Een virtualisatieraamwerk voor ingebedde systemen

A Virtualisation Framework for Embedded Systems

Niels Penneman

Promotoren: prof. dr. ir. K. De Bosschere, prof. dr. ir. B. De Sutter
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015 - 2016

UNIVERSITEIT
GENT

ISBN 978-90-8578-861-4
NUR 980
Wettelijk depot: D/2015/10.500/105

Brevis esse laboro, obscurus fio.
– Q. Horatii Flacci *Ars Poetica*.

Acknowledgements

This book tells the story of a journey that started over 6 years ago, when I was still completing my Masters degree at the Vrije Universiteit Brussel. I had gathered a fair bit of knowledge on the basics of electrical engineering in my Bachelor studies, but my all-time passion had driven me to pursue a Master in Applied Computer Science Engineering instead. Striving to combine my knowledge of both domains, I quickly gained interest in embedded systems, partly thanks to a course on operating systems taught by prof. Martin Timmerman. I was lucky to have a flexible curriculum that allowed me to pick several elective courses on topics related to embedded systems in the final year of my Masters programme. It was in one of those courses that I met prof. Bjorn De Sutter from Ghent University, whom I had very interesting discussions with. Close to the end of his course, we talked about research opportunities in the fields of compilation and virtualisation techniques for embedded systems.

I had been interested in virtualisation in the context of desktop systems and cloud computing for a while, and the thought of using virtualisation techniques on embedded systems was relatively new to me. I had previously considered cloud computing as a topic for my Masters thesis, but I eventually chose an entirely different subject. I realised that the limited amount of time given to complete such a thesis would be too short for me to create anything practically useful—and I really wanted to *create* something that was preferably both novel and useful.

Despite my interest in virtualisation, I was not immediately convinced that pursuing a Ph.D. was the right choice for me. My doubts, however, quickly vanished after a motivational talk with prof. Koen De Bosschere at Ghent University, and so I plunged into this adventure. Both Bjorn and Koen became my supervisors, and over the years this tandem has worked rather well. They helped me to obtain a grant for my research, and they have provided me with guidance and motivation in

times of need. I would therefore like to thank my supervisors, Bjorn and Koen, for their continued efforts. I would also like to thank prof. Dirk Vermeir of the Vrije Universiteit Brussel, and prof. Martin Timmerman of the Royal Military Academy, for providing me with pointers and insights that sparked my interest in embedded systems and virtualisation.

In the early days, when I was still determining my research topic, I met prof. Alasdair Rawsthorne from The University of Manchester for a discussion on embedded virtualisation. Little did I know at that time that this would be the start of our cooperation, which has lasted for several years, and through which I obtained the basic knowledge on hypervisors. I was later introduced to two of his students in Manchester, Danielius Kudinskas and Alex Merrick, who started a Bachelor project to virtualise ARM-based embedded systems using dynamic binary translation. During the next few years I worked together with Danielius to create the hypervisor that would form the basic platform for my research. I would therefore like to thank prof. Alasdair Rawsthorne, Danielius Kudinskas and Alex Merrick for their valuable input and contributions.

Furthermore, I would also like to thank the other members of my examination board for their time and their valuable feedback: dr. Jonas Maebe, prof. Wilfried Philips and prof. Luc Taerwe. Additionally I would like to thank Luc Perneel for his detailed reviews of my work.

The first four years of my research were funded by the agency for Innovation by Science and Technology (IWT) by means of a personal research grant. Afterwards, I contributed to and obtained results in the context of the EURO-MILS project, which was funded by the European Union's Seventh Framework Programme under grant agreement number ICT-318353. I would therefore like to thank the IWT and all people involved in getting the EURO-MILS project get off the ground.

Throughout those years, several Master students have contributed to my research project as a part of their thesis: Sarah Tawfik Adel El Shal from the Vrije Universiteit Brussel [51]; Peter Van Bouwel [129], Henri De Veene [43] and Jens Van den Broeck [130] from Ghent University; and Markos Chandras [31] and Cosmin Gorgovan from The University of Manchester. Even though not all of their contributions made it into the final version of the hypervisor developed as part of my research, they have provided me with useful insights. Guiding the students in Brussels and Ghent was a valuable experience for me. Additional thanks go out to Wim Meeus for his assistance in guiding Jens, and to prof. Pieter

Rombouts for providing us with the necessary equipment.

I would also like to thank ARM for providing the high-end development tools that made my life a lot easier when developing and debugging the hypervisor. ARM has also provided excellent support for their tools, from helping to get them to cooperate with officially unsupported hardware, to responding to and ultimately implementing our feature requests, even though we were only a small customer on a discounted contract.

While I have spent most of my time working alone, I was happy to find myself surrounded by a number of colleagues that made my work more bearable. Panagiotis, no one has given me as much motivation to go on as you did. Over the years we have become good friends, and we have had fun together both in Ghent and in several places abroad. You have inspired me on several occasions and in several activities unrelated to work, from photography to our workouts at the gym. Even though you will soon be returning to Greece, I hope that we will keep in touch.

Christophe and Jeroen, it was fun to have you around, and to cruise through Greece together to attend Panagiotis' wedding. I would also like to thank you for leaving our office mostly empty until noon, so that I could work without disturbances (and play music on the big speakers).

Bart, the same goes for you. As you started your Ph.D. a few years before me, you were a reliable source of non-technical information concerning the whole process. Your experience came in handy to fix my uncertainties while going through the final phases of my Ph.D.

Jonas, thanks again for sharing your experiences with me on the ARM architecture from the very beginning, and for staying calm even in the most frustrating situations we had in the EURO-MILS project.

Tim and Jens, we have spent a lot of time together with Panagiotis to teach and prepare introductory labs on electronics. At times this was fun, but we also shared quite some frustrations. Luckily, we all managed to keep our sanity, because we made a great and competent team that did not struggle to divide the workload—unlike some of our students.

Stijn, thanks for your input on low-level programming and reverse engineering. I have learnt a few useful things in your ethical hacking sessions. Bert, Farhadur, Hadi, Ronald, Peng, Sander and Wim, even though we had less interactions, it was nice to have you around nonetheless.

Marnix, you helped us to set up the lab, you kept track of equipment orders, and so much more. I am still amazed at how much work you

manage to do in one day. Together with Vicky, you took a lot of administrative work out of my hands. Thanks for all your efforts. Ronny, Michiel, and Klaas, thanks for being there whenever we had a problem with our IT infrastructure and those dreadful Drupal-based websites.

At the end of February 2015, I left Ghent University to start working for Televic. I was still finalising my thesis and my second journal paper. It was not an obvious task to do this after normal working hours, but luckily I ended up working in a stimulating environment, with people that provided me with moral support to finish my Ph.D. I would therefore also like to thank my colleagues and former colleagues at Televic.

Pursuing a Ph.D. is not a nine-to-five job. We were all given the freedom to start and stop working whenever we wanted. We could work from home if we wanted. In the end, though, we needed to deliver results so that we could eventually complete our research. We were given an informal deadline of four years, as most research grants—at least in our domain—typically cover four years. When you are working on a personal grant, you are mostly working for yourself. The more you get done in a day, the closer you get to finishing your Ph.D. It does not matter when you step outside the office and close the door behind you. Your research is stuck in the back of your mind, and you take it with you wherever you go. You go to bed with it, and you wake up with it. It changes you and the way you interact with the people close to you.

My parents have, without doubt, taken the majority of my frustrations. They have nevertheless motivated me to continue over and over again. They have done a lot for me only so that I would have more time to work on my research. Without their support, I would never have managed. I would also like to thank Mark and Magda. Having them around is almost like having an extra pair of parents—even though they have quite a different view on life, but that is part of the fun. I thank them for supporting me, for proofreading, and for everything else! Further thanks go to Martine and Jan, because they made time to support me and to have fun together even in difficult circumstances.

I would also like to thank my friends for bearing with me and supporting me up until today. I will not try to provide an exhaustive list of all their names. You know who you are, and please understand that I have really appreciated all your support. Special thanks go out to Magalie, for all the fun she has brought me on our time together in South Africa, the

Netherlands, and here in Belgium, and for the support she has given me whenever I needed it, even when we could not physically be together. Karolien, thank you for being there whenever I needed someone to talk to. Davy, Thierry, Jens, Freija, Bruno, Farid, Ilse, ... you all helped me to keep my sanity throughout this journey. Thank you!

Niels Penneman
Ghent, 7 December 2015

Examencommissie

Prof. Luc Taerwe, *voorzitter*
Vakgroep Bouwkundige Constructies
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Jonas Maebe, *secretaris*
Vakgroep Elektronica en Informatiesystemen
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Koen De Bosschere, *promotor*
Vakgroep Elektronica en Informatiesystemen
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Bjorn De Sutter, *promotor*
Vakgroep Elektronica en Informatiesystemen
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Wilfried Philips
Vakgroep Telecommunicatie en Informatieverwerking
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Em. Prof. Alasdair Rawsthorne
School of Computer Science
Faculty of Engineering and Physical Sciences
The University of Manchester

Prof. Martin Timmerman
Departement Wiskunde
Faculteit Polytechniek
Koninklijke Militaire School

Leescommissie

Prof. Wilfried Philips
Vakgroep Telecommunicatie en Informatieverwerking
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Em. Prof. Alasdair Rawsthorne
School of Computer Science
Faculty of Engineering and Physical Sciences
The University of Manchester

Prof. Martin Timmerman
Departement Wiskunde
Faculteit Polytechniek
Koninklijke Militaire School

Samenvatting

Systeemvirtualisatie ontkoppelt een besturingssysteem van de hardware van een fysieke machine door middel van een hypervisor. Het besturingssysteem wordt dan ook een *gast* genoemd. Systeemvirtualisatie heeft al verschillende toepassingen gevonden in de desktop- en serverwereld, gaande van het reduceren van de operationele kosten van datacenters door het consolideren van servers, tot het vergemakkelijken van softwareontwikkeling en het emuleren van software voor verouderde hardware. Ingebedde systemen worden alsmaar krachtiger, en hun hardware evolueert sneller dan hun software. Daarom worden deze systemen nu ook interessant om te virtualiseren. De bestaande technieken voor servers en desktops kunnen echter niet zomaar worden gebruikt voor ingebedde systemen, omdat deze vaak andere vereisten, andere toepassingen en andere architecturen hebben dan servers en desktops.

ARM is met ruime voorsprong de marktleider voor processors van ingebedde systemen en in het bijzonder van mobiele toestellen. In het voorbije decennium werden reeds verschillende hypervisors ontwikkeld voor de ARM-architectuur. Omdat deze architectuur echter geen ondersteuning kon bieden voor volledige virtualisatie, steunt de meerderheid van de bestaande hypervisors op paravirtualisatie, een techniek met veel nadelen. ARM heeft onlangs zijn ARMv7-A-architectuur uitgebreid met hardware-ondersteuning voor volledige virtualisatie. Hoewel hypervisors die van deze uitbreidingen gebruik maken niet dezelfde nadelen met zich meebrengen als hypervisors gebaseerd op paravirtualisatie, kunnen ze niet werken op de meerderheid van de ARM processors die momenteel in omloop zijn, omdat deze processors nog niet beschikken over de uitbreidingen voor volledige virtualisatie. Softwaregebaseerde technieken zoals dynamisch binaire vertaling (DBV) bieden een alternatieve oplossing om architecturen die vanuit de hardware geen volledige virtualisatie ondersteunen toch volledig te virtualiseren: problematische

instructies in de binaire code van de gastbesturingssystemen worden tijdens het uitvoeren herschreven. DBV is veelzijdiger dan hardware-uitbreidingen voor volledige virtualisatie, aangezien het inherent heel wat meer toepassingen biedt, gaande van het optimaliseren tussen besturingssystemen en applicaties, het emuleren en optimaliseren van verouderde softwarestapels, het beïnstrumenteren en testen van volledige softwarestapels tot zelfs het optimaal verdelen van de werklasten in heterogene multikernprocessors door het vertalen van binaire code over de verschillende architecturen heen.

De belangrijkste onderzoeksvraag die we in deze doctoraatsthesis wensen te behandelen is: welke technologische uitdagingen moeten overwonnen worden om deze toepassingen van DBV mogelijk te maken op de ARMv7-A architectuur? Als eerste stap om een antwoord te formuleren op deze vraag, onderzoeken we welke beperkingen de architectuur oplegt die volledige virtualisatie belemmeren. We stellen vervolgens generieke softwaretechnieken voor om de processor en het geheugenbeheer volledig te virtualiseren zonder gebruik te maken van de nieuwe hardware-uitbreidingen voor de ARMv7-A-architectuur. Voor de virtualisatie van het geheugenbeheer zullen we ons onderzoek toespitsen op technieken die gebruik maken van schaduwpaginatabellen.

We beginnen bij de theorie van klassieke virtualiseerbaarheid van Popek en Goldberg. We breiden hun theorie uit voor moderne RISC-architecturen (reduced instruction set computer) en moderne toepassingen: we voegen formalismen toe voor virtueel geheugen met paginatabellen, invoer en uitvoer, en onderbrekingen. We gebruiken vervolgens deze uitgebreide theorie om te analyseren waarom de oorspronkelijke ARMv7-A-architectuur niet klassiek virtualiseerbaar is, en hoe de recente hardware-uitbreidingen van ARM de architectuur wél virtualiseerbaar maken. Vertrekkend vanuit het standpunt dat hypervisors gebaseerd op dynamisch binaire vertaling in principe een evolutie zijn van de ideeën van Popek en Goldberg over hybride virtualisatie, tonen we aan hoe onze uitgebreide theorie helpt bij het construeren van een dynamisch binaire vertaler voor volledige systeemvirtualisatie.

We gebruiken onze analyse als basis voor het ontwerp van de *STAR hypervisor*, een nieuwe bare-metal hypervisor voor de ARMv7-A-architectuur. Onze hypervisor is de eerste open source hypervisor voor de ARMv7-A-architectuur dat een systeem volledige kan virtualiseren door uitsluitend gebruik te maken van softwaretechnieken. Besturingssystemen kunnen zonder wijzigingen als gast uitgevoerd worden bovenop

deze hypervisor, volledig afgeschermd van de fysieke CPU door DBV, en van de fysieke MMU door middel van schaduwpaginatabellen.

Bestaand werk omtrent DBV voor de ARM-architectuur richt zich vooral op procesvirtualisatie. Systeemvirtualisatie heeft zijn eigen unieke uitdagingen, waardoor technieken voor procesvirtualisatie niet zomaar kunnen worden overgenomen in een hypervisor voor systeemvirtualisatie. Dynamisch binaire vertalers voor procesvirtualisatie gaan er vaak van uit dat de applicaties die gevirtualiseerd worden zich steeds correct gedragen. Bovendien is het vaak niet belangrijk om de vertaler en de vertaalde applicatie strikt van elkaar te scheiden. Bij volledige systeemvirtualisatie is een correcte vertaling echter essentieel om de afscherming tussen de gast, de hypervisor, en de fysieke hardware te garanderen. De hypervisor kan geen enkele aanname maken over het gedrag van een gastbesturingssysteem, omdat de code van besturingssysteemkernen vaak handgeschreven instructiesequenties en gespecialiseerde systeeminstructies bevatten die bijzondere aandacht vereisen bij de vertaling.

We bestuderen hoe we de uitdagingen aan DBV voor volledige systeemvirtualisatie moeten aanpakken, uitgaande van een fundamenteel probleem bij de vertaling van binaire code: onze vertalingen hebben vaak een extra register nodig om tijdelijk informatie op te slaan, maar het is niet altijd mogelijk om een register te vinden dat vrij kan worden gebruikt. Onze vertaler moet daarom een register vrijmaken door tijdelijk de waarde van dat register elders op te slaan, en na gebruik deze waarde terug te herstellen. De opslagplaats moet bovendien toegankelijk zijn voor de vertaalde code van het gastbesturingssysteem. Dit probleem is heel eenvoudig op te lossen bij procesvirtualisatie. Bij systeemvirtualisatie is dit echter veel moeilijker, mede omdat de hypervisor zoveel mogelijk zijn eigen geheugen moet afschermen van het gastbesturingssysteem. Bovendien kan de hypervisor geen aannames maken over de inhoud van het geheugen van het gastbesturingssysteem. We stellen nieuwe oplossingen voor die deze problematiek aanpakken, en we evalueren hun impact op de uitvoeringstijd van vertaalde code.

Daarna bepalen we hoe we de vertaalde code verder kunnen optimaliseren. We trachten de oorzaak van de kosten verbonden aan de vertaling te identificeren aan de hand van metingen met micro-benchmarks uit van de lmbench suite. Deze benchmarks staan model voor typische interacties tussen de gebruikersapplicaties en de besturingssysteemkern voor Linux-gebaseerde besturingssystemen. We bestuderen welke instructies het vaakst een tussenkomst van de hypervisor vergen in vertaalde code

in elk van deze benchmarks. Vervolgens stellen we vertaaltechnieken voor die deze tussenkomsten zo veel mogelijk elimineren, en we evalueren de impact van elke nieuwe techniek op elke benchmark afzonderlijk. Zoals wel vaker het geval is bij DBV, valt op dat instructies die het controleverloop bepalen voor de grootste vertraging in uitvoeringstijd zorgen. Voor deze instructies kunnen bestaande technieken worden aangewend, met significante prestatieverbeteringen tot gevolg. We merken echter op dat onze optimalisaties specifiek voor volledige systeemvirtualisatie de resterende vertraging nog met gemiddeld 51% kunnen reduceren.

We bekijken vervolgens de impact van onze vertaaltechnieken op echte applicaties. We voeren benchmarks uit van de mibench suite die het gedrag van realistische applicaties modelleren. We vergelijken de uitvoeringstijd van deze benchmarks op een ongevirtualiseerd systeem, met de uitvoeringstijd op de oorspronkelijke versie van onze vertaler, en de uitvoeringstijd op de geoptimaliseerde vertaler. Onze metingen tonen aan dat applicaties tot 5 keer vertraagd worden door de oorspronkelijke vertaler. De geoptimaliseerde vertaler beperkt de vertraging van realistische applicaties tot slechts 39% in het slechtste geval.

De vertaler van de hypervisor staat enkel in voor CPU-virtualisatie. De hypervisor moet daarnaast ook nog het geheugenbeheer, de fysieke caches en de translation lookaside buffers (TLBs) virtualiseren. We beschrijven welke technieken we gebruiken om het geheugenbeheer te virtualiseren door middel van schaduwpaginatabellen, en hoe we cacheoperaties virtualiseren. We zien onder andere dat cacheoperaties van gastbesturingssystemen niet zomaar kunnen uitgevoerd worden op de fysieke caches, omdat de hypervisor hierdoor gegevens kan verliezen.

Onze hypervisor maakt gebruik van diverse softwarecaches, zowel in de vertaler als in de virtualisatie van het geheugenbeheer. De gegevens in deze softwarecaches moeten ten allen tijde overeenkomen met de interne toestand van het gastbesturingssysteem waarop ze betrekking hebben. De hypervisor kan dit probleem op twee verschillende manieren aanpakken. Een eerste aanpak steunt op geheugenbeheer: de interne toestand van de gast die interessant is voor het beheer van de caches wordt beschermd tegen schrijfoperaties van de gast, zodat de hypervisor weet wanneer de gast zijn interne toestand wijzigt. Een tweede aanpak koppelt het beheer van de softwarecaches aan hoe de gast de hardwarecaches probeert aan te sturen. De softwarecaches worden dan als het ware een gevirtualiseerde hardwarecache voor de gast.

De caches van de vertaler moeten bijgewerkt worden telkens wanneer

een gast zijn code bewerkt, verplaatst of verwijdert. De caches van het gevirtualiseerd geheugenbeheer, nl. de schaduwpaginatabellen, moeten op gelijkaardige manier worden onderhouden: telkens wanneer een gast zijn eigen paginatabellen aanpast, moet onze hypervisor de schaduwpaginatabellen ook aanpassen. Aangezien we een lui mechanisme gebruiken om de schaduwpaginatabellen in te vullen, hoeven we enkel die delen te onderhouden die reeds ingevuld werden.

We hebben zowel voor het beheer van de caches van de vertaler als voor de schaduwpaginatabellen de twee beschreven technieken geïmplementeerd en geëvalueerd. Uit onze metingen concluderen we dat we het beheer van de schaduwpaginatabellen op ARM best koppelen aan het TLB-beheer van de gast. De caches van de vertaler worden echter beter beheerd door gebruik te maken van geheugenbescherming. Dit is een rechtstreeks gevolg van de manier waarop de fysieke TLB en caches georganiseerd zijn in de ARMv7-A-architectuur.

Tijdens ons onderzoek hebben we een virtualisatieplatform ontwikkeld voor de ARMv7-A-architectuur, waarop alle bovenvermelde technieken geïmplementeerd en geëvalueerd werden. We hebben aangetoond dat het gebruik van DBV voor processorvirtualisatie aanvaardbare virtualisatiekosten met zich meebrengt, zolang de juiste optimalisaties aangewend worden. We hebben aangetoond hoe het geheugenbeheer volledig kan worden gevirtualiseerd, enkel gebruik makend van softwareoplossingen. We hebben verschillende technieken voorgesteld en getest om de nodige softwarecaches te beheren, en de beste technieken voor elke cache afzonderlijk vastgelegd. Ons onderzoek heeft dus de fundamentele uitdagingen opgelost om softwaregebaseerde virtualisatie met DBV mogelijk te maken op de ARMv7-A-architectuur.

Summary

In system virtualisation, an operating system is isolated from the hardware of a physical machine by means of a hypervisor. Such an operating system is then called a *guest* of that hypervisor. System virtualisation has already proven itself to be useful in many cases ranging from reducing operational costs of data centres through consolidation, to facilitating software development and emulating legacy software. So far, however, virtualisation has mainly been used on desktops and servers. As embedded systems are growing more powerful, and embedded hardware is evolving faster than software, embedded systems can now also benefit from virtualisation. Existing solutions for data centres and desktop computers can, however, not be readily applied to embedded systems, because of differences in requirements, use cases, and architecture.

ARM is by far the leading architecture in the embedded and mobile market. Over the past decade multiple hypervisors have been developed for it. Because the ARM architectures did not support full system virtualisation, most efforts started with paravirtualisation, a technique known to have several drawbacks. Recently, ARM extended its ARMv7-A architecture with hardware support for full virtualisation. While hypervisors using these extensions do not suffer from the drawbacks of paravirtualisation, they cannot run on the vast majority of ARM processors in use today, due to their need for special hardware support. Alternatively, architectures that do not support full virtualisation out of the box can be virtualised using dynamic binary translation (DBT) techniques: problematic instructions are patched at run time in the binary instruction stream of the guest. DBT is more versatile than architectural support for full virtualisation, as it transparently enables a multitude of other virtualisation usage scenarios such as optimisations across the border between operating system kernels and applications, emulation and optimisation of legacy software stacks, full system instrumentation and

testing, and even load balancing in heterogeneous multi-core systems through cross-architecture virtualisation.

The key research question we seek to address is: which technological challenges must be solved to enable the above mentioned use cases for DBT on the ARMv7-A architecture? To answer this question, we first identify the architecture's limitations to full system virtualisation. We then propose generic CPU and memory virtualisation solutions that offer full virtualisation without relying on specific hardware support for the ARMv7-A architecture. Our research on software-only memory virtualisation techniques will focus on shadow translation tables.

We start from Popek and Goldberg's classic virtualisability theory, and extend it for today's reduced instruction set computer (RISC) architectures and usage patterns: we add paged virtual memory, input/output (IO), and interrupts. We then use our updated model to show why the original ARMv7-A architecture is not classically virtualisable, and how ARM's hardware extensions make it virtualisable. We argue that DBT-based hypervisors are an evolution of Popek and Goldberg's ideas on hybrid virtualisation, and we show how our updated model can assist with the construction of a DBT engine for full system virtualisation.

We use our analysis to build the *STAR hypervisor*, a new bare-metal hypervisor for the ARMv7-A architecture. Our hypervisor is the first open-source software-only hypervisor for the ARMv7-A architecture that offers full system virtualisation. It runs unmodified guest operating systems, decoupled from the hardware through DBT, and uses shadow translation tables to virtualise the memory management unit (MMU).

Prior work on DBT for the ARM architecture mainly focuses on process virtualisation. As system virtualisation comes with its own unique challenges, the existing process virtualisation techniques cannot be used "as is" in the context of full system virtualisation. In user-space, DBT engines often take shortcuts which are only valid for well-behaving applications, and there is no strict requirement to isolate the DBT engine from the application. In full system virtualisation, DBT is used together with MMU virtualisation to isolate guests from the hypervisor and from one another. Furthermore, kernel code often contains handwritten assembly and special system instructions, requiring special care in the DBT engine.

We study how to address the challenges specific to using DBT for full system virtualisation. We start with a fundamental problem in the translator: translations often require an extra register, but such register may not always be available. Our translator therefore needs to spill and

restore registers to some location in memory or other hardware that is accessible to the guest. While solving this problem is trivial for process virtualisation, the solution is more involved for system virtualisation, as the hypervisor must write-protect as much of its data structures from the guest as possible, and the guest's data structures cannot be relied upon. We propose new solutions to the spilling problem, and we evaluate their impact on the performance of translated code.

We further analyse the bottlenecks in our DBT engine by running micro-benchmarks from the lmbench suite. Those benchmarks model typical interactions between Linux kernels and their user applications. We study which kinds of system instructions cause a trap from the guest to the hypervisor's DBT engine most frequently, and propose translation techniques to avoid as many of such traps as possible. We analyse the impact of each translation technique separately on each benchmark. As is typical with DBT, much of the run-time overhead can be attributed to control flow, and eliminating traps caused by control flow yields significant speedups. However, we also show that the remaining overhead can be further reduced by 51% on average, by using binary optimisations specific to ARMv7-A system virtualisation.

We then compare the run-time virtualisation costs of our optimised DBT engine with our naive, unoptimised DBT engine on real applications. We use benchmarks from the mibench suite to model these applications. We find that a naive configuration of our DBT engine makes the tested applications run up to 5 times slower than native. With our optimisations, however, the maximum perceived slowdown is limited to 39%.

The DBT engine only performs CPU virtualisation. As a hypervisor lives at the lowest level of the software stack, it is also responsible for managing the MMU, the hardware caches and the translation lookaside buffers (TLBs). We therefore discuss how to virtualise the MMU with shadow translation tables, how to virtualise cache operations. We show that guests' cache operations cannot always be applied directly to the hardware caches, because they can cause the hypervisor to lose data.

Our hypervisor makes extensive use of software caches both in the DBT engine and in MMU virtualisation. Such software caches must be kept up to date with the guests' internal state, either through clever memory management or by reusing a guest's hardware cache maintenance operations. The caches of the DBT engine must be updated whenever the guest unmaps, remaps, or modifies code that has already been translated. Similarly, the shadow translation tables must be updated whenever the

guest modifies its translation tables, and in particular when those modifications affect descriptors that have already been shadow mapped.

We have researched both a memory management-based approach and a software cache-based approach for both kinds of software caches. We evaluate our techniques and find that, due to the way cache and TLB organisation works on ARM, shadow translation tables are best managed like a software TLB. The caches of our DBT engine, however, are better managed through memory protection techniques.

During the course of our research, we have designed and implemented a proof-of-concept virtualisation platform for the ARMv7-A architecture to study and evaluate all of the above mentioned techniques. We have demonstrated that using DBT for CPU virtualisation results in acceptable levels of run-time overhead when properly optimised for the target architecture. We have shown how software-only MMU virtualisation can work on ARM, and what the best practices are to manage the shadow translation tables and the caches of the DBT engine. We have hence solved the fundamental challenges to enable software-only virtualisation on ARMv7-A.

Contents

Chapter 1

Introduction

The term *virtualisation* is used in many different contexts, ranging from storage and network technologies to execution environments, and even to virtual realities. The research presented in this dissertation focuses on *system virtualisation*—technology that allows multiple operating systems to be executed on the same physical machine simultaneously. It achieves this by isolating those operating systems, also called *guests*, from the physical hardware and from each other, by means of a *hypervisor*.

System virtualisation has already proven itself to be useful in the server and desktop computing worlds. It has many use cases ranging from reducing operational costs of data centres through consolidation, to facilitating software development and emulating legacy software. As embedded systems are growing more and more powerful, and embedded hardware is evolving faster than software, embedded systems can also benefit from virtualisation. Solutions for data centres and desktop computers can, however, not be readily applied to embedded systems, because of differences in requirements, use cases, and architecture.

ARM is by far the leading architecture in the embedded and mobile market [112]. Over the past decade multiple hypervisors have been developed for it. Because the ARM architecture did not support full system virtualisation, most efforts started with paravirtualisation, a technique known to have several drawbacks [39]. Recently, ARM extended its ARMv7-A architecture with hardware support for full virtualisation [13]. While hypervisors using these extensions do not suffer from the drawbacks of paravirtualisation, they cannot run on the vast majority of ARM processors in use today, due to their need for special hardware support.

In this dissertation, we propose generic CPU and memory virtuali-

sation solutions that offer full virtualisation without relying on specific hardware support for the ARMv7-A architecture. To fully virtualise the CPU, we dynamically rewrite the code of guest kernels at run time, a technique known as dynamic binary translation (DBT). As CPU and memory virtualisation are inevitably tied together, we have also researched memory virtualisation techniques based on shadow translation tables.

Prior work on DBT for the ARM architecture mainly focuses on process virtualisation. As system virtualisation comes with its own unique challenges, the existing process virtualisation techniques cannot be used "as is" in the context of full system virtualisation. Furthermore, process virtualisation does not require memory virtualisation. We will therefore propose new solutions to address the challenges of system-level DBT on ARM. We have implemented and evaluated our techniques in our own hypervisor, STAR, the first software-only hypervisor that offers full virtualisation using DBT for the ARMv7-A architecture. This hypervisor has been co-developed with The University of Manchester.

The remainder of this chapter provides a concise overview of prior work on system virtualisation in general, and for ARM-based embedded systems. Section 1.1 sketches the origins of virtualisation, and introduces *classic virtualisability*—a theory used to determine whether or not an architecture is suitable for full virtualisation. In Section 1.2, we describe the different techniques for system virtualisation in use today, and discuss their advantages and disadvantages. We use this to classify existing solutions for ARM-based embedded systems in Section 1.3. We illustrate why, even with hardware extensions for virtualisation, DBT-based virtualisation remains useful in Section 1.4. We conclude this chapter in Section 1.5 with an overview of the major contributions and an outline of the remainder of this dissertation.

1.1 A brief history of virtualisation

The earliest virtualisation solutions date from the late 1950s and consisted of time sharing systems, created to enable multiple users to use a single machine concurrently [66, 116]. In the early 1960s, IBM engineers working on the M44/44X project first used the word *virtual* in conjunction with computers. They worked on methods to partition the processor time and the memory of an IBM 7044 computer between multiple software images. Those images were referred to as *virtual machines* [44].

The M44/44X predates the inception of modern, *third-generation*, op-

erating systems with privilege separation and support for multiprogramming. Its virtual machines therefore merely contained the bare minimum abstractions necessary to run individual applications. Operating systems as they are known today only appeared in the early 1960s [29, 38]. The first hypervisor capable of fully virtualising its underlying architecture was IBM's CP-40. It was productised as the CP-67/CMS system and became the first commercial hypervisor at the end of the 1960s [1, 57].

In 1974, Popek and Goldberg [97] defined a set of formal system virtualisation requirements for third-generation computer architectures, based on experiences with virtualisation in contemporary mainframe systems such as the IBM 360/67 (with CP-67/CMS), and the DEC PDP-10. They formally proved that if an architecture meets their requirements, an "efficient" hypervisor can be constructed for that architecture. Their paper is now regarded as one of the groundworks of virtualisation, and their theory is referred to as "classic virtualisability".

Over time, as time-sharing systems replaced batch processing systems, third-generation operating systems matured and took over the role of the old hypervisors: there was no longer a need to run multiple pieces of system software from different users on a single machine; multiple users could run different user-space applications on a single operating system kernel instead. By the early 1980s computer hardware costs had dropped significantly and computers became affordable for the masses. Obsoleted by operating systems, virtualisation was regarded as nothing more but a relic of the past by both industry and academia [30, 105].

Operating systems became more feature-rich, but their complexity also brought stability issues and security concerns. As computer prices kept dropping, it became common practice in industry to dedicate individual machines to specific tasks, effectively reversing the multiprogramming revolution. However, each dedicated system had to be managed independently and most systems were idle most of the time, leading to management overhead and waste of resources. Virtualisation became relevant again, as it was a potential solution to all these problems [105].

Unfortunately, virtualisation techniques originally developed for mainframes could not readily be implemented on most commodity computers and servers: their architecture, the Intel x86, did not meet Popek and Goldberg's strict requirements for efficient hypervisors [2]. In 1999, VMware was the first company to come up with a full virtualisation solution for the x86 architecture using *dynamic binary translation* (DBT) techniques: instructions that made the architecture violate Popek and

Goldberg's classic virtualisability requirements were patched at run time in the binary instruction streams of the virtual machines (VMs) [104, 134].

DBT hypervisors are very complex to construct. It did not take long before researchers realised that the problem of virtualising the x86 architecture could also be solved by adapting the operating systems upfront rather than patching them at run time. The hypervisor then presents a customised interface to its VMs, similar but not identical to the underlying hardware. Such techniques are now widely known as *paravirtualisation*; although pioneered by IBM in their early virtualisation work, the term was first coined in 2002 by Whitaker et al. [136, 137] in their work on the Denali hypervisor, the first such hypervisor for the x86 architecture.

Virtualisation, a technology from the mainframe era, quickly conquered the server and desktop markets. By 2005, the trend was picked up by x86 hardware makers Intel and AMD, and the architecture gained hardware support for full virtualisation [3, 127].

1.2 Taxonomy of hypervisors

Over the years, many different kinds of virtualisation systems have been designed. The first classification of such systems dates back to 1973. Based on a formal model of how hypervisors interact with the underlying hardware platform, Goldberg [56] defined two types of hypervisors:

- *Type I* hypervisors run directly on top of the hardware, at the highest privilege level. Their guests run at lower privilege levels. These hypervisors are now known as *native* or *bare-metal* hypervisors.
- *Type II* hypervisors run on top of an existing host operating system. They are therefore also referred to as *hosted* hypervisors.

Figure 1.1 depicts the relationships between the hardware, the hypervisor and the operating systems for each type of hypervisor. Type II hypervisors can benefit from the services and abstractions provided by a host operating system. Therefore, the development and debugging processes for such hypervisors are easier than for bare-metal hypervisors: development and debugging tools can be run from the host, essentials such as C libraries and drivers can be reused, and the host can be reused as management interface to the hypervisor. However, not all features of the host are necessary for the hypervisor to operate, and they form a

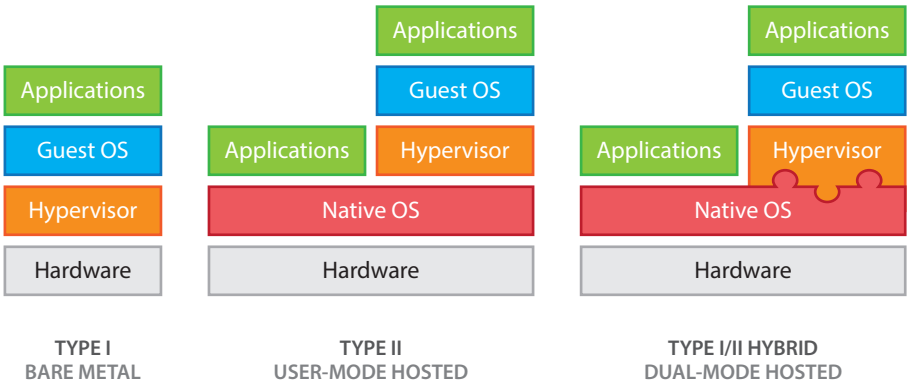

Figure 1.1: Classification of hypervisors according to Goldberg and Gallard

source of run-time overhead. Such overhead is more easily tolerated on desktop computers and in data centre environments than on resource-constrained embedded systems. Type I hypervisors avoid the overhead of a host operating system at the expense of more complicated development and debugging processes; furthermore, they require custom run-time libraries and drivers for each hardware platform.

Although modern hypervisors have evolved substantially since the publication of Goldberg's classification, most hypervisors today can be classified as either type I or type I/II hybrids. Gallard et al. [55] have extended Goldberg's formal model to accommodate for such hybrids.

Smith and Nair [113] distinguish between two kinds of hosted hypervisors: *user-mode* hosted hypervisors run fully on top of their host operating system at a lower privilege level than its kernel, while *dual-mode* hosted hypervisors—type I/II hybrids—also contain software components that run at the same privilege level as the host's kernel (such as drivers). Such hybrid designs are necessary for a hosted hypervisor to make use of hardware support for full virtualisation [3, 9, 13, 127].

Hypervisors can also be classified based on the techniques they use to virtualise their guests, as shown in Figure 1.2. Firstly, as explained in Section 1.1, we distinguish between *full virtualisation*, also called faithful virtualisation, and *paravirtualisation*. In full virtualisation, a hypervisor presents an interface to its guests that is identical to the underlying hardware platform. Any operating system designed for that hardware platform can then run unmodified as a guest. In theory, guests cannot even distinguish between their virtual copy of the hardware and the real hardware. In practice, the term full virtualisation may also be used for

hypervisors that virtualise a physical platform different from the one they run on. A hypervisor based on paravirtualisation presents a modified interface to its guests, which is similar but not identical to the underlying hardware platform. All operating systems must first be adapted to this custom interface before they can be run as a guest [113, 134, 136, 137].

There are many hypervisors based on paravirtualisation and most of them use their own customised interface between the hypervisor and its guests. None of the efforts to standardise those interfaces has gained sufficient momentum to spread to more than a few operating systems or hypervisors [4, 83, 106]. As a consequence, few operating systems support those interfaces out of the box. Instead, hypervisor vendors and third parties must provide source code patch sets to support specific hypervisor interfaces for specific operating system versions. On the one hand, selling and supporting patches constitutes a proper business model for paravirtualisation vendors. On the other hand, paravirtualisation solutions also come with four major drawbacks:

1. Developing, maintaining, and testing patch sets for each and every combination of a specific operating system version and a specific hypervisor interface is an expensive process. Although semantic patches may offer a solution to simplify patch management [15, 76, 95], the effort required for testing remains.

