THE UNIVERSITY *of* EDINBURGH

Efficient Cross-architecture Simulation of Multicore Systems

Martin Kristien

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2024

Abstract

Computer systems are continually becoming more complex and powerful in all areas of computing, from high-end servers to embedded devices. Machine virtualisation has become an instrumental technology to manage these vast hardware resources by separating applications running within virtual machines (guests) from the real hardware (hosts). This form of virtualisation is well supported by modern hardware as long as the guest and the host machines' Instruction Set Architectures (ISA) are matching. A mismatching (cross-ISA) virtualisation is more challenging, while remaining an important technology for hardware prototyping and software development.

In this context, Instruction Set Simulators (ISSs) are developed to provide functional cross-architecture simulation. A wide range of techniques can be utilised to achieve high simulation speeds for single-core guest applications. However, multicore support is limited. The state of the art tools are often making trade-offs between accuracy and speed. The lack of multicore support is exacerbated if the guest application requires a full-system simulation and/or exhibits dynamically generated code.

This thesis presents three contributions to accuracy, memory efficiency, and simulation speed of multicore cross-ISA simulation. Firstly, it presents a scalable and provably correct scheme for emulating atomic instructions. Most commonly in cross-ISA simulation, Reduced Instruction Set Computer (RISC) type guest atomics, Load-Linked/Store-Conditional (LL/SC), are emulated on Complex Instruction Set Computer (CISC) type host hardware providing a complex Compare-And-Swap (CAS) atomic instruction. Although the semantics of the RISC and CISC atomics are different, ISSs often emulate LL/SC using CAS instructions for improved performance. However, this results in a divergent execution inside a simulator relative to the real hardware. The scheme presented in this thesis faithfully emulates the LL/SC semantics while maintaining scalability to multicore systems.

Efficient use of simulator memory is especially important for interpreter-based ISSs, which enable quick prototyping without extensive engineering efforts and easy integration with instrumentation, profiling, and debugging tools. However, the computational overheads of the Fetch and Decode stages in instruction interpretation significantly increase the overall simulation time. This thesis proposes a number of memory efficient caching strategies with focus on memory sharing among multiple simulated cores. The novel schemes exhibit up to 1.57$\times$ speedup relative to state-of-the-art baseline scheme, while requiring only 27% of cache memory.

Dynamic Binary Translation typically translates and caches multiple guest instructions as a unit, resulting in faster simulation speed. However, code cache maintenance has to account for guest applications modifying its own instructions, necessitating invalidation of cached code. This maintenance mechanism in most ISSs is falsely triggered by applications dynamically generating code in proximity of previously cached code, resulting in needless code invalidation and poor performance. This thesis proposes an improved code tracking scheme that allows optimised guest code execution even when data and code are interleaved by the guest, achieving up to 1.42× speedup relative to state-of-the-art page-granular code protection mechanism.

Lay summary

Processors are the cornerstone of modern technology. Over the years, they became so cheap that we started putting processors into all products running on electricity. Today, they are everywhere, from our laptops and phones to cars, fridges, and coffee machines. In fact, most products have more than one processor collaborating to achieve a common task. But how are the processors designed and tested? Surely, processor designers don't want to have a full fridge in their offices. Instead, we can make a computer program that pretends to be the fridge processor, so that the designers can work conveniently from their desktop computers, using a fridge processor simulation. The main objective of this thesis is to make this simulation fast and convenient.

The first challenge in processor simulation is translating between mismatching processor languages. As multiple processors in a simulated system need to talk to each other, they employ a prescribed way of communicating and organising each other's work. However, the language of the simulated fridge processor is often different from the language used by the designers' desktop computers. The first idea presented in the thesis is to translate between these languages on the fly without significantly slowing the simulation down.

Processor simulation is laborious. To make simulation fast, the simulation program tries to remember previous simulation steps so that some of the already performed work can be reused in the future. However, we cannot remember all the past simulation steps, as doing so would quickly exhaust the storage space available on our desktop computers. This problem becomes even worse when we try to simulate multiple collaborating processors at once. The next idea presented in the thesis is to reorganise the storage of these simulation steps to allow simulators to remember more without requiring more storage.

Applications running inside our fridges are becoming more robust, even having access to the Internet and sending emails. As a result, these new applications exhibit more and more complicated behaviour at the processor level, which is much harder to make-pretend in a processor simulation. In particular, the novel application features often negatively interfere with various improvements and simulation shortcuts that were historically developed for simpler applications popular at the time. The last key idea of this thesis is to make the simulation fast, even when running these complex applications.

Acknowledgements

First and foremost, I would like to thank my industry supervisor, Igor Böhm, for his support regarding all aspects of my research. This includes technical implementation of my ideas in an industry tool used as a research vehicle, analysing evaluation data to discover new research direction, and providing motivation in times of limited research success. I would also like to thank my academic supervisors, Björn Franke and Tom Spink, for helping me navigate the academic world, from writing papers and choosing conferences, to polishing my research ideas, and conducting research in general.

Furthermore, I wish to express gratitude to friends and family around me who were there for me during my PhD. I and Matus Falis shared our PhD struggles and supported each other through many conversations over a high quality tea. Thanks also go to Zuz Olsinova, who encouraged me to start the PhD program, and to Natalia Grolmusova, who supported me to finish it.

Finally, I would like to thank everybody else, from friends and colleagues to admin and cleaning staff, who all contributed to my positive PhD experience.

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.

Some of the material in this thesis has been published in the following paper:

- **Martin Kristien**, Tom Spink, Brian Campbell, Susmit Sarkar, Ian Stark, Björn Franke, Igor Böhm, and Nigel Topham. 2020. Fast and correct load-link/store-conditional instruction handling in DBT systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3544-3554 [60]
  - This publication forms the basis of Chapter 4

- One or two publications will be submitted in the future, corresponding to the contributions presented in Chapter 5 and Chapter 6

(*Martin Kristien)*

Table of Contents

Acronyms

CAS Compare-and-Swap. 24, 26, 27, 30, 31, 41–45, 48, 62, 108

CISC Complex Instruction Set Computer. 8, 24, 30, 41, 107

DBI Dynamic Binary Instrumentation. 17

DBO Dynamic Binary Optimisation. 13

DBT Dynamic Binary Translator. 13, 15, 17, 19, 29, 34, 36, 65, 89, 91

DGC Dynamically Generated Code. 33, 87, 88, 92, 97–102

IR Intermediate Representation. 36

ISA Instruction Set Architecture. 8, 13, 15–18, 24–26, 36–39, 41, 71, 93, 100, 101

ISS Instruction Set Simulator. 6, 7, 14–18, 22, 26, 28, 29, 31, 33, 36, 37, 40, 41, 45, 46, 49, 62, 65–67, 69, 76, 86, 87, 89, 97–99, 111, 112

JIT Just-in-Time. 16–19, 21, 29, 34, 65, 87, 98–100, 102, 103, 106, 110, 111

LL Load-Linked. 24, 30, 41, 42, 46, 48, 50–55, 57–59, 62, 109

LL/SC Load-Linked/Store-Conditional. 8, 24, 26, 27, 30, 31, 41–52, 54–60, 62, 63, 107–109, 111

MMU Memory Management Unit. 20, 21, 68, 88, 89, 113

OS Operating System. 12, 36, 85

PC Program Counter. 13, 15, 29, 36, 65–71, 81

PTC Page Translation Cache. 35, 36, 49–51, 59, 89–91, 94–97

RISC Reduced Instruction Set Computer. 8, 24, 41, 107

SC Store-Conditional. 24, 26, 30, 41–43, 46, 51–55, 57–59, 62, 109

SMC Self-Modifying Code. 21, 22, 30, 87–90, 92, 94, 95, 98, 99, 102, 108, 112

TB Translation Block. 36

TCG Tiny Code Generator. 36, 37

TLB Translation Lookaside Buffer. 20, 29, 35

Chapter 1

Introduction

Computers proliferate every aspect of modern society. The success of computing can be attributed to steadily increasing computing power and decreasing computing costs due to miniaturisation of integrated circuits and economies of scale. Exemplified by Moore's law, every two years, the number of transistors and the resultant computing power on a single chip doubled. The resultant abundance of silicon allowed computers to specialise into several domains. The most powerful chips are used in *high performance* scientific computing and industrial-scale server farms. Commodity hardware can be readily employed in everyday *personal computers*. Energy efficient chips are crucial in extending battery life of *mobile devices*. Lastly, specialised low-powered *microcontrollers* are embedded in virtually every piece of electronics, often enabled with networking capabilities to create Internet of Things (IoT) systems.

In the very early days of programmable computing, virtualisation technology appeared in order to run multiple unmodified systems in isolation on the same hardware. In 1966, IBM CP-40 was the first operating system that provided a full virtual machine environment. This allowed multiple *guest* platforms to run side-by-side on a shared IBM System/360 hardware *host*. Nowadays, virtual machines are utilised in all domains of computing. For example, server consolidation [105] reduces hardware costs and energy consumption by increasing server utilisation and streamlining server management. In personal computing, virtualisation allows users to run multiple operating systems side-by-side [42, 82]. Android Studio provides a platform emulator [71] for development of mobile apps. Embedded developers can use Synopsys Virtual Prototyping tool [98] to prototype and debug applications without having to deploy to a real device. With the advent of computing in so many different forms, the use of virtual platforms became a fundamental tool across the field.

Another prominent technology in computing is multicore processors. While Moore's law stipulated miniaturisation of transistors, Dennard scaling predicted power dissipation per area to remain constant. However, after around 2005, this trend could no longer be maintained due to leakage currents. With the breakdown of Dennard scaling, the focus of computer architecture innovation shifted from speeding up a single processing core to providing multiple cores on the same chip.

The first commercially available multicore processor was POWER4, designed by IBM and launched in 2001. While this dual-core processor was geared towards high performance server applications, in 2005 Intel and AMD released their consumer focused dual-core processors. Since then, commodity hardware with tens of cores became commonplace. Furthermore, specialised graphics processing units and Artificial Intelligence (AI) accelerators can manifest thousands of relatively simple cores on a single chip. At the other end of the computing spectrum, mobile and embedded computing is also becoming multicore. For example, Qualcomm Snapdragon 8 features a chip with up to 8 heterogeneous cores with additional Graphical Processing Unit (GPU) and Digital Signal Processor (DSP) accelerators.

Naturally, virtual machines should also support multicore guest platforms. While creating virtual machines where the guest and the host are of the same hardware architecture only requires a thin virtualisation layer, cross-architecture virtualisation must rely on software simulation (i.e. modelling) of the guest hardware. The software solutions traditionally used for cross-architecture virtualisation create a particularly difficult challenge for scalability of multicore virtualisation. As a result, designers of these virtual platforms must often make trade-offs between accuracy and performance, potentially leading to incorrect behaviour. Nevertheless, cross-architecture simulation remains a critical technology for many current and future applications.

1.1 Motivation

Instruction Set Simulators (ISSs) play a prominent role in the development of embedded hardware and software [41]. They enable rapid prototyping of new instructions [78], developing and debugging applications before hardware exists [6], and architectural exploration [56]. In particular, ISSs decouple embedded software development from the availability of a physical device by creating virtual machines of architectures different from their host platforms, relying on cross-architecture simulation technology.

Cross-architecture simulation has been a subject of intense research. Through development of various optimisation techniques, cross-architecture simulation can achieve the performance necessary to support a real-time usage of the simulated guest platforms as virtual machines. In fact, guest programs can sometimes run faster in simulation than on a real physical target platform, mainly due to a more performant host hardware. However, cross-architecture simulation techniques have been primarily developed for single-core platforms. For example, the most popular ISS, QEMU [13], developed in 2005, originally only supported single-core simulation. Later, multicore guest simulation was enabled by duplicating guest core context data structures. However, execution of all virtual guest cores was performed in a single host thread using a round-robin scheduling. Due to its open-source nature, QEMU has become a research tool in academia, leading to the emergence of simulators with truly parallel execution on the host [34, 109] around 2010. However, the scalable multicore support development in the main QEMU repository only began around 2014 [57, 58].

Extending simulation tools to efficiently support the current and future multicore platforms is not trivial. Firstly, atomic instructions, responsible for consistent memory access in a multicore system, typically differ between the guest and the host platforms. A cross-architecture simulator must bridge this gap to create a virtual machine that can *correctly* execute unmodified guest code, i.e. the observable behaviour has to be identical regardless of whether the guest program is executed inside the virtual machine or on a real hardware. In single-core execution, the semantics of these instructions can be fully emulated in software with negligible performance overhead, as there is never more than one atomic instruction in flight at a time. However, multicore execution relies on faithful interactions of several atomic instructions issued by multiple guest cores. The main challenge is to accurately emulate these interactions without serialising execution (which would lead to rapid degradation of simulation performance).

Secondly, software based modelling of the guest platform does not scale to many-core applications, as naïve replication of single-core models produces *inefficiencies* resulting in poor memory utilisation, excessive synchronisation, and potential performance bottlenecks. The main challenge is to avoid duplication by sharing resources among multiple cores while maintaining parallel access to frequently modified data.

Finally, novel workloads with dynamically generated guest code exhibit execution patterns, for which the common cross-architecture simulation optimisation techniques were not designed, resulting in under-utilisation of those optimisations. In particular, caches of previously executed guest code are continually invalidated during generation

of new guest code due to falsely triggered code-modification protection. The main challenge is to allow code caching for applications exhibiting dynamic code generation without additional overheads to the code-modification protection mechanism.

This thesis aims to provide techniques to achieve correct, scalable, and efficient cross-architecture simulation.

1.2 Outline and Contributions

Chapter 2 introduces key terminology used throughout this thesis and provides background for the field of virtual machines and cross-architecture simulation. It also introduces the challenges of developing simulation technology for multicore computer systems. The presented techniques and approaches are accompanied by related work exemplifying the advancements in the field.

