upper}
```

We will associate translation properties of the kernel window with `high_ptable` in our MMU layout later in this section.

The Map between Page Table Roots and ASIDs: In paging, processes have their page tables with specific roots, and the OS kernel assigns ASIDs to these roots in order to enable TLB caching. The OS kernel is required to store the information of which ASIDs are assigned to which page tables. It might maintain an explicit map, or stores this information implicitly as part of a larger data structure. We formalise such a map between page table roots and ASIDs for our toy kernel.

We assign a contiguous portion of memory to store the page table roots and their assigned ASIDs. We specify this portion of memory as:

```
rt_map_lower :: paddr
rt_map_upper :: paddr
rt_map_area = {rt_map_lower..rt_map_upper}
```

The layout of `rt_map_area` is shown in the Figure 8.3. The data stored in the `rt_map_area` is read and logically interpreted as:

> `(heap s) n` *represents a root, and*
> `heap s (n + 1)` *represents an ASID, where*
> `rt_map_lower ≤ 2n < rt_map_upper, n ∈ {0, 1, 2, ..}`

We then define a function `root_set` to encode the data stored in `rt_map_area` in the form of:

```
root_set :: p_state ⇒ (paddr × asid option) set
```

where `p_state` is the record type for the program state. The `root_set` reads the `rt_map_area` from the `heap` of the given program state, and returns a set of pairs of page table roots (`paddr`) and their assigned ASIDs. The assigned ASIDs are modeled with the `option` type, representing the situation when some processes are not yet assigned an ASID. The page table roots are 32-bit physical addresses, while ASIDs are only 8-bit.

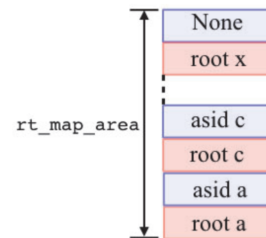

Figure 8.3: Roots Map Layout

Using the function `root_set`, we define the map of the page table roots to their ASIDs as:

```
root_map :: p_state ⇒ (paddr ⇒ asid option)
root_map s =
(λrt. if rt ∈ fst ` root_set s then SOME a. (rt, a) ∈ root_set s
      else None)
```

The function `root_map` converts the `root_set` of the given program state to the map from roots to the assigned ASIDs. Here, `SOME` denotes the Hilbert's choice operator.

With this we conclude presenting the page table roots map, we will assert MMU properties on this root map later in this section.

The Log of Page Table Roots: Given the root-map, we can compute a list of page table roots that have an ASID assigned to them. Given a program state, the log of the page table roots that have an ASID assigned is formalised as:

```
valid_roots :: p_state ⇒ 32 word list
valid_roots s = filter (λr. root_map s (Addr r) ≠ None) enum
```

```
root_log :: p_state ⇒ paddr list
root_log s = map Addr (valid_roots s)
```

Given a program state `s`, the function `valid_roots` returns the list of valid 32-bit roots from `root_map` after filtering those roots that do not have an assigned ASID. The function `enum` enumerates all physical addresses. We can then convert this list of valid roots to a set as:

```
roots :: p_state ⇒ paddr set
roots s = set (root_log s)
```

Where the function `set` converts an `'a list` to `'a set`.

Page Table Footprint: We find the footprint of a two-level page table starting at the location `rt` as:

```
pt_area :: p_state ⇒ paddr ⇒ paddr set
pt_area s rt ≡ ⋃v ptable_trace (heap s) rt v
```

This means, the footprint of a page table starting at root `rt` is the set of all addresses that can be produced by a `ptable_trace`.

Kernel Data: To obtain the memory the kernel maintained data structures reside in, we combine the `pt_area`s of all valid page tables and the `rt_map_area`:

```
kernel_data :: p_state ⇒ paddr set list
kernel_data s ≡ map (pt_area s) (root_log s) @ [rt_map_area]
```

```
kernel_data_area :: p_state ⇒ paddr set
kernel_data_area s ≡ ⋃set (kernel_data s)
```

With this we conclude the presentation of MMU-related kernel data structures, and proceed to the assertions for these data structures that the kernel is required to maintain for correct execution.

8.1.2 Assertions on MMU Layout

We now formulate the assertions for MMU layout; the OS kernel provides and preserves these assertions for ensuring the correct execution of user programs and of itself. The assertions will also be invariants for program verification in this case study.

The assertions on the MMU layout are

1. all kernel data structures reside in physical kernel memory,
2. no kernel data structures overlap,
3. page table roots are word-aligned,
4. the current ASID is associated correctly with the current page table root,
5. all page tables contain the kernel mappings,
6. kernel mappings are global and static; which means even the kernel is not allowed to change them,
7. user mappings are always non-global,
8. no page table contains mappings that allows user mode to resolve to physical kernel memory,
9. the mapping from page table roots to ASIDs is injective, and
10. the global set is saturated with the global kernel mappings.

Additionally, for our abstract TLB model, we assert that the global set `gset` is equal to the globally mapped virtual addresses of the current state. As kernel mappings are the only mappings marked as global and they are static in all the page tables, this assertion implies that the global set is equal to the globally mapped virtual addresses of all page tables.

The following two properties are true for most of the execution of the system, but are invalidated temporarily:

1. The kernel window minus the entries that encode kernel mappings is a safe set. This property only holds in kernel mode.

2. The ASID snapshots agree with the page table for that ASID/user. This property is invalidated for a specific ASID between page table manipulations and flush instructions.

Formally, the MMU layout is:

```
mmu_layout s ≡
kernel_data_area s ⊆ kernel_phy_mem ∧
non_overlapping (kernel_data s) ∧
aligned (roots s) ∧
root_map s (root s) = ⌊asid s⌋ ∧
kernel_mappings s ∧
user_mappings s ∧ partial_inj (root_map s) ∧ saturated_gset s
```

Where `kernel_phy_mem`, `kernel_data`, `kernel_data_area`, `root_map` are the same functions as presented in the previous section. We now explain the MMU layout assertions.

Non-overlapping Kernel Data Structures: We impose that the memory region `kernel_data_area` is located in the `kernel_phy_mem`, and the `kernel_data`

(the list) is `non_overlapping`. The definition of a list of sets of addresses being non-overlapping is:

```
non_overlapping [] = True
non_overlapping (x · xs) = (x ∩ ⋃set xs = ∅ ∧ non_overlapping xs)
```

From non-overlapping kernel data, we derive a slightly more direct predicate for non-overlapping page tables: the page tables with different page table roots do not overlap.