2. Patched operating systems may exhibit unexpected behaviour because their reliability is not guaranteed and patches may introduce new security issues.

3. Licenses may prevent or restrict modifications to operating systems source code, and often impose rules on the distribution of patch sets or patched code.

4. Previously certified software stacks will need to be re-certified after patching. The recertification process is expensive and always specific to a particular hypervisor interface, thereby stimulating vendor lock-in.

This analysis is shared by major players in industry including ARM, Nokia and STMicroelectronics [39]. The drawbacks of paravirtualisation can be avoided by using full virtualisation, which comes with the added benefit that it can be used to virtualise a priori unknown software.

Full virtualisation can be achieved either through hardware support or by using software techniques such as DBT. Computer architectures

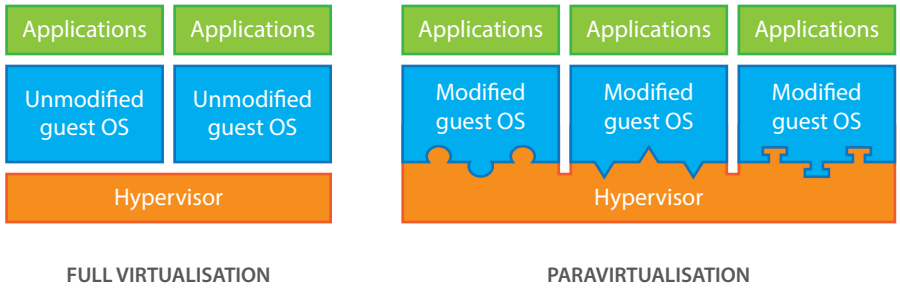

Figure 1.2: Classification of hypervisors by virtualisation technique

that meet the requirements set out by Popek and Goldberg [97] in 1974 support full virtualisation out of the box. Architectures such as x86 and ARM have gained hardware support for full virtualisation in later iterations through extensions [3, 9, 13, 127]. While hypervisors built for such hardware extensions easily outperform DBT-based hypervisor and are easier to construct, they do not run on older hardware. Furthermore, DBT remains useful in a handful of scenarios, ranging from full-system instrumentation to legacy system software emulation.

1.3 Virtualisation for embedded systems

Virtualisation in the server and desktop world has already matured, with both software and hardware solutions available for several years [2, 21, 24, 104, 113, 127, 135]. However, virtualisation of embedded systems is still an area of ongoing research [14, 52, 63]. While the motivation to virtualise embedded systems is similar to desktop and server virtualisation, solutions for the latter cannot be easily reused on embedded systems due to differences in requirements, use cases, and computer architecture.

ARM is by far the leading architecture in the embedded and mobile market [112], and several hypervisors have already been developed for it. A comprehensive overview of the existing solutions is shown in Table 1.1.

Early designs such as ARMvisor [47], B-Labs Codezero Embedded Hypervisor [18], KVM for ARM [40], TRANGO Virtual Processors Hypervisor [122], NEC VIRTUS [73], VirtualLogix VLX [15, 133], VMware MVP [22, 25] and Xen ARM PV [70, 82] all use paravirtualisation, as they predate the introduction of ARM's hardware support for full virtualisation. As stated in Section 1.2, such hypervisors have several drawbacks.

Table 1.1: Overview of ARM hypervisors

Name	Type			References
	Para	Full	Goldberg / Smith	
ARMvisor	●	○	dual-mode hosted	[47]
B-Labs Codezero Embedded Hypervisor	●	○	bare-metal	[18]
ITRI Hypervisor	●	○	dual-mode hosted	[110, 111]
KVM for ARM	●	○	dual-mode hosted	[40]
KVM/ARM	○	●	dual-mode hosted	[41, 42]
Open Kernel Labs OKL4 Microvisor	●	●	bare-metal	[62–64, 132]
Red Bend Software vLogix Mobile	○	●	bare-metal	[102]
SYSGO PikeOS	●	●	bare-metal	[78, 118]
NEC VIRTUS	●	○	bare-metal	[73]
TRANGO Virtual Processors Hypervisor (VMware MVP precursor)	●	○	bare-metal	[122]
Varmosa (VMware-funded MIT research project)	○	●	dual-mode hosted	[69]
VirtualLogix VLX	●	○	bare-metal	[15, 133]
VMware Mobile Virtualization Platform (MVP)	●	○	bare-metal	[22, 25]
Xem ARM PV (Secure Xen on ARM)	●	○	bare-metal	[70, 82]
Xen ARM with Virtualization Extensions	○	●	bare-metal	[140]
Xvisor	●	●	bare-metal	[141]

Xen ARM PV and KVM for ARM have been superseded by a new Xen port [140] and KVM/ARM [41, 42] in which paravirtualisation support was dropped in favour of ARM's hardware support. VirtualLogix VLX was used by Red Bend software as the basis for vLogix Mobile, which now also uses ARM's hardware support instead of paravirtualisation [102]. Some hypervisors support both paravirtualisation and full virtualisation, such as PikeOS [78, 118], OKL4 [64, 132] and Xvisor [141].

PikeOS and OKL4 are commercial microkernel-based hypervisors specifically designed for real-time systems virtualisation [62–64, 78]. Such hypervisors can directly run application software, referred to as native applications, a feature often used to run small real-time software. Full-fledged real-time operating systems are typically paravirtualised for scheduling reasons. When multiple software components with mixed criticalities are consolidated into one hardware system, the less critical components can benefit from full virtualisation. For this purpose, OKL4 has supported ARM's hardware extensions for full virtualisation since the early days through a partnership with ARM [131, 132]. PikeOS has only recently gained support for full virtualisation [118].

The ITRI Hypervisor presents an interesting case as it uses a combination of static binary translation (SBT) for CPU virtualisation and paravirtualisation for memory management unit (MMU) virtualisation [111]. SBT can be regarded as a form of paravirtualisation, although it does not require the sources of the guest kernels, and therefore aims to be a more generic approach. Any benefits of using SBT over traditional paravirtualisation are however lost, because the authors of the ITRI hypervisor deemed the MMU too complex to properly virtualise. Furthermore, SBT cannot support any form of self-modifying code at all, and it complicates dynamically loading code at run time, as is often done in operating system kernels for device drivers and other kernel modules. The ITRI Hypervisor therefore only supports non-modular kernels.

SBT on ARM is hard, because it is common for data to be embedded within code sections, and code may use a mix of 32-bit ARM and variable-width Thumb-2 instructions in which the encoding of each byte can only be deduced by control flow analysis. In the absence of symbolic debug information, extensive analyses and heuristics are required to distinguish code from data [33, 68], and any uncertainties must be dealt with at run time using a combination of interpretation and DBT. There is however no mention of such techniques in any of the ITRI Hypervisor documents [110, 111]. Relying on the availability of symbolic debug

information makes an SBT-based CPU virtualisation solution no better than traditional paravirtualisation: symbolic debug information is often not available for commercial software, and its representation may vary across operating systems, so that a translator must support each of them.

Varmosa, a product of a VMware-funded MIT project, is the only hypervisor capable of fully virtualising the ARM architecture without hardware extensions for full virtualisation, because it makes use of DBT techniques [69]. Unfortunately, it targets the dated ARMv4 architecture, and its translation techniques have never been published. All other hypervisors for the ARM architecture that offer full virtualisation rely on ARM's recently introduced hardware support. A VMware talk and patent on DBT techniques for ARM published by members of the team that supported Varmosa reveals that they used *in-place translation* [27, 46], a technique also known as patching [113]. As we shall see in Section 3.5.1, in-place translation cannot properly support self-verifying and self-modifying code on ARM due to architectural limitations.

1.4 Modern uses of dynamic binary translation

Although DBT has been used successfully to virtualise the Intel x86 architecture by VMware, Microsoft, QEMU, VirtualBox and others [24, 67, 134, 135], it has been deemed too complex for the virtualisation of other architectures after the emergence of paravirtualisation and hardware extensions for full virtualisation. In reality, hardware extensions introduce similar complexity, but at the hardware level. Although they make it possible to construct a much smaller hypervisor, thereby decreasing the potential for bugs and vulnerabilities, the design complexity of hardware extensions is illustrated by the fact that early implementations on x86 were outperformed by pure software virtualisation solutions [2].

On the ARM architecture, Varmosa is the only hypervisor we know of that has attempted full virtualisation using DBT [69]. While software-only paravirtualisation and modern hardware-based full virtualisation solutions easily outperform DBT [111, 132], it remains a useful technique by virtue of its versatility. Virtualisation is often considered as a solution to address software portability issues across different architectures. Hardware extensions simply introduce new portability issues at a different level. DBT is more versatile since it transparently enables a multitude of other virtualisation usage scenarios:

- **Optimisations across the border between operating system kernels and applications**: DBT can be used to optimise and reduce context switching between an operating system kernel and its applications under a hypervisor, as many operations become redundant when executed on virtual hardware [2].
- **Legacy emulation and optimisation**: DBT can be used as a glue layer to emulate legacy hardware platforms and to resolve backward incompatibility between generations of architectures [34]. Such emulation is useful as today's hardware has become more flexible than software [49, 61]: system-on-chips have an average lifetime of 4 years, while software stacks often have to last much longer. Because of implementation differences between system-on-chips and evolutions in hardware architecture, software must be continuously rewritten and recertified for newer platforms.
- **Full system instrumentation and testing**: entire software stacks may be instrumented and tested from underneath using a hypervisor. This idea was pioneered by IBM with their SIMMON test tool in the 1960s. The most notable recent example is PinOS [28].
- **Load balancing in heterogeneous multi-core systems**: DBT can be used to translate code from one core to another if cores have different instruction set architectures (ISAs) [72, 139]. If cores share the same ISA, such as in ARM's big.LITTLE [11], hardware extensions remain a feasible approach to virtualisation. But even in this case, DBT can potentially improve performance by dynamically optimising code for the micro-architectural features of the target architecture, such as a static branch predictor in little cores.

The key research question we seek to address in this dissertation is: which technological challenges must be solved to enable the above use cases for DBT on the ARMv7-A architecture? We can further break down this question as follows:

- What are the architectural limitations to full system virtualisation on ARMv7-A? Can we learn how to work around such limitations from an analysis of the architecture according to Popek and Goldberg's theory of classic virtualisability?
- Can existing work on process virtualisation techniques for ARM be adapted to perform CPU virtualisation in a hypervisor and how?

- Which software techniques can be used to virtualise the MMU?
- How do our virtualisation techniques affect guest performance?

1.5 Outline and contributions

The major contributions presented in this doctoral dissertation are:

1. An update to the model of Popek and Goldberg with paged virtual memory, input/output (IO), and interrupts;
2. The application of our updated model to study the virtualisability of the ARMv7-A architecture both without and with the virtualisation extensions;
3. A discussion on the trade-offs between the use of architectural extensions for virtualisation and DBT, in which we argue that both have their use in future systems;
4. The STAR hypervisor, the first open-source software-only hypervisor for the ARMv7-A architecture that offers full virtualisation using DBT;
5. An analysis of existing techniques for user-space DBT on ARM and several new solutions to adapt those techniques to full system virtualisation;
6. A study on the sources of the run-time overhead of the DBT engine of our hypervisor and new techniques to reduce this overhead specific to full system virtualisation on ARM;
7. An evaluation of all our techniques for improving the performance of the DBT engine on a real embedded hardware platform;
8. A discussion on multiple techniques to manage shadow translation tables and the software caches of the DBT engine, followed by an evaluation on a real embedded hardware platform.

Contributions 1 to 3 have been published as:

Formal Virtualization Requirements for the ARM Architecture
Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne,
Bjorn De Sutter, and Koen De Bosschere
In *Journal of Systems Architecture*
Volume 59, Number 3, March 2013, pp. 144-154 [96]

Contributions 5 to 7 are currently under revision for publication as:

Evaluation of dynamic binary translation techniques for full system virtualisation on ARMv7-A
Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne,
Bjorn De Sutter, and Koen De Bosschere
In *Journal of Systems Architecture*

The remainder of this dissertation is organised as follows: in Chapter 2, we extend Popek and Goldberg's theory on classic virtualisability to modern RISC architectures such as ARM, and to modern approaches such as DBT. We analyse the ARM architecture and outline the requirements for DBT on ARM. We use this analysis to build our own hypervisor in Chapter 3. We explain the minimal set of scientific challenges to overcome when fully virtualising the ARMv7-A architecture using a software-only approach. In Chapter 4, we propose new techniques for CPU virtualisation using DBT on ARM, and we study their performance based on both micro-benchmarks and real applications. We discuss and evaluate MMU and cache virtualisation techniques together with software cache management in Chapter 5. In Chapter 6, we provide an overview of some lessons we learnt during the design and the optimisation of our hypervisor that did not lead to publishably results. We then summarise our results and conclude this dissertation in Chapter 7.

Chapter 2

Formal virtualisation requirements for the ARM architecture

Popek and Goldberg published the seminal paper on virtualisation over 40 years ago [97]. Although its reasoning remains useful, we argue that its model is dated. In this chapter, we therefore revisit the formal requirements for virtualisability derived by Popek and Goldberg. We extend their abstract machine model to match today's reduced instruction set computer (RISC) architectures and usage patterns: we add paged virtual memory, IO, and interrupts. We then use our updated model to show why the original ARMv7-A architecture is not classically virtualisable, and how ARM's hardware extensions make it virtualisable.

Modern insights such as DBT enable efficient virtualisation beyond Popek and Goldberg's requirements. We argue that DBT-based hypervisors are an evolution of their hybrid virtualisation idea, and show how our model can assist with the construction of such hypervisors.

The work presented in this chapter was first published in the *Journal of Systems Architecture* [96].

2.1 Introduction

In 1974, Popek and Goldberg [97] wrote a paper on "Formal requirements for virtualizable third generation architectures". It defines system virtualisation criteria for computer architectures, and proves that if they are met, an "efficient" hypervisor can be constructed for that architecture. Since its publication, the paper has been used as a reference point for designing hardware platforms capable of supporting an efficient hypervisor, and it has become the groundwork of virtualisation technology.

The criteria defined by Popek and Goldberg are now known as the conditions for *classic virtualisability*. Architectures that meet these criteria are particularly suited for full virtualisation. Popek and Goldberg based their model on computers available at the time, such as DEC PDP-10 and IBM 360/67. Due to advances in microprocessor architecture, however, their model is no longer a good match for current architectures. We will update their model to fit contemporary architectures, such as ARMv7-A, with support for paged virtual memory, IO, and interrupts.

Popek and Goldberg also proposed a *hybrid virtualisation* technique that enabled virtualisation of architectures that did not meet their criteria: instead of interpreting only the sensitive instructions, *all* instructions normally executed in privileged mode are interpreted. New approaches in the construction of efficient hypervisors that do not fit Popek and Goldberg's model and criteria [55], such as DBT, have enabled virtualisation of many more contemporary architectures. We argue that hypervisors based on DBT, which do not require changes to the architecture, can be regarded as an evolution of their hybrid virtualisation ideas. Their analysis therefore remains useful for the construction of such hypervisors, and for understanding the advantages and disadvantages of both the hardware-supported and the DBT-supported approaches.

At the end of the chapter, we analyse the ARMv7-A architecture with our new model. This results of this analysis will help us to construct a DBT engine for full system virtualisation in Chapter 3.

2.2 Background and motivation

2.2.1 Classic virtualisability

The machine model used by Popek and Goldberg is deliberately simplified. It includes a processor and a linearly addressable memory, but

does not consider interaction with IO devices and interrupts.

The processor operates either in supervisor mode or in user mode. Supervisor mode is a privileged mode, meant for the operating system, while user mode is an unprivileged mode, meant for user applications. The processor also features a program counter and a relocation-bounds register, used for relative memory addressing. An ISA for such a processor can move, look up or process data and alter program control flow. Based on this description, they defined their concept of machine state:

Definition 1. The machine state S is defined by the contents of the memory E, the processor mode m, the program counter pc, and the relocation-bounds register r:

$$S \equiv \langle E, m, pc, r \rangle .$$

Since all variables that determine the machine state are finite, the set of all machine states Σ is also finite.

Definition 2. An instruction i is a function on the set of machine states Σ that maps one state to another:

$$\begin{array}{rccc} i: & \Sigma & \longrightarrow & \Sigma \\ & S_x & \longmapsto & i(S_x) = S_y. \end{array}$$

All instructions are classified in three categories:

- *Privileged* instructions execute correctly in privileged mode, and always trap in unprivileged mode.
- *Sensitive* instructions modify or query the configuration of the system. They are further classified as *control-sensitive* and *behaviour-sensitive* instructions. Control-sensitive instructions attempt to modify the processor execution mode, or to set the amount of available virtual memory resources by updating the relocation-bounds register. Behaviour sensitivity manifests itself in two ways. The result of behaviour-sensitive instructions either depends on the value of the relocation-bounds register (location sensitivity) or on the processor execution mode (mode-sensitivity).
- *Innocuous* instructions are those instructions that are not sensitive.

Upon a trap, the processor mode m, the program counter pc and the relocation-bounds register r are saved to the memory E. Control is

then transferred to a privileged mode and pc will point to a predefined trap handler. After handling the trap, the original processor mode, the program counter and the relocation-bounds register can be restored from memory.

Popek and Goldberg also defined three fundamental properties for virtualised systems:

1. **Efficiency**: all innocuous instructions are executed natively without hypervisor intervention;

2. **Resource control**: guest software is forbidden access to physical state and resources;

3. **Equivalence**: guest software behaves identical to when it is run on a system natively, assuming that it is free of timing dependencies.

Using the above definitions, they proved the following theorem:

Theorem 1. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

On an architecture on which all sensitive instructions are also privileged, a trap is generated and caught by the hypervisor whenever a guest attempts to execute a sensitive instructions. Such instructions must then be "interpreted" by the hypervisor. All other instructions (innocuous instructions) must be executed natively. This kind of hypervisor is also known as an *execute-to-trap* hypervisor. It is this ability to execute most of the code directly that enables "efficient" virtualisation. Popek and Goldberg also showed that if the efficiency property is loosened, allowing interpretation of all privileged guest code, a so-called *hybrid* hypervisor can be constructed for otherwise non-virtualisable architectures.

To specify the conditions for the construction of a hybrid hypervisor, Popek and Goldberg defined a new subset of sensitive instructions called *user-sensitive* instructions. An instruction is user-sensitive if there exists a state in the unprivileged mode for which the instruction is sensitive. Using the categories defined earlier, user-sensitive instructions can be further classified as control-sensitive and location-sensitive instructions. The set of user-sensitive instructions does not include any mode-sensitive instructions, as Popek and Goldberg's model only features one single unprivileged ("user") mode. They then proved the following:

Theorem 2. *A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user-sensitive instructions is a subset of the set of privileged instructions.*

Alternatively, a DBT engine can be used to rewrite instructions at run time at an acceptable performance cost [19]. If the conditions for the construction of a hybrid hypervisor are met, only privileged guest code must be rewritten. Otherwise, full virtualisation can still be achieved by rewriting all instructions, i.e. including unprivileged guest code. Using hybrid hypervisors and DBT, a much wider range of computer architectures can be virtualised.

2.2.2 Prior updates to the model

Dong and Hao [48] have attempted to extend the model by Popek and Goldberg [97] with interrupts and memory-mapped IO. Because they treat the subject from the view of user-space applications rather than from the view of the operating system or the underlying computer architecture, they exclude IO operations initiated by operating systems.

Their model introduces a set of possible IO states Γ, similar to the set of machine states Σ, and implicitly redefines instructions as functions with domain and range $\Sigma \times \Gamma$. However, they do not revise the original definitions from Popek and Goldberg at all, even though they alter the model on which those definitions are based. It is unclear what purpose their model serves, and whether it can be applied in practice.

Interrupts and exceptions are vital for today's computer software, not only for IO, but also because they enable communication between operating systems and their applications through software interrupts (system calls), and because they are key to modern memory management techniques such as swapping. However, Dong and Hao only consider interrupts and exceptions in the context of IO with devices. Their treatment of the subject is therefore incomplete. At the time of writing, we are not aware of any other extensions to Popek and Goldberg's model.

2.2.3 Advances in computing practice

Over the past 40 years, computer architectures have been significantly extended, use cases have changed, and users' expectations have evolved accordingly. The model by Popek and Goldberg [97] therefore no longer fits current computing practice.

Memory relocation and protection

The original model allows for a minimal form of memory relocation and protection, using a single combined base location and bounds register. Such a register is capable of relocating non-privileged applications under the control of an operating system running outside a virtualised environment, and can relocate operating systems themselves (and their applications) when running under the control of a hypervisor. Although computer systems with paged virtual memory were commercially available in 1974 [56], virtual memory was still seen primarily as a technique for optimising the utilisation of then-expensive physical memory. Most applications were designed without directly depending on the virtual memory facilities of an operating system. A simple relocation scheme, typically found on systems without virtual memory, was therefore adequate to model the behaviour of a real computer system.

Today's operating system designers, however, exploit virtual memory to offer applications many key facilities such as dynamic linking, shared libraries, and dynamically expandable heap and stack areas. It is no longer realistic to hide the details of these facilities in a discussion of virtualisation. Therefore, we will introduce the concept of a *memory map* in the definition of machine state. An operating system will use such maps to define the memory space accessible to applications. A hypervisor will also use such maps to define the memory space accessible to its guests. Correctness conditions for the latter are derived in Section 2.3.2.

Timing

A second difference that has arisen in recent years is the importance of timing in our use of computers. At the time Popek and Goldberg [97] published their results, much of the computing workload was still processed in batches, with input prepared off-line and output printed for later usage. Modern computing is significantly more time-sensitive—much of our interaction with personal computers and portable electronics is characterised as "soft real-time", in which failure to deliver a response in a timely manner is perceived as a vexatious malfunction.

The model by Popek and Goldberg does not aim for analysing timing-dependent behaviour. It also does not lend itself to study the influence multiple VMs running under the same hypervisor exert on each other. In this dissertation, we will also limit the discussion to whether an architecture lends itself to virtualisation in general, for which studying the case

of a single VM is sufficient, and to how a hypervisor can be constructed for that architecture. As we have stated earlier, we do not consider deeply embedded systems. This rules out hard real-time systems.

The underlying idea of efficiency in Popek and Goldberg's paper was that in an efficient system, relatively few instructions would trap. This implies that if one wants to retain soft real-time behaviour in a virtualised system, all time spent in traps to the hypervisor must be bounded.

IO, interrupts and exceptions

Most IO in contemporary computer architectures is either memory-mapped input / output (MMIO) or port-mapped input / output (PMIO). IO operations may result in interrupt generation, but generally this is not necessary, unlike what is suggested by Dong and Hao [48].

In an architecture that supports MMIO, IO device registers are mapped into the same address space as physical memory. We put forward that resource control is retained if all accesses to IO device registers trap or can be configured to trap a priori. The instructions that perform these accesses will be generic load and store instructions. However, we cannot require all such instructions to be privileged, because their sensitivity depends on the address they act upon. Load and store instructions that operate on main memory are clearly innocuous, given that they operate on virtual addresses. Treating all load and store instructions as sensitive therefore violates the efficiency property. Instead, virtual memory mechanisms can be leveraged to protect memory-mapped devices, such that all accesses to device memory cause a trap. The efficiency property is retained if memory protection is possible at such granularity that accesses to main memory are not affected.

PMIO can be used to separate memory accesses from device accesses in the hardware. Software must then use dedicated IO instructions to communicate with devices. Such instructions can be seen as controlling resources; as such, they are similar to control-sensitive instructions: it is sufficient that all dedicated IO instructions are also privileged to retain classic virtualisability. PMIO and MMIO each have their advantages: PMIO requires a separate interconnect between the CPU and devices, and it adds complexity to the instruction set. MMIO on the other hand enables generic load and store instructions, making all address modes of such instructions immediately available for all device accesses, but it requires extra hardware in the address decoding logic on the shared bus

to figure out whether accesses affect memory or devices. PMIO may be used in systems where the memory address space is too small to accommodate IO devices. On some architectures such as the Intel x86, it is a relic of the past, which has been preserved for backwards compatibility. Most contemporary architectures, including the ARMv7-A architecture, exclusively make use of MMIO.

2.3 An updated model

2.3.1 Machine state

We redefine the machine state concept from Definition 1 based on the features of modern computer architectures. We add the complete set of general-purpose registers and configuration registers (also known as system registers). We introduce paged virtual memory by substituting the relocation-bounds register with a fine-grained memory map. We also extend the machine state with the state of IO devices.

Definition 3. The machine state S is defined by the memory E, the processor mode m, the program counter pc, the general-purpose registers G, the configuration registers C, the memory map A, the MMIO device state D^M and the PMIO device state D^P:

$$S \equiv \left\langle E, m, pc, G, C, A, D^M, D^P \right\rangle .$$

The first three parameters, namely the memory E, the current processor mode m, and the program counter pc, retain their meaning. To generalise the concept of processor mode, we define $\mathcal{M}_P$ to be the set of privileged modes,[1] and m_U to be the unprivileged mode, such that $\mathcal{M} = (\mathcal{M}_P \cup \{m_U\})$ and $m_U \notin \mathcal{M}_P$. We also introduce four new parameters: G contains the value of all general-purpose registers excluding the program counter.[2] Similarly, C contains the value of all configuration registers. We let E refer to the contents of the entire physical address space. Physical memory is now accessed using the virtual to physical address translation map A. Last but not least, D^M and D^P refer to the state of IO devices. We omit all IO device registers from E.

[1] Some architectures provide more than one privileged mode. We assume that all such modes are equally privileged.

[2] The ARM architecture offers a set of 16 "general-purpose" registers, which contains the program counter (PC) as R15 [54].

In our model, instructions that communicate with devices always alter either D^M or D^P. We also exclude any external influences from modifying D^M and D^P during the execution of any instruction; in other words, we consider instruction execution to be an atomic operation. The state of devices can only change in between the execution of instructions. This limitation of the model is required to model instruction behaviour accurately and independently of the timing behaviour of devices.

2.3.2 Address mapping

The address map A is a many-to-one map, meaning that many virtual addresses can correspond to the same physical address. Each entry in A also contains an access permission specifier, which may be used to restrict access to read-only, or can forbid access altogether. Without loss of generality, we can assume that all memory accesses are virtual. When virtual addressing is turned off, no translations are performed and A would be a one-to-one identity map.

Definition 4. An address map A is a set of 3-tuples (v, p, x) in which $v \in \mathcal{V}$ represents a virtual address, $p \in \mathcal{P}$ represents the physical address v is mapped to, and $x \in \mathcal{X}$ represents the access permission specifier. For any $v \in \mathcal{V}$, there is at most one 3-tuple in A that contains v.

Definition 5. Let $\mathcal{V}_A$ be the set of virtual addresses mapped by A:

$$\mathcal{V}_A = \{ v \mid v \in \mathcal{V} \wedge \exists (p, x) \in (\mathcal{P}, \mathcal{X}) : (v, p, x) \in A \} .$$

The translation function T_A for the address map A is then defined as:

$$\begin{array}{rccl} T_A : & \mathcal{V}_A & \longrightarrow & (\mathcal{P}, \mathcal{X}) \\ & v & \longmapsto & T_A(v) = (p, x). \end{array}$$

Upon every memory access, the address translation function $T_A(v)$ takes the virtual address v of the access and finds its corresponding physical address p using the address map A. If A does not contain an entry for v or if the access permission specifier x indicates that the requested kind of access is not allowed, a *memory trap* occurs.

Both an operating system and a hypervisor will create and use their own address map. When such an operating system is executed on top of a hypervisor, the operating system's physical addresses become virtual in the context of the hypervisor. The hypervisor is then responsible for

mapping guest physical addresses (GPAs) to host physical addresses (HPAs). Let A_g and A_h denote the address maps for the guest operating system and the hypervisor, respectively; we then become:

$$\forall (v_g, p_g, x_g) \in A_g : \\ p_g \in \mathcal{V}_{A_h} \Rightarrow \exists (p, x) \in (\mathcal{P}, \mathcal{X}) : T_{A_h}(p_g) = (p, x) \wedge x \leq x_g.$$

The hypervisor does not necessarily need to map all GPAs to a HPA at all times: when the mapping does not exist, a memory trap occurs which must be handled by the hypervisor. To enforce privilege isolation within the guest, we require that the hypervisor enforces access permissions at least as strict as those specified by the guest ($x \leq x_g$).

A typical MMU can only perform one address translation. With such an MMU, a double translation, as described above, would require one translation to be done in software, which in turn would require all guest memory accesses to trap, thereby sacrificing the efficiency property. This problem can be solved by composing the guest's mappings with hypervisor mappings into a set of direct mappings A_s, which translates the guest's virtual addresses directly into host physical addresses. Such address map is also known as a *shadow address map* [2, 21, 113].

Creation and maintenance of shadow address maps is one of the most complicated tasks of a hypervisor. The shadow address map must contain all virtual addresses mapped by the guest. Since it is common for guests to be relocated in the physical address space by the hypervisor, a new formal requirement has to be imposed on the allocator. Let E_n be the contents of the physical address space, and let A_g be the address map that translates virtual addresses $v \in \mathcal{V}_g$ to physical addresses in E_n on a machine without a hypervisor present. Let E_v and A_s denote the physical address space and the address map with a hypervisor installed. The formal requirement is that at any time, any virtual address v_g mapped by both A_g and A_s accesses the same physical data, or results in a memory trap for every access from the guest otherwise:

$$\forall (v_g, p_g, x_g) \in A_g : \\ v_g \in \mathcal{V}_{A_s} \Rightarrow \exists (p, x) \in (\mathcal{P}, \mathcal{X}) : T_{A_s}(v_g) = (p, x) \wedge x \leq x_g \\ \wedge \Big(E_v[p] = E_n[p_g] \vee x \text{ causes a memory trap} \Big). \quad (2.1)$$

In addition to the mappings from A_g, a hypervisor will add its own set of mappings to A_s. These hypervisor mappings comprise the memory

space occupied by the hypervisor and by MMIO devices. Because the hypervisor and its guests have to coexist in the same address space, those mappings may overlap. This is certainly the case for MMIO devices. It is the job of the hypervisor to maintain separation of guest and hypervisor resources by setting appropriate access permissions on the overlapping regions in the map A_s.

2.3.3 Instruction behaviour

This section redefines the notions of privileged, sensitive and innocuous instructions using the new machine state model. We adopt Definition 2 that states an instruction is a function that maps one machine state to another. We write $\mathcal{I}$ for the set of all instruction functions, and $\mathcal{S}$ for the set of all possible machine state mapping functions. Because typically not every machine state can be reached through instructions, $\mathcal{I} \subset \mathcal{S}$.

Definition 6. An instruction i is privileged if for any two states

$$S_1\langle E, m_U, pc, G, C, A, D^M, D^P\rangle \in \Sigma \quad \text{and}$$
$$S_2\langle E, m_p, pc, G, C, A, D^M, D^P\rangle \in \Sigma,$$

where $m_p \in \mathcal{M}_\mathcal{P}$ and both $i(S_1)$ and $i(S_2)$ do not cause any memory trap, $i(S_1)$ causes a trap and $i(S_2)$ does not. This kind of trap is referred to as a privileged instruction trap.

All instructions which change the processor mode, modify the system registers, modify the address map or communicate with PMIO devices are control-sensitive. We do not consider instructions that communication with MMIO devices to be sensitive, to prevent treating all generic memory access instructions as sensitive; instead, MMIO devices must be protected from guest accesses by the hypervisor's address map.

Definition 7. An instruction i is control-sensitive if there exists a state

$$S\langle E_1, m_1, pc_1, G_1, C_1, A_1, D^M, D_1^P\rangle \in \Sigma$$

such that for

$$i(S) = \langle E_2, m_2, pc_2, G_2, C_2, A_2, D^M, D_2^P\rangle$$

we have:

$$(m_1 \neq m_2 \vee C_1 \neq C_2 \vee A_1 \neq A_2 \vee D_1^P \neq D_2^P) \wedge i(S) \text{ does not cause a memory trap.}$$

Mode-sensitive instructions behave differently when executed in machine states which differ solely in their mode.

Definition 8. An instruction i is mode-sensitive if, given two states

$$S_1\langle E, m_1, pc, G, C, A, d^M, d^P\rangle \in \Sigma \quad \text{and}$$
$$S_2\langle E, m_2, pc, G, C, A, d^M, d^P\rangle \in \Sigma,$$

such that for some $m_1 \neq m_2$, and $i(S_1)$ and $i(S_2)$ do not cause a memory trap, for

$$i(S_1) = \langle E_1, m_1^*, pc_1, G_1, C_1, A_1, D_1^M, D_1^P\rangle \quad \text{and}$$
$$i(S_2) = \langle E_2, m_2^*, pc_2, G_2, C_2, A_2, D_2^M, D_2^P\rangle$$

we have:

$$E_1 \neq E_2 \vee \Big((m_1^* \neq m_2^*) \wedge (m_1^* \neq m_1 \vee m_2^* \neq m_2)\Big) \vee pc_1 \neq pc_2$$
$$\vee\, G_1 \neq G_2 \vee C_1 \neq C_2 \vee A_1 \neq A_2 \vee D_1^M \neq D_2^M \vee D_1^P \neq D_2^P.$$

We introduce a new class of sensitive instructions, similar to mode-sensitive instructions. Their behaviour depends on the set of configuration registers C:

Definition 9. An instruction i is configuration-sensitive if, given two states

$$S_1\langle E, m, pc, G, C_1, A, D^M, D^P\rangle \in \Sigma \quad \text{and}$$
$$S_2\langle E, m, pc, G, C_2, A, D^M, D^P\rangle \in \Sigma$$

such that for some $C_1 \neq C_2$, and $i(S_1)$ and $i(S_2)$ do not cause a memory trap, for

$$i(S_1) = \langle E_1, m_1, pc_1, G_1, C_1^*, A_1, D_1^M, D_1^P\rangle \quad \text{and}$$
$$i(S_2) = \langle E_2, m_2, pc_2, G_2, C_2^*, A_2, D_2^M, D_2^P\rangle$$

we have:

$$E_1 \neq E_2 \vee m_1 \neq m_2 \vee pc_1 \neq pc_2 \vee G_1 \neq G_2$$
$$\vee \Big((C_1^* \neq C_2^*) \wedge (C_1^* \neq C_1 \vee C_2^* \neq C_2)\Big)$$
$$\vee\, A_1 \neq A_2 \vee D_1^M \neq D_2^M \vee D_1^P \neq D_2^P.$$

Popek and Goldberg also introduced the notion of location-sensitive instructions, which were able to bypass memory relocation. An example of a such sensitive instruction is the *Load Real Address* (`LRA`) instruction from the IBM 360/67 instruction set. In a modern architecture, such instructions would bypass address translation and expose absolute physical addresses, i.e., they expose the contents of the address map A. In virtualised systems it is common for guests to be relocated in physical memory by the hypervisor. Location-sensitive instructions would therefore break the equivalence property. We are not aware of any such instructions in modern processor architectures including MIPS, PowerPC, SuperH and even x86, with the exception of the address translation operations in the ARMv7-A architecture [12, 71, 74, 91–93, 117].

At first sight, a new difficulty arises on architectures that have an explicitly visible program counter. Use of the program counter as an operand in any instruction may break the equivalence property if a guest is relocated in memory by a hypervisor. However, current memory management techniques enable a hypervisor to map its guests to the same virtual addresses as they would see when executed natively. Given that requirement (2.1) in Section 2.3.2 is met, this will be the case, and any instructions that operate on the program counter will have no influence on the virtualisability of a system.

2.3.4 Events

Multiple definitions exist for the terms *interrupt* and *exception*. On ARM, for example, interrupts are seen as a particular subclass of exceptions [12], while the x86 manual describes them as different and unrelated types of *events* [75]. From this point on, we refer to the whole set of interrupts and exceptions as *events*.

Events cause the processor to save its current state and enter an event handler in a privileged mode. Events are either synchronous or asynchronous. A *synchronous* event directly results from the execution of an instruction. Synchronous events are already included in our model as traps. In modern architectures, several kinds of traps exist. They can usually be classified in one of the following categories:

- *Privileged* instruction traps are the result of executing a privileged instruction.
- *Memory* traps are caused by instructions or instruction fetches

attempting to access an address that is either invalid or inaccessible with respect to the current machine state S and access bits X.

- *Arithmetic* traps may occur when attempting to perform invalid arithmetic operations (such as division by zero), or when a hardware FPU lacks an implementation for a requested floating point operation, in which case software must emulate the operation.
- *Undefined instruction* traps may occur when executing an instruction which is not recognised by the processor.

An *asynchronous* event may happen at any time, unrelated to the instruction being executed. By definition, all interrupts generated by IO devices are asynchronous. When an asynchronous event happens while the processor is executing an instruction, it will cause the processor to revert or to complete that instruction. If not, the processor would be left in an inconsistent state. All instruction functions i are therefore independent of such events.

Definition 10. An asynchronous event is a function e that brings a processor from one machine state into another—the latter of which the mode is always privileged—without and unrelated to the execution of an instruction:

$$\begin{array}{rccl} e: & \Sigma & \longrightarrow & \left\{ S\langle E, m, pc, G, C, A, D^M, D^P\rangle \mid (S \in \Sigma) \wedge (m \in \mathcal{M}_P) \right\} \\ & S_x & \longmapsto & e(S_x) = S_y. \end{array}$$

As such, asynchronous events can be expressed in the same way as the execution of instructions. We write $\mathcal{E}$ for the set of asynchronous event functions. Because the mode of the machine state after an asynchronous event is restricted to the set of privileged modes $\mathcal{M}_P$, we find that $\mathcal{E}$ is a proper subset of $\mathcal{S}$.

2.3.5 Result

Based on our updated model and its definitions, we conclude that Theorem 1 remains unmodified. However, we need to impose a new formal constraint on the allocator of the hypervisor (see (2.1) in Section 2.3.2) in order to support paged virtual memory.

