
THE UNIVERSITY *of* EDINBURGH

# Efficient Cross-architecture Hardware Virtualisation

*Tom Spink*


Doctor of Philosophy

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2016

# Abstract

Hardware virtualisation is the provision of an isolated virtual environment that represents real physical hardware. It enables operating systems, or other system-level software (the *guest*), to run unmodified in a "container" (the *virtual machine*) that is isolated from the real machine (the *host*).

There are many use-cases for hardware virtualisation that span a wide-range of end-users. For example, home-users wanting to run multiple operating systems side-by-side (such as running a Windows® operating system inside an OS X environment) will use virtualisation to accomplish this. In research and development environments, developers building experimental software and hardware want to prototype their designs quickly, and so will virtualise the platform they are targeting to isolate it from their development workstation. Large-scale computing environments employ virtualisation to consolidate hardware, enforce application isolation, migrate existing servers or provision new servers. However, the majority of these use-cases call for *same*-architecture virtualisation, where the architecture of the guest and the host machines match—a situation that can be accelerated by the hardware-assisted virtualisation extensions present on modern processors. But, there is significant interest in virtualising the hardware of different architectures on a host machine, especially in the architectural research and development worlds.

Typically, the instruction set architecture of a guest platform will be different to the host machine, e.g. an ARM guest on an x86 host will use an ARM instruction set, whereas the host will be using the x86 instruction set. Therefore, to enable this cross-architecture virtualisation, each guest instruction must be emulated by the host CPU—a potentially costly operation. This thesis presents a range of techniques for accelerating this instruction emulation, improving over a state-of-the art instruction set simulator by $2.64\times$. But, emulation of the guest platform's instruction set is not enough for full hardware virtualisation. In fact, this is just one challenge in a range of issues that must be considered. Specifically, another challenge is efficiently handling the way external interrupts are managed by the virtualisation system. This thesis shows that when employing efficient instruction emulation techniques, it is not feasible to arbitrarily divert control-flow without consideration being given to the state of the emulated processor. Furthermore, it is shown that it is possible for the virtualisation

environment to behave incorrectly if particular care is not given to the point at which control-flow is allowed to diverge. To solve this, a technique is developed that maintains efficient instruction emulation, and correctly handles external interrupt sources.

Finally, modern processors have built-in support for hardware virtualisation in the form of instruction set extensions that enable the creation of an abstract computing environment, indistinguishable from real hardware. These extensions enable guest operating systems to run directly on the physical processor, with minimal supervision from a hypervisor. However, these extensions are geared towards same-architecture virtualisation, and as such are not immediately well-suited for cross-architecture virtualisation. This thesis presents a technique for exploiting these existing extensions, and using them in a cross-architecture virtualisation setting, improving the performance of a novel cross-architecture virtualisation hypervisor over state-of-the-art by $2.5\times$.

# Lay Summary

Processors are at the centre of any computer system, and they can be found in surprising places. Laptops, smart phones, fridges, toasters, televisions and ovens are all examples of where computer processors can be found in the modern world. A significant problem is that these processors all need to be designed and tested by someone, but how can you test the design for a processor that has not been created yet? The answer to this is to simulate the processor, and realistically, the simulator should be fast.

The easiest way to simulate a new processor is to make a computer program that pretends to be this new processor, and runs it step-by-step. But, this kind of approach to simulation is not very fast, so the underlying goal of this thesis is to speed it up. A standard technique to improve this is to convert a whole sequence of individual steps into one larger (but more efficient) step. However, this technique can be implemented in a number of ways, and the first key idea is to look at the connections between the steps in more detail, to make jumping between them more efficient.

If you want to simulate an entire computer system, however, this approach is still not good enough, because there are a lot more things to consider. For example, when you use a keyboard, it tells the processor to stop what it is doing, and look at the key that was pressed. This kind of behaviour also slows down simulators, so another idea presented is a fast means of handling this.

Finally, the key idea presented at the end of this thesis is that instead of writing a program that pretends to be a processor, you can take the similarities between a real processor and the simulated processor, and use this to speed up the simulation.

# Acknowledgements

There are a range of people that I would like to acknowledge for their support during my time as a PhD student, and I would like to begin by thanking the residents of office 1.34, who made my time at the University of Edinburgh thoroughly enjoyable, and were an excellent forum for ideas. Specifically, I would like to extend my thanks to Oscar Almer, Matthew Bielby, Bruno Bodin, Tobias Edler von Koch, Stephen Kyle and Volker Seeker for their excellent company and support. Deserving a special mention is my friend Harry Wagstaff, whom I have collaborated with over the past few years.

I would also like to extend my deepest thanks and gratitude to Björn Franke, who has been a fantastic supervisor—his guidance has been invaluable and his support and encouragement has made this research a thoroughly enjoyable experience.

And finally, I would like to thank my family, and especially my wife Jennifer, for the encouragement, the support and the patience they have continually shown me throughout this escapade.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Tom Spink*)

# Publications

The following publications have been made during the course of this PhD, some of which are used as the basis for chapters:

- **Tom Spink**, Harry Wagstaff and Björn Franke
  "Hardware Accelerated Cross-Architecture Full-System Virtualization"
  *In ACM Transactions on Architecture and Code Optimization (TACO) 13, 4, Article 36, October 2016*
  — **This publication forms the basis of Chapter 6**

- **Tom Spink**, Harry Wagstaff and Björn Franke
  "Efficient Asynchronous Interrupt Handling in a Full-system Instruction Set Simulator"
  *In Proceedings of the 2016 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES'16), Santa Barbara, CA, USA, June 2016.*
  — **This publication forms the basis of Chapter 5**

- **Tom Spink**, Harry Wagstaff, Björn Franke and Nigel Topham
  "Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator"
  *In Proceedings of the 2015 International Conference on Embedded Computer Systems, Architectures, Modeling and Simulation (SAMOS'15), Samos Island, Greece, July 2015.*

- **Tom Spink**, Harry Wagstaff, Björn Franke and Nigel Topham
  "Efficient code generation in a region-based dynamic binary translator."
  *In Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES'14), Edinburgh, UK, June 2014.*
  — **This publication forms the basis of Chapter 4**

- Harry Wagstaff, **Tom Spink** and Björn Franke
  "Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators"
  *In Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'14), New Delhi, June 2014.*

# Table of Contents

# List of Figures

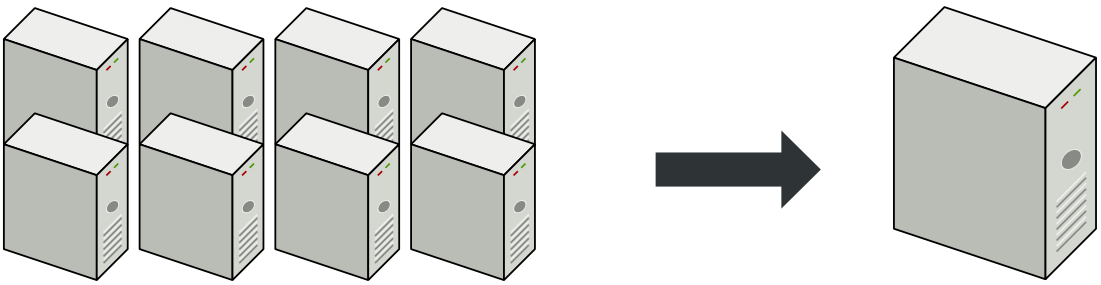
# List of Tables

Chapter 1

# Introduction

There are many uses for hardware virtualisation in today's modern computing environments. Data centres wanting to lower their hardware costs and increase resource utilisation will look to virtualisation as a way to consolidate servers [119], users wanting the convenience of running multiple operating systems on their computers without the inconvenience of rebooting will often use virtualisation to run two (or more) operating systems side-by-side [47] and hardware and software developers wanting to prototype, debug and benchmark their designs will use virtualisation to create an isolated test environment [64, 112] that represents the target platform.

Hardware virtualisation is the process of creating a virtual representation of a particular hardware platform, providing an environment that appears to be a separate physical machine. Virtualising an entire platform can be quite straightforward when the *guest* machine is of the same architecture as the *host* machine—modern processor vendors provide hardware support [65, 4] out-of-the-box for this, allowing unmodified guest operating systems to run in virtual machines at near-native speeds [117]. But, when the need arises for *cross*-architecture virtualisation, there is no longer a one-to-one mapping of architectural components from the guest to the host, and these mismatched components need to be emulated in software.



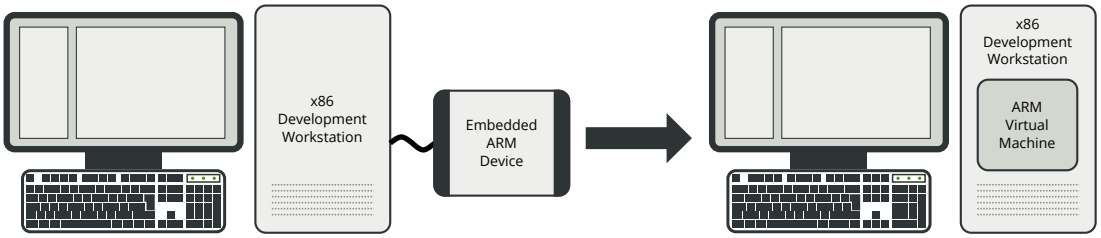



**Figure 1.1:** Instead of debugging applications on real hardware, virtualisation enables creating a virtual version of the hardware on which application development and testing can occur.

There is quite a large market for cross-architecture hardware virtualisation. It is used extensively during hardware development for rapidly prototyping platforms that may not yet exist, or to make modifications to an existing platform to observe how it may affect applications. Imperas [64] produce a suite of tools for developers that enable unmodified guest operating systems to run in a so-called *virtual platform*. This allows developers to boot an entire operating system compiled for a different architecture, in a virtual machine on their development workstation.

Software engineers use cross-architecture virtualisation to obtain a virtual representation of the platform their applications are targeting, allowing them to rapidly test and debug applications on their development workstation, without having to deploy it to a real device. Synopsys® produce tools under their *Virtual Prototyping* [112] offering that enables simulation of hardware platforms still under development, so that developers can begin producing applications for a platform that has not yet been materialised. Similarly, ARM® produce a configurable simulation tool called *Fast Models* [12] that enables developers to construct a virtual platform out of multiple architectural building blocks.

One of the most widely used [8] cross-architecture virtualisation systems is the Android® Emulator, which enables software developers to test their applications in the context of an unmodified ARM® Android® environment (Figure 1.1). This is important when developing native applications designed to run directly on ARM® processors, as it is time consuming and costly to continuously deploy to a real device for testing.

In order to support cross-architecture hardware virtualisation, it is necessary to emulate the behaviour of the hardware of the target platform on the host. This involves providing faithful emulation of guest instructions, and software

implementations of architectural components (such as the *memory management unit* (MMU) and interrupt controller). It also involves emulating platform devices, such as disk or network devices. If an unmodified guest operating system is to be booted in this *virtual* environment, these components must all behave exactly as they would in a *physical* environment.

Virtualisation requires support from a *virtual machine monitor* (VMM) or *hypervisor*, which allocates and manages physical resources for the virtual guest. There are many open-source and commercial hypervisors available for *same*-architecture virtualisation, and the wide availability of hardware-accelerated extensions for virtualisation makes it a viable technology for production use. However, the majority of hypervisors that support *cross*-architecture virtualisation are for research purposes or detailed simulation—the most notable exception being QEMU [21].

## 1.1 Background

Support for hardware virtualisation appeared in systems as early as 1972 (on the IBM System/370 mainframe), and was used as a method of partitioning the physical machine into multiple virtual machines that appeared to users as their own private system. But, towards the end of the 1970s, virtualisation lost traction due to a significant increase in computing power on commodity processors (coupled with more modern forms of process isolation) effectively negating the need for it at that time. Thus, virtualisation became a software problem, and in fact software solutions for virtualisation of x86 processors outperformed the initial hardware support provided by Intel® and AMD® in the early 2000s. A resurgence of interest in the virtualisation space has led to improvements in hardware support, and now modern processor vendors provide *instruction set extensions* (ISEs) that can be used to present an abstract computing platform, allowing unmodified guest operating systems of the same architecture to run virtualised at near-native speeds—an important property for businesses wanting to deploy virtualised systems.

The requirements for hardware virtualisation were formalised in 1974 by Popek and Goldberg [101], who identified two types of hypervisor (shown in Figure 1.2) and made three key insights about the operation of virtual machines:

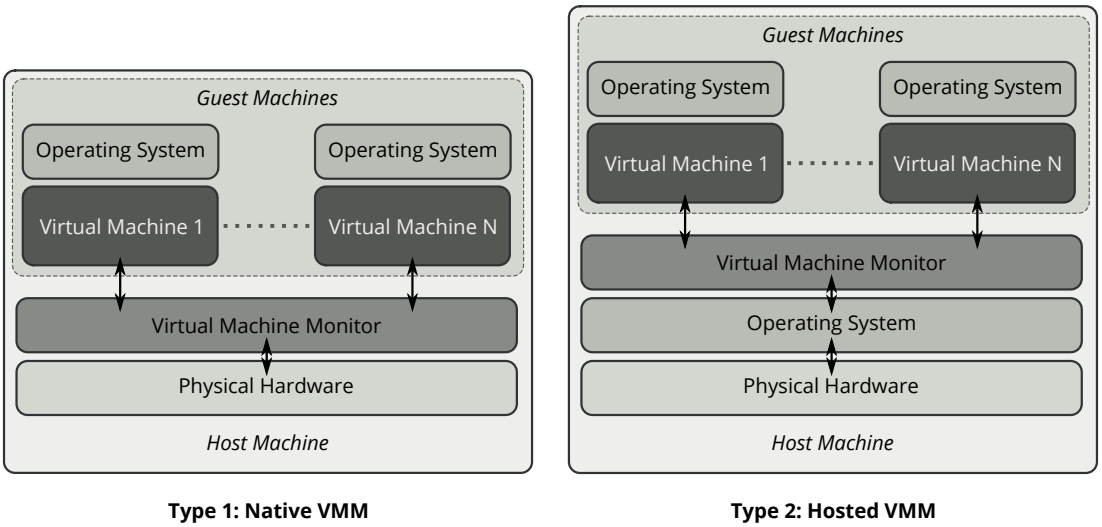



**Figure 1.2:** Popek and Goldberg identified two different types of *virtual machine monitor*. A **type 1** or **native** VMM runs directly on the host hardware, whereas a **type 2** or **hosted** VMM runs within the confines of an operating system.

1. A virtual machine must not exhibit a difference in behaviour to the physical machine it is modelling.
2. A virtual machine must be fast and efficient.
3. The *virtual machine monitor* (VMM) or *hypervisor* must remain in control of the physical machine's resources.

Whilst the authors applied these characteristics to virtual machines in general (without specifically targeting the same-architecture or cross-architecture use-case), the consequence of (2) is that software-based emulators and simulators were excluded from being classified as VMMs, since at the time of publication they could not satisfy the "efficiency" requirement. Unfortunately, this is a problem for cross-architecture virtualisation, as a software-based *instruction set simulator* (ISS) is *necessary* for performing the emulation of guest instructions. This is because the guest platform has a different *instruction set architecture* (ISA) to the host, and therefore guest instructions cannot execute natively on the host's physical processor. However, this observation assumed a slow interpreter-based ISS was used to emulate guest instructions, and so software emulation was discounted as a viable VMM for this reason.

However, more recent improvements to *dynamic binary translation* (DBT) have made software-based virtualisation systems more competitive with—and

in some cases more efficient than—hardware-assisted virtualisation. This means that a suitably engineered software-based virtualisation system can meet the requirements for hardware virtualisation, and hence cross-architecture virtualisation can be considered a viable form of virtualisation.

There are four major challenges that need to be addressed when developing a cross-architecture hardware virtualisation hypervisor, each with their own impact on the performance and correctness of such a system:

1. **Instruction emulation:** The faithful and efficient execution of guest machine instructions.
2. **Interrupt handling:** The timely, and correct, handling of external interrupts, by altering control-flow (e.g. to interrupt handlers) as required.
3. **Memory management unit virtualisation:** Performing efficient memory address translation for guest architectures with an MMU, or access permission checking for those with an MPU.
4. **Device emulation:** Providing implementations of devices that may exist on the platform.

Each of these virtualisation challenges shall be visited over the course of this thesis, and techniques to improve the efficiency of their implementation shall be presented.

## 1.2 Motivation

It is clear to see that cross-architecture virtualisation is a desirable technology across a range of disciplines, and it follows that due to the necessity of emulating architectural components in software, there is an unavoidable performance penalty. As mentioned previously, in the same-architecture case, modern processors from a range of vendors (Intel®, AMD®, ARM®, MIPS®, PowerPC®) provide hardware accelerated support for virtualisation, enabling unmodified guest operating systems to run natively on the host machine with very little supervision. This is because architectural behaviour is the same between guest and host, and platform devices can (if desired) be passed straight through to the guest. But, when the architecture of the guest is different to the host, there is no longer the possibility of mapping guest platform behaviour directly to host

platform behaviour, and the mismatched components of the guest system must be emulated.

Consider the use of the Android Emulator in a development environment. Developers want to debug and test their applications locally without having to deploy to real devices constantly. Deploying an application to a device can be time consuming, but so can using a slow emulator. In fact, Intel have recognised this particular issue and developed their own technique for improving the performance of the Android Emulator [109]. Whilst their technique uses hardware acceleration for virtualisation, it relies on an x86 version of Android, and so is not a cross-architecture solution. Therefore, this does not solve the problem of efficiently developing and debugging native ARM applications in an emulator. Furthermore, this is application specific, and does not solve the general problem of efficiently virtualising platforms that may not even exist yet.

Many architecture design companies that provide tools (such as compilers) for their architectures supply simulators as a basic tool. For example, ARM provide a development suite called DS-5 Development Studio that is available with a technology called *Fast Models*. This tool can be user-configured to virtualise an ARM platform, and can reach speeds that are close to native platform speeds. However, when additional components are enabled, the system quickly loses traction as its implementation is based on an event-driven framework. Synopsys provide a virtualisation tool called *Virtual Prototyping*, used to aid development on their own platforms.

QEMU [21] is a popular open-source full-system virtualisation hypervisor that supports a wide range of guest machine architectures, and is used throughout academia and the software/hardware development industry. In fact, it forms the basis of the Android Emulator as supplied as part of the Android *software development kit* (SDK). Out-of-the-box, QEMU supports many different guest architectures, but the software itself is not easily retargetable. Porting QEMU to another architecture requires manually coding an instruction decoder and building a translation routine that uses the internal *tiny code generator* (TCG) DBT to translate decoded instructions into QEMU's own intermediate representation.

### Goal

The goal of this thesis is to develop techniques that can be used by hypervisors for fast and efficient cross-architecture hardware virtualisation, with an additional focus on ease of use and retargetability.

## 1.3 Overview & Contributions

Chapter 2 shall introduce terminology associated with the virtualisation of computer systems, along with existing techniques for both same-architecture and cross-architecture virtualisation. These techniques shall be accompanied by related work in the area, relevant to the challenges that are being tackled in later chapters.

Following this, in Chapter 3, an introduction to the software tools developed and extended as part of this research shall be presented, along with descriptions of the frameworks used and the general methodology employed for performance evaluation.

Chapter 4 will tackle the initial challenge of efficient guest instruction emulation, by seeking to introduce techniques to increase the performance of a software-based *instruction set simulator* (ISS). This introduces fundamental and significant improvements to *dynamic binary translation* (DBT), which are necessary for efficient emulation of guest instructions.

In Chapter 5, the challenges associated with extending this ISS to support hardware virtualisation will be investigated, by focussing on asynchronous interrupts that are necessary for emulating a guest platform, and cause performance regressions in DBT-based virtualisation systems.

Chapter 6 exploits hardware-assisted virtualisation technology that is present on modern processors to develop a novel hypervisor for fast and efficient cross-architecture virtualisation. The four major factors for cross-architecture virtualisation are considered and novel techniques for accelerating each of these are presented.

Finally, Chapter 7 will summarise, conclude and present future work in the field of efficient hardware virtualisation.

Chapter 2

# Background & Related Work

This chapter shall first define terminology that will be used throughout the remainder of this thesis, followed by a brief introduction to some important concepts involved in cross-architecture hardware virtualisation. These concepts shall be supplemented with related work in the area of hypervisor performance and implementation strategies.

## 2.1 Terminology

There is a range of terminology in use when discussing virtualisation of computer systems, and this section shall define and describe the terms that will be used throughout the remainder of this thesis. The majority of these terms are widely used in literature, but due to the complex, multi-layered and potentially confusing nature of hardware virtualisation systems, it is important to define at this point how the particular term is intended to be perceived.

### 2.1.1 Overview

The three main terms that are used throughout this area of research are:

- **Virtualisation:** The provision of a *virtual* version of an existing *physical* component.
- **Emulation:** The imitation of the behaviour of a particular component.
- **Simulation:** The emulation of a particular component, but coupled with the ability to instrument and inspect the behaviour of that component.

This thesis is primarily concerned with *virtualisation*, but *emulation* of some form is a necessity for cross-architecture virtualisation. *Simulation* is an overlapping area, and is an important technique for debugging and monitoring purposes, however this thesis is not concerned with precise simulation, but shall introduce it as future work in Section 7.3.

### 2.1.2 Definitions

**Definition 1** (Architecture)**.** *The architecture of a computer system describes how the system functions, and how it is organised. Typically it also defines the instruction set architecture (ISA), which describes what instructions are available, how they are encoded, and how they behave.*

**Definition 2** (Platform)**.** *The platform is how a particular computer system is configured, i.e. which architecture it is built upon, what features are employed from that architecture, what type of processor(s) are in use, what devices are available and generally how the system "fits together".*

**Definition 3** (Virtual Machine)**.** *A virtual machine (VM) is an isolated, virtual representation of a computer system.*

**Definition 4** (Host Machine)**.** *The host machine is the **physical** computer, on which a virtual machine is intended to be created.*

**Definition 5** (Guest Machine)**.** *The guest machine is the particular computer system that is being virtualised and being represented by a virtual machine.*

**Definition 6** (User-mode Simulation)**.** *User-mode simulation, as shall be described in Section 2.5, is the act of simulating a computer program that is designed to be run on a particular architecture, on a different architecture (although the architectures could be the same). This kind of simulation is limited to single programs running inside an operating system, and requires the simulator to emulate the behaviour of the original guest operating system.*

**Definition 7** (Full-system Simulation)**.** *Full-system simulation is the act of simulating an entire computer system. This term can be synonymous with hardware virtualisation but as mentioned previously, simulation implies that some form of instrumentation or inspection is involved to gain insight into the behaviour of the system.*

**Definition 8** (Hardware Virtualisation)**.** *As described in Section 2.6, and as will be presented throughout the remainder of this thesis, hardware virtualisation is the act of providing a virtual machine that represents a real physical machine—including all of the architectural behaviour and hardware devices present on the platform being virtualised.*

**Definition 9** (Same-architecture Virtualisation)**.** *Virtualisation when the guest machine and the host machine are of the* ***same*** *architecture.*

**Definition 10** (Cross-architecture Virtualisation)**.** *Virtualisation when the guest machine and the host machine are of* ***different*** *architectures.*

**Definition 11** (Hardware-assisted Virtualisation)**.** *Not to be confused with hardware virtualisation,* ***hardware-assisted*** *virtualisation is when the host machine architecture provides additional support for performing hardware virtualisation, e.g. Intel VT [65] or AMD-V [4].*

**Definition 12** (Hypervisor)**.** *Sometimes termed a virtual machine monitor, a hypervisor is a piece of software that is responsible for managing the lifecycle of a virtual machine. Normally, the hypervisor creates and starts the VM, along with allocating and managing host machine resources (such as memory) that will be virtualised.*

Furthermore, there are two types of hypervisor that are relevant to hardware virtualisation:

**Type 1 (native):** A hypervisor that runs directly on the host machine hardware.

**Type 2 (hosted):** A hypervisor that runs inside a normal operating system.

**Definition 13** (Instruction Emulation)**.** *Instruction emulation is the act of executing a guest machine instruction, on the host machine, causing the state of the virtual machine to change as it would if the instruction was executed on a real guest machine.*

**Definition 14** (Device Emulation)**.** *Device emulation is generally a software implementation of a real hardware device that emulates the behaviour of that device when the virtual machine accesses it.*

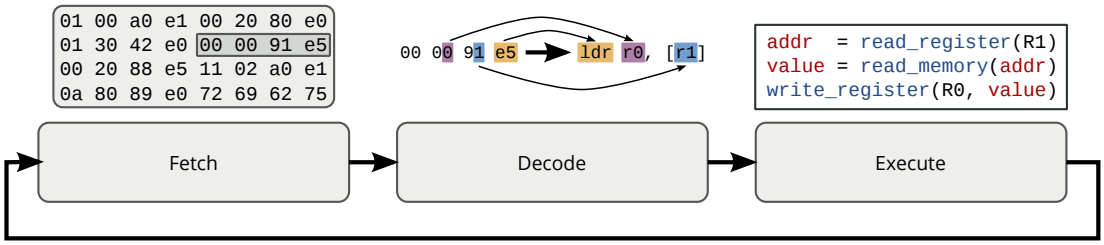



**Figure 2.1:** A typical *fetch*, *decode*, *execute* sequence.

## 2.2 Instruction Emulation

Any form of cross-architecture simulation or virtualisation requires the emulation of guest machine instructions, as these instructions cannot directly execute on the host machine. Typically, a guest machine will be in a particular *state*, and executing an instruction causes that state to change. The *behaviour* of a particular instruction is defined by the *instruction set architecture* (ISA).

From a functional perspective, processors execute instructions by **fetching** an instruction from memory (pointed to by the *program counter* (PC) register), **decoding** it into its constituent fields, and then **executing** the associated behaviour. This is shown in Figure 2.1. In reality, the situation is much more complex, with e.g. superscalar processors, pipelines, out-of-order execution, speculation, branch prediction and etc. all contributing to a highly complex execution model. However, this high-level sequence is typically the basis for the variety of execution models available to cross-architecture virtualisation systems.

There are various approaches to implementing the execution model in a virtualisation system, each with their own benefits and drawbacks. The two most common approaches are *interpretation* and *dynamic binary translation*, and these have further possible implementation choices. An overview of each of these approaches shall be given in the following sections, along with associated related work in the area.

### 2.2.1 Interpretation

An interpreter is effectively the implementation of the fetch-decode-execute cycle described above. It has the benefit of being a very straightforward approach to instruction emulation, but suffers from performance limitations.

Listing 2.1 shows a typical loop-based interpreter, where each instruction

**Listing 2.1:** Interpreter Loop

```
do {
  insn = FETCH();
  opcode = DECODE();

  switch (opcode) {
  case OPCODE_ADD:
    <behaviour>
    break;
  case OPCODE_SUB:
    <behaviour>
    break;
  }
} while (true)
```

**Listing 2.2:** Threaded Interpreter

```
jump_table = [&OPCODE_ADD, &OPCODE_SUB];

insn = FETCH();
opcode = DECODE();
goto &jump_table[opcode];

OPCODE_ADD:
  <behaviour>
  insn = FETCH();
  opcode = DECODE();
  goto &jump_table[opcode];

OPCODE_SUB:
  <behaviour>
  insn = FETCH();
  opcode = DECODE();
  goto &jump_table[opcode];
```

**Figure 2.2:** A typical interpreter-based virtualisation system. Listing 2.1 shows a loop-based interpreter, and Listing 2.2 shows a more efficient *threaded* implementation.

is fetched from the guest machine's memory, decoded, then a branch is made to the behaviour for that instruction. After the behaviour completes, the interpreter loops around and starts again. Even if the individual instruction behaviours are optimised aggressively, the rate at which instructions execute is effectively fixed.

A more efficient approach is to dispatch to the behaviour for the next instruction, immediately after the current instruction has finished executing, instead of looping around. This type of implementation is called a *threaded* interpreter, and is shown in Listing 2.2. Instead of executing in a loop, control-flow threads from one instruction behaviour to the next, by dispatching to instruction behaviour via a *jump table*.

A further optimisation that can be made is to introduce a decode cache, eliminating the cost of decoding guest instructions, if the instruction has been recently seen. This is important for looping control-flow in the guest, where the same instructions will be executed many times in quick succession.

Even if control-flow is improved, and decode caches are used, an interpreter will always reach a performance ceiling. This is because this method of execution considers each instruction individually, and invokes a distinct emulation for each instruction type.

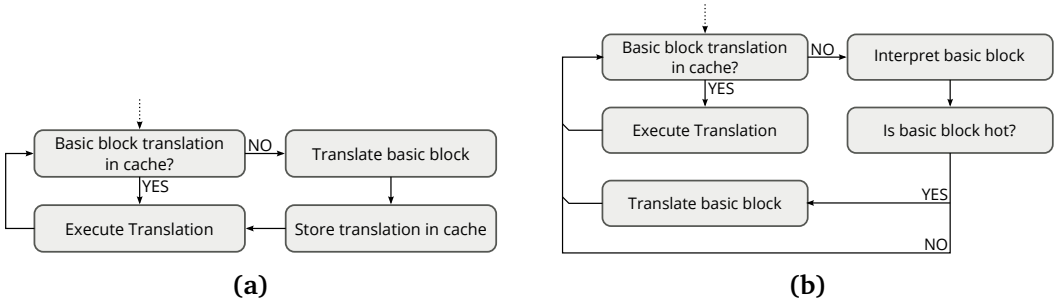



**Figure 2.3:** Two examples of possible dynamic binary translation implementations. (a) shows a synchronous DBT that translates guest basic blocks on demand. (b) shows a DBT that initially executes guest basic blocks in an interpreter, until they become hot. At this point, the basic block is translated.

### 2.2.2 Dynamic Binary Translation

Dynamic binary translation is an execution model that translates guest machine instructions into corresponding host machine instructions as the virtual machine is running. This technique opens up the scope for executing many guest instructions as a unit, because (unlike an interpreter) it is no longer constrained to operating on an instruction-by-instruction basis.

For example, a typical unit of translation in a DBT system is a basic block, where a basic block is a *straight-line*, *single-entry*, *single-exit* region of instructions. A DBT will decode each instruction in a guest basic block, and produce host machine code that represents the behaviour of that entire instruction sequence. There is not necessarily a one-to-one mapping between guest basic blocks and host basic blocks, as some instructions may raise exceptions (that require early exiting from the block) or the guest architecture may support *predicated* instructions (that only execute if certain flags are set).

Figure 2.3 shows two examples of possible DBT implementations. Figure 2.3a shows a synchronous block-based DBT, where the DBT will translate guest basic blocks on-demand, i.e. when a translation does not exist. Figure 2.3b shows a DBT/interpreter hybrid, where execution will initially proceed through an interpreter, until a block becomes "hot". At this point, the block will be translated, and execution will transition to native code, until a translation does not exist.

A modification may be made to the hybrid approach, by turning the translation of hot blocks into an *asynchronous* operation [24], meaning that execution

of the guest system can progress (in the interpreter) whilst translations are being performed in the background, hiding the compilation latency.

### 2.2.3 Translation Granularity

An important detail to consider when developing a DBT system is the *granularity* of the translation, or what comprises a translation unit. For example, is the translation performed instruction-by-instruction, or are entire basic blocks translated?

Typically, the more guest instructions that are considered in a translation unit, the more efficient the translated code will be. This is because optimisations can be applied across the translation unit as a whole, leading to highly efficient host machine code. The trade-off, however, is between translated code quality and compilation latency. Spending more time translating code (or simply translating more instructions at-a-time) results in a longer compilation time, impacting on the overall throughput of the system. However, the performance gains of DBT greatly outweigh the added latency of a translation phase, when compared to an interpreter-based system.

Translation granularity can be broadly classified into four different schemes:

- **Instruction:** A single guest instruction is translated into one or (usually) more host instructions. This is no better than an interpreter, however, and would in fact perform much worse due to the added translation penalty.
- **Basic Block:** A straight-line, *single-entry, single-exit* region of guest instructions are translated into multiple host instructions.
- **Linear Trace:** A sequence of guest basic blocks that only jump forward (i.e. there are no loops) are translated.
- **Region:** A *multi-entry, multi-exit* region of guest instructions, possibly comprising cyclic control flow, is translated to corresponding host machine code.

A region-based DBT offers the most scope for generating high-quality native code, but it requires dynamic profiling during the application's run in order to *form* regions and determine which discovered basic blocks are worth translating. Region forming is the process of determining which basic blocks should be

logically considered part of a particular region, and various schemes have been proposed for this purpose [26, 55, 57, 62]. An asynchronous form of this style of DBT will be discussed further in Chapter 4.

### 2.2.4 Region-based DBT Systems

Region-based JIT compilation has been used for some time in Java virtual machines, e.g. Suganuma et al. [110, 111], but has only been considered more recently for DBT systems [69, 24, 72]. The reason for this late adoption of region based policies has been presumably the increased latency for compilation and optimisation of larger regions, which has only been addressed recently with the introduction of decoupled, latency-hiding JIT compilation task farms [24]. The bulk of the work in this field has focussed on region selection though, and less on code generation and optimisation for dynamically discovered regions. In Jones and Topham [69] large translations units (i.e. regions) are introduced for dynamic binary translation, and region selection policies based on strongly connected components, control flow graph fragments and OS pages are compared. A refined page based region selection scheme is developed in Böhm et al. [24] and combined with a parallel JIT compilation task farm. Specific optimisations for a DBT system, which compiles guest- to host code via *Java Virtual Machine* (JVM) bytecode, are considered in Kaufmann and Spallek [72].

### 2.2.5 Code Generation and Optimisation in DBT Systems

Most DBT systems appear to have adopted a code generation strategy operating on individual basic blocks or linear traces of basic blocks. For example, QEMU [21] implements such an approach using its own *tiny code generator* (TCG) and additional block chaining, translation caching and lazy condition evaluation.

Dynamo [17] is a dynamic optimisation system, i.e. the input is a native instruction stream. Dynamo uses an interpreter for initial execution until a “hot” instruction sequence is identified. At that point, Dynamo generates an optimised version of the trace into a software code cache. Dynamo treats backward branches as trace delimiters, i.e. traces are by definition linear. After translation it emits an optimised single-entry, multi-exit, contiguous sequence of instructions for each trace. Trace optimisation in Dynamo considers branch types,

but is generally less aggressive than what can be achieved when considering a region that contains cyclic control-flow.

DynamoRio [27] is a successor of Dynamo. DynamoRio operates on two kinds of code sequences: basic blocks and traces. Both have linear control flow, with a single entrance and potentially multiple exits, but no internal join points. Optimisations are restricted to the linear control flow present in traces. The single-entry multiple-exit format simplifies analysis algorithms, but limits the scope of optimisations that can be applied.

Strata [56] is a retargetable DBT system offering additional uses for dynamic instrumentation and optimisation. Different fragment selection policies [57] have been evaluated for Strata, but all of these policies are based on linear traces, possibly spanning branch or function call boundaries. Strata uses chaining of traces to avoid overheads associated with returning to the main execution loop after every native trace. An ARM port of Strata considers architecture-specific optimisations, e.g. relating to the exposed PC [92].

The optimisations performed by UQDBT – a machine-adaptable dynamic binary translator – are discussed in Cifuentes and Emmerik [35], Ung and Cifuentes [116]. This tool uses an algorithm for finding hot paths using edge weight profiles, and optimises code in a machine-independent way, based on hot path information. Whilst units of translation in UQDBT are basic blocks, for its hot path (re)optimisation it groups hot basic blocks and their connecting control flow edges into regions. The paper focuses primarily on newly discovered hot paths and locality transformations, but does not provide a complete code generation strategy. A particular aspect of code generation in DBT systems, namely recovery of jump table case statements, is discussed in Cifuentes and Emmerik [34].

Rosetta [2] is a DBT that translates PowerPC G3, G4 and AltiVec instructions to x86 instructions. It is based on QuickTransit by Transitive [1], and was released by Apple in 2006, after the ISA of their Macintosh platform was changed from PowerPC to x86. Rosetta is a user-mode DBT, as its primary purpose was to allow legacy PowerPC-based Macintosh applications to run on modern Intel-based Macintosh computers.

Liu et al. [82] introduce a translation system based on “hybrid binary translation”, which involves an offline *static* binary translation phase and falling back to a run-time *dynamic* binary translation system to handle untranslated code.

### 2.2.6 DBT Systems using LLVM for JIT Compilation

LLVM [77] is a popular open-source compilation framework, that can be employed as a JIT compiler for DBT. It contains a wide range of high-quality optimisation passes that lead to the production of highly efficient machine code.

A parallel and concurrent JIT compilation task farm for use in DBT systems is presented by Böhm et al. [24]. The JIT compiler is based on the LLVM framework, which is used for translation of paged regions of target instructions to host instructions. The paper discusses a particular region selection scheme and parallel JIT compilation, but provides no details of the actual code generation approach used.

LnQ [61] extends QEMU with an LLVM-based JIT compiler, but does not consider code regions for translation, instead it uses linear traces.

HQEMU [59] is a multi-threaded dynamic binary translator, which extends QEMU with multiple instances of the LLVM compiler for JIT compilation. HQEMU builds on top of LLVM, but it only operates on linear traces and does not support region-based compilation.

Guo et al. [50] look at a particular DBT challenge, which is mapping the behaviour of guest machine vector instructions onto host machine vector instructions. Specifically, they look at optimising the dynamic translation of ARM Neon and *vector floating point* (VFP) into corresponding host machine instructions. Their approach is to generate LLVM bytecode that closely models the vector-specific behaviour of the guest instruction, which is highly amenable to lowering into host machine vector instructions.

## 2.3 Interrupt Handling

Virtualising an entire computer system means honouring the multitude of architectural behaviours that exist on the target platform. A particular challenge for virtualisation is the efficient handling of *asynchronous* interrupts, i.e. those interrupts that are raised by external signals (such as devices), and not related to the directly executing instruction.

In order to maintain consistency, virtualisation systems can only handle interrupts at well-defined points during execution, which at a minimum is an instruction boundary. Diverting control-flow during the execution of an instruc-

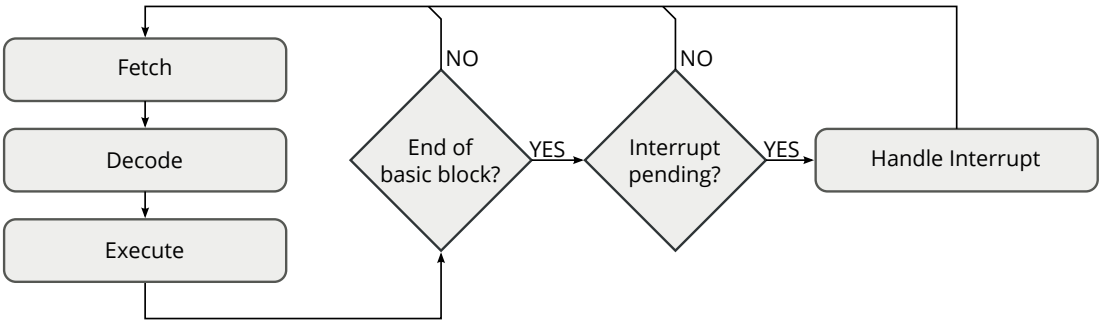



**Figure 2.4:** An illustration of how interrupt checking might work in an interpreter-based system. In this example, a pending interrupt is checked for after the interpreter has executed one basic block of guest instructions.

tion would lead to corruption of guest machine state, as guest instructions must appear to execute atomically.

As shown in Figure 2.4, in an interpreter based system, interrupt checking is very straightforward—a check can be made after each instruction, or after a certain number of instructions (which in the example is a basic block). However, for DBT-based systems, diverting control-flow to an interrupt handler, as specified by the architectural behaviour, requires adding interrupt checks to the translated code, ensuring any pending interrupts are handled. As shall be described in Chapter 5, it is not possible to arbitrarily place these interrupt checks without considering the effect they have on the behaviour and performance of the virtual machine.

The optimal placement of interrupt checks can be compared to the optimal insertion of profiling counters. However, updating profiling counters does not introduce additional control-flow—since the majority of cases are simple counter updates. Whilst reducing the number of counter updates can lead to performance improvements by reducing the amount of memory accesses, the problem for DBT is slightly different in that extra control-flow must be added to perform interrupt checks, thus causing additional latency in the optimiser, and resulting in less optimal code being generated. The technique described by Ball and Larus [18] addresses the optimal placing problem, but does not address the issues that are encountered with additional exit points being introduced.