Chapter 3 introduces the tools used as research vehicles to demonstrate the contributions of this thesis. Furthermore, it presents artifacts used as workloads in performance evaluation.

Chapter 4 explores the challenges of faithfully emulating atomic memory instructions of the Reduced Instruction Set Computer (RISC) based guest Instruction Set Architecture (ISA), namely Load-Linked/Store-Conditional (LL/SC), on a Complex Instruction Set Computer (CISC) based host machine. In the context of concurrent multicore guest execution, several approaches are explored and evaluated in terms of the impact on execution of regular (i.e. non-atomic) instructions and their scalability in many-core applications with intense use of atomic instructions.

Chapter 5 focuses on memory efficiency in interpretive execution. Several techniques to reduce duplication in core-private decode caches are introduced. The novel techniques enable sharing decode cache entries by multiple concurrently simulated cores, resulting in significantly better performance than a core-private cache design, while requiring less memory overall.

Chapter 6 tackles the performance bottleneck in virtual machines running workloads that exhibit dynamically generated code. The chapter introduces several techniques to track executed guest code to better differentiate self-modifying code from new code generation by tracking code and data residing on the same memory page. The proposed techniques provide consistent view of the guest memory even in the context of multicore guest simulation.

Finally, Chapter 7 summarises and concludes the work of this thesis. It also sug-

gests directions for future work in the field of cross-architecture simulation.

1.3 Setting of the PhD

This thesis has been produced as a part of an industry-funded PhD program. All research has been performed in close collaboration with Synopsys as the industry partner. The company's product, nSIM, was used as the primary research vehicle throughout the project. nSIM was originally developed at The University of Edinburgh as ArcSim [101, 52, 17] and later commercialised by Synopsys. The research problems tacked in this thesis were inspired by real challenges faced by nSIM. The resultant contributions have been or are planned to be integrated into a newer version of the product.

Chapter 2

Background and Related Work

This chapter introduces the field of the thesis. It discusses various technologies and approaches in the field, pointing out missing solutions to important issues.

The field related to the subject of this thesis is broad. For example, RTL simulation directly models digital circuits at a low level of abstraction. The modelling is extremely detailed and time-consuming. On the other end of the spectrum, language runtime executes an abstract language model that is far removed from the actual hardware an application runs on. Throughout this spectrum, domain experts refer to various technologies in the field as virtualisation, emulation, and simulation, sometimes interchangeably.

2.1 Terminology

The subject of this thesis lies in the middle of the virtualisation spectrum presented in Figure 2.1. To avoid possible confusion, precise definitions of the terminology used throughout the thesis needs to be provided. A significant uncertainty surrounds the usage and meaning of the terms *emulation* and *simulation* as they are often used interchangeably. However, exact definitions of these terms in different contexts might

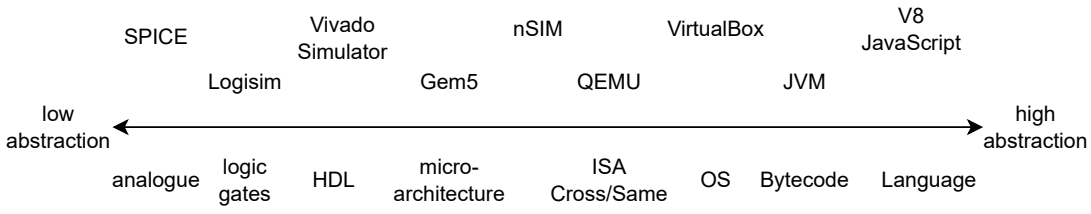


Figure 2.1: From Simulators to Virtual Machines

be contradictory. For example, in the context of developing mobile applications [46], *"emulation seeks to create an exact copy of a specific device"* while *"simulation creates an environment that imitates the behaviour"*. On the other hand, in the context of programming and software architecture [39], *"simulating a system or process consists of building a model"* while *"emulation means executing a system through another system"*. Another definition [94] stipulates that *"emulation is the process of mimicking the outwardly observable behavior"* and *"simulation involves modelling the underlying state"*.

Systems found on the low abstraction end of virtualisation spectrum are typically called simulators. Emulation is used more often on the high abstraction end of virtual machines, e.g. an Operating System (OS) system call can be emulated. As the guest platforms differ more and more from the host platform (on the low abstraction end of the spectrum) guest execution cannot be easily mapped onto the host hardware but the guest hardware must be modelled instead. There are some exceptions to this trend (e.g. EEPROM Emulator) and still a considerable overlap of terminology (e.g. Android Emulator and Apple Simulator are found at the same point on the virtualisation spectrum).

To avoid further confusion, explicit definitions of these terms is provided, to be used throughout the rest of the thesis:

- **virtualise** — create a virtual platform to provide an isolated execution environment
- **emulate** — imitate behaviour with focus on outwardly observable effects
- **simulate** — model a platform and its internal state in software
- **virtual machine** — an abstract/virtual platform that supports isolated execution of unmodified applications through the use of virtualisation and possibly also simulation and emulation
- **hypervisor** — software monitoring execution and resources of a virtual machine

Such definitions are typically used in the field of cross-architecture simulation. When the guest and the host architectures differ, hypervisors must simulate (i.e. model) the guest hardware. Executing guest instructions cannot be supported directly by the host hardware but must be emulated (i.e. imitated) to achieve the desired effect on the simulated guest platform. As a result, hypervisors create virtual machines that run unmodified guest applications in a timely manner.

2.2 Virtual Machines

A physical computer system can execute an application directly on the available hardware. The execution can be characterised by producing a certain output given an input, modifying the *architectural state* of the underlying machine in the process. The observable architectural state typically includes *memory* and the *CPU state* comprising Program Counter (PC), control and general purpose registers.

A virtual machine is an abstract model of a computer system. When a *guest* application is loaded into the virtual machine, the *host* system executes the guest application by performing changes to the model of the guest architectural state. These changes must be faithful to the specification of the guest architecture. As a result, the guest application should not be able to distinguish if it is being executed by a virtual or a physical machine.

A *hypervisor* executes directly on the host hardware and controls the execution of the guest application. The minimal functionality of a hypervisor is to provide an isolated runtime environment for an unmodified guest. Exemplified by a large scale server consolidation [105] or users running multiple operating systems side by side [82, 42], this type of hypervisor focuses on execution performance rather than precise modelling of the virtual guest machine.

Beyond providing functionally equivalent behaviour of the guest application, virtual machines can also facilitate advanced analysis and debugging of the guest applications. Dynamic Binary Translator (DBT) tools can analyse guest instruction streams, identify hot-spots, and insert custom instrumentation code at different execution points to enable memory analysis, cache simulation, or performance modelling. Notable examples include Pin [66] and Valgrind [81].

Furthermore, Dynamic Binary Optimisation (DBO) tools can leverage optimisation opportunities that only manifest at runtime to improve the guest performance in a hypervisor even beyond its native execution. Dynamo [10] and DynamoRio [20] rearrange the most frequent code snippets into a contiguous sequence at runtime, improving the instruction cache performance on the host. Focusing only on *hot* code snippets simplifies the code shape subject to optimisation, enabling removal of redundant jumps and loads.

These tools typically require the guest and the host ISA to be matching. While matching ISAs simplify guest instruction execution (as instruction emulation is not required), virtualising memory and I/O devices is still challenging. Various hardware

support technologies have been developed to alleviate the overheads of same-ISA virtualisation [104, 103]. For example, Intel-VT [104] introduces Extended Page Tables to speed up guest memory translation.

2.2.1 Cross-ISA Simulation

While same-ISA virtualisation can run guest binaries directly on the host hardware or through hardware-assisted virtualisation, cross-ISA virtualisation has to *emulate* (i.e. imitate the effects of) every guest instruction to accurately update the simulated guest machine model. Although this mode of execution is more difficult to implement efficiently, it enables additional use-cases on top of same-ISA virtualisation.

By the means of cross-ISA simulation, guest applications can run on any hardware, achieving platform portability. A prominent example is Apple's Rosetta [102] used to provide backward compatibility by emulating the PowerPC ISA on x86 hosts. Later, Rosetta 2 [8] similarly enabled adoption of ARM-based M1 and M2 chips.

Focus on platform portability is also seen in various language runtimes, where the language itself can be viewed as a particular guest ISA. The V8 engine by Google [43] can execute WebAssembly and JavaScript on a multitude of host platforms. Several implementations of Java Virtual Machines [7, 59] emulate a pre-compiled bytecode by providing an abstract model of a Java stack machine.

ISSs enable rapid software development and debugging from the comforts of a high performance development workstation without the need to deploy to an embedded target platform. For example, Synopsys® Virtualizer [99] provides tooling to facilitate software development targeting hardware platforms that are still in design and have not yet materialised. Another prominent example is Android® Emulator [71] based on QEMU [13], enabling development within an unmodified ARM Android environment.

Another use-case for an ISS is to support hardware development by enabling design space exploration of architectural and micro-architectural design trade-offs. Since a simulated model of the guest machine can be easily reconfigured to render different virtual machines, multiple design points can be evaluated in terms of performance and energy consumption. As opposed to functional simulation, performance and energy estimates require cycle-accurate modelling of the guest micro-architecture, making these simulators an order of magnitude slower. For example, gem5 [15] provides detailed pipeline models of out-of-order processors as well as cache coherence protocols. Although its accuracy has been validated against real hardware [44], slow simulation

speed (e.g. booting Linux takes 30 minutes) makes it impractical for application development. ArcSim [18] focuses on high-speed simulation of a simpler in-order processor model, allowing micro-architectural timing simulation to be interleaved into DBT-generated code.

2.3 Instruction Emulation

In cross-ISA simulation, the effect of each guest instruction is emulated to achieve the modification of the guest architectural state in accordance to the guest ISA description. Various emulation methods have been developed in increasing complexity and performance.

2.3.1 Interpretation

The simplest form of instruction emulation is interpretation, where each guest instruction is emulated in software one by one. For every instruction, the interpreter *fetches* the binary instruction encoding from the simulated guest memory, *decodes* the meaning and operands of the instruction, and *executes* the instruction by applying corresponding changes to the model of the guest architectural state. Although these operations are similar to typical hardware pipeline stages, they are not meant to model the guest hardware pipeline. Instead, they correspond to the design of the interpreter used to achieve functional correctness in ISSs.

Interpreters are relatively easy to implement, making them attractive for rapid development of new or existing architectures. Furthermore, since interpretation happens inline with execution, it is straightforward to integrate any profiling, instrumentation, or micro-architectural modelling tools. Another advantage of interpreters is low memory footprint and low execution startup cost, making them ideal for guest code regions that are not reused enough to amortise the overhead of more advanced emulation strategies.

Multiple techniques were developed to speed up interpretation. The most prominent optimisation is a decode cache. This is an interpreter data structure typically indexed by a guest instruction PC, returning a fully decoded instruction object. For repeated instructions, employing a decode cache can bypass complex guest memory access (required by the fetch operation) and avoid fine-grained bit manipulations of instruction encoding (required by the decode operation). This technique has been studied

extensively [25, 26, 53]. Note, decode cache is a hypervisor data structure typically implemented in software. Despite being often organised similarly to hardware caches (fixed size, set-associative), it is functionally more similar to a hashmap, as it is a software key-value data structure. Similarly to a hashmap, decode cache comprises *value* buckets accessed/indexed with a hash code computed from a *key*. A notable difference to a hashmap is that decode cache is statically allocated with fixed total size and the number of items in each bucket.

The Talisman ISS uses decoded instruction pages that contain slots for decoded instructions and mirror the guest instruction pages [12]. In Chen et al. [23], a custom hardware cache can store simple decode entries with an execute-dispatch function pointer. Stripf et al. [95] reduces the overhead of decode cache lookups by linking decode entries, relying on the fact that for non-branch instructions, the following instruction is always identical. Trade-offs for decode caches in ISSs are discussed extensively in Balderas-Contreras [11] and Jones and Topham [52]. An approach combining interpreted and compiled ISS for improved simulation performance is presented in Reshadi et al. [91].

Another set of optimisation techniques focuses on instruction execution. SimIcs [70] design comprises a threaded interpreter, which dispatches to *execute* operations directly without the need to loop around a top level switch statement in the main simulation loop. Furthermore, instruction behaviour can be specialised based on instruction-frequency statistics, and further optimised using partial evaluation.

2.3.2 Dynamic Binary Translation

Unfortunately, performance of interpreters is fundamentally limited. As guest instructions are emulated one by one, repeated lookups and control flow overhead in the hypervisor result in a vast number of host instructions required to emulate a single guest instruction. Furthermore, step-by-step execution prevents any inter-instruction optimisations.

A more efficient execution mode is achieved by taking several guest instructions and grouping the corresponding transformations of the guest architectural state into a single step. Translating multiple guest instructions into equivalent host instructions employs Just-in-Time (JIT) compilation, resulting in native execution of the original guest application translated for the host ISA. JIT compilation can exploit various optimisation opportunities depending on the size and shape of the units of translation.

The simplest grouping of guest instructions to form a translation unit is a basic block, defined as a linear code sequence with a single entry and a single exit. All instructions in a basic block are executed together, hence the compiled native code does not need to handle control flow. Instead, after execution of a basic block, the control switches back to the hypervisor, which finds the compiled code corresponding to the next basic block to be executed, and then jumps to it. This approach has been adopted by QEMU [13], which supports many ISAs by separating the JIT compiler into a guest specific front-end, a host specific back-end, and an intermediate representation linking the two. This allows great flexibility and portability when combining arbitrary guest/host ISAs. To reduce the overhead of dispatching JIT-ed code by the hypervisor, basic blocks can be linked by patching the return instruction to directly jump to the target basic block, i.e. block chaining. This is particularly useful for basic blocks terminated by direct branches or jumps. In practice, basic blocks in guest applications contain only a few instructions. Although this simplifies code generation, optimisation opportunities are limited.