The proof of the theorem also requires a subtle update. The consequence of adding asynchronous events to the model is that for any

instruction sequence $(i_1, i_2, \ldots i_n)$, in which neither instruction generates a synchronous event, we can no longer guarantee that $S_{k+1} = i_k(S_k)$ $(1 \leq k < n)$, because asynchronous events may occur at any given time. The same observation applies to changes in device state due to timing effects or external influences. Although both the model developed by Popek and Goldberg and our extended model do not rely on sequences of instructions, the proof of the theorem that builds upon this model does. More precisely, the formalisation of the equivalence property as a homomorphism on the set of machine states Σ does rely on the behaviour of sequences of instructions. However, as we have shown, asynchronous events can be modelled as functions that map one machine state onto another, similar to instructions. The same reasoning can be applied to asynchronous changes in IO device state. It suffices to generalise the notion of instruction functions $i \in \mathcal{I}$ to all functions in $(\mathcal{E} \cup \mathcal{I}) \subset \mathcal{S}$ to formalise the equivalence property under the new model.

2.4 Analysis of the ARM architecture

In this section, we present an analysis of the bare ARMv7-A architecture [12], i.e., without any extensions, based on our updated model. An analysis of ARMv7-A including the virtualisation extensions follows in Section 2.5.1. ARM provides four different instruction sets: 32-bit (fixed-width) ARM, mixed 16-bit and 32-bit width Thumb-2, Thumb execution environment (ThumbEE) and Jazelle. We will omit ThumbEE and Jazelle from our discussion. ThumbEE resembles Thumb-2, but is designed for dynamically generated code. It is however deprecated with the upcoming virtualisation extensions. Jazelle enables hardware execution of Java bytecode, but its specification is not publicly available, and it cannot be combined with the upcoming virtualisation extensions [9, 13].

2.4.1 Machine state

We recall the definition of machine state in our new model:

$$S \equiv \left\langle E, m, pc, G, C, A, D^M, D^P \right\rangle .$$

In the ARM architecture, multiple equally privileged modes exist; they are system mode (SYS), supervisor mode (SVC), interrupt mode (IRQ), fast interrupt mode (FIQ), abort mode (ABT) and undefined mode (UND).

There is only one unprivileged mode: user mode (USR). m is always one of these modes. The mode can be altered explicitly through instructions, or it can be changed automatically when an event arrives [12].

At any given time, 16 "general purpose" registers are available, labelled R0 to R15. We exclude R15 from G, because it represents pc. Some registers are duplicated for different modes; this technique is called *banking*. Non-banked registers are shared between all processor modes. Banked registers are only accessibly when the processor is in the appropriate mode, and in some cases through dedicated instructions that save and restore banked registers. The stack pointer register (R13) and the link register (R14) are banked for all privileged modes; the instances for SYS are however shared with USR. Furthermore, FIQ has its own banked set of R8 to R12. The set G includes all mentioned registers, excluding pc.

The currently active processor state is represented in the current program status register (CPSR). It stores the processor mode m, data memory endianness and interrupt mask bits among other fields. There are also five saved program status registers (SPSRs). These SPSRs are banked and all share their layout with the CPSR register. They are used to back up the CPSR upon entering an event handler.

Although it sounds logical to include all program status registers within the set of configuration registers C, there are two pitfalls. Firstly, because the mode field of the CPSR always reflects the current mode m, including m in C would render Definition 8 of mode-sensitive instructions pointless. Secondly, one part of the CPSR reflects global state, while the other part represents state specific to the current mode m. The latter part is also exposed to user mode; e.g., data memory endianness can be set by software running in user mode, and it will only affect user mode.

We clearly cannot add the entire CPSR to the set of configuration registers C, as it would make our analysis overly restrictive. Altering the mode-specific state for executing instructions in a specific mode m_S while the processor is executing instructions in another mode m, is always done by manipulating the SPSR for mode m_S, eventually followed by an exception return instruction to switch the processor back to the mode m_S. Hence, we can safely omit any mode-specific state from the CPSR in C, but we must include the SPSRs completely in C.

The set C also contains all of the system control coprocessor (CP15) registers, and an event register. The system control coprocessor is used for various functions such as cache maintenance, MMU control, performance monitors and whole system configuration. Its registers can be

Table 2.1: Sensitive and privileged 32-bit ARM instructions

	Sensitivity				
Instruction	**Control**	**Mode**	**Conf.**	**Loc.**	**Priv.**
CPS	●	●	○	○	○
LDC	○	●	●	○	○
LDM (exception return)	●	●	●	○	○
LDM (user registers)	○	●	○	○	○
MCR	●	●	○	●	○
MRC	○	●	●	○	○
MRS (SPSR)	○	●	●	○	○
MSR	●	●	○	○	○
RFE	●	●	○	○	○
SEV	●	○	○	○	○
SRS	○	●	●	○	○
STC	●	●	○	○	○
STM (user registers)	○	●	○	○	○
SVC	●	○	○	○	●
SUBS, ... (exception return)	●	●	●	○	○
WFE	●	○	●	○	○
WFI	●	○	●	○	○

accessed using dedicated coprocessor register read or write instructions. The event register is part of the mechanism of halting and resuming instruction execution based on machine specific system events and should not be confused with our usage of the term *event* introduced in Section 2.3.4 for the whole of interrupts and exceptions.

The ARMv7-A architecture provides an MMU with support for paged virtual memory.[3] Hence, address translation maps can be implemented using page tables. As all IO on ARM is memory-mapped, PMIO is not supported and D^P is empty. We do not treat interactions with coprocessors as IO operations.

2.4.2 32-bit ARM instruction behaviour

There are many sensitive instructions on ARM. Below, we provide a detailed analysis of all sensitive instructions, grouped by purpose. The results of our analysis are summarised in Table 2.1.

Coprocessor instructions

The ARM instruction set contains a number of instructions to interact with coprocessors. In this analysis, we limit the discussion to a basic implementation of the ARM architecture without extensions. Hence, only two coprocessors are available in the system: CP14, which is used for debugging and tracing, and CP15, the system control processor. All their registers are part of the set C. Out of all instructions there are only four that can operate on these coprocessors:

- `MCR` and `STC` are control-sensitive because they write data to a coprocessor's registers or memory;
- `MRC` and `LDC` are configuration-sensitive because they load data from a coprocessor's registers or memory.

Since some forms of the `MCR` instruction can be used to translate virtual addresses to physical addresses, they expose the address mapping. Such forms of the `MCR` instruction are therefore location-sensitive. A coprocessor can also deny accesses from USR mode. Therefore, all of the above instructions are also mode-sensitive.

In systems with extra coprocessors, these coprocessors have to be carefully analysed to determine whether any of their registers belong to C. The specification of the coprocessor also determines the set of instructions that can operate on it. This set may not be limited to the instructions discussed above; it may also include any of the following: `CDP`, `CDP2`, `LDCL`, `LDC2`, `LDC2L`, `MCR2`, `MCRR`, `MCRR2`, `MRC2`, `MRRC`, `MRRC2`, `STCL`, `STC2` and `STC2L`. The effect of each of these instructions, including the ones mentioned above, must be studied for each coprocessor individually to determine which instructions are sensitive.

[3] The application profile is the only architecture that provides an MMU. Other ARM architecture profiles provide a simpler memory protection scheme that does not support address translation.

Event handling

ARM provides a plethora of instructions for handling events, which are all mode-sensitive by definition:

- `LDM`, `SUBS`, `MOVS`, `MVNS`, `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `ORRS`, `RSBS`, `RSCS` and `SBCS` can all be used to return from event handlers. Since this operation updates the processor mode by definition, all these instructions are control-sensitive. These instructions are also configuration-sensitive, because they copy the SPSR into the CPSR.

- Another variant of `LDM` can be used to load values into USR mode registers from memory. A similar form of `STM` provides the inverse operation.

- `RFE` is control-sensitive because it loads values into the program counter (R15) and the CPSR from memory.

- `SRS` is configuration-sensitive because it stores the value of the link register (R14) and the current SPSR to memory.

All of the above instructions are also mode-sensitive, because their behaviour in USR mode is unspecified.

Direct modification of system registers

Some instructions directly read or write to system registers such as the CPSR or one of its banked versions:

- `MRS` reads from the SPSR, hence it is both configuration-sensitive and mode-sensitive.

- `CPS` and `MSR` write to system registers. The former acts as a `NOP` while the latter is unpredictable when executed in USR mode. Hence, they are both control-sensitive and mode-sensitive.

- `SVC` is used as system call (or software interrupt) by applications to call the operating system. It unconditionally changes the current mode to supervisor mode (SVC). Hence, it is control-sensitive.

Table 2.2: Sensitive and privileged Thumb-2 instructions

	Sensitivity				
Instruction	**Control**	**Mode**	**Conf.**	**Loc.**	**Priv.**
CPS	●	●	○	○	○
LDC	○	●	●	○	○
MCR	●	●	○	●	○
MRC	○	●	●	○	○
MRS (SPSR)	○	●	●	○	○
MSR	●	●	○	○	○
RFE	●	●	○	○	○
SEV	●	○	○	○	○
SRS	○	●	●	○	○
STC	●	●	○	○	○
SVC	●	○	○	○	●
SUBS (exception return)	●	●	●	○	○
WFE	●	○	●	○	○
WFI	●	○	●	○	○

Sleep and wake-up

In a multiprocessor or multi-core system, software executing on different processors or cores can communicate using events. The architecture provides two instructions for this purpose: send event (SEV) and wait for event (WFE). Software can also hint the processor that it is waiting for an interrupt or external event through the WFI instruction. While waiting, the processor may go to a low-power state. After an external event or interrupt occurs (even if interrupts are masked in the CPSR), the one-bit abstract event register is asserted, the processor wakes up and this bit is cleared. Since the abstract event register is part of the system state, the SEV, WFE and WFI instructions are control-sensitive.

2.4.3 Thumb-2 instruction behaviour

The Thumb-2 instruction set provides more or less the same instructions as the 32-bit ARM instruction set. Moreover, the set of sensitive Thumb-2 instructions is a proper subset of the set of sensitive ARM instructions, so no elaborate discussion is required. The results of our analysis are summarised in Table 2.2.

2.4.4 Conclusion

Investigating the set of sensitive instructions in the ARM and Thumb-2 instruction sets in Tables 2.1 and 2.2, it is clear that both instruction sets contain sensitive instructions that are not privileged. Based on our findings in Section 2.3.4, we conclude that the ARMv7-A architecture is not classically virtualisable.

2.5 Full virtualisation in practice

The criteria for virtualisability introduced by Popek and Goldberg [97] and extended in this chapter are sufficient but not necessary to construct an efficient hypervisor for a particular architecture. Advances in the construction of hypervisors have enabled full virtualisation on architectures that fail these criteria. However, pure software solutions using binary rewriting or emulation are typically deemed to introduce too much overhead or to be too complex.

Hardware vendors have also adapted to the need for virtualisation support, and architectures that were formerly not classically virtualisable such as x86 have already been made virtualisable [3, 127]. ARM is following the same path with its upcoming virtualisation and large physical address (LPA) extensions for ARMv7-A and ARMv7-R [6–9, 13].

In this section, we discuss both software and hardware approaches. For the latter, we analyse ARM's upcoming extensions.

2.5.1 Hardware support for full virtualisation

The upcoming virtualisation and LPA extensions introduce a new processor mode, referred to as HYP mode, and a number of new instructions. They also impact the behaviour of many existing instructions, and modify the mechanisms for interrupt handling, memory management and performance monitoring.

The new HYP mode is more privileged than the original set of privileged modes (SVC, SYS, ABT, UND, IRQ, FIQ); the latter is now referred to as the set of *kernel modes*. HYP mode enables a hypervisor to run below the operating system level without forcing the operating system kernel to run unprivileged. Instead, the operating system kernel can use all kernel modes transparently, as if no hypervisor was present. Events for the hypervisor are handled in the HYP mode, instead of the traditional

exception modes (ABT, UND, IRQ, FIQ), which are a subset of the kernel modes. This significantly reduces the set of sensitive instructions.

In order to make all sensitive instructions trap to HYP mode, when executed in any of the kernel modes, the virtualisation extensions add *configurable traps* to the architecture. A hypervisor can then make certain instructions trap as required. When running just one operating system without a hypervisor, the traps are disabled.

A quick analysis of the original set of sensitive instructions confirms the implications stated above:

- Coprocessor instructions can be configured to trap. These traps provide coarse-grained control over accesses to coprocessors CP0 to CP13. There are more fine-grained controls for the remaining coprocessors—the debug and execution environment support coprocessor (CP14) and the system control coprocessor (CP15).
- Memory access and event handler return instructions are no longer sensitive, since a guest running on a hypervisor with hardware extensions can use the exception modes in the same way as for the non-virtualised case.
- Instructions that directly modify system state now act on a guest's state, rather than on the entire system state. Hence, they are no longer sensitive.
- Instructions that deal with external events are adapted to work with a per-guest interrupt state. Each guest has its own virtual interrupts. Both `WFE` and `WFI` have independent configurable traps. These traps can be used as hints by the hypervisor to schedule other guests. However, the `SEV` instruction *cannot* be made to trap.

It may seem surprising that `SEV` cannot be made to trap. It is the only sensitive instruction that remains unprivileged with the new hardware extensions. However, none of the sleep and wake-up instructions can cause functionally incorrect behaviour of the hypervisor [12]. The configurable traps for `WFE` and `WFI` are provided so that a hypervisor is able to detect when a guest is idle. The `SEV` instruction cannot provide such useful information to the hypervisor.

Hence, a hypervisor can be constructed that executes VMs until they trap to HYP mode. All other additions by the virtualisation and LPA extensions are not strictly necessary but speed up common hypervisor

tasks and reduce the code size of the hypervisor. For example, the LPA extension adds a nested address translation mechanism in the MMU. This mechanism makes the hardware capable of combining a translation table for the guest with a translation table for the hypervisor, eliminating the need for software-based shadow maps.

2.5.2 Dynamic binary translation

Full virtualisation can also be supported without modifications to the hardware; this is typically achieved through DBT, sometimes also called software dynamic translation (SDT). The concept of a DBT hypervisor evolved from the theory of Popek and Goldberg's hybrid hypervisor. An analysis according to our model remains useful, because it can be used to determine whether an architecture is suitable for the construction of such DBT hypervisor. Furthermore, the analysis will reveal which instructions will need to be rewritten.

In a hybrid hypervisor, all instructions normally executed in a privileged mode are interpreted by the hypervisor. Instructions normally executed in unprivileged mode are executed natively [97]. In a traditional software stack, a hybrid hypervisor would interpret the guest operating system but execute applications natively. Popek and Goldberg [97] proved that a hybrid hypervisor can be constructed if all user-sensitive instructions are also privileged. In our model, user-sensitive instructions are defined as follows:

Definition 11. An instruction i is user-sensitive if there exists a state $S\langle E, m_U, pc, G, C, A, D^M, D^P\rangle \in \Sigma$ for which $i(S)$ is control-sensitive, configuration-sensitive or location-sensitive.

A DBT hypervisor can achieve a performance benefit over interpretation by rewriting instruction sequences at run time [19]. The basic idea is to execute as many rewritten instructions as possible natively, without intervention of the hypervisor. Sensitive instructions must be rewritten such that they trap to the interpreter of the hypervisor. Because the hypervisor needs to keep track of the execution of its VMs, it will also need to rewrite control flow instructions.

DBT for the ARM architecture

In order to construct a DBT hypervisor for ARM, it is required to determine how much of the guest code must be rewritten: if all user-sensitive

instructions are also privileged, only guest privileged code must be translated, otherwise, all code must be translated. We can extend our analysis from Section 2.4 to analyse user-sensitivity. As it turns out, there are four user-sensitive instructions, shared by both ARM and Thumb:

- the supervisor call instruction `SVC`, because it always changes the processor mode to SVC;
- the sleep and wake-up instructions `SEV`, `WFE` and `WFI`.

All other sensitive instructions either act as `NOP` or exhibit innocuous behaviour when executed in USR mode. Of all user-sensitive instructions, only `SVC` is privileged. Hence, ARM does not meet Popek and Goldberg's conditions for a hybrid hypervisor either.

In practice, the sleep and wake-up instructions are merely hints for a processor. Even a masked interrupt will wake up a sleeping processor. Furthermore, if any interrupts are pending, the processor will not sleep either [12]. Not intercepting these operations can therefore not lead to resource control violations. In other words, we can ignore them and still obtain functional correctness. When we effectively do ignore the presence of unprivileged sleep and wake-up instructions, we can construct a DBT hypervisor that only translates privileged guest code. The downside of this approach is that the hypervisor cannot intercept sleep instructions in unprivileged guest code, which could otherwise be used by the VM to tell the hypervisor that it is idle. The same problem applies to paravirtualisation solutions for ARM: because only guest privileged code (such as operating system kernels) is altered, they have to make the same assumption. They hence suffer from the same deficiency.

Using DBT also requires the hypervisor to keep track of guest control flow. On ARM, the PC is explicitly visible as R15 and can be altered by several instructions, including arithmetic logic unit (ALU) instructions. This complicates the design of an instruction decoder for DBT.

Furthermore, the PC can be used as source operand for even more instructions [12, 60, 94]. Because the width of instructions is fixed at 16 or 32 bits, absolute address operands cannot be encoded. Therefore, distributed literal pools—pools of data embedded in code—are used which are accessed using PC-relative addressing. Other architectures offer instructions that can encode absolute addresses in full, such as on x86, or use a single global offset table (GOT) instead of distributed embedded data pools, such as on Alpha [84].

We call instructions that depend on the value of the program counter *PC-sensitive*. Formally, they are defined as follows:

Definition 12. An instruction i is PC-sensitive if, given two states

$$S_1\langle E, m, pc, G, C, A, D^M, D^P\rangle \in \Sigma \qquad \text{and}$$
$$S_2\langle E, m, (pc+z), G, C, A, D^M, D^P\rangle \in \Sigma$$

such that for some offset $z \in \mathbb{Z} \setminus \{0\}$, and $i(S_1)$ and $i(S_2)$ do not cause a memory trap, for

$$i(S_1) = \langle e_1, m_1, p_1, g_1, c_1, a_1, d_1^M, d_1^P\rangle \quad \text{and}$$
$$i(S_2) = \langle e_2, m_2, p_2, g_2, c_2, a_2, d_2^M, d_2^P\rangle$$

we have:

$$e_1 \neq e_2 \vee m_1 \neq m_2 \vee p_1 \neq (p_2 - z) \vee g_1 \neq g_2 \vee c_1 \neq c_2 \vee a_1 \neq a_2 \vee d_1^M \neq d_2^M \vee d_1^P \neq d_2^P.$$

This class of instructions becomes important when part or all of a VM is relocated in the *virtual* address space; as opposed to Popek and Goldberg's location-sensitive instructions which matter when part or all of a VM is relocated in the *physical* address space. PC-sensitive instructions are innocuous to execute-to-trap hypervisors as long as requirement (2.1) on the address map is met (see Section 2.3.2).

PC-sensitive instructions wreak havoc with DBT hypervisors because privileged guest code may not always be executed in place, depending on the construction of the hypervisor, as opposed to unprivileged guest code. As we will show in Section 3.5.1, in-place translation and execution may not be feasible for several reasons: firstly, a hypervisor cannot prevent a guest from observing the alterations to its code, and secondly, the size of the translated code might not match the size of the original code. Instead, code is typically translated to a cache located elsewhere in the virtual address space. The program counter observed by the translated code will therefore be incorrect unless the hypervisor also intercepts PC-sensitive instructions. Optimisation techniques can be used to rewrite PC-sensitive instructions to equivalent non-sensitive instruction sequences, instead of merely replacing them by a trapping instruction [94].

2.6 Conclusions

Despite the popularity of paravirtualisation in today's embedded systems, full virtualisation remains important. The theory introduced by Popek and Goldberg is still useful for determining whether an architecture is suitable for the construction of efficient hypervisors for full virtualisation, which are based on the execute-to-trap principle. However, their model does not take into account features often found in modern architectures. Therefore we have extended their model with paged virtual memory by introducing the concept of an address map, and we derived a new formal constraint for the correctness of such maps. We also studied the effects of IO and events, and updated the model, definitions and results accordingly.

Our model can be applied to analyse modern computer architectures. We have demonstrated this in a formal analysis of ARMv7-A, which proved to be not classically virtualisable. Nevertheless, modern techniques in the construction of hypervisors such as DBT can enable full virtualisation on ARM. DBT requires neither hardware changes nor guest modifications and provides a solution to the lack of classic virtualisability on architectures such as ARM. In other architectures, DBT has already been shown to be able to match or even outperform the hardware assisted virtualisation approach. Even though it comes at a price of software implementation complexity and increased memory footprint, DBT technology opens the door to several more interesting possibilities such as cross-architecture virtualisation, legacy software stack support and optimising heterogeneous multi-core system utilisation.

In the mean time, industry has started to adapt to the interest in full virtualisation. Hardware virtualisation extensions attempt to make ARMv7-A compatible with the Popek and Goldberg classic virtualisation requirements, and implement certain parts of the hypervisor functionality in hardware for efficiency reasons sacrificing portability. Utilising the hardware functionality can reduce hypervisor design complexity at the cost of moving the complexity to the design of the hardware layer.

Chapter 3

The STAR hypervisor

In Chapter 2 we pointed out that the ARMv7-A architecture can be virtualised both by using DBT techniques and by relying on the new virtualisation extensions. DBT remains relevant for a number of scenarios such as legacy emulation, full system tracing and even for load balancing on heterogeneous multi-core architectures. However, none of the existing hypervisors for the ARMv7-A architecture support DBT.

In this chapter, we present the software translation for ARM (STAR) hypervisor, the first software-only hypervisor for the ARMv7-A architecture. Our hypervisor is a bare-metal hypervisor that fully virtualises the ARMv7-A architecture on top of an ARMv7-A-based platform. It runs unmodified guest operating systems, decoupled from the hardware through DBT, and uses shadow translation tables to virtualise the MMU.

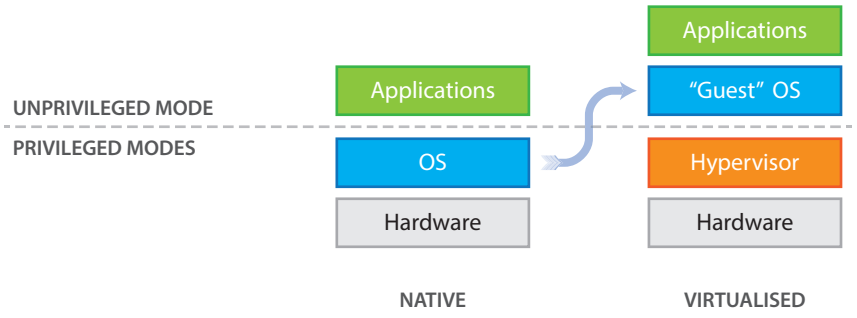

Figure 3.1: Native vs. virtualised privilege levels

3.1 Introduction

To have a framework to experiment with DBT on ARM, we needed either to extend an existing hypervisor or to create a new hypervisor from scratch. It was desirable for that hypervisor to be a bare-metal hypervisor, as we target embedded systems and therefore the overhead of hosted hypervisors is to be avoided. At the time when our research project started, all available bare-metal hypervisors for ARM used paravirtualisation for both CPU and MMU virtualisation. As such hypervisors were not suitable, we decided to create a new bare-metal hypervisor from scratch.

The goal of full virtualisation is to run a complete and unmodified software stack consisting of an operating system and user applications on top of a hypervisor. Because the hypervisor needs to remain in control of the hardware at all times, it needs to restrict access of guest operating systems to the hardware in the same way a traditional operating system separates application privileges from kernel privileges. The basic ARMv7-A architecture offers only two privilege levels. There are several equally privileged CPU modes, but only one unprivileged mode, *user* mode. In a traditional operating system, the privileged modes are used for the kernel, and the unprivileged mode is used for user applications. When virtualised, only the hypervisor is privileged and guest operating systems together with their user applications must be executed in the unprivileged mode, as shown in Figure 3.1. The hypervisor thus gains the additional responsibility of enforcing privilege separation between the guest operating system kernels and their user applications.

In order to make this work, the hypervisor has to solve two important problems. Firstly, as discussed in Chapter 2, executing a guest operating system kernel in the unprivileged mode causes all sensitive instructions

to behave differently, because the ARMv7-A architecture is not classically virtualisable. Instructions from guest kernels must therefore be translated. There are, however, no differences in instruction behaviour for user applications. The guest's original user-space code can hence be executed as is, and the DBT engine only has to translate kernel code.

Secondly, the MMU must be shared between the hypervisor and its guests: both the hypervisor and its guests will create their own memory mappings; furthermore, the hypervisor must be able to supervise and modify its guests' mappings as needed. This requires an MMU virtualisation mechanism. These two problems define the tasks of the two largest components of our hypervisor: the DBT engine, responsible for CPU virtualisation, and the virtual MMU.

Our hypervisor aims to be usable for several different use cases, including full-system debugging and instrumentation. The naive version of our hypervisor must therefore avoid techniques that cause guest-observable differences in behaviour. In Chapter 4, we present optimisations over our naive version, which can be enabled or disabled based on the usage scenario of the hypervisor as some of these optimisations may, in rare cases, cause observability issues.

3.2 Development history

Some of the design and implementation choices made in our hypervisor are closely related to its history and its initial development platform. We therefore need to revisit how our hypervisor came about before getting into the details of the different components and how they interact.

Our hypervisor started in 2009 as a bachelor's thesis project of two students at The University of Manchester, Alex Merrick and Danielius Kudinskas, under supervision of Alasdair Rawsthorne, former CTO of Transitive [80, 89]. Transitive was a company that provided cross-architecture user-space virtualisation solutions, such as the Rosetta technology that helped Apple users migrate from PowerPC-based computers to Intel x86-based computers [5, 123].

The original goal of the project was to create a prototype of an open and free hypervisor for ARM-based embedded systems, motivated by emerging trends in virtualisation of mobile devices and the fact that no hypervisor was available at the time that could do full virtualisation. In the context of mobile device virtualisation it only made sense to

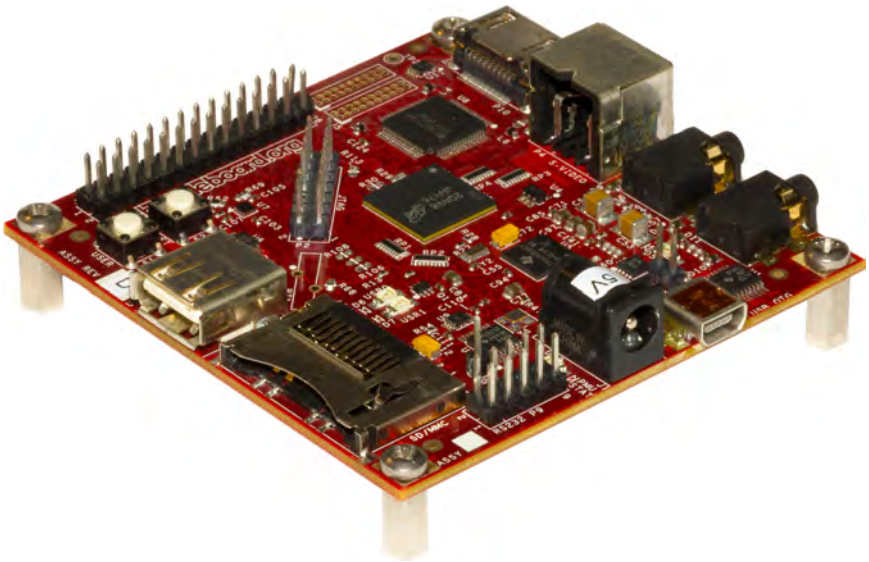

Figure 3.2: The Texas Instruments OMAP3-based BeagleBoard

start working on a platform that was suitable for such devices. Proper development hardware for the ARM architecture with full debug and trace capabilities is not affordable in the context of such projects, e.g., the ARM Versatile Express hardware platform costs several thousands of euros.[1] The BeagleBoard, a low-cost single board computer shown in Figure 3.2, was chosen as development platform because it is both affordable and based on a system-on-chip (SoC) used in several mobile handsets: the Texas Instruments OMAP3 SoC [37, 119, 120].

The OMAP3 SoC contains one single ARM Cortex-A8 core, together with a DSP and a set of peripherals. The Cortex-A8 core implements the ARMv7-A architecture [10]. The BeagleBoard integrates this SoC on a board with 256 MiB of random access memory (RAM), 256 MiB of NAND flash as well as a number of debug and expansion interfaces.

Based on the aim for full virtualisation, the VM interface presented to guests was designed to mimic the features of the BeagleBoard and its OMAP3 SoC. The hypervisor thus presents an OMAP3 SoC to guest systems, together with a virtual Cortex-A8 core. While our hypervisor has now been ported to different platforms, the virtual hardware has remained the same. Over time, the hypervisor has been redesigned such

[1] Based on private communication with ARM Ltd.

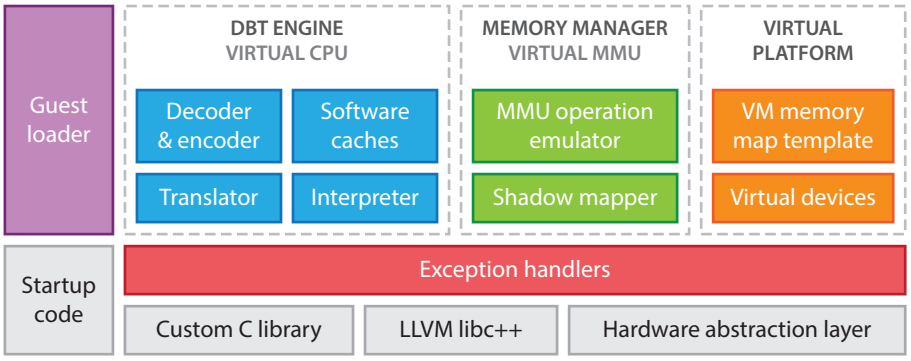

Figure 3.3: Overview of the functional blocks of the hypervisor run-time

that the virtual interfaces offered to VMs can be easily decoupled from the underlying physical platform.

The hypervisor originally started in Manchester became the subject of this thesis after the bachelor's project ended. Because at this stage, the hypervisor could only run the Linux kernel decompression loop, further development took place both in Manchester, by Danielius Kudinskas, and in Ghent, by the author of this dissertation. The remainder of this text, unless explicitly stated otherwise, discusses the current version of the hypervisor based on the work of the author of this dissertation.

3.3 Top-level design

Figure 3.3 presents an overview of the run-time functional blocks of our hypervisor. The DBT engine and virtual MMU make an abstraction of the processor architecture and memory. The virtual platform provides a set of virtualised coprocessors and IO devices. The state of all these functional blocks is unified in a data structure called the *guest context*.

The virtual platform should in theory be a copy of a real physical platform. In practice, however, hypervisors often take some shortcuts for performance reasons or to avoid complexity, even with full virtualisation [113]. One example of such a shortcut is our guest loader. On normal hardware, an operating system is loaded by a third-party primary boot loader such as Barebox [20] or U-Boot [45]. The primary boot loader typically performs some low-level hardware initialisation before jumping into the operating system kernel. In a virtualised system, low-level hardware initialisation happens once, i.e., before the hypervisor

is started or during hypervisor start-up, depending on whether or not a primary boot loader is used on the physical platform. This step is necessary after each reset of the system, because the physical platform dictates the default configuration after reset. In the virtual platform, the hypervisor dictates what is default; therefore it can choose to initialise this platform to a state suitable for booting a guest kernel immediately, without intervention of an external primary boot loader. The part of our hypervisor that prepares the guest context for booting a guest kernel in this way is called the guest loader.

Our hypervisor is a bare-metal hypervisor, i.e., it runs on top of hardware rather than on top of an operating system. Therefore, at the lower layers of abstraction, our hypervisor also contains exception handling code, responsible for context switches between the hypervisor and its VMs, a hardware abstraction layer, consisting of device drivers and support code for architectural features, our own customised C library, and a C++ standard library (LLVM's *libc++* [86]).

The hypervisor is constructed in a modular way, such that all runtime components are chosen, configured, and composed at build time. All builds are configured through a slightly modified version of Kconfig, a configuration language and a set of build-time configuration tools adopted from the Linux kernel [79, 108]. Kconfig is used to configure the target platform, to enable or disable specific features such as optimisations in the DBT engine, to switch between different implementations of a single feature, and to configure debug and test features.

3.4 MMU virtualisation: the memory manager

Operating systems use an MMU for various tasks ranging from address translation and memory protection to cache configuration. All these tasks are administered through translation tables. A hypervisor essentially performs the same tasks: it uses address translation and cache control for itself, and for relocating guests within physical memory. The hypervisor needs to isolate its own memory from guests, and guests from one another. A guest's memory may need to be protected to keep the hypervisor's software caches consistent with the guest. Furthermore, the hypervisor needs complete control over the hardware caches. The hypervisor must therefore set up its own translation tables, which must be used in combination with the guests' translation tables.

The MMU in the bare ARMv7-A architecture can only use one set of

translation tables at a time. Combining the hypervisor's translation tables with a guest's translation tables requires a software mechanism that merges the tables into a single set of *shadow* translation tables [2, 21, 113].

Operating systems enforce privilege separation between kernel and user space by specifying different permissions for memory accesses from privileged and unprivileged modes in their translation tables. As shown in Figure 3.1, once virtualised, both the guest kernel and its user applications run in the unprivileged mode. The MMU can therefore no longer distinguish between a guest's kernel and its user applications. To enforce privilege separation in the guest, the hypervisor creates two sets of shadow translation tables: a privileged set, which is set up to enforce access permissions for the guest's kernel, and an unprivileged set, configured to enforce access permissions for guest user applications. The MMU is configured by the hypervisor with the appropriate set of shadow translation tables depending on the guest's virtualised privilege level. This approach is known as *double shadowing* [47].

Because our hardware did not support ARM's large physical address extension (LPAE), we only discuss the *short descriptor model*, i.e. the MMU configuration model used prior to the introduction of the LPAE [7]. We use VMSAv7 to denote this memory management architecture, and LPAE to denote the revised architecture which adds support for the *long descriptor model*. The short and long descriptor models are substantially different in translation table structure, descriptor format, MMU configuration and size of the physical address space. In VMSAv7, all physical and virtual addresses are 32 bits wide. Both physical and virtual address spaces are therefore limited in size to 4 GiB.

3.4.1 The VMSAv7 MMU

The VMSAv7 MMU supports paged virtual memory using two levels of translation tables [12]. A first-level translation table maps the entire virtual address space. It consists of 4096 descriptors that each map 1 MiB of the address space. A second-level translation table, also called a *page table*, can be used to split one first-level descriptor into 256 second-level descriptors mapping 4 KiB each. This is shown in Figure 3.4.

The two levels of translation tables support mappings of four different sizes: supersections of 16 MiB, sections of 1 MiB, large pages of 64 KiB and small pages of 4 KiB. Supersections and large pages are created by using 16 consecutive identical descriptors. The advantage of supersections

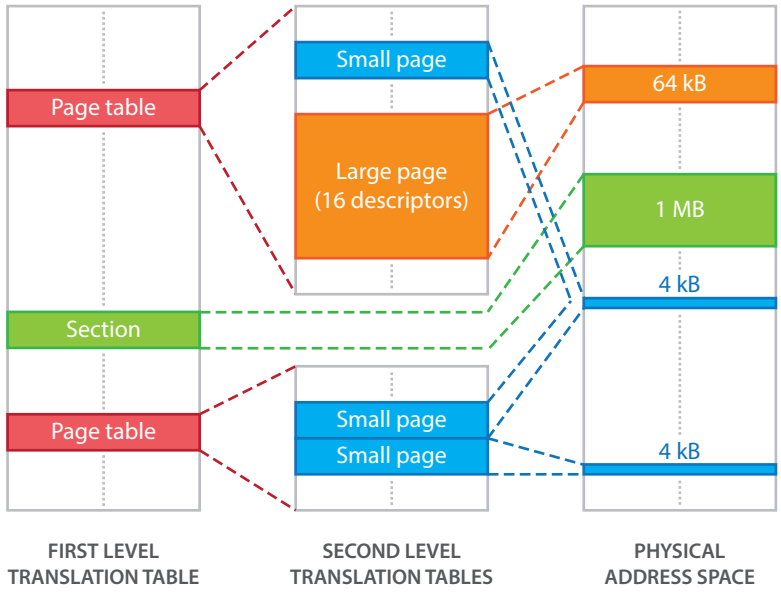

Figure 3.4: Paged virtual memory mapping with the VMSAv7 MMU

and large pages over sections and small pages lies in their translation lookaside buffer (TLB) behaviour: these repeated descriptors only use a single entry in the TLB, while 16 consecutive sections or small pages would result in 16 different entries being used in the TLB. Since TLBs are typically small on ARM-based systems, e.g., 32 entries for the L1 data and L1 instruction TLBs on the Cortex-A8 [10], the decrease in TLB pressure from using supersections and large pages whenever possible is significant. Supersection support is however optional in VMSAv7, and as our hardware did not support it, our hypervisor does not use supersections. Mappings of different sizes may overlap, i.e. several virtual addresses can refer to the same physical address.

At any given time, only two first-level translation tables can be active. They are used together to perform one single address translation. The addresses of both tables are configured in the system control coprocessor, in the registers TTBR0 and TTBR1. The translation table configured through TTBR1 is optional and can be used to override the translation table configured through TTBR0 for the upper part of the virtual address space. Originally, this two-table mechanism was intended to facilitate switching first-level translation tables for user applications: mappings specific to user applications would be made in the lower part of the vir-

tual address range, while the kernel is mapped using its own first-level translation table in the upper part of the virtual address space. During the implementation of our hypervisor, we have never observed a kernel that made use of this mechanism. Therefore, this functionality has not been virtualised and unless explicitly stated otherwise, TTBR1 is omitted from all further discussions on this subject.