```
non_overlapping_tables s ≡
∀rt rt'.
   rt ∈ roots s ∧ rt' ∈ roots s ∧ rt ≠ rt' ⟶
   pt_area s rt ∩ pt_area s rt' = ∅
```

Kernel Mappings: The `kernel_phy_mem` is mapped by the `kernel_mappings`. The `kernel_mappings` predicate states that

```
kernel_mappings s ≡ kernel_window s ∧ high_ptable_equal s
```

Where

```
   kernel_window s ≡
   ∀rt∈roots s.
      (∀va. rt + pd_offset va ∈ high_ptable rt ⟶
            global_static_kmappings (heap s) rt va) ∧
      (∀va. rt + pd_offset va ∉ high_ptable rt ⟶
            non_global (heap s) rt (Addr va))


   high_ptable_equal s ≡
   ∀rt rt'.
      rt ∈ roots s ∧ rt' ∈ roots s ⟶
      (∀va. rt + pd_offset va ∈ high_ptable rt ⟶
            get_pde (heap s) rt (Addr va) =
            get_pde (heap s) rt' (Addr va))
```

The function `kernel_window` asserts that all the valid page tables of the given state `s` have `global_static_kmappings` in their `high_ptable` area, while rest of the area is for `non_global` user addresses. The global and static kernel mappings of the page table starting at the root `rt` in the heap `hp` are asserted as:

```
global_static_kmappings hp rt va ≡
(∃p perms.
    get_pde hp rt (Addr va) = ⌊SectionPDE p perms⌋ ∧
    arm_p_nG perms = 0 ∧ ¬ user_perms perms) ∧
pt_lookup hp rt Kernel (Addr va) = ⌊Addr va - offset⌋ ∧
```

```
Addr va - offset ∈ kernel_phy_mem
```

Together with the `kernel_window`, the above assertion states that the kernel maps itself through `high_ptable` areas of all page tables to memory sections. Also, the kernel window provides offset translation, such that for all virtual addresses `va` in the kernel window, we get `Addr (va - offset)` as the physical address, i.e. the outcome of the translation is easily described statically. Additionally, we choose `offset` such that for all `va` in the kernel window, `Addr (va - offset) ∈ kernel_phy_mem`. This is a simple yet realistic setup, similar to what e.g. seL4 uses.

The assertion `high_ptable_equal` for `kernel_mappings` formalises the requirement that the `kernel_window` in all of the page tables of the given state `s` are identical. It simply equates the `get_pde`s of the `high_ptable`s of all the valid roots.

User Mappings: The restriction on user mappings is easily phrased with address translation predicates. We also assert that the user mappings are `non_global` (`nG = 1`):

```
user_mappings s ≡
∀rt∈roots s.
   ∀va pa.
      pt_lookup (heap s) rt User va = ⌊pa⌋ ⟶
      pa ∉ kernel_phy_mem ∧ non_global (heap s) rt va
```

Injectivity of Assigned ASIDs: The `mmu_layout` also asserts that the active root is correctly associated with the active ASID, and the ASIDs have injective mappings. The definition of injectivity is only for the parts where an ASID is assigned:

```
partial_inj f ≡ ∀x y. x ≠ y ⟶ f x ≠ f y ∨ f x = None ∧ f y = None
```

Saturated Global Set: Finally, the assertion `saturated_gset` in `mmu_layout` states that the global set of the abstract TLB is equal to the globally mapped virtual addresses of the active page table (effectively to all page tables as globally mapped virtual addresses are equal across all page tables).

```
saturated_gset s ≡
{va | root s + pd_offset (addr_val va) ∈ high_ptable (root s)} =
gset s
```

This concludes the formalisation of the main kernel invariants needed in the case study.

Consistency of Assigned ASIDs: To avoid flushing the TLB, we maintain for most of the execution the additional invariant that the TLB is fully consistent for

all ASIDs that we might switch to, and that for each ASID the TLB snapshot agrees with the page table that we *would* switch to for that ASID. This means, if there were page table modifications for a user we are about to switch to, we assume that the corresponding flush has already happened. Since the property is not valid for all ASIDs between page table modifications and flush, we provide a set of ASIDs as argument to exclude. If this set is empty, we will omit the argument in the notation. We assert ASIDs consistency as:

```
asids_consistent S s ≡
assigned_asids_consistent S s ∧ gset_consistent s
```

```
assigned_asids_consistent S s ≡
∀r a. let is = fst (pt_snpshot s a); snp_pt = snd (pt_snpshot s a);
          ptwalk = ptd_walk a (heap s) r
       in root_map s r = ⌊a⌋ ∧ a ∉ S ∪ {asid s} ⟶
         is = ∅ ∧ ptable_comp snp_pt ptwalk = ∅
```