An alternative approach translates guest code into traces, that are linear code sequences with a single entry and multiple exists. More guest instructions present in each translation unit enable more compiler optimisation based on rearranging and eliminating instructions on particular execution paths within a trace. As a result, the quality of the compiled code is much higher than in a basic block based DBT. Furthermore, switching between the guest code and the hypervisor is reduced. However, since traces comprise only linear code sequence, loop based optimisations are limited. DynamoSim [77] uses traces for ISA simulation. Trace-based approaches are also popular in Dynamic Binary Instrumentation (DBI) tools [66, 10, 20].

The best code quality is produced by region-based translation. Multiple basic blocks form a dynamic control-flow graph with multiple entry points and exists. Such a complex code shape allows for advanced loop optimisations, rearranging instructions across many basic blocks, thereby improving the native performance of the guest execution on the host. Region-based compilation has been used in Java virtual machines (e.g. Suganuma et al. [96]) and more recently adopted in ISSs [52, 17, 3]. SimIt-ARM [3] employs GCC as a JIT compiler to ease the implementation complexity. Böhm et al. [17] opt for the LLVM compiler framework to enable advanced optimisations. Both hide the significant compilation cost by offloading the region translation to parallel worker threads. Kaufmann and Spallek [54] introduce specific optimisations for ISS compiled into a Java bytecode.

Another configuration combining static and dynamic binary translation is exemplified by Apple's Rosetta 2 [8]. Although the exact operation of this technology is not openly available, reverse engineering attempts [79] revealed some internal design. Rosetta 2 aims to translate x86 binaries into ARM by leveraging Ahead-Of-Time compilation. The first time an x86 binary is started, its whole `.text` section is translated into a `.aot` file, which can be reused in the future. The Rosetta 2 runtime is responsible for mapping the `.aot` file into the main program's memory and resolving any missing references to `.data` section of the original x86 binary. Function references are resolved using stubs employing a lazy binding technique.

Rosetta 2 also employs JIT translation for shared library function calls and dynamically generated code. These fragments cannot be translated Ahead-Of-Time because the binary code is determined at runtime. The JIT engine records guest binary code at the granularity or basic block but it is unknown how the engine is organised and what shape of the translated code is produced.

Overall success of Rosetta 2 can also be attributed to direct mapping of the guest platform onto the host machine. The register file can be directly remapped onto the host registers using a proprietary ABI, since ARM ISA (on the host) can typically manifest more registers than what is required by the x86 guest. Guest memory does not need to be simulated, as the translated code is directly mapped into the host process. Furthermore, Rosetta 2 focuses on platform portability and does not need to simulate any guest hardware components or provide debugging/tracing features for the target platform. Currently, Rosetta 2 does not translate applications that comprise kernel extensions, virtualisation, and vector instructions (e.g. VX, AVX2, and AVX512).

2.3.3 Mixed Mode Execution

The overhead of JIT compilation can be significant, forcing ISSs to make trade-offs between compilation latency and produced code quality. The best performance can be achieved by first compiling the guest code with a fast low-latency base-level compiler, and later with an optimising compiler. Initially, code is compiled without costly optimisations to enable execution as soon as possible. Execution is interleaved with profiling, that determines which guest code regions are used most frequently. Code regions with enough execution heat are compiled a second time with a highly optimising compiler, and ideally will be executed in the future frequently enough to amortise the cost of the additional compilation and profiling overhead. This results in a mixed mode

execution, where guest code can be present and executed in multiple versions by the hypervisor. This is traditionally referred to as *tiered compilation*. Note, interpretation can be used instead of the base JIT compiler.

Such a design complicates hypervisor organisation, as it requires managing multiple versions of guest code compiled at various optimisation levels. In particular, guest execution must carefully transition to the optimised code without corrupting potentially concurrent execution of the initial unoptimised code. If an optimisation point is triggered during execution of a hot loop, transitioning to an optimised version of the code requires recreating the corresponding stack frame using on-stack replacement techniques [32]. A similar transition is required if an optimisation assumption (e.g. an optimised code assumed all numbers are integers) is violated. Furthermore, in case guest execution modifies its own code, all previously compiled code units containing the modified instructions must be invalidated.

A prominent example of tiered execution is the Java HotSpot VM [85]. Execution begins using interpretation, then a *client* JIT compiler generates unoptimised native code, and finally the so-called *server* compiler applies more powerful optimisations to frequently executed code. As a result, the VM exhibits quick start-up and low latency of interactive applications, while maintaining high performance for long-running applications. Another high-level language VM is Google's V8 JavaScript engine [43], that can recompile guest code at up to four optimisation levels. Due to the weakly typed nature of the JavaScript language, different optimisations levels might be achieved by specialisation, assuming variable types remain stable (e.g. small integers vs. double precision floating point types).

Dynamo [10] employs an interpreter for the initial execution. Hot code is transformed into traces to optimise code quality. DynamoRio [20] initially compiles guest code at basic block granularity before recompiling code into traces. HQEMU [49] extends QEMU DBT [13] with an LLVM-based second level compiler transforming guest code into traces, achieving up to $4\times$ speedup over a single-tiered QEMU JIT engine based on basic blocks. ArcSim [17] initially interprets guest code until region-based code is compiled by concurrent and parallel dynamic compilation worker threads.

2.4 Memory Simulation

Another element of guest architectural state that has to be modelled in virtual machine simulation is guest memory. As each emulated load and store instruction uses a guest-virtual address, a hypervisor must translate this address to a host-virtual address used to model guest memory before performing the requested memory access.

User-space guest memory can be hosted directly in the hypervisors address space as a flat array, denoting the start of the guest address space as a base offset. Then, any memory access performed by the guest application can easily be translated by just adding the guest base offset to obtain the corresponding host pointer. This approach is used by QEMU [13] in user-space mode.

However, simulating a full guest machine address space (i.e. full-system simulation) has to support arbitrary memory addresses. It is often infeasible to allocate a flat memory model (i.e. an array) for the whole guest address space, in particular if the address space of the host machine is *not* greater than the address space of the guest (e.g. a 32-bit guest on a 32-bit host). Instead, guest memory must be allocated on-demand, resulting in a fragmented model that requires re-mapping on a memory page granularity. Various techniques have been developed using caching or shadow page tables.

Note, guest to host memory translation is often paired with a guest Memory Management Unit (MMU) model, but they are not necessarily identical. In particular, the hypervisor translation cache can operate at a different granularity from guest pages. Furthermore, the hypervisor translation cache is required even if no guest MMU is modelled (e.g. user-space applications).

Simics [70] uses a Simulator Translation Cache to inline cache hits into interpretive emulation. Similarly, Topham and Jones [101] uses the notion of a Page Translation Cache dedicated individually to read, write, and fetch operations in a mixed interpretive and native execution. Successful lookups not only speed up translation of guest to host memory, but also indicate the corresponding access does not require any special handling and does not cause any side effect (e.g. misaligned access, Translation Lookaside Buffer (TLB) miss, guest cache miss, debug breakpoint). Another software solution can be seen in QEMU [13], where softMMU hypervisor cache is indexed by a guest physical address. The advantage of physical indexing is the persistence over guest MMU changes invalidating guest virtual address space.

An alternative approach uses a shadow page table that is enabled and used by the

host MMU every time the hypervisor switches execution to guest code. Unfortunately, this approach is efficient only in same-ISA simulation [4], as the guest-physical addresses are identical to host-virtual addresses. In cross-ISA simulation, an additional level of memory translation is required. Furthermore, as the guest code cannot execute on the host CPU directly, a shadow page switching is performed for every guest memory access.

Chang et al. [22] provided a hardware-assisted solution to cross-ISA memory simulation by embedding the guest shadow table into the host page table. However, the proposed solution requires the host address space to be larger than the guests' (e.g. a 64-bit host running a 32-bit guest). It also uses a host kernel module to manipulate the host MMU directly, which poses challenges for maintainability and platform portability. A similar approach has been adopted by Spink et al. [93] . This work further explores how to efficiently map a guest MMU onto the host, exploring hardware-accelerated extended page tables [104] (often used in same-ISA simulation [82]) for cross-ISA simulation. Wang et al. [108] propose a shadow page table embedded without requiring a kernel space modification. Instead, a shared memory mapping using the `mmap` system call is used.

2.4.1 Self-Modifying Code

Another challenge for memory simulation is data and code consistency in case the guest application modifies its own instructions, i.e. Self-Modifying Code (SMC). SMC is a broad term that generally refers to behaviour that treats code and data interchangeably, which is the default view in Von Neumann architectures that do not distinguish between code memory and data memory. Notable use-cases of applications modifying its own data include code obfuscation, encryption, and packing, as well as JIT compilers optimising previously generated code.

In virtual machines, SMC becomes problematic in the presence of a hypervisor code cache that can potentially store a stale version of the modified instruction. This warrants consideration of further use-cases that would not be conventionally regarded as SMC, such as the remapping of the guest MMU (the same virtual page might contain different code), dynamic linking and loading, and gcc trampoline executing from the stack. Therefore, SMC can be defined from the hypervisor's perspective as:

Definition 2.1. Self-Modifying Code is observed if two fetch operations from the same memory location (virtual PC) return different fetch values

Typically, a memory translation mechanism is repurposed to detect and handle SMC. For example, QEMU [13] and DynamoRio [20] use the host OS memory protection mechanism to artificially disable write access to host pages containing guest code. When the guest attempts to write into such a page, a page fault is triggered. The fault handling is performed by the hypervisor through inspecting the actual page access permission. If the permission allows writing into the page, SMC is detected, any corresponding code is invalidated, writes into the host page are re-enabled, and the guest execution resumes the faulting write operation.

An alternative approach using software caches for memory translation [70, 101] invalidates page entries for pages containing code and data. As a result, accessing these pages must be handled by the cache-miss slow path, where expensive detection and handling of SMC can be performed. In Chapter 6, this approach is modified to reduce the number of false-positive detections of SMC, resulting in improved performance of applications relying on dynamic code generation.

2.5 Multicore Simulation

Modern systems are becoming more and more multicore, and thus, ISSs have to be extended to also support such guest platforms. Although multicore simulation speeds up guest execution by leveraging parallel hardware available on the host, it comes with extra challenges and pitfalls. Firstly, multithreaded programming intrinsically requires more engineering effort to prevent parallel data races from corrupting program execution. Secondly, the hypervisor must faithfully support the guest's parallelism model even if it differs from the host's model.

2.5.1 Synchronisation

Parallel architectures allow multiple cores (i.e. instruction streams) to execute at the same time. Typically, a single shared memory is used to facilitate communication between the cores. However, cores modifying shared data must carefully synchronise to avoid data corruption that could lead to undefined behaviour. The goal is for each data update to be *atomic*, i.e. for all parallel cores to observe the update as a single indivisible operation without any intermediate state.

Atomicity of large data updates is achieved by software locks guarding *critical sections* (i.e. execution modifying or reading shared data). Each core desiring to enter

Core 1	Core 2
data = new_value;	while (flag != ready);
flag = ready;	read(data);

Table 2.1: Synchronisation using a `flag` indicating if `data` is ready to be used.

a critical section must first *acquire* an exclusive lock, potentially waiting if the lock is already acquired. Upon exiting the critical section, the lock is released to allow parallel cores to acquire it.

2.5.1.1 Memory Consistency

Any implementation of a lock scheme requires a consistent view of shared memory. Imagine a simplified synchronisation in Table 2.1. Core 1 produces a new value of `data` to be used by Core 2. After the new value is written to memory, core 1 sets the `flag` as *ready*. Core 2 waits for the `flag` to be observed as *ready* before reading the new value of `data`.

Intuitively, a read ought to return the value of the last write to the same memory location. In single-core execution, the sequential order specified by the program, i.e. the *program order*, precisely defines this aspect. However, the read and write of `data` in Table 2.1 are not related by program order, because they execute on two different cores. For example, core 1 can employ a write buffer, resulting in the write to `flag` being completed (i.e. externally visible) before the write to `data`, resulting in a reordering of the two write operations by core 1. Alternatively, core 2 might read the value of `data` before repeatedly reading `flag` until it is observed ready. This can result in the read `data` value to become stale due to reordering of the two read operations by core 2. Allowing reordering of memory operations enables significant performance optimisations but also increases complexity of reasoning about multicore communication through shared memory. The *memory consistency model* specifies how the memory system will appear to the programmer by defining which reorderings are allowed.

Multiple memory consistency models [5] have been proposed, with varying trade-offs between programmability and performance. For example, *sequential consistency* [62] requires that all memory operations appear to execute one at a time (i.e. no reordering is allowed). Under this model, a locking scheme using only regular loads and stores is possible [63]. This model, however, prohibits many performance optimisations performed by compilers and hardware micro-architectures, especially in the

presence of core-private data duplication by processor caches.

To achieve better performance on parallel architectures, many relaxed memory models were proposed, allowing memory operations to execute out-of-order. For example, *total store order* [50] allows future reads to be reordered before writes, enabling processors to use write buffers. Another example is PowerPC [72], a weakly ordered (i.e. relaxed) memory model allowing any memory operation reordering. This enables further optimisations by compilers, processor pipelines, and other micro-architectural components.

In order to still achieve a consistent view of shared memory, ISA designers provide various memory barriers and atomic instructions. These can be used to explicitly order memory operations to result in sequential consistency for desired instances.