Whilst there are a number of full-system simulators available, either open-source (e.g. QEMU [21], ARM-Iss [84] or MARSSx86 [97]) or under a commercial license (e.g. Simics [87]), only a few papers on interrupt handling in DBT

systems have been published [25].

Older versions of QEMU utilised a zero-overhead interrupt checking scheme, which suffered from serious race-conditions. However, later versions have addressed these issues by inserting checks at the head of every basic block.

ARM-Iss [84] is an instruction set simulator for the ARM architecture. It is based on an interpretive execution model with additional instruction caching. ARM-Iss checks for pending interrupts after each instruction. Whilst accurate, this further exacerbates the performance penalty of a DBT-based system.

MARSSx86 [97] is a full-system simulator for x86 CPUs. Under the hood, MARSSx86 uses QEMU for functional simulation and PTLsim for cycle-accurate modelling, using decomposition of x86 instructions into RISC-like $\mu$-ops and using basic block buffers to form traces of x86 $\mu$-ops. MARSSx86 delays the interrupt issued to the CPU until the CPU comes into the stable state, defined at opcode commit boundaries. Once the interrupts are issued to the CPU MARSSx86 switches from detailed simulation to functional emulation for correctly decoding the interrupt. The emulator mode sets up the correct CPU context to handle the interrupt but it does not start executing the interrupt handler. After the correct CPU context is set up, MARSSx86 switches back to the detailed simulation and starts simulating the interrupt handler code in kernel mode. Due to its cycle-accurate approach interrupt handling in MARSSx86 is precise, but it only operates at a throughput of about 200 kilo instruction commits per second (KIPS).

An improved mechanism for the precise simulation of interrupts in cycle-accurate simulators has been presented by Brandner [25]. The simulator speculatively executes instructions of the emulated processor assuming that no interrupts will occur. At restore-points this assumption is verified and the processor state reverted to an earlier restore-point if an interrupt did actually occur. Whilst effective at speeding up cycle-accurate simulation this is still too costly for high-speed functional ISS.

A software simulator based on COTSon [7] that faithfully simulates x86 hardware at a speed in the tens of MIPS range has been described by Ryckbosch et al. [104]. Details on interrupt handling are not provided, though. Similarly, the strategies for interrupt checking are not further specified for Giano [45], SimFlex [52] or Graphite [91]. Gem5 [22] performs per-instruction interrupt checking due to its ambition to support cycle-accurate simulation.

### 2.3.1 Virtual Machines

Somewhat related to interrupt checking in an ISS is exception handling in a Java VM. Java exceptions are *synchronous*, though, i.e. they are related to the currently executing instruction and not triggered externally. Two techniques for dealing with Java exceptions during JIT compilation, namely *on-demand translation* of exception handlers and *exception handler prediction* are presented by Lee et al. [78].

A notable exception are *yield points* in the JikesRVM [68] Java VM, where interrupt checks are inserted in method prologues and epilogues, and on back-edges. These checks are inserted to facilitate user-space scheduling of Java threads, but have been deprecated (as of version 3.1.0) in favour of native threading. JikesRVM inserts a yield point in a method prologue and epilogue, and on a control-transfer instruction (such as an `if`) when the target is backwards.

## 2.4 MMU Virtualisation

Virtualisation of the guest system's *memory management unit* (MMU) is arguably one of the most challenging parts of cross-architecture virtualisation. Memory accesses in a target program occur frequently, and so an inefficient implementation of the MMU will lead to severe performance penalties. MMU virtualisation involves translating the address of every memory access from a virtual address into a physical address, along with checking the permissions of the translation to see if the currently executing code is permitted to perform the particular operation.

On real hardware, these translations are defined by *page tables*, which map *pages* of virtual memory onto *pages* of physical memory and define flags that specify access permissions. Generally, an operating system will create a *virtual memory area* (VMA) for each process, and apply the necessary protection flags, for example, to ensure user code cannot interfere with kernel data structures.

For same-architecture virtualisation, modern processor vendors have recognised the performance penalty that emulating an MMU introduces, and have designed hardware support for accelerating virtualised MMUs. This hardware support is termed *second-level address translation* (SLAT), and examples of this

are Intel's *Extended Page Tables* (EPT) and AMD's *Rapid Virtualization Indexing* (RVI).

However, for cross-architecture virtualisation, all of the approaches to MMU virtualisation are software-based, employing techniques such as caching, exploitation of host memory protection features and shadow page tables in an attempt to accelerate costly memory address translations. Most of the work on accelerating virtualised MMUs is based on QEMU and aims to improve over its default software MMU implementation and caching strategy.

Early work in the context of Simics [86] introduced a software caching mechanism, which improved the performance of interpreted memory operations by reducing the number of calls to complex memory simulation code [85].

More recently, Chang et al. [31], Wang et al. [125], Hong et al. [60] have presented novel schemes for speeding up address translation in full-system simulators, by utilising shadow page tables and co-ercing the host operating system into maintaining a virtual memory mapping with `mmap`-based shared memory.

In Chang et al. [31], a shadow page table – called embedded shadow page table (ESPT) – is embedded into the address space of a cross-ISA dynamic binary translation (DBT) system. ESPT uses the hardware memory management unit in the CPU to translate memory addresses, instead of software translation.

However, the original ESPT approach has a few drawbacks. For example, its implementation relies on a loadable kernel module (LKM) to manage the shadow page table. Using LKMs is less desirable for system virtual machines due to portability, security and maintainability concerns. Hence, a different implementation – called HSPT – adopts a shared memory mapping scheme to maintain the shadow page table using only `mmap` system calls [125].

Dynamic resizing of a software TLB is proposed in [60]. Using per-page-table utilisation information, the size of the software TLB is adjusted for each process separately.

## 2.5 User-mode Simulation

Most instruction set simulators, either academic or commercial, are user mode simulators, which do not provide support for privileged instructions, interrupt handling, devices or a memory management unit. As such they are not capable of hosting an operating system, but only a single process which interacts with

| Simulator | Engine | Multi-core | Detailed | Target ISA |
|---|---|---|---|---|
| CMP$im | Bin. Instr. | Yes | Cache | Intel x86 |
| FastSim | Direct Exec. | No | Yes | SPARC v9 |
| Graphite | Direct Exec. | Yes | Yes | Intel x86 |
| HORNET | Interpreter | Yes | Yes | MIPS |
| Shade | DBT | No | No | SPARC v8/9, MIPS 1 |
| SimpleScalar | Interpreter | Yes | Yes | Alpha, PISA, ARM, x86 |
| SlackSim | Interpreter | Yes | Yes | SimpleScalar/PISA |
| Sniper | Direct Exec. | Yes | Yes | Intel x86 |
| QEMU | DBT | Yes | No | Multiple available |
| WWT II | Direct Exec. | Yes | Yes | SPARC v9 |
| ZSim | Direct Exec. | Yes | Yes | Intel x86 |

**Table 2.1:** Comparison of user-mode simulators: techniques and capabilities.

the simulator though emulated system calls. This form of simulation is not applicable to hardware virtualisation, but it is related to the instruction emulation requirement for cross-architecture virtualisation.

A number of user mode simulators are listed in Table 2.1 and are briefly discussed in the following paragraphs.

CMP$im [66] uses binary instrumentation as an alternative to execution-driven and trace-driven simulation methodologies. Using the binary instrumentation tool Pin [83], CMP$im is used to characterise cache performance and data sharing behaviour of multi-threaded workloads at speeds of 4-10 MIPS.

FastSim [107] is a cycle-accurate, direct-execution simulator of an out-of-order uni-processor. It models a SPARC v8 instruction set running on a MIPS R10000-like microarchitecture and simulates a single processor. FastSim's processor model supports out-of-order instruction execution, speculative execution, and an aggressive non-blocking cache.

Graphite [91] is an open-source distributed parallel multi-core simulator infrastructure. Graphite combines several techniques including: direct execution, seamless multi-core and multi-machine distribution, and lax synchronisation. Graphite is capable of accelerating simulations by distributing them across multiple commodity Linux machines. When using multiple machines, it provides the illusion of a single process with a single, shared address space, allowing it to run off-the-shelf pthread applications with no source code modification.

HORNET [80] is a configurable, cycle-level multi-core simulator with sup-

port for a variety of memory hierarchies, interconnect routing and *virtual channel* (VC) allocation algorithms, as well as accurate power and thermal modelling. Its multi-threaded simulation engine divides the work equally among available host processor cores, and permits either cycle-accurate precision or increased performance, at the cost of some accuracy, via periodic synchronisation. HORNET can be driven in network-only mode by synthetic patterns or application traces, or in full multi-core mode using a built-in MIPS core simulator.

Shade [36] is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyser. To reduce communication costs, Shade and the analyser are run in the same address space. To further improve performance, code which simulates and traces the application is dynamically generated and cached for reuse. It runs on SPARC systems and, to varying degrees, simulates the SPARC (Versions 8 and 9) and MIPS I instruction sets.

The SimpleScalar tool set [14] is a system software infrastructure used to build modelling applications for program performance analysis, detailed microarchitectural modelling, and hardware-software co-verification. Using the SimpleScalar tools, users can build modelling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution and state-of-the-art branch prediction. These SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations.

Slacksim [33] is a parallel simulation technique to accelerate microarchitecture simulation of *chip multiprocessors* (CMPs) by exploiting the inherent parallelism of CMPs. It simulates each core of a target CMP in one thread and then spreads the threads across the hardware thread contexts of a host CMP. Starting with cycle-by-cycle simulation Slacksim relaxes synchronisation conditions around using POSIX threads using a number of schemes, resulting in improved simulation performance for multi-threaded workloads.

Sniper [29] is a parallel, high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite [91] sim-

ulation infrastructure, allowing for fast and accurate simulation and for trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures. The Sniper simulator allows one to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications with 10s to 100+ cores.

The Wisconsin Wind Tunnel II [93] is a parallel, discrete-event, direct execution simulator supporting a wide range of platforms, such as desktop workstations, a SUN Enterprise server, a cluster of workstations, and a cluster of symmetric multiprocessing nodes.

The recent ZSim [105] simulator parallelises the core of the simulator. ZSim simulates applications in two phases, a bound and a weave phase, the phases are interleaved and only work on a small number of instructions at a time. The bound phase executes first and provides a lower bound on the latency for the simulated block of instructions. Simulated threads can be executed in parallel since no interactions are simulated in this phase. The simulator then executes the weave phase that uses the traces from the bound phase to simulate memory system interactions. This can also be done in parallel since the memory system is divided into domains with a small amount of communication that requires synchronisation. Since ZSim is built using the Intel Pin instrumentation framework, it only supports user-space x86 code and does not simulate any devices (e.g., storage and network). The main focus of ZSim is simulating large parallel systems.

## 2.6 Hardware Virtualisation

### 2.6.1 Same-architecture Virtualisation

Same-architecture virtualisation is well supported by modern hypervisors, and as mentioned in Chapter 1, the technology is used for a wide range of purposes. Table 2.2 shows a list of some of the most popular same-architecture virtualisers, and summarises their features. The majority of these systems support hardware-assisted virtualisation, but in some cases they also support a limited form of DBT, where privileged instructions are re-written. In VMware's case, a *scan before execute* (SBE) strategy is employed that determines if code that

| Hypervisor | Engine | Multi-core | Hardware Accelerated | Arch. |
|---|---|---|---|---|
| VirtualBox | DBT/HVM | Yes | Intel VT, AMD-V | x86 |
| Parallels Desktop | HVM | Yes | Intel VT | x86 |
| Xen(*) | HVM/PV | Yes | Intel VT, AMD-V | x86, ARM |
| QEMU/KVM | HVM/PV | Yes | Intel VT, AMD-V, ARM Virt. Ex. | x86, ARM |
| VMware ESXi | SBE/PV | Yes | Intel VT, AMD-V | x86 |
| Hyper-V | HVM | Yes | Intel VT, AMD-V | x86 |

(*) Xen previously had support for PowerPC and Intel IA-32/64, and has experimental support for MIPS.

**Table 2.2:** Comparison of same-architecture hardware virtualisers: techniques and capabilities. **DBT:** dynamic binary translation, **HVM:** hardware virtual machine, **PV:** para-virtualisation, **SBE:** scan before execute.

is about to be executed contains any special handling. Many of these systems also support para-virtualisation of some form, enabling modified guests to run at higher speeds.

Oracle VirtualBox [94] is an open-source hypervisor that virtualises the x86 architecture. It has support for hardware-assisted virtualisation (either Intel VT or AMD-V) and also supports *second level address translation* for efficient virtualisation of the MMU. VirtualBox runs on Linux, OS X, Windows and Solaris, and supports running many different guest operating systems, including Windows, Linux and BSD variants.

Parallels Desktop for Mac [47] is a popular commercial product for running virtual machines on an Intel-based Macintosh computer. The typical use-case is to run a Microsoft Windows virtual machine, so that users can run Windows-only software on their Mac computers, but Parallels Desktop also supports other guest operating systems, such as Linux. It also supports seamless integration of the guest graphical subsystem, to make it appear as though guest Windows applications are running natively in the OS X environment. Since Parallels Desktop is designed for the Intel Mac, it only supports hardware-assisted virtualisation through the use of Intel VT.

Xen [19] is an open-source *type 1* (native) hypervisor that is widely used for server consolidation, rapid provisioning, fault tolerance and virtual machine migration. Xen supports a number of operating modes, including hardware-assisted virtualisation and para-virtualisation. It is based on a microkernel design, and partitions guest operating systems into so-called “domains”. The

first domain (`dom0`) is a privileged domain, with full access to the host system hardware. Typically, `dom0` is started with a Linux or BSD-based system, and it is used to manage the hypervisor and launch unprivileged (`domU`) domains. Unprivileged domains can be unmodified operating systems (where privileged instructions are trapped by the host's hardware extensions), or para-virtualised operating systems (where the guest operating system uses hypercalls to communicate with the hypervisor).

QEMU/KVM [74] is a particular operating mode of the popular QEMU *type 2* (hosted) hypervisor that interfaces with the Kernel Virtual Machine (KVM) for hardware-assisted virtualisation on Linux host systems. Previously, a custom kernel module called KQEMU was developed to accelerate QEMU by allowing guest machine code to run directly on the processor and emulating privileged system instructions. However, shortcomings in its design and implementation led to it being phased out, preferring KVM for acceleration instead.

VMware ESXi [120] is another commercial hypervisor, but in contrast to Parallels it is a *type 1* native hypervisor and more closely related to Xen. Similar to Xen, it operates on a bare-metal system, and relies on the Linux kernel for infrastructure support. VMware ESXi supports unmodified and para-virtualised guests, but takes a significant performance penalty for unmodified operating systems, due to overhead associated with its virtualisation strategy.

Hyper-V is a commercial Microsoft Windows-based hypervisor. Although it is a *type 1* native hypervisor, it requires a 64-bit variant of Windows to operate, as it is closely tied with Microsoft's server operating system offering. Additionally, hardware-assisted virtualisation technology from Intel VT or AMD-V is required, and the system can utilise second-level address translation technology. Hyper-V, much like the majority of other commercial hypervisors, includes a para-virtualisation technology that enables efficient device I/O (branded Enlightened I/O), by enabling direct access to devices from the guest.

Penneman et al. [99] investigate DBT techniques for same-architecture virtualisation on ARM platforms that do not support the more modern ARM Virtualization Extensions. Similarly, Gorgovan et al. [48] introduce an efficient dynamic binary modification tool, but target instrumentation of applications and not hardware virtualisation.

| Simulator | Engine | Multi-Core | Detailed | Hardware Accelerated | Target ISA |
|---|---|---|---|---|---|
| ARCSIM | Async. DBT | Yes | Config. | No | User Retargetable |
| CAPTIVE | DBT | Yes | Yes | Yes | User Retargetable |
| Embra | DBT | Yes | Cache | No | MIPS R3000/R4000 |
| gem5 | Discr. Event | Yes | Yes | No | User Retargetable |
| MARSS | DBT | Yes | Yes | No | Intel x86 |
| OVPSim | DBT | Yes | No | No | Multiple available |
| pFSA | Direct Exec. | No | Sampling | Same ISA | Intel x86 |
| PTLsim | Virtualisation | No | Yes | No | Intel x86-64 |
| QEMU/DBT | DBT | No | No | No | Multiple available |
| PQEMU | DBT | Yes | No | No | ARM11MPCore |
| XEMU | DBT | Yes | No | No | Multiple available |
| Simics | Interpreter | Yes | Approx. | No | Multiple available |
| Simit-ARM | DBT | No | No | No | ARMv5 |
| SimNow | DBT | Yes | (COTSon) | No | Intel x86, AMD64 |

**Table 2.3:** Comparison of hardware virtualisers: techniques and capabilities.

### 2.6.2 Cross-architecture Virtualisation

Cross-architecture virtualisation, often described as full system simulation, is an active field of research and a large number of techniques for the efficient implementation of these systems have been published, e.g. Böhm et al. [24], Böhm et al. [23], Witchel and Rosenblum [127], Binkert et al. [22], Patel et al. [96], Sandberg et al. [106], Yourst [130], Bellard [21], Ding et al. [39], Magnusson et al. [86], Qin and Malik [102], AMD Developer Central [5]. Table 2.3 provides an overview of well-known full-system simulators, their capabilities and implementation techniques.

ARCSIM [24] is a configurable simulator, supporting both user mode and full system simulation, and which can be retargeted by means of a high-level architecture description [121]. It uses a parallel, optimising JIT compiler for the translation of non-linear regions of guest code to efficient host code [108]. Multi-core target platforms [3] as well as cycle-accurate performance modelling [23] are supported for a class of in-order processors. ARCSIM shall be described further in the following chapters.

CAPTIVE (Chapter 6) is a hardware virtualisation hypervisor, that supports hardware accelerated cross-architecture virtualisation. It uses the Linux Kernel-

based Virtual Machine (KVM) [74] framework available in modern Linux distributions to speed-up critical hardware virtualisation operations, required for virtualising a guest architecture that is different to the host architecture.

Embra [127] is an early example of a full system simulator, pioneering dynamic binary translation principles and chaining of translation units. Embra targets both uni-processors and cache-coherent multiprocessors, and can be configured to include a processor cache model. Address translation is supported by a software cache, which causes an eight instruction overhead per memory operation (for a cache hit).

gem5 [22] provides a highly configurable simulation framework, multiple ISAs, and diverse CPU models, complemented with a detailed and flexible memory system, including support for multiple cache coherence protocols and interconnect models. gem5 can be used either in user- or full system simulation mode, and similar to ARCSIM, it can be retargeted by means of an ISA description language. Depending on the mode of execution and accuracy of simulation detail the nominal simulation speed of gem5 varies between $\sim 3$ MIPS (fast-forwarding/ISS mode) and $\sim 300$ KIPS (detailed CPU and memory system), making it orders of magnitude slower than functional simulators such as ARCSIM, Embra, QEMU or CAPTIVE.

MARSS [96] is a full system simulation tool built on QEMU to support cycle-accurate simulation of superscalar homogeneous and heterogeneous multi-core x86 processors. MARSS includes detailed models of coherent caches, interconnections, chipsets, memory and I/O devices.

OVPsim is a commercial full system simulator, which uses dynamic binary translation technology and allows users to create their own processor, peripheral and platform models.

pFSA [106] extends gem5 with a new CPU module that uses the hardware virtualisation support available in current ARM- and x86-based hardware to execute directly on the physical host CPU. Similar to CAPTIVE, it uses standard Linux interfaces, such as KVM that exposes hardware virtualisation to user space. pFSA offers *Virtual Fast-Forwarding* (VFF), which executes instructions to a point-of-interest anywhere in an application and then switch to a different CPU module for detailed simulation, or take a checkpoint for later use. Whilst mainly concerned with sampling based performance estimation for same-ISA simulation, it does not offer the same hardware-assisted cross-ISA virtualisation

capabilities of e.g. CAPTIVE.

PTLsim [130] uses para-virtualisation to run the target system natively. Due to the use of para-virtualisation, PTLsim requires the guest operating system to be aware of the hypervisor. The guest system must therefore use a special para-virtualisation interface to access page tables and certain low-level hardware. This also means that PTLsim does not simulate low-level components like timers and storage components. A draw-back of PTLsim is that it practically requires a dedicated machine since the host operating system must run inside the para-virtualisation environment. PTLsim uses a fast virtualised mode for fast-forwarding and supports detailed processor performance models.

QEMU [21] is an open-source, full system simulator and virtualisation hypervisor, using a portable *just-in-time* (JIT) translation engine for cross-architecture emulation. For same-ISA virtualisation QEMU builds on top of KVM and uses hardware virtualisation extensions if available, whereas for cross-ISA emulation it relies on DBT and its own software MMU (with software caching). Wang et al. [124] extend QEMU further for high-performance cross-architecture virtualisation.

PQEMU [39] extends QEMU with more efficient multi-core target support. QEMU itself runs DBT-based multi-core simulations in a single-thread, with each virtual CPU being executed in a round-robin fashion.

XEMU [123] is a cross-architecture full-system simulator that is designed to be high-performance for multi-core simulation. XEMU pays special attention to the translation of atomic instructions, and the communication overheads associated with multi-core simulation.

The commercial Simics simulator [86] employs a software caching mechanism, which improves the performance of interpreted memory operations by reducing the number of calls to complex memory simulation code. This is also supported by a lazy memory allocation scheme, which reduces the size of the simulator process. Overall, Simics' interpreter, based on threaded code, is not competitive any more when compared to state-of-the-art JIT based simulators, which provide a magnitude or more better simulation performance.

Simit-ARM [102] is an instruction-set simulator that runs both system-level and user-level ARM programs. It supports interpretation and dynamic-compiled simulation. SimIt-ARM supports the ARMv5 architecture, including the memory management unit and some fundamental I/O devices. On a Pentium D

2.8GHz desktop, the interpreter runs at around 30 MIPS. The dynamic-compiled simulator runs even faster, especially for long simulation tasks or when a multi-core workstation is used, but it does not reach the performance of QEMU or CAPTIVE. Compared to the direct translation approach in QEMU, the GCC-based approach in Simit-ARM is easier to implement, but at the cost of translation speed. To reduce translation delay, SimIt-ARM distributes translation tasks to other CPUs or workstations via either pthreads or sockets.

COTSon [6] combines AMD SimNow [5] (a JIT-based functional x86 simulator) with a set of performance models for disks, networks, and CPUs. The simulator achieves good performance by using a dynamic sampling strategy that uses online phase detection to exploit phases of execution in the target.

Penneman et al. [98] review the formal virtualisation requirements for the ARM architecture. In Liu et al. [81] the CASL hypervisor for the ARM architecture is presented.

STAR [100] is an open-source hypervisor for ARMv7-A with full virtualisation using DBT.

Gutierrez et al. [51] investigate the sources of error in full system simulation. They concentrate their effort on identifying the underlying causes for inaccuracies of several key microarchitectural statistics. As this thesis is not concerned with microarchitectural simulation, but aims at providing a high-performance virtualisation system, the sources of error identified in [51] do not apply to this work.

## 2.7 Summary

There is a significant amount of related work in the area of simulation and virtualisation, with a lot of effort going into performance improvements for functional virtualisation. Of primary concern in this thesis is cross-architecture virtualisation, and the following chapters shall describe new approaches and techniques that enable efficient virtualisation of a guest architecture that is different to the host's.

Chapter 3

# Infrastructure

The work being presented in the following chapters depends on certain existing and new tools, and this chapter shall describe the frameworks and infrastructure used in the remainder of this thesis.

Throughout the following chapters, there will be three main tools used to implement the ideas and techniques being presented:

- **GENSIM:** A tool that generates hypervisor components from a high-level architecture description.
- **ARCSIM:** A high-performance instruction set simulator, with support for hardware virtualisation.
- **CAPTIVE:** A hardware accelerated cross-architecture hypervisor.

GENSIM and ARCSIM are pre-existing tools, but have been extended throughout the course of this research. CAPTIVE is a new tool that has been developed specifically for the techniques presented in Chapter 6. Each of these tools shall now be described in turn.

## 3.1 GENSIM

Any form of cross-architecture virtualisation requires a description of the architecture being virtualised, and hard-coding this description into a hypervisor is time consuming, error prone, hard to debug and can lead to poor code quality. For this reason, GENSIM was created as a tool that accepts a high-level architecture description (in an ArchC-like [16] *architecture description language* (ADL))

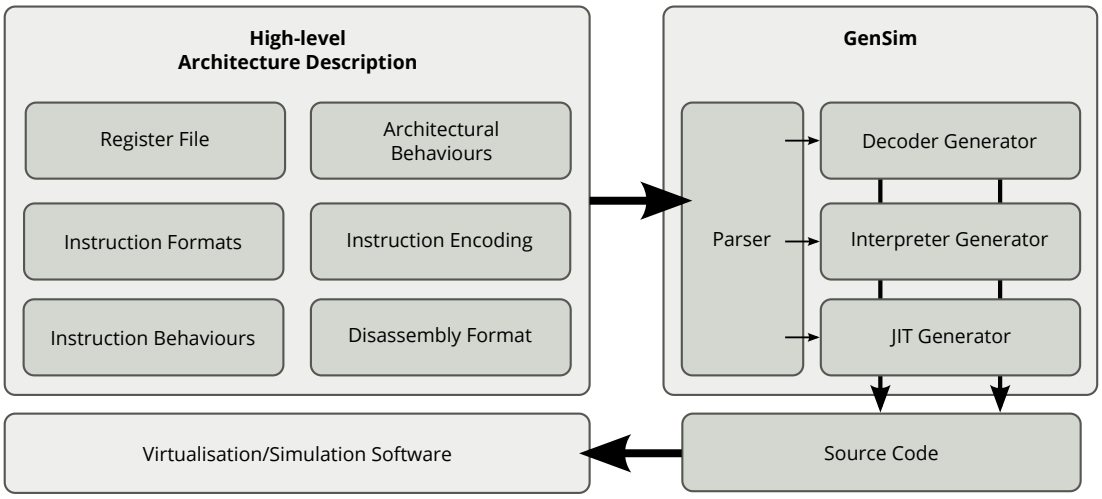



**Figure 3.1:** A high-level overview of the GENSIM generation tool. GENSIM accepts a high-level architecture description, which is parsed and used by *generators* to produce source-code as output. This source-code can be included in simulation or virtualisation software, to provide services (such as instruction decoding, or JIT compilation) specific to the architecture being described.

and produces source-code as output that can be included in simulation or virtualisation systems. This leads to a very fast turn-around time for producing simulators for architectures that may or may not exist (enabling rapid prototyping of hardware platforms), along with benefits such as automated model testing, guest application debugging and ease of development.

Figure 3.1 shows the basic operation of GENSIM. The ADL describes the architecture's register file, instruction formats and encodings, instruction behaviours, assembly-language formats and architecture-specific behaviours, such as how page faults are handled. GENSIM parses the ADL and maintains an internal representation of this description. The output of GENSIM is governed by *generators*, which interrogate the internal representation, and produce output files (which are usually source code files) that perform actions specific to the described architecture.

GENSIM is designed to produce modules that contain implementations of a particular "service" that is found in a generic simulation framework. For example, a simulator needs to have the ability to decode guest machine instructions (e.g. for interpretation) and GENSIM can produce a module that implements an instruction decoder for the described architecture. The following list contains examples of the kind of modules that can be produced:

- **High-speed instruction decoders:** Used for decoding guest instructions, for example, in order to execute associated behaviours or for dynamic binary translation.
- **Disassemblers:** Used primarily for debugging during development of applications, or even the simulator itself.
- **Interpreters:** Used for simple, but slow, guest instruction emulation.
- **JIT compilers for DBT:** Used for efficient, guest instruction emulation.
- **Support infrastructure:** Used to provide a representation of the target CPU state (e.g. the register file), and for implementing specific architectural behaviours.

Internally, the generator system in GENSIM is pluggable, and so it is trivial to develop a generator that produces, for example, an instruction decoder to be used as part of a particular application. GENSIM also optimises much of the internal representation, to ensure generators produce output modules that contain highly optimal code.

This thesis will focus on virtualising an ARM guest system on an x86 host system, and the following sections will give an overview of the high-level architectural model used throughout, and how GENSIM is employed to generate the guest architecture specific components. The description in use has been developed over a number of years by a range of individuals, and has been further extended to support hardware virtualisation as part of this thesis.

### 3.1.1 High-level Architecture Description

The ADL is structured such that the *top-level description file* describes certain architecture-wide definitions. Listing 3.1 shows an extract from the ARMv7-A model that describes the register file present on the ARM guest CPU. In this example, lines 2–5 define the general purpose register bank, and in particular identifies the register that is the *program counter* (PC). This identification is required, as simulators generally need knowledge of the PC for control-flow purposes. Lines 8–15 define the CPU flags, which are updated by instructions that set flags after an operation. The most common flags (C-carry, Z-zero, N-negative, V-overflow) are specially identified for simulators that support flag setting optimisations.

```
1  // General Purpose Registers
2  ac_regspace(64) {
3    bank RB (uint32, 16, 4, 4, 0);
4    slot PC (uint32, 4, 60) PC;
5  }
6
7  // General Flags
8  ac_regspace (6) {
9    slot C (uint8, 1, 0) C;
10   slot Z (uint8, 1, 1) Z;
11   slot N (uint8, 1, 2) N;
12   slot V (uint8, 1, 3) V;
13   slot Q (uint8, 1, 4);
14   slot A (uint8, 1, 5);
15 }
```

**Listing 3.1:** High-level architectural register description.

```
1  // Define the MOVTW instruction format: 4-bit condition, 4-bit operand, ...
2  ac_format<Type_MOVTW> = "%cond:4 %op:4 %subop:4 %rn:4 %rd:4 %imm32:12";
3
4  // Link movt, movw instructions to the Type_MOVTW instruction format
5  ac_instr<Type_MOVTW> movt, movw;
6
7  // Define the ARM instruction set architecture
8  ISA_CTOR(arm) {
9    // Instruction: movt, where (op = 0x3) and (subop = 0x4)
10   movt.set_decoder(op = 0x3, subop = 0x4);
11   // Assembly mnemonic
12   movt.set_asm("movt[%cond] %reg, #%imm", cond, rd, imm32);
13   // Instruction behaviour is defined in the "movt_behaviour" function
14   movt.set_behaviour(movt_behaviour);
15
16   // Instruction: movw, where (op = 0x3) and (subop = 0x0)
17   movw.set_decoder(op = 0x3, subop = 0x0);
18   // Assembly mnemonic
19   movw.set_asm("movw[%cond] %reg, #%imm", cond, rd, imm32);
20   // Instruction behaviour is defined in the "movw_behaviour" function
21   movw.set_behaviour(movw_behaviour);
22 }
```

**Listing 3.2:** High-level instruction format description.

Listing 3.2 shows an extract of an instruction format description, again from the ARMv7-A model. Line 2 contains a bit-level representation of the instruction format, and line 5 declares two instructions (`movt` and `movw`) that conform to this pattern. Lines 10 and 17 further specialise the pattern, specific to the two instructions, by placing constraints on the values of the fields defined in the instruction format. Lines 12 and 19 assist debugging by producing a disassembly format for the instructions, in a `printf`-style declaration. Finally, lines 14 and 21 attach semantic behaviours to the instructions.

The semantic behaviour of instructions is defined in a C-like *domain specific language* (DSL) called GenC. Listing 3.3 shows the GenC that describes the

```
// Instruction behaviour for "movt"
execute(movt_behaviour)
{
  uint32 orig = read_register_bank(RB, inst.rd) & 0xffff;
  uint32 rn = inst.imm32 | (inst.rn << 12);

  uint32 result = orig | (rn << 16);
  write_register_bank(RB, inst.rd, result);
}

// Instruction behaviour for "movw"
execute(movw_behaviour)
{
  uint32 result = inst.imm32 | (inst.rn << 12);
  write_register_bank(RB, inst.rd, result);
}
```

**Listing 3.3:** Semantic description of the behaviour of the `movt` and `movw` instructions.

behaviours of the corresponding instructions.

The semantic instruction behaviours are parsed by GENSIM, and an internal *static single assignment* (SSA) form is maintained for each behaviour. This SSA is analysed so that the partial evaluation technique described by Wagstaff et al. [121] can be employed, and then optimised so that generators that utilise it can produce efficient output.

### 3.1.2 Output Components

GENSIM produces source-code as output, which can be compiled directly into the virtualisation system or compiled into a shared library that is loaded by the system dynamically. As GENSIM is employed for both ARCSIM and CAPTIVE, command-line options dictate the target of the output, causing the appropriate generators to be invoked.

When generating modules for ARCSIM, a high-speed instruction decoder, interpreter, LLVM-based JIT compiler, a disassembler (for debugging purposes) and a CPU model are generated. GENSIM outputs a number of C++ source-code files that contain the generated implementations of the various services. For example, a particular source-code file contains the implementation of an instruction decoder for the described architecture and another contains a disassembler. After generation, these files are then compiled and linked to produce a single shared object. This shared object is loaded by ARCSIM at start-up, and accessed by the generic simulation framework, when the appropriate architecture-specific service is required.

A similar output is produced for CAPTIVE, the difference being that the generators produce source-code that targets the CAPTIVE API.

### 3.1.3 Automated Model Testing

A particular benefit of using a high-level architecture description is that the model is amenable to automated testing. Testing of a high-level architectural model is described by Wagstaff et al. [122], with this technique being developed and used throughout the engineering of the ARM model. Employing this testing process significantly reduces the amount of debugging required to locate problems (such as bugs in instruction implementations), by verifying the behaviour of the model against a reference platform and simulator.

## 3.2 ARCSIM

ARCSIM is a high-performance instruction set simulator, developed at the University of Edinburgh. It supports both *user-mode* and *full-system* simulation.

It employs a region-based asynchronous DBT system for obtaining high simulation throughput rates, and uses LLVM to generate highly efficient native code. Originally, ARCSIM was designed solely to simulate the EnCore microprocessor [114], but has since been extended to accept modules generated by GENSIM in order to simulate other architectures. Even though ARCSIM has been extended further, and can now simulate much more than the ARC architecture, the name remains.

Figure 3.2 shows the main execution loop of ARCSIM, which employs an interpretive component and farm of concurrent JIT compiler threads to achieve maximum speed. Execution begins by running the target program through the interpreter and collecting profiling information about the basic blocks by building a region-oriented *control flow graph* (CFG). A heuristic decides if a region is eligible for compilation, by considering the number of basic blocks discovered within the region, and the *heat* (the number of times a basic block has been executed) of those blocks. It will then be dispatched to a JIT compiler worker thread, which will translate the region to native code. This process is asynchronous, and the target program will continue executing in the interpreter while the region is being translated. Once the native code has been compiled, it

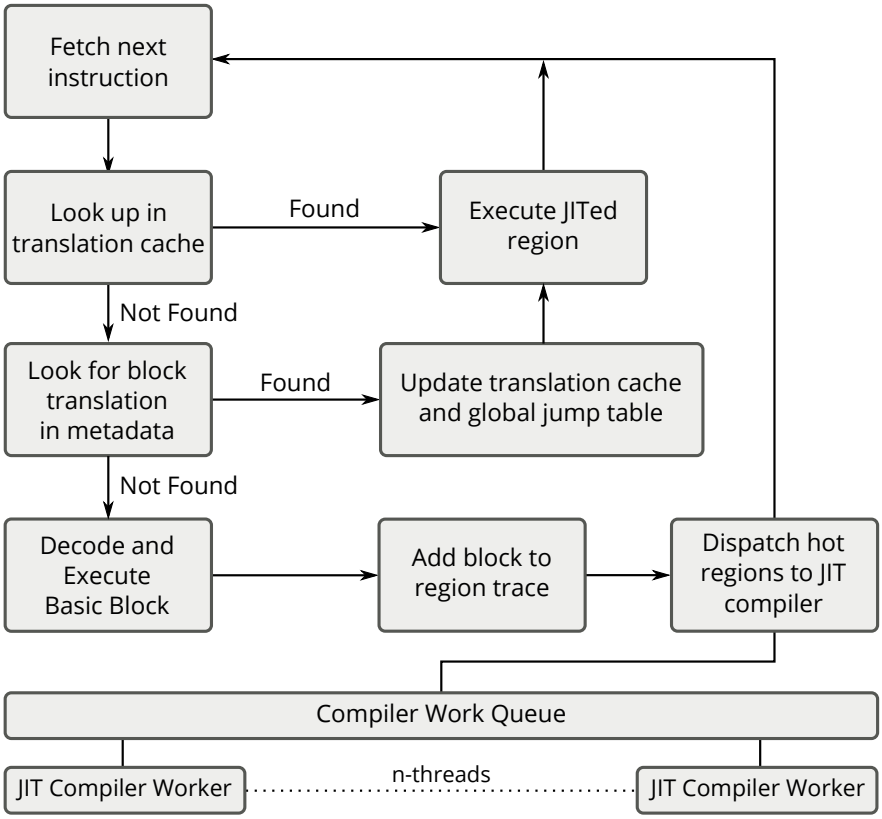



**Figure 3.2:** Main execution loop of ARCSIM with decoupled, concurrent JIT compilation threads. Initially, execution of guest basic blocks is performed with an interpreter, but profiling information is collected to identify basic blocks that are "hot". Once a region of "hot" basic blocks has been discovered, it is dispatched to a JIT worker thread, which compiles the region to native code. Then, execution transitions into native code, if a translation is available.

will be made available by registering *region entry points* in block metadata, and when the interpreter encounters a registered block, it will update the translation cache and begin executing the native code. Once inside native code, execution will remain there as long as blocks are available to execute. If a basic block is encountered that has not yet been compiled, control will return to the interpreter and profiling information updated accordingly. Gathering further profiling information about a region may lead to a region becoming eligible for recompilation, which gives rise to progressively optimal code, much like *tiered* or *staged* compilation [70].

### 3.2.1 LLVM Compiler Infrastructure

LLVM [77] is a modern framework for compiler development. The core framework consists of a high-level *intermediate representation* (IR), a number of transformation passes, and a wide range of back-end machine code generators, targeting many different architectures. Various front-ends (such as the Clang C/C++ compiler) produce LLVM IR, which is then passed through a series of optimisation passes, until finally being lowered to machine code by the back-end. LLVM also provides a JIT compilation interface, making it suitable for use in a DBT system.

LLVM is widely used by scientific and research institutions, as well as being used in commercial deployments. Its design is highly amenable to extension and modification, and can be conveniently integrated into new and existing software—unlike GCC, which is notoriously less flexible. Although there have been recent efforts to improve the extensibility of GCC [63], the nature of LLVM is such that it still remains the compilation framework of choice for modern research.

In ARCSIM, JIT compilation of the guest basic blocks is performed by translating the guest instructions in those blocks into corresponding LLVM IR. At this point, the guest instructions have already been decoded, so for each instruction its *translation function* is called, which generates the IR that represents its behaviour. The translation functions are generated by GENSIM, as described previously. Once the IR has been generated for a region, it is passed through the standard LLVM optimisations, until finally being compiled into native host machine code using the LLVM JIT compiler.

## 3.3 CAPTIVE

CAPTIVE is a new cross-architecture hardware virtualisation hypervisor that has been developed as part of this research. It is designed to utilise host machine hardware extensions for virtualisation, e.g. Intel VT, and it accesses these extensions through the KVM framework (see Section 3.3.1). Figure 3.3 gives a high-level overview of the interaction of the main components in CAPTIVE.