Three mechanisms control access to mapped memory regions: access permission bits, an execute-never bit and domains. The access permission bits determine whether the memory is readable or writable to each of the two privilege levels, with the limitation that unprivileged access always implies privileged access. The execute-never bit determines whether or not instructions can be (pre)fetched, and has no effect on data accesses. The rationale behind this bit is to prevent the prefetcher from accessing device memory, because such accesses may have unintended side effects. Domains can be used to override the access permissions and the execute-never bit of entire sets of mappings. Each first-level descriptor is assigned to a domain. In the VMSAv7 MMU, each domain can be configured independently in one of the following three modes:

1. *Disabled*: all accesses fail with a domain trap;
2. *Client mode*: memory accesses are allowed or denied based on the access permissions and the execute-never bit;
3. *Manager mode*: memory accesses are allowed regardless of the access permissions and the execute-never bit.

Although the access permissions and the execute-never bit can be configured at page granularity, domains can only be set in first-level descriptors. Therefore, all pages mapped by a second-level translation table always reside in the domain of their first-level page table descriptor.

3.4.2 Shadow translation tables

Even though the VMSAv7 MMU supports two first-level translation tables to be active simultaneously, it can only use one such table per translation. Combining the hypervisor's translation tables with a guest operating system's translation tables requires either a hardware MMU capable of performing two address translations in a row using two different sets of translation tables, as provided by the ARMv7-A virtualisation extensions, or a software mechanism that merges the tables and then

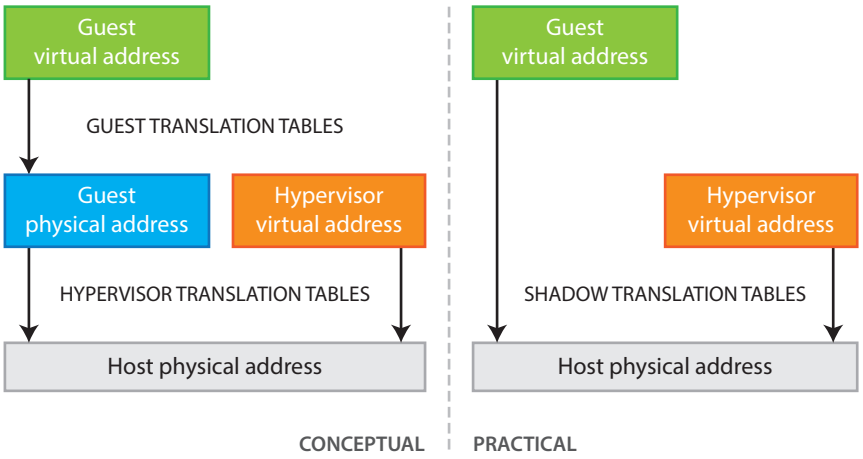

Figure 3.5: Overview of virtual addressing using shadow translation tables

presents the merged tables to the hardware MMU. Such merged tables are called shadow translation tables [2, 21, 113].

In a basic implementation of shadow translation tables, a hypervisor intercepts all guest accesses to the control registers of the MMU, and creates or updates the shadow translation tables whenever the guest creates or updates its own translation tables. The guest should not be able to read hypervisor-created tables, so that it cannot detect that it is being virtualised. The hypervisor can arbitrarily remap guest memory; therefore, addresses viewed as physical by a guest are not necessarily physical addresses to the hypervisor. We refer to virtual addresses mapped by a guest as guest virtual addresses (GVAs). Physical addresses at the virtual machine level, as seen by the guest, are called GPAs. Similarly, physical addresses at the hardware level, as seen by the hypervisor, are called HPAs. These concepts are illustrated on the left-hand side of Figure 3.5.

Ideally, the virtual MMU offered to the guest is an exact replica of the physical MMU as used by the hypervisor. The guest would then have complete control over its own virtual address space. With shadow translation tables, the guest's translation tables are combined with the hypervisor's translation tables as shown on the right-hand side of Figure 3.5. Therefore, part of the virtual address space will be dedicated to the hypervisor, and part may be shared by both hypervisor and guests. To make the distinction between virtual addresses originating from mappings in the guest's original translation tables and virtual addresses corresponding to mappings of the hypervisor, we shall refer to the latter

as hypervisor virtual addresses (HVAs).

3.4.3 Lazy double shadowing

The descriptors in VMSAv7 translation tables define two sets of access permissions: one set for code executing in a privileged mode, and another for code executing in the unprivileged mode. When an operating system is executed natively, this split is sufficient to isolate its kernel from user-space applications. In our virtualised system however, the hypervisor is the only piece of software that runs in a privileged mode; all guest kernel code is executed in the unprivileged mode. Therefore, the access permissions in the shadow translation tables must be used to isolate the hypervisor from the guest; they cannot isolate the guest kernel from its user-space applications without additional software mechanisms. Our hypervisor therefore maintains two shadow translation tables at any given time: one for guest code originally meant to run in a privileged mode, such as an operating system kernel, and another one for guest code meant to run in the unprivileged mode, such as user-space applications.

A virtual machine starts with its virtual MMU disabled. To handle this case, the hypervisor uses only a single first-level translation table, which maps both the hypervisor's and the guest's memory and devices. Since the guest's virtual MMU is disabled, the hypervisor does not need to enforce any kind of privilege isolation in the MMU between any of the guest's privileged modes and the guest's unprivileged mode.

When a guest wants to turn on its virtual MMU, the hypervisor allocates two first-level shadow translation tables: one for the guest's privileged modes, and one for the guest's unprivileged mode. Both tables are initialised with descriptors required by the hypervisor only. No descriptors from the guest's translation tables are copied into either table during initialisation; the shadow translation tables are populated lazily. The hypervisor then activates the shadow translation table for the guest's privileged modes. Eventually, control returns from the memory manager to the DBT engine, which then needs to translate the next block of code from the guest. This will cause a translation fault, since up to this point, the memory manager has not copied any descriptors from the guest's translation tables into the shadow translation tables. The translation fault is handled by the memory manager, which consults the guest's translation tables to find the descriptor for the faulted memory access. Given that the descriptor is valid, the active set of shadow translation tables is updated and control is returned to the DBT engine.

Every access from the guest or the hypervisor to a GVA for which the descriptor has not yet been copied from the guest's translation tables into the active shadow translation tables will cause a translation fault. Similar to the translation of the initial block of code after turning on the MMU, the memory manager copies guest descriptors as needed into the shadow translation tables. Guest descriptors are not copied exactly as they are: the memory manager must take into account the access permissions in order to enforce isolation between the hypervisor and its guests and between the guest's privileged and unprivileged modes. We will therefore refer to this process as *shadow mapping* rather than copying.

A guest can switch from a privileged to the unprivileged mode using an exception return instruction. Vice versa, it can switch from the unprivileged mode to a privileged mode by taking (or causing) an exception or an interrupt. Whenever this mode switch occurs, the memory manager ensures that the set of shadow translation tables is also switched.

The memory manager must keep track of modifications to a guest's translation tables. When a guest modifies a descriptor that has been shadow mapped, the shadow translation tables must be updated accordingly. There are multiple ways in which this problem manifests itself: a guest can edit a descriptor directly, or it can update a system control coprocessor register that modifies the meaning of a set of descriptors. We shall discuss and evaluate multiple approaches to manage the shadow translation tables in Chapter 5.

3.4.4 Hypervisor mappings vs. guest mappings

The hypervisor takes up part of physical memory and of the virtual address space for its own code and data. For practical purposes, the hypervisor lies to Linux guests about the available memory, such that Linux guests will never attempt to use physical memory dedicated to the hypervisor. This is however not sufficient to separate hypervisor memory from guest memory, and it cannot prevent the guest from trying to map or use a virtual address that is also used by the hypervisor.

Whenever the memory manager needs to construct a shadow mapping for a guest descriptor, it will look up the kind of device or memory the guest is trying to map in the virtual platform. The virtual platform contains definitions of all devices and memories usable by the guest, and based on this information the memory manager can prevent the guest from mapping memory dedicated to the hypervisor. When such

an anomaly is detected, the memory manager can either kill the guest, or alter the physical address to allocate a different chunk of free memory to the guest. Since we have never run into this situation, our implementation of the memory manager just kills the guest operating system.

A malicious or ill-constructed guest can still perform direct accesses to HVAs. The only way to block such accesses is to enforce proper memory protection on all hypervisor code and data. Some data structures such as the cache of translated code must however be readable to the guest's privileged modes but not to the guest's unprivileged mode. To avoid situations in which guest accesses to hypervisor data structures go by unnoticed, the amount of data visible to the guest must be minimised.

During the time that a guest has its virtual MMU disabled, it is the hypervisor's responsibility to make sure that the guest's memories and devices are accessible. The shadow translation tables will therefore be initialised with mappings for all guest necessary devices upfront. In order to reduce TLB pressure, all mappings are constructed using the largest descriptors possible, e.g., when the virtual MMU is disabled, guest main memory is mapped in sections of 1 MiB each. Mappings for the hypervisor's code, data and devices are often constructed using small and large pages, in order to enforce strict memory protection rules, and to minimise the potential overlap in the virtual address space between HVAs and GVAs. Once the guest turns on its virtual MMU, it will make its own device mappings as needed. Guests such as Linux also maximise the size of the descriptors for devices. One descriptor can then map any number of devices and even some unused regions in the physical address space. In order to enforce strict memory protection, the memory manager must split such descriptors into smaller descriptors to enforce different access permissions for the different devices mapped by those descriptors. Splitting is an expensive operation, because it requires allocating and filling second-level shadow translation tables, it increases maintenance costs on guest updates, and it increases TLB pressure.

3.5 CPU virtualisation: the DBT engine

Our DBT engine consists of an instruction decoder and encoder, a translator, an interpreter, a software cache for translated code, called the *code cache*, and a second software cache to hold metadata about translated code, called the *metadata cache*. Figure 3.6 depicts the operational cycle of our DBT engine. Translated guest kernel code executes from the code

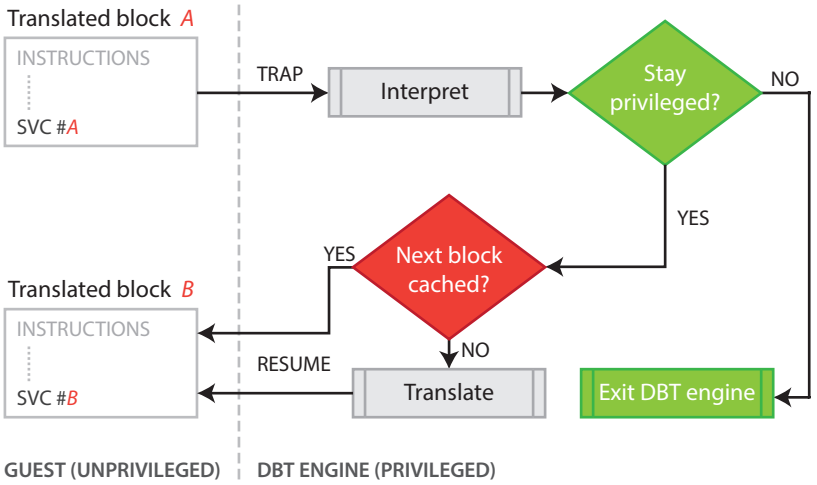

Figure 3.6: The basic operational cycle of our DBT engine

cache in unprivileged mode. Two such translated code blocks in the code cache are shown on the left of Figure 3.6. Note that as explained earlier, guest user applications are not touched by our DBT engine.

In the translated code blocks, control flow and sensitive instructions have been rewritten to trap to the interpreter, where their behaviour is emulated (in privileged mode) on the guest context. The guest context contains a shadow register file, which consists of all general-purpose registers, the program status registers, etc. On ARM, some registers are *banked*, i.e. physically duplicated, for different modes; e.g., almost every mode has its own stack pointer (SP) and link register (LR). Our shadow register file stores all copies of the banked registers separately.

When the interpreter is done, the DBT engine checks whether or not the guest is switching from kernel to user mode. If so, the DBT engine is exited and control is transferred to the guest's user-space code. Otherwise, the address of the next block in the code cache is looked up in the metadata cache. If this look-up fails, the block is first translated to the code cache. The hypervisor then resumes execution in the code cache, either with the retrieved or with the freshly translated block.

We implement traps to the interpreter using the `SVC` instruction, which is normally used by an operating system to implement system calls. Instructions that trap to the interpreter terminate translated code blocks. When the hypervisor receives a system call trap, it first checks whether the trap came from the translated code, or from a guest's user application.

In the latter case, the trap is forwarded to the guest's exception handlers.

System call traps in translated code are routed to the interpreter. However, the DBT engine must first look up the code block's corresponding metadata to determine which instruction was replaced, and which interpreter function must be invoked. To ensure a fast look-up process with $O(1)$ complexity, we have implemented our metadata cache as a fixed-size array, and we use the indexes of the cache items as operands to the `SVC` instructions that terminate our translated code blocks.

After interpretation has completed, the hypervisor checks whether or not the guest has changed its execution mode. When the guest has switched to the unprivileged mode, e.g., to execute a user application, the DBT engine is disabled, and the hypervisor performs a context switch to the guest's user application. When the guest remains executing kernel code in one of the privileged modes, the native PC value in the guest's virtual machine state must be mapped to a code cache address. This requires another look-up in the metadata cache. We implement this look-up using hashing with open addressing; this means that newly translated blocks are assigned an entry in the metadata cache based on their source address. When the look-up fails, the block is first translated.

The translator operates on the instructions of the source block one by one. Instructions are decoded to determine whether or not they can be copied to the code cache as is. Table 3.1 provides an overview of all instructions that require special handling, based on the virtualisability analysis we presented in Section 2.4. Most instructions do not alter control flow, and are not sensitive in any way; they can therefore be copied without modification. For control flow and sensitive instructions, the decoder decides whether they are replaced by traps to the interpreter, or by equivalent sequences of instructions that can be executed without run-time intervention of the DBT engine. For example, instructions that make use of the PC but are otherwise innocuous are translated to make sure they observe the value of the PC as if they were executed natively. After a translation or successful cache look-up, the hypervisor resumes executing the translated guest kernel code within our code cache.

Table 3.1 contains a number of instructions that were not identified as sensitive in our previous analysis: `LDRT`, `STRT` and similar instructions perform memory accesses as if they were executed from the unprivileged mode, regardless of their actual privilege level. They are used by a kernel to access memory with the permissions of an application; this is necessary to efficiently enforce privilege separation and to support

Table 3.1: Control flow and sensitive instructions in the 32-bit ARM instruction set

Instruction	Control flow	Sensitive
ALU* branches (e.g., `MOV pc, lr`)	●	○
ALU* exception returns (e.g., `MOVS pc, lr`)	●	●
`B`, `BL`, `BLX`, `BX`, `BXJ` branches	●	○
`CPS`, `MSR` mode changes	○	●
`CDP`, `LDC`, `MCR`, `MCRR`, `MRC`, `MRRC`, `STC` coprocessor accesses	○	●
`LDM`, `LDR`, `POP` branches	●	○
`LDM`, `RFE` exception returns	●	●
`LDM`, `STM` user mode multiple register restore and save operations	○	●
`LDRT`, `LDRBT`, `LDRSBT`, `LDRHT`, `LDRSHT`, `STRT`, `STRBT`, `STRHT` load and store as unprivileged	○	●
`MRS`, `MSR`, `SRS` accesses to banked registers, the CPSR and the SPSR	○	●
`SVC` system calls	●	●
`WFE`, `WFI` sleep instructions	○	●

* In the 32-bit ARM instruction set, all ALU instructions that write their result into a register can be used as a branch or exception return. They are `ADC`, `ADD`, `AND`, `ASR`, `BIC`, `EOR`, `LSL`, `LSR`, `MOV`, `MVN`, `ORR`, `ROR`, `RRX`, `RSB`, `RSC`, `SBC` and `SUB`.

copy-on-write semantics when copying data to or from user applications within kernel code. Such instructions no longer behave correctly when double shadowing is used, because the permissions for the guest's unprivileged mode are unknown to the physical MMU at the time guest kernel code is being executed, as discussed in Section 3.4.3.

3.5.1 Translation strategies

Our DBT engine translates code to a separate cache. Code can also be translated in-place, such that the virtual address and hence the PC at which code is executed does not change. In-place translation thus has the advantage that PC-sensitive instructions do not require special treatment. It is, however, flawed due to limitations of the ARMv7-A MMU.

In-place translation for the ARM architecture was developed independently at both The University of Manchester [80] and VMware [27, 46, 69]. This translation strategy is also known as patching [113]. It was used in the first version of our hypervisor. With in-place translation, hypercalls are injected directly into the guest kernel's code stream for both sensitive and control flow instructions. Smith and Nair [113] use different hypercalls depending on whether an instruction is a control flow instruction or not, such that each translated block is a basic block. Our approach uses only one single type of hypercalls for both types of instructions: end-of-block hypercalls. Our notion of a translated block therefore does not match the generally accepted definition of a basic block.

In-place translation works for both the 32-bit ARM and the Thumb-2 instruction sets. In 32-bit ARM, each instruction has a fixed width, and the translator can easily replace a sensitive or control flow instruction with a hypercall. In Thumb-2, instructions are either narrow (16-bit) or wide (32-bit); fortunately, all Thumb-2 encodings of the `SVC` instruction are narrow. We replaced wide instructions with the combination of a narrow `NOP` and a hypercall. The original instruction was stored together with a reference to the translated block into a separate metadata cache.

Unfortunately, in-place translation has a number of flaws that have ultimately led us to abandon it. The MMU on the ARMv7-A architecture cannot distinguish between data accesses and instruction fetches while enforcing access permissions on memory mappings. For code to be executable, it must therefore also be readable [12]. The guest kernel can hence observe all modifications to its code made by the hypervisor's DBT engine. This is problematic in a number of scenarios, e.g.:

1. A guest kernel may want to calculate and verify a checksum of itself, but the checksum changes on every alteration of the guest kernel's code by the hypervisor's DBT engine;

2. The guest may relocate code that was previously modified by the hypervisor's DBT engine, causing the hypervisor to lose the connection between the hypercalls in the code and its metadata describing the translated blocks.

The first scenario cannot be detected and always leads to incorrect guest behaviour. With full virtualisation, there is no generic way for a hypervisor to detect incorrect guest behaviour. The hypervisor can only guess that something went wrong if the guest eventually performs an invalid operation or attempts to force a system reset.

The second scenario is detected when the hypervisor first encounters a hypercall in a location for which the hypervisor did not maintain any metadata. This happens only when the guests starts executing copied code from its new location. Since there is no efficient way for the hypervisor to know that a guest has relocated its own code, the hypervisor cannot distinguish between a system call in the guest's code, which also uses the `SVC` instruction, and a hypercall installed by the hypervisor's DBT engine.[2] Even if the hypervisor can identify the instruction as a hypercall, there is no guarantee that the original instruction can be restored. Since our metadata cache is bounded, translated blocks will eventually be evicted, causing the original instruction to be lost.

Lastly, in-place translation does not offer any headroom for binary optimisations, as the translated code cannot be larger than the original code. All problems caused by in-place translation are solved by storing the translated code separately from the original code in a software cache.

3.5.2 Design choices and limitations

Our design of the metadata cache imposes two important constraints on translated blocks: blocks have exactly one entry point and exactly one exit point. The first constraint is a result of our requirement for efficient metadata cache look-ups: by using hashing based on the source address

[2] If the guest follows the ARM Linux embedded application binary interface (EABI), then all system calls are encoded with an immediate operand of zero [138]. If the hypervisor then avoids using zero as immediate operand for hypercalls, it can distinguish between system calls in the guest kernel and hypercalls placed by the DBT engine.

of the first instruction of a translated block, the hypervisor cannot easily look up blocks using the source address of other instructions in the block. This causes some code fragments (i.e. those reachable through branches as well as a fall-through path) to be translated more than once, but requires less metadata to be maintained for each translated block.

Our DBT engine replaces conditional control flow and conditional sensitive instructions with unconditional traps to the interpreter, resulting in the second constraint of having exactly one exit point. This also simplifies the metadata we store for each translated block, because the metadata stores which instruction is replaced and which interpreter function should be invoked to emulate its behaviour. By letting each unconditional trap replace only one instruction, the metadata structure is kept simple and fixed in size for all translated blocks. Replacing conditional instructions by unconditional traps causes unnecessary traps for instructions whose condition would normally not be met; these traps can, however, be eliminated without altering the structure of the metadata as we shall show in Section 4.2.

Unlike existing user space DBT engines, the STAR hypervisor features a bounded code cache with a fixed size of 1 MiB. When it is full and more code needs to be translated, all previously translated blocks are unconditionally flushed from the metadata cache and the code cache. In theory, it is possible to selectively clean cold blocks from the cache. In practice, however, such techniques cause substantial run-time overhead as they complicate software cache management, and they require more metadata to be stored to enable optimisations that create links between translated blocks. We shall present such optimisations in Section 4.2.

3.5.3 Translating PC-sensitive instructions

ARM instructions frequently use the PC as a source operand: as they can only embed small constants of at most 16 bits, larger constants are placed in between code, in so called *literal pools*, which are accessed through instructions that use PC-relative addressing. All such instructions should be translated without trapping to the interpreter if they do not alter control flow and if they are not sensitive. They are:

- the move and shift instructions `MOV`, `MVN`, `ASR`, `LSL`, `LSR`, `ROR` and `RRX`, which copy the PC into another register, optionally shifting or complementing its value;

- the generic computation instructions ADC, ADD, AND, BIC, EOR, ORR, RSB, RSC, SBC and SUB, which can be used to perform calculate addresses based on the PC;
- the test and compare instructions CMN, CMP, TEQ and TST, which perform computations on the PC without storing the result, only updating the condition flags;
- the load and store instructions LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, STR, STRB, STRD and STRH, for PC-relative addressing to access to literal pools; and
- the store instructions STR and STM, as they can store the value of the PC to memory.

The translations of all these instructions other than STM is straightforward; example translations are shown in Table 3.2. The most simple instruction is a MOV that copies the PC into another general-purpose register, as shown in example (1). We translate this instruction into an equivalent pair of MOVW and MOVT instructions. Moore et al. [94] instead embed the PC value as a constant in the code cache and use a PC-relative load to retrieve it, because their techniques target the ARMv5 architecture, which predates the introduction of the MOVW and MOVT instructions.

In general, if an instruction writes to some destination register Rd that is not also used as source register, that Rd can be reused as substitute for the PC, as shown in example (2). When Rd is used as source register, or in the absence of some Rd, an unrelated register must be used instead. We do not perform full liveness analysis on the block to be translated to find such register, as blocks are small and the opportunities for finding available registers are limited. Instead, we always spill and restore an unrelated register before and after the translation of a single instruction. Since we propose multiple techniques for spilling and restoring registers in Section 4.1, the examples in Table 3.2 use the SPILL and REST pseudo-instructions. Translations of conditional instructions, such as in example (3), can only be made wholly conditional if the condition flags remain the same throughout the translation. Otherwise, we skip the translation using a conditional branch with a condition code opposite to the one of the original instruction, as shown in example (4).

Moore et al. [94] fold computations on the PC if all operands are known constants at translation time. For example, an addition of a constant offset to the PC, as shown in Table 3.2, example (2), can be translated

Table 3.2: Example translations of PC-sensitive instructions

(1)	**Instruction:**	`MOV<c> Rd, PC`
	Translation:	`MOVW<c> Rd, #(NativePCValue[15:0])` `MOVT<c> Rd, #(NativePCValue[31:16])`
(2)	**Instruction:**	`ADD<c> Rd, PC, #constant`

	Translation:	`MOVW<c> Rd, #(NativePCValue[15:0])`
		`MOVT<c> Rd, #(NativePCValue[31:16])`
		`ADD<c> Rd, Rd, #constant`

(3)	**Instruction:**	`ADD<c> Rd, PC, Rd`

	Translation:	`SPILL<c> Rs`
		`MOVW<c> Rs, #(NativePCValue[15:0])`
		`MOVT<c> Rs, #(NativePCValue[31:16])`
		`ADD<c> Rd, Rs, Rd`
		`REST<c> Rs`

(4)	**Instruction:**	`ADDS<c> Rd, PC, Rd` (with c $\neq$ AL)

	Translation:	`B<¬c> skip`
		`SPILL Rs`
		`MOVW Rs, #(NativePCValue[15:0])`
		`MOVT Rs, #(NativePCValue[31:16])`
		`ADDS Rd, Rs, Rd`
		`REST Rs`
		`skip:`

SPILL and REST are pseudo-instructions whose implementations will be discussed in Section 4.1.

into a pair of MOVW and MOVT instructions that directly writes the result of the addition to the destination register, without requiring an extra ADD instruction in the translation. Folding has limited applicability as it cannot eliminate updates of the condition flags. It is a micro-optimisation as it at best avoids emitting one ALU instruction. It is in other words not essential and our DBT engine does not yet support it.

Translating STM instructions requires an approach that is very different from the other instructions. Their most common use case is to push registers to the stack, but handcrafted assembly may use them in entirely different ways. We therefore need a generic translation algorithm.

In a limited number of cases, the PC can be substituted with another register. If not, the STM instruction must be split. The simplest way is to split the STM instruction into individual store instructions. However, this may require up to 16 stores, inflating the size of the translated code. We

therefore propose an algorithm that splits the original STM into one STM of all registers other than the PC, and an individual store of the PC.

All STM instructions store registers to consecutive words in memory in the order of their index: R0 first, PC (R15) last. There are four different addressing modes: increment after (IA), increment before (IB), decrement after (DA) and decrement before (DB). For incrementing modes, the address to which the first register is stored is the base address, optionally incremented by one word with the "before" mode. In decrementing modes, the last address is the word before or at ("after") the base address; the first address thus depends on the number of registers being stored. All addressing modes also support "writeback", i.e. updating the value of the base address register at the end of the operation.

Algorithm 3.1 shows how we translate STM instructions. In all cases we spill to obtain a scratch register Rs. If there is some Rs different from the base address register, and indexed lower than the PC but higher than all other registers being stored, then we do not split the STM as Rs can substitute the PC (lines 8 and 19). Otherwise, we split the instruction into a pair of STM and STR instructions; the base address must then be corrected to make up for any differences caused by storing one register less in the translated STM. For descending addressing modes, the base address must first be adjusted (line 13). The addressing mode of the STR instruction is chosen such that no further arithmetic is required to reach the final value of the base address register (lines 21 to 30).

Spilling can be avoided if the STM is split, and one of the registers other than the PC is used as scratch register. The scratch register can then be restored from guest memory. However, such optimisations may be observed by the guest. We therefore omitted them from our algorithm. In case any of the translated instructions cause an MMU fault, our approach introduces extra overhead in handling this fault over splitting the STM into individual stores: depending on the location of the abort, both the base address register and Rs must be restored. When there are few such faults, however, the run-time overhead of our solution is lower compared to using individual stores. In practice, such MMU faults rarely occur in kernel code. This validates our design choice.

3.6 Exception handling

In a virtualised system, exceptions may occur for both the hypervisor and the guest. Our hypervisor receives all exceptions and therefore, it must

Algorithm 3.1: 32-bit ARM `STM` instruction translation

Data: OriginalSTM, OriginalPC
Result: an instruction sequence of which the functional behaviour is equivalent to OriginalSTM, and which can be executed from a virtual address different from OriginalPC

```
C ⟵ get_condition_code(OriginalSTM)
Rn ⟵ get_base_address_register(OriginalSTM)
Registers ⟵ get_registers_to_store(OriginalSTM) \ {15}
Rs ⟵ max(Registers ∪ {Rn}) + 1
HaveSubstitute ⟵ Rs < 15
TranslatedSTM ⟵ OriginalSTM

if HaveSubstitute then
    set_registers_to_store(TranslatedSTM, Registers ∪ {Rs})
else
    set_registers_to_store(TranslatedSTM, Registers)
    Rs ⟵ min({0,1} \ {Rn})
    if not is_incrementing(OriginalSTM) then
        emit(SUB<C> Rn, Rn, #4)
    emit(TranslatedSTM)

emit(SPILL<C> Rs)
emit(MOVW<C> Rs, #OriginalPC[15:0])
emit(MOVT<C> Rs, #OriginalPC[31:16])

if HaveSubstitute then
    emit(TranslatedSTM)
else
    if is_incrementing(OriginalSTM) = is_write_back(OriginalSTM)
    then
        if is_incrementing(OriginalSTM) = is_before(OriginalSTM)
        then
            emit(STR<C> Rs, [Rn, #4]!)
        else
            emit(STR<C> Rs, [Rn], #4)
    else
        Offset ⟵ 4 · | Registers |
        if is_incrementing(OriginalSTM) = is_before(OriginalSTM)
        then
            Offset ⟵ Offset + 4
        emit(STR<C> Rs, [Rn, #Offset])

emit(REST<C> Rs)
```

route each type of exception to its appropriate set of exception handlers. Exceptions for the hypervisor are caused by one of the following:

1. Traps to the interpreter in translated code;
2. MMU faults caused by lazy shadow mapping;
3. Interrupts from devices used by the hypervisor.

An exception that does not match any of the above criteria is routed to the guest kernel's exception handlers.

3.6.1 Guest mode-dependent exception handling

The hypervisor frequently catches exceptions. In Chapter 4, we introduce binary optimisations that eliminate a significant number of the traps to the interpreter from translated code. Even then, a proper implementation of the exception handlers remains critical to limit the execution time overhead incurred by the hypervisor. On each exception that requires hypervisor intervention, the full state of the guest's virtual CPU must be saved to and restored from the guest context.

Storing the guest's virtual CPU state is complicated by ARM's use of banked registers. The hypervisor must keep track of all physical registers, i.e., registers which have been duplicated for different processor modes are stored separately in the guest's virtual processor state. Upon each context switch, the hypervisor must therefore check the guest's virtual processor mode to verify where to set and get the guest's register values. The check in the save operation can be removed by using different register saving routines based on the virtual processor mode of the guest. We have implemented the save operation for each possible mode. This leads to some code duplication, but uses one register less, and therefore fewer stack operations. It also enables us to implement a simple hypercall handler for binary optimisations (see Section 4.1.1). Restoring the guest's virtual CPU state cannot be optimised in the same way, as the guest's virtual processor mode may have changed while handling the exception.

3.6.2 Guest exception handling

Handling exceptions in translated code is a problem typical to system-level virtual machines. Exceptions can occur either synchronously or

asynchronously. Synchronous exceptions are tied to the execution of a particular instruction; e.g., a faulting memory operation, an undefined instruction, or a system call. Asynchronous exceptions occur due to external influences, such as an interrupt from a peripheral device. Guest exceptions outside translated code, i.e. in guest user space, are handled by forwarding the exception to the guest's exception handler. Exceptions in translated code are more complex to handle, as the code cache cannot be exposed to the guest, and the guest state must be consistent upon delivery of an exception.

Synchronous exceptions

System calls and undefined instructions in a guest's kernel can be recognised at translation time, while faulting memory operations must be handled at run time. For exceptions handled at run time, the hypervisor must map the PC value at which the exception occurred to the native PC value of the guest, before delivering the exception to the guest's kernel. Such mapping can be achieved by maintaining a mapping of code cache addresses to native addresses, or through retranslation. Storing the address mapping for every instruction that can fault consumes a lot of memory. In practice, few instructions cause MMU faults. We have therefore chosen to use the retranslation approach. The native addresses are known on the boundaries of a translated block. In order to map an address in the code cache to a native address, we retranslate the block without writing to our software caches, up to the location of the faulting instruction. When exceptions are caught in code that spills and restores registers, the hypervisor must first make sure all register values are restored such that the guest is in a consistent state, before delivering the exception to the guest's kernel, as the guest's exception handler cannot return within the translated block.

Providing the guest with the original PC value of the exception ensures that the guest cannot observe the presence of our hypervisor. More importantly, guest fault handling code may depend on the address at which the exception occurred; one such dependency can be found in the `do_page_fault` function of the Linux kernel.

Asynchronous exceptions

Asynchronous exception delivery is postponed until the end of a translated block. This increases interrupt latency but avoids the cost of map-

ping the guest's current PC value in the code cache to its native PC value. Furthermore, when an asynchronous exception would be delivered to a guest in the middle of a translated block, the guest would return to an address that has already been translated, but as our translated blocks only have a single entry point, the translation would not be usable. We would therefore have to translate a new block, starting from the return address. Since asynchronous exceptions can arrive on every instruction, the number of new translations that would be created during exception handling is virtually unbounded. Breaking blocks for asynchronous exceptions would therefore nullify all benefits of caching translated code. By postponing the delivery until the end of a block is reached, this is avoided completely.

Chapter 4

Evaluation of dynamic binary translation techniques

Using DBT for full system virtualisation comes with a few challenges not present in user-space DBT engines. In user-space, DBT engines often take shortcuts which are only valid for well-behaving applications, and there is no strict requirement to isolate the DBT engine from the application. In full system virtualisation, DBT is used together with MMU virtualisation to isolate guests from the hypervisor and from one another. Furthermore, kernel code often contains handwritten assembly and special system instructions, requiring special care in the DBT engine.

In this chapter, we study how to address the challenges specific to using DBT for full system virtualisation. We start with the problem of spilling and restoring registers in translated code. We then study how we can optimise the performance of translated code, by eliminating traps to the interpreter, and we evaluate the impact of our techniques using both micro-benchmarks and real applications.

The work presented in this chapter has been submitted to the *Journal of Systems Architecture*.

4.1 Spilling and restoring registers

Translating instructions that use the PC as source operand requires finding a substitute register to hold the native value of the PC. As outlined in Section 3.5.3, this may involve spilling and restoring a register. On the one hand, spilling and restoring should ideally avoid a full context switch from the guest to the hypervisor, as that would defeat the purpose of translation—we try to avoid the cost of interpreting the instruction, which mainly consists of the cost of the context switch. On the other hand, all cached translations must be protected from guest modification for security and reliability reasons; therefore, we cannot spill register values to the code cache without a trap to enter a privileged mode.

We propose two generic solutions to the spilling problem. In our first solution, we use a lightweight trap, which avoids a full context switch, but at the same time enables translated code to write to otherwise protected locations in memory. Our second solution consists of using user-mode accessible coprocessor registers as temporary spill locations. When a guest is known to exhibit a certain behaviour, other techniques may be employed such as using the guest's stack. Such non-generic solutions are, however, out of the scope of our research.

4.1.1 Lightweight traps

Our hypervisor uses the system call mechanism (`SVC` instruction) for traps to the DBT engine. We implement lightweight traps using an undefined instruction trap, to avoid adding complexity to the system call handler. The sole objective of lightweight traps is to switch to a privileged mode: when this mode switch is performed in the middle of a translated code block, which is normally executed in the unprivileged mode, the subsequent instructions in the block will execute in the privileged UDF mode until a later instruction in the block changes the mode to unprivileged again. While executing in the privileged mode, the translated code can access the otherwise protected locations for spilling.

The undefined instruction trap and hence the mode switch are initiated by an undefined instruction, inserted by the DBT engine in the translated code block. Guests cannot exploit this mechanism to enter a privileged mode, because (1) they cannot alter the contents of the software cache, and (2) because in case that same undefined instruction is encountered in the guest's instruction stream, the hypervisor will handle it as a sensitive instruction, which is translated into a trap to the DBT en-

Listing 4.1: Lightweight trap handler

```
1 MRS     sp_und, spsr
2 AND     sp_und, sp_und, #PSR_MODE
3 TEQ     sp_und, #PSR_MODE_USR
4 LDREQ   sp_und, [lr_und, #-4]
5 MVNSEQ  sp_und, sp_und
6 BXEQ    lr_und
```

gine before it can be executed. Kernels normally do not cause undefined instruction traps. Our lightweight trap mechanism therefore does not introduce any additional overhead in the kernel's normal operation.

The key aspect of this approach is the design of an undefined instruction trap handler that can (1) classify caught traps, (2) return control to the translated code in some privileged mode when the trap is classified as the hypervisor's lightweight trap, (3) invoke a generic undefined instruction trap handler otherwise, and (4) does so in as little as possible time to minimise the performance overhead.

We have managed to come up with a mechanism and a handler that requires only eight instructions to enter the privileged UDF mode from within the code cache. This mechanism relies on a specific instruction encoding that is permanently defined to generate an undefined instruction trap by the ARM architecture [13], encoded as `0xFFFFFFFF`. This encoding is used in the code cache to perform the lightweight trap. The trap is caught by the hypervisor's exception vector, which contains branches to handlers for all kinds of exceptions. We use different exception vectors based on the virtual mode of the guest, so that undefined instruction traps from guest user space cannot reach our lightweight trap handler. Our exception vectors for the guest's privileged modes contain a simple branch to the lightweight trap handler shown in Listing 4.1.

In lines 1 to 3 of our handler, we check whether the undefined instruction trap was taken while executing in the unprivileged mode. If so, we are sure that the trap was caused by translated guest kernel code executed from the code cache. If not, lines 4 to 6 are not executed. In line 4, we load the instruction that caused the trap. In line 5, the encoding of the instruction is complemented, and the condition flags are updated; they are used in line 6 to jump back to the code cache if the bitwise complement of the instruction equals zero, which only happens if the trap was caused by our lightweight trap instruction with encod-

ing `0xFFFFFFFF`. When the branch in line 6 is not executed, control falls through to a generic undefined instruction trap handler.

Our lightweight trap handler uses only banked registers to avoid saving guest registers. After jumping back to the code cache, both SP and LR are usable, such that spilling to memory can sometimes be avoided. For example, translated ALU instructions that do not update the condition flags, and that make use of neither SP nor LR, can be executed safely while in the privileged UDF mode; the banked SP or LR can then be used as a substitute register to hold the native value of the PC. To resume execution in the unprivileged mode, the DBT engine inlines a PC-relative exception return instruction in the translated code.

We have implemented two distinct usage scenarios of our lightweight traps for spilling. In the first scenario, a private spill location is appended as needed at the end of a translated block inside the code cache. As is traditional on many ARM systems, our hardware platform has separate L1 instruction and data caches, and a unified L2 cache. In order to simplify cache maintenance requirements when translating new blocks, the code cache is mapped write-through at the L1 cache level, and write-back write-allocate at the L2 cache level. Therefore, spilling to and restoring from the code cache may cause many L1 cache misses. In a second scenario we spill to a shared spill location outside the code cache, which is mapped write-back and write-allocate at both L1 and L2 cache levels. This trades L1 cache hits for increased TLB pressure; since we aim to avoid guest-readable data structures outside the code cache, reading from the shared spill location may require a second lightweight trap if the translated instruction cannot be executed from a privileged mode. Such a second trap is never required in the first scenario.