```
gset_consistent s ≡
∀r a. let is = fst (pt_snpshot s a); snp_pt = snd (pt_snpshot s a);
          ptwalk = ptd_walk a (heap s) r
       in r ∈ roots s ⟶ gset s ∩ (is ∪ ptable_comp snp_pt ptwalk) = ∅
```

The assertion `assigned_asids_consistent` ensures the consistency of the stored TLB snapshot for the assigned ASIDs excluding the active ASID as well as known inconsistent ASIDs. It asserts for these ASIDs that the stored inconsistent set of virtual address is empty and the stored page table agrees with the respective page table in the memory (empty `ptable_comp` set). Similarly, the assertion `gset_consistent` ensures the TLB-consistency of the globally mapped virtual addresses for all the page table present in the memory.

This concludes the formalisation of the necessary kernel invariants.

8.2 User Execution

With the invariants in place, we can proceed to reduction theorems. The simplest of these is user-level execution: when the kernel has switched to user mode, the inconsistent set should be empty for the current ASID, and since the user cannot perform any actions that adds addresses to this set, it will remain empty. Most actions that have any effect on the inconsistent set are explicitly privileged, i.e. unavailable in user mode. Only assignments could possibly have an adverse effect.

The following theorem shows that they do not, and that any arbitrary assignment in user mode will preserve not only this property of the inconsistent set, but, almost

trivially, also all kernel invariants. In that sense it is a simple demonstration of the separation that virtual memory achieves between kernel and user processes.

Theorem 32. *When the kernel invariants hold, we are in user mode, the inconsistent set is empty, then these three conditions are preserved, and the heap is updated as expected. We additionally have to assume that the address the left-hand side resolves to is mapped.*

$$\{\!|\lambda \texttt{s. mmu_layout s} \wedge \texttt{mode s = User} \wedge$$
$$\quad \mathcal{IC}\ \texttt{s = } \emptyset \wedge [\![\texttt{lval}]\!]\ \texttt{s = } \lfloor \texttt{vp} \rfloor \wedge [\![\texttt{rval}]\!]\ \texttt{s = } \lfloor \texttt{v} \rfloor \wedge \texttt{Addr vp} \hookrightarrow_s \texttt{p}|\!\}$$
$$\texttt{lval ::= rval}$$
$$\{\!|\lambda \texttt{s. mmu_layout s} \wedge \texttt{mode s = User} \wedge \mathcal{IC}\ \texttt{s = } \emptyset \wedge \texttt{heap s p = } \lfloor \texttt{v} \rfloor|\!\}$$

Proof. We take the set of all user-mapped addresses as the safe set, and then using the assignment theorem for the safe set (Theorem 31) reason that the update is safe i.e. the inconsistent set remains empty. Moreover, we also know from the kernel invariants that user addresses do not map to page tables, so the heap update cannot modify a page table, and therefore does not add anything to the inconsistent set. Neither can the update touch any of the other kernel data structures, which all reside in kernel memory only. □

The invariant part of the rule above could be moved to the definition of validity and be hidden from the user completely. We would still have to assume that the address `vp` is mapped, because we do not distinguish between recoverable page faults and program failure. In the settings we are interested in, we aim to avoid page faults. In a setting with dynamically mapped pages, e.g. by a page fault handler, the logic can be extended to take this conditional execution into account, for instance using an exception mechanism or a conditional jump. In that case, the condition that addresses are mapped can be dropped, and we arrive at a standard Hoare logic assignment rule.

8.3 Kernel Execution

User execution boils down to standard reasoning. Using our safe set concept from the previous chapter (Sect. 7.2) we can show that kernel execution without virtual memory modifications does as well.

As mentioned in the MMU layout description (Sect. 8.1), the safe set for kernel execution is the entire kernel window, i.e. the virtual addresses that are mapped by the kernel mappings, minus the addresses of the page table entries that encode these kernel mappings. Since we will need to re-establish this set every time we switch to a different page table, and it is always safe to reduce the safe set, we not only take out the page table entries that encode the kernel mappings for the current page table, but we also take out the addresses of the high page table region

of all page tables the kernel maintains and might switch to. Formally, the kernel safe set is:

```
kernel_safe s = vas_by_kmappings (root s) - vas_to_kmappings s
```

Where the function `vas_to_kmappings` models the set of virtual addresses mapped to the high page tables of all the roots in the memory, i.e.

```
vas_to_kmappings s ≡
let rt = root s; hp = heap s; high_ptables = ⋃r∈roots s high_ptable r
in {va ∈ vas_by_kmappings rt | ptable_trace hp rt va ⊆ high_ptables}
```

Since we know the form of global mappings from `mmu_layout`, we can give a short, closed form of translation for addresses in `kernel_safe`:

```
k_phy_ad vp = Addr vp - offset
```

With these, we can formulate a theorem for assignments in kernel mode that do not touch any of the virtual memory data structures, i.e. when the write does not take place in any of the addresses covered by `kernel_data`.

Theorem 33. *If the* `mmu_layout` *invariants hold, we are in kernel mode, and we are performing a write in the kernel safe set that does not touch any MMU-relevant data structures, then the* `mmu_layout` *invariants are preserved and the effect is a simple heap update with known constant address translation.*

```
{|λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s ∧
     asids_consistent ∅ s ∧ [[lval]] s = ⌊vp⌋ ∧ [[rval]] s = ⌊v⌋ ∧
     Addr vp ∈ kernel_safe s ∧ k_phy_ad vp ∉ kernel_data_area s|}
lval ::= rval
{|λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s ∧
     asids_consistent ∅ s ∧ heap s (k_phy_ad vp) = ⌊v⌋|}
```

Proof. The safe set theorem for the assignment from the previous chapter (Theorem 30) gives us preservation of the kernel safe set and the fact that the write is safe and has the expected simple heap update semantics. The fact that the address is part of the kernel safe set and `mmu_layout` defines constant mappings for this set gives us the simple, closed address translation, and the fact that the write is outside any of the MMU-relevant data structures gives us the stronger fact that we do not add *any* entries to the inconsistent set, which allows us to re-establish `asids_consistent` after the write. The other components of `mmu_layout` are preserved the same way as in a user-level write: since none of the data structures change, they remain true in the new heap. □

This lemma covers kernel code that is uninteresting for the purposes of the MMU and TLB, which is the majority of code in a normal kernel. We will see an example of reasoning about a page table modification further below in Sect. 8.5.

8.4 Context Switch

We have so far shown reduction theorems for simpler reasoning when nothing interesting happens to the TLB. This section is the opposite: context switching. There are many ways for the OS to implement context switching — our example shows one where we change to a new address space, i.e. a new page table and ASID, without flushing the TLB, establishing the conditions of heap assignment (Theorem 32) for user-level reasoning.

Switching page table roots without TLB flushing is non-trivial, and the ARM architecture manual (ARM, 2008, Chapter B3.10) provides certain code sequences to achieve this. We have summarised these sequences in Chapter 3 while explaining OS kernel TLB management, and encourage the reader to find more details in Sect. 3.3. The manual provides these sequences, because speculative execution might otherwise contaminate the new ASID with mappings from the old page table, i.e. the TLB might still contain entries from the previous user, or the previous user might be contaminated with content from the new table. In this section, we show that our model is conservative for speculative execution, but precise enough so we can reason about these sequences and see why they are safe.

We choose the recommended sequence listed on Page 36 for our case study. This sequence switches to a new user-level page table and ASID by using a reserved ASID (in this case 0). It first switches to this reserved ASID, then sets the new page table root, then switches to the ASID for that root, before it switches to user mode. A real kernel would at this point also restore registers, which we omit.

Theorem 34. *The context switch sequence to a new ASID* `a` *and new page table root* `r` *preserves the* `mmu_layout` *and ASID snapshot consistency invariants and establishes the conditions for user-level reasoning, provided that the TLB has no inconsistent addresses at this point, that the reserved ASID 0 is not used for any user page table, and that that* `r` *is a known page table associated with ASID* `a`*.*