Since virtualisation extensions are primarily designed for same-architecture virtualisation, the key idea is to map the behaviour of a guest machine, onto the

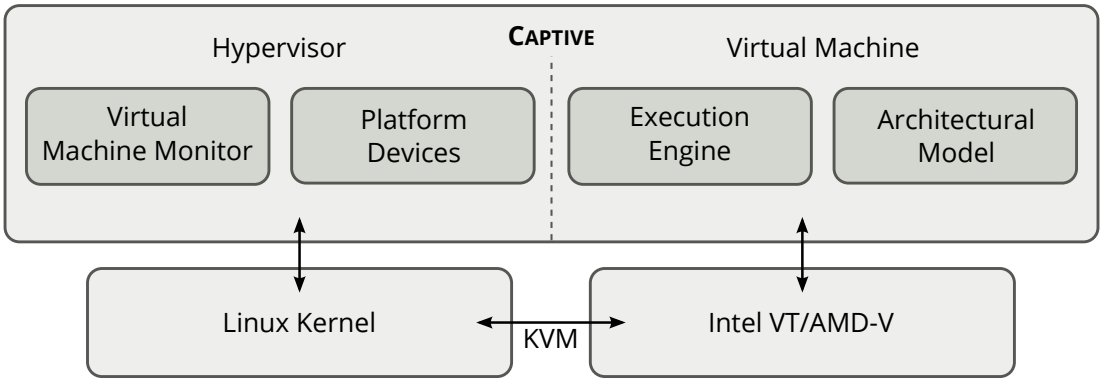



**Figure 3.3:** CAPTIVE uses KVM to create a same-architecture virtual machine on the host system. An architectural model (generated by GENSIM) maps the behaviour of the guest machine to behaviour of the host machine. The CAPTIVE hypervisor provides software implementations of platform devices, for use by the guest system.

behaviour of a host machine. For example, when using CAPTIVE to virtualise an ARM platform on an x86 host machine, the behaviour of the ARM *memory management unit* (MMU) is mapped onto the behaviour of the x86 MMU, enabling high performance memory address translation.

CAPTIVE can easily be retargeted to different host and guest machines. Retargeting to a different host requires the host Linux kernel to support KVM, the implementation of host-specific machine setup code and the development of a back-end to the JIT compiler. The guest machine is retargeted by creating an architectural model, and using GENSIM to generate the architecture-specific components. Additionally, CAPTIVE contains software implementations of various devices, so that guest platforms that require them can be fully virtualised. This device model is pluggable, enabling new device implementations to be developed externally, and loaded dynamically.

### 3.3.1 KVM

Hardware accelerated virtualisation is well supported by multiple processor vendors, e.g. Intel with Intel VT, AMD with AMD-V and ARM with ARM Virtualization Extensions. Whilst these extensions all produce the same effect, i.e. they create an abstract computing platform on which to run unmodified operating systems, they are implemented and accessed completely differently. All of these extensions are geared towards creating a virtual machine of the same architecture as the host machine, by enabling operating system software to run

directly on the host hardware, with minimal supervision.

KVM [74] is a *virtual machine monitor* implemented as part of the Linux kernel, which can utilise any supported hardware accelerated virtualisation extensions, on any platform. Its job is to abstract the details of the virtualisation extensions, and to provide a generic interface for creating and managing a virtualisation environment. KVM also fully supports additional virtualisation extensions present on x86 platforms, such as Intel *extended page tables* (EPT), or AMD *rapid virtualization indexing* (RVI), which are used to accelerate virtualised MMUs. KVM itself is only an interface to hardware virtualisation extensions, it will not work in the absence of these.

A common misconception is that KVM depends on (or requires) QEMU [21] to be used, but although KVM was developed in tandem with QEMU, it is an independent technology that is part of the standard Linux kernel, and available for use by any developer wishing to create a platform-independent hypervisor, an example being the Native Linux KVM tool [53].

### 3.3.2 Intel VT

Intel VT [65] is a set of hardware extensions introduced by Intel to provide support for virtualising x86 processors. It provides new instructions for setting up and managing virtual machines, transitioning between hypervisor and virtualised execution, and support for virtualising the guest MMU with *extended page tables* (EPT). EPT provides an extra level of page tables that are walked by hardware when a virtual memory address that is not present in the TLB is encountered while running in the virtual machine.

This technology consists of a number of "sub-technologies" that enable efficient virtualisation of hardware resources:

- **Intel VT-x:** This is the CPU virtualisation technology, that enables efficient virtualisation of a physical processor. It provides the isolated run-time environment for operating system software.
  - **EPT:** EPT or *extended page tables* is part of Intel VT-x, and is hardware acceleration for MMU virtualisation.
- **Intel VT-d:** This technology enables efficient virtualisation of devices on the host platform.

- **Intel VT-c:** This technology gives the ability to virtualise network and communication devices efficiently.

## 3.4 QEMU

QEMU [21] is an open-source hardware virtualisation system that is widely used in academia and industry. It supports both user-mode simulation, and hardware virtualisation by means of dynamic binary translation. QEMU can also use KVM to perform highly efficient same-architecture hardware virtualisation, using the host machine's hardware virtualisation extensions.

In DBT mode, QEMU employs a custom JIT compiler called the *tiny code generator* (TCG). The guest architecture implementation produces *TCG ops*, which are then optimised and lowered into host machine code. However, it does support many different guest architectures, and it has an extensive library of device implementations.

QEMU is not easily retargetable, as the guest architecture is developed directly in the main source-code distribution, and the instruction decoder is hard-coded into the translation implementation.

The majority of the related work presented in Chapter 2 uses QEMU as a baseline for comparing their own techniques, making QEMU suitable for the performance comparisons in this thesis, as it is widely accepted as state-of-the-art in the existing scientific literature.

## 3.5 Evaluation

The evaluation of the techniques described in the following chapters are all made in a similar fashion, and since the underlying idea is to increase the performance of virtualisation systems, the key results take the form of a performance comparison between the ideas presented and existing state of the art implementations. In general, the direct competitor is QEMU, but in certain cases, more recent and related work that has extended QEMU to improve performance is considered.

Each individual chapter describes the specific methodology used during evaluation, but in general performance measurements are made by running the

SPEC-CPU2006 benchmark suite, described in Section 3.5.2.

### 3.5.1 Guest Architecture and Platform

Implementing the behaviour of a guest architecture is a time-consuming and error-prone process, and so a particular architecture is chosen for the basis of the experiments in this thesis. The chosen architecture is ARM, as this architecture sees widespread real-world usage, and much of the related work uses this architecture for their research. It is a very popular architecture, with detailed information freely available from ARM making development of application and system software very easy, and therefore also the development of simulators. Being the architecture of choice for modern mobile phones, it is well supported by the Linux kernel, on which the Android® operating system is built, making it an ideal target for virtualisation.

The choice of architecture itself is not enough for performing hardware virtualisation, as it is a particular *platform* that dictates the implementation of the architecture. The platform describes the type and configuration of the CPU core(s), along with the versions of the various architectural components in use. The platform also describes the configuration and location of the *devices* that are present in the system. These *platform devices* may be I/O devices, such as disk, network and graphics devices or they may be internal devices such as timers, debugging and control interfaces.

In Chapter 5, the ARM platform implemented is an ARM Versatile Application Baseboard [11], and in Chapter 6, the platform is an ARM RealView Platform Baseboard for Cortex-A8 [9]. These two platforms have quite similar device requirements, i.e. the majority of the devices are the same, but they are located in a different place in *physical memory*. The two most notable differences between the platforms are the version of the interrupt controller and the *memory management unit* (MMU).

These platforms were chosen because, like the ARM architecture itself, detailed information about them are freely available from ARM, including the behaviour of the platform devices. This makes the implementation quite straightforward.

| Benchmark | Application |
|---|---|
| 400.perlbench | Mail filtering using the Perl programming language |
| 401.bzip2 | Data compression using the BZIP2 algorithm |
| 403.gcc | C Compilation |
| 429.mcf | Vehicle scheduling using combinatorial optimisation |
| 445.gobmk | Artificial intelligence for the 'Go' board game |
| 456.hmmer | Protein sequencing using hidden Markov models |
| 462.libquantum | Prime factorisation by simulated quantum computation |
| 464.h264ref | The h.264 video codec |
| 471.omnetpp | Networked system simulation |
| 473.astar | The A* Pathfinding algorithm |
| 483.xalancbmk | XSLT transformation |

**Table 3.1:** A list of the integer benchmarks in the SPEC-CPU2006 benchmark suite. Each benchmark is designed to be representative of a real-world workload.

### 3.5.2 SPEC-CPU2006 Benchmark Suite

As mentioned previously, the SPEC-CPU2006 benchmark suite is used throughout the evaluation sections to benchmark performance of the system being described. This particular benchmark suite is developed by SPEC (*The Standard Performance Evaluation Corporation*), who also produce a range of other benchmark suites for various workloads/platforms. The suite itself contains a number of realistic workloads, designed to test integer and floating point performance of the system being benchmarked. The SPEC benchmarks are widely accepted by the research community and appear in the majority of related literature for performance comparisons.

To simplify the implementation of the guest platform, the floating point workloads are not used for evaluation purposes. This is because implementing the required instructions to support floating point instructions in the guest platform would be a significant time investment. It is possible to compile the floating point benchmarks for software-emulated floating point (*soft-fp*), however this simply produces code that emulates floating point instructions with integer instructions, and hence does not add value to the evaluation. The specific benchmarks used in this thesis, along with their classification, are listed in Table 3.1.

| Category | Description |
|---|---|
| Automotive | FFTs, cosine transforms, and general compute benchmarks. |
| Consumer | JPEG encoding/decoding and image filtering kernels. |
| Networking | Pathing and packet handling kernels common in network equipment. |
| Office | Text and image processing and manipulation kernels. |
| Telecom | Signal processing kernels including Viterbi and FFT transformations. |

**Table 3.2:** The EEMBC v1.1 benchmark suite contains over thirty benchmarks suitable for use in embedded systems. This table summarises the categories that these benchmarks fall into.

### 3.5.3 EEMBC Benchmark Suite

Another benchmark suite that is used in Chapter 4 is the EEMBC (The Embedded Microprocessor Benchmark Consortium) v1.1 benchmark suite [37, 79], which are a series of benchmarks designed for embedded systems. The individual benchmarks are much smaller than those in SPEC-CPU2006 as they are designed to be run on power-, compute- and memory constrained embedded systems, which may or may not host an operating system. The benchmark suite comprises over thirty benchmarks, and so for brevity Table 3.2 only lists the categories that they fall into.

### 3.5.4 Choice of Benchmarks

As mentioned previously, the SPEC and EEMBC benchmarks are widely accepted in literature for computational performance measurements, but benchmarks for embedded platforms (such as those found in the Android operating system) would require a significant engineering effort to implement. This is because there would be additional work required to develop the target platform in the virtualisation systems.

Chapter 4

# Efficient Dynamic Binary Translation

For cross-architecture virtualisation, the emulation of guest machine instructions is a necessity. A straightforward approach to this is *interpretation*, where the execution behaviour of each instruction is defined as a function, which is called when that instruction is encountered at run-time. Whilst it is possible to build more efficient interpreters [20], their underlying nature cannot be avoided and a performance ceiling is reached without any realistic opportunity of exceeding. This is because interpreters only consider single instructions at-a-time without performing any optimisations across instruction sequences. To improve on this, *dynamic binary translation* translates sequences of guest instructions to host instructions at run-time, producing host machine instructions that closely resemble guest machine instructions. This technique has a lot of scope for optimisation, and this chapter presents a complete strategy that can be employed in an efficient dynamic binary translation system.

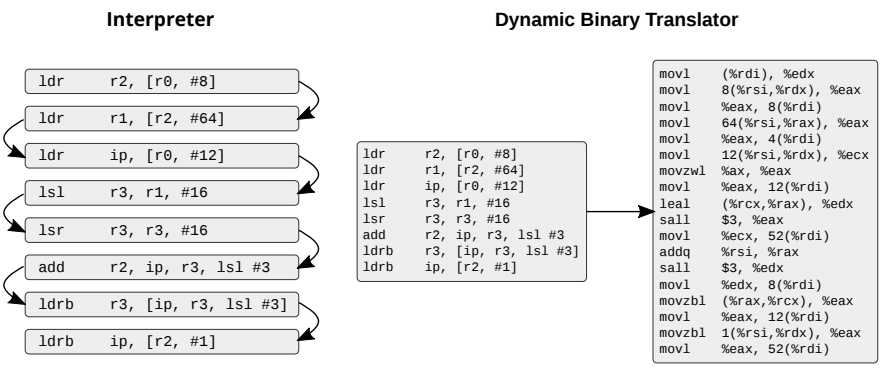



**Figure 4.1:** An interpreter will consider each guest instruction in turn, and execute an associated *behaviour*. A dynamic binary translator will take a sequence of guest instructions and produce a corresponding sequence of (optimised) host instructions.

## 4.1 Introduction

Efficient *dynamic binary translation* (DBT) relies heavily on *Just-in-Time* (JIT) compilation for the translation of guest machine instructions to host machine instructions. Although JIT compiled code generally executes much faster than interpreted code, JIT compilation incurs an additional overhead, namely the cost of compilation, or the *compilation latency*. For this reason, normally only the most frequently executed code fragments are translated to native code whereas less frequently executed code is still interpreted. Of central concern are the *size* and *shape* of these translation units presented to the JIT compiler. While smaller code fragments such as individual instructions or basic blocks take less time for JIT compilation, larger fragments such as linear traces or regions comprising control flow offer more scope for aggressive code optimisation [15]. For this reason, many modern DBT systems rely on regions as translation units for JIT compilation and several different region selection schemes have been proposed in the literature [26, 55, 57, 62]. However, it remains an open question as how to efficiently exploit such regions of any size and shape for JIT code generation, to ultimately result in improved performance.

### 4.1.1 Key Ideas

This chapter presents a complete, region-based, JIT code generation strategy considering optimised handling of branch type information and region exits, registration of JIT compiled code in translation caches, continuous profiling and recompilation, region chaining, and host code generation including domain specific alias analysis. These *key ideas* are summarised as follows:

- **Branch type information** is collected during code discovery and profiling, and is used to make decisions about how to generate optimised control-flow from the end of a basic block. Branches that occur within a region are kept internal, which directly improves code quality as unnecessarily exposed branch targets defeat control and data flow analysis.
- Only identified **region entry points** are registered in the *translation lookup cache*—arbitrary entry to a region is not allowed, again aiding control and data flow analysis and widening the scope for aggressive code optimisation.

- A form of **region chaining** is implemented, to improve performance when branching between regions.
- Execution is **continuously profiled**, growing and recompiling regions using up-to-date profiling information to include newly discovered guest basic blocks and control flow.
- **Domain-specific alias-analysis** is used during guest to host code translation, exploiting knowledge about the structure of the code, which is difficult to uncover using standard alias analysis techniques.

### 4.1.2 Motivating Example

Most DBT systems will use some form of CPU state structure that lives in memory and contains the active state of the register file and any CPU flags, along with various other control information. Similar to an interpreter-based ISS, a DBT that works on an instruction-by-instruction basis will usually access this structure for every target instruction being executed, as most instructions will involve a read or write to one or more registers and may or may not alter flags. However, a DBT that translates on a block-by-block basis (such as QEMU [21]) will typically treat the execution of a basic block as an atomic operation, and will introduce optimisations that only update the CPU state structure once the entire basic block has been executed. This is because intermediate values from the results of guest instructions can be kept in host registers, and re-used throughout the block until the last moment. This important optimisation significantly reduces the amount of reads and writes to memory, and can therefore greatly increase performance.

Traditional region-based DBTs still work on a block-by-block basis, and will allow entry to the region via any block that is part of the region, however the consequence of this is that the native code address of each basic block must be taken, and doing so hinders *inter*-block optimisations[1]. Whilst *intra*-block optimisations[2] can still be applied, more aggressive *inter*-block optimisations cannot, as guarantees about CPU state must be maintained on entry to and exit from each block. In contrast, trace-based DBTs generate inherently linear control-flow graphs, which are only ever entered from the top (the *trace head*)

[1] Optimisations across basic blocks within a region.
[2] Optimisations within a single basic block as a unit.

```
1 BEGIN: mov r2, #1
2 LOOP:  cmp r0, #0
3        beq END
4        mul r2, r2, r0
5        sub r0, r0, #1
6        b LOOP
7 END:   mov r0, r2
8        bx lr
```

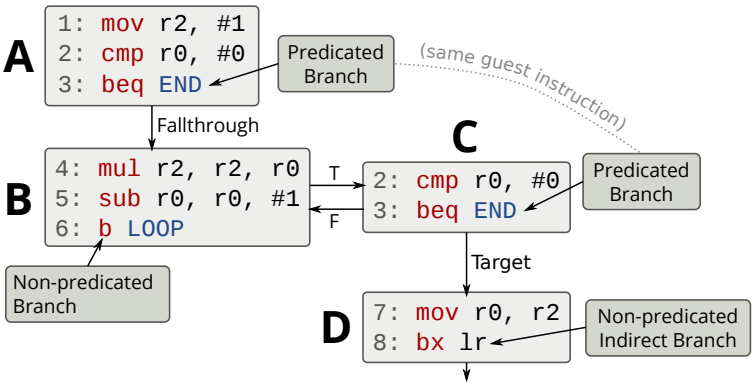



**Figure 4.2:** An example ARM function that calculates the factorial of a number. The control flow graph shows the guest basic blocks and dynamic control flow discovered by the profiler, along with meta-information about the branch instructions.

and are usually only exited at the bottom. This enables optimisations to be applied across the entire trace but due to the lack of *interesting* control-flow, they miss out on loop optimisations.

The benefit of a region-based DBT is that *non*-linear control-flow is allowed within the region, which can lead to optimisations that would not be possible with linear control-flow, but this benefit is restricted if addresses of individual blocks within the region are taken and, e.g. inserted into an *indirect branch target table* (IBTT). This limits the ability of the optimiser to keep intermediate values (such as loop induction variables) in host registers, and to defer updating the CPU state structure until an exit point is reached.

Figure 4.2 shows an extract of an ARM function that calculates the factorial of a number (supplied in guest register `r0`), along with the dynamic *control flow graph* (CFG) discovered by the profiling engine. Meta-information about the control flow instructions (i.e. the branch instructions) is stored in the profile, which records whether or not the branch is predicated, and whether or not the branch is direct or indirect. For both predicated and non-predicated direct branches, the target `PC`s of the branch and of the fallthrough are known, and so these values are stored in the profile. For indirect branches, the target `PC` is not known at compilation time, but if the branch is predicated, the fallthrough `PC` is known.

If native code was to be generated for this sequence, and entry was permitted via any basic block, then the native code generated for each basic block would need to load the values of the registers in use from the emulated reg-

ister file, and cannot re-use values held in a host register from a predecessor. Furthermore, at the end of a basic block, the register file must be updated with any changes in register values. This particular problem can pollute native code with unnecessary loads and stores when certain blocks are not actually region entries, and with careful profiling and capturing of CFG edge information, it can be determined which blocks are internal to the region.

**Definition 15** (Region Entry)**.** *A **region entry** is a guest basic block within a region that has been observed by the profiler to have been entered from a different region.*

In the example CFG given in Figure 4.2, block $A$ is a region entry, and blocks $B$, $C$ and $D$ are only branched to by control-flow from other blocks within the region. Two branch fall-through edges exist as $\overrightarrow{AB}$ and $\overrightarrow{CB}$, and two direct branches exist as $\overrightarrow{BC}$ and $\overrightarrow{CD}$. There is one indirect branch out of block $D$ (on line 8 of the code), which is a standard ARM function return sequence. It is important to note that there are two basic blocks ($A$ and $C$) discovered with overlapping code. This is because (given the input $r_0 \geq 1$) the profiler will discover the fall-through edge $\overrightarrow{AB}$ first, and then discover the direct edge $\overrightarrow{BC}$ that branches inside $A$, and will hence create a new basic block $C$ containing the latter half of $A$.

If entry to the region was allowed via any basic block, guest register values would need to be loaded from the CPU state structure when required in each block. This would be detrimental to performance, especially in the case of the loop between $B$ and $C$, as the value of the induction variable in `r0` would need to be read from memory in $C$ and written to memory in $B$, rather than keeping `r0` in a host register.

However, if the constraints are changed to only allow entry via basic block $A$, and keep $B$, $C$ and $D$ as *region local* basic blocks, then an optimised form can be generated that loads initial register values into host CPU registers, which are reused throughout the loop. When the code sequence exits, the updated register values are written back into the CPU state structure.

This difference is clearly demonstrated in Listings 4.1 and 4.2, where Listing 4.1 shows an example of x86 assembly generated for the code sequence presented in Figure 4.2. When every basic block has its address taken, the generated code must access memory to request the value of the target machine

**Listing 4.1:** Native code with block addresses taken

```
1  BLOCK_A:
2    movl $0x1, 8(%rdi)
3    mov  0(%rdi), %eax
4    test %eax, %eax
5    jz   BLOCK_D
6  BLOCK_B:
7    mov  8(%rdi), %ecx
8    mov  0(%rdi), %eax
9    imul %eax, %ecx
10   mov  %ecx, 8(%rdi)
11   sub  $1, %eax
12   mov  %eax, 0(%rdi)
13 BLOCK_C:
14   mov  0(%rdi), %eax
15   test %eax, %eax
16   jnz  BLOCK_B
17 BLOCK_D:
18   mov  8(%rdi), %eax
19   mov  %eax, 0(%rdi)
20   mov  56(%rdi), %eax
21   and  $0xfffffffe, %eax
22   mov  %eax, 60(%rdi)
```

**Listing 4.2:** Native code without block addresses taken

```
1  BLOCK_A:
2    mov  $0x1, %ecx
3    mov  0(%rdi), %eax
4    test %eax, %eax
5    jz   END
6  LOOP:
7    imul %eax, %ecx
8    sub  $1, %eax
9    jnz  LOOP
10 END:
11   mov  %ecx, 0(%rdi)
12   mov  %ecx, 8(%rdi)
13   mov  56(%rdi), %eax
14   and  $0xfffffffe, %eax
15   mov  %eax, 60(%rdi)
```

**Figure 4.3:** Host machine code generated using a naïve scheme and using the integrated, region-based code generation methodology. Listing 4.1 shows that guest registers need to be read from the register file in each basic block, but Listing 4.2 shows that guest registers can remain live in host registers by keeping some basic blocks internal.

register from the CPU state structure. In Listing 4.2, an optimised form is generated where host registers are used to track the state of the target machine register, until the end of the sequence, at which point the values are written back to memory. In this example, the x86 register `EAX` is chosen to shadow the loop induction variable, which for the guest machine exists in ARM register `r0`. This removes all memory accesses from the loop between block $B$ and $C$, and can exploit host ISA features to generate an extremely efficient loop.

In general, the guiding principle is speculation and optimisation for the common case, i.e. profiling information on branch types, region entries, and indirect branch targets is used immediately for code optimisation even if there is the possibility of later updates of this information, possibly initiating recompilation.

### 4.1.3 Contributions

This chapter is not concerned with developing new ways of region selection, but its focus is on a strategy for efficient code generation and optimisation for regions, once these have been formed using any of the techniques presented in the literature [26, 55, 57, 62]. Furthermore, it does not seek to propose new techniques for resolving indirect branches, but rather it is shown how branch type and control-flow information can be exploited on top of any of the existing mechanisms for resolving indirect branches [58, 38, 75, 128, 67]. Overall, this chapter makes the following contributions:

1. A novel approach to region-based JIT code generation, that involves keeping region-local basic blocks internal, and aggressively optimising across these blocks.
2. The exploitation of **branch type profiling information** to improve back-end code generation (by means of **loop optimisations**, for example).
3. Employing two caches with **different translation granularities** to implement efficient and light-weight region chaining.
4. A new **domain-specific alias analysis** that allows more accurate separation of independent memory accesses, enabling improved back-end code generation.

### 4.1.4 Overview

The remainder of this chapter is structured as follows:

- **Section 4.2** builds upon the background of the DBT system ARCSIM introduced in Section 3.2 and, in particular, the region selection scheme used throughout this chapter.
- **Section 4.3** describes the novel code generation strategy used to achieve high performance.
- **Section 4.4** presents an empirical evaluation of an implementation of the techniques developed in Section 4.3.
- **Section 4.5** summarises and concludes the chapter.

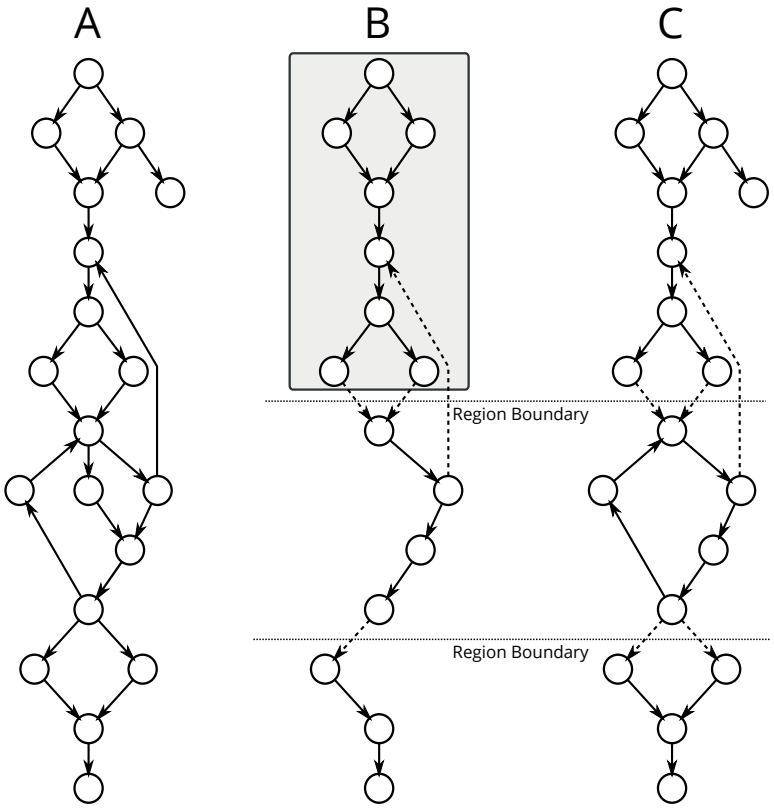



**Figure 4.4:** Ⓐ Example of a (static) whole-program control flow graph. Ⓑ Parts of the control flow graph from Ⓐ dynamically discovered after some time of execution, including region limits at page boundaries. Ⓒ Additional control flow has been dynamically discovered after some more time executing the program.

## 4.2 Background

### 4.2.1 Region Compilation

As previously described in Section 3.2, a JIT worker thread is responsible for translating a region into native code. It dequeues a *translation work unit* from the work queue and builds an LLVM function that represents the region to be translated, passes it through the LLVM optimiser and finally lowers it to native machine code. During translation, the compiler uses profiling information passed in with the translation work unit to make decisions about how to generate the LLVM IR that represents the basic blocks within the region.

The translation work unit consists of a list of basic blocks to compile (which represents the discovered basic blocks within the region), the associated control-flow graph connecting those basic blocks together, and a list of the blocks which are *region entries*. The compiler then translates each block in turn (on an

instruction-by-instruction basis) into LLVM IR. Finally, when each basic block has been translated, a *local jump table* (sometimes also referred to in literature as an *indirect branch target buffer* (IBTB)) is generated, which contains the addresses of each basic block that is a *region entry*, and each block that has been observed by the profiler to be the target of an indirect jump.

The *region prologue* is a small piece of set-up code common to each *region function*, which loads values that are reused throughout the native code (such as pointers to the various CPU state structures). Following this setup, an indirect branch via the previously generated *local jump table* is performed to begin execution at the desired basic block. A *region function* therefore, contains the translated native code for every block discovered (and marked as hot) in the region, and invoking this function will branch to the block that is to be executed, by accessing the *program counter* from the CPU state structure.

In Figure 4.4, the control-flow graph labelled $A$ describes the static control-flow of the target program, where $B$ and $C$ show the *discovered* control-flow, along with region boundaries. The shaded portion of $B$ is magnified in Figure 4.5, which shows how blocks within a region are compiled to a region function, and how the function chains to other region functions by means of the *global jump table*.

### 4.2.2 Region Selection

In a DBT system, region selection is concerned with forming the *shape* of translation units, where a region is typically a collection of basic blocks connected by control flow edges. This stage follows code discovery and profiling, and it determines the boundaries of a fragment of recently discovered guest code which is then prepared for translation into native host code. A number of region selection schemes for use in JIT compilers and DBT systems have been developed (e.g. Bruening and Duesterwald [26], Hiniker et al. [55], Hiser et al. [57], Hsu et al. [62]), but the focus of these papers has been on *policies* for region selection, i.e. decisions on how far and for how long to grow a region and they do not explore code generation strategies for regions. Often regions are distinguished from *traces*, whilst technically traces are degenerate regions they are often treated separately due to their linear shape, i.e. the absence of multiple control flow successors and, in particular, loops.

JIT compilers present in e.g. Java VMs would have *meta-information* about the structure of the program being executed, and could use this information for *method*-based region selection techniques. But, the presence of this meta-information is not guaranteed and cannot be relied upon, and indeed is not present in a raw instruction stream, so the DBT must rely on dynamic profiling information to effectively perform region selection [126]. In this chapter, a page-based region selection scheme is used, similar to the one presented by Böhm et al. [24]. Such a scheme enables efficient MMU emulation and detection of self-modifying code through page protection mechanisms provided by the host OS.

As shown in Figure 4.4 Ⓑ a dynamic CFG is built, and basic blocks are inserted with corresponding control flow edges between wherever dynamic control flow is encountered. After a certain interval (in terms of number of basic blocks executed in the interpreter) the CFG is scanned and regions are formed, depending on their *heat*, and whether this is above a certain, adaptive threshold. In this scheme, page boundaries are also compulsory region boundaries. Regions are then passed to the JIT compiler for code generation, and profiling execution continues, possibly extending the dynamically discovered CFG further (see Figure 4.4 Ⓒ).

## 4.3 Methodology

### 4.3.1 Region Entry Optimisation

As described in the motivating example (Section 4.1.2), allowing arbitrary entry to a region via any basic block hinders the ability of the code optimiser to produce efficient host code. Therefore, to improve code quality, only basic blocks that need to be visible externally should be registered in the *local jump table*. These externally visible basic blocks are termed *region entries*, and are identified during the profiling phase. Keeping other blocks internal allows LLVM to be more aggressive during the optimisation, phase—potentially even merging basic blocks together and performing *inter*-block optimisations.

During construction of the LLVM region function, initially an LLVM basic block is produced for each guest basic block that is part of the region profile. If the profile indicates that the block is a region entry, then a corresponding LLVM

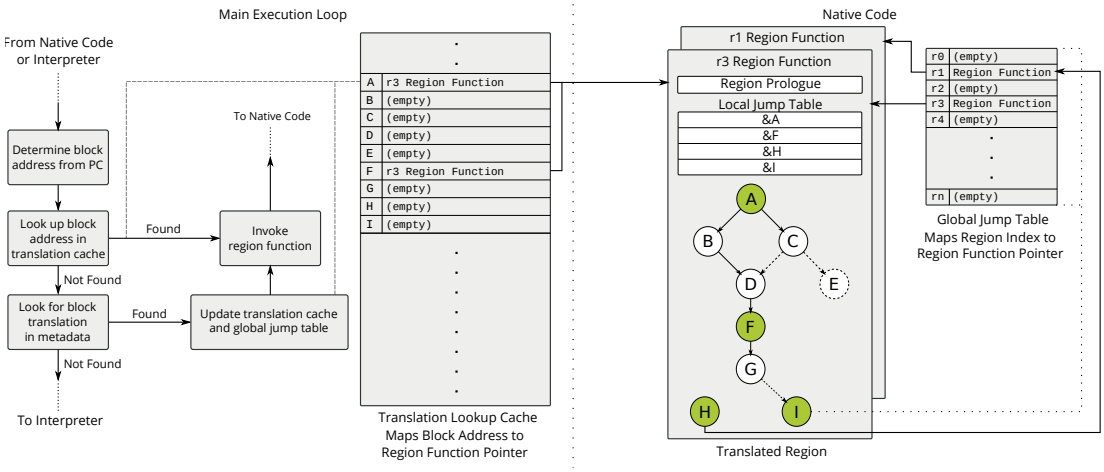



**Figure 4.5:** Interaction between regions via the *global jump table* and the internal interactions between basic blocks, either directly or via the *local jump table*. The control flow graph represents the region in the shaded area in Figure 4.4 Ⓑ.

block address is obtained, and inserted into the *local jump table*.

### 4.3.2 Translation Lookup Cache

The *translation lookup cache* is a structure that lives in the execution engine component of the DBT and is used to resolve addresses of guest basic blocks to native code. In fact, it is a mapping of block addresses to the region function that contains the native translation of a particular block. Only *region entry* blocks are entered in to the translation lookup cache, as it is only possible to branch to *region entry* blocks from the *local jump table*.

### 4.3.3 Branching

A basic block is defined as a single-entry, single-exit linear code sequence, and as such the *terminating instruction* is always a branch to one or more basic blocks. There are two types of branches that can be made out of a basic block:

1. **Direct**: A branch whose destination is known at JIT compilation time, i.e. the destination is a PC-relative or absolute address.

2. **Indirect**: A branch whose destination is not known at JIT compilation time, i.e. the destination address is calculated from the value a guest register.

These two cases can further be classified as *predicated* and *non-predicated*, which impose additional constraints on the control-flow out of a basic block. When a branch is *predicated*, the fall-through block for the *branch not taken* case can be treated as a *direct* branch to the following instruction, as the fall-through address can be trivially computed at compilation time.

In Figure 4.5, each node in the CFG (except for $E$) has been discovered by the profiler, and as such the CFG has been compiled to LLVM IR on a block-by-block basis. Node $E$ and the corresponding edge $\overrightarrow{CE}$ have not yet been discovered by the profiler, i.e. they have not yet executed, or have not exceeded the compilation threshold.

Nodes $A$ and $F$ are *region entries*, and $H$ and $I$ are the targets of indirect branches. As such, these nodes have their block addresses taken, and a corresponding entry added to the *local jump table*. The other nodes are never accessed by an indirect jump (as far as the current profiling information is concerned) so their block addresses are not taken, and no entry is registered in the *local jump table*.

This leads to the case where native code may be available for a basic block, i.e. it has been compiled, but it is not reachable from outside the region.

#### 4.3.3.1 Direct Branches

Where there is a direct branch from basic block $A$ to $B$, (and $B$ has no indirect branch predecessors), it is not required to add the address of $B$ to the local jump table and instead LLVM IR can be emitted to perform a direct branch to $B$.

There are two approaches that can be taken when generating the proper control-transfer sequence, and they depend on whether or not the terminating branch is predicated or non-predicated.

For a *non-predicated branch*, given the jump target is known at compile time, if the target lies outside the region boundary, code is generated that transfers control via the *global jump table*—as shown by node $H$. This means it is possible to chain directly to the region containing the destination block, if it is available. If the target lies within the region, as shown by node $A$, then if that particular block is present in the current work unit, LLVM IR is emitted that directly branches to it. If the destination block is not in the work unit, then an

exit sequence is emitted that returns immediately to the interpreter, as a native translation is not available in this round of compilation.

For a *predicated branch*, the same sequence applies as before, except it must be determined whether or not the branch is to be taken. If the branch is not taken, then the fall-through block is directly branched to (if present in the work unit).

#### 4.3.3.2 Indirect Branches

Naturally, at JIT compilation time, the destination of an indirect branch is not known, since it depends on the run-time value of a guest register. However, the profiling phase can provide insight into possible destinations and by considering the previously seen destinations of an indirect branch, it is possible to speculate where the branch may land. Indirect branch instructions that are predicated can be considered to have a direct edge to the fall-through block, and may be treated as in the direct branch case (Section 4.3.3.1).

If there is no edge information for the branch, then control must be transferred via the *global jump table*. This is demonstrated in Figure 4.5 as node $I$, and is because it is known that the *local jump table* cannot satisfy the jump (since an entry would only be available if that particular edge was encountered by the profiler). Exiting via the *global jump table* is required because the indirect branch may be to a different region. If it turns out that this speculation is incorrect (or if the destination region does not contain a translation for the target block), control is returned to the interpreter.

If the edge information contains exactly one edge, then a simple comparison instruction is emitted to determine whether or not that edge should be taken. If the edge is correct, a direct branch to that basic block occurs, otherwise control falls back to the *global jump table*.

Node $C$ (before discovery of $E$) is an example of this, where there is a single indirect edge $\overrightarrow{CD}$, but $\overrightarrow{CE}$ has not (yet) been discovered.

Finally, for a block with multiple indirect successors (such as node $G$), code is emitted to check that the target block lies within the same region, and if so an indirect branch via the *local jump table* is performed. If the target block lies outside the current region, control is transferred via the *global jump table*.

Other implementations of *local jump tables* are possible, e.g. some of the

techniques presented in Hiser et al. [58], Koju et al. [75], Jia et al. [67], Yin et al. [128], Dhanasekaran and Hazelwood [38] could act as drop-in replacements, however, this implementation has been found to provide sufficiently low lookup times and high hit rates.

### 4.3.4 Region Chaining

Chaining is becoming a common feature in trace based JIT compilation systems, such as in the Dalvik VM [42] and TraceMonkey [43]. This technique typically involves profiling execution flow between compiled traces, and updating the translated code for hot edge source nodes of inter-trace jumps, to jump directly to the destination translation unit. In this strategy, *trace chaining* is extended to *region chaining*, which deals with hot control flow between regions. This can be the result of hot inter-region edges emerging only after some warm-up time, where region selection has already partitioned code into regions, or due to unavoidable region limits such as page boundaries introduced by the region selection scheme (see also Section 4.2.2).

To simplify code generation a weak form of region chaining is implemented, where a *global jump table* tracks the native code of translated regions. It is important to distinguish this from the translation lookup cache—the *global jump table* is only a jump table at region (page) granularity and is not used when transitioning from the interpreter into native code. Conversely, the translation lookup cache contains translation information at basic block granularity and is only used when transitioning from the interpreter to native code.

The global jump table contains one entry (initially empty) for each possible region. Each entry consists of a single function pointer. In ARCSIM, there is at most one region per guest memory page, and given that in this example the guest platform is 32-bit, the global jump table contains $4GB/4kB = 1,048,576$ entries, and so is 8MB in size. These entries are updated when a miss occurs in the translation lookup cache described above. When a region is retranslated, the corresponding translation cache entry for that region is invalidated, ensuring that the *global jump table* always points to the most up-to-date translation for each region.

The *global jump table* is used when it is determined that a translated branch might have another region as its destination. This determination is made dif-

ferently depending on the branch type information:

1. For a **direct branch**: if the target is outside the current region, then the global jump table is used if the branch is taken.
2. For an **indirect branch**:
   i) If **no targets** within the current region have been encountered so far: the global jump table is used immediately.
   ii) if **one or more targets** within the current region have been encountered: if the branch resolves to an address within the current region, then the *local jump table* is used, otherwise the *global jump table* is used.

Since the global jump table is initialised with blank entries, the requested entry must be checked before it is used (essentially a null-pointer check), and if the requested entry is null, execution flow leaves translated code and returns to the interpreter.

### 4.3.5 Region Registration in Translation Caches

Every basic block that is encountered by ARCSIM has metadata held about it, which describes certain properties about the block (e.g. whether or not the block is a region entry), and contains a pointer to the region function containing its implementation (if it has been identified as a region entry). When the execution engine begins executing a block, it first looks up the block metadata and checks to see if a native translation exists—if so, the translation cache is updated and native code is entered. Additionally, the *global jump table* is updated with a pointer to the function for the region containing the block. If a region is recompiled, the block metadata will be updated to reflect the new function pointer and the change would propagate through to the translation lookup cache.

### 4.3.6 Continuous Profiling and Recompilation

The mixing of instructions and data, and the presence of indirect branching make it impossible to fully and accurately determine the precise control flow of

a program from machine code only. Although techniques exist which attempt to extract control flow information from programs statically [73] these often must be extremely conservative and thus DBT systems using them suffer from poor performance.

On the other hand, techniques for extracting control flow information at run time are becoming increasingly effective [70]. These techniques often do not capture all possible control flow paths through a program in their first pass – thus, it is necessary to profile continuously.

It is therefore possible to discover new control flow within regions which have already been translated and compiled. If the relevant regions are not retranslated when such control flow is encountered at run time, it can only be evaluated sub-optimally. For example, a block which was previously excluded from the region *local jump table* may in fact be a region entry. In this case, control is returned to the interpreter to execute this block, since a translation cache entry does not exist.

This technique does not require any special treatment for the retranslation of regions. Instead, the profiling system does not distinguish between already translated and non-translated regions. If previously untranslated code or control flow is encountered in a translated region, it is executed using the interpreter and profiled. If it is frequently executed and becomes hot, the full region will be retranslated in order to include the new code and control flow.