2.5.1.2 Atomic Instructions

Atomic instructions are fundamental to multicore execution where shared memory is used for synchronisation purposes. CISC architectures typically provide various *read-modify-write* instructions, that perform multiple memory accesses (usually to the same memory address) with atomic behaviour. A prominent example of these instructions is the Compare-and-Swap (CAS) instruction. A CAS instruction atomically updates a memory location only if the current value at that location is equal to a particular expected value. The semantics of CAS instructions are shown in Figure 2.2. For example, Intel's x86 processors provide the `CMPXCHG` instruction to implement *compare-and-swap* semantics.

RISC architectures avoid complex atomic *read-modify-write* instructions by dividing these operations into distinct *read* and *write* instructions. In particular, Load-Linked (LL) and Store-Conditional (SC) instructions are used. The operation of these instructions is shown in Figure 2.3. The LL instruction performs a read from a memory address and *links* the address for exclusive access. Later, the SC instruction can perform a write to a memory access only if the memory has been linked by LL. Any concurrent SC instruction or a regular store to the same address breaks the created linkage, causing the SC instruction to fail. In this case, the whole LL/SC sequence has to be retried.

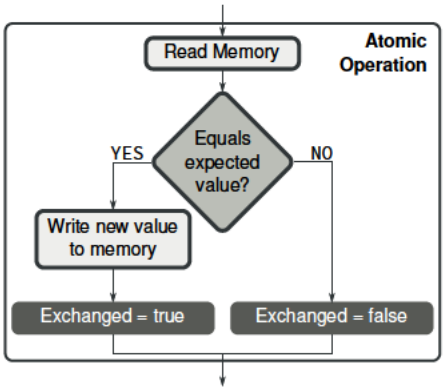


Figure 2.2: The compare-and-swap instruction atomically reads from a memory address, and compares the *current value* with an *expected value*. If these values are equal, then a *new value* is written to memory. The instruction indicates (usually through a return register or flag), whether or not the *value* was successfully written.

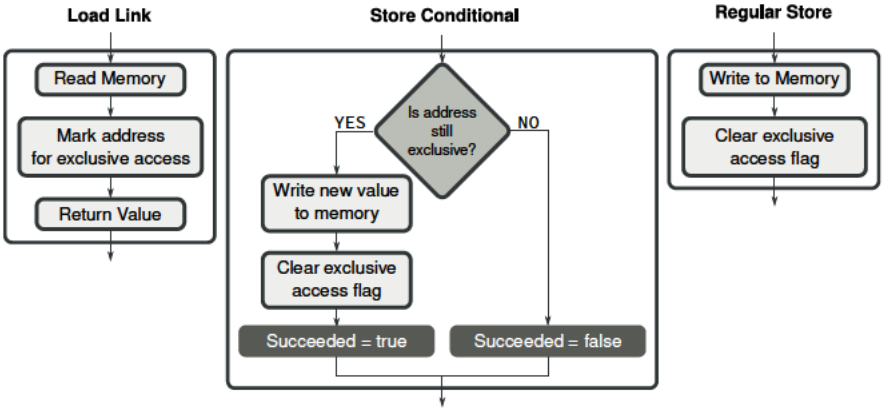


Figure 2.3: A load-link instruction reads memory, and internally marks that memory for exclusive access. A store-conditional instruction checks to see if a write has occurred by observing if the memory is still marked for exclusive access. If the memory has been written to, the store-conditional fails, otherwise the new value is written and the instruction succeeds. A regular store instruction clears the exclusive access marker.

While ISA specification only describes observable behaviour of atomic instructions, hardware designers are free to implement the required semantics in several ways, balancing between implementation complexity and performance. For example, x86's `CPXCHG` instruction is emitted with a *LOCK* prefix. *LOCK* prefixed instruction semantics implies asserting a memory bus lock, granting the CPU exclusive access to memory for the duration of the instruction. However, modern processors do not assert the memory bus lock if the affected memory is cached. Instead, only a cache line lock is acquired, delegating exclusive access semantics to a cache coherence protocol (see

Section 8.1.2 ”Bus Locking” of Intel’s Developer’s Manual [27]).

Uniprocessor implementations can potentially optimise out the bus locking altogether, while Non-Uniform Memory Access (NUMA) socket implementations suffer extraordinary latency for this type of instruction. David et al. [30] performed a comprehensive analysis of several hardware and software synchronisation methods on four x86 implementations with varying cache coherence protocols and the number of sockets.

On the other hand, LL/SC atomic instructions do not require locking the memory bus. Instead, each core can monitor the cache coherence traffic to detect concurrent stores (regular writes as well as SC instructions) at cache line granularity. After detecting a concurrent write into a cache line corresponding to a core’s private LL/SC linkage, the linkage is invalidated, resulting in the future SC instruction failing.

Although the performance of LL/SC implementations is typically better than the performance of CAS implementations under average congestion pressure, it can degrade in the worst case. For example, LL/SC can suffer from a live-lock due to false-sharing of the cache lines (i.e. a concurrent store targets the same cache line as the LL/SC target). Furthermore, TLB misses or interrupts can also break the LL/SC linkage. Therefore, in the worst-case scenario, LL/SC can experience spurious failures while CAS guarantees forward progress for at least one core even in a heavily congested context. Furthermore, CAS instruction performs read-write operations atomically, and thus cannot be spuriously disrupted by external interrupts or TLB misses.

Nevertheless, LL/SC atomics are being adopted by modern ISA designers. For example, RISC-V ISA [110] opted for LL/SC due to a simpler instruction format, more efficient implementation, and better suitability to build lock-free data structures. To guarantee forward progress, LL/SC sequences must fit within 64 bytes of static code and can only contain baseline instructions. Furthermore, regular loads and store are not allowed within the sequence, in order to minimise false-sharing.

In either case, atomic instructions are used to construct primitive locks for mutual exclusion of parallel instruction streams even in the presence of relaxed memory models. As a result, data races are prevented from corrupting shared data. Examples of spinlock implementations using CAS and LL/SC instructions are provided in Listing 2.5.1. Chapter 4 presents novel techniques of atomic instruction emulation in the context of an ISS.

```
lock_acquire:
  while (!CAS(lock, 0, 1));



lock_release:
  lock = 0;
```

(a) Compare-And-Swap operation atomically acquires a variable only if it is currently not acquired.

```
lock_acquire:
  do {
    while (LL(lock));
  } while (!SC(lock, 1);

lock_release:
  lock = 0;
```

(b) Load-Linked repeatedly reads the lock value, until it is observed as not acquired. Then, Store-Conditional attempts to atomically acquire the lock.

Listing 2.5.1: Implementations of a spinlock using CAS and LL/SC atomic instructions.

2.5.1.3 Transactional Memory

Another approach facilitates atomic updates at a much coarser granularity. Several memory accesses can be packaged into a single transaction to appear atomic. If a transaction conflicts with another concurrent transaction (i.e. there is a data race among the memory accesses), at least one of the transactions must be aborted and the corresponding memory accesses reverted. A software implementation of tracking memory accesses of transactions and resolving potential conflict is possible but requires substantial computational overhead. On the other hand, hardware transactional memory is implemented directly in silicon, resulting in much better performance. Special instructions can be issued to indicate the start and end of memory transactions. Transactional execution is speculative until the transaction is successfully committed. Typically, the hardware implementation monitors cache coherence traffic to detect data races. Although more efficient than software tracking, this approach requires specialised hardware support, resulting in worse portability. Furthermore, as hardware transactional memory tracks data races at cache line granularity, it can suffer from false sharing of cache lines and spurious transaction aborts. This is analogous to a typical LL/SC implementation. Chapter 4 utilises hardware transactional memory to emulate atomic instructions.

2.5.2 Hypervisor Design

Supporting multicore guest simulation requires a redesign of the hypervisor. Firstly, each guest core produces a unique instruction stream for guest execution. These

streams have to interact with each other, either directly or through a single shared memory. Secondly, parallel instruction streams can also result in concurrent accesses to hypervisor data structures, complicating their design.

2.5.2.1 Guest Execution Model

Simulating parallel guest applications requires duplication of the architectural state, resulting in multiple virtual CPUs used to emulate multiple individual guest instruction streams. In user-space simulation, system calls spawning guest threads can be intercepted to allocate virtual CPUs on demand [13, 20]. On the other hand, full-system simulation requires a configuration of the multicore guest machine at startup.

Simulating multiple virtual CPUs does not necessarily require the actual execution to be parallel. SIMICS [70] and QEMU full system [13] execute multiple virtual CPUs sequentially using round-robin scheduling by a single-threaded hypervisor (QEMU only parallelises execution when coupled with KVM hypervisor [45]). Similarly, detailed cycle-accurate simulation in gem5 [15] executes in a single thread to accurately model processor pipelines and memory interconnects. This is achieved by multiple guest core objects generating events at each simulated clock tick.

Single-threaded simulation of multicore architectures has distinct advantages. The simplicity of single-threaded programming requires relatively low engineering effort, as accessing hypervisor data structures does not need to be synchronised. Furthermore, the deterministic ordering of simulation events enables reproducibility and debugging of guest execution. However, the performance of single-threaded simulation is fundamentally limited, as it fails to utilise parallel hardware available on the host machine.

A more efficient approach executes each guest core by a dedicated host thread. As a result, true parallelism can be explored in simulation by allowing arbitrary interleaving of multiple guest execution streams. Furthermore, this simulation approach manages to utilise the underlying parallel hardware available on the host, resulting in improved simulation throughput.

This approach has been adopted by multiple user-space simulators [20, 13, 113, 76] and full-system simulators [34, 109, 106, 84, 107].

2.5.2.2 Hypervisor Data Structures

ISSs perform a lot of repetitive work. For example, an interpreter has to fetch and decode the same few guest instructions many times during emulation of an intensive

guest loop. To avoid repetitive work, ISSs employ many software caches to store results of frequent operations. After an initial warm-up period filling the various caches, simulation performance settles in a more optimal steady state, mostly using cached results to execute guest code.

The main performance improving cache in interpretive ISSs is a decode cache. Given a PC, the decode cache can bypass fetching and decoding the corresponding instruction by returning a fully decoded instruction object. Since this cache is accessed (and potentially updated) for every guest instruction, multicore simulators typically use core-private caches to avoid extensive synchronisation. As a result, decode caches often contain duplicate entries and consume a significant amount of memory.

In DBT based simulators, guest code compiled for native execution is cached for future reuse to amortise the cost of the initial compilation. Depending on the compilation granularity, a native cache is often more persistent with less frequent updates than a decode cache used for interpretation. Core-private and shared designs of the native cache are explored by Bruening et al. [21]. Although a core-private cache is much simpler to manage than a shared cache, it can result in inefficient memory usage. This is particularly relevant for server workloads spawning hundreds of threads sharing the same code. PQEMU [34] also experiments with both private and shared native caches, arriving at a similar conclusion.

Pico [28] employs a shared code cache. Although this design requires a course-grained lock during initial JIT code translation, its authors argue that translation is rare and most runtime is spent executing. On the other hand, simulators with core-private caches (e.g. QSim [55], Parallel Embra [64], COREMU [109]) can easily scale during code translation. Qelt [29] proposes fine-grained locking to enable parallel code generation while maintaining a shared native cache for memory efficiency.

Software solutions for guest memory address translation also employ caches to store host pointers for frequently accessed guest addresses [70, 101, 13]. Similarly to decode caches in interpreters, memory translation caches are accessed for every guest memory access. Therefore, they are implemented as core-private caches. This design can also be easily repurposed to model guest TLB structures.

Although a core-private memory translation cache allows each guest core to populate the cache without synchronisation, the caches might also be used to force special handling for selected memory regions on a global basis. This requires cross-core cache manipulation and thus poses a synchronisation challenge. For example, emulating atomic instructions manipulates the memory translation caches to force execution

of concurrent guest cores to return to the hypervisor for stop-the-world handling of atomicity.

Another example is SMC, which employs memory translation caches to disable cache hits for memory regions containing code, in order to detect and handle code modifications. This example is particularly challenging, as it also interacts with code caches described previously. On the other hand, OS page protection mechanisms used to detect SMC apply globally. That, in turn, introduces the problem of a *concurrent writer*. As the page fault handling during code modification needs to re-enable the page write permission globally, other concurrent cores could potentially write into the page without being detected. This problem has been explored by Hawkins et al. [47] by creating a parallel memory mapping with writable permission and additional instrumentation of the faulting write instructions. This work is the closest related publication to the contributions presented in Chapter 6.

2.5.3 Multicore Cross-ISA Simulation

Functional correctness in multicore cross-ISA simulation faces mismatches between the guest and the host notion of parallelism. Firstly, the guest and the host might have different memory consistency models. Simulating a weaker guest model on a stronger host model (e.g. arm-on-x86) is relatively trivial, as the host model guarantees no required orderings of guest memory operations are violated. However, if a stronger guest model is simulated on a weaker host, memory barriers must be inserted to achieve correct execution of the guest. This has been addresses by Lustig et al. [67] through a state machine design guiding the insertion of required memory barriers based on the differences between the guest and host memory models.

The second challenge is emulating differing atomic instructions. In particular, achieving the split-atomicity of LL/SC guest instructions on CISC-based hosts has been the subject of investigation. Rigo et al. [92] highlights the issues in correct handling of LL/SC instructions in QEMU, and provides a possible solution. Here, the slow-path of the memory translation cache is used to trap regular stores interfering with the LL/SC sequence.

A popular approach approximates LL/SC semantics using CAS instruction. During SC emulation, interleaving writes are detected using a CAS instruction to verify the memory location holds the same value as when the corresponding LL instruction was emulated. This, however, does not correctly simulate the semantics of LL/SC, as it suf-

fers from the ABA problem: a memory location is changed from value A to B and back to A [31]. Prior solutions developed in PQEMU [34] and COREMU [109] are shown to suffer from the ABA problem. Pico [28] also proposed a CAS approximation and further introduces support for hardware transactional memory for emulation of LL/SC synchronization, while an intermediate software approach uses extensive locking. This work is the closest related publication to the contributions presented in Chapter 4.