```
⦃λs. mmu_layout s ∧ asids_consistent ∅ s ∧ mode s = Kernel ∧
     𝓘𝓒 s = ∅ ∧ 0 ∉ ran (root_map s) ∧ root_map s (Addr r) = ⌊a⌋⦄
UpdateASID 0;; UpdateRoot (Const r);; UpdateASID a;; SetMode User
⦃λs. mmu_layout s ∧ 𝓘𝓒 s = ∅ ∧ mode s = User ∧ asids_consistent ∅ s⦄
```

Proof. Using the weakest-precondition rules from the logic for the commands `UpdateASID`, `UpdateRoot`, and `SetMode`, we see that the switch to ASID 0 means that any TLB entries that might be inconsistent between the current page table and `r` are only loaded under ASID 0. They do not affect ASID `a`, because the inconsistent set update only happens for the `UpdateRoot` instruction. The global mappings remain consistent even for ASID 0, since they are never changed. It is then safe to switch to ASID `a`, because no further page table modifications or page table root updates will happen. We know from assumption `asids_consistent` that the inconsistent set for ASID `a` is empty, establishing the $\mathcal{IC}$ part of the post condition.

Since we are switching to a known root and all other conditions of `mmu_layout` only depend on the heap, which this sequence does not modify, the `mmu_layout` invariant is also preserved, as is `asids_consistent`, because `assigned_asids_consistent` does not mention ASID 0 and `gset_consistent` holds for all the roots. □

For compiler correctness, we would additionally need to know that ASID 0 does not have inconsistent entries for the code and data areas of the kernel, which is maintained if ASID 0 is used only in the way above. To make this more explicit, we could add a static set to the program logic for code and data that must always be consistency, and the condition `asids_consistent` would maintain that at least the global kernel mappings are consistent in ASID 0.

8.5 Page Table Operations

In this final example we show the effect of updating the page table. There are a number of different scenarios in which the kernel might change a page table, usually to map a new page, change an existing mapping, or remove an existing mapping. The update could happen on the currently active page table or on one of the page tables for a currently inactive user, in which case it looks like a simple heap access, but would have TLB-relevant effects the next time we switch to that page table.

As the representative examples, we map and unmap a section, updating the first level of the current page table (the page directory). Mapping a section is a single heap update: writing one word that encodes the new page directory entry `pde` to the previous `InvalidPDE`'s location. Similarly, unmapping a mapped section is writing a word to its page directory entry to make it `InvalidPDE`. For these operations, the write is not outside the kernel data structures and does change virtual memory mappings. We observe that for mapping the section, we do not get TLB-inconsistent addresses, since our model allows transitions from `is_fault` to `no_fault` in the page table comparison. This is identical to the informal TLB maintenance reasoning OS engineers perform. While unmapping a section, we do get inconsistent addresses for the current ASID, i.e. there are now addresses the kernel must not access. However, since we are not writing to the kernel mappings, we still do know that everything in the kernel window remains safe, and we can delay a TLB flush until a later time, before we return to the user — we might for instance be in a loop to change multiple mappings in one kernel call.

Mapping a Section: We now present the theorem for mapping a section.

Theorem 35. *If the* `mmu_layout` *kernel invariants hold, and the address* `vp` *does not point to the encoding of a kernel mapping (*`Addr vp` $\in$ `kernel_safe s`*), if the physical address for* `vp` *is part of the current page table, if the heap at that address*

contains an invalid section entry, and if the new entry `pde` *is a section entry, then the heap access is safe, and there is no resultant TLB-inconsistency.*

```
⦃λs. mmu_layout s ∧
     mode s = Kernel ∧
     safe_set (kernel_safe s) s ∧
     Addr vp ∈ SM ∧
     SM = kernel_safe s ∧
     𝓘𝓒 s = ∅ ∧
     asids_consistent ∅ s ∧
     k_phy_ad vp ∈ pt_area s (root s) ∧
     heap s (k_phy_ad vp) = ⌊w⌋ ∧
     decode_pde w = InvalidPDE ∧
     decode_pde pde = SectionPDE base perms ∧ arm_p_nG perms = 1⦄
Const vp ::= Const pde ⦃λs. 𝓘𝓒 s = ∅ ∧ asids_consistent ∅ s⦄
```

Proof. The inconsistent set remains empty because the `ptable_comp` function does not track unmapped addresses. From injectivity of the `root_map` in `mmu_layout` and the condition that the current ASID is part of the `root_map`, we know that no other ASID overlaps it. Additionally, we know from the page table footprint condition that the memory area does not overlap with the `root_log` and `root_map`, or any other page table in the system. This allows us to conclude that all other ASIDs in `asids_consistent` remain consistent. From the fact that the write is in the safe set and not to a kernel mapping, we can conclude that the write is safe for the kernel to perform and that the safe set is preserved. □

We leave out the proof that the mapping update maintains the kernel invariants. It does of course do so, but the proof is mainly concerned with the technicalities of page table encoding and is not interesting for TLB reasoning.

Unmapping a Section: We now present the theorem for unmapping a mapped section.

Theorem 36. *If the* `mmu_layout` *kernel invariants hold, and the address* `vp` *does not point to the encoding of a global mapping (*`Addr vp ∈ kernel_safe s`*), if the physical address for* `vp` *is part of the current page table, if the heap at that address contains a section entry, and if the new entry* `pde` *is an* `InvalidPDE`*, then the heap access is safe, and the only inconsistency that we introduce is for the current ASID.*