### 4.3.7 Host Machine Code Generation

A *translation work unit* is the unit provided to a JIT compiler worker thread and consists of a list of *basic block descriptors*, along with basic block edge information, representing a particular region. The *basic block descriptors* contain a list of decoded instructions. Each instruction in a block is translated to LLVM IR one-by-one, using a technique similar to Wagstaff et al. [121] and once the instructions have been translated, a *block epilogue* is emitted. This epilogue is generated based on the type of control-flow associated with the block (as described in the previous sections), and essentially consists of the LLVM IR that transfers control to the next block.

Finally, after all the blocks in the *translation work unit* have been compiled, and the region prologue has been generated, a single LLVM function remains

that represents the region just compiled. This function is then passed through the LLVM optimiser, as described in Section 4.3.7.1.

After the optimisation passes have completed, the LLVM IR is compiled to native machine code using the LLVM JIT compiler interface, and when the native code is available, each basic block that is marked as a *region entry* has a pointer to the newly compiled function stored in its metadata.

#### 4.3.7.1 LLVM Optimisation Passes

During the translation phase, an LLVM module is built containing the function that represents the region being translated. The module also contains *helper functions*, which are highly amenable to inlining. All the helper functions are marked as internalisable, and an inlining pass is applied. Typically, the helper functions will provide a very small function (such as reading the `PC` register, or writing to target machine memory), and are easily inlined.

After inlining, the resulting module is subjected to a number of LLVM passes, based on the standard Clang `-O3` optimisation level. The main difference is that instead of using an LLVM provided alias analysis implementation, a specialised implementation (described in Section 4.3.7.2) is employed.

Since some basic blocks are not region entry points, this has opened up more scope for aggressive loop optimisation, which yields the full benefit of a region-based DBT. With a trace-based DBT, loop optimisations rarely happen, as traces are inherently linear. However, with the region-based approach, a significant amount of loop optimisations can be performed across the entire control-flow within a region that would not be possible if entry was allowed to the region from any basic block.

#### 4.3.7.2 Alias Analysis

Alias analysis of pointers is an important phase that enables further program optimisations to reason better about data flow. For example, a *dead store elimination* pass uses pointer aliasing information to determine whether or not a redundant store to a memory location can be eliminated, based on any memory accesses that happen between those stores.

Listing 4.3 shows how incomplete pointer aliasing information can lead to the optimiser being unable to remove dead stores. The stores on lines 1 and

**Listing 4.3**

```
1 store i32 36076, i32* %4
2 %42 = load i64* inttoptr (i64 61931224 to i64*)
3 %43 = add i64 %42, 6
4 store i64 %43, i64* inttoptr (i64 61931224 to i64*)
5 store i32 36076, i32* %4
6 ...
7 store i32 36092, i32* %4
```

**Listing 4.4**

```
1 movl $37076, 60(%r12)
2 addq $6, 61931224
3 movl $37076, 60(%r12)
4 ...
5 movl $36092, 60(%r12)
6
7
```

**Figure 4.6:** Remaining dead-stores in LLVM IR (Listing 4.3) after optimisation, and resulting x86 machine code (Listing 4.4) due to incomplete alias analysis.

5 are killed by the store on line 7, but because the optimiser cannot detect that the operations on pointers in lines 2-4 do not alias, it cannot remove the stores. This directly translates to machine code as shown in Listing 4.4, which is safe (and correct), but in this case not at all optimal and severely impacts performance.

In the example shown in Figure 4.6, the problem stems from the alias analysis implementation (quite correctly) being unable to determine whether or not the pointer held in `%4` aliases with the constant pointer value `61931224`. Assuming that `%4` and `61931224` alias is a safe assumption and as such generates safe code. But, armed with the knowledge about the implementation of ARCSIM, it is known that `%4` contains a pointer to an emulated CPU register (present in the CPU state structure), and that the constant pointer is an address that does not intersect with the CPU state structure. Hence, it can be said that they do not alias. Providing this guarantee to LLVM's *dead store elimination* optimisation pass enables the pass to remove the redundant stores, and generate better code. The particular example described above is important for region-based compilation, as redundant updates to the CPU state are eliminated, hence reducing the number of memory operations occurring in a particular sequence and improving execution throughput.

When a loop is involved, keeping target machine register values in host registers instead of constantly reading and writing to the CPU state structure improves performance significantly—but this kind of loop optimisation can only work to its full potential when combined with the *jump table optimisation* technique described in Section 4.3.3.

| Vendor & Model | Dell™ PowerEdge™ R610 |
|---|---|
| Architecture | x86-64 |
| Processor Model | 2× Intel© Xeon™ X5660 |
| Number of cores | 2× 6 |
| Clock Frequency | 2.80 GHz |
| FSB Frequency | 1.33 GHz |
| L1-Cache | 2× 6× 32K |
| L2-Cache | 2× 6× 256K |
| L3-Cache | 2× 12 MB |
| Memory | 36 GB |
| Operating System | Linux 2.6.32 |
| User Space | Scientific Linux 6.6 |

**Table 4.1:** Host configuration.

| DBT Parameter | Setting |
|---|---|
| Target architecture | ARMv5T |
| Host architecture | x86-64 |
| Translation Model | Asynchronous |
| Tracing Scheme | Region-based [24] |
| Tracing Interval | 30000 blocks |
| Translation Cache | 8192 Entries |
| JIT compiler | LLVM 3.4 |
| Compilation Threads | 10 |
| IR Generation | Part. Eval. [121] |
| JIT Optimisation | `-O3` |
| JIT Threshold | 20 (Adaptive [24]) |

**Table 4.2:** ARCSIM configuration.

## 4.4 Experimental Evaluation

This section evaluates the DBT/JIT code generation approach using the SPEC-CPU2006 integer benchmark suite, with its *reference* input set. This benchmark suite is widely used and considered to be representative of a broad spectrum of application domains. The benchmarks have been compiled using the gcc 4.6.0 C/C++ cross-compilers, targeting the ARMv5T architecture (without hardware floating-point support) and with `-O2` optimisation settings.

### 4.4.1 Experimental Methodology

The elapsed real time between invocation and termination of each benchmark in ARCSIM has been measured using the UNIX `time` command on the host machine described in Table 4.1 with ARCSIM configured as in Table 4.2. The dynamic guest instruction count (which is invariant) and the average elapsed wall clock time across ten runs of each benchmark is used to calculate execution rates (using MIPS in terms of target instructions) and speedups. For summary figures, the harmonic mean weighted by dynamic target instruction count is presented. For the comparison to the state-of-the-art, the ARM target of QEMU 1.4.2 is used as a baseline.

Additionally, ARCSIM's performance is evaluated using the EEMBC-1.1 benchmark suite. These benchmarks are typically shorter running and serve to evaluate the performance of the JIT compiler portion of the DBT system. In order

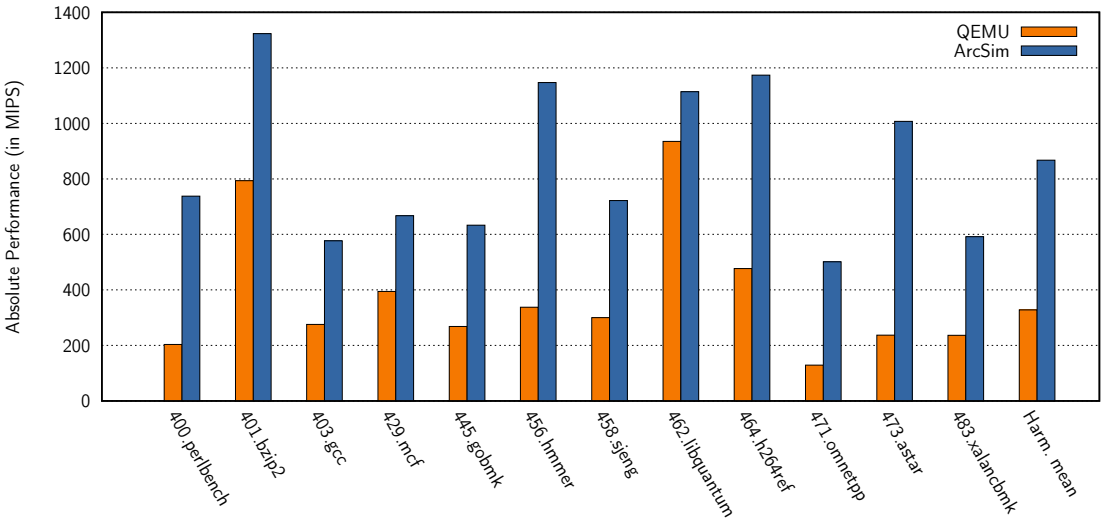



**Figure 4.7:** Absolute performance figures (in MIPS) for the *long-running* SPEC-CPU2006 integer benchmarks for both QEMU-ARM and ARCSIM—higher is better. In all cases, ARCSIM outperforms QEMU.

to normalise performance to a particular duration, the iteration count of each benchmark was adjusted so that it ran for approximately ten seconds in QEMU, then the benchmark was invoked with the same iteration count in ARCSIM, and performance was measured in the same manner as for SPEC.

### 4.4.2 Experimental Results for SPEC-CPU2006

Figure 4.7 gives an overview of the absolute performance of QEMU vs. ARCSIM. In every case, ARCSIM improves over QEMU, and on average achieves a $2.64\times$ improvement in absolute performance.

The biggest improvement is achieved for 473.astar, which can be attributed to the benchmark responding well to the ability to apply loop optimisations within a region. The relative performance improvement of 473.astar when *region chaining* is enabled is negligible, and so indicates that the majority of time is spent in region local code. Aggressive loop optimisations are performed within this region (where the bulk of the algorithm lies). This explains the excellent performance improvement over QEMU, which performs no such optimisations. This explanation can also be applied to 464.h264ref, which also benefits greatly from the ability to optimise loops better than QEMU.

The smallest improvement is for 462.libquantum, which may be due to the benchmark itself being heavy in arithmetic instructions, but not so much in

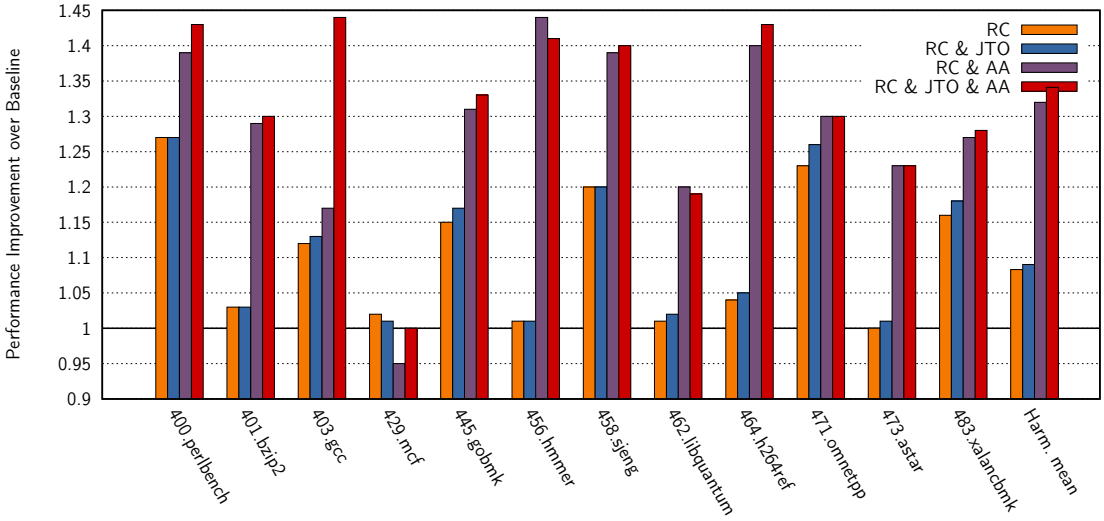



**Figure 4.8:** A breakdown of the performance impact of different optimisations. The baseline is standard LLVM `-O3` and partial evaluation [121] at JIT compilation time. Additional region chaining (RC), jump table optimisation (JTO) and alias analysis (AA) complement each other.

looping constructs. This particular characteristic explains the excellent performance of QEMU, and hence why there is only a $1.2\times$ improvement in this case. QEMU's block-based optimisations work well here, due to the linear nature of the arithmetic instructions and larger basic block sizes.

Interestingly, the relative performance improvements as optimisations are enabled (shown in Figure 4.8 and detailed in Section 4.4.3) of 462.libquantum are similar to that of 473.astar, and the absolute performance of both the benchmarks are within the same area - but 462.libquantum is already fast in QEMU.

### 4.4.3 Impact of Optimisations

Figure 4.8 shows how combinations of the strategies described in Section 4.3 affect the relative performance of ARCSIM. The baseline is using standard LLVM `-O3` optimisation and partial evaluation, but without any of the optimisations described in this chapter applied.

Overall, the addition of custom alias analysis improves every benchmark, except for 429.mcf. On average this gives a $1.32\times$ performance improvement, but it is the combination of all the strategies that yield the best result. *Jump table optimisation* on its own does not give rise to a significant performance improvement, but responds well when combined with alias analysis. This may

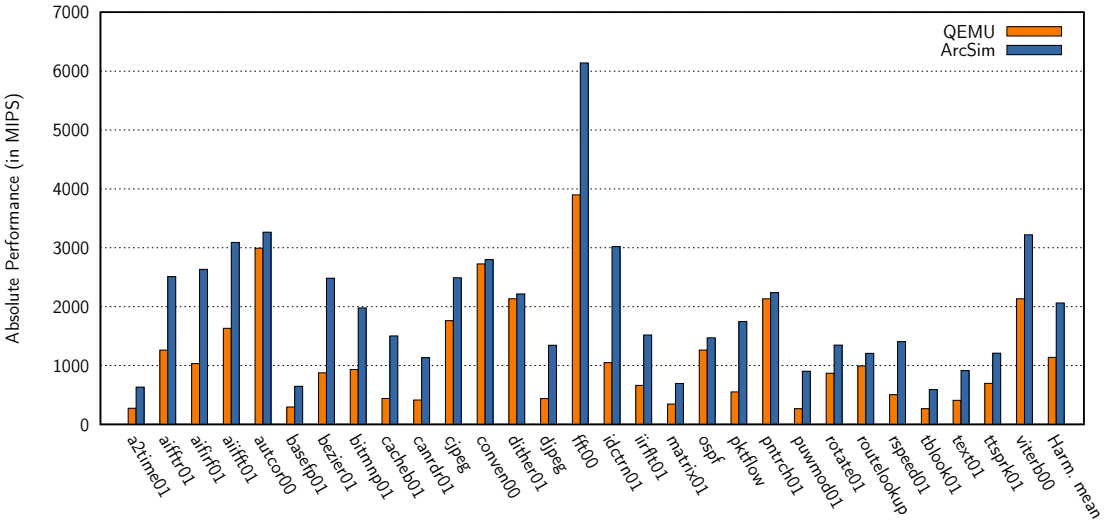



**Figure 4.9:** Absolute performance figures (in MIPS) for the *shorter-running* EEMBC benchmarks for both QEMU-ARM and ARCSIM, indicating that JIT startup time and compilation performance of ARCSIM is more than competitive with QEMU-ARM, despite the use of aggressive code optimisations.

be due to the fact that the most interesting optimisation to apply across basic blocks is to remove dead stores and to keep host registers live with frequently used values (potentially from the CPU state structure). Without the precise aliasing information this kind of optimisation is not possible to do effectively, and so the combination of both *jump table optimisation* and *custom alias analysis* gives rise to the best performance improvements.

473.astar remains at baseline performance when the *region chaining* optimisation is applied, and this may be due to the majority of execution being spent in region-local code. It has an absolute performance figure of $> 1000$ MIPS, which indicates fast running code, but the benefits of *region chaining* are minimal, due to the lack of inter-region control-flow.

403.gcc is a particularly control-flow heavy benchmark, and responds well to the combination of all the optimisations together. Also of interest is the 429.mcf benchmark, which does not consistently improve in performance like the majority of the other benchmarks. Despite this, 429.mcf is more than $1.5\times$ faster in ARCSIM than in QEMU.

### 4.4.4 JIT Compilation Performance

The execution time of the SPEC-CPU2006 benchmarks with their *reference* data sets is dominated by the time spent executing native code, whereas the fraction accounted for JIT compilation time is small. For such long-running benchmarks *code quality* is paramount and this is where region-based code optimisations outperform simpler basic block or trace based schemes. However, JIT compilation time is still important for shorter-running applications, or programs that exhibit phased behaviour and, hence, exercise the JIT compiler more heavily.

To evaluate JIT compilation performance of ARCSIM, results are shown for smaller, shorter running benchmarks, where time for JIT compilation constitutes a larger portion of the overall execution time (see Figure 4.9). In every case, ARCSIM beats QEMU in absolute execution performance, but as in the SPEC results, the relative performance improvements vary greatly. As can be seen, the most significant result here is that fft00 is executing at a rate of $6138$ MIPS, compared to QEMU's $3897.95$. However, this only shows a modest relative performance gain of $1.5\times$, whereas idctrn01 outperforms QEMU by $2.85\times$. These variances can again be attributed to the characteristics of individual benchmarks in the suite, where benchmarks that are amenable to loop optimisations, i.e. contain more intra-region loops, show a greater *relative* performance improvement.

Overall, these results demonstrate that even for shorter-running applications where JIT compilation latency plays a greater role than absolute code quality, ARCSIM is highly competitive despite its use of larger translation units and aggressive code optimisations.

## 4.5 Summary & Conclusions

This chapter has presented a novel, integrated approach to JIT code generation within region-based DBT systems. Branch type information is exploited to optimise end-of-block control flow, region chaining is introduced to improve control flow between code regions, selective region registration in translation caches is developed to improve intra-region code generation, continuous profiling and recompilation is employed to produce more optimal native code representative of the behaviour of the target program, and finally custom alias analysis is em-

ployed to enable aggressive code optimisations, which would not be possible in a DBT scheme based on linear traces. The region-based JIT code generation approach is implemented in ARCSIM, and is evaluated using the SPEC-CPU2006 benchmarks, compiled for the ARMv5T ISA. In comparison to state-of-the-art QEMU, an average speedup of $2.64\times$ is achieved, and up to $4.25\times$ for individual benchmarks. Each of the techniques developed in this chapter on their own contributes to increased code quality, but it is the particular combination of code generation steps that results in performance improvements greater than the sum of its parts.

This chapter has been concerned with efficient instruction emulation, but without considering the additional challenges present in full hardware virtualisation. The next chapter shall extend ARCSIM with support for hardware virtualisation, and describe a particular challenge that can directly affect the performance gains made by the techniques described in this chapter.

Chapter 5

# Efficient Interrupt Virtualisation

As presented in the previous chapter, emulating guest instructions is necessary for cross-architecture virtualisation, and is generally sufficient for the straightforward execution of a guest binary on a host system (i.e. user-mode simulation). However, supporting *hardware virtualisation* requires extended capabilities that go beyond the simple execution of a stream of user-mode instructions. Cross-architecture hardware virtualisation needs to support additional guest architectural features including emulating a *memory management unit* (MMU), handling the operation of privileged system instructions, emulating platform devices and handling asynchronous interrupts—all of which are necessary for the virtualisation of a complete system capable of hosting an unmodified *operating system* (OS). *Asynchronous interrupts* (e.g. those raised by a timer device) present a challenge to efficient DBT, as they introduce *adverse control-flow*, which unpredictably diverts the current execution path of the emulated processor. This chapter develops a strategy to mitigate the performance impact of interrupt handling for DBT-based cross-architecture hardware virtualisation.

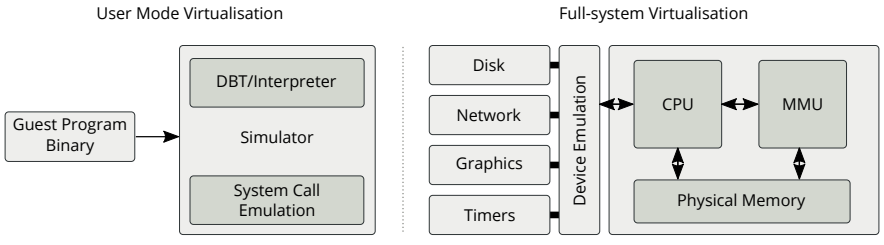



**Figure 5.1:** User-mode simulation only requires the emulation of guest instructions, and an OS system-call emulation layer. Cross-architecture hardware virtualisation requires emulating guest platform devices, supporting privileged system instructions, implementing memory management units and handling other architectural nuances.

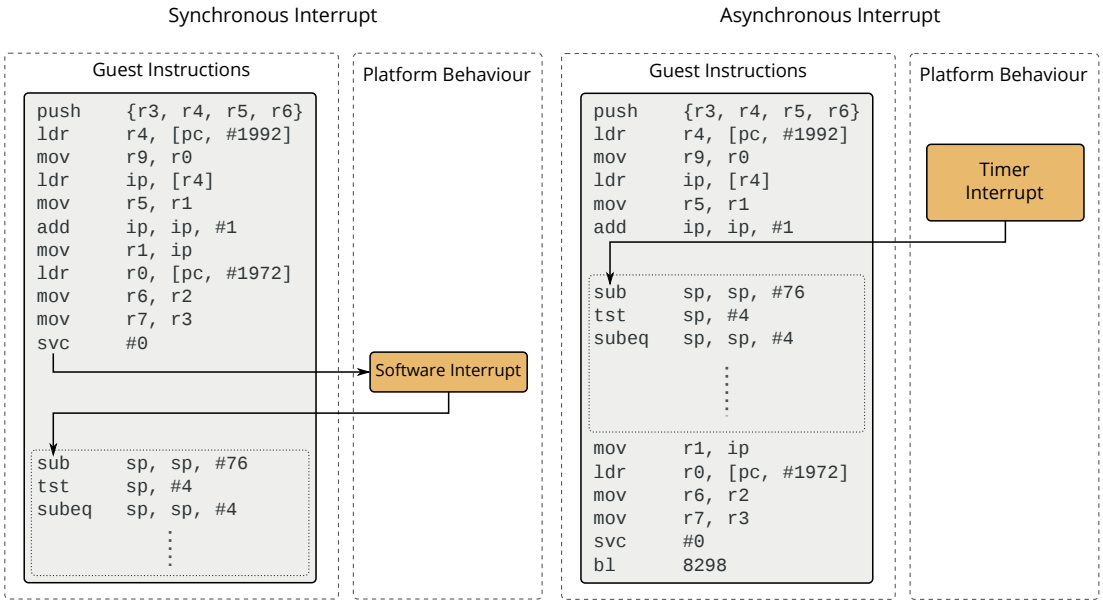



**Figure 5.2:** An extract of ARM machine code that demonstrates the behaviour of a synchronous interrupt in the form of a system call instruction (`svc`), versus an asynchronous interrupt that is invoked by an external timer device. Synchronous interrupts happen predictably, based on the currently executing instruction, whereas asynchronous interrupts happen unpredictably at any time during execution.

## 5.1 Introduction

In operational terms, it is relatively straightforward to build an *instruction set simulator* (ISS) that will run a user-space program compiled for one *instruction set architecture* (ISA) on another. These ISSs are termed *user-mode simulators*, as they simply execute a stream of user-mode instructions and emulate system calls by providing an OS emulation layer. This is acceptable for running programs that are normally run inside an OS, but if the goal is to run an entire, unmodified guest OS, a simple ISS is no longer sufficient and *hardware virtualisation* is required.

Operating systems naturally expect to be running on top of real hardware, and subsequently expect that hardware to behave in a particular way, e.g. the OS will control the MMU to implement virtual memory systems, provide abstractions for hardware devices, handle interrupts coming in from those devices, manage TLBs and take care of other architectural concerns that a user-mode program generally has no knowledge of. To be able to support this, the virtualisation environment must emulate these architectural features faithfully.

A particular feature that requires attention when virtualising hardware is *asynchronous* interrupts. These are generally raised by devices at unspecified times. As shown in Figure 5.2, a *synchronous interrupt* (or *exception*) is the diversion of control-flow to an interrupt handler that is expected, e.g. a guest instruction that invokes a software interrupt, or a divide instruction that *may* raise a divide-by-zero exception. These synchronous interrupts can be easily handled, as it is known at instruction execution time that control-flow may diverge. However, *asynchronous interrupts* may be raised at any time, during any instruction, and without warning.

For an interpreter-based hardware virtualisation system, this is generally not a problem. After each instruction has executed, the system can check to see if an *interrupt request* (IRQ) is pending, and if so, instead of progressing to the next instruction, the necessary platform-specific behaviour will be invoked to divert control-flow to the interrupt handler. For DBT-based systems, however, the situation becomes more complex as facilities to handle pending IRQs must be built into translated code.

The efficiency of interrupt handling is of particular importance as interrupts need to be processed frequently and require a fast response time. Unfortunately, efficient interrupt handling is at odds with the region-based DBT presented in the previous chapter, as interrupts interfere unpredictably with the "natural" control flow of an application and divert it away from the current region of code to another. To capture this behaviour, additional checks need to be inserted into translated code, which initiate an interrupt handling sequence if an IRQ is pending. These additional checks are costly to perform and can inhibit aggressive region-based optimisations, resulting in a reduction of virtualisation performance by more than an order of magnitude, if inserted naïvely e.g. after each guest instruction. Typically, translated guest instructions will map to multiple host instructions, meaning that it is not possible to arbitrarily divert control-flow when executing native code. This is because the CPU state may be left inconsistent if a guest instruction is only partially executed, and guest instructions must appear to be atomic. Thus, the minimum granularity for interrupt checking is a single guest instruction.

By their very nature, pending IRQs can be deferred by a small period of time until a more "convenient" moment. For example, an OS might mask certain interrupts during critical sections and only process pending IRQs after leaving

such a section. This particular trait can be exploited to reduce the number of interrupt checks inserted into generated code, thus reducing the overall performance impact. The central questions being answered in this chapter are:

- *What* is the *minimum* number of interrupt checks that need to be inserted to maintain correctness?
- *Where* in generated code should interrupt checks be inserted?

### 5.1.1 Key Idea

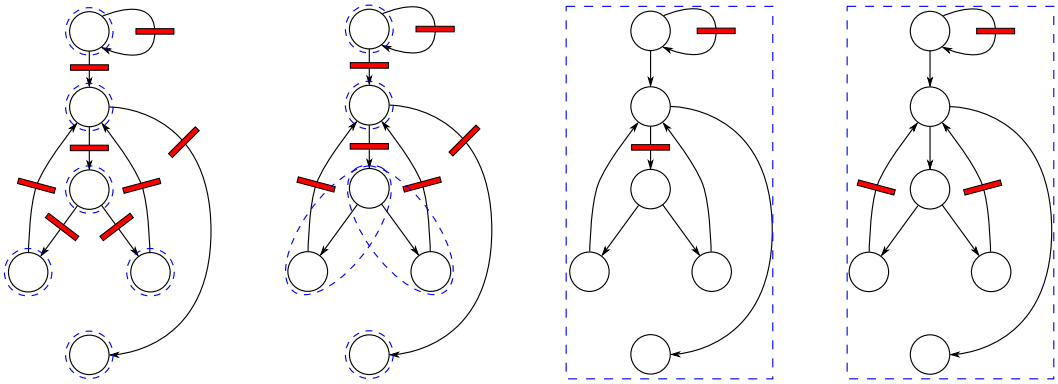
This chapter presents a new scheme for *interrupt check placement*, that indicates to a JIT compiler where asynchronous interrupt checks should be inserted into translated code. To evaluate this, ARCSIM is extended from user-mode simulation to support hardware virtualisation, and this scheme is implemented as part of the region-based DBT strategy presented in Chapter 4. The key idea can be summarised as follows:

- During the profiling and compilation phase, control flow loops within a region are identified, and interrupt checks are inserted within each control flow cycle to maintain correctness.

This is important for loops which depend on interrupt handling for their termination. Whilst inserting a strictly minimal number of interrupt checks is an NP-hard problem [32], an existing approximation algorithm that is suitable for use in a performance-critical JIT environment is used instead, which for most practical cases computes an almost optimal solution.

### 5.1.2 Motivating Example

A *basic block*-based DBT is one which translates *guest* basic blocks one-at-a-time into corresponding *host* code (see Figure 5.3a). These blocks are treated as independent units, and control-flow is performed by jumping to existing code in a cache, or causing an on-demand translation if the code has not yet been seen. Extending a basic block-based DBT to consider multiple blocks along a path leads to a *trace*-based DBT, which allows for optimisations to cross basic block boundaries, as the trace is considered its own unit (see Figure 5.3b). Whilst trace-based DBTs only consider linear control flow, a *region*-based DBT



**(a)** Block-based **(b)** Trace-based **(c)** Region-based (optimal) **(d)** Region-based (suboptimal)

**Figure 5.3:** Interrupt checks, represented by bars on control flow edges, inserted by various interrupt check placement schemes in the control flow graph representing the example program from Figure 5.5. Translation units (basic blocks, linear traces, regions) are highlighted.

can exploit cyclic control-flow (such as loops) within a particular region of code (see Figures 5.3c and 5.3d).

User-mode simulation of applications requires only that a target binary is emulated on the host, and does not usually depend on interrupts or device emulation. Generally, a simple OS emulation layer is enough to execute most user binaries. Notable exceptions are applications that utilise *asynchronous* Unix signals, but the majority of benchmarks (and certainly those present in the SPEC-CPU2006 suite [54]) do not depend on this behaviour and so are ill-suited for testing this important requirement of hardware virtualisation. Figure 5.4 shows that in user-mode simulation, where interrupt checks can be ignored, ARCSIM performs on average $2.23\times$ faster than QEMU [21], also in user-mode configuration.

Region-based DBTs blur the mapping between dynamically discovered *guest* basic blocks and translated *host* basic blocks, as some optimisations may involve merging or splitting the guest basic blocks to improve control-flow and enable further cross-block optimisations. Normally, these kind of optimisations are not a problem when a region of code is considered as a unit, with only a few entry and exit points into and out of the region—the optimiser is free to do any kind of transformation provided that upon region exit, the CPU state is correct. However, it is possible to hinder these optimisations by inserting additional

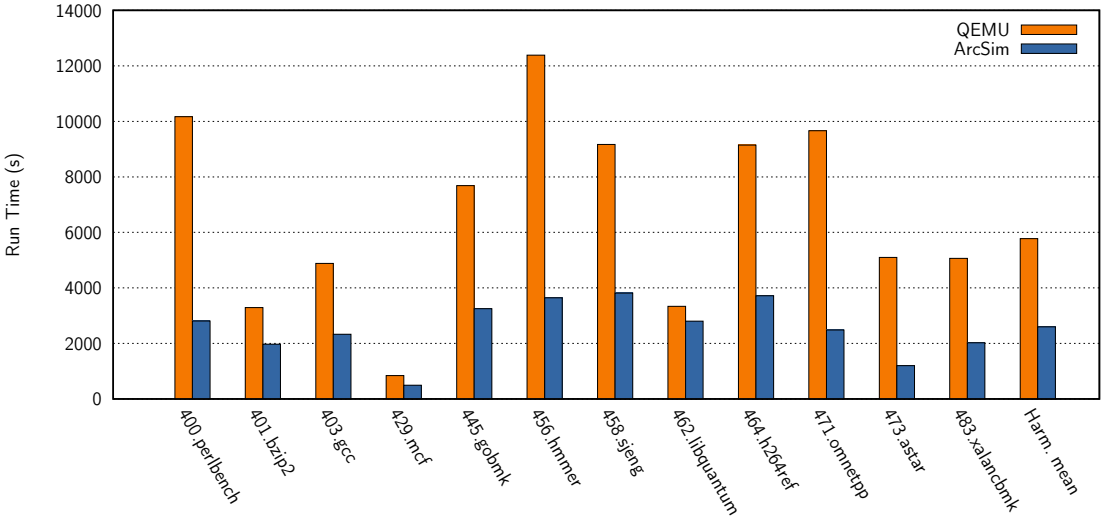



**Figure 5.4:** In user-mode simulation ARCSIM clearly outperforms QEMU. Aggressive region-based optimisations substantially reduce absolute run times (in seconds, lower is better) of the SPEC-CPU2006 benchmarks. Region-based DBT, however, presents a challenge to *hardware virtualisation*, where interrupt checks need to be performed. This chapter presents a methodology, which is (a) correct and (b) retains the performance advantage of region-based DBT in a hardware virtualisation context.

side exits into a region, such as an interrupt check. Typically, these checks test a flag to determine if an IRQ is pending, and then exit the region so that the virtualisation system may process the pending interrupt. It is therefore desirable to reduce the amount of these side exits, so that aggressive region optimisations can continue to be effective for hardware virtualisation, where interrupt checks are mandatory. For user-mode simulation that does not depend on asynchronous signals, these interrupt checks can be removed entirely. But this is not possible for hardware virtualisation, as interrupt checks must be placed in native code, and their cost tolerated.

As of more recent versions, QEMU's approach to interrupt checking is to insert a check at the head of every translation block, resulting in a check before a guest basic block is executed. If an event occurs which requires QEMU to leave native code, a flag is set and the native code will exit to the main execution loop where the event is processed. The approach in ARCSIM is similar, in that a flag is maintained that indicates if an *asynchronous action* is pending, and a check of this flag is inserted into a basic block to determine if the flag is non-zero. If this condition is met, execution will leave native code and return to the interpreter, where the event will be handled.

```
1  void main()
2  {
3          // Wait for some hardware signal
4          while (received_irq == 0) usleep(10);
5
6          // Now do some computation in a loop
7          for (int i = 0; i < 10; i++) {
8                  output[i] = inputa[i] * inputb[i];
9
10                 if ((i & 1) == 0) {
11                         output[i] += inputc[i];
12                 } else {
13                         output[i] -= inputc[i];
14                 }
15         }
16 }
```

**Figure 5.5:** Example code requiring interrupt checking. Termination of the **`while`** loop in `main()` is dependent on an interrupt, hence an interrupt check must be inserted inside this loop. Termination of the **`for`** loop is not dependent on an interrupt, but delaying interrupt checking until after the loop may introduce an unacceptably large interrupt latency.

The difference arises, and opportunities are presented, when the alternative approaches taken to native code generation between ARCSIM and QEMU are considered. As QEMU does not apply any form of *inter*-block optimisation, inserting an interrupt check would only introduce an easily predicted branch-not-taken penalty. However, since ARCSIM considers regions of basic blocks as a unit, the insertion of interrupt checks can inhibit certain optimisations that could be made in the absence of these checks, significantly increasing the amount of code the JIT compiler must translate.

#### 5.1.2.1 Correct Handling of Interrupt Dependent Behaviour

The code given in Figure 5.5 contains a loop that waits for an interrupt handler to run (line 4) followed by a loop-based computation on three input arrays (lines 7–15). The control flow graph for this code is given in Figure 5.3. These two loops demonstrate two distinct scenarios:

**5.1.2.1.1 Interrupt Dependent Behaviour** The **`while`** loop on line 4 depends on an interrupt to proceed, therefore for correctness, an interrupt check must be placed somewhere in this loop. In the absence of a check, a pending IRQ would never be detected, and hence the loop would never terminate.

This kind of behaviour is present in hardware virtualisation, most prominently in operating system kernels, which may wait for an external device to indicate that a buffer is full and ready for processing.

**5.1.2.1.2 Non-interrupt Dependent Behaviour** The computation inside, and the termination of the `for` loop on lines 7–15 is not dependent on an interrupt, but an interrupt check must nonetheless be inserted to avoid an unacceptably large interrupt latency[1], should an IRQ be raised.

Checking for interrupts in an interpreter-based system is straightforward—a check can simply be made at the end of each interpreted instruction or basic block, or alternatively check after a given number of instructions have been executed. These all produce correct behaviour, and impose varying latencies on servicing the interrupts depending on the scheme in use. However, for a DBT based system, there are many more opportunities for determining where to place an interrupt check, along with the challenges of minimising latency and ensuring correctness.

### 5.1.3 Contributions

This chapter makes the following contributions:

1. A new scheme for the optimised handling of asynchronous interrupts in the context of a region-based DBT is presented.
2. The algorithm for inserting interrupt checks is efficient and suitable for JIT processing, and does not introduce unbounded interrupt response times.
3. The scheme improves virtualisation performance and I/O throughput in ARCSIM, when virtualising an ARM guest platform running Linux.

### 5.1.4 Overview

The remainder of this chapter is structured as follows:

- **Section 5.2** briefly introduces the problem of interrupt check placement present in a DBT.

[1] **Interrupt latency** is the time taken from an interrupt being raised, to the interrupt being serviced by the operating system.

- **Section 5.3** gives an overview of the various schemes available, and a description of a new interrupt check placement scheme.
- **Section 5.4** presents an empirical evaluation of the schemes described in Section 5.3.
- **Section 5.5** summarises and concludes the chapter.

## 5.2 DBT Granularity and the Problem of Inserting Interrupt Checks

There are many more options for placing interrupt checks in a DBT system, compared to an interpreter-based system. One of the biggest details, and thus one of the biggest factors in how interrupts are addressed, is whether the DBT translates on a basic block basis, a trace basis, or a region basis (see Figure 5.3).

Basic block-based DBTs must check for interrupts at least once per basic block, as shown in Figure 5.3a. Since each block is permitted to be entered from any predecessor, then if an interrupt check was not performed at the end of a block, the program may get stuck in a loop waiting for an interrupt which is never detected. As control-flow between basic blocks in this kind of DBT is relatively straightforward (usually a map lookup from virtual address to translated code), returning from interrupt handlers is also straightforward as the next instruction to be executed will be the head of a basic block and can be looked up from the mapping.

Trace based DBTs have slightly more flexibility in that checks for pending interrupts can either be made at the end of each basic block within a trace, or at the start of a trace. This is illustrated in 5.3b. Control-flow within a trace is linear, but there may be multiple exit points and so checking at the trace head ensures that if a trace is exited early, there is a check for pending interrupts in the head of the next trace. Checking more frequently may reduce interrupt latency but will impact performance. Checking less frequently may result in the same problem as in the basic block case, where an interrupt necessary for the simulated program to proceed is never detected.

An extension of this can be seen in previous versions of QEMU [21], where the DBT system is built on chained block translations. Each block translation

contains a list of pointers to the address of possible next block translations, and is able to cause a translation to be produced if one does not already exist. Until recently, an asynchronous interrupt initiated a recursive tree walk (which races with the native code execution) from the currently executing block to erase the next block pointers of all child blocks. When the execution engine finished executing a block (and cannot proceed due to the lack of next block pointers), the default case is invoked which causes an interrupt check to be performed. This method, whilst highly optimal for the non-interrupt case (zero overhead, in fact), is an extreme solution to the problem and introduces significant overhead when an interrupt actually does occur. Additionally, as admitted in the source-code (as of version 1.4.0), it suffers from serious race conditions when executing an *symmetric multi-processor* (SMP) emulation. Whilst these issues have been fixed in later versions of QEMU, this comparison aims not to single out QEMU, but to address performance penalties that may occur in any ISS that does not implement intelligent handling of interrupts.

Although the strategies discussed so far work effectively for block and trace based DBT systems, they are inadequate for region based DBTs, which take advantage of dynamically-extracted control-flow information to optimise the generated code across basic block boundaries, and to apply certain loop optimisations to a region of code (Figures 5.3c and 5.3d). An optimisation phase may even split or merge guest program basic blocks during a transformation pass, which will produce highly optimised and correct behaviour, but the representation of the original basic block will be lost.

Unlike in basic block or trace based DBT systems, the generated translations are able to contain looping control flow, which means some care must be taken to ensure interrupts are serviced in a timely manner. A naïve DBT system may decide to insert interrupt checks at the end of each translated basic block. However, this negates many of the benefits of a region based DBT as each interrupt check may result in an exit from translated code, making optimisations which span loops and basic blocks much less effective. Making interrupt checks on entry to or exit from a region (as in tracing DBT systems) will also cause incorrect behaviour, as interrupt dependent loops may be encountered within a region, as shown in the example in Figure 5.5.

Instead, analysing the control flow graph of the region will identify the minimum set of blocks that must contain interrupt checks, while still ensuring cor-

rect behaviour. In this case, it must be ensured that there is at least one interrupt check in at least one *unconditional* basic block of each loop in the CFG. If the JIT fails to insert an interrupt check into a very long running loop, an interrupt may be postponed for an unacceptable length of time (potentially indefinitely). Furthermore, if the JIT fails to insert an interrupt check into a loop that has behaviour which depends on an interrupt being serviced, then the DBT will behave incorrectly.