4.1.2 User-mode accessible coprocessor registers

All modern ARM processors up to ARMv7-A support coprocessors. Coprocessors are devices whose registers can only be accessed using specific instructions. Such accesses typically exhibit lower latencies than accesses to memory-mapped device registers. Each ARMv7-A processor comes with at least two standardised coprocessors: the debug coprocessor and the system control coprocessor. The latter is used for CPU identification, MMU configuration, cache and TLB maintenance, etc.

The system control coprocessor contains a few registers that are accessible from user mode. We therefore researched the feasibility of

using such registers as spill location. Suitable registers do not affect the behaviour of the hypervisor or the underlying hardware, are both readable and writable from user mode, and can hold a 32-bit word. At least three coprocessor registers in the ARMv7-A architecture match our criteria: we can either use one of the performance counter registers (PMCCNTR and PMXEVCNTR), or the user-writable software thread ID register (TPIDRURW) [13]. The generic timer extension, when available, also provides suitable registers, but using those registers for spilling renders the generic timers unusable for the hypervisor.

Using hardware coprocessor registers as spill location does not interfere with a guest kernel's usage of coprocessors, as guest kernels normally do not interact with hardware coprocessor registers directly; our DBT engine ensures that the guest can only access a virtual copy of the coprocessor registers. Since our hypervisor runs user applications unmodified, all user-mode readable registers must be restored upon a guest context switch from kernel to user mode. All user-mode writable registers must additionally be saved upon a guest context switch from user mode to kernel. When such registers are not used for spilling, we can let the kernel update them directly to eliminate the cost of saving and restoring those registers on every guest context switch.

Using performance counter registers for spilling comes with the disadvantage that the hypervisor can no longer make full use of the performance counters for itself. Performance counter registers are made accessible to the unprivileged mode through configuration of the system control coprocessor. Spilling to the performance counter registers therefore also requires maintenance on every context switch, to disable and re-enable access accordingly. Unlike the thread ID register, the maintenance for using the performance counters does not require extra memory accesses during guest context switching.

4.2 Tackling DBT-related overhead

The naive version of our hypervisor translates all control flow and sensitive instructions shown in Table 3.1 into traps to the interpreter. This gives the hypervisor maximum visibility into the guest kernel's execution, which is useful for tracing, but far too slow for any other practical purpose. We have therefore studied which traps occur most frequently, and we propose optimisations to eliminate them.

All DBT-related overhead observed in user applications originates

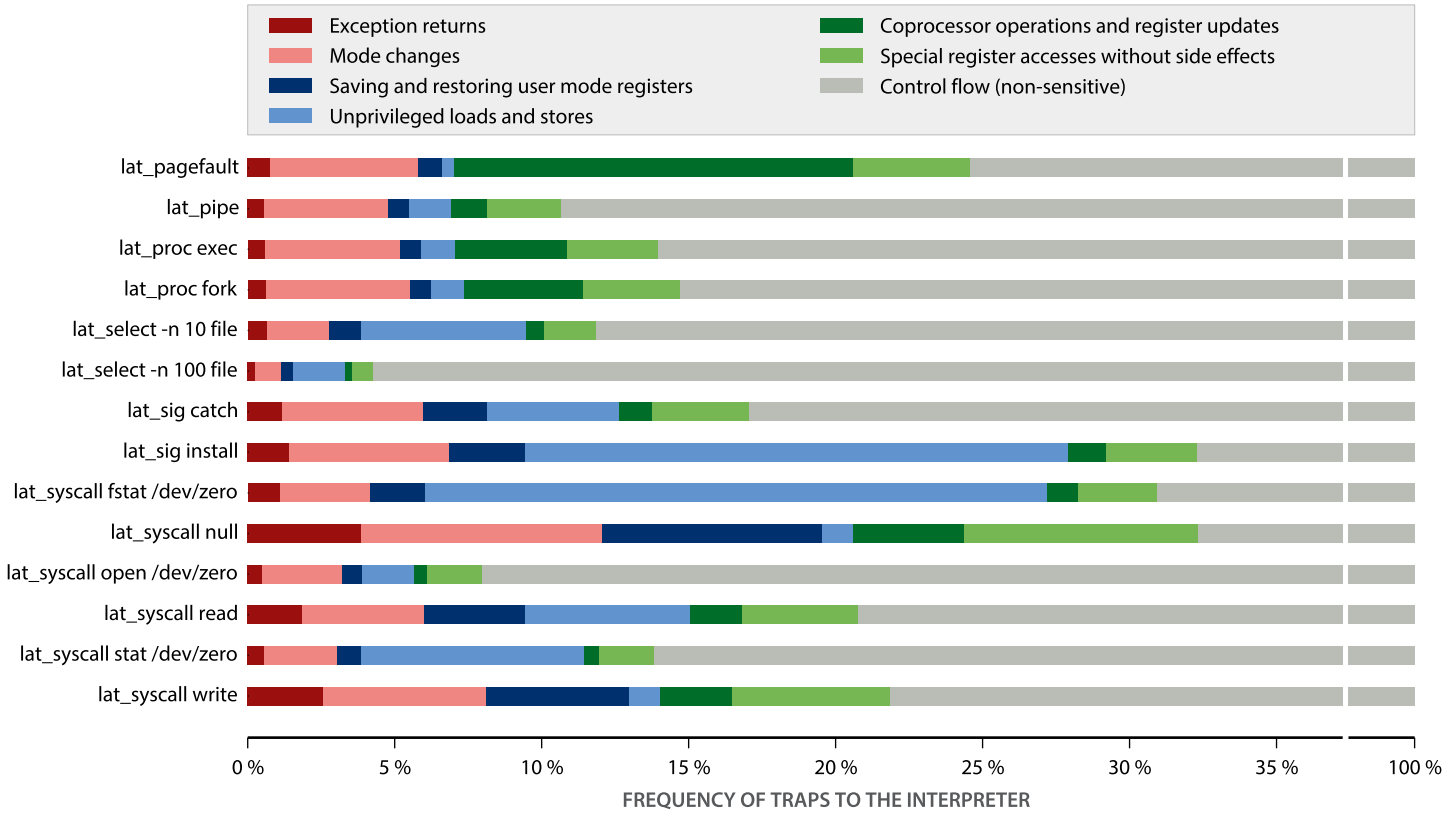

Figure 4.1: Frequency of traps to the interpreter in the naive version of our hypervisor by instruction class

from interactions with the kernel and interrupt handling, because our hypervisor only translates kernel code. For a Linux guest, this means that we can analyse the origins of DBT-related overhead by running micro-benchmarks for system calls and signal handling.

To do so, we have executed selected benchmarks from the *lmbench* version 3 suite [88] on a BeagleBoard. As guest OS, we have used a vanilla Linux 2.6.28.1 kernel. This version of the kernel was used throughout the development of the hypervisor. Using newer versions of the Linux kernel does not impose new challenges to the DBT engine, but requires more engineering work in device emulation. Our test kernel contains two upstream patches normally not present in the 2.6.28.1 release to enable building with GCC 4.9.1 and recent versions of GNU make. The benchmarks we use measure the virtualisation cost on frequently-used kernel functionality:[1]

- `lat_pagefault`: measures the cost of faulting on pages from a file.
- `lat_pipe`: uses two processes communicating through a UNIX pipe to measure inter-process communication latencies. The benchmark passes a token back and forth between the two processes which perform no other work.
- `lat_proc`: creates processes in different forms, to measure the time it takes to create a new basic thread of control.
  - `fork`: measures the time it takes to split a process into two nearly identical copies and have one of them exit.
  - `exec`: measures the time it takes to create a new process and have that new process run a new program, similar to the inner loop of a command interpreter.
- `lat_select`: measures the time it takes to perform a `select` system call on a given number of file descriptors.
- `lat_sig`: measures the time it takes to install and catch signals.
- `lat_syscall`: measures the time it takes for a simple entry into the operating system kernel.
  - `fstat`: measures the time it takes to `fstat()` an open file whose inode is already cached.

[1] Benchmark descriptions copied and adapted from the lmbench manual pages.

- `null`: measures the time it takes to perform a `getppid` system call. This call involves only a minimal and bounded amount of work inside the kernel, and therefore serves as a good test to measure the round-trip time to and from the kernel back to user space.
- `open`: measures the time it takes to open and then close a file.
- `read`: measures the time it takes to read one byte from `/dev/zero`.
- `stat`: measures the time it takes to `stat()` a file whose inode is already cached.
- `write`: measures the time it takes to write one byte to `/dev/null`.

We have executed each benchmark once and measured the source and frequency of traps to the interpreter. Internally, we have forced each benchmark to execute its test 100 times in a loop, such that we can easily distinguish the traps specific to the benchmark from the traps caused by the creation and destruction of the benchmark process. Figure 4.1 shows an overview of the source and frequency of traps for each individual benchmark. As is typical in DBT, most of the overhead is caused by control flow. Control flow optimisations have already been studied extensively, so we have implemented some of the existing optimisations. For the traps not related to control flow, we propose new translations specific to full virtualisation on ARMv7-A.

4.2.1 Control flow

Our measurements indicate that over all benchmarks, 68% to 96% of our overhead is caused by non-sensitive control flow instructions. As shown in Figure 4.2, most control flow instructions are direct branches. Traps caused by direct branches can be eliminated through lazy optimisation: after a trap caused by a branch, the block targeted by that branch is translated, and the trap is replaced by a branch within the code cache to the newly translated block. We say that the blocks are *linked* together.

Conditional direct branches are initially translated into two traps: a conditional trap with a condition code opposite to the original instruction, which models the original fall through path, and a non-conditional trap. Both traps can be replaced lazily by a direct branch within the code cache. In general, an extra conditional trap is added to every translated conditional instruction, which can be linked like a direct branch.

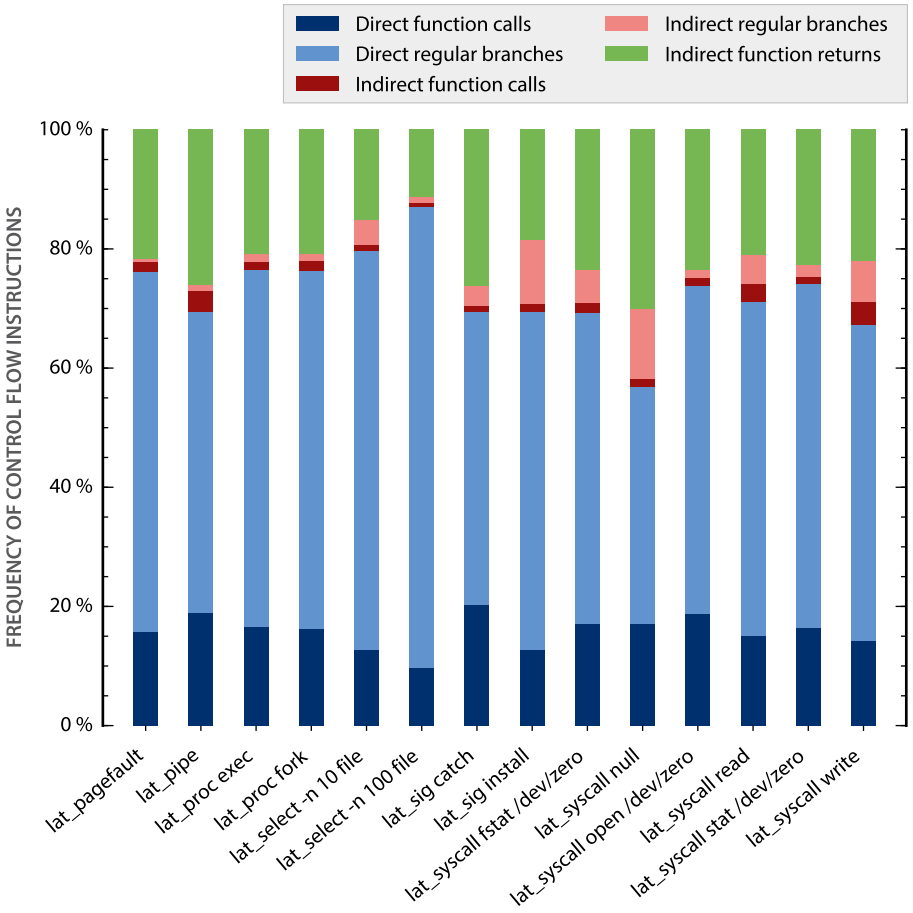

Figure 4.2: Frequency of traps caused by control flow instructions

Indirect branches

Blocks with indirect branches cannot be linked at translation time, as the destination addresses of the branches are not known until they are executed, and their destination addresses may vary between executions. Several solutions to eliminate traps caused by indirect branches already exist. Moore et al. [94] use an *indirect branch target cache*, a global hash table of code cache addresses indexed by native PC values [65], which is used by the translated code to look up translations of branch targets. A similar technique, called a *sieve*, implements the hash table in instructions rather than in data [115]. A limited set of code cache addresses known to be frequently targeted by specific indirect branches can also be inlined

in the translation of that indirect branch [60, 65]. Most techniques treat function returns separately, as the target of a function return can be predicted easily by keeping track of function calls.

Function calls can be recognised in branch instructions that automatically write the return address to LR (`BL` and `BLX`). Function returns are harder to identify, as the ARM architecture does not have a dedicated function return instruction, such as `RET` on x86. We therefore identify function returns based on the patterns recognised by ARM's hardware return address stack; they are:

- an indirect branch to the value in the LR (`BX lr`);
- a copy of the LR into the PC (`MOV pc, lr`);
- a load of the PC from the stack using any of the `LDR`, `LDMIA` and `LDMIB` instructions[2].

To avoid the traps caused by function returns, our hypervisor implements a software return address stack, also known as a shadow stack, similar to Pin [60, 77]. The patterns we use to identify function returns are not fully accurate, as on average they identify 4% more function returns as there are function calls. We suspect these false positives are partly caused by handwritten assembly in the kernel. Our shadow stack is nevertheless capable of significantly reducing DBT-based overhead, as we will show in our evaluation in Section 4.3.

Unlike Pin and other user-space DBT engines, our shadow stack implementation does not make use of the guest's stack for saving and restoring registers during manipulations of the shadow stack; instead, we use the register spilling techniques that we proposed in Section 4.1.

As the amount of indirect branches other than function returns is fairly low, our hypervisor currently does not implement a generic indirect branch optimisation algorithm.

Effects on asynchronous exception delivery

Interrupts are always enabled during guest execution, regardless of whether or not the guest has enabled any interrupts, because the hypervisor needs its own interrupts to function properly, and because it

[2] ABI-compliant code uses descending stacks which are always accessed using the incrementing addressing modes of the `LDM` instruction.

must be capable of rescheduling guest interrupts as discussed in Section 3.6.2: when the guest is executing in kernel space, we postpone the delivery of all asynchronous exceptions such as interrupts until the end of a translated block. This is straightforward as long as the blocks end in a trap to the interpreter. When blocks are linked together, however, they can create loops in the code cache that do not contain a single trap. The execution of translated code can then continue in the code cache for quite some time without any trap being executed. The hypervisor cannot wait until all such loops have finished executing.

To overcome this problem, the DBT engine partially undoes block linking whenever the hypervisor has to deliver an exception to the guest, by replacing any direct branches at the end of the active translated block by traps to the interpreter. Alternatively, if the block makes use of our shadow stack to perform a function return within the code cache, we corrupt the top entry of the shadow stack to ensure that the next look-up causes a trap. The hypervisor then briefly resumes the guest so that it can finish executing the translated block. The end of that block is now certain to trap to the DBT engine, which gives the hypervisor the opportunity to deliver the exception while the guest is in a clean state.

4.2.2 Exception returns and other mode changes

Exception returns affect numerous data structures in the hypervisor, and involve hardware reconfiguration; it is therefore hard to avoid a full context switch from guest to hypervisor and back. As shown in Figure 4.1, exception returns can cause up to 4% of all traps in an optimised DBT engine. Mode changes using `CPS` and `MSR` instructions are very similar to exception returns, but do not influence control flow. Figure 4.1 shows that close to 10% of the traps can be caused by such mode changes.

Some of the mode change instructions merely enable and disable interrupts. Those instructions can be translated to equivalent instruction sequences that update the program status register of the guest in its shadow register file without requiring a trap to the interpreter.

4.2.3 Saving and restoring user mode registers

When an operating system performs a context switch from an application running in user mode to its kernel or vice versa, it must save and restore all registers of that application. The ARM architecture provides special

load and store multiple instructions (LDM and STM) that access the user mode SP and LR rather than the registers of the active mode, such that a kernel does not need to switch modes to access the user mode registers. These instructions are otherwise functionally equivalent to load and store multiple instructions that only access registers from the active mode.

Figure 4.1 shows that up to 8% of all traps in our DBT engine originate from instructions to save and restore user mode registers. In order to avoid these traps, we split the instructions into a first part that only involves the non-banked registers, and a second part that involves the banked registers. Only the second part requires accesses to the shadow register file to be inlined in the translation. When the instruction loads non-banked registers, the load of the banked registers is reordered to happen before the load of the non-banked registers, as all affected non-banked registers can then be used as scratch registers.

Listing 4.2 shows how we translate `LDMDB sp,{r0-r12,sp,lr}^`, an instruction taken from the Linux kernel's system call return routine. This instruction restores all registers other than PC from the kernel's stack. Instead of updating the current SP and LR, however, it writes to their user mode counterparts. We use a lightweight trap to access the guest's shadow register file from the privileged UND mode (line 2). Because the base address resides in the banked SP register, we copy it into a scratch register before switching modes (line 1). We make use of the banked registers in the UND mode to hold the address of the guest's user mode registers in its shadow register file (lines 3–4), and to load values from the guest's stack using LDRT instructions (lines 5 and 7). Using LDRT makes the MMU authorise the loads using the guest's permissions, which is necessary to enforce guest isolation. We then store the loaded values into the guest's shadow registers (lines 6 and 8). LDRT instructions always operate on the address held in the base address register, and update its value with a specified constant after the load operation. We must therefore adjust the base address register upfront (line 1) and in between the loads from the shadow register file (line 5). Afterwards, we return to the unprivileged mode (line 9), and we load the non-banked registers (line 11). As the original LDMDB instruction uses a decrementing mode, its base address must be adjusted to compensate for the removal of the banked registers; to avoid restoring the base address register, we write the adjusted base address into the scratch register (line 10).

Store multiple instructions that save user mode registers are treated similarly. Algorithm 3.1 shows how to compensate for the modified base

Listing 4.2: Translation of `LDMDB sp, {r0-r12,sp,lr}^`

```
1  SUB     r0, sp_usr, #8
2  LWTRAP
3  MOVW    sp_und, #(AddressOfUSRRegs[15:0])
4  MOVT    sp_und, #(AddressOfUSRRegs[31:16])
5  LDRT    lr_und, r0, #4
6  STR     lr_und, [sp_und]
7  LDRT    lr_und, r0
8  STR     lr_und, [sp_und, #4]
9  SUBS    pc, pc, #4
10 SUB     r0, r0, #4
11 LDMDB   r0, {r0-r12}
```

address caused by altering the register list for all addressing modes.

4.2.4 Unprivileged loads and stores

Instructions that perform load and store operations as if they were executed from the unprivileged mode are used by the Linux kernel to access data from user applications. When such instructions are executed from the unprivileged mode, they behave as normal loads and stores. However, as explained in Section 3.5, this would cause the loads and stores to be executed with the permissions of the guest's kernel rather than those of its user applications due to double shadowing.

Emulating the behaviour of unprivileged loads and stores requires authorising the operation based on the permissions assigned to user applications, either by using a software-based look-up in the guest's translation tables, or by switching the active set of shadow translation tables to the unprivileged set, and then performing the check in hardware. Trapping to the interpreter on every unprivileged load and store is costly: in the signal installation and `fstat` system call benchmarks shown in Figure 4.1, they account for about 20% of all traps.

Our initial approach to eliminate the trap consisted of inlining a switch to the unprivileged set of shadow translation tables before and after every unprivileged load and store instruction. Switching translation tables must be done from a privileged mode and therefore requires a lightweight trap. The code cache should not be accessible to the guest's user applications, in order to prevent exposing the translated kernel. The translated instruction must therefore be executed from a privileged

mode. As the instruction performs memory accesses as if it was executed in the unprivileged mode, these accesses are always authorised with the permissions of the guest. There is a problem, however, when the instruction uses one or more banked registers: the value of the user mode registers must be fetched into the registers of the privileged UDF mode, and any updated registers must be restored afterwards. On the basic ARMv7-A architecture, the user mode registers can be retrieved by using `LDM` and `STM` instructions, or by switching to the `SYS` mode using `CPS` instructions, as the privileged `SYS` mode uses the user mode register bank. While using `CPS` instructions leads to longer translated instruction sequences, we found it to perform slightly faster than using `LDM` and `STM` instructions. The virtualisation extensions offer a more elegant solution: they add special `MRS` and `MRS` instructions that copy values between banked and non-banked registers.

As noted by the authors of the ITRI hypervisor, our initial approach can be improved by eliminating unnecessary translation table switches in between consecutive unprivileged load and store instructions, as they often appear grouped together in the Linux kernel [111]. Notable examples of functions that contain such groups are `__copy_from_user` and `__copy_to_user`. Other functions such as `restore_sigframe`, which is used when returning from a signal handler, contain sequences that alternately perform an unprivileged load and a normal store; such instruction sequences cannot be optimised any further.

4.2.5 Coprocessor operations and register updates

A small fraction of the overhead in our benchmarks results from invoking operations on coprocessors and updating the registers of the system control coprocessor. As shown in Figure 4.1, this fraction is mostly limited to 5%, except in the page fault benchmark where it reaches 15%, because page fault handling involves several accesses to the configuration registers of the MMU. While the ARMv7-A architecture offers several instructions to access coprocessors (see Table 3.1), only `MCR` and `MRC` are functional for the system control coprocessor. All other attempts to access the system control processor will cause an undefined instruction trap. `MRC` instructions read from the coprocessor registers without causing any side effects; they are dealt with in the next section.

The `MCR` instruction is used to invoke an operation such as cleaning a cache line, and to update register values such as setting the address of the active translation table. Because the coprocessor register num-

ber is encoded as a constant within the MCR instruction, it is known at translation time. We can hence selectively inline equivalent instruction sequences specific to a coprocessor operation or register.

Coprocessor instructions that enable or disable caches and the MMU, or that reconfigure the MMU, always trap to the hypervisor. Some cache and TLB maintenance operations can however be inlined, depending on whether or not they are used by the hypervisor to keep track of a guest's modifications to its code and translation tables. Our hypervisor supports multiple such tracking mechanisms; some of which obsolete guest TLB maintenance, in which case a guest's TLB maintenance operations are removed by the translator. Some operations cannot be executed as is: e.g., the guest should not be allowed to invalidate the data caches without writing back dirty lines to memory, as this may discard hypervisor state.

Some cache maintenance operations, such as instruction cache maintenance, can be executed without modification; they are therefore inlined using lightweight traps. Other operations may require modifications to prevent the guest from discarding the hypervisor's data from dirty caches, or due to incompatibilities between the virtual platform and the physical hardware platform. The difficulties and technicalities of virtualising cache maintenance operations are further discussed in Chapter 5.

Some coprocessor operations can be executed regardless of the privilege level; this is the case for the deprecated coprocessor barrier instructions. We translate them to the newer equivalent instructions, without inserting any kind of trap.

4.2.6 Special register accesses without side effects

The remainder of the traps to the interpreter consists of accesses to special registers, such as coprocessor registers reads using the MRC instruction, banked register reads using the MRS instruction, and writes to the SPSR using the MSR instruction. These accesses account for 1% to 9% of the traps to the interpreter. They do not cause any side effects beyond a simple register copy. We translate all such instructions into an equivalent sequence that accesses the guest context using a lightweight trap.

4.2.7 Summary

We explained how our hypervisor handles control flow by linking blocks together in the code cache, and how block linking affects exception han-

dling. We showed that while a majority of the overhead is caused by control flow, sensitive instruction traps are not negligible. We therefore proposed optimisations to translate sensitive instructions to equivalent instruction sequences rather than to a trap for several ARM-specific instructions such as saving and restoring user mode registers, unprivileged loads and stores, and coprocessor accesses. In the next section, we evaluate how these optimisations reduce the DBT overhead.

4.3 Evaluation

As already mentioned, guest user-space code runs without intervention of the hypervisor, so in our evaluation we again focus on events related to mode switches and kernel operations, and we evaluate all our techniques using the same hardware and benchmarks as in Section 4.2.

We first configure the hypervisor for the different register spill techniques that we proposed in Section 4.1, to determine which technique performs best. To achieve a fair comparison between the different techniques, we enable only those optimisations that are not tied to a particular register spilling technique. We then proceed by demonstrating how our optimisations reduce the DBT-based overhead of the hypervisor, while using the best performing register spilling technique.

All lmbench benchmarks perform their own timing measurements. In order to ensure that these measurements are accurate, we grant the guest full and unsupervised access to a minimal set of hardware timers.

It is important to note that the used micro-benchmarks are stress tests that execute high-overhead events at a much higher rate than standard applications. The reported overhead numbers are therefore by no means representative of the performance of real-world applications; they only serve to identify the best optimisations for our DBT engine.

4.3.1 Register spilling techniques

We compare the results of running the benchmarks using our register spilling techniques to a set of results obtained with a fully writable code cache. To reduce the impact of external influences such as interrupts, we have executed each benchmark 100 times and we report averages.

Figure 4.3 shows the extra overhead incurred by our spilling techniques on each benchmark, when spilling and restoring at most one

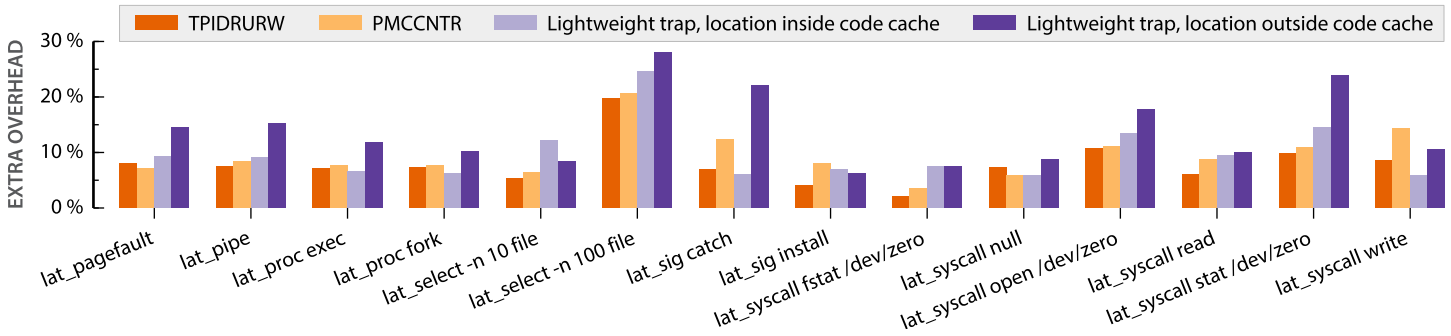

Figure 4.3: Overhead of register spilling techniques over a fully writable translation store, using at most one register

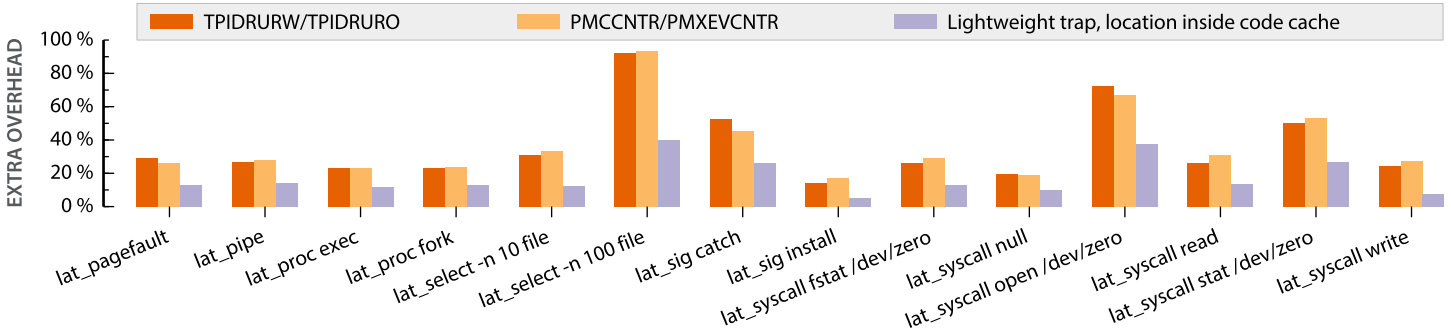

Figure 4.4: Overhead of register spilling techniques over a fully writable translation store, using at most two registers

register. Spilling using lightweight traps to a location outside the code cache consistently performs worse, except in the `fstat` and signal installation benchmarks, where the difference with spilling inside the code cache is negligible. The extra overhead when spilling to an external location is caused by the extra instructions required to construct the address of the spill location into a register, and by increased TLB pressure.

On almost all benchmarks, spilling to the performance counter register PMCCNTR incurs slightly more overhead than using the user-writable software thread ID register (TPIDRURW). We expected the performance counter registers to have higher access latencies than the thread ID register, as only the latter is intended to be updated frequently. As it turns out, the subtle differences between both measurements are only caused by the differences in maintenance requirements upon a guest context switch; the registers are otherwise equivalent.

On several benchmarks, spilling inside the code cache performs only slightly worse than spilling to the thread ID register. This implies that the latency to access the thread ID register is of the same order of magnitude as the time taken to spill and restore using a lightweight trap. With lightweight traps, however, the majority of the overhead is caused by the trap to enter a privileged mode, rather than by the memory accesses that perform the actual spill and restore operations. Therefore, when spilling multiple registers at once, or when the spill location is accessed more frequently, we expect coprocessor latency to dominate.

To investigate the impact of coprocessor latency, we have performed a second series of tests in which we have enabled the shadow stack optimisation. This optimisation requires spilling two registers, and accesses the spill locations several times. We do not provide measurements for spilling to a shared location using lightweight traps, since it is clear from our first set of measurements that this technique causes too much overhead. In order to support spilling two registers, we have adapted the remaining techniques as follows:

- When spilling to a private location inside the code cache, we resize the spill location to fit two words.
- When spilling to TPIDRURW, we use the user-read-only thread ID register (TPIDRURO) as secondary spill location. We do not require a second user-mode writable register as our shadow stack implementation only updates the secondary spill location from a privileged mode.

- When spilling to the performance counter registers, we use PMCCNTR as primary and PMXEVCNTR as secondary spill location.

As illustrated in Figure 4.4, coprocessor spilling incurs more than twice the overhead of spilling inside the code cache, when applied to our shadow stack. This overhead is caused by the access latency of the coprocessor registers. Our results indicate that using a coprocessor register to spill can, however, be beneficial in translations that only need to spill one register, and with few accesses to the spill location. In such scenarios, using coprocessor operations can be slightly faster and results in smaller translated code size. In other scenarios, spilling inside the code cache using lightweight traps should be preferred.

4.3.2 Optimisations to avoid traps to the DBT engine

We analyse the performance improvements gained from the optimisation techniques described in Section 4.2 on the same set of benchmarks from the lmbench suite. We have run each benchmark on different configurations of the hypervisor; we started with a configuration in which only direct branch linking was enabled, and we then enabled our optimisations one by one, in the order they were discussed in Section 4.2. We have normalised the results based on a set of measurements obtained on a native system, running the same Linux kernel on the same hardware. All configurations unconditionally make use of lightweight traps to spill registers to a spill location inside the code cache.

Figure 4.5 provides an overview of our results. The unoptimised version of our hypervisor, visualised by the leftmost bar, causes slowdowns ranging from a factor 22 to a factor 114 compared to executing the benchmarks natively. We observe the lowest slowdown on larger benchmarks such as page fault latency (`lat_pagefault`), process creation (`lat_proc`) and `lat_select` with 100 file descriptors. In the benchmarks for pipe latency and simple system calls the overhead of guest context switches dominates and causes bigger slowdowns. The optimisations proposed in this paper reduce the slowdown on largest benchmarks from a factor 22 to a factor 7 over native. On the smaller benchmarks, we achieve an overhead reduction from a factor 114 to a factor 20 over native.

When we investigated the major contributions to overhead reduction, we observed that enabling our shadow stack causes a significant speedup in all benchmarks. Mode change optimisations have little impact, as they only affect instructions that toggle interrupt masking; the majority of

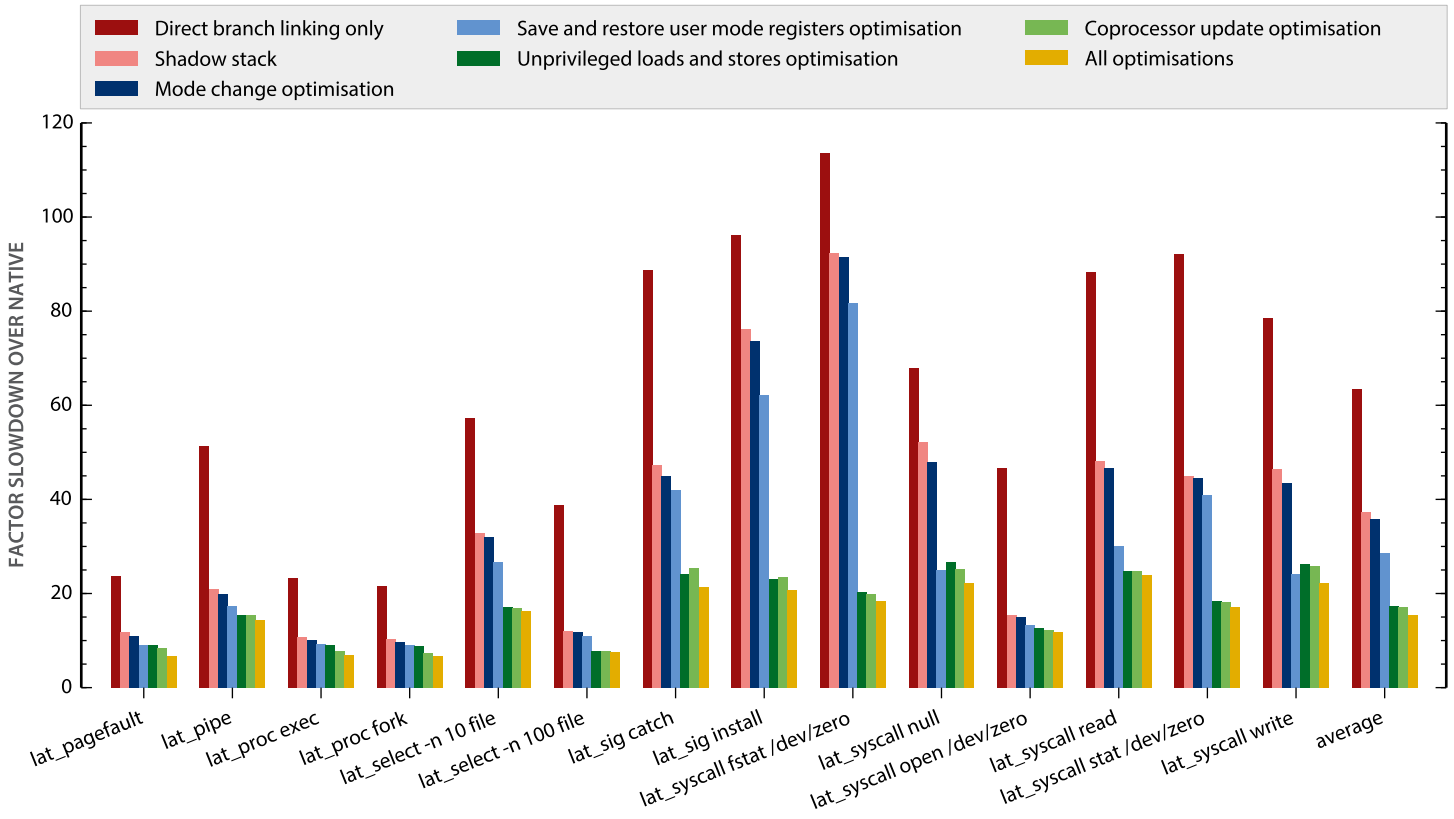

Figure 4.5: Slowdown over native for the different optimisations of the DBT engine

mode change instructions affects the processor mode and privilege level and must trap to the interpreter. Eliminating traps for instructions that save and store user registers yields significant speedups for `lat_syscall null`, `read` and `write`, as predicted in Section 4.2.

Optimising unprivileged loads and stores results in significant speedups for the `lat_sig install` and `lat_syscall fstat` benchmarks. However, as this optimisation inlines translation table switches in translated code, it comes with significant space overhead. When applied to cold code, the optimisation has adverse effects on hardware caches, resulting in a net slowdown in `lat_syscall null` and `write`. A similar issue manifests itself with inlining coprocessor update instructions. In the last step of our evaluation, we eliminated traps to the interpreter for accesses to coprocessor registers and banked registers. This optimisation causes minor speedups in all benchmarks.

Our results confirm that the architecture-specific optimisations presented in Section 4.2 are useful to reduce the overhead of our hypervisor beyond generic control flow optimisations: on average, we achieve a 51% reduction in slowdown when comparing the results from enabling the shadow stack to the fully optimised case.

4.3.3 Perceived slowdown

Micro-benchmarks are useful for studying the worst-case overhead in interactions between a guest's user applications and kernel. Many real applications however do not continuously perform system calls; the perceived slowdown while running real applications should therefore be lower than the slowdowns obtained from running micro-benchmarks.

In order to evaluate the perceived slowdown caused by our hypervisor, we run a selected number of benchmarks from the *mibench* version 1 suite [59]. Mibench is a benchmark suite that aims to be representative of commercial software normally run on embedded systems. We use wall time as a measure for perceived performance. We use the large input sets provided with mibench and run all benchmarks three times on the same Linux system: once natively, and to show the impact of our optimisations, virtualised without and with our optimisations. Similar to the measurements we did on micro-benchmarks, we do not disable block linking as doing so renders the system unusable.