```
⦃λs. mmu_layout s ∧
     mode s = Kernel ∧
     safe_set (kernel_safe s) s ∧
     Addr vp ∈ SM ∧
     SM = kernel_safe s ∧
     asids_consistent ∅ s ∧
     k_phy_ad vp ∈ pt_area s (root s) ∧
```

```
     heap s (k_phy_ad vp) = ⌊pde⌋ ∧
     (∃p perms. decode_pde pde = SectionPDE p perms) ∧
     decode_pde pde' = InvalidPDE⦄
Const vp ::= Const pde' ⦃λs. asids_consistent {asid s} s⦄
```

Proof. From injectivity of the `root_map` in `mmu_layout` and the condition that the current ASID is part of the `root_map`, we know that no other ASID overlaps it. Additionally, we know from the page table footprint condition that the memory area does not overlap with the `root_log` and `root_map`, or any other page table in the system. This allows us to conclude that all other ASIDs in `asids_consistent` remain consistent. From the fact that the write is in the safe set and not to a global mapping, we can conclude that the write is safe for the kernel to perform and that the safe set is preserved. □

As mentioned, since we still maintain the safe set, we can live with some (user-level) addresses being inconsistent and delay the flush instruction. It is straightforward to see that a flush for this specific ASID will re-establish consistency for all users.

Theorem 37. *An ASID flush for ASID* `a` *re-establishes the TLB consistency invariant for that ASID.*

```
⦃λs. asids_consistent {a} s ∧ mode s = Kernel⦄ Flush (flushASID a)
⦃λs. asids_consistent ∅ s⦄
```

Proof. The effect of this specific flush instruction is to remove all pairs from the inconsistent set that have ASID `a` as their first component, establishing the condition of `asids_consistent` for `a`. The consistency of other ASIDs follows directly from the precondition. □

Since flush instructions only make things safer, they trivially also maintain the rest of the kernel invariants.

Theorem 38. *ASID flushes maintain kernel invariants.*