An algorithm for computing the minimum set of blocks which must contain interrupt checks can be based on computing the *minimum feedback arc set* [71, 32] of the control flow graph. This identifies the minimum set of edges in the CFG which, when cut, remove all cycles from the CFG. Inserting interrupt checks into the root blocks (*source nodes*) of these edges ensures that the minimum number of interrupt checks necessary are placed, thus ensuring correct behaviour while maintaining good performance. In the example above, this would yield the interrupt checks seen in Figure 5.3c.

However, in order to ensure good performance in ARCSIM, the *warm-up* time of the JIT must also be considered. That is to say, the performance of generated code against how quickly that code can be produced must be balanced. Computing the exact feedback arc set of a graph is expensive (the problem is NP-hard [32]), whereas computing an approximation is much faster [44], and is unlikely to result in a significant degradation of performance in generated code versus computing the exact feedback arc set. An approximation of the feedback arc set algorithm in this example might yield the interrupt checks shown in Figure 5.3d, but of course there are many other possibilities.

## 5.3 Region-based Interrupt Checking

This section builds upon the operation of ARCSIM described in Chapter 4, and details the challenges associated with interrupt handling in a region-based DBT.

### 5.3.1 Avoiding Interrupt Edge Bloat

Asynchronous interrupts are a source of adverse control-flow and can significantly degrade collected profiling information by introducing spurious edges from profiled basic blocks. To account for this, an *interrupt stack* is maintained,

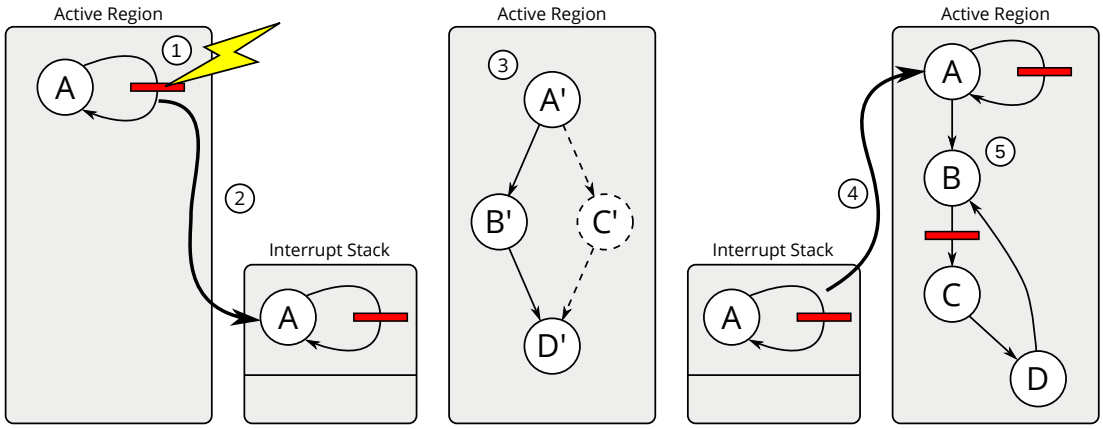



**Figure 5.6:** Flow of region forming when an interrupt occurs. At (1), while executing the IRQ-detection loop (and before any other code has been discovered), an interrupt is detected. At (2), the current region state is pushed onto the *interrupt stack*. At (3), the interrupt handler is executed, treating it as a totally distinct region to the original region. At (4), the system returns to normal execution and pops the previous region state from the stack. At (5), the tight loop is exited, and the profiler continues forming the original region. Crucially, no superfluous edges which link the 'normal' region to the 'interrupt' region have been created.

which allows for control-flow to be profiled at the currently executing *interrupt level*. By default, execution begins in a special *no interrupt* level and profiling information is collected as execution progresses. When an interrupt check indicates an IRQ is pending, the interrupt level is pushed to the *interrupt stack* and execution continues in the interrupt handler with the profiler now collecting information in the new level. Once the interrupt handler completes (possibly returning to user-code), the interrupt level is popped from the stack and execution continues from where it left off, with profiling information from the point of interrupt maintained. A stack is used to accommodate nested interrupts. Figure 5.6 shows how the region forming process proceeds in the presence of interrupts. Rather than superfluous edges being formed between block $A$ and block $A'$, and $D'$ and $B$, control flow is discovered as it exists in the original executable.

### 5.3.2 Interrupt Check Placement Schemes

Region-based DBT gives rise to a number of opportunities for the placement of the interrupt check. The following three schemes can be used to place interrupt

checks within a region, and are implemented in ARCSIM for evaluation:

1. **Full Placement:** An interrupt check is placed before every basic block within a region. This scheme produces correct behaviour, as every block will check for pending IRQs.

2. **Backwards Placement:** An interrupt check is placed before every basic block that is the target of a *backwards* branch within a region. Backward branches indicate looping control-flow, and so this scheme produces correct behaviour, as a pending IRQ cannot be missed.

3. **Optimised Placement:** An interrupt check is placed before the basic blocks that are selected by the algorithm described in Figure 5.7.

Whilst the most accurate algorithm for computing the feedback arc set of the region graph could be used to select basic blocks in which to emit interrupt checks, instead an approximation based on *Tarjan's Strongly Connected Components (SCC)* [113] algorithm as described in Figure 5.7 is used. The use of the approximation ensures that ARCSIM retains its fast warm-up time, by reducing the latency introduced in employing this analysis phase.

Interrupts are always checked for after a basic block has been executed by the interpreter (regardless of the scheme in use)—these schemes apply to how interrupt checks are inserted by the JIT compiler, as it is the performance of translated native code that is important.

During the compilation phase, a *compilation work unit* (containing *guest* basic blocks and their control-flow information) is subjected to analysis by the selected interrupt checking scheme, which determines which blocks should contain interrupt checks. Once those blocks are identified, interrupt checks are inserted where necessary by the translator, as each block is translated.

Tarjan's SCC algorithm requires a minor modification to work with ARCSIM. In particular, self-loops (a basic block with itself as a successor) must be detected, which the algorithm proper does not. Additionally, the algorithm implements the suggestion by the authors for testing whether or not a node is on the stack in constant time by maintaining an `OnBlockStack` flag for each node. Otherwise, the algorithm remains unmodified.

```
define ApplyChecks(WorkUnit):
  NextIndex := 0   # Initialise algorithm state
  do:
    RMCount := 0   # Reset remaining counter

    foreach Block in WorkUnit.Blocks:
      if Block.HasSelfLoop:                 # If the block is a self-loop, then
        Block.HasInterruptCheck := True     # make it have an interrupt check.
      else if not Block.HasInterruptCheck:  # Otherwise, if it doesn't already have
        call StrongConnect(WorkUnit, Block) # an interrupt check, run the algorithm.
  while RMCount != 0   # Loop until nothing remains

define StrongConnect(WorkUnit, StartBlock):
  StartBlock.Index   := NextIndex  # Give the block an index.
  StartBlock.LowLink := NextIndex  # Assume the block is a root block.
  StartBlock.Seen    := True       # Mark the block as seen.
  NextIndex++                      # Consume an index number.

  BlockStack.Push(StartBlock)      # Add the block to the stack.
  StartBlock.OnBlockStack := True  # Mark the block as being on the stack.

  # Loop over each successor
  foreach Successor in StartBlock.SuccessorBlocks:
    if Successor.HasInterruptCheck:  # Ignore blocks that already have checks.
      continue

    if not Successor.Seen:
      # If the block has not been seen, recursively visit it.
      StrongConnect(WorkUnit, Successor)
      StartBlock.LowLink := min(StartBlock.LowLink, Successor.LowLink)
    else if Successor.OnBlockStack:
      # If the block is on the stack, it is part of this SCC
      StartBlock.LowLink := min(StartBlock.LowLink, Successor.Index)

  if StartBlock.LowLink == StartBlock.Index:  # If the block is a root node...
    Count := 0
    do:                                       # Pop the stack to build the SCC.
      StackedBlock := BlockStack.Pop()
      StackedBlock.OnBlockStack := False
      Count++
    while StackedBlock != StartBlock

    if Count > 1:                             # If the SCC contains > 1 blocks,
      StartBlock.HasInterruptCheck := True    # make the root block have an
      RMCount++                               # interrupt check.
```

**Figure 5.7:** Optimised interrupt check placement algorithm for arbitrary code regions, based on Tarjan's [113] algorithm. The algorithm maintains a flag for each node to determine if it exists on the block stack, and a test to handle basic blocks that loop to themselves.

### 5.3.3 Servicing an Interrupt

Asynchronous interrupts may be asserted by any emulated component at any time, and from any host machine thread. In order to abstract asynchronous events (which may not necessarily be IRQs), the concept of *pending actions* is introduced to indicate the presence of an action that must interrupt normal execution. A bitfield in the CPU state structure is used to indicate what type

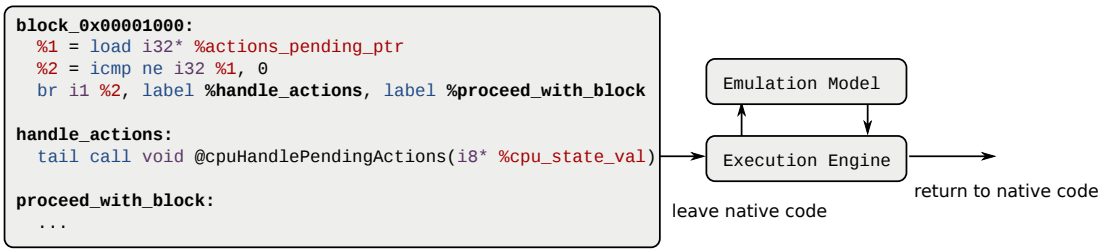



**Figure 5.8:** LLVM IR emitted for interrupt checking at the head of a block determined to be an interrupt check block. If the CPU state structure indicates that an action is pending, control leaves native code via a tail-call back to the execution engine, where the pending action is handled.

of action may be pending. It is this bitfield that is queried when determining whether or not an IRQ is pending, and the native code emitted for interrupt checking simply tests the bitfield. If the value is determined to be non-zero, then it is known that an asynchronous action is pending, and that execution must leave native code to service it.

A pending action may be an IRQ (in hardware virtualisation), a Unix signal (in user-mode simulation) or a special internal signal such as *abort* or *dump state*. This allows a user-mode guest program, for example, to register signal handlers and have a host signal propagated through. In hardware virtualisation, usually a guest operating system (during the OS initialisation phase) will populate an *interrupt vector table* (IVT) with locations to branch to when a particular IRQ is pending. When an emulated platform device asserts an interrupt, control-flow will branch via this IVT to the location specified by the guest OS.

Figure 5.8 shows that when an interrupt check block is executed, and the *pending actions* bitfield is non-zero, control returns from native code via a tail-call back into the execution engine. The execution engine then invokes the necessary routines to service the pending action. As handling the action may result in adverse control-flow, (i.e. an unexpected change to the PC), execution cannot return to native code from where it left, and instead must continue execution via the normal execution engine path. This *may* result in returning to native code, but if the *interrupt service routines* (ISR) have not yet been compiled, then execution will proceed through the interpreter (potentially marking the ISR as hot and leading to compilation).

| Vendor & Model | Dell™ PowerEdge™ R610 |
|---|---|
| Architecture | x86-64 |
| Processor Model | 2× Intel© Xeon™ X5660 |
| Number of cores | 2× 6 |
| Clock Frequency | 2.80 GHz |
| FSB Frequency | 1.33 GHz |
| L1-Cache | 2× 6× 32K |
| L2-Cache | 2× 6× 256K |
| L3-Cache | 2× 12 MB |
| Memory | 36 GB |
| Operating System | Linux 2.6.32 |
| User Space | Scientific Linux 6.6 |

**Table 5.1:** Host configuration.

| DBT Parameter | Setting |
|---|---|
| Target architecture | ARMv5T |
| Target OS | ARM Linux 3.17.0 |
| Translation Model | Asynchronous |
| Tracing Scheme | Region-based [24] |
| Tracing Interval | 30000 blocks |
| Translation Cache | 8192 Entries |
| JIT compiler | LLVM 3.4 |
| Compilation Threads | 10 |
| IR Generation | Part. Eval. [121] |
| JIT Optimisation | `-O3` |
| JIT Threshold | 20 (Adaptive [24]) |

**Table 5.2:** ARCSIM configuration.

## 5.4 Experimental Evaluation

### 5.4.1 Experimental Methodology

Measurements are made on two different types of workloads to evaluate the impact of the various interrupt check placement schemes. One workload is an interrupt-heavy workload in the form of an I/O benchmark, using the standard Linux I/O benchmarking tool `hdparm` [88], another is a compute-heavy workload using the SPEC-CPU2006 integer benchmarks. This comparison will evaluate the impact of the different placement schemes on workloads that require low-latency interrupt servicing, and those that do not rely on interrupts and hence are not sensitive to interrupt latency.

### 5.4.2 Experimental Setup

All of the workloads are executed as applications running *inside* an ARM Linux 3.17.0 *guest* operating system, running an Arch Linux ARM user-space on ARCSIM in hardware virtualisation mode. The host machine for simulation is described in Table 5.1, and the configuration of ARCSIM is described in Table 5.2.

A comparison is also made to the state-of-the-art full-system DBT QEMU version 2.1.50. This comparison is to indicate that the region-based approach to compilation can yield significant performance improvements, even with the added complexity of inserting interrupt checks.

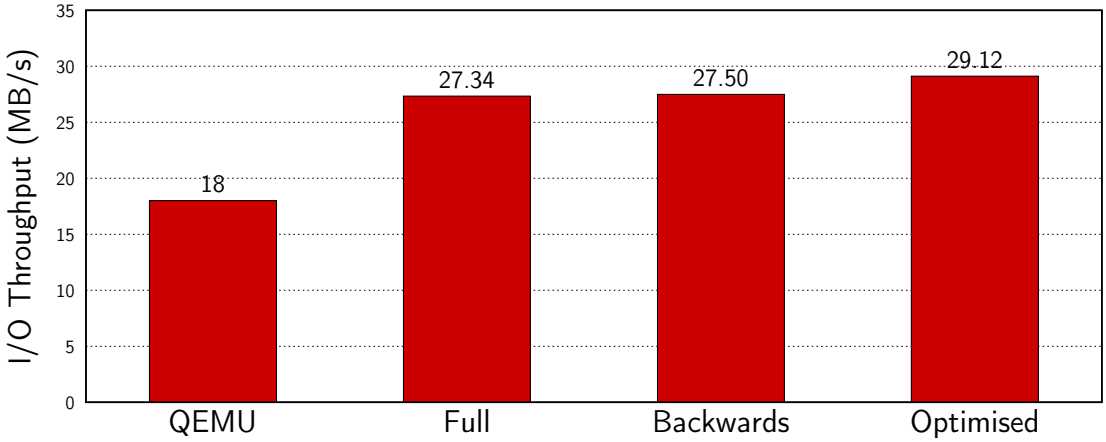



**Figure 5.9:** Absolute I/O throughput in MB/s measured with the `hdparm` benchmark—higher is better. In all cases, ARCSIM has a higher I/O throughput than QEMU, and improves over the baseline *full placement* scheme by 7%.

#### 5.4.2.1 Platform Configuration

An unmodified (*vanilla*) Linux 3.17.0 kernel is used as the guest operating system to host the experiments. The kernel is configured for an ARM Versatile Application Baseboard [11], but the platform as specified includes only 128MB of physical memory, which is not enough to run the SPEC-CPU2006 benchmark suite. For this reason the platform is modified in both ARCSIM and QEMU to include additional memory, enabling the benchmarks to run. The default kernel configuration for the platform is used, except for the addition of the VirtIO block device module. This kernel boots unchanged on both ARCSIM and QEMU.

The VirtIO specification, as detailed in [103], is implemented in ARCSIM in order to provide a block device implementation to the guest Linux operating system. This block device contains the root filesystem for booting, and is also used as the target of the I/O benchmark for testing.

The ARM Versatile Application Baseboard includes a single ARM926 CPU, as well as many external devices such as timers and I/O modules. Most of these devices are supported by ARCSIM, excluding those which are irrelevant to the experiments (such as the FPGA), or for which no documentation is publicly available.

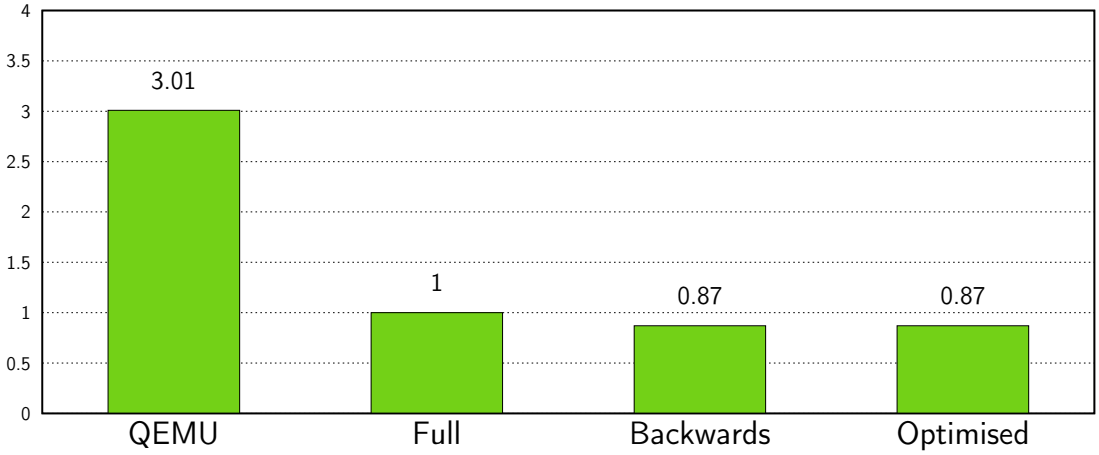



**Figure 5.10:** Relative reduction in wall-clock run-time of the SPEC-CPU2006 integer benchmark against the baseline **full placement** scheme—lower is better. In all cases, ARCSIM is faster than QEMU, and improves over itself by 13% using the *optimised placement* scheme.

### 5.4.3 Key Results for I/O-bound Workloads

For measuring I/O throughput, it is not the performance of the underlying storage device that's important, but rather the performance of interrupt handling in the DBT. For this reason, the `hdparm` benchmark is suitable for stressing the interrupt system as the I/O device is implemented as a VirtIO [103] block device which uses interrupts to convey I/O completion information back to the guest. Measuring the performance of ARCSIM can be accomplished by measuring I/O throughput, as this will correspond directly to the rate at which interrupts can be serviced. Testing different I/O access patterns (such as sequential, random, etc) is also not important, as this will not have any effect on the interrupt system.

Figure 5.9 shows a 61% improvement in I/O throughput on the `hdparm` benchmark over QEMU, and a 7% relative improvement when using the *optimised placement* scheme versus the backwards and full checking schemes.

### 5.4.4 Key Results for CPU-bound Workloads

For hardware virtualisation, it is not possible to remove all interrupt checks—even when there are no interrupts are raised for some time—and as such CPU-bound workloads will incur a small performance penalty due to occasional interrupt checking. Therefore, this experiment considers the impact the place-

ment schemes have on the run-time of a CPU-bound workload. Figure 5.10 shows that ARCSIM experiences a **13%** reduction in the run-time of the SPEC CPU2006 integer benchmarks when employing the more optimal placement algorithms. This can be attributed to the higher quality of native code that is generated as a result of inserting fewer interrupt checks.

#### 5.4.4.1 Comparison to QEMU

As demonstrated, ARCSIM improves over itself when using the more optimal placement algorithms, but it is also important to consider the effect that this has on the benefits of the region-based strategy presented in the previous chapter. A comparison to the state-of-the-art QEMU is made, to show that ARCSIM maintains its performance advantage. Figure 5.10 shows that against the baseline *full placement* scheme, QEMU is $3\times$ slower in full-system mode, confirming that the region-based DBT approach maintains its ability to optimise code across block boundaries, despite the inserted interrupt checks.

Furthermore, an advantage of using the VirtIO infrastructure is that QEMU can be configured to use exactly the same kernel image, filesystem and block device, enabling a direct comparison of I/O throughput between ARCSIM and QEMU.

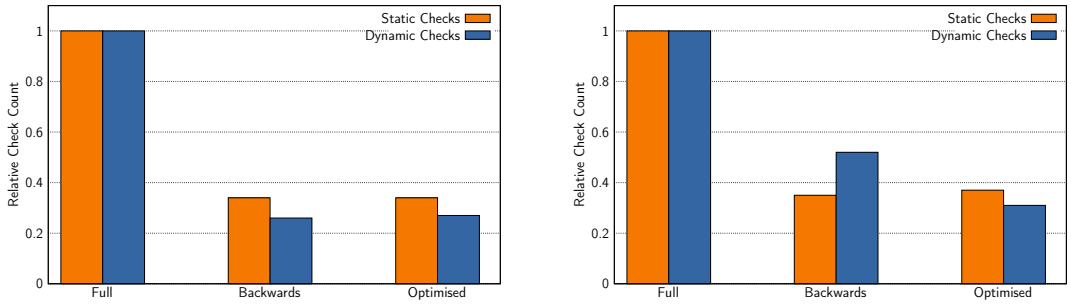
### 5.4.5 Further Analysis

#### 5.4.5.1 Static and Dynamic Interrupt Checks

A **static** interrupt check corresponds to the decision to place an interrupt check in a given basic block, where a **dynamic** interrupt check corresponds to the execution of a **static** interrupt check at run-time. The aim of the optimal placement scheme is to minimise the number of *static* interrupt checks placed, and correspondingly reduce the number of *dynamic* checks made. A reduction in *static* interrupt checks serves two purposes:

1. The amount of IR the LLVM JIT compiler is presented with is lower, thereby reducing the amount of work the optimiser and compiler have to do and subsequently improving compilation time, and

2. the optimiser is free to perform more aggressive optimisations within the




**(a)** I/O bound workload **(b)** CPU bound workload

**Figure 5.11:** Reduction in *static* and *dynamic* interrupt checks for I/O and CPU-bound workloads on the three different interrupt checking schemes—lower is better. For I/O-bound workloads, the *optimised placement* scheme reduces the amount of *static* checks by 66% and *dynamic* checks by 73%. For CPU-bound workloads, *static* checks are reduced by 63% and *dynamic* checks by 69%.

region, and produce higher quality (and subsequently more performant) native code.

Figures 5.11a and 5.11b both show that fewer *static* interrupt checks are placed, and as a consequence generally perform fewer *dynamic* checks. The exception to this is when using the *backwards branch* scheme in a CPU bound workload where an increase in *dynamic* checks is observed. This can be attributed to CPU-bound workloads spending more time in *hot* looping control-flow, where the scheme will have necessarily inserted an interrupt check, and therefore increase the *dynamic* interrupt check count. The optimised placement scheme places **66%** less interrupt checks than the baseline scheme, and causes **73%** fewer dynamic checks to occur.

#### 5.4.5.2 Interrupt Latency

The interrupt latency is the time it takes for a simulated interrupt to be raised, until the point at which the execution engine begins executing the ISR. A reduction in interrupt latency will improve the throughput of an I/O bound workload, as data requests are served more quickly. These results show the impact that the placement schemes have on interrupt latency, to ensure IRQs are not being deferred for an unacceptable period of time. These measurements are taken for the I/O-bound workload, as interrupt latency will not affect the throughput of CPU-bound workloads.

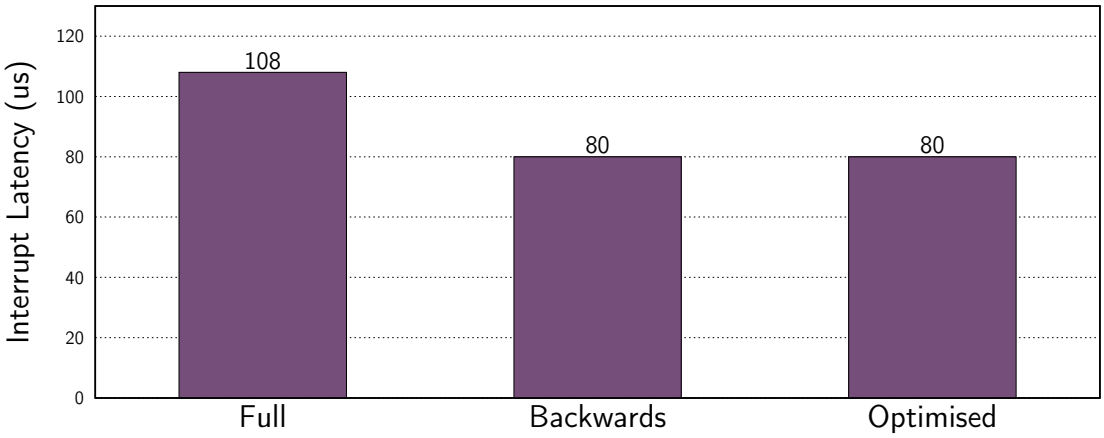



**Figure 5.12:** Absolute interrupt latency in $\mu s$ as measured when running the `hdparm` I/O benchmark—lower is better. The backwards and optimised placement schemes reduce interrupt latency by 26%.

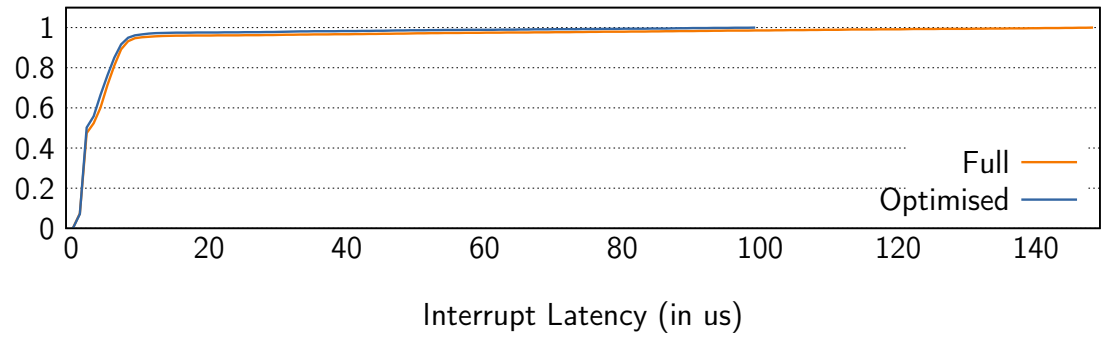



**Figure 5.13:** Cumulative distribution of interrupt latencies for the optimised placement scheme, compared against the full placement scheme.

Whilst it may seem that there should be a lower latency for the *full* placement scheme (i.e. more checks, means more opportunities to respond to an interrupt) the impact that the scheme has on generated code quality is such that higher latencies are actually observed (**108$\mu s$**, over **80$\mu s$** on average) when employing this. Figure 5.12 shows that interrupt latency is reduced in the schemes which reduce the amount of *static* checks inserted, and for the optimised algorithm, latency is reduced by **26%**. The higher quality of native code that is generated leads to faster execution rates, and accounts for the fact that interrupts can be served more quickly.

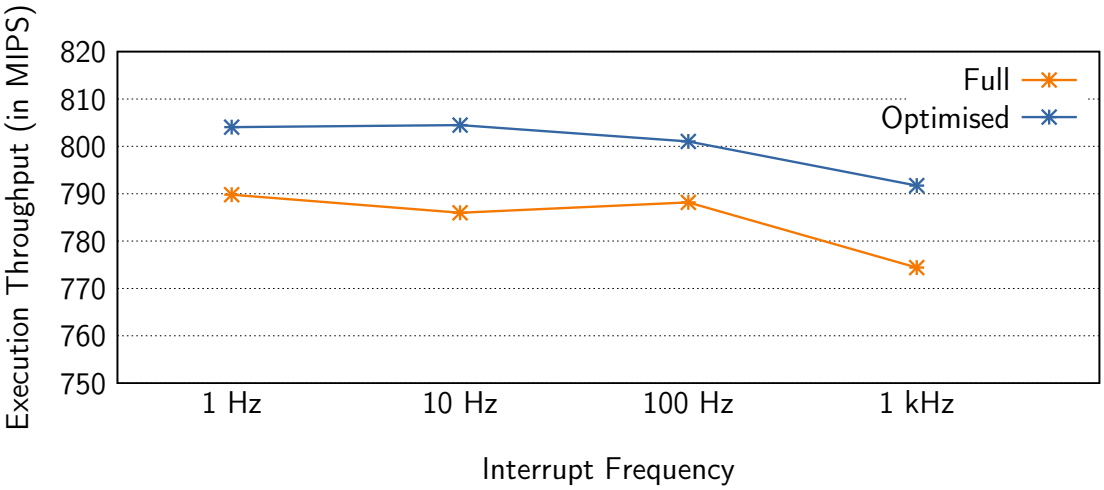



**Figure 5.14:** Comparison of the full placement scheme versus the optimised placement scheme using a range of interrupt frequencies from 1Hz to 1kHz. Using the full placement scheme causes a significant performance overhead at high interrupt frequencies, while the optimised policy continues to provide good performance.

#### 5.4.5.3 Distribution of Interrupt Latencies

Figure 5.13 shows how various latencies for serviced interrupts are distributed between execution with the *full* and *optimised* scheme. The cosine similarity of the latencies produced between these schemes is $\cos\theta = 0.999$, indicating that the differing schemes do not significantly vary in the latencies yielded by the system. The cumulative latency distribution shown in Figure 5.13 is in fact comparable to that of the ARM port of the popular Xen hypervisor running on actual hardware [129].

#### 5.4.5.4 Scalability

In order to determine the scalability of ARCSIM, a range of frequencies from 1Hz to 1kHz were artificially inserted to test how ARCSIM scales to higher frequencies, and Figure 5.14 shows that the optimised placement scheme consistently performs better than the naïve full placement scheme. Furthermore, it can be seen that ARCSIM maintains a relatively consistent level of performance across the frequency range, only dropping by approximately 2%.

#### 5.4.5.5 Comparison to Hardware

With simulation throughput approaching actual hardware performance, it is important to ensure that interrupt handling in ARCSIM is on even terms with the hardware being emulated, i.e. there is not an unacceptable amount of latency being introduced.

Interrupt response time observed on actual, non-simulated systems is the sum of a hardware dependent time and some operating system induced overhead. The hardware dependent time is determined by the micro-architecture of the processor and its current state, the system configuration and the type of interrupt. Operating system overheads may vary greatly between best and worst case scenarios, and are generally worst when the kernel (temporarily) disables interrupts.

According to the manufacturer's specification [13] the interrupt latency seen by a Linux driver running on an ARM1176JZ(F)-S with two levels of cache is approximately 5000 cycles. This is largely caused by overheads in the operating system itself. 5000 cycles at 300MHz is 16.7$\mu$s, and ARCSIM yields an average latency of 80$\mu$s when using the optimised placement scheme.

## 5.5 Summary & Conclusions

This chapter has developed an optimised scheme for the efficient placement of asynchronous interrupt checks in a cross-architecture hardware virtualisation system, that employs region-based dynamic binary translation. This technique detects control flow loops of any structure and nesting level and inserts a near-minimal number of interrupt checks. It also provides correctness through the guarantee that at least one check for pending IRQs is performed for each iteration of any enclosing loop. On average, the number of dynamic interrupt checks is reduced when virtualising an ARM platform by 73%, in comparison to a scheme that checks for interrupts at the end of each basic block. Despite the reduced frequency of interrupt checks the latency for servicing interrupts is reduced by 26% due to increased opportunities for code optimisation between interrupt checks. ARCSIM also maintains a performance advantage over state-of-the-art QEMU, where I/O throughput is improved by $1.6\times$ and virtualisation performance improved by $3.4\times$ in hardware virtualisation across a range

of benchmarks.

This chapter has introduced a specific challenge involved in efficient hardware virtualisation, but there are many more concerns that need to be addressed. The next chapter shall introduce a novel approach to cross-architecture hardware virtualisation, considering the major factors that affect the performance of such a system.

Chapter 6

# Hardware Accelerated Cross-architecture Virtualisation

The previous chapter tackled an important issue for efficient cross-architecture hardware virtualisation in the form of asynchronous interrupt handling, but this is just one challenge in a large problem space. Hardware virtualisation is a popular technology used for workload consolidation, application sandboxing, debugging software, prototyping hardware and simply running a different operating system on a host machine. Modern processor vendors have introduced architectural support for hardware virtualisation (so-called *hardware assisted virtualisation*) in the form of *instruction set extensions* (ISE). These ISEs enable very efficient virtualisation of host system hardware resources by permitting guest operating systems to run directly on the physical hardware, with minimal supervision. These extensions, however, are geared towards *same*-architecture virtualisation, where a virtual guest machine is of the same architecture as the physical host machine. This chapter develops a novel hypervisor called CAPTIVE that exploits the hardware accelerated virtualisation extensions present on modern processors to efficiently virtualise an architecture that is different to the host.

## 6.1 Introduction

Hardware virtualisation is the provision of an abstract, virtual computing environment on a host machine, such that operating systems and system software can run in isolation, under the impression that they are running on real hard-

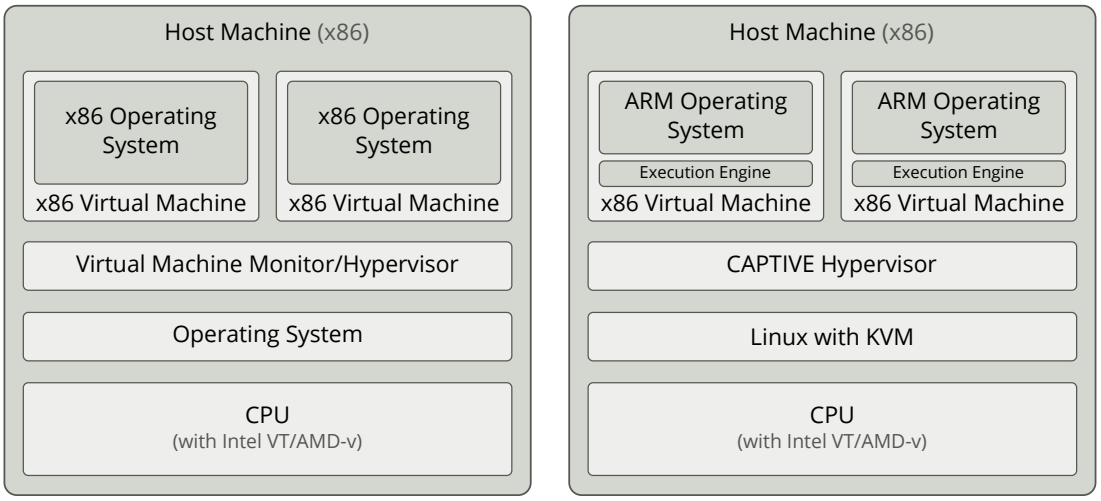



**Figure 6.1:** Modern processors support hardware assisted virtualisation, but these extensions are for same-architecture virtualisation only. CAPTIVE exploits these extensions to accelerate cross-architecture virtualiation by mapping behaviour of a guest system, to corresponding behaviour of the host.

ware. For this to work, and to ensure isolation, virtual machines must be monitored by a so-called *hypervisor* (or *virtual machine monitor* (VMM)) that ensures they can not interfere with the overall working of the physical machine.

A software-based solution, such as ARCSIM described in the previous chapters or QEMU (without KVM), implements hardware virtualisation through dynamic binary translation and hardware device emulation, e.g. a virtual CPU executes guest machine instructions, and when device accesses are made, they are routed to software implementations. However, there are more challenging problems for DBT-based virtualisation than simple instruction and device emulation.

As mentioned previously, hardware virtualisation requires emulation of architectural components, such as the *memory management unit* (MMU), interrupt controllers, devices, etc. and a software-based solution introduces a significant amount of run-time overhead. For example, architectures that have a virtual memory system must translate virtual memory addresses to their corresponding physical addresses, along with checking access permissions, in order to emulate this virtual memory model. Recognising these challenges, modern processor vendors provide *hardware extensions* that are designed to support same-architecture virtualisation, by allowing guest operating systems to run unmodified directly on the physical CPU. Some examples of this technology are

Intel VT and AMD-V on x86 processors and ARM Virtualization Extensions on ARM processors.

However, these hardware-assisted virtualisation extensions are geared towards *same*-architecture virtualisation, where both the guest VM and the physical host machine share the same architecture. For *cross*-architecture virtualisation (where the guest and host architectures are different), translation between ISAs, emulation of the guest system's MMU, interrupt handling and I/O devices are typically implemented entirely in software, resulting in a substantial performance loss. For example, in full-system mode the gem5 architectural simulator [22] takes about 30 minutes to boot into Linux on a current, mid-range host machine. While this performance level is sufficient for some computer architecture research, it is far too slow for any practical applications. QEMU [21], a popular cross-architecture full-system virtualiser using dynamic binary translation (DBT), is substantially faster, but still suffers an up to $20\times$ slow-down [69] compared to native execution on the host.[1]

While same-architecture hardware virtualisation has become ubiquitous there are fewer, but nonetheless important applications for cross-architecture virtualisation, e.g. Android software development using the QEMU-based Android Emulator shipped with the Android Studio [46], which provides an ARM environment for software developers using an x86 host machine; building ARM Docker [89] containers on x86 machines; providing fast-forwarding in sampling based simulators [106]; or early-stage software development without a hardware target [30]. All of these applications critically depend on fast cross-architecture virtualisation due to unavailability or deliberate absence of a hardware platform supporting the chosen target ISA.

In this chapter, a novel approach for speeding up *cross*-architecture hardware virtualisation is developed, and these ideas are implemented in a new hypervisor called **CAPTIVE**. This chapter moves away from ARCSIM as described in previous chapters, as these new ideas cannot be applied to this existing system. However, many of the techniques previously described and used in ARCSIM are carried into CAPTIVE, where appropriate.

[1] Native execution of a binary suitably compiled to the host ISA from the same sources, which have been used to build the binary for the guest's ISA.

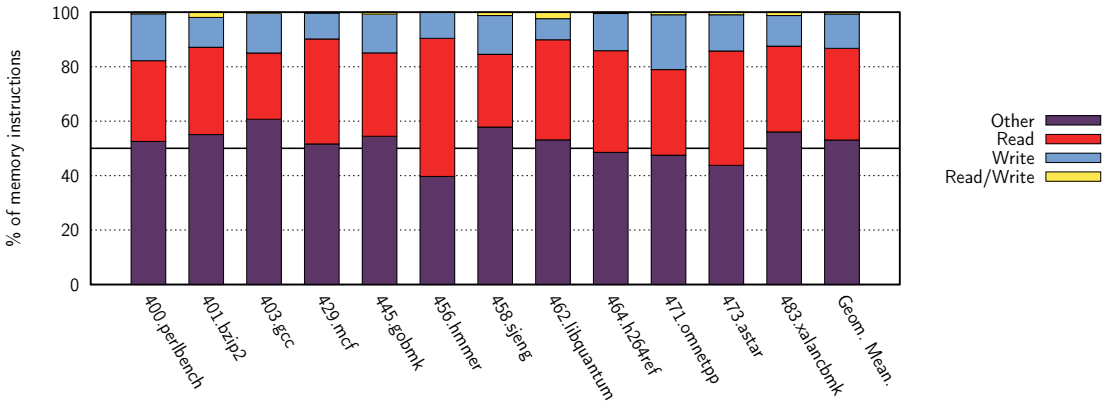



**Figure 6.2:** Distribution of operations in the SPEC-CPU2006 integer benchmarks. On average, around 50% of all instructions executed perform memory operations (either *read*, *write*, or both), which require expensive virtual-to-physical address translation using a software MMU.

### 6.1.1 Key Idea

The *key idea* is to eliminate performance bottlenecks by exploiting the existing virtualisation hardware extensions originally devised for *same*-architecture virtualisation, and mapping guest architecture behaviour onto corresponding host architecture behaviour.