Figure 4.6 shows that the perceived slowdown in user-space applications can reach almost a factor of five over native in an unoptimised

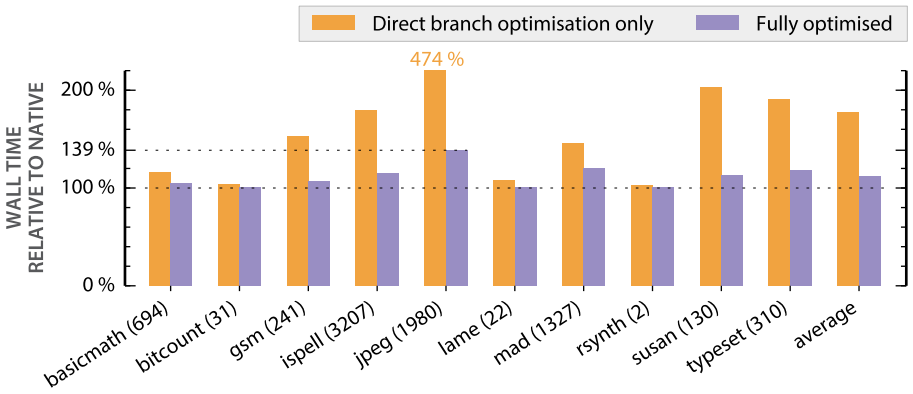

Figure 4.6: Virtualised execution time of selected mibench benchmarks, relative to native execution time

hypervisor, depending on the rate at which the application performs system calls. In our optimised hypervisor, however, the maximum perceived slowdown is limited to 39%, as observed on the `jpeg` benchmark.

The labels in Figure 4.6 also shows for each benchmark the rate at which the application performs system calls. `bitcount` and `rsynth` (speech synthesis) are small benchmarks that perform few system calls. `lame`, an MP3 encoder, is a computationally intensive and long-running benchmark. We can see that for `bitcount`, `rsynth` and `lame`, the very low rate of system calls results in almost negligible overhead. The `basicmath` benchmark is typically used to measure ALU performance. It performs small arithmetic operations and prints the result of every operation; every such print is a `write` system call. Similarly, the `gsm` benchmark, an audio codec, mainly consists of `read` and `write` system calls. Both `basicmath` and `gsm` benchmarks still have a fairly low rate of system calls and therefore the overhead on both benchmarks on an optimised version of our hypervisor is also negligible. For the remaining benchmarks `ispell`, `jpeg`, `mad`, `susan` and `typeset` the rate of system calls is not correlated with the overhead, as they use a wider variety of system calls, some of which are more expensive than others. The remaining overhead in the `jpeg` benchmark is largely caused by memory management.

4.4 Conclusions

We analysed existing techniques for user-space DBT on ARM, and argued that they are not directly suitable for full system virtualisation. We therefore proposed new techniques for spilling registers, and we showed that the best technique to use depends on the usage scenario.

We measured the sources of DBT-based overhead for typical interactions between virtualised kernels and their user applications. As is typical with DBT, much of the overhead can be attributed to control flow, and eliminating traps caused by control flow yields significant speedups. However, we also showed that the remaining overhead can be further reduced by 51% on average, by using binary optimisations specific to ARMv7-A system virtualisation. We found that a naive configuration of our hypervisor makes real applications run up to 5 times slower than native. With our optimisations, however, the maximum perceived slowdown is limited to 39%.

Chapter 5

Trade-offs in cache and memory management

As a hypervisor lives at the lowest level of the software stack, it becomes responsible for managing the hardware caches and TLBs. Guests running on top of the hypervisor will also try to manage these resources, as they are tricked by the hypervisor to assume they are running natively. Our hypervisor must therefore virtualise all cache and TLB operations.

Cache management is not limited to hardware caches, as our hypervisor makes extensive use of software caches both in the DBT engine and in MMU virtualisation. We must keep our software caches up to date with the guests' internal state, either through clever memory management or by reusing a guest's hardware cache maintenance operations. In this chapter, we describe how we have implemented those techniques for both kinds of caches. We evaluate our techniques and find that, due to the way cache and TLB organisation works, shadow translation tables are better managed like a software TLB, while the caches of the DBT engine must be kept up to date through memory protection techniques.

5.1 Introduction

Our hypervisor maintains software caches for its DBT engine and for MMU virtualisation. The DBT caches comprise metadata and translated code; the whole can be regarded as a software instruction cache. Shadow translation tables can be regarded as a software TLB. It is therefore possible to translate a guest's hardware cache maintenance operations into operations that maintain its software caches. Another approach to ensure that the contents of the software caches are consistent with a guest's state consists of write-protecting the sources of the cached data in the guest's virtual address space. Such an approach is required when the guest chooses not to make use of the hardware caches. Picking the best method to manage the hypervisor's software caches is not straightforward, and the best choice depends on the behaviour of the guest.

Configuration registers of the MMU further complicate shadow translation table maintenance, as their values can affect attributes such as permissions and cacheability of several translation table descriptors at once. Performing the equivalent operations on the shadow translation tables proves to be complex: any change to permissions and cacheability by a guest must not affect the hypervisor's own mappings, even though the shadow translation tables contain an indistinguishable mix of mappings for both the hypervisor and the guest.

5.1.1 Overview of hardware instruction and data caches

The ARMv7-A architecture does not define a fixed cache layout; instead, it provides room for up to seven levels of caches, and it offers an interface to query their layout from software. Although specific implementations of the ARMv7-A architecture may also contain scratchpads or tightly coupled memories (TCMs), such memories are not required to be present nor does the architecture provide a standardised interface to query their layout. We therefore exclude TCMs from further discussions.

Each cache level may consist of either separate instruction and data caches, or a unified cache. The hardware platforms we used during the development of our hypervisor typically had two levels of caches, with separate L1 instruction and data caches and a unified L2 cache.

The system control coprocessor provides a register to enable and disable the hardware caches system-wide. Its functionality is, however, limited to controlling all caches of a given type independent of their

level. Furthermore, there are no dedicated controls for unified caches; instead, unified caches must be configured using the data cache controls. Some processor implementations offer additional controls that allow independent control of cache levels, e.g., on a Cortex-A8 processor the L2 cache can be disabled separately from the L1 caches.

Software can perform cache maintenance by writing to registers of the system control processor. Instruction caches are maintained separately from data (or unified) caches, and the available maintenance operations are different for each type of cache. Instruction cache maintenance is limited to two operations that affect all cache levels equally:

1. discarding cache lines that contain an instruction from a given virtual address;
2. discarding all cache contents.

Data (or unified) cache maintenance operations are conceived entirely differently. There are three kinds of operations, namely *clean*, *invalidate*, and *clean and invalidate*. A *clean* operation causes dirty cache lines to be written back to the next cache level or to main memory. An *invalidate* operation discards cache contents, regardless of whether or not a cache line is dirty. There are two ways to select the cache levels and cache lines that will be affected by a data cache maintenance operation:

1. by virtual address, for all cache levels up to the *point of coherence*;
2. by virtual address, for all cache levels up to the *point of unification*;
3. by specific cache level, set, and way.

No single instruction can fully clean or invalidate one or all levels of data caches. Furthermore, it is not possible to clean or invalidate specific addresses at specific cache levels. Instead, operations affect all cache levels up to the point of coherence or unification. The point of coherence is defined as the point in which all 'agents' in the system that can access main memory will observe the same data. Therefore, the point of coherence is typically the main memory. In a uni-processor system, the point of unification, only available in clean operations, is defined as the point at which instruction caches, data caches and translation table walks are guaranteed to observe the same data for any given memory address. This point is typically the first unified cache level [12].

5.1.2 Fine-grained hardware cache control

The MMU can be used to control whether caches are used and how they are used at the level of individual translation table descriptors. Each descriptor contains five bits that define the *memory region attributes*. These attributes define whether memory accesses can be reordered and cached. The ARMv7-A architecture distinguishes between *normal* and *device* memory. Device memory cannot be reordered nor cached. Normal memory can be reordered, and optionally stored in the caches. For cacheable memory regions, *inner* and *outer* caches can be configured independently. Inner caches are those caches closest to the processor, including L1 caches; outer caches are the caches which are the closest to main memory. A cache is either an inner or an outer cache, but it is not architecturally defined from which level the caches are considered to be outer caches. On a system with only two levels of caches, the L1 caches are inner caches and the L2 caches are outer caches. The caches can be configured to operate in write-back or write-through mode, and optionally with write-allocation, if supported by the hardware [12].

An operating system rarely uses all possible cache configurations in its translation tables. As it can be useful to have some 'spare' bits in the translation table descriptors to put data which is ignored by the hardware, the ARMv7-A architecture offers a feature called *TEX remap* which reduces the size of the memory region attributes to three bits. Those three bits are used as an index to a cache configuration stored in registers of the system control processor. The remaining two bits can then be used by the operating system. Every update to the cache configuration registers in the system control processor can change the cache configuration for several translation table descriptors at once, similar to how domains can be used to bypass access permissions for several descriptors at once.

5.1.3 Hardware TLBs

Hardware TLBs cache address translations and first-level translation table descriptors. The ARMv7-A architecture does not define a fixed TLB layout; instead, it defines a set of criteria that TLB implementations must meet. Similar to the caches, implementations may either consist of separate TLBs for instructions and data, or use only a single unified TLB.

When updating translation table descriptors that may have been cached in a TLB, appropriate maintenance is required. Unlike the cache

maintenance operations, ARMv7-A defines different maintenance operations for each type of TLB, and unified TLB operations apply to both data and instruction TLBs on systems with separate TLBs. The individual data and instruction TLB operations are therefore declared as obsolete in the reference manual, but they are nevertheless still used by Linux.

Operating systems such as Linux typically create one set of translation tables per user application process. All these tables contain common descriptors for the kernel and process-specific descriptors. To speed up switching between these sets of translation tables, the ARMv7-A architecture provides a mechanism to reduce TLB maintenance requirements: descriptors can be either *global* or process-specific. An address space identifier (ASID) can be used to associate process-specific descriptors with their process in the TLB. When switching between processes, changing the active ASID together with the active set of translation tables ensures that the TLBs only use entries that belong to the new process [12].

Our hypervisor also leverages the ASID mechanism to avoid TLB maintenance when switching between different sets of shadow translation tables. It therefore does not have to flush the TLB every time the guest switches from a privileged mode to the unprivileged mode and vice versa. This, however, requires all shadowed guest mappings to be non-global, and all guest TLB maintenance operations that would normally apply to a single ASID will hence also affect descriptors marked global in the guest's translation tables. This is an acceptable trade-off, as otherwise the hypervisor needs to flush *all* contents of *all* TLBs on every guest context switch.

5.2 Hardware cache management

We use the hardware caches for both the hypervisor and guests; disabling those caches for either hypervisor or guest causes unacceptable performance penalties. Besides simply enabling the caches, we perform fine-grained cache configuration in the hypervisor's translation tables. Proper tuning reduces hardware cache maintenance requirements and reduces the run-time virtualisation costs.

5.2.1 Tuning cache configurations

We tune the caches for individual memory regions of our hypervisor based on their usage patterns. Read-only regions such as the hypervisor's

code and constants do not require tuning as they are never updated. Only the frequently updated regions, such as the DBT engine's code cache and dynamic memory allocation pools used for shadow translation tables and other data structures, can benefit from tuning.

The code cache has strict cache maintenance requirements, as its contents are continuously modified as 'data' and subsequently executed as instructions. When a block of code is first translated, the translated instructions are written one by one and pass from the L1 data cache through the unified L2 cache to main memory. After translation, the block is executed and the processor fetches the translated instructions through the L1 instruction cache. Cache maintenance is required to ensure coherence: firstly, the L1 data cache must be flushed to the unified L2 cache, and secondly, any cache lines containing instructions from the affected addresses must be cleared from the L1 instruction cache.

By configuring the L1 data cache in write-through mode, we do not explicitly have to flush the newly written instructions to the L2 unified cache. To avoid the performance penalties for accessing external memory, we could configure the unified L2 cache in write-back mode with write-allocation, such that newly written instructions stay inside the L2 cache. When resuming guest execution, instructions will—ideally—be fetched from the unified L2 cache into the L1 instruction cache.

Shadow translation tables require frequent updates to keep them consistent with the guest's translation tables. The hardware TLBs must be kept coherent with the shadow translation tables at all times. A naive implementation may choose to flush all modified descriptors to main memory, either by configuring the caches in write-through mode, or by explicitly flushing the caches after each batch of updates. However, the ARMv7-A architecture specifies that a TLB walk will access the first level of caches or memory after the point of unification. On our platform, the TLBs therefore read through the unified L2 cache and it is sufficient to ensure that any modified descriptors are up to date in that cache, similar to translated instructions. To avoid L1 cache maintenance requirements, we must either disable the L1 caches, or configure them in write-through mode. As we will show in Section 5.5, the best configuration for the unified L2 cache depends on the techniques used to maintain the shadow translation tables.

5.2.2 Virtualising hardware cache operations

A guest running on top of our hypervisor will issue cache maintenance operations as if it were running on native hardware. As the guest can make use of the hardware caches, its cache maintenance operations must be forwarded to the hardware. However, some cache operations for data and unified caches have the potential to discard dirty cache lines containing hypervisor data. Furthermore, data and unified cache operations that explicitly specify a cache level, set and way are not portable across different physical hardware platforms.

On ARM, data and unified caches can only be used when the MMU is enabled. A guest will therefore not maintain the caches when its virtual MMU is disabled. The hypervisor must nonetheless keep the caches and the MMU enabled at all times. To handle this situation, we configure the hardware caches in write-through mode for the guest's memory.

Once the guest enables its virtual MMU, it typically also enables the data and unified caches. In this step, and on later occasions, the guest may attempt to completely invalidate those caches without writing back dirty pages to external memory. This is perfectly valid when the guest runs on its own, but once virtualised, it shares the caches with the hypervisor. The caches may therefore contain hypervisor data which has not yet been written to external memory. Ideally, the hypervisor should either invalidate only non-hypervisor addresses, or clean all hypervisor addresses prior to a full invalidation of the affected caches. Such a clean solution is however not feasible due to architectural limitations.

As noted in Section 5.1.1, fully cleaning or invalidating an entire data or unified cache requires individually cleaning all sets and ways of that cache. The hypervisor can therefore not determine which virtual addresses will be affected by an operation from the guest on a particular set and way. Furthermore, it is not feasible to ensure that the hypervisor's memory is cleaned prior to the operation, as this would require as many cache maintenance operations as the memory footprint of the hypervisor divided by the size of a cache line. Cleaning the hypervisor's entire virtual address space (16 MiB) from a specific cache with a 64-byte line size requires 2^{18} consecutive cache maintenance operations. Our hypervisor therefore forcibly cleans the cache before invalidating its contents whenever invalidation is requested by the guest. This introduces an extra performance cost which cannot be easily quantified.

Cache maintenance operations that affect specific cache levels, sets,

and ways are not portable across different hardware platforms. There are two strategies that a hypervisor can use to virtualise these operations: it can try to detect when a guest wants to perform maintenance on an entire cache, or it can expose the actual physical cache layout to the guest. The ARMv7-A architecture manual recommends that software first queries the cache layout and then performs the appropriate operations by set and way in a loop that affects all sets and ways of a particular cache level. The hypervisor must therefore detect such loops, and translate them into equivalent operations on the hardware caches. Our hypervisor uses a simple translation scheme that translates an operation on the first or last set and way to an operation on the entire cache, and it ignores the guest's operations that affect the remaining sets and ways.

5.3 Shadow translation table management

An operating system frequently modifies its translation tables. When a mapping is updated that has already been shadow mapped, the memory manager needs to find out and update the shadow translation tables accordingly. There are two approaches one can take to implement this update mechanism. The first approach ensures that the guest cannot modify its translation tables without knowledge of the hypervisor, by protecting all associated memory in the shadow translation tables. This approach works rather well for a single set of translation tables, but does not scale if we want to cache multiple sets of shadow translation tables. A second approach is based on regarding the shadow translation tables as a software TLB for the guest. The guest operating system can then manage its shadow translation tables using TLB maintenance operations. This approach can however not be used to identify faults in a guest's memory management code, as it relies on the correctness of the guest.

The access permissions and the cache configurations of translation table descriptors are tied to the MMU configuration registers in the system control coprocessor. As explained in Section 3.4.1, access permissions can be overridden using domains. Similarly, cache configurations depend on TEX remap, as discussed in Section 5.1.2. When a guest changes the configuration of domains or TEX remap, that change may affect the access permissions or the cache configurations of several shadowed descriptors at once. The hypervisor must update the configuration of the physical MMU and the shadow translation tables accordingly.

5.3.1 The memory protection approach

Using memory protection techniques to keep the hypervisor informed of a guest's updates to its translation tables is by far the simplest approach to manage the shadow translation tables. By protecting every mapping created by the guest to physical memory that contains one of its translation tables, all guest updates immediately cause a memory trap, which can then be used to update the shadow translation tables.

The first access of the guest to any virtual address containing one of its active translation tables will *always* be caught by the hypervisor, because the shadow translation tables are filled lazily. During shadow mapping, the hypervisor checks whether the mapped physical memory contains part or all of the guest's first-level translation table, or of one of its second-level translation tables which has been shadow mapped previously. If so, the new mapping is made read-only to the guest.

The hypervisor can easily check whether a given guest physical address (GPA) lies within the memory of the guest's first-level translation table, as the GPA of the latter is always known from the guest's virtual MMU configuration registers. To identify GPAs within the guest's second-level translation tables, however, the hypervisor must maintain a record of the GPAs of all such tables in the guest's active set of translation tables. Furthermore, whenever a second-level table is shadow mapped, the hypervisor must iterate over all pre-existing shadow mappings to protect those that map the memory of the newly mapped second-level table. This is expensive, because the entire set of active shadow translation tables must be verified. Alternatively, the hypervisor could maintain a reverse translation map from all mapped GPA to their corresponding guest virtual addresses (GVAs). However, such reverse mapping merely redistributes the run-time cost of scanning the shadow translation tables while adding a lot more complexity, as every shadow mapping operation must then also maintain the reverse mapping. This only makes sense when the hypervisor needs to shadow map many second-level translation tables, which is not the case for Linux and the benchmarks we have run. Our current implementation therefore scans the shadow translation tables instead of maintaining a full reverse mapping.

Whenever the guest performs a store operation to a protected virtual address, the operation causes a memory trap. The memory manager then performs a software walk of the guest's translation tables to determine whether or not the trap should be forwarded to the guest's kernel. If not, the trap was caused by the hypervisor's memory protection mech-

anisms, and the store operation must be emulated. Emulating loads and stores is a generic task taken care of by the virtual platform. Firstly, it determines which virtual device is affected by the operation based on the target GPA. Only when the guest stores to a memory device, the virtual platform instructs the memory manager to check whether the operation affects a descriptor of one of the guest's translation tables, similar to how the shadow mapper decides which mappings to write-protect, and any existing shadow mappings are updated as necessary.

The obvious downside of leveraging memory protection to maintain the shadow translation tables is that *every* single store of the guest to its translation tables or to surrounding memory will trap. Updates to descriptors that have not been shadow mapped do not need to be handled by the hypervisor, but there is no way to protect the guest's memory at a granularity smaller than a small page, i.e. 4 KiB. This implies that memory surrounding second-level tables, which are only 1 KiB in size, will be protected, resulting in unnecessary traps. To make the situation more complicated, guests may map the memory that contains their translation tables using section descriptors, mapping 1 MiB at a time, to lower TLB pressure. Our hypervisor splits such mappings into smaller descriptors, so that it can avoid unnecessarily protecting the memory surrounding the guest's translation tables. This slightly increases TLB pressure, but avoids the cost of unnecessary memory traps, which would otherwise cause the system to run an order of magnitude slower.

Another downside of using memory protection is its inherent inability to handle batched updates efficiently. A guest may update multiple descriptors consecutively, only performing the minimum necessary cache and TLB maintenance operations at the end of the batch. Once virtualised, every modification to the guest's translation tables is handled individually, and the appropriate cache and TLB maintenance are performed for each update to the shadow translation tables individually.

Last but not least, keeping track of multiple sets of guest translation tables is a complex task with memory protection. Guests such as Linux use a different set of translation tables for each process. During context switches, Linux switches back and forth between different sets of translation tables. In order to avoid recreating the shadow translation tables upon every such guest context switch, the hypervisor should cache multiple sets of shadow translation tables and keep track of the corresponding guest translation tables. To ensure that the cached shadow translation tables stay coherent with the guest's translation tables, however, each

shadow translation table must write-protect all tracked guest translation tables. The cost of this process grows exponentially with the number of sets of shadow translation tables that we want to keep in the cache. In other words, it grows exponentially with the number of applications between which the guest can switch without taking a performance hit for shadow translation table creation. We have therefore omitted caching from our implementation of the memory protection approach.

5.3.2 The software TLB approach

The only way to avoid unnecessary traps and to solve the batched update problem is to avoid using memory protection to manage shadow translation tables altogether, and instead make guests responsible for maintaining their own shadow translation tables, in the same way as they would normally have to maintain the hardware TLBs. In this approach, the hypervisor will not be aware of any of a guest's edits to its translation tables until it performs the necessary TLB maintenance operations. It is reliable, because TLBs cannot be disabled, unlike the hardware caches, and operating systems therefore always have to perform proper maintenance whenever the MMU is enabled. The downside of this approach is that a system that does not use the TLB correctly may, when virtualised, malfunction in entirely different ways than when executed natively.

As explained in Section 5.1.3, entries in the hardware TLBs are tagged with an ASID. When an operating system switches between translation tables, it can avoid TLB maintenance by switching the active ASID, if all non-common descriptors are marked non-global. We can reuse this mechanism to cache multiple sets of shadow translation tables: one set for each ASID [124–126]. Because we use a bounded allocation pool for shadow translation tables, there is a limit on how many sets of shadow translation tables we can keep cached. We therefore evict entire sets using a least-recently-used policy whenever the pool fills up.

5.3.3 Handling guest domains

A guest can use domains to override the access permissions of groups of translation table descriptors. We refer to the combined effects of access permissions and domain configuration as *effective* access permissions. The hypervisor should make all affected shadow mappings behave as intended by the guest's domain configuration, but this cannot be done by applying the guest's domain configuration to the physical MMU:

allowing the guest to configure *manager* mode means allowing it to bypass the hypervisor's memory protection. To virtualise guest domain configurations, the hypervisor can make use of three mechanisms: reconfiguring the physical MMU, updating the guest's access permissions in the shadow translation tables and load/store emulation.

The only prior work we found on guest domain virtualisation are three VMware patents on ARMv6 [124–126]. These patents all describe a hypervisor in which every memory mapping by a guest, regardless of its size, is split into small pages. Therefore all updates of a guest to one of its first-level descriptors always affects a second-level shadow translation table. The patents present six methods for virtualising domains:

1. *L1 iterate and L2 drop/repopulate*: all shadow mappings of the guest are put in one physical domain. The shadow mapper makes the access permissions of shadow descriptors reflect the effective access permissions of the guest's descriptors. Whenever the guest changes the configuration of its domains, the effective access permissions of all mappings in that domain may change. The hypervisor iterates over all first-level shadow descriptors and unmaps (drops) or updates (repopulates) all second-level shadow translation tables whose effective permissions are affected by the change. Repopulation may occur either lazily or immediately [124].

2. *L1 iterate and L2 swizzle*: similar to method 1, but the hypervisor now maintains two second-level shadow translation tables for each first-level guest descriptor in each set of shadow translation tables. One such table contains shadow mappings with access permissions that reflect client mode and the other table contains the access permissions that reflect manager mode [125].

3. *Domain track and L2 drop/repopulate*: improves method 1 by avoiding iteration over all first-level shadow descriptors. To achieve this, the hypervisor maintains a map of guest domains to second-level shadow translation tables and their respective first-level shadow descriptor [126].

4. *Domain track and L2 swizzle*: improvement similar to method 3 for method 2.

5. *L1 tagging*: maintains separate sets of first-level shadow translation tables for each guest domain configuration.

6. *Observational equivalence*: an optimisation to methods 1 to 4 that keeps track of whether or not a change to a domain configuration has an impact on the effective access permissions of first-level guest descriptors; i.e., if the guest descriptor is executable, readable and writable to all privilege levels, its effective access permissions are the same in both client and manager mode. This optimisation can be used to avoid dropping, repopulating and swizzling.

Methods 1 and 3 are slow when a guest frequently changes its domain configuration, because they require updating several shadow descriptors at once. For each edited descriptor, the hypervisor may need to query the guest's original descriptor, and it must ensure that write-protected mappings are not unprotected. Furthermore, such potentially large edits almost always require a TLB flush of all guest descriptors—i.e. all non-global descriptors—for the affected shadow translation tables.

Methods 2 and 4 are an attempt to eliminate the overhead of editing second-level shadow descriptors, at the expense of using more memory to store duplicated tables. To avoid TLB maintenance when switching guest domains, however, separate ASIDs must be allocated per first-level shadow translation table and per guest domain, raising the number of ASIDs used to shadow one set of guest translation tables from exactly 2 to 32 in the worst case scenario, when a guest makes use of all 16 domains. This is only acceptable for a guest that is known not to use many domains or ASIDs. Lastly, all duplicated tables increase the maintenance burden on the shadow translation tables. This will result in extra runtime overhead whenever the guest edits a descriptor, and whenever the hypervisor write-protects guest memory. Method 5 eliminates the need to edit first-level shadow descriptor, but comes with the same disadvantages as methods 2 and 4. Method 6 brings nothing new as it is merely an optimisation to all of the previous methods.

Our hypervisor implements a different approach: it maps 15 of a guest's domains to 15 physical domains. The hypervisor uses a separate domain for its own memory mappings. We let the guest configure its assigned physical domains to either disabled or client mode. Whenever the guest sets a domain to manager mode, the hypervisor configures the associated physical domain in client mode to prevent the guest from bypassing the hypervisor's write-protections. Accesses to virtual addresses whose shadowed access permissions do not match their effective access permissions are emulated by the virtual platform. This approach works well for a guest that does not often require manager mode. For guests

that heavily depend on manager mode to operate correctly, it would, however, cause a lot of extra traps to the memory manager.

All of the techniques described above work well for some guests, but none of them is a perfect solution for all cases. Unlike the DBT techniques presented in Chapter 4, the performance of guest domain virtualisation techniques heavily depends on the behaviour of a particular guest. It is therefore not feasible to carry out an unbiased evaluation of those techniques on a hypervisor that only supports Linux guests.

Domains are a relic present in the ARM architecture since at least ARMv3, and they have been deprecated with the recent virtualisation extensions. The version of the Linux kernel we used during the development of our hypervisor (2.6.28.1) only uses three domains, and the updates it performs to the domain configuration, once it has started the first user-space application, are limited to switching one domain, containing only kernel-specific mappings, from client to manager mode and vice versa. This is done to temporarily allow drivers unrestricted access to the kernel address space. Domain configuration switches therefore mainly occur during device IO operations, but our hypervisor currently only virtualises a minimal set of devices. The Linux drivers for this set of devices do not frequently reconfigure domain access. Starting from Linux 2.6.38, domain support has been disabled for ARMv7-A-based systems because in manager mode, the execute-never bit is ignored, which can result in speculative prefetching of IO device memory [87]. Therefore, domain virtualisation has become irrelevant for Linux guests.

5.3.4 Handling guest cache configurations

Both the hypervisor and its guests can perform fine-grained hardware cache configuration as described in Section 5.1.2. The hypervisor does not use TEX remap to have maximum flexibility when tuning the caches. Our shadow mapping mechanism honours the cache configurations specified by the guest's translation table descriptors. When the guest has not enabled TEX remap, cache configurations can be copied as is into the shadow descriptors. Otherwise, the hypervisor must look up the actual cache configuration in the guest's virtual cache configuration registers. This process is similar to determining effective access permissions based on the guest's domain configuration, and therefore creates a similar maintenance problem: whenever a guest updates its cache configuration registers, all affected shadow descriptors must be updated.

Unlike domain configuration, cache configuration is usually set once by a guest. Our hypervisor therefore implements a simple strategy to ensure that the cache configuration in the shadow translation tables is consistent with the guest's: it drops *all* shadow translation tables whenever the guest cache configuration is updated.

5.4 DBT cache management

The caches of the DBT engine store translated guest kernel code, which can be accessed by its virtual address. If, somehow, the guest updates the *physical* memory that was used as source for a translation, that translation should be invalidated. This may happen because the guest unmaps or remaps a virtual memory region, or, less commonly, because of self-modifying code. Similar to shadow translation table management, the hypervisor can keep track of such updates by leveraging memory protection, or by treating the DBT caches as a software instruction cache.

5.4.1 The memory protection approach

The hypervisor can keep track of changes to a guest's code by write-protecting the associated physical memory. We will refer to blocks of guest code used as input by the translator as *input* blocks, to distinguish them from the translator's output, i.e. the translated blocks. As explained in Section 3.5, blocks are translated as needed, and they are looked up based on the value of the PC in the guest context. This value is a GVA when the guest MMU is enabled, and a GPA when the guest MMU is disabled. For the latter case, write-protecting the input block is easy: it is sufficient to write-protect the corresponding mapping in the hypervisor's translation table. When the guest MMU is enabled, however, the solution is more complex. Firstly, the shadow descriptor that maps the GVA of the input block must be write-protected. Secondly, the hypervisor must ensure that any other mappings to the corresponding GPA are also write-protected, as multiple virtual addresses may be mapped to the same physical address. This operation should be carried out on all existing shadow descriptors as well as any new shadow descriptors created after the translation, for as long as translated code remains in the code cache.

Automatically protecting all new shadow mappings to existing input blocks is straightforward: the DBT engine can maintain a list of physical addresses to protect, which is then queried by the shadow mapper when-

ever a shadow mapping must be created. Our hypervisor implements this list as a search tree of physical addresses of small pages, instead of using the addresses of individual input blocks. This doesn't matter to the shadow mapper as the smallest granularity of memory it can protect is a small page, and it helps to keep the tree reasonably small.

Protecting all existing shadow mappings that map same physical memory is more involved. It is similar to how second-level guest translation tables should be protected when they are shadow mapped, as discussed in Section 5.3.1. The hypervisor either needs to iterate over all existing shadow mappings, or it must maintain a reverse mapping from GPA to GVA. Unlike shadow mapping second-level guest translation tables, translating guest code occurs far more frequently. Maintaining a reverse mapping is therefore a better suited approach.

When using the software TLB approach to manage shadow translation tables, multiple sets of shadow translation tables may be cached, one for each guest ASID, and all cached tables must protect the translator's input blocks. This protection must be applied immediately after the translation of a new block for the active set of shadow translation tables only; inactive sets may be updated lazily. Updating all tables at once will waste time on obsolete tables, as guests do not inform the hypervisor which ASIDs are no longer in use. A lazy updating scheme is thus preferred. Our hypervisor manages a queue per cached set of shadow translation tables that contains the pending protection operations to apply upon the next time that those tables are reactivated.

To prevent the queues for pending protection operations from growing large, and to ensure that the time necessary to switch between different guest ASIDs remains small and bounded, we limit the size of the queues to a fixed number of entries. When a queue overruns, the set of shadow translation tables for the associated guest ASID is flushed.

5.4.2 The software instruction cache approach

Managing the caches of the DBT engine through memory protection comes with problems similar to those studied in shadow translation table management in Section 5.3.1. The trivial approach to overcome the complexity of using memory protection consists of managing the caches of the DBT engine based on how guests manage their instruction caches. Unlike TLBs, however, instruction cache entries on ARMv7-A are not tagged with ASIDs. This means that instruction cache entries cannot be

tied to the kernel and to individual processes. When switching between processes, it is hence necessary to invalidate the entire instruction cache. As we shall see in Section 5.5, such behaviour renders the software instruction cache approach wholly unsuitable for any practical use.

5.5 Evaluation

We evaluate all our techniques using the same hardware and the same lmbench micro-benchmarks as used in Chapter 4. We first evaluate how cache tuning, presented in Section 5.2, affects the performance of our hypervisor's memory manager. We then evaluate the performance impact of the different techniques to manage the hypervisor's software caches. Firstly, we evaluate and discuss our two approaches to manage shadow translation tables from Section 5.3. Secondly, we evaluate the techniques to manage the caches of the DBT engine presented in Section 5.4.

As the lmbench benchmarks perform their own timing measurements, we again grant the guest full and unsupervised access to a number of hardware timers to ensure that the measurements are accurate.

5.5.1 Hardware cache configuration tuning

We evaluate the impact of fine-grained cache configuration tuning on the dynamic memory allocation pool used for shadow translation tables. As the different approaches to manage the shadow translation tables presented in Section 5.3 use this pool in different ways, we have performed separate evaluations for each approach. In order to reduce the impact of external influences such as interrupts, we have executed each benchmark 100 times for each possible cache configuration, and we report averages. We prevent the DBT engine from requesting changes to the shadow translation tables by disabling most protections for self-modifying code.

We do not test cache configurations that put the L1 data cache in write-back mode. As discussed in Section 5.2, such configurations require extra cache maintenance operations to ensure that a store to a shadow translation table is observed by the TLB. Furthermore, using write-allocation for the L1 data cache would wipe that cache whenever a new shadow translation table is initialised, as the first part of the initialisation process fills the entire table with zeros, and both first-level and second-level translation tables are far larger than typical L1 caches.

We compare the results of running the benchmarks with different

cache configurations to a set of results obtained with the L1 and L2 hardware caches disabled. Figure 5.1 presents the results obtained with the memory protection approach to manage shadow translation tables. Figure 5.2 presents the results obtained with the software TLB approach.

When using memory protection, enabling the caches generally causes a small performance improvement, except for `lat_syscall open`. Benchmarks that perform several context switches, i.e. `lat_pipe` and `lat_proc`, require the memory manager to create and modify translation tables more often, and hence run 60 to 90% faster with caches enabled. When comparing the performance among the different cache configurations, the best configuration seems to be "L1 write-through, L2 write-back write-allocate", although the differences are so small that the impact on perceived performance will probably not be significant.

Repeating the measurements while using the software TLB approach to manage shadow translation tables yields an entirely different picture. There is now little to no benefit in enabling the hardware caches for `lat_pipe`, and the `lat_proc` benchmarks run over 10% slower when configuring the unified L2 cache in "write-back write-allocate" mode. Contrary to the measurements with the memory protection approach, the `lat_syscall open` benchmark now benefits from enabling the caches and runs around 10% faster on all tested configurations.

When using memory protection, the guest's shadow translation tables must be write-protected for the guest. Furthermore, every edit by the guest of its translation tables traps to the memory manager, which then compares the guest's edited descriptor with the existing shadow mapping. The hypervisor therefore frequently loads from and store to the shadow translation tables. When using the software TLB approach to manage shadow translation tables, the hypervisor does not have to edit shadow descriptors for guest write-protection. Furthermore, edits are then handled through TLB flushes: a guest TLB flush causes the hypervisor to discard entries in its shadow translation tables. This does not involve reading existing shadow descriptors. The hypervisor thus mostly stores to the shadow translation tables. As can be seen from the results, in such scenario write-allocation and write-back mode in general are detrimental to the performance of the hypervisor: they cause the caches to be filled with data which will never be read again. The best configuration for the software TLB approach therefore puts both the L1 cache and the unified L2 cache in write-through mode.

The `lat_syscall open` benchmark shows interesting behaviour for

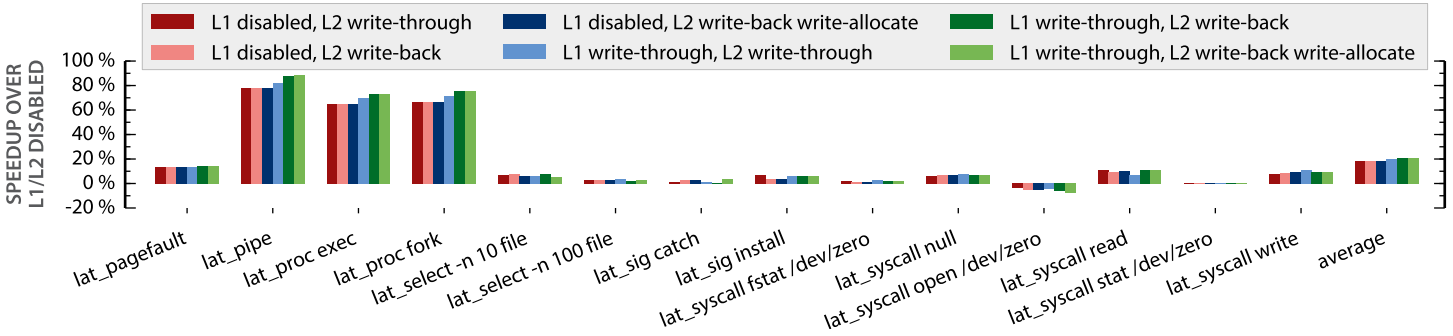

Figure 5.1: Impact of different cache configurations; using memory protection to manage shadow translation tables

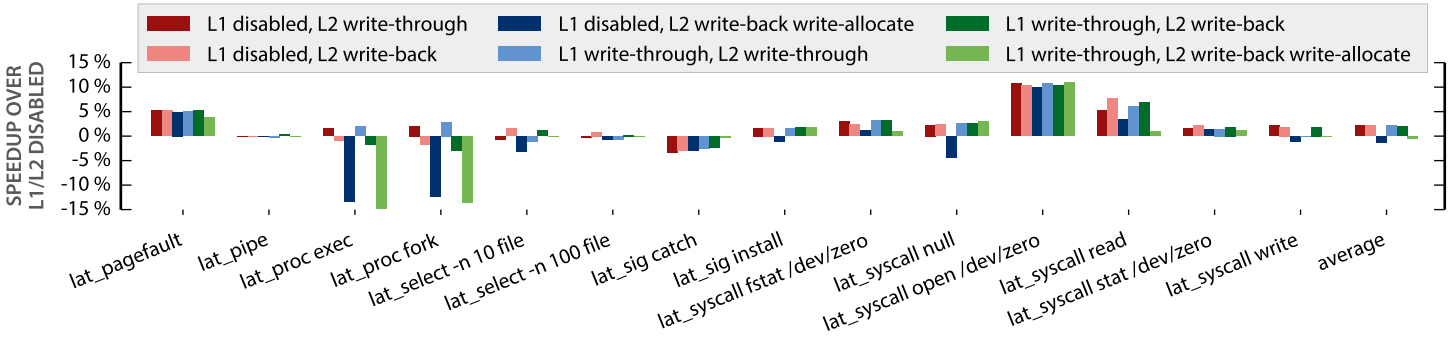

Figure 5.2: Impact of different cache configurations; treating shadow translation tables as a software TLB

which we have not yet identified the underlying causes. When using memory protection to manage the shadow translation tables, enabling the hardware caches on the allocation pool for shadow translation tables causes slight performance degradation. When using the software TLB approach, however, the same benchmark now shows the highest performance gain of all, around 10%, regardless of the cache configuration. Further research is needed to investigate this behaviour.