Jiang et al. [51] employs a lock-free FIFO queue for LL/SC emulation, but the paper is hard to follow and lacks a convincing correctness argument. XEMU [106] considers guest architectures with native LL/SC support. Gao et al. [40] presents a method to adapt algorithms using LL/SC to a pointer-size CAS without the ABA problem by using additional memory for each location, which would not be suitable for an emulator. The implementation presented assumes sequential consistency, but does include a formal proof mechanised in the Prototype Verification System (PVS) proof assistant. A more theoretical approach to LL/SC implementation without performance evaluation is taken in Michael [75]. A wait-free multi-word CAS operation is the subject of Feldman et al. [38].

2.6 Summary

There is substantial related work in the area of cross-architecture simulation, focusing on modelling various aspects of guest systems to enable the use of fast and accurate virtual machines. This chapter provided a description of the commonly used techniques employed by ISSs, including caching of decoded instructions using a Decode Cache for interpretive emulation and the caching of address translations using a Page Translation Cache for memory simulation.

The following chapters extend this body of work in the area of simulating multicore guest architectures that are incompatible with the host hardware. The new techniques focus on correctness, memory efficiency, and performance.

Chapter 3

Infrastructure

The contributions of this thesis are built using various tools and evaluation frameworks. This section describes the main ISS used as the research vehicle, nSIM, as well as its direct competitor, QEMU. QEMU's design is particularly important when used as an example workload for nSIM. Running QEMU as a guest application in nSIM demonstrates the challenges of hosting nested virtual machines. More generally, this use case is relevant for any instance of guest application exhibiting Dynamically Generated Code (DGC), e.g. instrumenting and profiling a V8 JavaScript engine.

The merits of different contributions are evaluated using industry standard benchmarks with varying characteristics. Both sequential and parallel benchmarks are used. Furthermore, contributions are also evaluated using targeted micro-benchmarks, but these will be introduced in theirs respective technical chapters.

3.1 nSIM hypervisor

nSIM is a high speed ISS used as a research vehicle throughout this thesis. It has been developed at the University of Edinburgh as ArcSim [101, 52, 17] and later commercialised by Synopsys. nSIM supports both user-space and full-system simulation of ARCompact processors [97]. Furthermore, it provides a rich framework for software and hardware designers, including instruction tracing, pipeline modelling for cycle-accurate simulation [18, 87, 100], and functional guest cache modelling. nSIM also supports scalable simulation of multicore guests by executing each guest virtual core in a separate host thread [61, 100].

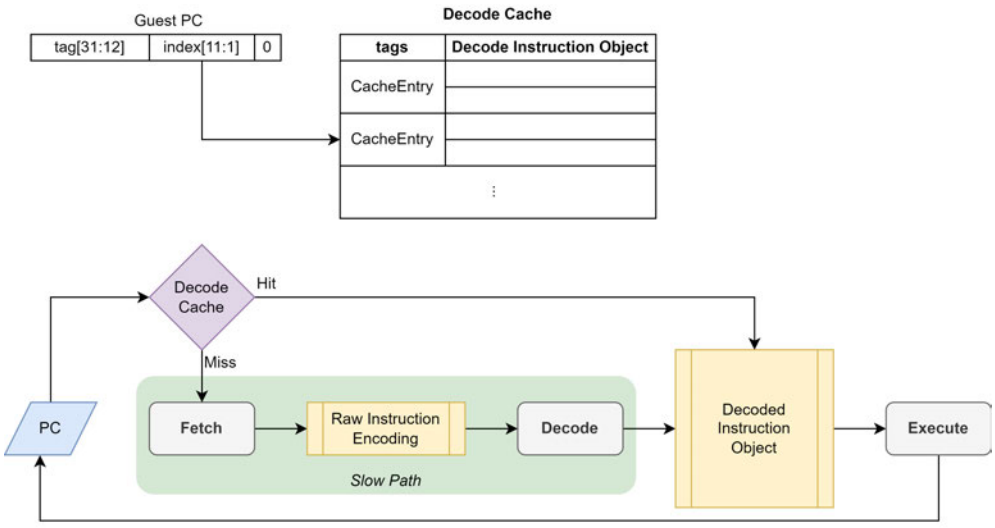


Figure 3.1: **Decode Cache** data structure and operation flow.

3.1.1 Instruction Emulation

Guest code execution is performed using a seamless combination of interpretation and DBT. The interpreter employs core-private decode caching to speed up fetch and decode operations for the most frequent instructions. If an instruction is not found in the decode cache, expensive *Fetch* and *Decode* operations must be performed. Afterwards, the resultant *Decoded Instruction Object* is inserted into the cache to be reused in the future.

The size and associativity of the decode cache is configurable but is fixed during runtime operation. Note, decode cache is a software-only data structure allocated by the host. It is functionally similar to a hash map with a rigid memory layout. The decode cache data structure and its integration in the interpretation loop in depicted in Figure 3.1.

In addition to standalone interpretation, nSIM can emulate guest instructions using a mixed interpretation and native/DBT mode. Such operation is shown in Figure 3.2. At startup, the instruction emulation is performed using interpretation with additional profiling to identify code regions with significant execution heat (indicated as *Tracing*). After each tracing interval (10000 guest instructions), hot code regions are collected into compilation units that are dispatched to concurrent JIT compilation workers [17].

Each compilation unit comprises a complex control flow graph of multiple basic blocks bounded only by the page boundaries in the guest instruction space. As a result, the code quality of the generated code is high at the cost of compilation latency.

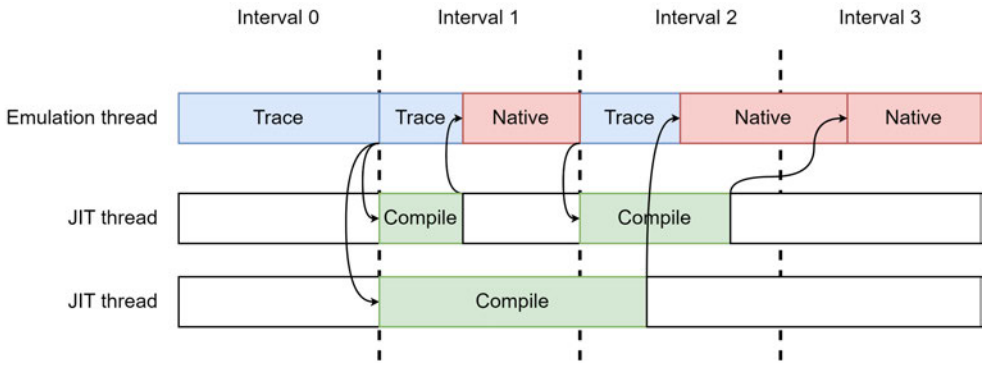


Figure 3.2: Instruction Emulation in a **mixed mode** with concurrent and parallel JIT compilation.

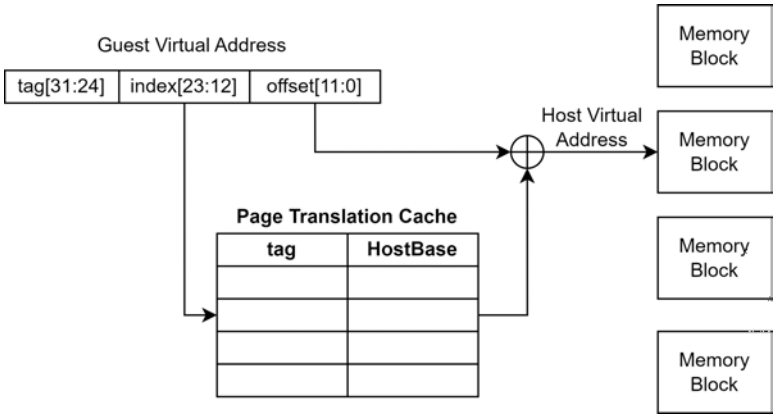


Figure 3.3: **Page Translation Cache** facilitates translation of guest virtual address to host virtual address. Page Translation Cache (PTC) is instantiated for *read*, *write*, and *execute* access types.

However, the latency is partially hidden by continued instruction emulation using the interpreter.

3.1.2 Memory Simulation

Guest memory address translation is facilitated using core-private PTC [101]. Each core has three PTCs dedicated for read/write/execute access types. A page entry is present in PTC only if the guest page is currently mapped by the guest TLB, the guest process has sufficient permission for the corresponding access type, and the memory location does not have any side effects (e.g. memory mapped I/O addresses are excluded from PTC).

Upon a cache hit, guest memory access is performed by a simple pointer indirection to the backing memory on the host. However, if a guest memory address is not found

Simulation mode	user-space, full system
Execution mode	interpreted, interpreted/native
JIT design	asynchronous, parallel JIT workers
Number of JIT workers	3
JIT granularity	page-bound regions
Multicore execution	one guest core per host thread
Native cache sharing	global
Native cache capacity	unlimited, dynamically allocated
Decode cache sharing	core-private
Decode cache capacity	4K entries, statically allocated
Page Translation Cache	core-private
Page Translation Cache capacity	1024 entries per type, statically allocated

Table 3.1: Default nSIM configuration

in PTC, the hypervisor has to perform expensive memory translation, potentially allocating a new *Memory Block* to provide backing of the guest memory, and check access permissions. The data structure of PTC is depicted in Figure 3.3.

A brief overview of nSIM default configuration and functionalities is depicted in Table 3.1.

3.2 QEMU hypervisor

QEMU [13] is an alternative ISS targeting many guest and host ISAs. Simulation capabilities include both user-mode application and full system simulation capacity to run an entire multicore OS. It is popular in both academia and industry and has been extended or incorporated in many other simulation tools [34, 109, 49, 106, 83, 86, 24]

Guest code execution employs DBT at a basic block granularity. Tiny Code Generator (TCG) translation is divided into a guest-specific front-end producing a TCG Intermediate Representation (IR). Simple optimisations are applied to the IR of each guest basic block before being passed to the TCG back-end. Host specific code is generated into a contiguous buffer as a Translation Block (TB). Each TB start address is stored in a translation cache, indexed by the guest physical PC address for future reuse. In multicore execution, translation is synchronised using a global lock, i.e. generating new native code is sequentialised.

In this thesis, QEMU is used as a direct comparison point in Chapter 4 as well as an evaluation artifact in Chapter 6. In both cases, QEMU version `6.2.50` is considered.

3.3 Benchmarks

A wide range of industry standard benchmark suites has been used to evaluate the novel contributions of this thesis, exemplifying common computational problems across different domains.

3.3.1 SPEC CPU Benchmark Suite

SPEC CPU benchmark suite has become the most frequently used and widely accepted suite for computer architecture research. It is developed by SPEC (The Standard Performance Evaluation Corporation), a non-profit organisation. The workload is designed to represent realistic real-world applications testing cpu performance based on integer computation. Benchmarks testing floating point computation were excluded. Since nSIM (the ISS used as a research vehicle for this thesis) is not designed for emulation of floating point arithmetic, these benchmarks would have to employ software-emulated floating point using integer instructions, and hence does not add further value to the evaluation.

This thesis uses SPEC-CPU2006 Int benchmarks [48] (table 3.2) to evaluate various contributions by directly running in nSIM. These benchmarks were already ported to the ARC ISA and used in previous works [17, 35]. Furthermore, SPEC-CPU2017 Int benchmarks [65] are compiled for the ARM ISA to run as a workload for QEMU running under nSIM. This evaluation mode does not affect the nSIM workload directly. Instead, each benchmark uniquely stresses QEMU's TCG, which acts as a direct workload for nSIM.

3.3.2 BioPerf Benchmark Suite

BioPerf [9] is a domain specific benchmark suite targeting bioinformatics applications using computational methods sifting through massive biological data. The purpose of these benchmarks is to evaluate high-performance computer architectures. The individual benchmarks are particular invocations of the packages presented in table 3.3.

The BioPerf suite was already compiled for the ARC ISA and used in previous work [17, 16].

Benchmark	Application Domain
400.perlbench	Mail filtering using the Perl programming language
401.bzip2	Data compression using bzip2 modified to do most work in memory
403.gcc	C compilation based on gcc Version 3.2
429.mcf	Combinatorial optimization in vehicle scheduling of public transport
445.gobmk	Artificial Intelligence: Plays the game of Go
456.hmmer	Search Gene Sequence of protein using profile hidden Markov models
458.sjeng	Artificial Intelligence: A highly-ranked chess program
462.libquantum	Quantum Computing simulation running Shor's factorization algorithm
464.h264ref	Video compression using H.264/AVC
471.omnetpp	Discrete Event Simulation using the OMNet++ simulator
473.astar	Path-finding Algorithms for 2D maps using A* algorithm
483.xalancbmk	XML Processing transforming XML documents to other types

Table 3.2: Description of benchmarks from the SPEC CPU2006 Integer suite.

3.3.3 EEMBC MultiBench Benchmark Suite

EEMBC (The Embedded Microprocessor Benchmark Consortium) develops and maintains benchmarks designed for embedded systems. EEMBC MultiBench [36] allows designers to evaluate multicore processors with three forms of concurrency: fine-grained parallelism, data parallelism, and task parallelism. The suite comprises many benchmarks that are defined using the computational kernels described in table 3.4.

3.3.4 SPLASH-2 Benchmark Suite

SPLASH-2 [111] is an updated version of SPLASH (Stanford Parallel Applications for Shared Memory) benchmarking suite consisting of parallel workloads, including mostly algorithms from linear algebra and computational physics. The suite is designed to measure the performance of these applications on centralized and distributed shared-address-space machines. The individual benchmarks are described in table 3.5.