### 6.1.2 Motivating Example

It has been well established [85, 31, 125, 60] that emulation of a guest MMU is one of the most time-consuming parts of cross-architecture virtualisation, therefore this motivating example will focus on the memory address translation process required for virtualisation. For this, consider the diagram in Figure 6.2, which shows the percentage of memory operations w.r.t. the total number of executed instructions in the SPEC CPU2006 integer benchmarks. About 50% of the instructions in these benchmarks perform memory accesses. This indicates that when running these benchmarks in a virtualised ARM guest environment on an x86 host, on average, every other instruction demands an expensive virtual-to-physical memory translation using a virtualised ARM MMU (typically implemented in software). If these address translations can be sped-up, one of the most severe cross-architecture virtualisation performance bottlenecks will be eliminated.

Figure 6.3 shows that in a 32-bit ARMv7-A system with an ARMv7-A MMU,

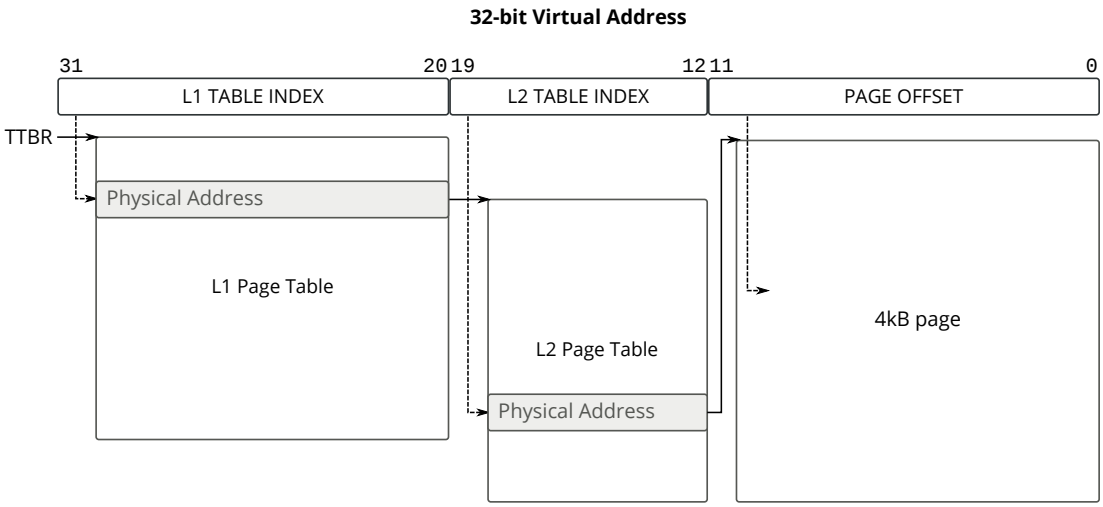



**Figure 6.3:** The operation of an ARMv7-A MMU. A 32-bit virtual address is translated to its corresponding physical address, by indexing an L1 and L2 page table. The `TTBR` points to the base of the L1 page table, and the entry in the L1 page table points to the base of the L2 page table. The entry in the L2 page table then points to the base of the corresponding physical memory page.

there are at most two levels of page tables representing a virtual memory area. To translate a virtual address into its corresponding physical address, the first-level page table (an L1 page table), pointed to by the `TTBR` register, is indexed by bits 20–31 of the virtual address and the entry interrogated to determine if the mapping is to a *section* (a 1MB contiguous chunk of memory) or a small page (a 4kB contiguous chunk of memory). If the page table entry indicates a section, then the base address points to the physical base address of the memory. If the page table entry indicates a small page, then the base address points to the physical base address of a second-level page table (an L2 page table). The L2 page table is indexed by bits 12–19 of the virtual address, and the base address in the L2 page table entry points to the physical base address of the page corresponding to the mapping. A page table entry in the L1 (if pointing to a section) or L2 page table in addition to the base address pointer contains flags that indicate the access permissions of the page, e.g. whether or not the page is readable and writable, and if it is accessible whilst executing in the user privilege level.

In a 64-bit x86 system, there are four levels of page tables that represent a 48-bit virtual address space. Pointers are always 64-bit wide, but can only access 48-bits of virtual address space. Virtual addresses *must* be in *canonical*

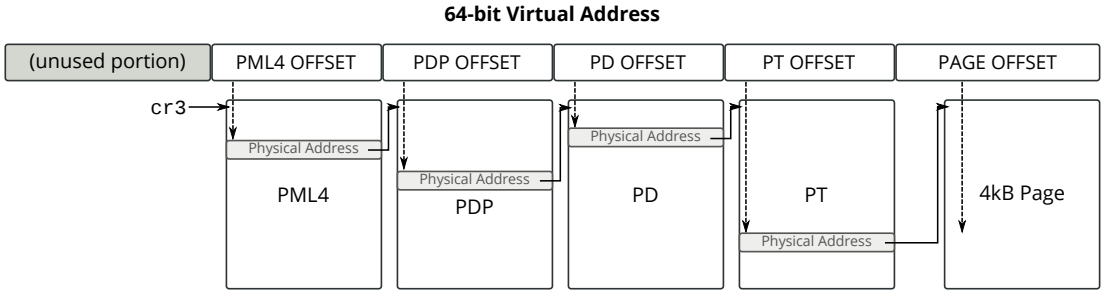



**Figure 6.4:** The operation of an x86 MMU. The `cr3` register points to the base of the *page map level 4 table*, which in turn points to the *page descriptor pointer table*, then the *page descriptor table*, and finally the *page table*. The unused portion of the address (bits 48 to 63) are copies of bit 47, making this a virtual address in *canonical form*.

*form*, where bits 48–63 of the address are copies of bit 47. Any memory access to a non-canonical address will result in a general protection fault. Address translation operates in a similar fashion to ARM, with each level of page table being traversed to translate a 64-bit virtual address into a 64-bit physical address, subject to permissions which can be applied at any level of the page table—the higher level permissions taking precedence over the lower levels.

To avoid a costly page table walk for every memory access, both architectures employ a *translation lookaside buffer* (TLB), which caches the result of a previous hardware translation. If the page tables are modified, or the pointer to the top-level page table changed, the TLB *must* be flushed.

From this description it should be clear that the structure of the ARM and x86 MMUs are substantially different, yet fundamentally they both provide a mechanism for the translation of virtual addresses to physical addresses with permission checking. This chapter proposes to exploit this hardware address translation capability, and show how to map the behaviour of an ARM MMU onto a *virtualised* Intel MMU. By intercepting ARM TLB invalidations and maintaining entries in the x86 page table that represent entries in the ARM page table, address translations can be accelerated. Instead of using a slow software implementation of the ARM MMU, guest address translations are redirected to the fast, host virtualised Intel MMU, which CAPTIVE keeps consistent with the guest's ARMv7-A MMU. Using existing extensions originally devised for *same*-architecture virtualisation critical *cross*-architecture address translations can be sped-up over a pure software MMU implementation.

### 6.1.3 Contributions

This chapter targets four distinct *cross*-architecture virtualisation challenges, and makes the following contributions:

1. Virtual-to-physical address translation is accelerated through the use of virtualisation extensions, by mapping behaviour of the guest MMU onto corresponding behaviour of the host MMU—despite substantial differences between the two MMUs.

2. A DBT system for the translation from the guest to host ISA is presented, where a fast, block-based, domain-specific, *just-in-time* (JIT) compiler that lives *inside* the native virtual machine translates guest instructions to host instructions, exploiting techniques not usually available to a user-space JIT.

3. An efficient mechanism to emulate the guest's memory mapped I/O devices is developed, by exploiting the host's MMU to detect device accesses.

4. Finally, an interrupt handling scheme is developed, which correctly honours the guest's instruction boundaries, even if one guest instruction is mapped onto several host instructions, thus implementing precise, yet efficient guest interrupts.

### 6.1.4 Overview

The remainder of this chapter is structured as follows:

- **Section 6.2** will expand the background on MMU virtualisation in full-system simulators, and introduce further detail on Intel virtualisation technology (Intel VT), and KVM.
- **Section 6.3** will present the novel *cross*-architecture virtualisation techniques, and how they are implemented in the new hypervisor CAPTIVE.
- **Section 6.4** will present the results from an empirical evaluation of the techniques.
- **Section 6.5** summarises and concludes the chapter.

## 6.2 Background

The technologies used by CAPTIVE have previously been introduced in Chapter 3, and this section shall extend these descriptions, and explain how they apply to the implementation.

### 6.2.1 KVM

CAPTIVE requires a virtual machine backed by hardware extensions, and uses the KVM infrastructure provided by the Linux kernel to accomplish this. As described in Section 3.3.1, KVM provides a convenient abstraction for accessing the hardware virtualisation extensions of the host system, and so works in the presence of technologies such as Intel VT and AMD-V. KVM also supports other host architectures such as ARM (with ARM Virtualization Extensions), PowerPC and MIPS, meaning that porting CAPTIVE to other host architectures would be quite straightforward.

The key idea is to create a *regular same*-architecture virtual machine on the host (with KVM), and map the behaviour of a guest platform to the behaviour of the host.

### 6.2.2 Intel VT

As described in Section 3.3.2, Intel VT (Intel Virtualization Technology) are the collection of virtualisation extensions available on modern Intel processors, and CAPTIVE depends on this technology for hardware acceleration. Generally, access to virtualisation extensions requires operating system level (i.e. privileged) access to the host machine and since CAPTIVE is started as user-space process, it would require co-operation from the OS kernel to enable and run the extensions. Access to Intel VT is mediated by KVM, with the alternative being to develop a custom kernel module.

## 6.3 Virtualisation Infrastructure

ARCSIM is a well-engineered simulator that fully supports hardware virtualisation, but it can not be readily adapted to support *hardware-assisted* virtualisation. This is because ARCSIM operates entirely in software, within the con-

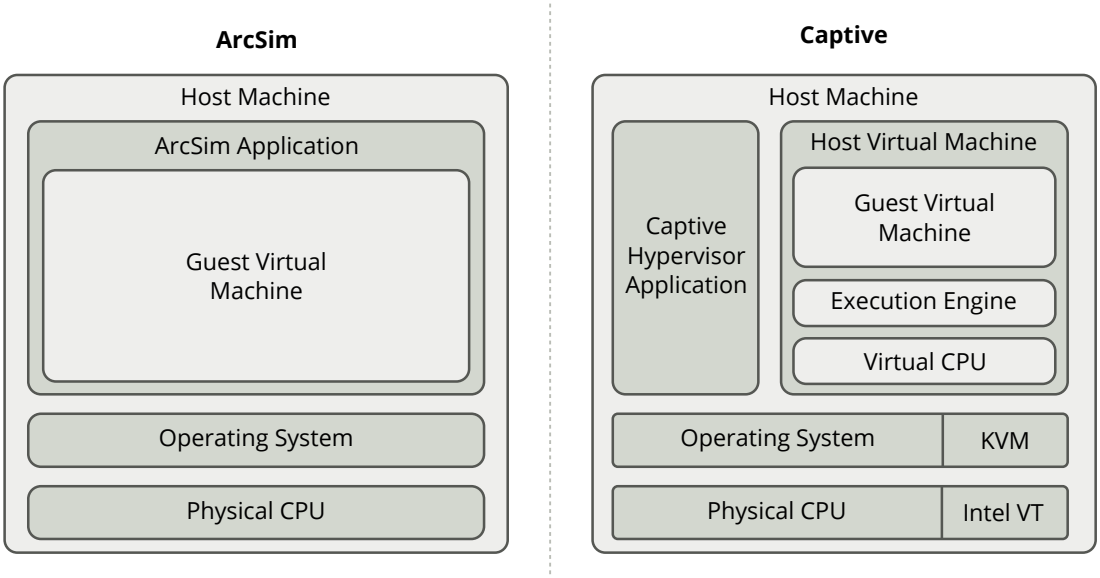



**Figure 6.5:** ARCSIM is a software-based hardware virtualisation system, and CAPTIVE is hardware-accelerated. ARCSIM provides a *guest virtual machine* by emulating the platform in software, whereas CAPTIVE utilises host hardware extensions to map behaviour of a guest platform to behaviour of the host system.

straints of an operating system, with its infrastructure tightly coupled to user-space libraries such as `glibc` and `llvm`. The approach taken by CAPTIVE is to instantiate a virtual machine on the host machine, in which the virtualisation takes place, exploiting the ability to utilise all hardware features normally only available to operating system software on the host system. The fundamental purpose of each system is to provide a *guest virtual machine*, which represents the platform being virtualised, and Figure 6.5 shows how both ARCSIM and CAPTIVE realise this goal.

A consequence of this is that there is no operating system *inside* the virtual machine, and as such CAPTIVE must itself perform architectural setup and management that would normally be handled by an OS.

Figure 6.6 gives a high-level overview of how CAPTIVE is structured, with the virtualisation infrastructure consisting of three main components:

1. A **hypervisor** component, which runs on the host machine and uses KVM to control the host's hardware virtualisation extensions.

2. An **execution engine** component, which runs *inside* a normal virtual machine on the host.

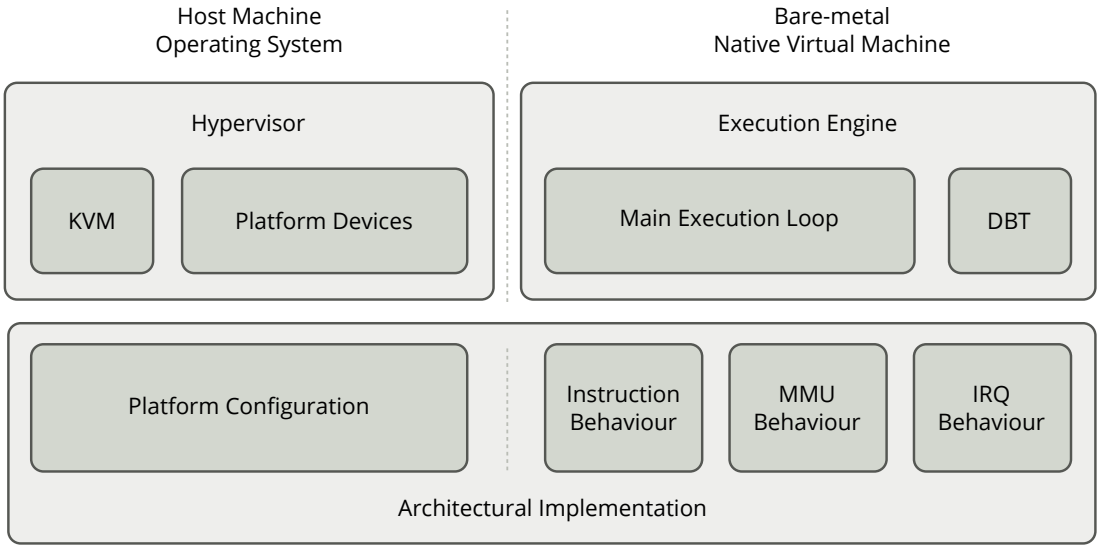



**Figure 6.6:** A high-level overview of CAPTIVE's infrastructure, showing the three main components that make up the system. The *hypervisor* runs in the host machine's operating system and the *execution engine* runs inside a *native virtual machine.*. The *architectural implementation* contains the configuration of the target platform, and specifies architecture-specific behaviour.

3. An **architectural implementation**, which specifies the behaviour of the architecture being virtualised and defines the configuration of the guest platform.

The remainder of this chapter will assume that an **ARMv7-A guest** architecture is being virtualised, based on an *ARM RealView Platform Baseboard for Cortex-A8* [9]. The host machine will be a standard **x86-64** machine with Intel VT virtualisation extensions and KVM support.

Due to the multi-layer nature of this system, it is important to define a particular term at this point, to identify exactly what aspect of the system is being described.

**Definition 16** (Native Virtual Machine)**.** *The hardware extensions provided by the **host** machine naturally provide a same-architecture virtual machine, e.g. using* QEMU *with KVM on x86 would result in an x86 virtual machine. In this case, the **Native Virtual Machine (Native VM)** refers to the virtual machine provided by these hardware extensions, which are utilised in the infrastructure. Therefore, in this chapter, the **Native VM** is of the same architecture (x86-64) as the **host**.*

### 6.3.1 System Components

The following sections shall briefly describe the three main components that make up CAPTIVE.

#### 6.3.1.1 Hypervisor

The **hypervisor** component of CAPTIVE is the part of the system that runs inside the (Linux) operating system of the host machine—it is the software that the user directly operates to start a virtualisation session. It interfaces with KVM to access the host platform's hardware-assisted virtualisation extensions, and contains software implementations of the devices that are to be virtualised for the guest platform.

Typically, the user will provide a guest kernel and a disk image for the architecture being virtualised, then indicate which type of architecture is being virtualised. The CAPTIVE hypervisor is a generic component, and is not dependent on either the host or guest architecture—it only depends on KVM being supported by the host system's Linux kernel.

#### 6.3.1.2 Execution Engine

The **execution engine** is the core component that performs the virtualisation of the guest platform's CPU, and maps guest architectural behaviour to the host architecture. It runs inside the *native virtual machine* and can be thought of as a lightweight operating system.

The execution engine is specific to the host architecture, but independent of the guest architecture. This is because the execution engine runs inside the Native VM and must know how to configure the host MMU, install interrupt handlers, and perform privileged operations (such as TLB flushes).

The engine also contains an optimising JIT compiler that translates a generic *intermediate representation* (IR) (produced by the *architectural implementation*) into the instruction set of the host machine.

#### 6.3.1.3 Architectural Implementation

The **architectural implementation** is the component that defines the behaviour of the guest architecture that is ultimately being virtualised. It is automatically

generated from a high-level description using GENSIM, as described is previous chapters.

This component contains a virtual representation of the guest CPU, actions to perform when architectural events (such as page faults and interrupts) occur and a service that produces generic IR for a basic block of guest instructions. The execution engine will invoke the IR generator when it needs to translate guest instructions to host instructions.

The architectural implementation also contains the configuration of the guest platform, which is used by the *hypervisor* component when initialising the platform devices.

For the example virtualisation set-up described in this chapter, the architectural description from the previous chapters is used *unmodified*. The only changes needed are to the GENSIM generators that produce the output modules.

### 6.3.2 Overview

This section shall give an overview of the operation of CAPTIVE, and will continue to use the example of an x86-64 *host* machine, with a RealView Platform Baseboard Cortex-A8 [9] guest platform.

CAPTIVE starts by instantiating a native virtual machine on the host, using the KVM framework. KVM is the interface to the Intel VT virtualisation extensions, and starts by initialising the required structures that represent a virtual machine on an Intel processor (e.g. creating the VMCS structure, and issuing the `VMXON` instruction). Then, a single virtual x86-64 CPU is requested from KVM which will ultimately represent the virtual ARM processor.

The native VM must be supplied with *virtual* physical memory, i.e. the physical memory space of the native VM must be backed by virtual memory from the hosting application. This memory is allocated lazily[2] by the hypervisor with the `mmap` system call, and represents the physical memory provided by the guest platform. In this particular case, 512MB of physical memory should be allocated, but in order to run larger benchmarks (such as SPEC-CPU2006) for evaluation purposes, the guest platform physical memory is artificially extended to 2GB. An additional block of physical memory is also installed that contains

[2]Using the `MMAP_NORESERVE` and `MMAP_ANON` flags.

the execution engine binary, heap space for memory allocations, critical architectural data structures (such as the *global descriptor table* (GDT) and the MMU page tables), and the compiled code cache.

Once the physical memory has been configured, the execution engine is loaded into its own region, and the guest kernel (an ARMv7-A Linux Kernel), is copied into the location specified by the platform boot protocol. Finally, a virtual memory mapping is created that maps the execution engine into virtual memory, and the virtual x86-64 CPU is configured to start up in 64-bit mode, with the instruction pointer at the entry-point of the execution engine. Control is then transferred to the native VM, and the execution engine running inside takes over.

Once inside the native VM, CAPTIVE has full control of a bare-metal x86-64 machine, the execution engine is essentially an x86-64 kernel, with full privileged access to this virtual machine. The virtual memory of the native VM is configured in such a way as that the lower 4GB portion represents either a one-to-one mapping of guest physical memory (if the guest MMU is turned off) or the actual virtual memory mapping of the guest machine (detailed in Section 6.3.4). The execution engine itself resides in the high portion of virtual memory, and certain other virtual memory areas are mapped to the heap and stack.

When first started, the execution engine performs architectural initialisation of the native VM, including setting up the x86 *interrupt descriptor table* (IDT), and then begins executing guest ARM code. The guest kernel to be executed has already been loaded into guest physical memory, so execution begins from this entry-point, using JIT compilation of the guest ARM instructions to native x86-64 instructions (detailed in Section 6.3.3).

Any access by guest instructions to the memory of the guest machine is performed with a normal memory access, without having to translate/transform any virtual memory addresses—they are simply made to the address to which they would be made if running on a non-virtualised ARM system. The guest platform being virtualised is a 32-bit platform and so any memory access can only be in the 0-4GB ($2^{32}$) range of lower virtual memory. Virtualisation of a 64-bit platform is outside the scope of this chapter, but would require an extra layer of software indirection and is planned for future work. When an access to a particular virtual address occurs for the first time, a page fault is generated and handled by installing a mapping of the corresponding virtual page to guest

physical memory (subject to the operation of the ARM guest MMU). The native VM also tracks the currently executing mode of the guest system, by running in either x86 ring 0 (when running in privileged mode), or x86 ring 3 (when running in user mode). This also enables the MMU to perform efficient permission checking when dealing with user/kernel page table permissions.

External interrupts generated by devices (such as timer devices, network devices, disk devices, etc) are propagated as real interrupts into the guest system, which causes a flag to be set to indicate that translated code should stop executing at the next *safe point*. At a minimum, a *safe point* is an instruction boundary, but safe points are inserted at guest basic block boundaries as the DBT scheme in use is basic block-based.

### 6.3.3 CPU Virtualisation

Same-architecture virtualisation is easily supported by modern processors that include hardware support for virtualisation. Technologies such as Intel VT and AMD-V allow guest code to run directly on the host processor, without modification or instrumentation for maximum performance. Certain privileged operations (such as changes to control registers, and TLB invalidations) are trapped by the host CPU and handled by a hypervisor (for example KVM).

This virtualisation is trivial in the same-architecture case, because both the guest and the host have the same *instruction set architecture* (ISA) and are therefore binary compatible. However, this presents a problem for cross-architecture CPU virtualisation, as the ISAs are different, and completely incompatible.

As shown in previous chapters, techniques such as *interpretation* or *dynamic binary translation* (DBT) are used to virtualise the guest ISA on the host ISA, the former being straightforward to implement, and the latter being recognised as one of the fastest ways [115, 41, 21] to emulate guest instructions on a host machine. Emulation of guest instructions is a necessity for cross-architecture virtualisation, and techniques for doing so have been well studied and presented in DBT improvement articles such as [76, 49].

CAPTIVE's approach to instruction emulation is based on a basic block *just-in-time* (JIT) compiler engine, which takes guest basic blocks discovered at runtime, and compiles them into corresponding host basic blocks. Block compilation is synchronous to the execution of the guest system, and occurs on-demand

when a translation for a particular guest basic block is not available. Generated host code is stored in a code cache, for later use. This is similar to the approach taken by QEMU, except for two important differences:

1. Code is generated in such a way that it is independent of the virtual address of the guest basic block.
2. The JIT compiler lives *inside* the native virtual machine as part of the execution engine.

This approach is clearly different to the more advanced region-based strategy presented in Chapter 4, but there are a number of implementation challenges that necessitate starting from a simpler approach:

1. The region-based approach relies on LLVM to perform JIT compilation, but it is not feasible to incorporate such a large library inside the native VM, as there is no operating system or C-library, which LLVM requires.
2. The execution engine has no concept of *threads* and so asynchronous compilation inside the native VM cannot be performed.
3. LLVM cannot easily be made to perform some important optimisations and generate specific code that a domain-specific JIT compiler can.

The first two restrictions all point to a region-based DBT having to live *outside* the native VM, and inside the hypervisor component, but this would lead to additional overhead when translating guest instructions. This would be due to communication overheads between the hypervisor and execution engine when regions are to be compiled. However, as will be shown in the evaluation section, the choice of a synchronous basic block-based DBT strategy does not impede the ability of CAPTIVE to operate at a high guest instruction throughput and improve significantly over state-of-the-art. Future work would be to integrate the region-based DBT strategy described in the previous chapters in CAPTIVE, resulting in even more performance gains.

#### 6.3.3.1 Translated Code Re-use

QEMU has implemented an advanced caching strategy that initially uses a fast first-level cache indexed by virtual address to look up the code associated with

a guest basic block, which is invalidated when page mappings change. As basic blocks are translated for specific *virtual* addresses, the virtual PC may be constant-folded into the translations. However, the translation cannot be reused if the same physical address is mapped to different virtual addresses. To handle this situation, when a miss occurs in the first-level cache, a second-level cache that is indexed by virtual PC, physical PC and memory access flags is consulted. If this cache misses, then the guest basic block is translated. This results in a guest basic block being translated for each distinct virtual address mapping. QEMU can suffer this penalty because it has a very fast JIT compiler, it is very cheap to produce a translation. In contrast, CAPTIVE always indexes the code cache by physical PC, and translates code in a way that is independent of its virtual address, meaning the same translation can be used for multiple virtual addresses. The JIT in CAPTIVE contains more optimisations than QEMU, and so naturally takes a slightly longer time to produce a translation. But, this compilation latency is ameliorated by the high quality of code produced, and the ability to keep translations around for a long time.

#### 6.3.3.2 Domain-specific JIT Compiler

In DBT terms, the CAPTIVE JIT compiler operates in a similar fashion to the ARCSIM JIT compiler, by translating guest instructions into an *intermediate representation* (IR), optimising the IR and finally lowering the IR into host machine code. However for the reasons described above, the CAPTIVE JIT is not based on LLVM. In fact, it is a new JIT compiler built from scratch specifically for the task of DBT. GENSIM is still used to generate the architecture-specific component of the JIT compiler (i.e. the guest instruction to IR translator) from the high-level architecture description, but instead of generating a JIT module that produces LLVM IR, a new *domain-specific* IR is produced instead. This IR is highly amenable to code generation for a DBT system, for example, it contains instructions that directly work with the guest register file, eliminating the need to perform special alias analysis to distinguish classes of memory access.

#### 6.3.3.3 Guest Instruction Translation

The CAPTIVE execution engine compiles *guest* basic blocks at a time, but will extend to a trace-based approach if the branch targets are static and land on

**Listing 6.1:** ARM guest basic block

```
ldr   r3, [r4]        ; Load value from memory
cmp   r3, #0x1000000  ; Compare value
movcs r0, #1          ; Set r0 to 1 (if carry)
bcs   5412dc          ; Branch if carry
```

**Listing 6.2:** x86 host basic blocks

```
    mov   0x10(%rdi),%eax
    mov   (%rax),%eax     ; Load from memory
    mov   %eax,0xc(%rdi)
    sub   $0x1000000,%eax ; Compare to constant
    setae 0x140(%rdi)
    seto  0x143(%rdi)
    sete  0x141(%rdi)
    sets  0x142(%rdi)
    lea   0x8(%r15d), %r15d ; Increment PC
    cmpb  $0, 0x140(%rdi)
    jnz   1f                ; Skip if not carry
    movl  $0x1,(%rdi)       ; Set r0 to 1
    lea   0x4(%r15d), %r15d ; Increment PC
    lea   -0x1d8(%r15d),%eax
    and   $0xfffffffe,%eax
    mov   %eax,%r15d  ; Set PC to branch target
    jmp   2f
1:  lea   0x8(%r15d), %r15d ; Increment PC
2:  // Epilogue
```

**Listing 6.3:** Execution Engine IR

```
    // Load r4
b0: ldreg       i4 $0x10, i4 v0
    // Read memory
    ldmem       i4 v0, i4 v0
    // Store value in r3
    streg       i4 v0, i4 $0xc
    // Subtract, and update
    // guest flags
    sbc flags   i4 $0x1000000,
                i4 v0,
                i1 $0x1
    inc-pc      i4 $0x8
    // Read carry, and branch
    ldreg       i4 $0x140, i1 v0
    branch      i1 v0, b1, b2

    // Set r0 to 1
b1: streg       i4 $0x1, i4 $0x0
    inc-pc      i4 $0x4
    // Calculate branch target
    ldpc        i4 v0
    sub         i4 $0x1d8, i4 v0
    and         i4 $0xfffffffe,
                i4 v0
    streg       i4 v0, i4 $0x3c
    jmp         b3
    // Increment PC
b2: inc-pc      i4 $0x8
b3: ret
```

**Figure 6.7:** Example inputs and outputs during the JIT compilation phase of CPU virtualisation. ARM guest code is initially translated to an internal representation for optimisation, before x86 host code is generated and emitted.

the same *guest* memory page. Guest basic blocks are terminated at page boundaries for memory protection purposes. Normal control flow out of a block is optimised utilising techniques presented in Chapter 4, which includes directly chaining to other basic blocks that are part of the same memory page to avoid costly returns to the main execution loop. If a translation does not exist, or the destination does not live on the same page, control is returned to the main execution loop, which will then handle the situation accordingly. A general purpose x86 host register (specifically `%r15d`) is dedicated to tracking the guest PC, instead of keeping this value up-to-date in the emulated register file. This significantly improves performance by avoiding an increment to a memory location (i.e. the emulated guest register file) on each instruction, and leads to the generation of efficient PC-relative load instructions.

As in ARCSIM, GENSIM is used to generate the instruction decoder, and corresponding translation functions. This means that the partial evaluation tech-

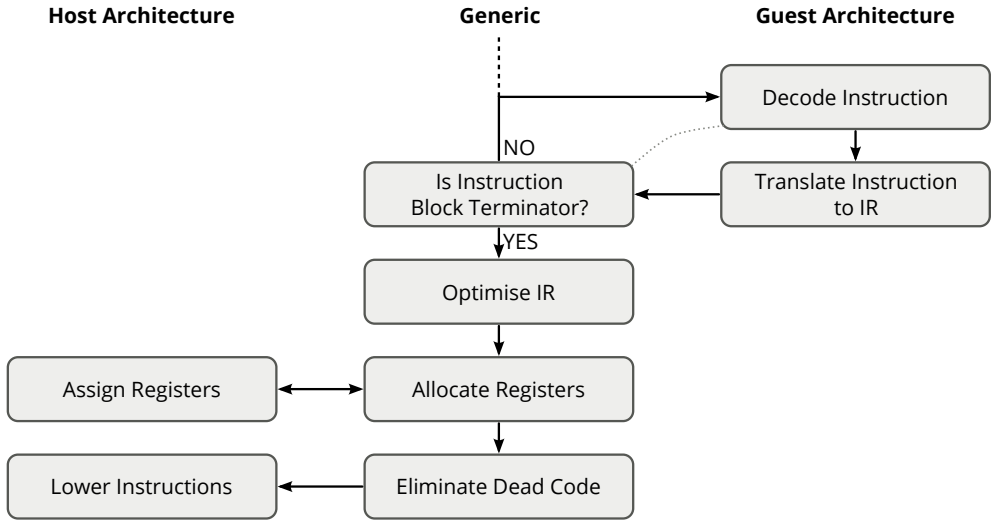



**Figure 6.8:** An overview of the operation of the CAPTIVE JIT when translating a guest basic block into host machine code.

nique described in [121] is again used to constant-fold values known at compilation time into the IR.

Figure 6.8 gives a high-level overview of the actions performed when a translation needs to be produced. The execution engine calls the guest architecture specific instruction decoder to decode a single instruction, then calls a translation function to translate the particular instruction into CAPTIVE's domain-specific IR. Then, the next instruction is decoded and the translation continues. This happens in a loop until the decoder indicates that the instruction was an end-of-block instruction (or a page boundary has been reached). After producing the IR that represents the basic block being translated, the JIT then applies a series of optimisation passes. These passes are designed to operate quickly on the IR, to reduce compilation latency. The passes are selected and implemented specifically to deal with the domain-specific IR, and consist of:

- Jump threading
- Dead code elimination
- Basic block merging
- Constant propagation
- Live value re-use

After this initial optimisation run, a linear-scan register allocator allocates registers, with assignment dealt with by the host-architecture specific code. A

final dead code elimination pass is performed before the IR is passed to a host-architecture specific instruction lowerer. The lowering pass is template-based to optimally match sequences of IR instructions to corresponding host instructions, and directly emits host machine code—in this case it lowers CAPTIVE IR into x86 instructions.

Listing 6.1 shows an example ARM guest basic block that is encountered during the Linux kernel boot process. The IR emitter iterates over this block and after the optimisation phase produces the IR shown in Listing 6.3. Finally, a quick template-based lowering pass produces the native x86 machine code shown in Listing 6.2.

It can be seen here that multiple *host* basic blocks are produced from a single *guest* basic block. In this example, this occurs because of a predicated ARM instruction (`movcs`) that may or may not be executed, depending on the current state of the flags. Since predicated instructions are not classed as basic block terminators, additional control flow is required to account for this.

#### 6.3.3.4 Privilege Level Tracking

Given that the execution engine operates in a bare-metal environment, it has full control of an x86 machine and so exploits system-level features that are not normally available to a user-mode process running in the confines of an operating system.

A particular feature that CAPTIVE can exploit is the ability to switch the virtual x86 CPU into privileged (*ring 0*) mode, and into user (*ring 3*) mode. ARM platforms also have two privilege levels, *PL0* which is the lowest privilege level and *PL1* which is the highest privilege level.[3]

This feature is used to execute translated guest code in the corresponding privilege mode that it would be running in on a real ARM CPU. For example, ARM guest code that runs in ARM *PL0* is executed in x86 *ring 3*, and code that runs in ARM *PL1* is executed in x86 *ring 0*. This mapping of guest platform privilege levels to host system privilege levels enables CAPTIVE to utilise the user/kernel memory protection available in x86 page tables, by mapping it to the corresponding ARM page table permissions. Switching between *ring 0* and *ring 3* is implemented with the fast system call instructions (`syscall` and

[3]There is actually a third privilege level *PL2* for use by hypervisors, but supporting nested virtualisation is not considered in this chapter.

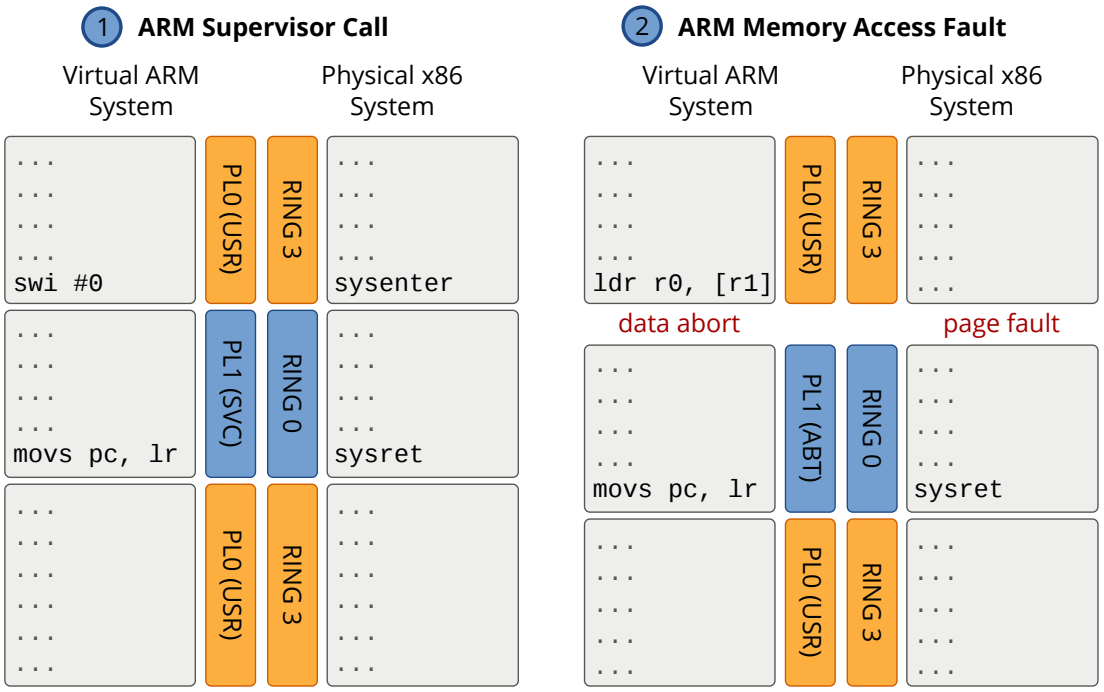



**Figure 6.9:** An example of guest system mode tracking for two different scenarios. ① tracks the mode when an ARM `swi` instruction is issued, and ② tracks the mode when an ARM memory access faults. Even though there are multiple execution modes, an ARM system still uses only two privilege levels, *PL0* and *PL1*. These privilege levels are mapped respectively to the corresponding x86 privilege levels *ring 3* and *ring 0*.

`sysret`) available on modern x86 processors, avoiding the overhead associated with a software interrupt instruction (`int`).

Figure 6.9 shows two situations where the ARM guest system changes mode in response to some event, and so the privilege level of the native VM is changed to match. In Figure 6.9 ①, when the ARM guest system makes a *supervisor call* (with the `swi` instruction), `SVC` mode with privilege level *PL1* is entered. A `sysenter` instruction is generated by the JIT to transition into x86 *ring 0*. When the guest system returns to `USR` mode with privilege level *PL0* (using the `movs pc, lr` instruction), an x86 `sysret` instruction is used to transition back to *ring 3*.

Figure 6.9 ② shows how the mode changes when a memory access fault occurs. In this case, a memory instruction (`ldr`) has attempted to access a memory address that is not allowed, and so an ARM *data fault* occurs (along with a corresponding x86 *page fault*). To handle this, the ARM system enters `ABT` mode, and the native VM enters *ring 0*. Upon return from the *data access*

*abort handler* (which is the ARM equivalent of the x86 *page fault handler*), the guest returns to USR mode, and the native VM returns to *ring 3*.

#### 6.3.3.5 Exploitation of Architectural Features

CAPTIVE makes use of the general purpose *segment register* FS to point to a per-CPU data structure, which contains the state of the emulated ARM CPU. In future work, this will help to enable multi-core virtualisation. Here, it conveniently keeps a pointer to the CPU state available, which is a structure that is frequently accessed by translated code. Using a segment register for this purpose keeps the general purpose registers free for use by the JIT compiler, and the compiler will produce instructions that use the FS register when required. The GS segment register is employed for efficient user-mode emulated memory accesses as described in Section 6.3.4.4.

Another architectural feature that can be utilised is the x86 *call gate* mechanism. This feature is only available to privileged applications (e.g. operating systems) as it requires inserting an entry into the *global descriptor table* (GDT). Originally, call gates were intended to be used to implement system calls, as they allow (controlled) arbitrary changes to the *current privilege level* (CPL). CAPTIVE does indeed use them for this purpose, and such calls are generated by the JIT for invoking helper functions from user mode that require kernel mode permissions. The use of call gates is again an alternative to the slower software-interrupt based mechanism (i.e. using the `int` instruction), as the fast system call mechanism is already used by the execution engine to implement fast privilege-level switching.

#### 6.3.3.6 Code Cache

In order to improve execution performance, translated guest basic blocks are kept in a *code cache*, indexed by physical address. The benefit of using physical addressing is that if and when the guest system's page tables are invalidated, the translated code does not need to be invalidated, as the translation is still valid. A cache indexed by virtual address would need to be invalidated each time the guest page tables change, since virtual addresses across a page table change can point to different physical pages (and hence to different guest code). A downside to this approach is that the guest's *program counter* (PC) contains a

virtual address, and so to lookup the corresponding translation from the cache requires converting the virtual PC into a physical PC, which in the worst case would require a walk of the guest's page tables. However, employing same-page block chaining helps to avoid this particular cost.

#### 6.3.3.7 Self-modifying Code

The only time that translated code needs to be invalidated is in the presence of self-modifying code, or more generally when a page that has previously been executed is written to. To detect this occurrence, physical pages that have been executed are marked with a flag, and those pages are protected from being written to. When a page fault occurs because of a write, and the page has been flagged, all cached translated code corresponding to that page is invalidated. A special case is if the memory fault was to an address that is within the page that is currently executing. This situation means that the translation that is being executed may no longer be valid, as guest instructions that make it up have potentially been modified. Instead of returning to executing the translation (as would normally happen in the simple case), the memory access is emulated (to ensure the write is performed) and then execution returns to the main execution loop via a non-local jump.