5.5.2 Shadow translation table management

We now compare the performance impact of the two approaches to manage shadow translation tables we described in Section 5.3. We have executed each benchmark 100 times for each approach, and we report averages. We have normalised our results based on a set of measurements collected on a native system, which was also used in Section 4.3.

Our results are presented in Figure 5.3. The software TLB approach nearly always outperforms the memory protection approach, especially on benchmarks that pressurise the memory manager. The `lat_pagefault` benchmark causes frequent updates to the shadow translation tables. This benchmark maps a 1 MiB file in memory; in Linux, such mappings are constructed using a second-level shadow translation table. It then uses the Linux `msync` call to flush the file contents from memory; this causes all associated descriptors to be invalidated. The benchmark then tries to accesses the file page by page. The Linux kernel will map the file lazily, and each access to a new page causes a page fault that must be handled by the kernel. The kernel updates its translation tables and resumes the application, which then causes a second trap to the hypervisor due to lazy shadowing. The performance difference for this benchmark hence results from the frequency at which Linux edits its translation tables. As already predicted in Section 5.3, the software TLB approach always performs better in such scenarios.

The exact number of context switches and the difference between the memory protection approach and the software TLB approach in slowdown over native execution is shown for each benchmark in Table 5.1. We observe that `lat_pipe` and `lat_proc` also put a lot of stress on the memory manager, but for entirely different reasons than `lat_pagefault`: they all perform a large number of context switches. Upon every context switch, the memory manager needs to switch the active set of shadow translation tables. The software TLB approach has a clear advantage over the memory protection approach, because we did not implement

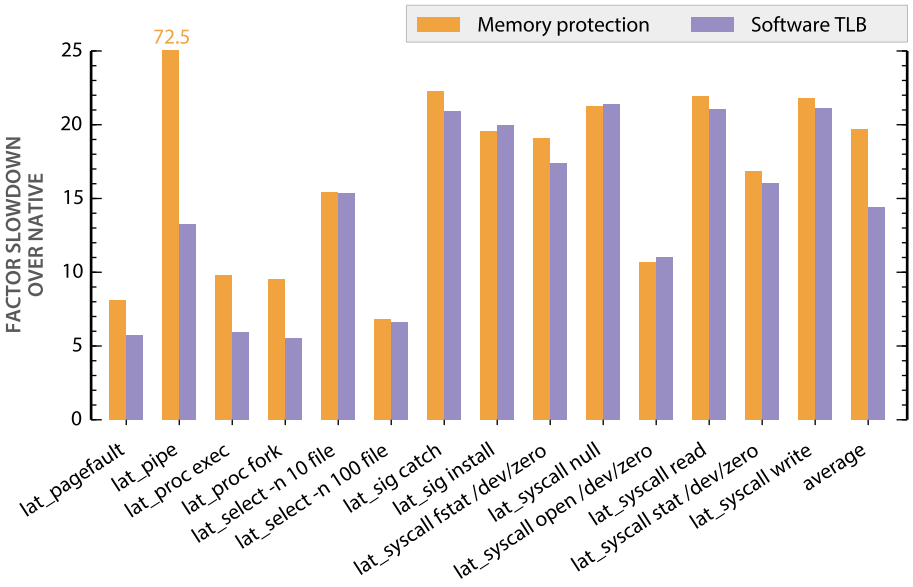

Figure 5.3: Slowdown over native for the different shadow translation table management approaches

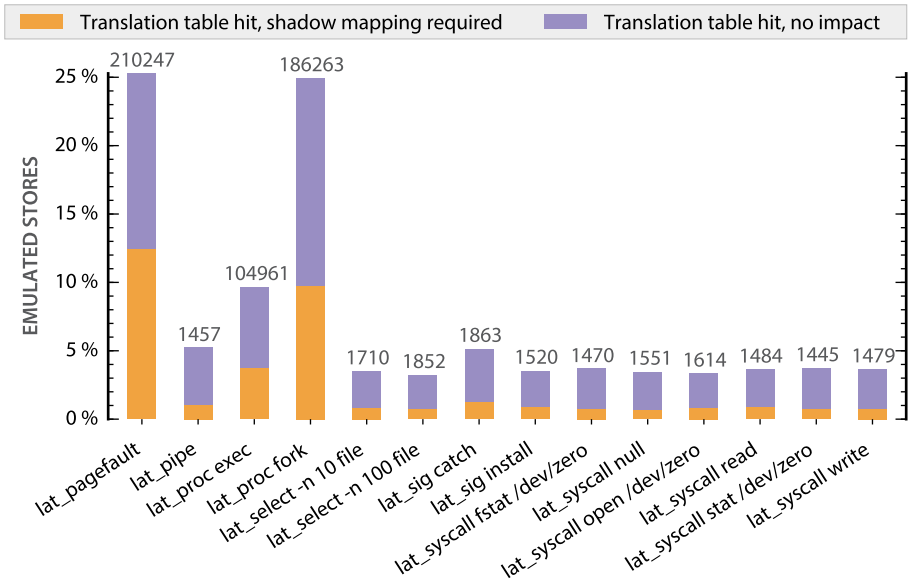

Figure 5.4: Distribution of memory traps when using memory protection to manage shadow translation tables

Table 5.1: Difference in slowdown over native versus guest context switches

Benchmark	Δ times slowdown	Context switches
`lat_pagefault`	2.38	24
`lat_pipe`	59.26	143370
`lat_proc exec`	3.91	617
`lat_proc fork`	3.98	2821
`lat_select -n 10 file`	0.04	12
`lat_select -n 100 file`	0.22	12
`lat_sig catch`	1.36	22
`lat_sig install`	-0.42	12
`lat_syscall fstat /dev/zero`	1.68	12
`lat_syscall null`	-0.16	12
`lat_syscall open /dev/zero`	-0.28	12
`lat_syscall read`	0.87	12
`lat_syscall stat /dev/zero`	0.84	12
`lat_syscall write`	0.69	12

shadow translation table caching for the latter. Without caching, a new set of shadow translation tables must be allocated and initialised upon every context or ASID switch. This causes `lat_pipe` to run almost 60 times slower with the memory protection approach.

On the remaining benchmarks both approaches perform similarly, although the software TLB approach often has a slight advantage over the memory protection approach. This is caused by two drawbacks of the memory protection approach: firstly, it introduces extra TLB pressure by splitting large guest descriptors into small shadow descriptors to achieve fine-grained memory protection. Secondly, the smallest granularity of memory protection is a small page, which is four times as large as a second-level translation table. We thus protect up to 75% more memory than needed, and all stores to protected regions cause memory traps to the hypervisor, which then needs to emulate those store operations.

We have measured the number of emulated stores for each benchmark to better understand the additional sources of overhead for the memory protection approach. Figure 5.4 illustrates how many of the emulated stores actually targeted the guest's second-level translation tables, and how many of those stores updated descriptors that had already been shadowed. On top of each set of bars, for every benchmark, we

mention the total number of emulated stores for that benchmark. On the benchmarks that do not pressurise the memory manager, up to 95% of the stores that trapped targeted memory regions that were not of interest. If we also count the traps for stores to descriptors that were not shadow mapped, we find that up to 99% of the traps were not useful.

The "best" results are obtained by `lat_pagefault`, because it causes the kernel to frequently edit its translation tables: 25% of all emulated stores target protected translation tables, and about half of them actually affect existing shadow mappings. The `lat_proc` benchmarks also cause a large number of stores to be emulated, mainly due to copy-on-write techniques used in the implementation of the `fork` system call.

5.5.3 DBT cache management

We evaluate the two approaches to manage the caches of the DBT engine described in Section 5.4 twice, once for each shadow translation table management approach. This is useful to analyse the impact of caching multiple sets of translation tables on self-modifying code protection. We have executed each benchmark 100 times for each set of measurements, and we report averages. We have normalised our results based on a set of measurements obtained with minimal protection for self-modifying code. This minimalist approach only clears translations from the DBT engine's caches when the associated memory mappings are edited or removed, and therefore enables us to better isolate the run-time overhead specific to proper self-modifying code protection techniques.

Figure 5.5 presents a comparison of our approaches to manage the DBT engine's caches when using memory protection to manage shadow translation tables. Figure 5.6 shows the same comparison for the software TLB approach to shadow translation table management.

As predicted, the software instruction cache approach causes severe slowdowns because the DBT engine's caches are cleared too often. The memory protection approach is therefore the only practical solution to support self-modifying code. We find that, regardless of the approach used to manage shadow translation tables, the worst-case slowdown of self-modifying code protection is then limited to 20%. The execution time of some benchmarks improves when enabling self-modifying code protection. For the software instruction cache approach, these speedups are caused by a combination of few guest cache maintenance operations with breaking the link between shadow translation table management

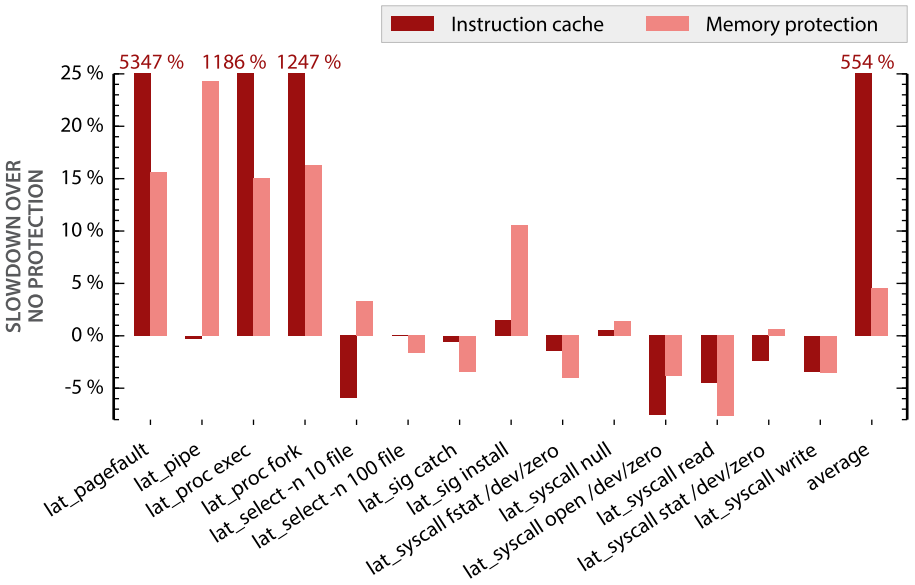

Figure 5.5: Slowdown of self-modifying code protection when managing shadow translation tables using memory protection

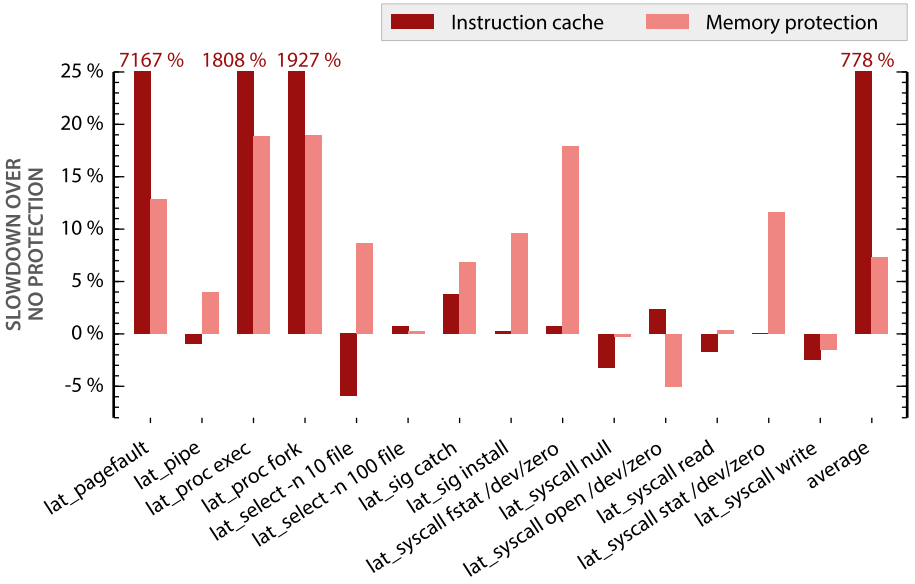

Figure 5.6: Slowdown of self-modifying code protection when managing shadow translation tables as a software TLB

and the caches of the DBT engine. For the memory protection approach, we suspect that the speedups result from different TLB behaviour due to write-protecting and splitting guest memory mappings.

On the benchmarks that do not stress the memory manager, we notice that the memory protection approach to manage the DBT engine's caches performs slightly slower when caching shadow translation tables. This slowdown is caused by queuing memory protections on inactive sets of translation tables.

5.6 Conclusions

We argued that fine-grained tuning of hardware cache configurations is critical to the performance of the hypervisor. The best cache configuration for a given memory region depends on its usage patterns. We have evaluated the impact on the allocation pool for shadow translation tables. We found that, depending on the technique used to manage shadow translation tables, improper cache tuning can cause a slowdown of 15% over not enabling the caches. Proper cache tuning, however, can achieve up to 90% speedup on benchmarks that stress the memory manager. Our evaluation further showed that for the memory protection approach to manage the shadow translation tables, no single cache configuration excels over the others. For the software TLB approach however, we identified the best approach to be write-through mode for all cache levels, contrary to the expectations we put forward in Section 5.2.

We showed how a guest's cache operations cannot always be applied directly to the hardware caches, because they could cause the hypervisor to lose data. We briefly explained how our hypervisor virtualises those operations. The impact of such modification is, however, hard to quantify; we therefore did not evaluate cache operation virtualisation techniques.

We discussed two different approaches to manage shadow translation tables: the memory protection approach and the software TLB approach. Our evaluation confirmed that the memory protection approach cannot handle guest translation table edits efficiently. Furthermore, it showed that caching different sets of shadow translation tables is a necessity when context switches are frequent. For the memory protection approach, the overhead caused by store emulation is rather limited, but the number of emulated stores which are of interest to the hypervisor turns out to be far lower than our expectations: even though we protect only up to 75% more memory than needed, on benchmarks that do not stress

the memory manager up to 99% of all emulated stores did not provide useful information to the hypervisor.

Lastly, we described and evaluated two approaches to manage the caches of the DBT engine: the memory protection approach, and the software instruction cache approach. We predicted that the software instruction cache approach would perform poorly, and this was confirmed by our evaluation. The memory protection approach is the only practically feasible approach, and causes a 20% slowdown in the worst case on our selected lmbench micro-benchmarks.

Chapter 6

Other lessons learnt

A research project starts with several questions and unknowns. Few of those questions actually get answered within the time frame of one Ph.D., and several more questions are raised while looking for those answers. The few answers that yield improvements and positive results have been detailed in the previous chapters. In this chapter, we take a deeper look at a few lessons we have learnt during the course of our research that did not yield easily publishable results. We nevertheless believe that they are valuable information, should anyone attempt to pursue a path similar to ours, or to continue where we left off.

One such lesson stems from lack of experience in embedded systems development, prior to engaging in this project to build a hypervisor from scratch. This caused us to make several mistakes in the design of the software. We started with the wrong language and the wrong hardware abstractions, and had no idea about how to properly test our code.

The second lesson presented in this chapter discusses our insights into performance optimisation of the translator itself, rather than the translated code. This was useful in the early days of the hypervisor, when few binary optimisations were implemented, or few optimisations in general could be enabled during bug-hunting and testing. Later on, however, as the run-time performance of our translations improved, and DBT cache management improved, the performance of the translator itself was no longer a bottleneck and the impact of our work became relatively small in comparison to other performance improvements.

6.1 Design and implementation

Some of the early choices and goals for the design and implementation of the hypervisor were not properly thought out. They resulted in a hypervisor that could only run on a single platform, that could barely support any kind of testing infrastructure, and in which it was fairly hard to incorporate third-party code, both by choice and due to implementation. Eventually all those deficiencies were fixed, but each of them slowed us down on the way to building a fully functional prototype.

6.1.1 Rapid prototyping vs. marketability

We originally wanted to be able to release the hypervisor as an open source project under a very permissive license such as the BSD license. This stopped us from integrating code from popular projects with incompatible licenses. For example, it could have been interesting to integrate drivers from popular open source projects such as U-Boot and the Linux kernel, but we did not pursue that path due to their usage of the GPL.

The only consequence of our choice to avoid viral licenses was that it slowed us down in the development of a working prototype. Recently, we chose to open up the hypervisor under the GPL anyhow. Lesson learnt: unless the software developed during a small-scale research project like ours is highly likely to be marketable, it is most probably not profitable to care about permissive licensing when building a prototype.

6.1.2 Design for testability

Another remarkably bad choice was to tie our prototype implementation to a specific hardware platform, the BeagleBoard. We did not have any experience with other platforms. It was therefore not feasible to design generic and portable abstractions for all necessary hardware devices. We should perhaps have invested some of our time in studying the driver model of the Linux kernel. The ties between our implementation and the BeagleBoard became problematic as the hypervisor kept growing.

In the beginning, the only testing we did involved running a Linux guest until it crashed. Those crashes became increasingly harder to debug. As the amount of code executed prior to the crash kept increasing, tracing became difficult, and no two traces would match due to the unpredictable arrival pattern of timer interrupts. We therefore decided

it was time to set up an environment for unit test automation.

Test automation on hardware is feasible, but hard and expensive. It requires a reliable JTAG debugger that can interact with the hardware regardless of how poorly the software running on that hardware is conceived. It should be able to automatically recover from all kinds of crashes. Unfortunately, all systems and debuggers eventually get stuck, up to the point where power cycling the target, the debugger, or both is required. Good hardware debuggers are also quite expensive, and therefore testing on real hardware would limit our ability to parallelise our tests. Furthermore, when things go wrong, capturing the state of the hardware such as memory and device registers is a really slow process.

We were convinced that a software-based test solution was the better approach. We thus started searching for a suitable simulator. As our implementation was tied to the BeagleBoard and its OMAP3 processor, we had very little choice: the only simulator we found was a QEMU fork. This fork turned out to be far from feature-complete and had problems with timer interrupt delivery. Furthermore, it ran our hypervisor several orders of magnitude slower than real hardware. We wasted some time on patching the simulator, while we should have realised sooner that if the simulator was unstable, it would never make a good testing platform.

Eventually, we had to rework a significant part of the hypervisor to remove as many dependencies on the BeagleBoard hardware as possible. We then ported our hypervisor to ARM's Versatile Express platform, so that we could use ARM's Fast Models-based simulators to automate our tests. While the investment in the rework and the development of the test infrastructure was fairly large, it quickly paid off as it revealed several bugs in the DBT engine and in memory virtualisation that were often silently causing issues with Linux virtualisation, without leading to actual crashes.

6.1.3 C++ for embedded bare-metal software

For a long time we believed that C was the implementation language of choice for our bare-metal hypervisor. The only alternative we considered was C++, and we judged it as heavyweight and difficult to get working on a bare-metal target. We thought we would have to spend considerable effort to port an implementation of the Standard Template Library (STL) to our bare-metal target. Over time, most of these concerns turned out to be based on common myths about the C++ language and the STL.

We eventually discovered the truth and switched the implementation language of our hypervisor to C++11. Our primary motivation was to avoid code duplication, thereby improving stability and correctness.

Our first misconception was that code written in C++ would have a larger footprint than functionally equivalent code written in C. Most valid (ANSI) C code can be compiled by a C++ compiler and will produce a binary which is neither better nor worse in terms of size and run-time performance than the one generated by a C compiler. The same is true even when using object orientation and other C++-specific features. In fact, much of the new syntax introduced in C++ easily maps to concepts that can also be implemented in C with some effort, although often not in a reusable way. One of the benefits of C++ is to avoid such efforts.

Half way through the development of our hypervisor, we switched to a C++ compiler. Contrary to our expectations, the main difficulty was figuring out the right compiler and linker flags to disable a number of C++ features that required run-time support, such as exceptions and run-time type information. We initially did not port any implementation of the STL, as we quickly learnt that C++ compilers do not depend on it.

Eventually, we found ourselves in need of data structures typically provided by the STL. After adapting our hypervisor's custom C library to be more or less standards-compliant, it turned out that much of the STL—headers only—could easily be reused. The STL enabled us to quickly prototype new features and tricks in the hypervisor, as we could finally focus on the specifics of virtualisation rather than wasting time on the implementation of existing algorithms and data structures.

6.2 Translator performance

In the early days of the development of our hypervisor, few optimisations had been implemented, both in the DBT engine and in memory virtualisation. Over time, the performance of our hypervisor had gradually improved from decompressing a Linux kernel overnight, to booting to the shell in about one minute. During this time, improvements to further reduce the boot time were still a logical choice. We therefore investigated which functionality was on the critical path and could be optimised.

One component we identified to be interesting, was the decoder of the DBT engine. Our decoder was basically an adaptation of the

ARM disassembler of the GNU binutils suite.[1,2] This disassembler was originally written to identify and decode each kind of ARM instruction individually. We only had to rework its code to flag sensitive instructions in order to get a functional decoder for our DBT engine. We regarded its ability to distinguish all other non-sensitive instructions individually as a bonus, as we imagined one day to implement a full-fledged interpreter for all instructions. Such an interpreter would enable us to switch dynamically between interpretation and translation, an idea used in Varmosa to reduce the memory usage of the DBT engine's caches by avoiding to translate cold code [69]. Later on, it became clear that we would implement only a minimal interpreter and focus on translating as much code as possible instead. We were left with a huge decoder that provided far more information than the hypervisor actually needed.

A simple way to optimise the decoder was to remove all unnecessary information. Due to the way the decoder was structured, however, this proved to be a daunting and error-prone task. Furthermore, after such optimisation, the readability of the decoder would be reduced, which would make finding bugs substantially harder. We therefore investigated alternative ways to specify and structure our decoder.

6.2.1 Related work

We were definitely not the first to investigate decoders: they are an essential part of compiler toolchains, binary translation tools, instruction set simulators and reverse engineering tools. Decoders are closely related with encoders, as they perform the reverse operation. Encoders and decoders are therefore often found together in the same tools.

The basic function of an instruction decoder is to match binary instruction words (e.g., `0x0100A0E1` for `MOV r0,r1`) to instruction patterns (e.g., `MOV` register), typically based on Boolean functions. For each instruction pattern, we can create such a Boolean function that describes the exact combination of bits required for matching a given instruction word. There are different approaches with respect to the implementation of the Boolean functions, the mapping from a match of a Boolean function to the necessary metadata, etc.

[1] See `opcodes/arm-dis.c` in the GNU binutils source code distribution.

[2] GNU binutils is licensed under the GPL. Reusing its decoder happened during early development of the hypervisor at The University of Manchester, and contradicts our licensing strategy which was devised later. This decoder is the only piece of GPL code that has been reused in our hypervisor.

It is desirable for both encoders and decoders to be generated from a single specification of the ISA they target, because encoding and decoding instructions involves many bit manipulations, and writing all such code by hand is error-prone and nearly always results in code duplication. In literature, such specification languages are also referred to as architecture description languages (ADLs). There are a number of publications that discuss approaches and tools to generate decoders. Few papers also discuss encoder generation. The most notable works in this field are the New Jersey machine code (NJMC) toolkit [99, 100] and language for instruction set architecture (LISA) [32, 128].

The NJMC toolkit can generate encoders and decoders based on a specification of the ISA written in their own ADL, the specification language for encoding and decoding (SLED). SLED is an assembly-style language that on the one hand aims to be close to the kind of instruction set specifications typically found in ISA manuals, but on the other hand aims to avoid duplication even within the descriptions by providing abstractions for recurring (bit) fields. While generating decoders from such descriptions is fairly straightforward, generating the necessary bit manipulation code for encoders is more complex, as it involves solving linear equations with non-linear operators such as bit slicing and sign extension [98]. Several ISA specifications have already been written in SLED, but none of them models any of the ARM architectures.

LISA is a more generic ADL than SLED. It aims to bridge the gap between the hardware design of a processor and its supporting software tools, such as compilers and simulators. LISA models specify the full implementation of an ISA, from instruction coding and behaviour down to the micro-architectural details such as pipeline implementation.

Several other ADLs exist with goals similar to LISA [121]. Some of them, such as ArchC, come with open source tools, but they only focus on the user-mode part of an architecture [17, 103]. While they can be used to generate instruction set simulators, these simulators are only capable of running user-mode applications. They forward system calls to the operating system on which the simulator is executed, similar to how user-mode emulation is implemented in QEMU [23, 24]. Such models and tools do not provide any benefits for our DBT engine.

The hardware-centric approach of LISA and similar ADLs is only feasible if software designers can access the necessary hardware models and the tools to operate on them. To avoid performing error-prone ad hoc work, all such models should be obtained from hardware vendors, which

is far from obvious due to intellectual property protection. Furthermore, the necessary tools are often far too expensive for small-scale research.

The NJMC toolkit follows a software-centric approach. Such an approach has the disadvantage that all models are decoupled from real hardware, but its specifications do not require the same level of detail as with hardware-centric approaches. This makes it feasible for software developers to write and share SLED specifications without depending on hardware vendors and without restrictive non-disclosure agreements.

The NJMC toolkit seems to be the only one of its kind. Other software-centric ADLs and tools exist but they only focus on decoder generation. One such ADL is the generic decoder specification language (GDSL) by Sepp et al. [107]. It claims to improve over SLED and the NJMC toolkit in several ways. Firstly, its syntax would be closer to manufacturer's manuals, thereby improving maintainability. This claim is unfounded because it is subjective. Its tools perform compile-time error checking, but this is not new. They also claim that their prototype tools can generate instruction decoders competitive with decoders in existing software. In the evaluation section of their paper, however, their decoder turns out to be the worst in terms of performance of all decoders evaluated, and this anomaly is dismissed with the argument that decoders are almost never the bottleneck. In a follow-up paper, their evaluation is limited to a comparison with only one other tool; therefore, it cannot be treated as an objective evaluation [109]. GDSL also accommodates for attaching abstract semantics to instructions; while not supported by the original NJMC toolkit, multiple extensions that tie SLED specifications to instruction semantics and even to ABI specifications are described in literature [35, 36, 81, 101]. Lastly, GDSL is only suitable for binary program analysis, as unlike SLED, it cannot generate encoders.

Compiler infrastructure projects like GCC [53] and LLVM [85] use their own tools to generate as much as possible of the compiler backend based on machine specifications. Such machine specifications can also be regarded as a kind of ADL. The tools from GCC, "machine desc", and LLVM, "tblgen", are tightly coupled with the rest of the infrastructure in their respective projects. The machine descriptions used in GCC consist of a blend of architecture descriptions with GCC internals. The LLVM tblgen tool is designed to generate several different kinds of LLVM-specific code, and requires different tblgen backends for the actual code generation. Those backends are often not generic, e.g., there is a specific backend dedicated to generating code for an ARM instruction decoder.

Table 6.1: Encoding of the 32-bit ARM branch instruction (`B`)

31 30 29 28	27	26	25	24	23 ... 0
CC	1	0	1	0	24-bit immediate

We therefore did not further investigate the GCC and LLVM tools.

The NJMC toolkit is quite complex and we never got it to work with our hypervisor. In an attempt not to reinvent the wheel, we decided to look at existing tools that decode and encode binary representations of instructions, such as assemblers, disassemblers, and simulators. It turns out that most of these tools still use ad hoc solutions [58]. We therefore decided to further investigate one of the ideas behind the NJMC toolkit: generating and optimising decoders using decision diagrams [99].

6.2.2 Boolean function representation

Regardless of the model used to describe instruction decoding, matching of instruction patterns always boils down to evaluating a Boolean function. These Boolean functions can be represented in multiple ways.

The source code of the decoders found in GNU binutils, QEMU[3] and SimpleScalar[4] uses a form of $(mask, value)$ pairs to express these Boolean functions [16, 24]. An instruction word matches an instruction pattern if and only if the bitwise conjunction of the instruction word and the mask yields the value. Even though $(mask, value)$ pairs have limited expressibility, they are sufficient for most instruction patterns.

Example The encoding of the 32-bit ARM `B` instruction, a simple branch, is shown in Table 6.1 [12]. Constructing a $(mask, value)$ pair for this instruction is trivial: the mask must contain ones for all the constant bits and zeros for the variable bits; the value contains only the constant bits. The $(mask, value)$ pair for the `B` instruction is then $(\texttt{0x0F000000}, \texttt{0x0A000000})$.

In some cases, instructions impose constraints on their variable bits. For example, two bits may not be allowed to be set at the same time. In other cases, the DBT engine will need to distinguish between different

[3] See `target-arm/translate.c` in the QEMU source code distribution.

[4] See the `target-arm` directory in the SimpleScalar/ARM source code distribution.

uses of the same instruction, such as whether or not the PC is used and how the PC is used. Not all of these constraints can be expressed in a single $(mask, value)$ pair. An example of such a constraint is the encoding of the condition code (CC) field. This field does not allow all bits to be set at the same time, since 32-bit ARM instructions of which the four most significant bits (MSBs) are all set belong to a category of *unconditional* instructions which have entirely different semantics [13]. We therefore need to ensure that we first match unconditional instructions, using one or more separate $(mask, value)$ pairs, and only then we can match all other instructions. Hence, the limited expressibility of $(mask, value)$ pairs puts ordering constraints on their evaluation.

In hardware, Boolean functions are typically implemented using multiplexers and look-up tables. The CAD software used to synthesise such structures makes use of decision trees, in the form of binary decision diagrams (BDDs), to represent Boolean functions [26, 50, 90]. Similar techniques can be used in software. According to Ramsey and Fernández, the NJMC toolkit internally uses BDD optimisation techniques [99].

A BDD represents a Boolean function as a rooted directed acyclic graph (DAG). All internal nodes are called *decision nodes*, and they are labelled with exactly one Boolean variable x_i. There are at most two distinct external nodes: logic one and logic zero. The edge from an internal node with label x_i to one of its children represents the assignment of x_i to either logic one or logic zero. The edge that represents the assignment to logic one is called the *then* edge; the other edge is called the *else* edge.

BDDs can be constructed from any Boolean function by recursively applying Boole's expansion theorem. Let $B = \{0, 1\}$ and let $f : B^n \to B$ be a Boolean function of $n \in \mathbb{N}$ variables. By assigning a value to one variable x_i, we can split f into "then" and "else" Shannon cofactors $f_i^T, f_i^E : B^{n-1} \to B$. The function f then always equals one of these two cofactors, depending on the value of x_i:

$$\begin{aligned} f(x_1, \ldots, x_n) &= x_i f_i^T(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \\ &\quad + \overline{x_i} f_i^E(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \end{aligned}$$

This expansion can also be represented in a graph, as shown in Figure 6.1. Typically, the "then" edge is represented by a solid line and the "else" edge is represented by a dashed or dotted line. A BDD can be obtained by recursively splitting all remaining functions in the external nodes, until all external nodes are either logic one or logic zero.

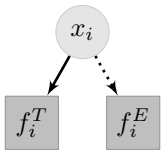

Figure 6.1: First step in BDD construction: one expansion of f

Example The 32-bit ARM branch instruction (B), of which the encoding is shown in Table 6.1, is matched by the following Boolean function:

$$f(x_{31}, ..., x_0) = \overline{(x_{31}x_{30}x_{29}x_{28})}x_{27}\overline{x_{26}}x_{25}\overline{x_{24}}$$

We have assigned the MSB to x_{31}. Figure 6.2 shows an equivalent BDD, constructed by recursively applying Boole's expansion theorem for the variables x_{31} down to x_{24}.

6.2.3 Instruction pattern matching

Identifying a given instruction word with one of many instruction patterns requires evaluating one or more Boolean functions. A simple way to implement a decoder is thus to try such Boolean functions one after the other until a match is found. For the case where the Boolean functions are represented by $(mask, value)$ pairs, this can be implemented by linearly searching a list of such pairs until the given instruction word matches one of those pairs, or by using an equivalent series of conditional statements, each testing the given instruction against one $(mask, value)$ pair. When BDDs are used to represent Boolean functions, we can either evaluate each BDD separately on the given instruction, similar to testing $(mask, value)$ pairs, or we can join all Boolean functions together in one big DAG that directly maps instruction words to instruction patterns.

For architectures with many different instructions, using linear search in the decoder of a hypervisor or emulator is not desirable. Most decoders implemented in practice are hybrids. GNU binutils identifies instructions with $(mask, value)$ pairs. The pairs are not stored in a single list, however. They are split in multiple lists according to an instruction category to speed up decoding. The ARM architecture manual defines six such categories, which can all be easily identified with a few known fixed bits [13]. Instruction category matching in GNU bintuils is also implemented using linear search through a list with $(mask, value)$ pairs.

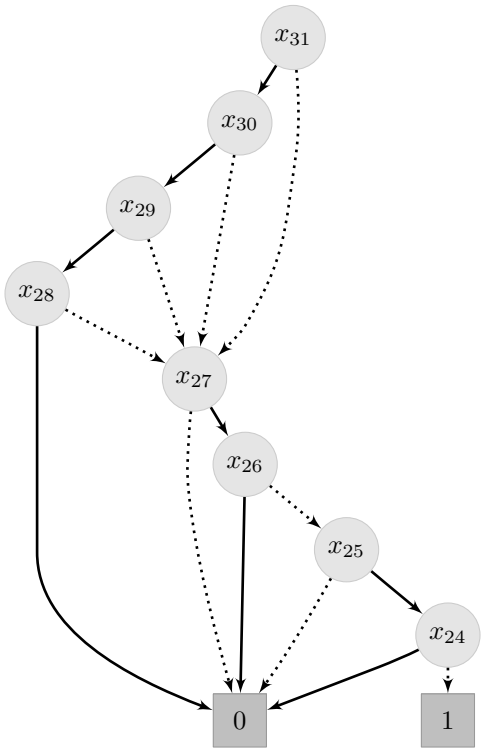

Figure 6.2: BDD to match 32-bit ARM branch instructions

Decoders based on linear search can only be optimised by introducing more hierarchy, because the limited expressibility of $(mask, value)$ pairs introduces ordering constraints. These ordering constraints prevent us from reordering the entries in descending probability of occurrence, which could otherwise be used to improve the average performance of the decoder. The decoders in QEMU and SimpleScalar contain more hierarchy than the decoder in GNU binutils. They also implement all tests using conditional statements rather than lists. Unfortunately, since both decoders are hand-written, the combination of the added hierarchy and the intermingling of code with data renders them unreadable.

To completely avoid the cost of linear search, the decoder can be made "fully hierarchical" by implementing the matching process as searching a single DAG. Such a DAG can be constructed by combining the Boolean functions that match single instruction patterns into one large algebraic decision diagram (ADD). An ADD is similar to a BDD, but it represents a function that outputs a real value instead of logic one or logic zero.

The outcome of the ADD can be chosen so that it uniquely identifies the matched instruction pattern. Because of the inherent properties of BDDs and ADDs, the maximum depth of the search is limited to the bit-width of the instruction.

Example There are two instructions to load a byte from memory into a register in the 32-bit ARM ISA: one performs a load with the current privilege level (`LDRB`) and the other one performs the load as if executed from the unprivileged mode (`LDRBT`). The Boolean functions for their encodings only show subtle differences; therefore the ADD constructed by joining the BDDs of these instructions together, shown in Figure 6.3, simplifies to a small graph which is more efficient for instruction decoding than testing each instruction-specific BDD for a match individually.

6.2.4 Implementation and results

As shown in the previous sections, there are several ways to implement an instruction decoder and encoder. Different parts of our DBT engine require different pieces of information about the encoding and decoding of instructions. Firstly, the translator requires a *coarse-grained* decoder to quickly decide which action to undertake for each instruction word of the guest kernel. This action can indicate that the instruction should be translated into an equivalent sequence to avoid exposure of the modified PC, it can indicate that the instruction should be replaced by an end-of-block hypercall, or that the instruction is safe to copy as-is. Only after this action has been decided, the different fields of the instruction may be queried by specialised translation and interpretation functions. This requires instruction-specific encoders and *fine-grained* decoders.

We designed our DBT engine to support multiple, independent coarse-grained decoders to enable performance comparisons between the different decoders. Fine-grained decoding and encoding is taken care of by a separate part of the hypervisor, which only has a single implementation, implemented from scratch, without the help of external tools and independently of the coarse-grained decoders.