Benchmarks have been compiled for the ARC ISA and linked against a custom pthread library to run in a bare-metal mode in nSIM. These benchmarks employ relatively small computational kernels with focus on HPC data processing. Therefore, the benchmarks can easily scale to a large number of cores.

Package	Application Domain
blast	Word-based sequence homology
hmmer	Profile-based sequence homology
fasta	Pairwise sequence alignment
tcoffee	multiple sequence alignments
dnapenny, promlk	Parsimony/likelihood phylogeny
grappa	Gene rearrangement phylogeny
predator	Protein structure prediction
glimmer	Gene Finding
ce	Molecular dynamics

Table 3.3: Description of benchmarks from the BioPerf suite.

Kernel	Application Domain
md5	MD5 checksum over multiple input buffers
h.264	H.264/AVC video encoding
iDCT	Inverse Discrete Cosine Transform of an integer matrix
RGB to CMYK	Image processing for colour printers
Image Rotation	Rotating greyscale or colour images in memory
IP Packets Check	Checksum in network simulation
IP Reassembly	Reassembling fragmented network packets
TCP	Transmission Control Protocol simulation

Table 3.4: Description of benchmarks from the EEMBC MultiBench suite.

3.3.5 PARSEC Benchmark Suite

Princeton Application Repository for Shared-Memory Computers (PARSEC) [14] is a modern benchmark suite for parallel computing. It comprises diverse examples of emerging workloads with larger problem sizes than SPLASH-2 benchmarks, including recognition, mining and synthesis of large-scale data inputs. The PARSEC suite is considered more representative of current parallel workloads compared to SPLASH-2 suite. The individual benchmarks are presented in table 3.6.

The suite supports several parallel models, namely `pthread`, `openmp`, and `tbb`. In this thesis, the benchmarks were compiled for the ARC ISA using a custom pthread library to enable running in a bare-metal environment. Due to the larger size of the benchmarks, the suite is run in up to 10 cores configuration, hence supporting 10 guest

Benchmark	Application Domany
barnes	N-body problem simulation in 3D
cholesky	Sparse matrix factorisation
fft	Fast Fourier transform algorithm
fmm	Fast Multipole Method N-body simulation in 2D
lu	Dense matrix factorisation
ocean	Ocean movements based on eddy and boundary currents
radiosity	Light distribution in a scene of polygons
radix	Radix sort kernel
raytrace	3D scene rendering using ray tracing
volrend	3D volume rendering using ray casting
water	Water molecules simulation

Table 3.5: Description of parallel benchmarks from the SPLASH-2 suite

threads. This configuration reserves 2 threads for spawning and managing workers, and up to 8 threads processing a given workload.

Benchmark	Application Domain
blackscholes	Financial Analysis of European stock options
fluidanimate	Animation using smoothed particle hydrodynamics
ferret	Content-based similarity search
streamcluster	Data mining using online clustering
x264	H.264/AVC video encoder compression

Table 3.6: Description of parallel benchmarks from the PARSEC suite

3.4 Summary

This chapter presented tools and benchmarking frameworks used to demonstrate and evaluate contributions of this thesis. In particular, all contributions are implemented in a high performance mixed-mode ISS, nSIM. A varied collection of benchmarks is used to stress and evaluate different parts of the simulation tool, ranging over several application domains and both single and multicore contexts. The described infrastructure will be used throughout the following three technical chapters.

Chapter 4

Emulation of Atomic Synchronisation Instructions

Atomic instructions are the key building blocks of parallel software running on multicore hardware. Various architectures adopt different design philosophies to support atomicity at the ISA level. CISC architectures opt for complex read-modify-write instructions (e.g. CAS) while RISC architectures split atomicity into two LL and SC instructions (see Section 2.5.1.2), in line with the RISC philosophy that focuses on reducing the complexity of instructions performed by the hardware. For ISSs to successfully simulate cross-ISA platforms, they must faithfully emulate guest atomic instructions on the host platforms.

The **key challenge** is how to emulate guest LL/SC instructions on hosts that only support CAS instructions. In this case, guest atomicity cannot be directly mapped onto the host ISA. Instead, a software approach must ensure guest atomicity is emulated accurately. In other words, full semantic behaviour of guest LL/SC instruction must be supported by the simulator. As a result, it should not be possible to construct a guest program that would behave differently inside a simulator and on a real hardware.

Since LL/SC linkage ought to be broken by any write to the same memory, the efficiency of detecting intervening memory writes becomes crucial for overall performance. Monitoring linkage invalidations in a single-threaded hypervisor (including a hypervisor with round-robin scheduling of multiple guest cores) is trivial. LL/SC linkage can be simulated together with a model of each virtual core. Then, the linkage can be verified across the whole simulated system during emulation of an SC instruction. On the other hand, hypervisors emulating each guest core in a dedicated host thread must support concurrent emulation of multiple LL/SC instructions. This poses extra

pressure on accurately mapping the guest synchronisation model onto the host execution. In particular, the emulated execution must ensure invariants (i.e. intervening stores cause LL/SC to fail) are never violated.

A simple naïve approach, is to synchronise all store instructions, using critical sections to enforce atomicity when executing the instruction. However, such a naïve scheme incurs a high runtime performance penalty. A better approach to detect intervening stores efficiently is to use a CAS instruction. In this approach, the guest LL instruction reads memory, and stores the value of the memory location in an internal data structure. Then, the guest SC instruction uses this value as the *expected* parameter of the host CAS instruction. If the value of the memory location has not changed, then the SC can succeed. Unfortunately, although fast, this approach does not preserve the full semantics of LL/SC instructions, as it suffers from the ABA problem: A memory location is changed from value A to B and back to A [31]. In this scenario, two interfering writes to a single memory location have occurred, and the SC instruction should fail. However, since the *original value* has been written back to memory, this goes unnoticed by the host CAS instruction, and the emulated SC instruction incorrectly succeeds as a result. This broken approach can be referred to as the *CAS approximation*, since this emulation strategy only *approximates* the LL/SC semantics.

4.1 Motivating Example

Figure 4.1 shows how the *CAS approximation* implements the operation of the *load-link* and *store-conditional* instructions. The *load-link* instruction records the value of memory (into `linked-value`) at the given address, and returns it as usual. The *store-conditional* instruction performs a compare-and-swap on the guest memory, by comparing the current memory value to `linked-value` (from the load-link instruction), and swapping in the new value if the old values match. If the CAS instruction performed the swap, then the *store-conditional* is deemed to have been successful.

4.1.1 Trivial Broken Example

The sequence in Table 4.1 describes a possible interleaving of guest instructions, which will cause LL/SC instructions implemented with the *CAS approximation* to generate incorrect results.

The *CAS approximation* approach violates the semantics of the LL/SC pair at **t3**.

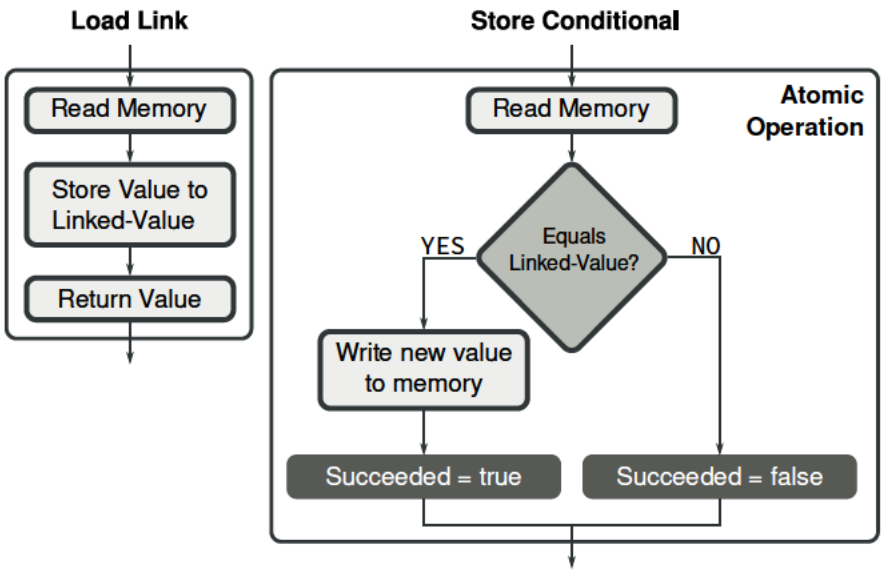


Figure 4.1: An example of how compare-and-swap semantics can be used to approximate load-link/store-conditional semantics.

Time	Core 1	Core 2
t_0	LL(0x1000) $\rightarrow$ #1	
t_1		Write(0x1000, #2)
t_2		Write(0x1000, #1)
t_3	SC(0x1000, #3) $\rightarrow$ CAS(0x1000, #1, #3) $\rightarrow$ **SUCCESS**	

Table 4.1: A trivially incorrect example of LL/SC emulating using CAS approximation

At this time point, the value in memory has been restored to its original value from **t0**, and so the SC instruction **incorrectly** assumes no writes were made, and hence succeeds. However, this assumption is **wrong**, since two interfering writes (at **t1** and **t2**) have been performed, and thus should cause the *store-conditional* instruction to return a `FAILED` result.

A simple program to test this behaviour was executed on both a real 64-bit Arm machine, and an emulation of a 64-bit Arm machine using QEMU. Resultant analysis shown that the test binary behaved as expected on the real Arm machine, but incorrectly performed a successful *store-conditional* in QEMU. When nSIM was configured to also use the CAS approach, it too exhibited the incorrect behaviour.

4.1.2 ABA Problem in Lock-free Data Structures

In practice, the trivial broken example described previously can lead to the ABA problem appearing in multicore data structures that have collaborating operations [31], and

```
1 ElementType *pop(Stack *stack) {
2   ElementType *originalTop, *newTop;
3   do {
4     originalTop = LoadLink(stack->top);
5     newTop      = originalTop->next;
6   } while (StoreConditional(stack->top, newTop) == FAILED);
7   return originalTop;
8 }
```

Listing 4.1: Implementation of a lock-free stack `pop` operation.

Time	Core 1	Core 2	Stack State
t_0	**Begin `pop()` operation**		✓ `top->A->B->C`
t_1	`originalTop = LoadLink(top) == A`		
t_2	`newTop = originalTop->next == B`		
t_3		`pop() == A`	✓ `top->B->C`
t_4		`pop() == B`	✓ `top->C`
t_5		`push() == A`	✓ `top->A->C`
t_6	`StoreConditional(top, newTop)` ⇒ `Compare-and-Swap(top, A, B)`		✓ `top->A->C` ✗ `top->B->?`

Table 4.2: An example thread interleaving of lock-free stack manipulation exhibiting the *ABA* problem, when the *CAS approximation* is used to implement LL/SC instructions.

that utilise LL/SC as the synchronisation primitives.

For example, a lock-free stack implementation based on linked-list loads its *shared* `top` pointer (the *load-link*), prepares the desired modifications to that data structure, and only updates the shared data provided there has been no concurrent update to the shared data since the previous load (using the *store-conditional*). A typical implementation of the `pop` operation, using LL/SC semantics is given in Listing 4.1.

Table 4.2 shows a possible interleaving of operations on a lock-free stack, which leads to incorrect behaviour when the *CAS approximation* is used to emulate LL/SC instructions. The sequence of events is as follows:

- **t0:** *Core 1* starts executing a `pop()` operation.
- **t1:** *Core 1* performs a *load-link* on the stack top pointer, and stores the resulting element in `originalTop`. In this case, object `A`.
- **t2:** *Core 1* now resolves the `next` pointer, which points to object `B`, and stores it

in `newTop`.

- **t3:** *Core 2* now pops object `A` from the stack.
- **t4:** *Core 2* then pops object `B` from the stack.
- **t5:** *Core 2* finally pushes object `A` back on the stack.
- **t6:** At this point, *Core 1* attempts to update the top of the stack pointer, using a *store-conditional* instruction to write back `newTop` (which is object `B`).

Now, with correct LL/SC semantics, the *store-conditional* instruction will return `FAILED`, to indicate that an intervening write has occurred (even though the value in memory has not changed). However, if the *CAS approximation* is used to implement the *store-conditional* instruction, then it will not detect that any writes to `stack->top` have occurred (since the *load-link* in **t1**). As a result, the top pointer will be incorrectly updated to `B`, since the current value of the pointer matches the expected value `A` (i.e. `originalTop == stack->top`). This can lead to undefined behaviour, as the correct stack state has been lost. For example, stack element `B` can now point to reused, or invalid/freed memory.

Since the *CAS approximation* detects modifications to memory only through changes of values, other techniques must be used to prevent the ABA problem and guarantee correctness [33, 74, 73].

Similar to the QEMU experiment detecting incorrect behaviour for the trivially broken example, a test program of an LL/SC based implementation of a lock-free stack can also be constructed. We discovered that in QEMU, interleaving stack accesses as depicted in Table 4.2 results in stack corruption.

This chapter presents a novel, provably correct scheme for implementing *load-link*/*store-conditional* instructions in an ISS. In other words, under the correct scheme, all possible guest programs behave identically inside a simulator and on a real hardware. Furthermore, this chapter shows that the correct implementation delivers application performance levels comparable to the broken *CAS approximation* scheme.

4.2 Methodology

This section introduces four schemes for handling LL/SC instructions in ISS systems. These schemes range from a naïve baseline scheme, through to an implementation utilising hardware transactional memory (HTM).

1) **Naïve Scheme** (Section 4.2.1): This scheme inserts standard locks around *every* memory write instruction for tracking linked addresses, effectively turning memory writes into critical sections. The scheme is correct, but impractical due to the extensive performance penalty associated with extensive synchronisation.