### 6.3.4 MMU Virtualisation

One of the most important requirements for hardware virtualisation is the faithful emulation of the *memory management unit* (MMU), which if implemented incorrectly will lead to an unusable system, and if implemented poorly can lead to severe performance penalties. Hardware extensions for same-architecture virtualisation provide accelerated means of virtualising the MMU of a guest machine on the host, but a problem arises when virtualising a guest with a different architecture. As described in the motivating example (Section 6.1.2), the MMUs between two different architectures behave quite differently, and traditional cross-architecture hardware virtualisation uses a (correct, but slow) software MMU implementation to emulate this subsystem. Thus, much work has been done [125, 31, 60] in the area of software MMUs to reduce the address translation penalty and hence increase overall throughput of the virtualisation system.

Fundamentally, the function of the MMU is to translate a *virtual address* to a *physical address*, applying any permissions that may be defined for that access. Usually, this mapping is represented with *page tables*, with various levels of indirection to suit the granularity of the mapping. Hardware virtualisation requires that every instruction that accesses virtual memory is subject to the behaviour of the MMU. For the same-architecture case, memory instructions are mapped one-to-one, and the hardware extensions take care of performing the virtual-to-physical translation and permission checking, but for cross-architecture virtualisation, each memory access must be emulated in such a way as to perform the address translation and permission checking subject to the behaviour of the guest platform.

Software approaches when faced with a memory access (in the base case), will manually traverse the guest page table to resolve the physical address, and check that the access satisfies the permissions imposed by the translation. These accesses will be subsequently sped up by introducing a software cache, much like a software *translation lookaside buffer* (TLB), so that future memory accesses do not incur a penalty of a costly page table walk. When the guest page tables change, the software TLB will be flushed, and the process will start again.

In the unconstrained environment of the native VM, CAPTIVE has full control of the x86 MMU, and uses it to reflect the mappings of the guest, allowing unmodified guest virtual addresses to be used by the execution engine, to enforce the same memory access permissions, and to emulate the memory access with a single native instruction.

**Definition 17** (Native MMU)**.** *The **native MMU** is the MMU that is part of the **Native VM**. In this example, the MMU is an x86-64 MMU, which has a 4-level hierarchy.*

**Definition 18** (Native Page Table Entry)**.** *A **native page table entry** is an entry in the page table of the **native MMU**.*

**Definition 19** (Guest MMU)**.** *The **guest MMU** is a (software) implementation of the MMU that is part of the guest machine. In this example it is an implementation of an ARMv7-A MMU. It is implemented as a service that takes a virtual address (along with access permissions), and returns either success (along with the corresponding physical address and a bitmask of allowed permissions), or failure (along*

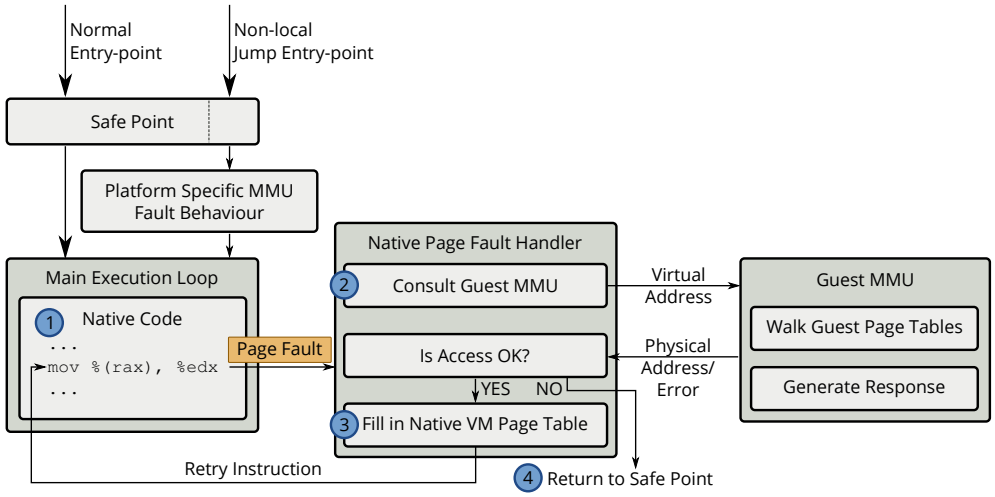



**Figure 6.10:** Operation when virtualising memory accesses. A memory access ① is performed as a single native instruction, which when accessing a virtual address for the first time will cause a page fault in the Native VM. The native page fault handler will ② consult the guest MMU implementation, to determine if the mapping is valid, then either ③ fill in a native page table entry, or ④ perform a non-local jump from the page fault handler back to a safe point to invoke platform specific memory fault handling.

*with the type of failure)*.

CAPTIVE's approach to cross-architecture virtualisation of the MMU is to present the lower 4GB (i.e. virtual addresses `0x0` to `0xffffffff` in the native VM's 48-bit address space) of virtual memory to the execution engine, as the 4GB ($2^{32}$) of virtual memory required for the 32-bit guest machine (see Figure 6.13). This area is now an exact 1:1 mapping of guest virtual addresses to native VM virtual addresses.

Figure 6.10 shows how the various components work together. When a memory access from the guest is emulated (whether a load, store or fetch), that access is performed on the *unmodified* memory address directly, which will of course (for the 32-bit system being virtualised) lie in the lower 4GB region. The first time a memory address is accessed, it will cause a page fault inside the native VM, and at this point the software implementation of the guest's MMU is consulted. The response is either the corresponding *guest* physical address, or a fault condition. If the access is to be allowed, the x86 page table of the native VM is populated with an entry that maps the associated virtual page to the corresponding physical page of the guest, and execution returns to retry the

memory instruction (which will now succeed). Further accesses to this page will no longer fault and will go via the native VM's page tables, and the hardware TLB.

To improve performance, when the guest MMU is asked for the translation, it also returns the allowed permissions associated with that mapping (as defined by the guest pages tables), so that the native VM's page tables can be pre-populated with this information. This means that a read to a page that is also permitted to be written to will only fault once—the first time it is accessed.

On a 64-bit x86 machine, there are four levels of page tables, which shall be referred to as **L4** thru **L1**. The first entry in the (top-level) **L4** page table represents the lower 0–512GB of virtual memory, and this region is reserved for the entire 4GB virtual address space of the guest, starting at virtual address 0. This simplification enables the use of a convenient feature of the x86 MMU, in that each level of the x86 page table can specify permissions that govern the lower levels. Therefore, to protect the whole 4GB guest virtual memory space, only the flags of the first entry in the **L4** page table need to be marked as protected.

If the guest system alters the content of its page tables, just as on actual hardware it is required to issue a TLB flush instruction, which is intercepted and used as a signal to invalidate the native VM's page tables. Using the native VM page table hierarchy, access to the entire lower 4GB area of virtual memory is denied, by clearing the *page present* flag in the first entry of the **L4** page table (followed by a native TLB flush). This makes invalidations very quick to perform. The next time a memory access happens, a page fault will occur, and the page tables will be rebuilt. Normally, this invalidation technique also applies when the guest changes the value of their own page table base pointer (which involves an implicit TLB flush), but can be optimised if the guest supports an *address-space identifier* (described in Section 6.3.4.2).

#### 6.3.4.1 Privilege Level Access Permissions

As mentioned in Section 6.3.3.4, the privilege level of the native VM is matched with the privilege level of the guest. Tracking the mode in this way enables an optimisation to be made when performing permission checking on guest memory accesses, in that pages in the native x86 page tables can be marked

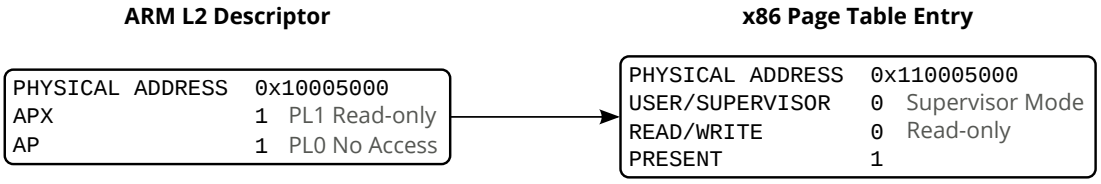



**Figure 6.11:** An example mapping of an ARM L2 descriptor to an x86 page table entry. In the ARM descriptor, read-only permission is granted when operating in a *privileged* mode, and access from *unprivileged* mode is denied. This is reflected exactly by the corresponding x86 page table entry, where permission is not granted in *ring 3*, and write access is denied.

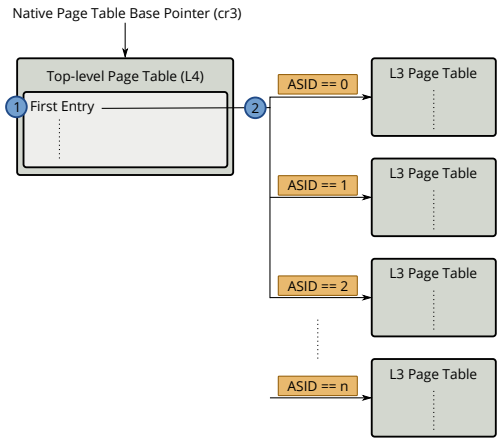



**Figure 6.12:** The top-level (**L4**) page table remains static, and the pointer to the **L3** page tables ① are tracked with the ASID ②.

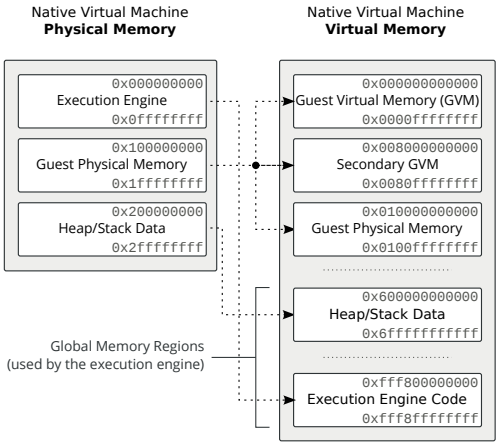



**Figure 6.13:** Native VM *Physical* and *Virtual* memory organisation. The bottom 512GB contains the entire 4GB virtual address space.

with the same privilege level that exists in the ARM page tables.

Figure 6.11 shows an example of this. If an entry in the ARM page table specifies that a page is *read-only* and only accessible in *PL1*, then the corresponding x86 page table entry that will be created for this mapping will have both the `READ/WRITE` and `USER/SUPERVISOR` flag cleared.

#### 6.3.4.2 Address-space Identifier

Usually, changing the page table base pointer naturally causes a TLB invalidation, as the previous mappings are no longer valid. However, since the page table base pointer is changed on every context switch, this can lead to a severe performance penalty, especially in this virtualisation environment when the na-

tive page tables need to be rebuilt each time. An approach to reduce this penalty is described by [125] as "Private SPT", which utilises the ARM address space identifier (ASID) register to quickly switch between pre-populated mappings.

Inspiration is taken from this approach, and CAPTIVE uses the ASID register to point to multiple **L3** mappings, as shown in Figure 6.12. The top-level native page table (the **L4** page table) remains static, but when the current ASID is changed by the guest, the base pointer to the **L3** page table (in the first slot of the **L4** page table) is replaced, and the native TLB is invalidated. As previously described, the first entry in the **L4** page table is used solely for the purpose of managing guest virtual memory, so even though it represents an address space >4GB, it simplifies both the fast invalidation technique, and changing the corresponding page tables that represent the guest 4GB address space.

If this is the first time the ASID has been seen, the normal page-fault lazy resolution process will occur as described previously, but if the ASID has already been encountered, the page tables already contain mappings ready to be used (unless they were explicitly invalidated), without incurring any page faults.

The special invalidation instructions issued by the guest are trapped in order to invalidate TLB entries by ASID, and these signals are used to invalidate the page tables that are associated with that particular ASID.

This optimisation only holds for guest platforms that have the concept of an ASID, and guest kernels that actually use it (a limitation also encountered by [125]). However, it is possible to extend this approach to track the guest platform's page table base pointer, and maintain a set of mappings for "seen" page table bases.

#### 6.3.4.3 Native VM Memory Layout

As described previously, CAPTIVE has full control over the native VM's virtual memory space, and so exploits this opportunity for manipulating the virtual page mappings arbitrarily. Page mappings are established for the execution engine and heap/stack data areas, and these entries are marked as *global*, so that they are not flushed from the TLB when the TLB is explicitly flushed. A one-to-one mapping of *guest* physical memory is installed in the virtual memory space so that data can be accessed by guest physical address. This mapping is particularly useful for the emulated MMU, as it uses physical address pointers

to traverse the guest page tables.

#### 6.3.4.4 Secondary Guest Virtual Memory

The secondary guest virtual memory mapping is part of an optimisation for handling ARM `ldrt` and `strt` instructions, which perform memory accesses subject to user-mode memory permission checking, whilst executing in kernel mode. These instructions are notoriously difficult to optimise [40], as they invoke behaviour that must be specially handled. As they are defined, there is no direct mapping of this behaviour from an ARM system to an x86-64 system, however to maintain performance a second region of guest virtual memory is employed to optimise these accesses specially.

Since it is known at JIT compilation time that a particular memory access has these special semantics, an optimised `mov` instruction is emitted, that offsets the virtual memory address against a base pointer held in the x86 `GS` register. This base pointer points to the base of the second virtual memory region, and so all memory accesses are made into this second region. Then, when a page fault occurs CAPTIVE applies the appropriate semantics when faulting the page in. Whilst this may sound like a guest architecture-specific optimisation, it is implemented independent of the target architecture, and so may be used (or not) by any platform that requires it. It also opens up scope for more exotic permission combinations that can not be mapped from a guest system to the host system.

#### 6.3.4.5 Comparison to QEMU

QEMU uses software-based MMU virtualisation, and Listing 6.4 shows an example ARM instruction that accesses memory, from a PC-relative offset. This instruction loads a value from memory, residing at the address $PC + 92 + 8$. Listing 6.6 shows the QEMU generated native code for this single instruction, which involves accessing a software cache, with a branch to a handler if a cache miss occurs. The output code (shown in Listing 6.5) from CAPTIVE consists of performing the memory access directly on memory itself, using the unmodified value from the guest instruction.

The other slight difference is the optimisation performed for a PC-relative lookup. In QEMU's case, it can constant-fold the address of the memory access

**Listing 6.4:** ARM input assembly

```
; Read memory at address PC + 92 + 8
; (0x100a0) into r0
ldr  r0, [pc, #92]
```

**Listing 6.5:** CAPTIVE output assembly.

```
; Read memory from PC + offset + 8
mov    0x64(%r15d), %eax
; Store into r0
mov    %eax, (%rdi)
; Increment PC
lea    0x4(%r15d), %r15d
```

**Listing 6.6:** QEMU output assembly.

```
; Prepare memory address
mov    $0x100a0,%ebp
mov    %rbp,%rdi
mov    %ebp,%esi
; Calculate cache entry address
shr    $0x5,%rdi
and    $0xfffffc03,%esi
and    $0x1fe0,%edi
lea    0x2c18(%r14,%rdi,1),%rdi
cmp    (%rdi),%esi      ; Check cache tag
; Restore destination address
mov    %ebp,%esi
jne    0x7f4d682a718f   ; Cache-miss?
add    0x10(%rdi),%rsi
mov    (%rsi),%ebp      ; Read memory
mov    %ebp,(%r14)      ; Store into r0
```

**Figure 6.14:** An example of a PC-relative load instruction being translated by CAPTIVE and QEMU. CAPTIVE tracks the (virtual) `PC` in `%r15d`, and emits three instructions for this memory access whereas QEMU emits 13 instructions that involve interrogating its address cache.

(`0x100a0`) into the generated assembly because it generates basic blocks for virtual pages. However, as CAPTIVE generates basic blocks for physical pages (which may be accessed by any virtual address), a virtual address cannot be constant folded in. But, since the guest `PC` is always mapped to a host register, this improves code quality and adds virtually no performance penalty. This improvement in code quality is not because of an improvement in the quality of the JIT itself, but rather that memory accesses can be made in this fashion.

### 6.3.5 Device Virtualisation

In order to faithfully emulate a guest platform, the devices present on that platform must also be emulated. Such devices may be timers, interrupt controllers, I/O devices, etc. In order to do this, the hypervisor component of CAPTIVE contains software emulations for the various devices that make up the platform. On a *real* guest platform, these devices are accessed by the guest through the memory subsystem; they are mapped into the physical memory space (and then mapped by the guest operating system into the virtual address space) and device registers are written to and read from with normal memory accesses. This approach to device communication increases flexibility (e.g. device accesses are

subject to MMU translations and permission checks), and reduces complexity for operating systems, but adds a layer of complexity to virtualisation frameworks wishing to emulate devices in a particular platform, as they must detect these accesses to device memory, and handle them accordingly.

As the CAPTIVE device implementations live in the hypervisor (i.e. outside of the native VM), memory accesses by the guest must be trapped back to the hypervisor, so that they can be forwarded to the particular device being accessed. The most straightforward way to accomplish this with CAPTIVE would be to use the *memory-mapped I/O* (MMIO) feature of KVM to intercept memory accesses to regions of guest physical memory that correspond to devices, and handle them accordingly. This approach works well, but suffers from a severe performance penalty, as every access to a device must perform a costly **VM exit**, then the native guest instruction must be emulated by the hypervisor to fill in the data that was read, or to extract the data that is to be written.

Device accesses in a full-system occur quite frequently. For example, a Linux system configured with a 100Hz timer will be interrupted 100 times a second, and each interrupt requires the guest to interrogate the interrupt controller device to ascertain the cause of the interrupt, then the timer device to read timing related data, then write to the devices to acknowledge and complete the interrupt.

Another approach is to make a hypercall using *port-based I/O* (PIO) instructions, which have slightly faster VM exit sequences, but this suffers from a fundamental problem: detecting a device access. As mentioned previously, a device access to a memory-mapped device is indistinguishable from a normal memory access at the instruction-level—it is performed with a normal memory access instruction (e.g. `ldr` in ARM). Therefore, CAPTIVE needs to detect accesses to device memory, and trap to the host using a faster *hypercall* mechanism. Utilising the native VM's MMU again (and armed with the knowledge of the locations of devices in physical memory—which is part of the platform configuration) any device page is marked as inaccessible, so that every memory access traps in the native VM, rather than in the hypervisor.

Now that page faults are being received in the native VM (which is faster than trapping to the hypervisor), there are two approaches to take:

1. Translate the device access into a (slightly) faster PIO access, which still results in a VM exit, or

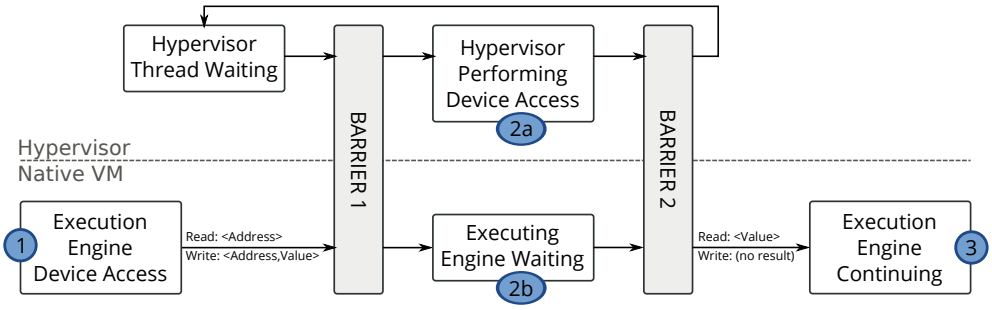



**Figure 6.15:** An illustration of the fast device access operation. When a device access is made ①, barrier 1 is entered by the guest (at which the host is already waiting) and the host performs the access on the emulated device (2a). Meanwhile, the guest waits for the host to complete the operation (2b). Then, when the access is complete, barrier 2 is entered by the host and execution by the guest continues ③.

2. use a message-passing implementation to communicate with the hypervisor, avoiding a VM exit.

It is desirable to avoid VM exits as much as possible, as they introduce a significant amount of overhead [95]. A VM exit with Intel VT and KVM requires storing the entire state of the virtual machine, and performing a context switch back to user-space code. Returning to the VM (a VM entry) involves restoring this saved state.

For this reason, CAPTIVE implements (2), and once the native VM receives a page fault to a device memory page, a synchronisation barrier system is used to communicate with a hypervisor thread. This avoids a costly VM exit, as the virtualised CPU is simply spinning on a barrier, waiting for a response from the hypervisor. This sequence is shown in Figure 6.15. When a device access is to be made Fig. 6.15 ①, a data structure is prepared by the execution engine inside the native VM, and a synchronisation barrier is entered. A hypervisor thread (which is already waiting on this barrier) resumes execution and deals with the device access request Fig. 6.15 (2a). Meanwhile, the guest waits on a second barrier Fig. 6.15 (2b) whilst the hypervisor is servicing the request, and when the request is complete, the hypervisor writes the result back into the data structure, and enters the barrier. This causes the execution engine to resume execution Fig. 6.15 ③, extracting the necessary data from the request structure. The guest cannot proceed until the hypervisor has signalled that the data has been processed by the emulated device, and this is the reason for the second barrier.

#### 6.3.5.1 Speculative Instruction Rewriting

A further optimisation opportunity is *speculative device access instruction rewriting*, where the (native) instruction that caused the device access is rewritten in the hopes that it is only ever used to perform a device access. The rewritten instruction will invoke the behaviour to perform the device access immediately, instead of trapping in the memory management system. This speculative rewriting was implemented in CAPTIVE to test the idea, and it was actually observed that speculation was correct 100% of the time, however, there are two major flaws with this approach:

**Instruction Size:** The limited size of the native memory instructions (which in the worst case was two bytes) meant that the instruction could only be rewritten to something of equal (or smaller) size.

**Address Translation:** Device accesses are performed on physical addresses, but the memory instructions operate on virtual addresses, and so a costly virtual-to-physical translation must be performed for each access.

Each of these flaws are now considered in more detail.

**6.3.5.1.1 Instruction Size** The guest memory access instructions generated by the JIT are typically x86 `mov` instructions (although other instructions can have memory operands these are not produced), and the smallest memory access instruction is three bytes in length. These three bytes contain a one byte prefix, a one byte opcode, and a *ModR/M* byte that specifies the operands. Occasionally, the instructions are longer if they use registers that require additional prefixes, or additional encoding, but since the smallest size is three bytes, the instruction rewriting approach is restricted to three bytes for encoding a new instruction.

The new instruction must be chosen so that control-flow is diverted to the device access handler, but it must also encode the original operands of the memory access, so that the original instruction can be emulated. Therefore, the only suitable instruction to use would be an architecturally undefined instruction (which is one byte in length) to replace the prefix byte, leaving the second and third byte unchanged. Using an architecturally undefined instruction would cause an *illegal opcode exception* to occur when it is encountered,

and so a special exception handler can be used to perform the device access. However, trapping an illegal opcode is expensive, as it requires storing the machine state on the stack in order to enter an *exception handler frame*. This is also a fundamental problem with instruction rewriting anyway, as to call a device access function would require saving and restoring live registers to and from the stack.

**6.3.5.1.2 Address Translation** The location of devices in physical memory is specified by the platform configuration, but accesses to the devices are made with virtual addresses. Therefore, as part of the device access handler, the virtual address must be translated to a physical address. As shown in previous sections, this operation is costly to perform.

#### 6.3.5.2 Device Implementations

Unlike traditional same-architecture virtualisation, where the possibility exists to *para-virtualise* hardware that exists on the host for use by the guest, or simply pass-through real hardware devices (e.g. using Intel VT-d) this same kind of mapping does not exist for cross-architecture virtualisation as it is unlikely that there are any 1-to-1 compatible devices available on the host system. Therefore, all guest platform devices are implemented in software, which faithfully emulate the behaviour of the device they represent. An example of a device implemented by CAPTIVE in software is the ARM PrimeCell SP804, which is a two-channel timer device. This device is configured and interrogated by the guest through registers that are memory mapped. It is also capable of raising interrupts when a timeout occurs, depending on the mode of operation of the timer.

Future work would be to investigate mapping similar devices (e.g. a timer device) to existing hardware devices. Even though their interfaces may be incompatible, it may be possible to configure the behaviour of the devices in similar ways and avoid having to use full software implementations of the device.

#### 6.3.5.3 Device Interrupts

Platform devices may raise interrupts to indicate that an event has occurred, such as a timer has timed-out, or data is ready to be read. On a physical plat-

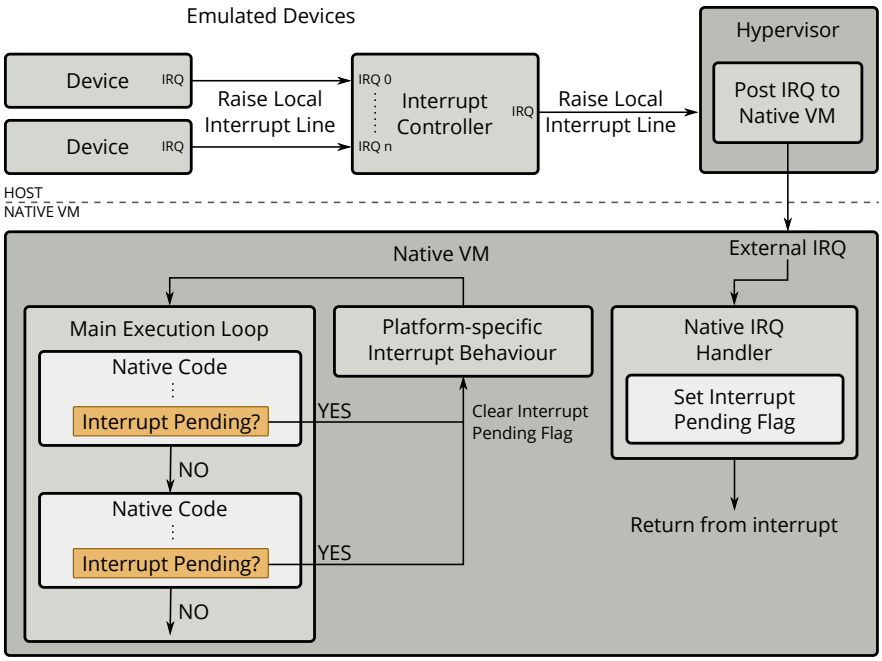



**Figure 6.16:** An illustration of the injection of an IRQ into the native virtual machine, to indicate that an emulated IRQ line has gone high.

form, an interrupt controller would aggregate the individual interrupts from each device, and trigger a physical interrupt line on the CPU, to indicate that an interrupt has been raised. The CPU would enter its external interrupt handling routine, and interrogate the interrupt controller to work out which device(s) raised the interrupt. The RealView Platform Baseboard Cortex-A8 [9] has such a setup with an ARM GIC (generic interrupt controller), that receives interrupts from devices and posts these to the CPU. CAPTIVE implements the GIC in software, but posts real IRQs to the guest system, when the interrupt controller triggers a physical interrupt line on the CPU. This process is described in Section 6.3.6.

### 6.3.6 IRQ Virtualisation

As described in the previous section, emulated devices may issue interrupts to the guest system by means of an interrupt controller. For the platform being virtualised, the interrupt controller is an ARM *generic interrupt controller* (GIC), which aggregates interrupts from other platform devices, and presents them to the CPU.

Fundamentally, the CPU has a single physical interrupt line that is raised

when an interrupt is pending, and lowered when the interrupt is acknowledged. This interrupt line is toggled by the emulated GIC, and is visible to the virtualised CPU. On the rising edge of the interrupt line, a native IRQ is injected into the native VM, to inform it that the line has been raised. These interrupts of course happen asynchronously, e.g. a timer device will run as a separate thread on the host machine, and when its timeout occurs, it will trigger its own interrupt line, propagating through the interrupt controller and into the guest. The ideal situation would be to immediately invoke the platform-specific interrupt handling code, on the rising edge of the interrupt line, but this is not feasible for two reasons:

1. The guest may not be running in translated code (it may be handling a page fault, or performing some "book-keeping").
2. Single guest instructions are generally compiled to multiple host instructions, which means the interrupt may happen part-way through the emulation of a guest instruction.

This is unacceptable, as guest instructions are not necessarily re-entrant and may have partially changed the state of the guest system mid-way through. Guest instructions need to appear to be atomic, and so they must have completed before control-flow can be diverted to the interrupt handling behaviour. This exact problem is described in Chapter 5, and the solution was to place an interrupt check at particular block boundaries. However, since the CAPTIVE JIT is no longer region based, the interrupt checks must be placed at every block boundary.

The implementation in CAPTIVE is similar to ARCSIM, as an *interrupt pending flag* is set when the native VM enters the x86 IRQ handler. This flag indicates that the emulated interrupt line has gone high, and is checked by native code at the end of a guest basic block before it chains to the next. If the flag is set, it is immediately cleared and translated code is left to perform the guest platform behaviour associated with servicing an interrupt.

## 6.4 Experimental Evaluation

This section shall evaluate CAPTIVE's performance by using industry standard benchmarks to compare its performance to the state-of-the-art cross-architecture

virtualiser QEMU. As in previous chapters, the SPEC CPU2006 integer benchmark suite is used. For the key results, the *reference* input set is used, which requires a minor modification to the guest platform to increase the available guest physical memory for running the benchmarks. The amount of physical memory presented to the guest system is independent of the amount of physical memory available on the host system, as it is defined by the platform being emulated. The platform implemented in CAPTIVE is a RealView Platform Baseboard Cortex-A8 [9], which specifies only 512MB of physical memory [10], but this is insufficient for running the reference input set of the benchmark suite. To overcome this limitation, the amount of physical memory in the guest platform is artificially increased to 2GB in both CAPTIVE and QEMU, enabling the benchmark suite to run.

### 6.4.1 Experimental Setup

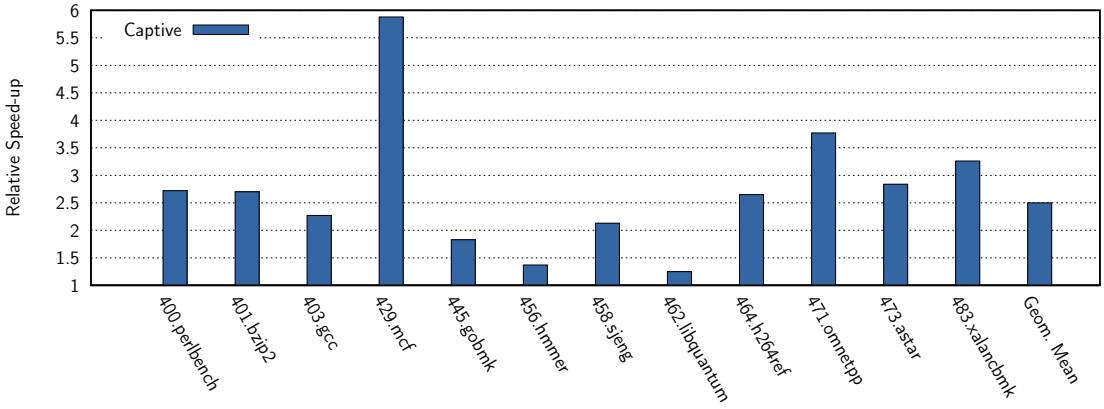
The platform being virtualised is a RealView Platform Baseboard for Cortex-A8, which is fully supported by QEMU. These experiments use a vanilla ARM Linux 4.3.0 kernel, with the default configuration for the platform, except for the addition of a VirtIO block device to provide storage to the guest and an increase in physical memory as described previously. The user-space is Arch Linux ARM. The host machine is described in Table 6.1.

### 6.4.2 Key Results

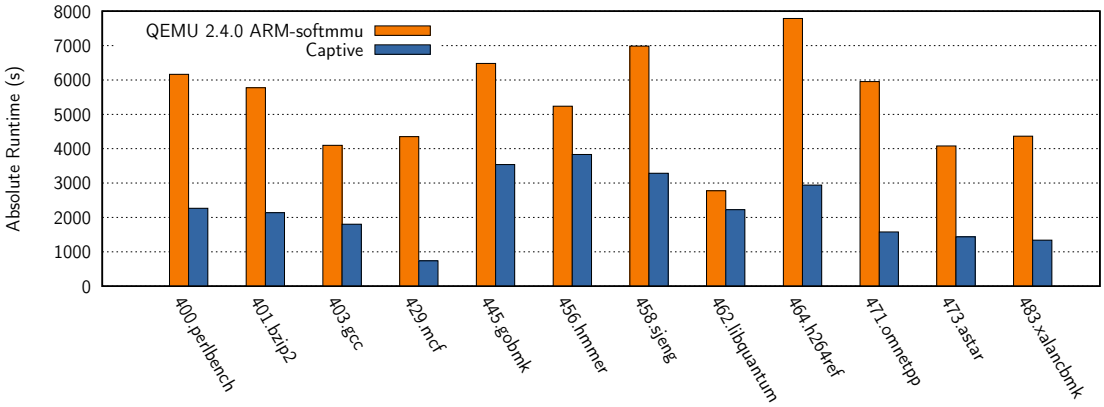
These *key results* compare the performance of CAPTIVE to QEMU version 2.4.0. Figure 6.17a shows the relative speed-up of CAPTIVE, compared to QEMU. In all cases CAPTIVE outperforms QEMU, and on average by a factor of $2.5\times$. Figure 6.17b shows the absolute run-time of each benchmark in seconds.

| **System** | Dell™ PowerEdge™ R610 | | |
|---|---|---|---|
| *Architecture* | x86-64 | *Model* | Intel™ Xeon™ E5-1620 |
| *Cores/Threads* | 4/8 | *Frequency* | 3.50 GHz |
| *L1-Cache* | $1\times$ $4\times$ 32kB (*I$ & D$*) | *L2-Cache* | $1\times$ $4\times$ 256kB |
| *L3-Cache* | $1\times$ 10 MB | *Memory* | 16 GB |

**Table 6.1:** Host configuration.




**(a)** Relative speed-up of CAPTIVE over QEMU, with the SPEC benchmark suite (using the reference input set)–higher is better.




**(b)** Absolute run-time of the SPEC benchmark suite (using the reference input set) in seconds–lower is better.

**Figure 6.17:** Key Results: (a) shows relative speed-up, and (b) shows absolute run time. On average, CAPTIVE is $2.5\times$ faster than QEMU.

Of interest is 429.mcf, which gains a performance improvement of $5.88\times$. This is because the benchmark responds well to the optimising DBT system, which produces highly optimised run-time code based on the dynamic behaviour of the benchmark, versus the static optimisation that is performed at compile time.

Only two out of twelve benchmarks show speed-ups less than $1.5\times$, yet still outperform the baseline QEMU. Given the acceptance of SPEC as a realistic workload, there are multiple characteristics that can affect simulation performance, and it is clear that the range of benchmarks exercise the simulation system in numerous ways, making it difficult to pin-point any particular feature that causes fluctuations in performance.

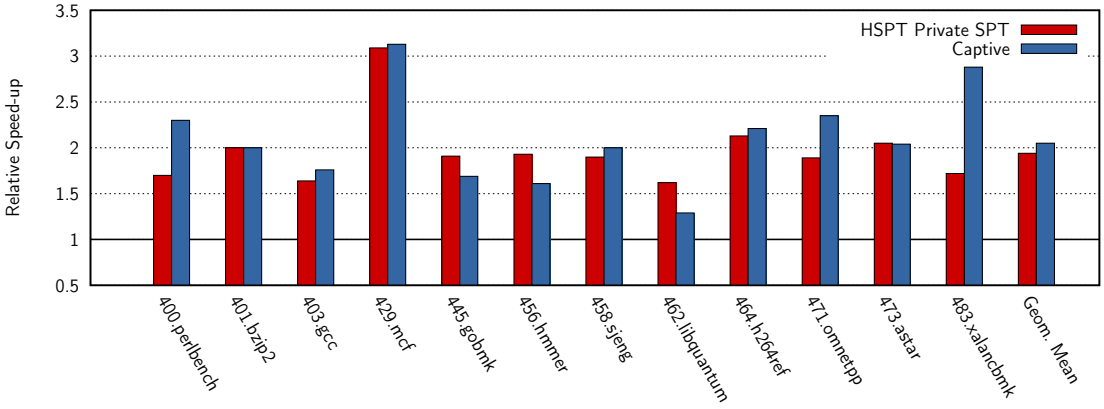



**Figure 6.18:** Relative performance improvement of SPEC benchmarks by HSPT and CAPTIVE over the Android Emulator baseline–higher is better.

### 6.4.3 Comparison to Existing Techniques

One of the most recent efforts to improve memory address translation performance in full-system simulators is presented in [125] (herein referred to as HSPT), which describes a practical implementation of an *embedded shadow page table*, using Linux system calls (specifically `mmap`) to create an efficient *guest*-virtual to *host*-physical mapping similar to this approach, but operating in QEMU, and hence unable to exploit the hardware MMU to its full potential. In order to compare CAPTIVE to the HSPT implementation, the published results from [125] have been extracted, and the same experimental setup has been implemented. Comparing the performance of CAPTIVE to the same version and configuration of the Android Emulator as used by HSPT, enables a relative performance comparison against the same baseline to be presented, even in the presence of different host machines.

Figure 6.18 shows that HSPT has achieved an average improvement of $1.94\times$ (geometric mean) over the Android Emulator, using the *Private SPT* technique, whereas on average, CAPTIVE achieves a performance improvement of $2.05\times$ (geometric mean).

In the majority of cases, CAPTIVE equals or surpasses the speed-up presented by HSPT, in particular 483.xalancbmk in CAPTIVE shows a much greater speed-up of $2.88\times$, compared to $1.72\times$ in HSPT. This is due in part to the I/O nature of this particular benchmark, and the optimised I/O and IRQ handling techniques giving a clear advantage here.

### 6.4.4 I/O Performance

This section aims to evaluate the performance of the I/O virtualisation strategy, using the standard Linux I/O performance measuring tool `hdparm`. The I/O performance of a variety of virtualisation configurations is measured, including taking a measurement on the host system itself. This section also introduces Oracle VirtualBox as another virtualisation platform that uses Intel VT extensions, and as such only supports same-architecture virtualisation. For measurement of same-architecture virtualisation I/O performance, VirtualBox and QEMU are given an x86 Linux distribution containing the `hdparm` tool. For cross-architecture virtualisation, QEMU and CAPTIVE are provided with a file-system that exists as a normal file on the host machine's file-system. For QEMU/ARM and CAPTIVE/ARM, the platform device used to communicate this data back and forth is a VirtIO block device, which is fully supported by both hypervisors. VirtIO is a virtualisation technology that enables efficient paravirtualisation of various platform devices, such as network and disk devices. The emulated disk device in CAPTIVE is based on a VirtIO block device, and is the only paravirtualised device in the platform.

Table 6.2 shows the absolute I/O throughput of the virtualisation configurations, along with throughput on the native host platform, using two distinct metrics: *cached* and *buffered*.

**Cached** reads are subject to the Linux kernel page cache, and as such represent the performance at which disk data can be accessed from the page cache in the guest system. VirtualBox and QEMU/KVM make these accesses at virtually the same rate as the host platform, since there is no virtualisation overhead

| Hypervisor | Execution | Arch. | Cached | Buffered |
|---|---|---|---|---|
| None | Native | x86 | 12384.21 MB/s | 173.52 MB/s |
| VirtualBox | Intel-VT | x86 | 11941.43 MB/s | 91.64 MB/s |
| QEMU | KVM | x86 | 11881.06 MB/s | 102.72 MB/s |
| QEMU | DBT | x86 | 1265.03 MB/s | 79.80 MB/s |
| QEMU | DBT | ARM | 157.02 MB/s | 105.77 MB/s |
| CAPTIVE | KVM/DBT | ARM | 1695.29 MB/s | 155.72 MB/s |

**Table 6.2:** Absolute I/O throughput for various configurations of execution environments. Cached reads are subject to the Linux kernel's page cache, and buffered reads go directly to the real or emulated disk device.