We chose to adopt the ARM disassembler of GNU binutils as our first decoder, mainly because it is easily readable, and it has clear connections with the instruction definitions in the ARM reference manual. This was important, because the decoder had to be adapted to the specific needs of our DBT engine. Pruning unused instruction patterns from this decoder

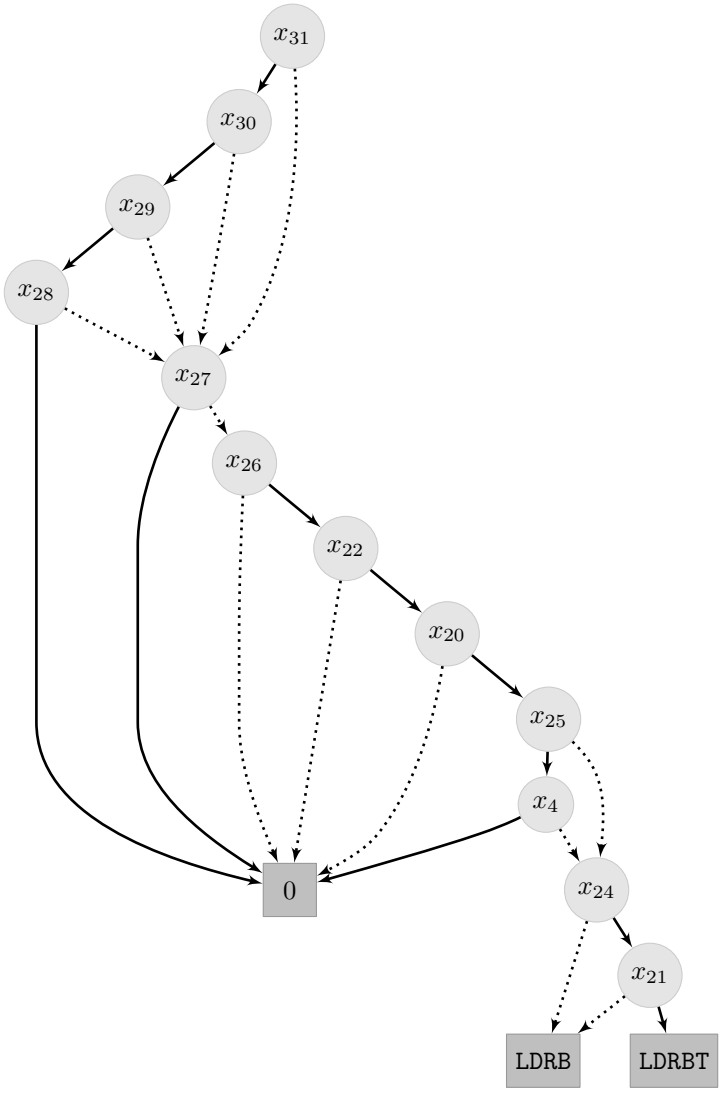

Figure 6.3: ADD to match 32-bit ARM `LDRB` and `LDRBT` instructions

was not a straightforward task: due to ordering dependencies in its lists of $(mask, value)$ pairs, some patterns had to be left in place, and others had to be merged manually. As stated earlier, the resulting decoder was much harder to read and maintain than the original.

Even though we did not get the NJMC toolkit to work with our hypervisor, we still wanted to try to optimise the decoder automatically starting from a readable specification. We therefore created a custom tool that starts from the decoder specified as lists of $(mask, value)$ pairs and creates one large ADD for all instructions combined. Our tool then uses decision diagram reduction techniques based on variable reordering to create a minimal decoder. Because the problem of finding the best variable ordering is NP-hard, our tool uses heuristics implemented in the CUDD library [114]. This yielded a decoder which, at the time, significantly improved the boot time of a Linux guest on top of our hypervisor. At the same time, the specification was as readable as before.

As the hypervisor evolved and changes to the decoder were becoming rare, we eventually fully optimised our original binutils-based decoder by hand. The resulting decoder turned out to be more compact than the automatically optimised decoder, resulting in better cache behaviour. Even though we have tried to further tune the automatically generated decoder, the performance of both decoders is mostly equivalent in the current version of the hypervisor. Furthermore, the translator itself is no longer the biggest performance bottleneck in the hypervisor. We therefore stopped investigating this path.

Chapter 7

Conclusions and future work

This chapter summarises our conclusions of the previous chapters, and relates those conclusions to our original research question: which technological challenges must be solved to enable the wide range of use cases we outlined for DBT on the ARMv7-A architecture? While we have conquered the basics, our work, not unlike the typical doctoral research project, raises more questions than can be answered in one dissertation. We therefore also provide our thoughts on future research opportunities.

7.1 Conclusions

System virtualisation has already proven itself to be useful in many scenarios, both in data centres and for desktop computers. Recent advances in embedded computer hardware and in use cases for embedded systems have sparked interests in virtualisation technology for embedded systems. The existing solutions for data centres and desktop computers can, however, not be readily applied to embedded systems, because of differences in requirements, use cases, and computer architecture.

We therefore researched the challenges involved in fully virtualising an embedded architecture. We chose the ARMv7-A architecture, because ARM is the leading architecture in the embedded and mobile market. While multiple hypervisors for the ARM architecture had already been developed before the start of our research, the majority of them used paravirtualisation techniques due to architectural limitations. Such techniques are, however, known to have several drawbacks. During the course of our research, ARM extended its ARMv7-A architecture with hardware support for full virtualisation. Hypervisors that use these extensions do not suffer from the disadvantages of paravirtualisation, but cannot run on the vast majority of ARM processors in use today.

We proposed to fully virtualise the ARMv7-A architecture using software-only techniques such as DBT and shadow translation tables. Even though such techniques come at a price of software implementation complexity and increased memory footprint, they enable a wide range of use cases beyond the capabilities of hardware extensions, such as optimisations across the border between operating system kernels and applications, emulation and optimisation of legacy software stacks, full system instrumentation and testing, and even load balancing in heterogeneous multi-core systems through cross-architecture virtualisation.

We started our research by identifying the architectural limitations to full system virtualisation on ARMv7-A. We used Popek and Goldberg's classic virtualisability theory as the basis for our analysis. While their theory remains useful today for determining whether or not an architecture is suitable for the construction of efficient hypervisors for full virtualisation, their model and definitions are dated, and do not easily map to modern embedded systems architectures and use cases. We therefore extended their model with paged virtual memory by introducing the concept of an address map, and we derived a new formal constraint for the correctness of such maps. We then studied the effect of IO and

events, and updated their model, definitions and results accordingly.

Our updated theory can be applied to analyse modern RISC architectures, as we have demonstrated with our analysis of ARMv7-A. We identified several problematic instructions that make the ARMv7-A architecture fail the strict requirements for classically virtualisability. We then showed that the architecture is suitable for Popek and Goldberg's hybrid virtualisation, and we argued that DBT-based virtualisation can be regarded as an improvement over their original hybrid virtualisation idea. We then showed how our updated model can be used to identify all instructions that are problematic for DBT-based virtualisation.

We used our knowledge on the architectural limitations of ARMv7-A to build the STAR hypervisor, the first software-only hypervisor for the ARMv7-A architecture. The STAR hypervisor is a bare-metal hypervisor that fully virtualises ARMv7-A on top of an ARMv7-A-based platform. It runs unmodified guest operating systems, decoupled from the hardware through DBT, and uses shadow translation tables to virtualise the MMU.

We analysed existing techniques for user-space DBT on ARM, and argued that they are not directly suitable for full system virtualisation: in user-space, DBT engines often take shortcuts which are only valid for well-behaving applications, where there is no strict requirement to isolate the DBT engine from the application. In full system virtualisation, DBT and MMU virtualisation must be used together to enforce this isolation. Furthermore, kernels often contain handwritten assembly with special system instructions, which require special care in the DBT engine.

We studied how to address the challenges specific to using DBT for full system virtualisation. We started with a fundamental problem in the translator: translations often require an extra register, but such register may not always be readily available. Our translator therefore needs to spill and restore registers to some location in main memory or in other hardware that is accessible to the guest. While solving this problem is trivial for process virtualisation, the solution is more involved for system virtualisation, as the hypervisor must write-protect as much of its data structures from the guest as possible, and the guest's data structures cannot be relied upon. We proposed new solutions for spilling and restoring registers, i.e., spilling to coprocessor registers and spilling using lightweight traps. Our evaluation showed that the best technique to use depends on the usage pattern. When occasionally spilling and restoring one register at a time, coprocessor spilling performs well. When spill and restore operations are frequent, or when multiple register values have to

be spilled and restored together, however, coprocessor access latencies cause severe slowdowns and lightweight traps are to be preferred.

Solving the spilling problem yielded a naive but functional DBT engine suitable for full system virtualisation. The translated code executes far slower than natively, however, as all control flow instructions and all system instructions are handled by traps to the interpreter. We measured which traps are prevalent in typical interactions between virtualised kernels and their user applications. As was to be expected, the majority of the traps can be attributed to control flow instructions, and eliminating these traps yields significant speedups. We then proposed new binary optimisations specific to ARMv7-A system virtualisation that further reduced the remaining overhead by 51% on average. In a test with real applications, we found that our naive DBT engine causes the applications to run up to 5 times slower than native. The optimised version of our DBT engine limits the perceived slowdown to 39% in the worst case.

We continued our research with cache management and memory virtualisation. Cache management comprises of both hardware and software cache management. Our hypervisor makes extensive use of software caches both in the DBT engine and in MMU virtualisation. Those software caches must be kept up to date with the guests' internal state. The caches of the DBT engine must be updated whenever the guest unmaps, remaps, or modifies code that has already been translated. Similarly, the shadow translation tables must be updated whenever the guest modifies its translation tables, and in particular when those modifications affect descriptors that have already been shadow mapped.

We studied the influences of fine-grained hardware cache tuning on the run-time virtualisation cost. We have shown that such tuning can have a big influence on the run-time performance of our hypervisor. The best cache configuration for a given memory region depends on its usage patterns. We evaluated the impact on the allocation pool for shadow translation tables. We found that, on micro-benchmarks, enabling the caches can cause performance decreases of up to 15% with improper tuning, and increases of up to 90% with the best configurations.

We showed that a guest's cache operations cannot always be applied directly to the hardware caches, because they can cause the hypervisor to lose data. However, they do give us valuable information which can be used to manage our hypervisor's software caches. Alternatively, those software caches can also be kept up to date through clever memory management techniques. We discuss how each of these approaches can

be used for shadow translation table management and DBT cache management. We then evaluate both approaches for each kind of software cache. We found that, due to the way caches and TLBs are organised on ARMv7-A, shadow translation tables are best managed by relying on a guest's TLB maintenance operations. The caches of our DBT engine are, however, better managed through memory protection techniques.

We have designed and implemented a proof-of-concept virtualisation platform for the ARMv7-A architecture to study and evaluate all of the above mentioned techniques. We have demonstrated that using DBT for CPU virtualisation results in acceptable levels of run-time overhead when properly optimised for the target architecture. We have shown how software-only MMU virtualisation can work on ARM, and what the best practices are to manage the shadow translation tables and the caches of the DBT engine. We have hence solved the fundamental challenges to enable software-only virtualisation on ARMv7-A. In addition, we have shared some valuable lessons that we have learnt during the design and optimisation of our hypervisor.

7.2 Future work

Stepping back to the larger picture of embedded virtualisation and the uses cases of DBT, we can only conclude that the scientific challenges addressed by our research were a small but necessary step towards software-only virtualisation of ARMv7-A. There is still some interesting research left before a software-only hypervisor like ours can support the use cases we detailed in the introduction.

7.2.1 Scalability

Our research mainly focused on virtualising one guest on one processor core. This is reflected in both our extended model for classic virtualisability and in our hypervisor. While our model can be used to determine whether modern RISC architectures are suitable for the construction of efficient execute-to-trap hypervisors, much like the original, it does not lead to any conclusions on whether or not an architecture can support more than one guest. For example, routing of asynchronous events to hypervisors and guests may prove to be difficult if such events have priorities, as hypervisors have to make sure lower-priority guests cannot block higher-priority guests. Another issue that remains untouched by

our model is multi-threading of privileged code, which may happen on multi-processor, multi-core and multi-threading architectures. It will be interesting to research in which parts of our hypervisor concurrency is to be avoided, where it can be useful, and how it should be dealt with. In the most simple scenario, all work for a particular guest is performed on the same hardware processor, core or thread that executes that guest's code. More interesting scenarios could offload predictive translation and optimisation work to a different, idle core for a performance boost.

In our hypervisor, we have not studied any interactions between guests. We never tried, because we had several interesting research questions to solve just to virtualise a single guest. The main effort in adapting our hypervisor to run multiple guests on multiple processor cores involves implementation work. It will nevertheless be interesting to evaluate how those guests influence each other, both in terms of processor scheduling and in terms of cache and TLB behaviour.

7.2.2 DBT engine and memory manager

There is still room for improvement even for virtualisation of a single guest. Our DBT engine can be further optimised, as the translator mostly operates on single instructions. For specific instruction patterns such as sequences of unprivileged loads and stores, considering multiple instructions at a time can be used to avoid unnecessary, potentially expensive, operations. Similar optimisation opportunities exist with other instructions, but we expect the impact on run-time performance to be of little significance. Furthermore, our translator currently does not handle all kinds of indirect branches in the most efficient way. We have only implemented a shadow stack to handle indirect function returns, and all shadow stack misses and other indirect branches trap to the interpreter. For some of the micro-benchmarks studied in Chapter 4, up to 20% of all control flow instructions executed in the kernel were indirect branches not related to a function return. Implementing a generic binary translation scheme for indirect branches will hence further improve the performance of the translated code. The research question here is: which existing technique, if any, is best fit for the remaining indirect branches?

Another opportunity for further research of the single guest case lies within the memory manager. Although some of the micro-benchmarks we ran tested the overhead of context switches, it will be interesting to investigate how well the system will scale when running several applications concurrently on a full-fledged Linux distribution. Such an

evaluation will show how many different sets of shadow translation tables the hypervisor should be able to cache, how to tune the depth of the protection queues to protect against self-modifying code when using the software TLB approach to manage shadow translation tables, and to better understand how the run-time overhead incurred by the memory manager scales when changing all of those parameters.

During the development of the hypervisor, we have almost exclusively focused on virtualising the Linux kernel. Linux is widely used in embedded systems and uses many interesting features of the ARMv7-A architecture. Virtualising different guest systems mostly boils down to extending the virtual platform, rather than requiring changes to fundamental techniques. We have seen, however, that certain choices in MMU virtualisation, such as the domain virtualisation mechanism, may depend on the run-time behaviour of the guest. It could therefore be useful to study which domain virtualisation techniques are practically useful, depending on the guest operating system and its applications.

7.2.3 Combining DBT with hardware virtualisation

The most interesting work is perhaps realising one of the many practical use cases that DBT offers, such as leveraging the hypervisor for full system debugging, instrumentation and testing. In this context, one could study whether DBT can be combined with hardware virtualisation extensions, by dynamically switching a guest between hardware virtualisation and software virtualisation. Such switching is useful for debugging, to run a guest at native speed up to an interesting point for debugging, and then switching to software techniques that maximise the possibilities to inspect and modify the state of the guest.

The main difficulty in switching from hardware CPU virtualisation to using the DBT engine lies in properly capturing all of the guest's virtual CPU state into the hypervisor's guest context. The DBT engine can then be enabled as if switching from user space to kernel space in the current implementation. Likewise, for memory virtualisation, the MMU configuration must be captured, before we can construct shadow translation tables. Such tables can be constructed from scratch when switching from hardware virtualisation to software virtualisation, similar to how a guest's translation table switch is currently handled. The shadow translation tables will be populated lazily, as needed. This approach to switching CPU and MMU virtualisation requires the hypervisor's software caches to be flushed upon every switch.

Device and IO virtualisation should be treated differently, as for most devices it is very hard, if not unfeasible, to fully save and restore their state. Therefore, device virtualisation should not be switched on and off dynamically. Guests can be granted unrestricted access to non-interesting devices; accesses to other devices should pass through our load/store emulation layer. Several interesting questions remain:

- Is the overhead of switching between software and hardware virtualisation acceptable for debugging and testing?
- Are there better solutions that can avoid clearing the hypervisor's software caches on every switch?
- How do we initiate the switch between software and hardware virtualisation?
- Can we combine hardware CPU virtualisation with software MMU virtualisation to quickly detect cache and TLB management bugs in a guest?

List of tables

List of figures

List of abbreviations

ABI	application binary interface
ADD	algebraic decision diagram
ADL	architecture description language
ALU	arithmetic logic unit
ANSI	American National Standards Institute
ASID	address space identifier
BDD	binary decision diagram
BSD	Berkeley Software Distribution
CC	condition code
CPU	central processing unit
CTO	chief technology officer
DAG	directed acyclic graph
DBT	dynamic binary translation
DSP	digital signal processor
EABI	embedded application binary interface
FPU	floating-point unit
GCC	GNU Compiler Collection

GDSL	generic decoder specification language
GOT	global offset table
GPA	guest physical address
GPL	(GNU) General Public License
GVA	guest virtual address
HPA	host physical address
HVA	hypervisor virtual address
ID	identifier
IO	input/output
ISA	instruction set architecture
ITRI	Industrial Technology Research Institute of Taiwan
JTAG	Joint Test Action Group
KVM	Kernel-based Virtual Machine
LISA	language for instruction set architecture
LPA	large physical address
LPAE	large physical address extension
MIT	Massachusetts Institute of Technology
MMIO	memory-mapped input / output
MMU	memory management unit
MSB	most significant bit
NJMC	New Jersey machine code
PC	program counter

PMIO	port-mapped input / output
RAM	random access memory
RISC	reduced instruction set computer
SBT	static binary translation
SDT	software dynamic translation
SLED	specification language for encoding and decoding
SoC	system-on-chip
STAR	software translation for ARM
STL	Standard Template Library
TCM	tightly coupled memory
ThumbEE	Thumb execution environment
TLB	translation lookaside buffer
VM	virtual machine
VMSA	virtual memory system architecture

List of symbols

Γ	Set of IO states (Dong and Hao [48])
Σ	Set of machine states
$\mathcal{E}$	Set of all asynchronous event functions
$\mathcal{I}$	Set of all instruction functions
$\mathcal{M}_P$	Set of equally privileged processor modes
$\mathcal{P}$	Set of all physical addresses
$\mathcal{S}$	Set of all possible machine state mapping functions
$\mathcal{V}$	Set of all virtual addresses
$\mathcal{V}_A$	Set of virtual addresses mapped by address map A
$\mathcal{X}$	Set of all access permission specifiers
A	Memory map
C	Configuration registers
D^M	MMIO device state
D^P	PMIO device state
E	Contents of physical memory (Popek and Goldberg [97]); contents of the physical address space excluding device registers (revised)
G	General-purpose registers
m	Processor mode
pc	Program counter
r	Relocation-bounds register

S	Machine state
T_A	Translation function of address map A
e	Asynchronous event function
i	Instruction function
m_U	The unprivileged processor mode
p	Physical address
v	Virtual address
x	Access permission specifier

Bibliography

[1] Robin J. Adair, Richard U. Bayles, Les W. Comeau, and Robert J. Creasy. A virtual machine system for the 360/40. Technical Report 320-2007, International Business Machines Corporation, Cambridge Scientific Center, 1966.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 2–13, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

[3] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology – Secure Virtual Machine Reference Manual*, 3.01 edition, May 2005.

[4] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. VMI: an interface for paravirtualization. In *Proceedings of the Linux Symposium*, volume 2 of *2006 Linux Symposium*, pages 371–386, July 2006.

[5] Apple Inc. Rosetta. The most amazing software you'll never see, 2011. URL `https://web.archive.org/web/20110107211041/http://www.apple.com/rosetta`.

[6] ARM Architecture Group. *ARM® Generic Timer Specification*. ARM Limited, PRD03-GENC-009660 9.0x edition, October 2010.

[7] ARM Architecture Group. *Large Physical Address Extensions Specification*. ARM Limited, PRD03-GENC-008469 15.0x edition, October 2010.

[8] ARM Architecture Group. *ARM® Performance Monitoring Architecture version 2 Virtualization Extensions*. ARM Limited, DSA09-PRDC-010447 6.0x edition, October 2010.

[9] ARM Architecture Group. *Virtualization Extensions Architecture Specification*. ARM Limited, PRD03-GENC-008353 14.0x edition, October 2010.

[10] ARM Limited. *Cortex™-A8 Technical Reference Manual – Revision: r3p2*. ARM Limited, ARM DDI 0344K (ID060510) edition, May 2010.

[11] ARM Limited. big.LITTLE processing, 2011. URL `http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php`.

[12] ARM Limited. *ARM® Architecture Reference Manual: ARM®v7-A and ARM®v7-R edition – errata markup*. ARM Limited, ARM DDI 0406B_errata_2011_Q3 (ID120611) edition, December 2011.

[13] ARM Limited. *ARM® Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. ARM Limited, ARM DDI 0406C.c (ID051414) edition, May 2014.

[14] François Armand and Michel Gien. A practical look at micro-kernels and virtual machine monitors. In *6th IEEE Consumer Communications and Networking Conference*, CCNC 2009, pages 1–7, Piscataway, New Jersey, USA, January 2009. IEEE.

[15] François Armand, Gilles Muller, Julia Laetitia Lawall, and Jean Berniolles. Automating the port of Linux to the VirtualLogix hypervisor using semantic patches. In *4th European Congress ERTS Embedded Real Time Software*, ERTS 2008, pages 1–7, January 2008.

[16] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002. ISSN 0018-9162.

[17] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5): 453–484, 2005. ISSN 0885-7458.

[18] B-Labs Ltd. Codezero project overview, 2012. URL `https://web.archive.org/web/20120101123359/http://www.l4dev.org/codezero_overview`.

[19] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: the design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett Packard Laboratories, June 1999.

[20] Barebox. The Barebox bootloader. URL `http://www.barebox.org/`.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37:164–177, October 2003. ISSN 0163-5980.

[22] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 44:124–135, December 2010. ISSN 0163-5980.

[23] Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo. Emulating operating system calls in retargetable ISA simulators. Technical Report IC-03-029, Universidade Estadual de Campinas, Instituto de Computação, São Paulo, Brazil, December 2003.

[24] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, USENIX '05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[25] Jon Brodkin. VMware acquires Trango, debuts mobile hypervisor. *Network World*, November 2008. URL `http://www.networkworld.com/article/2269455/smartphones/vmware-acquires-trango--debuts-mobile-hypervisor.html`.

[26] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986. ISSN 0018-9340.

[27] Prashanth Bungale. ARM virtualization: CPU & MMU issues, December 2010. URL `https://labs.vmware.com/download/68`.

[28] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual Execution Environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1.

[29] Jeffrey P. Buzen and Ugo O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, AFIPS '73, pages 291–299, New York, NY, USA, 1973. ACM.

[30] Martin Campbell-Kelly and Daniel D. Garcia-Swartz. Economic perspectives on the history of the computer time-sharing industry, 1965–1985. *IEEE Annals of the History of Computing*, 30(1):16–36, 2008. ISSN 1058-6180.

[31] Markos Chandras. ARM virtualization. Master's thesis, The University of Manchester, School of Computer Science, August 2011.

[32] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*, chapter 5, pages 95–130. Morgan Kaufmann, June 2008. ISBN 978-0-12374-287-2.

[33] Jiunn-Yeu Chen, Bor-Yeh Shen, Quan-Huei Ou, Wuu Yang, and Wei-Chung Hsu. Effective code discovery for arm/thumb mixed isa binaries in a static binary translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 19:1–19:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1400-5.

[34] Cristina Cifuentes and Vishv M. Malhotra. Binary translation: static, dynamic, retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 340–349. IEEE, November 1996.

[35] Cristina Cifuentes and Shane Sendall. Specifying the semantics of machine instructions. Technical Report 422, University of Queensland, Department of Computer Science and Electrical Engineering, Brisbane, Australia, December 1997.

[36] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[37] Gerald Coley. *BeagleBoard System Reference Manual Revision C4*. BeagleBoard.org, BB_SRM Revision 0.0 edition, December 2009.

[38] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[39] CORDIS. Report on the EC Workshop on Virtualisation, Consultation Workshop "Virtualisation in Computing", September 2009. URL `ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/computing/report-on-the-ec-workshop-on-virtualisation_en.pdf`.

[40] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of the 12th Annual Linux Symposium*, pages 45–56, July 2010.

[41] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences building the Linux ARM hypervisor. Technical Report CUCS-010-13, Columbia University, Department of Computer Science, April 2013.

[42] Christoffer Dall and Jason Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5.

[43] Henri De Veene. Virtualisatie van android. Master's thesis, Ghent University, Faculty of Engineering and Architecture, June 2012.

[44] Peter J. Denning. Origin of virtual machines and other virtualities. *IEEE Annals of the History of Computing*, 23(3):73, 2001. ISSN 1058-6180.

[45] DENX Software Engineering. Das U-Boot – the universal boot loader. URL `http://www.denx.de/wiki/U-Boot`.

[46] Scott W. Devine, Lawrence S. Rogel, Prashant P. Bungale, and Gerald A. Fry. Virtualization with in-place translation, December 2009. URL `http://www.google.com/patents/US20090300645`. US Patent App. 12/466,343; original assignee: VMware Inc.

[47] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. ARMvisor: System virtualization for ARM. In *Proceedings of the Linux Symposium*, pages 93–107, July 2012.

[48] Hanfei Dong and Qinfen Hao. Extension to the model of a virtualizable computer and analysis on the efficiency of a virtual machine. In *Second International Conference on Computer Modeling and Simulation*, volume 2 of *ICCMS 2010*, pages 503–507, Los Alamitos, California, USA, January 2010. IEEE Computer Society.

[49] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC vision, 2010. URL `http://www.hipeac.net/roadmap`.

[50] Rüdiger Ebendt, Görschwin Fey, and Rolf Drechsler. *Advanced BDD optimization*. Springer, Dordrecht, The Netherlands, first edition, August 2005.

[51] Sarah Tawfik Adel El Shal. Virtualization for embedded systems and smartphones - an ARM hypervisor. Master's thesis, Vrije Universiteit Brussel, Faculty of Applied Sciences, December 2011.

[52] Daniel R. Ferstay. Fast secure virtualization for the ARM platform. Master's thesis, The University of British Columbia, Faculty of Graduate Studies (Computer Science), June 2006.

[53] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. URL `https://gcc.gnu.org/`.

[54] Steve Furber. *ARM system-on-chip architecture*, page 39. Addison-Wesley, Boston, Massachusetts, USA, second edition, 2000. ISBN 0201675196.

[55] Jérome Gallard, Adrien Lèbre, Geoffroy Vallée, Christine Morin, Pascal Gallard, and Stephen L. Scott. Refinement proposal of the Goldberg's

theory. In *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '09, pages 853–865, Berlin - Heidelberg, Germany, 2009. Springer-Verlag. ISBN 978-3-642-03094-9.

[56] Robert P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM.

[57] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(9): 34–45, September 1974. ISSN 0018-9162.

[58] Dai Guilan, Zhang Suqing, Tian Jinlan, and Jiang Weidu. A study of compiler techniques for multiple targets in compiler infrastructures. *SIGPLAN Notices*, 37(6):45–51, June 2002. ISSN 0362-1340.

[59] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15. URL `http://dx.doi.org/10.1109/WWC.2001.15`.

[60] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6.

[61] Thomas Heinz and Reinhard Wilhelm. Towards device emulation code generation. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 109–118, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3.

[62] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-126-2.

[63] Gernot Heiser. The Motorola Evoke QA4—a case study in mobile virtualization. Technology white paper, Open Kernel Labs, July 2009.

[64] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0195-4.

[65] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms

in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: 10.1109/CGO.2007.10. URL `http://dx.doi.org/10.1109/CGO.2007.10`.

[66] Gerhard E. Hoernes and Leo Hellerman. An experimental 360/40 for time-sharing. *Datamation*, 14(4):39–42, April 1958.

[67] Jerry Honeycutt. *Microsoft Virtual PC 2004 Technical Overview*. Microsoft Corp., November 2003.

[68] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[69] Perry L. Hung. Varmosa: just-in-time binary translation of operating system kernels. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2009.

[70] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, CCNC 2008, pages 257–261, Piscataway, New Jersey, USA, January 2008. IEEE. ISBN 978-1-4244-1457-4.

[71] IBM. *PowerPC® Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. International Business Machines Corporation, 3.0 edition, July 2005.

[72] IBM. PowerVM Lx86 for x86 Linux applications, July 2011. URL `http://www.ibm.com/developerworks/linux/lx86/index.html`.

[73] Hiroaki Inoue, Akihisa Ikeno, Masaki Kondo, Junji Sakai, and Masato Edahiro. VIRTUS: a new processor virtualization architecture for security-oriented next-generation mobile terminals. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 484–489, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6.

[74] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z*. Intel Corporation, December 2011.

[75] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, December 2011.

[76] Neil Jones and René Hansen. The semantics of "semantic patches" in Coccinelle: program transformation for the working programmer. In

Zhong Shao, editor, *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 303–318. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76636-0.

[77] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 34–42, New York, NY, USA, 1991. ACM. ISBN 0-89791-394-9. doi: 10.1145/115952.115957. URL `http://doi.acm.org/10.1145/115952.115957`.

[78] Robert Kaiser and Stephan Wagner. The PikeOS concept – history and design. Whitepaper, SYSGO AG, January 2008. URL `http://www.sysgo.com/nc/news-events/document-center/whitepapers/pikeos-history-and-design-jan-2008/`.

[79] Greg Kroah-Hartman. The kernel configuration and build process. *Linux Journal*, 2003(109):3, May 2003. ISSN 1075-3583.

[80] Danielius Kudinskas. Virtualizing the ARM - the ARM hypervisor. Bachelor's thesis, The University of Manchester, School of Computer Science, April 2010.

[81] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 291–300, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2.

[82] Sung-Min Lee, Sang-Bum Suh, and Jong-Deok Choi. Fine-grained I/O access control based on Xen virtualization for 3G/4G mobile devices. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 108–113, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5.

[83] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. In *13th Asia-Pacific Computer Systems Architecture Conference*, ACSAC 2008, pages 1–9, Los Alamitos, California, USA, August 2008. IEEE Computer Society. ISBN 978-1-4244-2682-9.

[84] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999. ISBN 1558604960.

[85] LLVM project. The LLVM compiler infrastructure, . URL `http://llvm.org/`.

[86] LLVM project. "libc++" C++ standard library, . URL `http://libcxx.llvm.org/`.

[87] Catalin Marinas. ARM: 6384/1: Remove the domain switching on ARMv6k/v7 CPUs, 2010. URL http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=247055aa21ffef1c49dd64710d5e94c2aee19b58.

[88] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268299.1268322.

[89] Alex Merrick. ARM hypervisor - virtualising the ARM processor. Bachelor's thesis, The University of Manchester, School of Computer Science, June 2010.

[90] Shin-ichi Minato. *Binary decision diagrams and applications for VLSI CAD*, volume 342 of *The Springer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Norwell, MA, USA, first edition, 1996. ISBN 0-7923-9652-9.

[91] MIPS. *MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set*. MIPS Technologies, Inc., MD00086, 3.02 edition, March 2011.

[92] MIPS. *MIPS® Architecture for Programmers Volume II-B: The microMIPS32™ Instruction Set*. MIPS Technologies, Inc., MD00582, 3.05 edition, April 2011.

[93] MIPS. *MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set*. MIPS Technologies, Inc., MD00087, 3.02 edition, March 2011.

[94] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3.

[95] Yoann Padioleau, René Rydhof Hansen, Julia Laetitia Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, PLOS '06, pages 10:1–10:6, New York, NY, USA, 2006. ACM. ISBN 1-59593-577-0.

[96] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the ARM architecture. *Journal of Systems Architecture*, 59(3):144–154, March 2013. ISSN 1383-7621.

[97] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17: 412–421, July 1974. ISSN 0001-0782.

[98] Norman Ramsey. A simple solver for linear equations containing nonlinear operators. *Software—Practice & Experience*, 26(4):467–487, April 1996. ISSN 0038-0644.

[99] Norman Ramsey and Mary F. Fernández. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA, January 1995.

[100] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19(3):492–524, May 1997. ISSN 0164-0925.

[101] Norman Ramsey, Jack W. Davidson, and Mary F. Fernández. Design principles for machine-description languages. Unpublished draft available from `http://www.cs.tufts.edu/~nr/pubs/desprin-abstract.html`, 2001.

[102] Red Bend Software. vLogix Mobile for mobile virtualization, 2014. URL `https://www.redbend.com/en/products-solutions/mobile-virtualization/vlogix-mobile-for-mobile-vitrualization`.

[103] Sandro Rigo, Rodolfo J. Azevedo, and Guido Araujo. The ArchC architecture description language. Technical Report IC-03-015, Universidade Estadual de Campinas, Instituto de Computação, São Paulo, Brazil, June 2003.

[104] Mendel Rosenblum. VMware's virtual platform: a virtual machine monitor for commodity PCs. In *Hot Chips 11 (1999)*, August 1999.

[105] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, May 2005. ISSN 0018-9162.

[106] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42:95–103, July 2008. ISSN 0163-5980.

[107] Alexander Sepp, Julian Kranz, and Axel Simon. GDSL: A generic decoder specification language for interpreting machine language. *Electron. Notes Theor. Comput. Sci.*, 289:53–64, December 2012. ISSN 1571-0661.

[108] Steven She and Thorsten Berger. Formal semantics of the Kconfig language. Technical note, January 2010.

[109] Axel Simon and Julian Kranz. The GDSL toolkit: Generating frontends for the analysis of machine code. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 7:1–7:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2649-0.

[110] Alexey Smirnov. KVM-ARM hypervisor on Marvell Armada-XP board. Talk at Cloud Computing Research Center for Mobile Applications (CCMA) Low Power Workshop (Taipei), June 2012. URL `ftp://220.135.227.11/Low-Power%20Workshop%202012%20June%204th%20PDF/Low-Power%20Workshop%202012%20June%204th%20PDF/07-CCMA-workshop-taipei.pdf`.

[111] Alexey Smirnov, Mikhail Zhidko, Yingshiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-Cker Chiueh. Evaluation of a server-grade software-only ARM hypervisor. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 855–862, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5028-2.

[112] Brad Smith. ARM and Intel battle over the mobile chip's future. *Computer*, 41(5):15–18, May 2008. ISSN 0018-9162.

[113] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2005. ISBN 1558609105.

[114] Fabio Somenzi. CUDD: CU Decision Diagram package, 2012. URL `http://vlsi.colorado.edu/~fabio/CUDD/`.

[115] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. Hdtrans: An open source, low-level dynamic instrumentation system. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 175–185, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1220166. URL `http://doi.acm.org/10.1145/1134760.1220166`.

[116] Christopher Strachey. Time sharing in large, fast computers. In *International Conference on Information Processing*, pages 336–341, 1959.

[117] SuperH. *SuperH™ (SH) 64-Bit RISC Series: SH-5 CPU Core, Volume 1: Architecture*. SuperH, Inc., 05-cc-10001, v1.0 edition, February 2002.

[118] SYSGO AG. PikeOS comes with full virtualization for Renesas R-CAR H2, June 2014. URL `http://www.sysgo.com/news-events/press/press/details/article/pikeos-comes-with-full-virtualization-for-renesas-r-car-h2/`.

[119] Texas Instruments Inc. *OMAP35x Applications Processor: Technical Reference Manual*. Texas Instruments Inc., SPRUF98D edition, October 2009.

[120] Texas Instruments Inc. *OMAP35x Applications Processor: Technical Reference Manual*. Texas Instruments Inc., SPRUF98Y edition, December 2012.

[121] Hiroyuki Tomiyama, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. Architecture description languages for systems-on-chip design. In *Proceedings of the Asia Pacific Chip Design Language (APChDL) Conference*, pages 109–116, October 1999.

[122] TRANGO Virtual Processors. Virtualization for mobile, 2009. URL `https://web.archive.org/web/20090103050359/http://www.trango-vp.com/markets/mobile_handset/usecases.php`.

[123] Transitive Corp. Cross-platform virtualization, 2008. URL `https://web.archive.org/web/20080914184751/http://www.transitive.com`.

[124] Harvey Tuch, Prashanth P. Bungale, Scott W. Devine, and Lawrence S. Rogel. Virtualizing processor memory protection with "L1 iterate and L2 drop/repopulate", June 2012. URL `http://www.google.com/patents/US20120151116`. US Patent App. 12/966,766; original assignee: VMware Inc.

[125] Harvey Tuch, Prashanth P. Bungale, Scott W. Devine, and Lawrence S. Rogel. Virtualizing processor memory protection with "L1 iterate and L2 swizzle", June 2012. URL `http://www.google.com/patents/US20120151168`. US Patent App. 12/966,782; original assignee: VMware Inc.

[126] Harvey Tuch, Prashanth P. Bungale, Scott W. Devine, and Lawrence S. Rogel. Virtualizing processor memory protection with "domain track", June 2012. URL `http://www.google.com/patents/US20120151117`. US Patent App. 12/966,805; original assignee: VMware Inc.

[127] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5): 48–56, May 2005.

[128] Vojin Živojnović, Stefan Pees, and Heinrich Meyr. LISA – machine description language and generic machine model for HW/SW co-design. In Wayne Burleson, Konstantinos Konstantinides, and Teresa Meng, editors, *Proceedings of the IEEE Workshop on VLSI Signal Processing, IX (San Francisco)*, pages 127–136. IEEE, October 1996. ISBN 0-7803-3134-6.

[129] Peter Van Bouwel. Ondersteuning voor zelf-wijzigende code in een ARM hypervisor. Master's thesis, Ghent University, Faculty of Engineering and Architecture, June 2011.

[130] Jens Van den Broeck. Prestatiemetingen voor systeemsoftware m.b.v. FPGA. Master's thesis, Ghent University, Faculty of Engineering and Architecture, June 2013.

[131] Prashant Varanasi. Implementing hardware-supported virtualization in OKL4 on ARM. Master's thesis, School of Computer Science and Engineering, The University of New South Wales, November 2010.

[132] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (Shangai, China)*, APSys 2011, pages 11:1–11:5, New York, NY, USA, July 2011. ACM.

[133] VirtualLogix. VLX for mobile handsets, 2009. URL `https://web.archive.org/web/20090308134358/http://www.virtuallogix.com/products/vlx-for-mobile-handsets.html`.

[134] VMware Inc. Understanding full virtualization, paravirtualization, and hardware assist. White paper, November 2007.

[135] Jon Watson. VirtualBox: bits and bytes masquerading as machines. *Linux Journal*, 2008, February 2008. ISSN 1075-3583.

[136] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 10–15, New York, NY, USA, 2002. ACM.

[137] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, Seattle, Washington, USA, 2002.

[138] Wookey. The new arm ABI (EABI) and Debian armel port. Talk at Debconf (Edinburgh, Scotland), 2007. URL `http://wookware.org/talks/armeabidebconf.pdf`.

[139] Youfeng Wu, Shiliang Hu, Edson Borin, and Cheng Wang. A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2011, pages 236–245, Piscataway, New Jersey, USA, April 2011. IEEE Computer Society. ISBN 978-1-61284-355-1.

[140] Xen Project. Xen ARM with virtualization extensions whitepaper, April 2014. URL `http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper`.

[141] Xvisor. eXtensible Versatile hypervISOR. URL `http://xhypervisor.org`.