```
⦃λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s⦄
Flush (flushASID a)
⦃λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s⦄
```

Proof. In all cases the inconsistent set is mentioned in the invariants (e.g. in the safe set), a smaller inconsistent set is safer, and the flush instruction reduces the set. □

In this example we have flushed the entire ASID, i.e. all addresses this user might have accessed in the past. The logic would also allow us to be more targeted and flush only precisely the addresses that were affected in this page table update.

This concludes the case study examples for our logic. We have seen that we can reason about user code, 'uninteresting' kernel code, and kernel code that manipulates paging structures, each at their appropriate level of abstraction.

8.6 Summary and Remarks

In this chapter, we have presented a case study that uses the seL4 microkernel as inspiration to distill out code sequences for a toy kernel that manages page tables and the TLB. We have extracted invariants and necessary conditions for correct TLB operation that mirror the informal reasoning of OS engineers. Our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations

CHAPTER

NINE

Conclusions

This chapter concludes the thesis, and is organised as: we summarise the novel contributions presented in the thesis towards verifying low-level programs under cached address translation. We then provide the proof effort of our modeling and reasoning framework. The thesis concludes with the future research and engineering directions.

9.1 Summary of Novel Contributions

In this thesis, we have presented a verified sound abstraction of the memory management unit of the ARMv7-A architecture including a two-stage TLB with address space identifiers (ASIDs) and global entries. We have used this abstraction as the underlying model to develop a logic for reasoning about low-level programs in the presence of cached address translation and demonstrated how the logic behaves in a number of examples.

This thesis has claimed the following novel contributions:

1. Formal modeling for ARMv7-A TLB in Isabelle/HOL,
2. Formal modeling for ARMv7-A MMU including the TLB,
3. Abstraction of multiple, increasingly complex MMU models using data refinement,
4. Hoare-style logic for program verification under cached address translation; and
5. Demonstrative case study of reasoning about low-level programs.

We have presented the above contributions in the following structure:

In Chapter 4, we have developed an operational model of the ARMv7-A memory management unit (MMU) including the TLB that caches entries *without* ASIDs. We have developed a base MMU model for such a TLB and have provided a series of refinements to stepwise abstract away the hardware details and to reach at an abstract MMU model that captures the essential TLB functionality and is easier to reason about.

In Chapter 5, we have extended the MMU model of Chapter 4 with the TLB caching page table entries under ASIDs and global tags. We have again built a refinement stack to abstract away the hardware details and also have explained how our refinement framework handles the added features.

In Chapter 6, we have formalised a separate page directory cache (PDC) to develop a two-stage TLB model caching the partial and complete page table walks. We have again built a refinement stack to abstract away the hardware details and have also explained how our refinement framework handles the additional PDC.

In Chapter 7, we have used the most abstract MMU model of Chapter 6 to de-

velop a logic for reasoning about low-level programs in the presence of TLB address translation. We have defined the syntax and semantics of a heap based language with necessary instructions for TLB management, we have then presented the Hoare logic rules for the operational semantics. We have also provided simplification rules for memory write to further facilitate the program reasoning in the presence of TLB effects.

In Chapter 8, we have applied the logic to extract invariants and conditions necessary to reason about the user-level and kernel-level executions, context switching and page table operations. This case study has shown that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching and page table manipulations.

9.2 Proof Effort

We now summarise the proof effort towards mechanising this thesis in Isabelle/HOL. The thesis implementation has two parts: the machine model and the logic.

The machine model includes the MMU specification, and is formalised at the ISA level of the ARMv7-A architecture. We have specified the MMU model as a type class in Isabelle/HOL, and have integrated this type class with the formal ARM ISA model. The type class has equipped us with a generic MMU interface, that we have instantiated to different MMU models varying in their TLB features. Using Isabelle's extensible records, we have extended the machine state with different TLB layouts to access these MMU instantiations. Using the overall setting of type class and record states, we have developed three machine models:

1. The MMU model with TLB caching entries without ASIDs:
   This machine model has the following four instantiations: the nondeterministic TLB, the deterministic TLB, the saturated TLB and the most abstract TLB.

2. The MMU model with TLB caching entries under ASIDs and global tags:
   This machine model has the following four instantiations: the nondeterministic TLB, the deterministic TLB, the saturated TLB and the most abstract TLB.

3. The MMU model with TLB and PDC caching entries under ASIDs and global tags:
   This machine model has the following three instantiations: the nondeterministic TLB, the saturated TLB and the most abstract TLB.

The machine model (excluding the updated ARM ISA theories, page table abstraction and bit-word libraries) consists of approximately 15k lines of Isabelle

code. It builds on existing large formalisations. Of these, the Cambridge ARM ISA model is approximately 50k, the page table abstraction is approximately 2.2k, and bit-word library is approximately 9.2k lines of Isabelle code.

The logic and the case study constitute the second part of thesis implementation. The logic, along with the abstract memory model and simplification rules, consists of approximately 1.4k lines of Isabelle code. The case study, including the modeling definitions and reasoning, has approximately 2k lines of Isabelle code. The resultant simplicity in the reasoning demonstrates the objective of the extensive refinement effort.

We have also spent considerable effort to generalise modeling and simplification theorems through out the formalisation. For example, the TLB model is type classed to have a generic TLB lookup function. Similarly, the page table walk function is defined generically to decode entries for different TLB configurations. The MMU operations are also grouped together for proving the refinement theorems collectively. All Isabelle/HOL theories for this thesis are available online (Syeda, 2019).

9.3 Comments on the TLB Modeling

We now comment on the TLB modeling presented in this thesis.

We have modeled ASID specific and global TLB entries, but currently do not treat locked (pinned) TLB entries. The modeling can easily be extended to include them: pinned TLB entries would have the effect of explicitly allowing inconsistency between the TLB and the page table, with the TLB taking preference.

Our logic does not address concurrency aspects — they are orthogonal. In a multi-core setting, each core has its own TLB which reads from global memory. Modifying a page table that is active on another core is almost never safe, unless the change merely adds new mappings or the change happens in the same safe set style presented here, where the execution on all cores must adhere to the intersection of all safe sets.

Weak memory and caches do have an interaction point with the TLB, because page table walks are subject to both and caches can be either virtually or physically indexed. We expect our safe set reasoning to transfer directly, requiring cache flushes and/or barrier instructions in addition to TLB flushes. We leave a cache formalisation for future work.

The model presented here does not have full fidelity for any specific ARM architecture version, but shows the principles to be applied for constructing such a model. If the intent is to reason on the ISA directly, a useful next step would be to lift the refinement theorems for the memory interface we have shown to the entire ISA