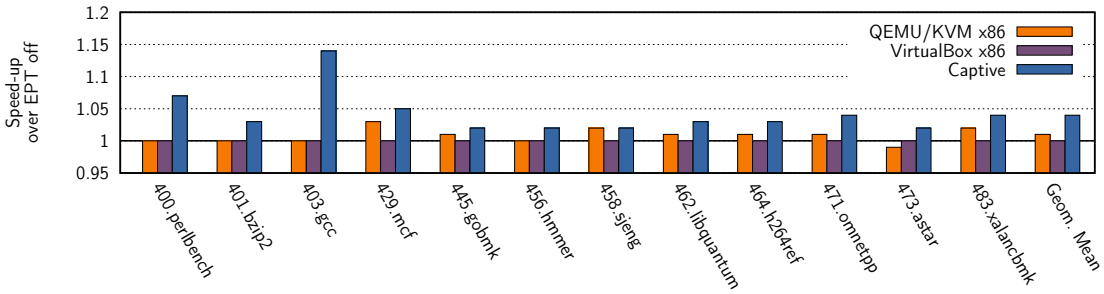



**Figure 6.19:** Relative performance improvement gained by turning on Intel's *extended page tables* (EPT) for the SPEC CPU2006 integer benchmark suite–higher is better. These results show that the use of EPT has virtually no effect on the virtualisation performance for the SPEC CPU2006 benchmark suite.

for memory accesses. For QEMU/DBT, the accesses to this cache are subject to the software MMU implementation, and therefore incur an access penalty. For QEMU/DBT, in the x86-on-x86 case this causes a slow-down of $9.79\times$, and a slow-down of $78.87\times$ for the ARM-on-x86 case. In CAPTIVE, the slow-down is only $7.3\times$, improving over QEMU by $10.8\times$.

**Buffered** reads indicate the rate at which data can be accessed directly from disk—bypassing the kernel page cache. For these experiments, *host* caching was disabled in each hypervisor, causing all accesses to the virtual disk device to go directly to the host file-system, and then onto the underlying storage medium. All hypervisors suffer a slow-down over native for this case, as there will be overhead in accessing the virtual disk on the host file-system, but the slow-down over native for CAPTIVE is only $1.11\times$, compared to QEMU/ARM being $1.64\times$. Virtualisation of the x86 guest machines on VirtualBox, QEMU/KVM and QEMU/DBT all have even worse slow-downs, but this may be due to the implementation of the virtual disk device, which for these three hypervisors is an emulated IDE disk drive, as opposed to the para-virtualised VirtIO device used in QEMU/ARM and CAPTIVE.

### 6.4.5 Additional Hardware Support for MMU Virtualisation

The latest version of Intel VT includes hardware support for managing virtualised guest page tables, which is branded as *extended page tables* (EPT). AMD also have similar support, branded as *rapid virtualisation indexing* (RVI) (formerly NPT). KVM can make full use of this technology, and this section evalu-

ates the performance improvement of EPT over non-EPT backed virtualisation. This evaluation uses QEMU/KVM and Oracle VirtualBox to measure the impact of EPT on same-architecture virtualisation. To produce this data, six distinct experiments have been ran: three with EPT disabled in the respective hypervisor, and three with EPT enabled. Then, the relative speed-up of each hypervisor with EPT enabled, over EPT disabled is presented in Figure 6.19. For QEMU/KVM and Oracle VirtualBox the experiments were naturally made on a virtualised x86-64 system, with x86-64 versions of the SPEC benchmark suite. For CAPTIVE, the same setup as described in Section 6.4.1 is used, with EPT enabled and disabled.

The data shows that in these experiments, EPT does not make any significant improvement to the workloads being tested. This is contrary to some published experiments, e.g. VMware have conducted a performance evaluation of EPT in [118], which shows that EPT can improve performance of MMU-intensive benchmarks by 48%, and MMU-microbenchmarks by up to 600%. However, the measurement of the impact of EPT on the SPEC CPU2006 benchmarks shows that the performance increase to be negligible, which is also the conclusion drawn by [90, 28]. Figure 6.19 shows the relative performance improvement of the SPEC benchmark suite, running on both a virtualised x86 system (using QEMU/KVM and Oracle VirtualBox) and on a virtualised ARMv7-A system (using CAPTIVE). On average, there is virtually no improvement for QEMU and VirtualBox, and only 3% for CAPTIVE.

### 6.4.6 Slow-down over Native Execution on High-End Hardware

This section evaluates the performance of CAPTIVE, compared to execution of the benchmarks on a physical ARM hardware platform. Run times for the benchmarks were collected on an ODROID-XU, and Figure 6.20 shows the relative slow-down of both CAPTIVE and QEMU. On average, CAPTIVE is $1.4\times$ slower than native execution of SPEC on an ARM platform, whereas QEMU is $3.51\times$ slower. Again, of interest is the 429.mcf benchmark that actually shows a speed-up over native due to the JIT compiler discovering optimisations that can be made dynamically, which work better than the compile-time optimisations of the benchmark.

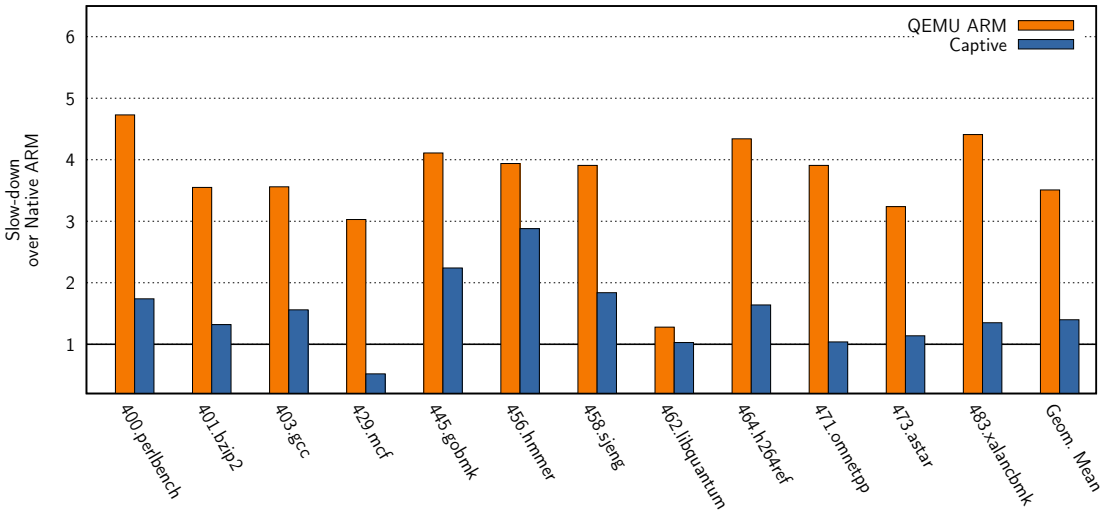



**Figure 6.20:** Relative slow-down of QEMU and CAPTIVE over native execution on a physical ARM platform (ODROID-XU using Samsung Exynos5422 Cortex-A15 2.0Ghz quad-core and Cortex-A7 quad-core CPUs)–lower is better. On average, CAPTIVE is $1.4\times$ slower than the hardware platform, compared to a $3.51\times$ slow-down for QEMU.

## 6.5 Summary & Conclusions

This chapter has introduced new techniques for cross-architecture hardware virtualisation, using hardware accelerated virtualisation extensions originally designed for same-architecture virtualisation, and has implemented these ideas in a hypervisor called CAPTIVE. The key contribution is the mapping of guest system MMU behaviour to host system MMU behaviour, and this improves over the state-of-the-art simulator QEMU on average $2.5\times$.

Although the previous chapters built upon an existing simulator (ARCSIM), the ideas presented in this chapter have required a brand new implementation (CAPTIVE), as ARCSIM was built with software-based virtualisation in mind. This new implementation is a dedicated hypervisor for cross-architecture virtualisation, and benefits from being able to use existing architectural models.

Chapter 7

# Conclusions

This thesis has sought to accelerate cross-architecture hardware virtualisation, by focussing on four main challenges:

1. **Instruction emulation:** Executing guest machine instructions on the host machine.
2. **Interrupt handling:** Correctly diverting guest machine control-flow when asynchronous interrupts are pending.
3. **Memory management unit virtualisation:** Translating virtual memory addresses into physical addresses, as per guest machine behaviour.
4. **Device emulation:** Fully emulating the behaviour of guest platform devices.

Chapter 4 introduced techniques for efficient instruction emulation in a software-based instruction set simulator, followed by the introduction of a particular challenge for efficient full-system virtualisation in Chapter 5. Finally, Chapter 6 exploited hardware acceleration for same-architecture virtualisation, to build a novel cross-architecture hypervisor that outperforms existing state-of-the art cross-architecture hypervisors.

Each contribution has directly led to performance improvements over existing techniques, and opens up the scope for future work in this area. The following sections shall describe the main contributions of this thesis in more detail, followed by a critical analysis of these contributions and closing with possible future work.

## 7.1 Contributions

The underlying theme of this thesis is to improve the performance of cross-architecture hardware virtualisation systems. These systems are particularly important to those in development environments, who want to prototype software for new and existing platforms that are different to their development machine. Existing techniques for this purpose perform well, but can be vastly improved by employing the techniques described.

### 7.1.1 Efficient Dynamic Binary Translation

Chapter 4 described a dynamic binary translation system for the emulation of guest instructions, and contributed a complete strategy for a region-based DBT system suitable for inclusion in a high-speed instruction-set simulator.

Existing region-based solutions do not perform adequate analysis to take advantage of optimisations that can be made within a region, and so branch type profiling information was used to improve back-end code generation, by means of exploiting loop optimisations that could be kept local to the region.

A form of light-weight region chaining, borrowing concepts from trace chaining, improved control-flow dramatically when transitioning between distinct regions of code.

A problem with using an existing JIT compiler was overcome by employing a custom domain-specific alias analysis phase, which identified memory accesses that were part of the instruction execution behaviour, and distinguished them from memory accesses that were part of the infrastructure. Classifying pointers correctly leads to aggressive elimination of dead loads and stores, and hence to much more performance in translated code.

### 7.1.2 Efficient Interrupt Virtualisation

Instruction emulation is one of the challenges faced when performing cross-architecture virtualisation, and Chapter 5 extended the simulation infrastructure to support hardware virtualisation. However, the additional requirements for hardware virtualisation introduce performance overheads in DBT systems, and specifically, the handling of asynchronous interrupts causes adverse control-flow to negate some of the control-flow optimisations made in Chapter 4.

To minimise this disruption, Chapter 5 presented a new scheme for the optimised handling of asynchronous interrupts in the context of a region-based DBT, by optimising the placement of the necessary interrupt checks. The algorithm is efficient and suitable for JIT processing, and does not introduce unbounded interrupt response times. The scheme improves virtualisation performance and I/O throughput in ARCSIM, when virtualising an ARM guest platform running Linux.

### 7.1.3 Hardware Accelerated Cross-architecture Virtualisation

Finally, an observation is made that modern processor vendors support same-architecture virtualisation with hardware extensions that enable unmodified guest operating systems to run in an isolated virtual machine with minimal supervision. These extensions are exploited in Chapter 6 to build a new cross-architecture hypervisor that supports the virtualisation of a guest platform that is significantly different to the host platform.

Tackling four important challenges, Chapter 6 accelerates normally software-based virtual-to-physical address translation by mapping the behaviour of the guest MMU onto corresponding behaviour of the host MMU—despite substantial differences between the two MMUs.

For instruction emulation, a DBT system for the translation from the guest to host ISA is presented, where a fast block-based, domain-specific, *just-in-time* (JIT) compiler that lives *inside* the native virtual machine translates guest instructions to host instructions. The JIT compiler is designed specifically for DBT, unlike a commodity compilation framework such as LLVM, and so can generate highly efficient host machine code quickly. The DBT also takes advantage of the fact that it can generate system-level instructions, and use architectural features (such as privilege levels) to efficiently map guest to host execution behaviour.

Finally, an efficient mechanism to emulate the guest's memory mapped I/O devices is developed, by exploiting the host's MMU to detect device accesses, and using a message passing infrastructure to reduce guest-to-hypervisor communication costs.

## 7.2 Critical Analysis

When implemented, the techniques described clearly provide performance gains over existing implementations, but they require a critical analysis to identify key assumptions made, and possible limitations.

### 7.2.1 GENSIM Limitations

GENSIM is an excellent tool for generating simulation components from an architectural description, but it currently lacks support for architectural features, such as:

1. Branch delay slots.
2. Instruction prefixes, or specifically context-sensitive instruction decoding.
3. Non-standard control-flow (e.g. zero-overhead loops).
4. VLIW architectures.
5. Out-of-order execution.

This means that it is currently unable to express some guest architectures. However, these limitations are purely engineering-based, and given suitable resources the required functionality can be implemented. This may lead to future research in the area of architecture description languages, and automatic simulator generation.

### 7.2.2 Significantly Different Memory Management Units

The behavioural mapping of an ARM MMU to an x86 MMU works well, because the configuration and operation, whilst incompatible, are similar in behaviour. This may not necessarily be the case for other architectures, e.g. an implementation of the ARC architecture uses special instructions that access a separate I/O space containing the virtual page mappings. This particular behaviour may work in CAPTIVE's favour, however, but it is reasonable to envisage a configuration that does not map so well.

### 7.2.3 Assumptions

The evaluation of these techniques imposed some assumptions on the guest platform being virtualised, as experiments have been made throughout for a particular architecture. Although the techniques as described can be applied to other architectures, an ARMv7-A platform was chosen to model as architectural information is readily available, and previous work in this area enables straightforward modelling of the platform. Additionally, the benefit of choosing this particular platform is that it is a 32-bit platform, and hence the guest virtual address space fits well within the larger 64-bit address space of the host machine. Additionally, the chosen platform was a single-core platform, as implementing multi-core virtualisation directly (without first attempting single-core virtualisation) would be unwise.

The hardware assisted virtualisation technique described in Chapter 6 relies on the availability of KVM in the host machine operating system, but most modern Linux distributions come with KVM available as default. KVM is a convenient abstraction for accessing hardware-assisted virtualisation extensions, and makes an implementation much easier, but it must cater for the lowest common denominator. As it turns out, most hardware virtualisation extensions can be presented with a uniform interface and so in practice this is not a problem. KVM does, however, contain a mechanism that indicates what features are available on the host platform, and in some cases provides architecture-specific virtualisation controls.

Technically, it would be possible to interface directly with Intel VT or AMD-V extensions, as this is the approach VirtualBox takes. However, for running on a Linux operating system it would require developing a custom kernel module, which would be a significant time investment and would restrict virtualisation to only those supported host systems. Furthermore, it would require non-standard additions to a host platform, which may deter companies from deploying the hypervisor.

## 7.3 Future Work

There is quite a bit of scope for future work in the area of cross-architecture virtualisation, with potential users of systems ranging from software engineers,

hardware developers, security/malware researchers and data centre administrators. The following sections shall describe expected future work in this area, and the expected scale of this research.

### 7.3.1 Efficient Interrupt Virtualisation

#### 7.3.1.1 Exact Check Placement

Chapter 5 described techniques for improving interrupt virtualisation, but there are even more opportunities to investigate in this area. The technique was to use an algorithm to place interrupt checks in the *correct* places, but an investigation to the effect that the exact placement of interrupt checks has on the quality of generated code could be made. For example, does the placement of an interrupt check enable the optimiser to produce better code when inserted into a loop condition block, as opposed to the loop body?

**—This work could lead to a research paper on the topic.**

#### 7.3.1.2 Dynamic Placement Schemes

The interrupt check placement schemes were inherently static, in that they placed interrupt checks at region translation time. Triggering region recompilation during high I/O bound workloads, and instructing the JIT to be more aggressive in placing interrupt checks, may lead to higher throughput.

**—This work could lead to a research paper on the topic.**

#### 7.3.1.3 Synchronous Exceptions

Clearly, asynchronous interrupts lead to unpredictable control-flow, but there are also sources of inefficiency when diverging control-flow is known at JIT compilation time. Such a source of this may be exceptions, where an instruction has the ability to divert control-flow to an exception handler in certain cases. The possibility of divergence is known statically, because the instruction description contains the control-flow that makes this happen. For example, a divide instruction will divert to an exception handler if the denominator operand is zero and a memory access instruction will divert to an exception handler if a page fault occurs.

Such conditions should be rare (they are *exceptional*) and as such, optimising for the common non-exceptional case is preferred. This work would investigate opportunities for optimising control-flow through instructions that exhibit this behaviour.

**—This work could lead to a research paper on the topic.**

## 7.3.2 Hardware Accelerated Cross-architecture Virtualisation

### 7.3.2.1 Integrating Region-based DBT/Domain-specific JIT

As has been shown in Chapter 4, the region-based approach to DBT is highly efficient and so integrating this with CAPTIVE would clearly lead to even further performance improvements. However, there is also scope for extending the performance and code-generation quality of the current block-based JIT, and by considering the domain-specific nature of JIT compilation for DBT, this could lead to further advances in instruction emulation performance.

**—This work could lead to one or two research papers on the topic.**

### 7.3.2.2 64-bit Support

As noted in the critical analysis, a simplifying assumption was that the guest platform is 32-bit. Now that 64-bit embedded systems are being developed and deployed, it is becoming increasingly important for engineers to virtualise these platforms. Therefore, a major route to extend this work would be to implement support for 64-bit guest platforms, exploring efficient ways to support the larger address space.

**—This work could lead to a research paper on the topic.**

### 7.3.2.3 Multi-core Support/Heterogeneous Modelling

Recently, there has been an explosion in the deployment of multi-core embedded systems, with platforms sporting multiple processing cores, heterogeneity and accelerators (such as DSPs). Although support was built into CAPTIVE from the start for multi-core virtualisation, the engineering effort to implement a multi-core platform (and the necessary interactions of the CAPTIVE infrastructure, e.g. the code cache) would lead to further research in this area—especially if one wishes to model heterogeneous platforms.

**—This work could lead to a journal article on the topic.**

#### 7.3.2.4 Further Hardware Assistance

CAPTIVE has taken advantage of existing hardware assistance for performing cross-architecture virtualisation, but as noted, these extensions were not originally designed to support this. A longer-term research project would be to investigate how these extensions could be modified to support cross-architecture virtualisation, or what new extensions could be introduced to assist hypervisors operating like this.

For example, instruction emulation is typically performed with DBT, and so can hardware extensions be introduced that assist DBT? What do these extensions look like? What other assistance can be provided by hardware to accelerate the main cross-architecture virtualisation challenges?

**—This work could lead to a PhD thesis.**

#### 7.3.2.5 Hardware-assisted Device Virtualisation

Whilst the emulated device implementations necessary for cross-architecture hardware virtualisation may be efficient, it would be interesting to map devices with similar behaviours (e.g. timer devices) from the guest system to the host system, using a hybrid approach where guest platform devices are backed by real hardware, as opposed to being purely software-based. This would keep much of a device's implementation internal to the VM, eliminating the need for costly hypervisor communication.

**—This work could lead to a research paper on the topic.**

#### 7.3.2.6 Nested Virtualisation

Many ARM-based platforms have hardware support for virtualisation themselves, so modelling this virtualisation would require exploring the nested virtualisation capabilities of the host, and attempting to exploit this support for modelling the virtualisation capabilities of a guest.

**—This work could lead to a research paper on the topic.**

#### 7.3.2.7 Cache Simulation

So far the described techniques have improved performance of hardware virtualisation, but with a DBT-based system, and control of the operation of a guest platform, it is possible to instrument certain guest architectural components. An increasing concern for application developers is cache utilisation, and existing cache simulators are quite slow. Building support into CAPTIVE for efficient cache simulation would enable useful measurements to be taken, that may assist developers in debugging performance regressions, or maximising utilisation.

**—This work could lead to one or two research papers on the topic.**

#### 7.3.2.8 Data Centre Virtualisation

The recurring example presented throughout this thesis was to virtualise an embedded ARMv7-A system on a more powerful Intel x86 host machine. However, an open question is how would this system translate to virtualising a larger, possibly distributed system? How would one virtualise a system on the scale of a data centre? If the virtualisation itself was distributed over a cluster, what infrastructure would need to be built in to the hypervisors to support inter-nodal communication? Can parts of the simulation be accelerated in some way? What level of abstraction is used to model a data centre?

**—This work could lead to a PhD thesis.**

#### 7.3.2.9 Compiler Generation from a High-level Architecture Description

The *architecture description language* (ADL) used by GENSIM to produce modules for ARCSIM and CAPTIVE has the potential for more than just simulator generation. Already, GENSIM can produce a disassembler for the described architecture, but it would be interesting to investigate if the description be used to produce the back-end for a compiler. This would have further benefits for embedded application developers, as not only could they be provided with a virtual environment to test their application, but also a compiler for an architecture that may not even exist yet.

**—This work could lead to a journal article on the topic.**

#### 7.3.2.10 Automated Cost Modelling

Another benefit of the ADL is that detailed analysis of instruction behaviours can be performed, and potentially used to generate an *execution cost model*. If sufficient intrinsics are used, or particular code patterns can be detected, then it may be possible to generate high-level profiling information for an instruction, and use this to seed a simulator to perform accurate measurements of latency, cycle counts, pipeline behaviour, etc.

This would require first analysing the semantic behaviour of an instruction, extracting the features that contribute to architectural operations, and then mapping these features to an architecture-specific model that describes how they translate to the various profiling metrics.

**—This work could lead to one or two research papers on the topic.**

## 7.4 Summary and Final Remarks

This thesis has presented a range of techniques for accelerating cross-architecture hardware virtualisation. It has shown that it is possible to improve virtualisation performance significantly, and exploit existing hardware support to build an efficient cross-architecture hypervisor.

# Bibliography


[1] Transitive Technology: The Rosetta Stone for binary translation. URL http://www.cs.manchester.ac.uk/our-research/research-impact/transitive-technology/. Retrieved 23-August-2016.

[2] Apple Rosetta (via Web Archive), 2011. URL https://web.archive.org/web/20110107211041/http://www.apple.com/rosetta. Retrieved 23-August-2016.

[3] Oscar Almer, Igor Böhm, Tobias Edler Koch, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. A parallel dynamic binary translator for efficient multi-core simulation. *International Journal of Parallel Programming*, 41(2):212–235, 2012. ISSN 1573-7640. doi: 10.1007/s10766-012-0222-9. URL http://dx.doi.org/10.1007/s10766-012-0222-9.

[4] AMD. AMD Virtualization, 2016. URL http://www.amd.com/en-us/solutions/servers/virtualization. Retrieved 17-May-2016.

[5] AMD Developer Central. AMD SimNow simulator. http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/, 2010.

[6] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, January 2009. ISSN 0163-5980. doi: 10.1145/1496909.1496921. URL http://doi.acm.org/10.1145/1496909.1496921.

[7] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1496909.1496921.

[8] Ariel. App stores growth accelerates in 2014, 2015. URL `http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/`. Retrieved 18-May-2016.

[9] ARM. RealView Platform Baseboard for Cortex-A8 user guide, 2011. URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0417d/index.html`. Retrieved 02-June-2016.

[10] ARM. About the PB-A8, 2011. URL `http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/BABCHBFC.html#CHDFGCFB`. Retrieved 02-June-2016.

[11] ARM. Versatile Application Baseboard for ARM926EJ-S user guide, 2011. URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0225d/index.html`. Retrieved 20-July-2016.

[12] ARM. Fast Models, 2016. URL `http://www.arm.com/products/tools/models/fast-models/`. Retrieved 26-Auguest-2016.

[13] ARM Ltd. ARM security technology building a secure system using TrustZone technology, 2005–2009. URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch05s03s03.html`.

[14] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002. ISSN 0018-9162. doi: 10.1109/2.982917. URL `http://dx.doi.org/10.1109/2.982917`.

[15] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL `http://doi.acm.org/10.1145/857076.857077`.

[16] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and

tools. *Int. J. Parallel Program.*, 33(5):453–484, October 2005. ISSN 0885-7458. doi: 10.1007/s10766-005-7301-0. URL http://dx.doi.org/10.1007/s10766-005-7301-0.

[17] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL http://doi.acm.org/10.1145/349299.349303.

[18] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4): 1319–1360, 1994.

[19] 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. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, 2003.

[20] James R Bell. Threaded code. *Communications of the ACM*, 16(6), 1973.

[21] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

[22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL http://doi.acm.org/10.1145/2024716.2024718.

[23] Igor Böhm, Björn Franke, and Nigel P. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In Fadi J. Kurdahi and Jarmo Takala, editors, *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-7937-5. doi:

10.1109/ICSAMOS.2010.5642102. URL
http://dx.doi.org/10.1109/ICSAMOS.2010.5642102.

[24] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993508. URL
http://doi.acm.org/10.1145/1993498.1993508.

[25] Florian Brandner. Precise simulation of interrupts using a rollback mechanism. In *Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '09, pages 71–80, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-696-0. URL
http://dl.acm.org/citation.cfm?id=1543820.1543833.

[26] Derek Bruening and Evelyn Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *In Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pages 13–20, 2000.

[27] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL
http://dl.acm.org/citation.cfm?id=776261.776290.

[28] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and R Taheri. Methodology for performance analysis of VMware vSphere under tier-1 applications. *VMware Technical Journal*, 2(1), 2013.

[29] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014. ISSN 1544-3566. doi: 10.1145/2629677.

[30] Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A high-level virtual platform for early MPSoC software development. In *Proceedings of the 7th IEEE/ACM*

*International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-628-1. doi: 10.1145/1629435.1629438. URL http://doi.acm.org/10.1145/1629435.1629438.

[31] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-ISA system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 117–128, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2764-0. doi: 10.1145/2576195.2576201. URL http://doi.acm.org/10.1145/2576195.2576201.

[32] Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is NP-hard for tournaments. *Comb. Probab. Comput.*, 16(1):1–4, January 2007. ISSN 0963-5483. doi: 10.1017/S0963548306007887. URL http://dx.doi.org/10.1017/S0963548306007887.

[33] Jianwei Chen, Murali Annavaram, and Michel Dubois. SlackSim: A platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37 (2):20–29, July 2009. ISSN 0163-5964. doi: 10.1145/1577129.1577134. URL http://doi.acm.org/10.1145/1577129.1577134.

[34] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0179-6. URL http://dl.acm.org/citation.cfm?id=520033.858247.

[35] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptive binary translation at low cost. *IEEE Computer*, 33(3):60–66, 2000.

[36] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM. ISBN 0-89791-659-X. doi: 10.1145/183018.183032. URL http://doi.acm.org/10.1145/183018.183032.

[37] Embedded Microprocessor Benchmark Consortium et al. EEMBC benchmark suite, 2009.

[38] Balaji Dhanasekaran and Kim Hazelwood. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, RESoLVE'11, pages 11–18, 2011.

[39] J. H. Ding, P. C. Chang, W. C. Hsu, and Y. C. Chung. PQEMU: A parallel system emulator based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 276–283, Dec 2011. doi: 10.1109/ICPADS.2011.102.

[40] 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 *Ottawa Linux Symposium*, 2012.

[41] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, Jun 2001. ISSN 0018-9340. doi: 10.1109/12.931892.

[42] David Ehringer. The Dalvik virtual machine architecture. 2010.

[43] Brendan Eich. TraceMonkey: JavaScript lightspeed. *Retrieved on August*, 24: 2010, 2008.

[44] G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998. ISSN 0178-4617. doi: 10.1007/PL00009191. URL `http://dx.doi.org/10.1007/PL00009191`.

[45] A. Forin, B. Neekzad, and N. L. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, WA, 2006.

[46] Adam Gerber and Clifton Craig. *Learn Android Studio: Build Android Apps Quickly and Effectively*. Apress, Berkely, CA, USA, 1st edition, 2015. ISBN 1430266015, 9781430266013.

[47] Parallels IP Holdings GmbH. Parallels Desktop for Mac, 2016. URL `http://www.parallels.com/uk/products/desktop/`. Retrieved 17-May-2016.

[48] Cosmin Gorgovan, Amanieu D'antras, and Mikel Luján. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):14, 2016.

[49] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. DBT path selection for holistic memory efficiency and performance. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 145–156, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-910-7. doi: 10.1145/1735997.1736018. URL http://doi.acm.org/10.1145/1735997.1736018.

[50] Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen, and Jenq-Kuen Lee. Translating the ARM Neon and VFP instructions in a binary translator. *Software: Practice and Experience*, 2016.

[51] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor N. Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel C. Paver. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 13–22. IEEE Computer Society, 2014. ISBN 978-1-4799-3604-5. doi: 10.1109/ISPASS.2014.6844457. URL http://dx.doi.org/10.1109/ISPASS.2014.6844457.

[52] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. SimFlex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, March 2004. ISSN 0163-5999. doi: 10.1145/1054907.1054914. URL http://doi.acm.org/10.1145/1054907.1054914.

[53] Asias He. Native Linux KVM tool, 2011. URL http://www.linux-kvm.org/images/c/c5/2011-forum-native-linux-kvm-tool.pdf. Retrieved 12-August-2016.

[54] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL http://doi.acm.org/10.1145/1186736.1186737.

[55] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society. ISBN

0-7695-2440-0. doi: 10.1109/MICRO.2005.22. URL http://dx.doi.org/10.1109/MICRO.2005.22.

[56] Jason D. Hiser, Naveen Kumar, Min Zhao, Shukang Zhou, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Techniques and tools for dynamic optimization. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 279–279, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898699.1898797.

[57] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, and Bruce R. Childers. Evaluating fragment construction policies for SDT systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 122–132, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134778. URL http://doi.acm.org/10.1145/1134760.1134778.

[58] 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.

[59] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259030. URL http://doi.acm.org/10.1145/2259016.2259030.

[60] Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. Optimizing control transfer and memory virtualization in full system emulators. *ACM Trans. Archit. Code Optim.*, 12(4):47:1–47:24, December 2015. ISSN 1544-3566. doi: 10.1145/2837027. URL http://doi.acm.org/10.1145/2837027.

[61] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong,

Pen-Chung Yew, and Wei-Chung Hsu. LnQ: Building high performance dynamic binary translators with existing compiler backends. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 226–234, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.57. URL http://dx.doi.org/10.1109/ICPP.2011.57.

[62] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 23–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451519. URL http://doi.acm.org/10.1145/2451512.2451519.

[63] Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Jörn Rennecke, and Grigori Fursin. Transforming GCC into a research-friendly environment: Plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*, Pisa, Italy, Jan 2010. URL https://hal.inria.fr/inria-00451106.

[64] Imperas. DEV - virtual platform development and simulation, 2016. URL http://www.imperas.com/dev-virtual-platform-development-and-simulation. Retrieved 18-May-2016.

[65] Intel. Intel Virtualization Technology (Intel VT), 2016. URL http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html. Retrieved 26-April-2016.

[66] A. Jaleel, R.S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly single/multi-core cache simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, MoBS, 2008.

[67] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. SPIRE: Improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 1–12, New York, NY, USA,

2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451516. URL http://doi.acm.org/10.1145/2451512.2451516.

[68] Jikes RVM. Threading and yieldpoints, 2007. URL http://jikesrvm.org/Threading+and+Yieldpoints.

[69] Daniel Jones and Nigel Topham. High speed CPU simulation using LTU dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_6. URL http://dx.doi.org/10.1007/978-3-540-92990-1_6.

[70] Rahul Joshi, Michael D. Bond, and Craig Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 239–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977660.

[71] Richard M Karp. *Reducibility among Combinatorial Problems*. Springer, 1972.

[72] Marco Kaufmann and Rainer G. Spallek. Superblock compilation and other optimization techniques for a Java-based DBT machine emulator. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 33–40, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451521. URL http://doi.acm.org/10.1145/2451512.2451521.

[73] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-93899-6. doi: 10.1007/978-3-540-93900-9_19. URL http://dx.doi.org/10.1007/978-3-540-93900-9_19.

[74] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[75] Toshihiko Koju, Xin Tong, Ali Ijaz Sheikh, Moriyoshi Ohara, and Toshio Nakatani. Optimizing indirect branches in a system-level dynamic binary translator. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 5:1–5:12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1448-0. doi: 10.1145/2367589.2367599. URL http://doi.acm.org/10.1145/2367589.2367599.

[76] Naveen Kumar, Bruce R. Childers, Daniel Williams, Jack W. Davidson, and Mary Lou Soffa. Compile-time planning for overhead reduction in software dynamic translators. *Int. J. Parallel Program.*, 33(2):103–114, June 2005. ISSN 0885-7458. doi: 10.1007/s10766-005-3573-7. URL http://dx.doi.org/10.1007/s10766-005-3573-7.

[77] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[78] SeungIl Lee, Byung-Sun Yang, and Soo-Mook Moon. Efficient Java exception handling in just-in-time compilation. *Softw. Pract. Exper.*, 34(15):1463–1480, December 2004. ISSN 0038-0644. doi: 10.1002/spe.v34:15. URL http://dx.doi.org/10.1002/spe.v34:15.

[79] Markus Levy. EEMBC and the purposes of embedded processor benchmarking. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software,*, page 1. IEEE, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/ISPASS.2005.1430553.

[80] Mieszko Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, Christopher W. Fletcher, Omer Khan, and Srinivas Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 175–185, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-367-4. doi: 10.1109/ISPASS.2011.5762734. URL http://dx.doi.org/10.1109/ISPASS.2011.5762734.

[81] Chien-Te Liu, Kuan-Chung Chen, and Chung-Ho Chen. CASL hypervisor and its virtualization platform. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1224–1227, May 2013. doi: 10.1109/ISCAS.2013.6572073.

[82] I-Chun Liu, I-Wei Wu, and Jean Jyh-Jiun Shann. Instruction emulation and OS supports of a hybrid binary translator for x86 instruction set architecture. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1070–1077. IEEE, 2015.

[83] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[84] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu. ARMISS: An instruction set simulator for the ARM architecture. In *International Conference on Embedded Software and Systems*, ICESS '08, pages 548–555, 2008. doi: 10.1109/ICESS.2008.73.

[85] Peter S. Magnusson and Bengt Werner. Some efficient techniques for simulating memory. Technical Report R94, Swedish Institute of Computer Science technical report, 1994.

[86] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: a full system simulation platform. *j-COMPUTER*, 35(2): 50–58, February 2002. ISSN 0018-9162 (print), 1558-0814 (electronic). URL `http://dlib.computer.org/co/books/co2002/pdf/r2050.pdf; http://www.computer.org/computer/co2002/r2050abs.htm`.

[87] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2): 50–58, February 2002. ISSN 0018-9162. doi: 10.1109/2.982916. URL `http://dx.doi.org/10.1109/2.982916`.

[88] Mark Lord. hdparm(8): get/set SATA/IDE device parameters, 2012. URL `http://linux.die.net/man/8/hdparm`.

[89] Dirk Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[90] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3947-6. doi: 10.1145/2892242.2892258. URL http://doi.acm.org/10.1145/2892242.2892258.

[91] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[92] 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. doi: 10.1145/1542452.1542472. URL http://doi.acm.org/10.1145/1542452.1542472.

[93] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October 2000. ISSN 1092-3063. doi: 10.1109/4434.895100. URL http://dx.doi.org/10.1109/4434.895100.

[94] Oracle. VirtualBox, 2016. URL https://www.virtualbox.org/. Retreived 23-August-2016.

[95] David Ott. Virtualization and performance: Understanding VM exits, 2009. URL https://software.intel.com/en-us/blogs/2009/06/25/virtualization-and-performance-understanding-vm-exits. Retrieved 07-June-2016.

[96] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 CPUs. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1050–1055, June 2011.

[97] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Proceedings of the Design Automation Conference*, DAC '11, 2011.

[98] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the ARM architecture. *J. Syst. Archit.*, 59(3):144–154, March 2013. ISSN 1383-7621. doi: 10.1016/j.sysarc.2013.02.003. URL http://dx.doi.org/10.1016/j.sysarc.2013.02.003.

[99] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Evaluation of dynamic binary translation techniques for full system virtualisation on ARMv7-A. *Journal of Systems Architecture*, 65: 30–45, 2016.

[100] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Evaluation of dynamic binary translation techniques for full system virtualisation on ARMv7-A. *Journal of Systems Architecture*, 65: 30–45, 2016. doi: http://dx.doi.org/10.1016/j.sysarc.2016.03.001.

[101] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361073. URL http://doi.acm.org/10.1145/361011.361073.

[102] Wei Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 556–561, 2003. doi: 10.1109/DATE.2003.1253667.

[103] Rusty Russell. VirtIO: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400108. URL http://doi.acm.org/10.1145/1400097.1400108.

[104] Frederick Ryckbosch, Stijn Polfliet, and Lieven Eeckhout. Fast, accurate, and validated full-system software simulation of x86 hardware. *IEEE Micro*, 30(6): 46–56, November 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.95. URL http://dx.doi.org/10.1109/MM.2010.95.

[105] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485963. URL http://doi.acm.org/10.1145/2485922.2485963.

[106] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*, pages 183–192, Oct 2015. doi: 10.1109/IISWC.2015.29.

[107] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 283–294, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0. doi: 10.1145/291069.291063. URL http://doi.acm.org/10.1145/291069.291063.

[108] Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pages 3–12. ACM, 2014.

[109] Costas Stylianou. Speeding up the Android Emulator on Intel architecture, 2013. URL https://software.intel.com/en-us/android/articles/speeding-up-the-android-emulator-on-intel-architecture. Retrieved 24-August-2016.

[110] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 312–323, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781166. URL http://doi.acm.org/10.1145/781131.781166.

[111] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*,

28(1):134–174, January 2006. ISSN 0164-0925. doi: 10.1145/1111596.1111600. URL http://doi.acm.org/10.1145/1111596.1111600.

[112] Synopsys. Virtual Prototyping, 2016. URL https://www.synopsys.com/Prototyping/VirtualPrototyping/Pages/default.aspx. Retrieved 17-May-2016.

[113] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[114] N Topham. EnCore: A low-power extensible embedded processor. In *Presentation at HiPEAC Industrial Workshop*, 2009.

[115] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 41–51, New York, NY, USA, 2000. ACM. ISBN 1-58113-241-7. doi: 10.1145/351397.351414. URL http://doi.acm.org/10.1145/351397.351414.

[116] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. *SIGARCH Comput. Archit. News*, 29(1):55–65, March 2001. ISSN 0163-5964. doi: 10.1145/373574.373590. URL http://doi.acm.org/10.1145/373574.373590.

[117] VMware. A performance comparison of hypervisors. Technical report, VMware, 2007. URL https://www.vmware.com/pdf/hypervisor_performance.pdf.

[118] VMware. Performance evaluation of Intel EPT hardware assist. Technical report, VMware, 2009. URL https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[119] VMware. Server consolidation, 2016. URL http://www.vmware.com/uk/consolidation/overview. Retrieved 18-May-2016.

[120] VMware. VMware ESXi, 2016. URL http://www.vmware.com/products/esxi-and-esx.html. Retrieved 24-August-2016.

[121] Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the Annual Design Automation Conference*, DAC '13, pages 21:1–21:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488760. URL http://doi.acm.org/10.1145/2463209.2488760.

[122] Harry Wagstaff, Tom Spink, and Björn Franke. Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[123] Huang Wang, Chao Wang, and Huaping Chen. XEMU: A cross-ISA full-system emulator on multiple processor architectures. *International Journal of High Performance Systems Architecture*, 5(4):228–239, 2015.

[124] Huang Wang, Xianglan Chen, and Huaping Chen. A cross-ISA kernelized high-performance parallel emulator. *International Journal of Parallel Programming*, 44(6):1118–1141, 2016. ISSN 1573-7640. doi: 10.1007/s10766-015-0379-0. URL http://dx.doi.org/10.1007/s10766-015-0379-0.

[125] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 53–64. ACM, 2015.

[126] John Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504295. URL http://doi.acm.org/10.1145/504282.504295.

[127] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM. ISBN 0-89791-793-6. doi:

10.1145/233013.233025. URL
http://doi.acm.org/10.1145/233013.233025.

[128] Liao Yin, Jiang Haitao, Sun Guangzhong, Jin Guojie, and Chen Guoliang. Improve indirect branch prediction with private cache in dynamic binary translation. In *International Conference on High Performance Computing and Communication and International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 280–286, 2012. doi: 10.1109/HPCC.2012.45.

[129] Seehwan Yoo, Kuen-Hwan Kwak, Jae-Hyun Jo, and Chuck Yoo. Toward under-millisecond I/O latency in Xen-ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 14:1–14:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1179-3. doi: 10.1145/2103799.2103816. URL http://doi.acm.org/10.1145/2103799.2103816.

[130] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, April 2007. doi: 10.1109/ISPASS.2007.363733.