2) **Broken: Compare-and-Swap-based Scheme** (Section 4.2.2): This scheme is used in state-of-the-art ISS systems such as QEMU. The scheme maps guest LL/SC instructions onto the host system's *compare-and-swap* instructions, resulting in high performance. However, it violates LL/SC semantics.

3) **Correct: Software-based Scheme** (Section 4.2.3): This scheme utilises facilities available in the ISS system to efficiently manage linked memory addresses, by taking advantage of the *page translation cache*.

4) **Correct: Hardware Transactional Memory** (Section 4.2.4): This scheme exploits the *hardware transactional memory* to efficiently detect conflicting memory accesses in LL/SC pairs.

The handling of LL/SC instructions in ISS systems closely follows the observed hardware implementation of these instructions. Each *load-link* instruction creates a CPU-local record of the *linked* memory location (i.e. storing the memory address in a hidden register). Then, the system monitors all writes to the same memory location. If a write is detected, the linkage of *all* CPUs is broken, to ensure that no future *store-conditional* instruction targeting the same memory location can succeed.

Emulating *store-conditional* instructions requires verifying that the local linkage is still valid. If so, the *store-conditional* can succeed, and invalidate the linkage of other CPUs for the same memory location atomically.

Since emulating an SC instruction comprises several checks and updates that must appear to be atomic, this emulation must be properly synchronised with the emulation of concurrent SC and LL instructions. Furthermore, concurrent *regular* stores that interleave between the LL/SC pairs have to be detected so that future SC instructions cannot succeed. Detecting interleaving regular stores is the main challenge in efficient emulation of LL/SC instructions.

4.2.1 Naïve: Lock Every Memory Access

A naïve implementation of LL/SC instructions guards all memory operations to the corresponding memory address with the same lock. Conceptually, a global lock can be

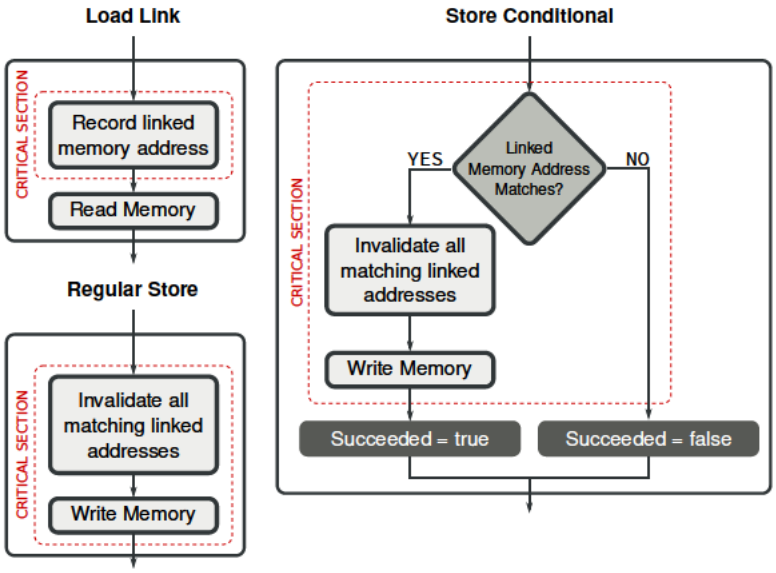


Figure 4.2: A naïve scheme to emulate *load-link* and *store-conditional* instructions. Critical sections are used to serialize all (including regular) memory accesses.

used to guarantee mutual exclusion of all LL/SC and regular stores. In practice, more fine-grained locking can be used to improve performance, by allowing independent memory locations to be accessed concurrently.

This emulation scheme is presented in Figure 4.2. A *load-link* instruction enters a critical section, and creates a local linkage under mutual exclusion with respect to *store-conditional*, and regular stores to the same locations.

The *store-conditional* instruction checks the local linkage, and if it matches, then it performs the actual write to the memory address. The linkages of other CPUs corresponding to the same memory address are also invalidated, so that no future *store-conditional* can succeed. The emulation of a regular store invalidates the linkage on all CPUs corresponding to the same memory address. The actual write is performed unconditionally.

Although simple, this scheme suffers a significant performance penalty, due to the critical sections causing a slow-down of all regular store instructions. Lock acquisition has to be performed by every regular store, even those that do not access memory locations used by any LL/SC pairs. For typical applications, the majority of regular stores are slowed down unnecessarily.

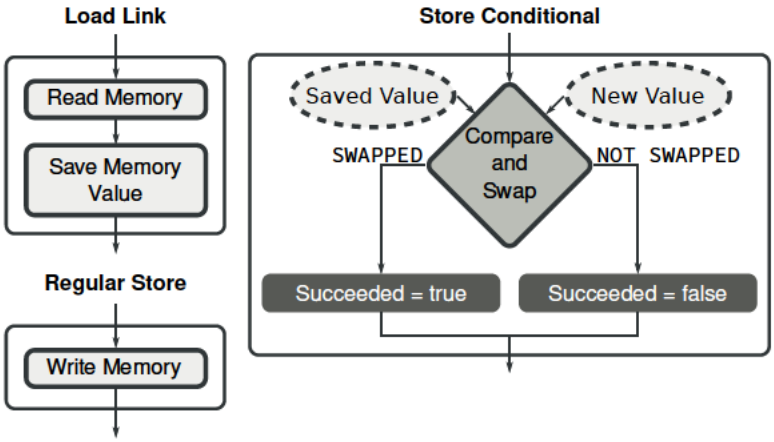


Figure 4.3: CAS approximation of LL/SC.

4.2.2 Broken: Using CAS Style Semantics

This scheme uses a *compare-and-swap* instruction to approximate the semantics of *load-link/store-conditional* pairs. The LL/SC linkage comprises not only the *address*, but also the memory *value* read by the LL instruction.

Emulating the *load-link* instructions saves the linked address as well as the linked value (i.e. the result of the corresponding *read*). Then, emulating the *store-conditional* instruction uses the linked value for comparison with the current value of the linked memory location using the CAS instruction. If the current value of the memory at the previously stored linked address does not match the linked value saved from the previous LL instruction, an interleaving memory access has been detected. Since intervening writes to the memory location are detected by changes in *memory value*, no linkage invalidation of other CPUs is required. Similarly, regular stores can proceed without any need of linkage invalidation or synchronisation.

This scheme offers great performance, since the emulation of LL/SC and regular stores does not need to synchronise at all. Furthermore, the *compare-and-swap* instruction is a well established synchronisation mechanism, and thus its performance is optimised by the host hardware. However, as we have demonstrated, the CAS scheme only approximates the semantics of LL/SC instructions and in particular, utilising the CAS scheme for this purpose can cause the execution of guest programs to break.

4.2.3 Correct Software-Only Scheme

This scheme improves upon the approach taken by the naïve scheme. The **key idea** is to slow down only those regular stores that access memory locations that are cur-

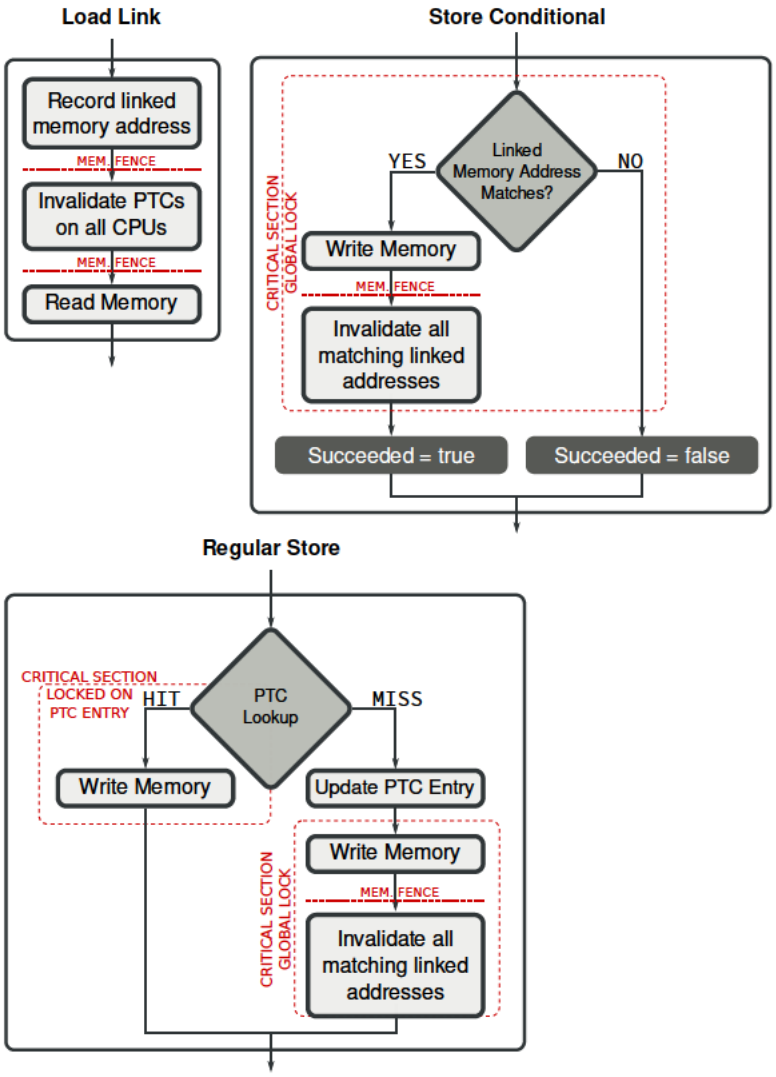


Figure 4.4: A software-only scheme emulation of LL/SC, utilising the ISS system's page translation cache.

rently being used by LL/SC instructions. To achieve this, we take advantage of a software Page Translation Cache (PTC), which is used in the ISS system to speed up the translation of *guest* virtual addresses to *host* virtual addresses. Execution of guest memory accesses can query this cache to avoid a costly full memory address translation, which may incur walking guest page tables, and resolving guest physical pages to corresponding host virtual pages (see Section 3.1).

Upon a hit in the cache, the hypervisor can use a fast-path version of the instruction emulation. For regular stores, fast-path version involves no synchronisation with LL/SC instructions. Note that each core has its own private PTC, and that there exists a number of separate per-core PTCs based on memory access type (e.g. read, write, execute).

Time	Core 1	Core 2
t_0	**Regular Store**	**Load-link**
t_1	PTC lookup **hits** in cache	
t_2		Create linkage
t_3		Invalidate PTC
t_4		Read data from memory
t_5	Write data to memory	

Table 4.3: A particular interleaving of operations that leads to a race-condition in the Software-only scheme.

This scheme allows regular stores that are not conflicting with LL/SC memory addresses to proceed without any slowdown, by only synchronising if the Write-PTC query misses in its corresponding cache. To achieve this behaviour, emulating a *load-link* instruction involves invalidating the Write-PTC entry for the corresponding memory address. Then, future regular stores will be guaranteed to miss in the cache, and will be forced to synchronise with concurrent LL/SC instructions. In other words, no concurrent regular store can enter the fast-path without invalidating all LL/SC linkages for the corresponding memory address.

4.2.3.1 Page Translation Cache Race Condition

Implementing the Software-only scheme without regards for the ordering of operations between cores leaves room for a race condition to appear. This particular scenario is depicted in Table 4.3.

In this interleaving, on *Core 1*, the emulation of a regular store performs a successful PTC lookup, and enters the fast-path at time t_0. At this point, *Core 2* performs the full emulation of a *load-link* instruction, i.e. the linkage is established, the corresponding PTC entry is invalidated, and the data is read from memory (t_2 through t_4). Then, back on *Core 1*, the emulation of the regular store actually updates the data in memory, at time t_5.

After this execution, a *store-conditional* instruction on *Core 2* can still succeed, as *Core 1* did not invalidate the underlying linkage. However, the data read by *Core 2* at time t_4 is now out of date, since an interleaving write to the memory address (*Core 1* at t_5) has been missed.

This race-condition manifests itself when PTC invalidation performed by a LL instruction happen after a successful PTC lookup by a regular store, but *before* the

actual data write by the regular store instruction.

To prevent such behaviour, we protect the PTC entry during fast-path execution of regular stores by making a cache line tag annotation. In particular, successful PTC lookups atomically annotate the underlying tag value by setting a bit corresponding to an ongoing fast-path execution. This tag bit can be cleared only after the actual data update has happened. While the tag bit is set, PTC invalidation during LL emulation is blocked.

4.2.3.2 Optimisations

There is a number of optimisations that can be applied to the software-only scheme, to make further improvements.

Firstly, since the linkage comprises a single word, it can be updated atomically. This means that mutual exclusion is not required for LL emulation. However, operations still have to have a particular order to guarantee correctness. In the case of LL emulation, the data read must happen *after* the PTC invalidation, which must happen after the linkage is created. The required ordering is depicted in Figure 4.4 using a red dotted line to indicate a memory fence across which operations cannot be reordered.

Secondly, since the emulation of LL itself does not use locks for mutual exclusion, the state of the underlying lock can be used to infer if a concurrent write operation is in flight. This can be used to immediately fail *store-conditional* instructions without checking whether or not the linkage is present. If the emulation of an SC instruction observes the lock as being already held, then there is already a concurrent update to the memory location in progress, so the SC instruction will fail anyway. In this scenario, the SC instruction fails immediately. In other words, *store-conditional* emulation acquires the underlying lock if and only if it has enough confidence that it is still possible to succeed. Note that the emulation of regular stores *always* acquires the lock, as this write is unconditional.

These optimisations reduce the amount of code that executes within a critical section, and especially in many-core applications, this results in less code having to serialise, thus resulting in much better scalability.

4.2.3.3 Proof of Correctness

This software-only scheme has been formally proven to correctly emulate the semantics of LL/SC instructions. The proof was developed in collaboration and is not pre-

sented here as my own sole work.