model. This is a mostly mechanical exercise, since the refinement theorems show equality for the effect of the memory operations on the state the rest of the ARM model cares about. If the intent is to reason about higher-level languages, we have laid the groundwork for compiler correctness in the presence of a TLB and the main reduction needed for a program logic: we know we only have to keep track of and avoid TLB-inconsistent addresses. All other low-level TLB complexity can be abstracted away.

9.4 Future Research and Engineering Directions

We now summarise the potential research and engineering directions.

Formal Reasoning about Software-Visible Hardware Components:
In this thesis, we have followed the idea of modeling software-visible hardware components and to reason about their OS kernel management. We have modeled the TLB, which is a software-visible hardware cache for address translation. However, the TLB is not the only hardware component that is managed by the OS kernel. Other examples include caches and system control registers.

Similar to the TLB management, the OS kernel is responsible for cache management at the ISA level. Caches classify themselves into data and instruction caches. Data cache management is conceptually similar to TLB management: the OS kernel is required to flush data caches to maintain their coherency, and it is required to use barriers for propagating write effects across memory models. We propose a modeling similar to this thesis for reasoning about the functional correctness of data cache management. The instruction cache is interesting, because the contents of the instruction cache remain static during the code execution unless the processor is executing self-modifying code. In the latter case, the OS kernel might be required to manage the instruction caches, and this management can also be verified using the modeling and refinement techniques presented in this thesis.

Similarly to the ASID and TTBR0 registers of the ARM architecture, we can reason about operations causing changes to the system control register and hence requiring the system management by the OS kernel.

The scope of reasoning about software-visible hardware components extends further than proving functional correctness. For example, once we have modeled the hardware features of caches, we can reason about the possibility of timing channels. We can capture the essential microarchitecture details causing the timing channels, can abstract these details to obtain only their effects, and can reason to avoid or solve the timing channels. Heiser et al. (2019) are proposing to use ideas similar to this thesis for reasoning about time protection mechanisms.

The same line of reasoning can be used for modeling defective hardware, and enforcing the hardware-software contract. For example, the TLB model of this

thesis can capture the effect of attacks such as Meltdown (Lipp et al., 2018) which exploits the fact that permission bits of TLB entries are not checked during speculative execution on some platforms, and uses a cache side channel to thereby make kernel-only TLB mappings readable to user space. To conservatively formalise the effect of this attack, one could change the model to ignore read restrictions in TLB entries. A system that can be proved safe under that conservative model, should then be safe under Meltdown.

Closing the TLB Management Assumption in the seL4 Verification:
In this thesis, we have crafted an infrastructure for validating the TLB management assumption in the seL4 verification. We now briefly outline how we can incorporate this infrastructure into the seL4 verification. The proposed steps are:

1. We would begin by adding the most abstract TLB model to the translation validation/binary verification step of the seL4 verification, this would make sure we are forced to prove all necessary side conditions.
2. These side conditions would demand the inclusion of code, stack, and globals area in the safe set.
3. In the next layer up (C verification), we would have to prove the additional side conditions generated by the translation validation; potentially still based on a TLB model like in Chapter 7.
4. The proof of the additional TLB side conditions would make use of invariants proved on higher levels, e.g. that all memory accesses are within the safe set, and in particular that the kernel never writes to global mappings, and flushes correctly before returning to user.
5. The upper refinement layers would not have an explicit TLB model, but they are now required to produce the invariants necessary for the refinement to succeed. Almost all of these invariants are already proved.

With these suggestions, we leave the seL4 TLB validation as a future engineering project.

Application of Reasoning Methodology to other Architectures:
The model and case study presented in this thesis use the ARMv7-A architecture, but our interface to page table encodings is generic and should apply to all architectures with conventional multi-level page tables. The details of TLB maintenance may differ between architectures, but the ideas of the model should again transfer readily. We now outline the engineering roadmap for applying the TLB reasoning framework to the x86, RISC-V and MIPS architectures, as well as ARMv7-R and ARMv7-M profiles.

TLB Management in the x86 Architecture: The x86 architecture supports multi-level page tables, most 32-bit implementations use 2-level page tables, while

64-bit implementations usually scale up to 4-level page tables. The TLB is implemented stage-wise to cache complete and partial page table walks, and the architecture supports ASID-specific, global and pinned TLB entries. On a TLB miss, the hardware does the page table walk and reloads the respective stages of the TLB with translation entries. Similar to the ARM architecture, the OS kernel is responsible for invalidating TLB entries after updating the page tables. The hardware flushes the TLB automatically during a context switch; therefore the OS kernel is not required to flush the TLB or to follow specific code sequences to avoid TLB flushes, as in for the ARM architecture.

The MMU modeling presented in Chapters 4 to 6 can readily scale to the above mentioned MMU setting for the x86 architecture. We can increase the number of levels in the abstract page table interface, and can model the multi-stage TLB as we have modeled the two-stage TLB in Chapter 6. We can then formalise memory and MMU operations such as TLB flushing. Once we have the base MMU model, we can apply stepwise data refinement to abstract away the hardware details and to extract a page table comparison function for TLB-related operations, as we have done in Chapters 4 to 6 for the ARM architecture. The instructions of a heap based language would be identical, the semantics would now take the updated abstract model into account, and the case study reasoning will readily apply to the x86-based MMU layout. As the OS kernel does not have to flush the TLB in this case, we expect even simpler context switching reasoning to that of the ARM architecture.

TLB Management in the RISC-V Architecture: The RISC-V architecture allows a configurable number of page table levels. The TLB can be implemented in single or multi-stage, with ASID-specific, global and pinned entries. The RISC-V architecture specifies TLB flushing as a fence instruction for cleaner semantics. The architecture emphasises on a hardware loaded TLB, but an implementation can choose to implement software TLB refills using a machine-mode trap handler.

The main steps of our TLB reasoning framework are: TLB and MMU modeling, refinement of the machine model, logic on top of the abstract MMU model, and the case study for a specific MMU layout. These generic steps are equally applicable to the OS kernel management for the RISC-V architecture. The roadmap for this verification is similar to the one outlined for the x86 adaptation.

TLB Management in the MIPS Architecture: The MIPS architecture leaves the page table implementation, TLB refills and eviction, and page table walks to the OS kernel. The OS kernel typically implements ASID-specific and global TLB entries. The relevant kernel traps handle the address translation errors and TLB misses. The OS kernel can also flush the TLB during a context switch.

Given the MIPS-TLB as a software-maintained cache, we can formulate a nicer formal model to begin with, avoiding the hardware details. We can also apply the refinement and again compute an address comparison function as the abstract TLB model. The logic and case study will be applicable straightforwardly.

TLB Management in ARMv7-R and ARMv7-M Profiles: The ARMv7-R and ARMv7-M profiles use protected memory system architecture (PMSA). The PMSA has control registers in a memory protection unit (MPU) instead of page tables; which means that it does not have the non-deterministic behaviour introduced by potential TLB misses.

TLB Management in the ARMv8-A Architecture: The TLB functionality and the kernel's TLB management are functionally similar in the ARMv7-A and ARMv8-A architectures. The ARMv8-A architecture provides more insights though, for example, it recommends specific *break-before-make* sequences for the kernel to avoid TLB conflicts. Again, both the modeling and reasoning framework of this thesis are scalable to the reasoning about TLB management in the ARMv8-A architecture.

9.5 Final Remarks

In this thesis, we have developed multiple and increasingly complex MMU models and their refinement stacks for the ARMv7-A architecture in Isabelle/HOL. We have also developed a program logic for reasoning about low-level programs in the presence of cached address translation, with a multi-stage TLB, ASIDs, and global entries. The logic allows us to prove reduction theorems that mirror the informal reasoning OS engineers perform when they write kernel code. It also allows us to drop into a simpler setting when we reason about code that does not affect virtual memory mappings. In these cases, we only need to show that memory accesses are within a set of safe addresses. Our work shows that reasoning in the presence of a TLB does not need to be significantly more onerous than without.

Bibliography

The Coq proof assistant. http://coq.inria.fr. Accessed: April 2019.

The HOL4 proof assistant. http://hol.sourceforge.net/. Accessed: April 2019.

Achermann, R, Humbel, L, Cock, D, and Roscoe, T. Physical addressing on real hardware in Isabelle/HOL. In Avigad, J and Mahboubi, A, editors, *Interactive Theorem Proving*, pages 1–19, Cham, 2018. Springer International Publishing.

Alkassar, E, Schirmer, N, and Starostin, A. Formal pervasive verification of a paging mechanism. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 109–123. Springer Berlin Heidelberg, 2008.

Alkassar, E, Cohen, E, Hillebrand, M, Kovalev, M, and Paul, W. J. Verifying shadow page table algorithms. In *Formal Methods in Computer Aided Design*, pages 267–270, Oct 2010.

Alkassar, E, Hillebrand, M. A, Paul, W, and Petrova, E. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments*, pages 40–54. Springer Berlin Heidelberg, 2010.

Alkassar, E, Cohen, E, Kovalev, M, and Paul, W. J. Verification of TLB virtualization implemented in C. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 209–224, Philadelphia, PA, USA, Jan 2012.

ARM Architecture Reference Manual, ARMv7-A and ARM v7-R. ARM Ltd., Apr 2008. ARM DDI 0406B.

ARM Cortex-A15 MPCore Processor Technical Reference Manual. ARM Ltd., June 2013. ARM DDI 0438I.

Barthe, G, Betarte, G, Campo, J. D, and Luna, C. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *25th CSF*, pages 186–197, 2012.

Baumann, C, Schwarz, O, and Dam, M. Compositional verification of security properties for embedded execution platforms. In *PROOFS 2017. 6th International Workshop on Security Proofs for Embedded Systems*, volume 49 of *EPiC Series in Computing*, pages 1–16. EasyChair, 2017.

Bevier, W. R. Kit: a study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, Nov 1989. ISSN 0098-5589. doi: 10.1109/32.41331.

Bolignano, P, Jensen, T, and Siles, V. Modeling and abstraction of memory management in a hypervisor. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*, pages 214–230. Springer-Verlag New York, Inc., 2016.

Boyer, R. S and Moore, J. S. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988. ISBN 0-12-122952-1.

Dam, M, Guanciale, R, Khakpour, N, Nemati, H, and Schwarz, O. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 223–234. ACM, 2013.

Daum, M, Billing, N, and Klein, G. Concerned with the unprivileged: User programs in kernel refinement. *Formal Aspects Comput.*, 26(6):1205–1229, Oct 2014.

Fox, A and Myreen, M. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st ITP*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010.

Gu, L, Vaynberg, A, Ford, B, Shao, Z, and Costanzo, D. CertiKOS: A certified kernel for secure cloud computing. In *2nd APSys*, 2011.

Gu, R, Shao, Z, Chen, H, Wu, X, Kim, J, Sjöberg, V, and Costanzo, D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association.

Harrison, J. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

Heiser, G, Klein, G, and Murray, T. C. Can we prove time protection? *CoRR*, abs/1901.08338, 2019. URL http://arxiv.org/abs/1901.08338.

Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

Khakpour, N, Schwarz, O, and Dam, M. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs*, pages 276–291. Springer International Publishing, 2013.

Klein, G, Elphinstone, K, Heiser, G, Andronick, J, Cock, D, Derrin, P, Elkaduwe, D, Engelhardt, K, Kolanski, R, Norrish, M, Sewell, T, Tuch, H, and Winwood, S. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009.

Klein, G, Andronick, J, Elphinstone, K, Murray, T, Sewell, T, Kolanski, R, and Heiser, G. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

Klein, G, Andronick, J, Fernandez, M, Kuz, I, Murray, T, and Heiser, G. Formally verified software in the real world. *Commun. ACM*, 61(10):68–77, Sept. 2018. ISSN 0001-0782.

Kolanski, R. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, UNSW, Sydney, Australia, Jul 2011. Available from publications page at http://ts.data61.csiro.au/.

Kolanski, R and Klein, G. Mapped separation logic. In *Verified Software: Theories, Tools, Experiments*, pages 15–29. Springer Berlin Heidelberg, 2008.

Kolanski, R and Klein, G. Types, maps and separation logic. In *TPHOLs*, pages 276–292, Munich, Germany, Aug 2009.

Kovalev, M. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, Germany, 2013.

Liedtke, J. Toward real microkernels. *Commun. ACM*, 39(9):70–77, Sept. 1996. ISSN 0001-0782.

Lipp, M, Schwarz, M, Gruss, D, Prescher, T, Haas, W, Mangard, S, Kocher, P, Genkin, D, Yarom, Y, and Hamburg, M. Meltdown. *ArXiv e-prints*, 1801.01207, Jan. 2018.

Lutsyk, P. *Correctness of Multi-core Processors with Operating System Support*. PhD thesis, Saarland University, Saarbrücken, Germany, 2018. URL https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27182.

MIPS Architecture For Programmers, MIPS32 and microMIPS32 Privileged Resource Architecture. MIPS Technologies, July 2015.

Morgan, C. *Programming from Specifications (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994. ISBN 0-13-123274-6.

Naraschewski, W and Wenzel, M. Object-oriented verification based on record subtyping in higher-order logic. In *11th TPHOLs*, volume 1479 of *LNCS*, pages 349–366, Canberra, Australia, Sep 1998.

Nemati, H, Dam, M, Guanciale, R, Do, V, and Vahidi, A. Trustworthy memory isolation of Linux on embedded devices. In Conti, M, Schunter, M, and Askoxylakis, I, editors, *Trust and Trustworthy Computing*, pages 125–142. Springer International Publishing, August 2015a.

Nemati, H, Guanciale, R, and Dam, M. Trustworthy virtualization of the ARMv7 memory subsystem. In *41st SOFSEM*, volume 8939 of *LNCS*, pages 578–589, Jan 2015b.

Neumann, P. G and Feiertag, R. J. PSOS revisited. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 208–216, Dec 2003. doi: 10.1109/CSAC.2003.1254326.

Ni, Z, Yu, D, and Shao, Z. Using XCAP to certify realistic systems code: machine context management. In Schneider, K and Brandt, J, editors, *Theorem Proving in Higher Order Logics*, pages 189–206. Springer Berlin Heidelberg, 2007.

Nipkow, T and Klein, G. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014. ISBN 3319105418, 9783319105413.

Nipkow, T, Paulson, L, and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Paul, W, Rieden, T. I. d, and Broy, M. The Verisoft XT project. http://www.verisoftxt.de, 2010. Accessed: April 2019.

Roever, W.-P. d and Engelhardt, K. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1st edition, 2008. ISBN 9780521103503.

Stallings, W. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, 6th edition, 2008.

Syeda, H. T. Isabelle/HOL program logic for cached address translation. https://github.com/SEL4PROJ/tlb, 2019.

Syeda, H. T and Klein, G. Reasoning about translation lookaside buffers. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 490–508. EasyChair, 2017.

Syeda, H. T and Klein, G. Program verification in the presence of cached address translation. In *Interactive Theorem Proving*, pages 542–559, Cham, 2018. Springer International Publishing.

Syeda, H. T and Klein, G. Formal reasoning under cached address translation. In *Journal of Automated Reasoning (JAR), Special Edition ITP2018*, 2019. submitted, under Review.

Tanenbaum, A. S and Bos, H. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014. ISBN 013359162X, 9780133591620.

Walker, B. J, Kemmerer, R. A, and Popek, G. J. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, Feb. 1980. ISSN 0001-0782. doi: 10.1145/358818.358825. URL http://doi.acm.org/10.1145/358818.358825.