We assume that the host has a Total-Store-Order memory model like x86, where each core's writes appear in memory in program order. However, guest architectures with LL/SC such as ARM will often have a weaker memory model, so first we define the expectations for LL/SC in such a model.

4.2.3.3.1 Axiomatic Definition of Atomicity In an axiomatic memory consistency model such as ARMv8 [88], we assume a coherence order relation (co) between all Writes to the same address, and a reads-from relation (rf) between a Write and a Read that reads its value. A from-reads relation (fr) can be derived from these as between a Read and a Write that is later (in co) than the Write the Read reads-from. We can also consider parts of these relations that relate events from different external threads, and call these coe, fre, rfe.

The atomicity condition for LL/SC is then that there is no fre followed by coe between any LL and a successful SC. Expanding this definition, since the LL incorporates a Read, and there cannot be a Write from a different thread after (in coherence order co) the one that reads-from and before the Write from a successful SC. This captures ARMv8-like LL/SC correctness. For architectures which allow no Writes (not even same-thread Writes) between the LL and SC, the condition can be stated as there is no fr followed by co between any LL and a successful SC.

4.2.3.3.2 Axiomatic Definition of Properly Locked Executions For locks, we have two events, lock acquire and lock release. The definition of Properly Locked executions says that any successful lock acquire is followed by the corresponding lock release, and preceded by the previous release (or no lock events for the initial acquire). This order must be common to all threads. Further, same thread events (Writes, Reads) after the lock acquire are globally ordered after the acquire, and same thread events before the release are globally ordered before the release.

4.2.3.3.3 Correctness of the Naïve Scheme All LL, SC and Writes are guarded by locks, and furthermore the same lock is used for any particular address.

4.2.3.3.4 Proof We have a global order for any properly locked executions. Since all LL, SC and Write events are between successful lock acquires and releases on the same thread, they are globally ordered by the lock ordering. We can read off coherence

order co (and coe) as a subset of this global order, and similarly for from-reads order fr (and fre). Note that the place of a LL in this lock order corresponds to when its address is saved, not when the read is done. This is safe because if any write (SC or normal) from other threads intervenes in the lock order between the LL and an SC, the lock address is guaranteed to be invalidated, and thus the SC must fail. The correctness of the atomicity condition is thus ensured by the check done for successful SC (within a locked region) that no other thread has done a Write since the last LL.

4.2.3.3.5 Correctness of the Software-only Scheme The SC and slow path Writes are still guarded by locks, but Writes on the fast path used when the page table cache lookup succeeds are guarded by a different lock (tag bit) in the cache entry. An LL invalidates the cache entries for the matching virtual address on all other threads (within a locked region), spinning on the lock in the old cache entry to avoid races with ongoing fast path Writes, *before* invalidating all other locked addresses, *and then* reading the value. Plain Writes then check whether the virtual address is valid on that thread. If so, the Write action is done immediately (fast path). If not (slow path), the thread acquires the lock, performs the Write action, and *then* clears matching locked addresses on all other threads (and releases the lock). An SC checks whether the locked address is still held, and if so then does the Write and succeeds, otherwise it fails.

4.2.3.3.6 Proof If all Writes go through the slow path then the proof is analogous to the naïve version. The only wrinkle is that the LL is not locked and therefore does not participate in the lock ordering. Consider however that the LL only does the underlying read of the value after saving the locked address, and the locked address is invalidated by any such write, *after* the write. So either the LL reads the old value, and then the write (and possible invalidation) occurs, or the LL reads the new value, but if so the locked address will get invalidated before the writing thread releases the lock. Both cases are safe. Now, let us consider the case that one or more Write goes through the fast path. We show that no such Write can come in between the LL and a successful SC (in the time order implied by the ordering between the LL and SC's acquire). Indeed, using the fast path means the virtual address is valid on the Write's thread. Moreover, it must be valid *when the Write happens* because it is protected by the cache tag entry lock. Since the LL invalidates virtual addresses on all threads before doing the Read, this means the Write must be ordered before the invalidation which is part of the LL, or else the write cannot go on the fast path until after the SC mutex release. In the

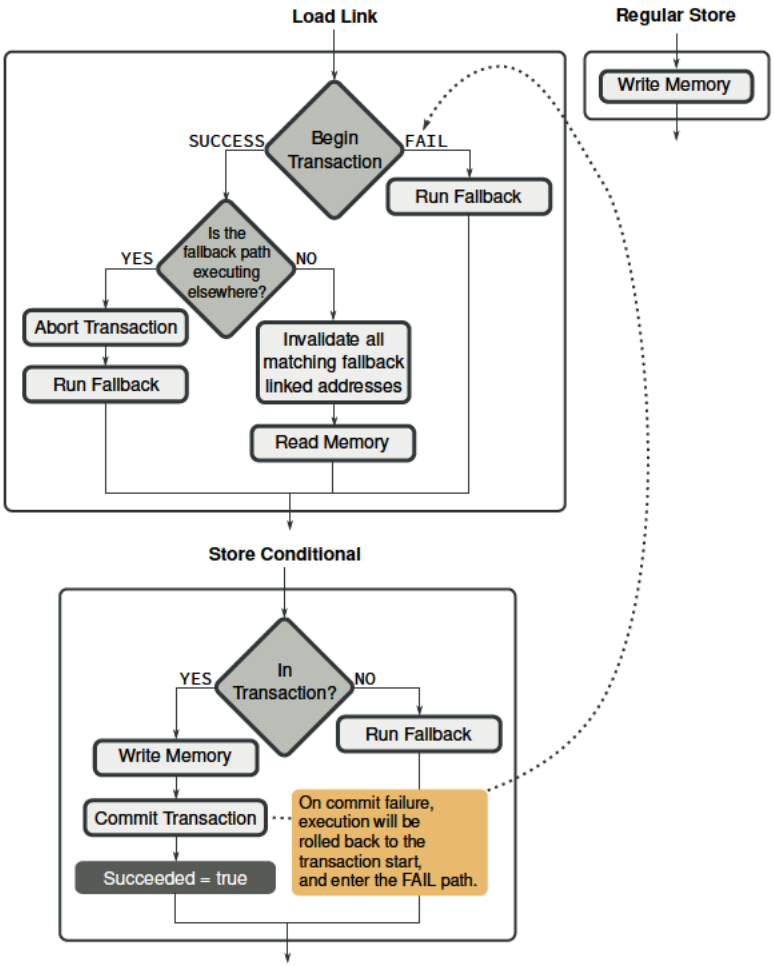


Figure 4.5: Scheme employing Hardware TM.

first case, since the LL does the invalidation while spinning on the lock in the cache entry, the Write must have been done before the invalidation completed, i.e. definitely before the underlying read of the LL. In the second case, such a fast-path Write is by definition not between the LL and SC. Of course, when the LL has invalidated the Writer's page table entry, the write can still go on the slow-path, but then we are back to the previous case.

4.2.4 Correct Scheme Using Hardware Transactional Memory

Hardware Transactional Memory (HTM) has been used for implementing memory consistency in cross-ISA emulation on multicore systems before [80]. We take inspiration from this approach, and develop a scheme for implementing LL/SC handling using similar HTM constructs.

HTM allows the execution of a number of memory accesses to be perceived as

atomic. This can be exploited for handling of LL/SC instructions by encapsulating the whole LL/SC sequence in a transaction. In particular, the emulation of a *load-link* begins a transaction (e.g. the `XBEGIN` Intel TSX instruction) and then reads the data. The emulation of a *store-conditional* writes the new data, and then commits the ongoing transaction (e.g. `XCOMMIT`).

Since all memory accesses between LL and SC instructions (inclusive) are part of a transaction, any concurrent transactions (i.e. another LL/SC pair) succeed only if there are no conflicts between any memory accesses. Furthermore, even memory accesses that are not part of a transaction will abort ongoing transactions in case of a conflict.

This behaviour is guaranteed by the strong transactional semantics supported by many modern architectures. As a result, this scheme can avoid any locking or linkage invalidation by relying on transactional semantics to resolve conflicts between concurrent LL/SC pairs, and to detect intervening regular stores.

Using HTM always requires a fallback mechanism to be in place, as HTM is not guaranteed to succeed. In the case of this scheme, the fallback mechanism is the Software-only scheme, described in Section 4.2.3.

The emulation of an SC instruction can check if it is executing within a transaction by e.g. using the Intel TSX `XTEST` instruction. For any core, both LL and SC instructions either take the transactional path or the fallback path. However, it is still possible for one core to execute a LL/SC sequence transactionally, while another core concurrently executes LL/SC using the fallback mechanism. Therefore, the transactional path has to correctly interact with the fallback path.

Since the transaction is perceived to be atomic, we analyse the interaction from the perspective of the fallback execution, assuming that transactional execution happens instantaneously. The transaction can happen while inside a critical section, during the emulation of a *store-conditional* in the fallback path.

To guarantee correct execution, we use a lock elision technique [89] that aborts (e.g. `XABORT`) the transaction if any cores are currently executing in the fallback path (i.e. within a critical section).

Another scenario is when the transaction happens while the fallback execution emulates instructions between LL and SC pairs. Here, the lock elision technique cannot be used, since there is no lock to elide. Instead, the fallback linkage is invalidated, causing the future fallback SC instruction to fail. This approach allows the transactional execution to continue, reducing the number of transaction aborts.

Benchmark	SC %	Description
`space_<x>`	3.84	atomically increment a random counter in an array of size `x`
`space_indep`	16.67	atomically increment a thread-private counter
`time_<x>_<y>`	variable	workload loop performs `y` instructions, `x` of which are inside LL/SC sequence
`stack`	2.32	alternate pushing and poping element in a lock-free stack [1]
`prgn`	16.67	generate random numbers by a lock-free random number generator

Table 4.4: Micro-benchmarks: Heavy use of LL/SC corresponds to a high percentage of executed SC instructions.

System	Dell® PowerEdge® R740xd		
Architecture	x86-64	*Model*	Intel® Xeon® Gold 6154
Cores/Threads	36/72	*Frequency*	3 GHz
L1 Cache	I$128kB/D$128kB	*L2 Cache*	1MB
L3 Cache	10 MB	*Memory*	512 GB

Table 4.5: ISS Host System

4.3 Evaluation

In addition to the application benchmarks, a custom suite of micro-benchmarks was constructed. These are designed to stress the use of LL/SC instructions. The benchmarks vary in the level of congestion in space and time. The micro-benchmarks are described in Table 4.4 and the details of the host machine used for experimentation are shown in Table 4.5.

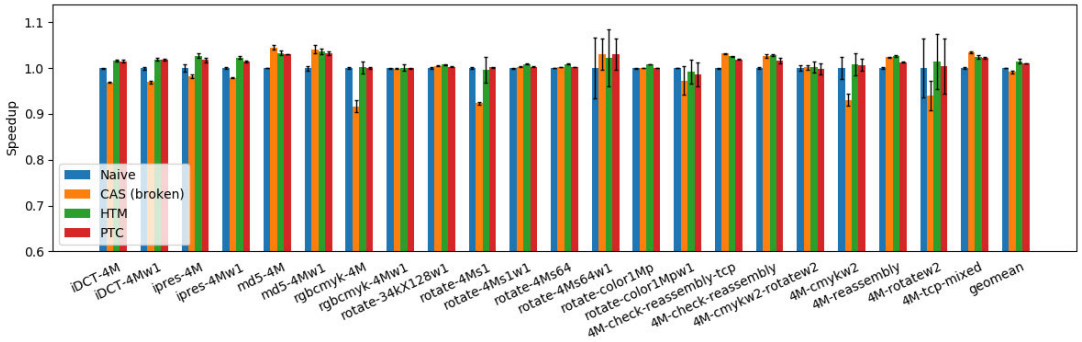


Figure 4.6: Results for EEMBC Multibench suite with 10 cores.

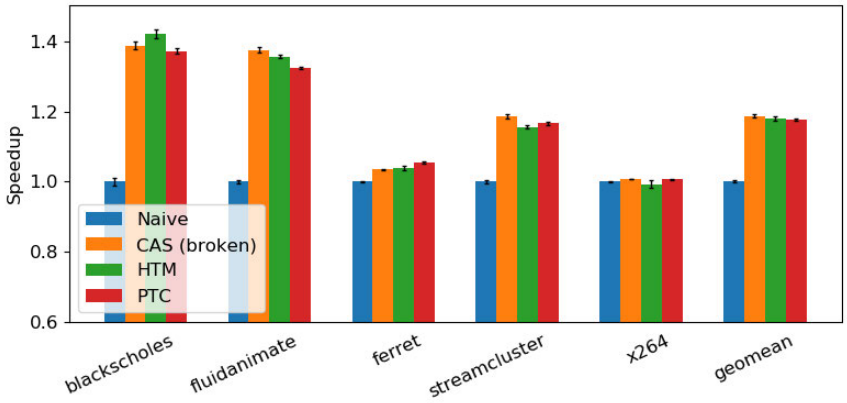


Figure 4.7: PARSEC suite with 10 cores.

4.3.1 Key Results: Application Benchmarks

In this section, we present results demonstrating that the different schemes do not adversely affect normal application performance. In these benchmarks, the number of LL/SC sequences are low, but since the LL/SC mechanism interacts with regular memory accesses, we show that our schemes do not incur a significant performance overhead during normal operation.

Figure 4.6 shows that the EEMBC Multibench suite does not exhibit much change in performance. For most benchmarks, all schemes result in the runtime performance falling within 5% relative to the Naïve scheme. This is due to the infrequent execution of the affected instructions (i.e. LL, SC, and regular store) by these benchmarks. For example, data-parallel benchmarks synchronise using LL/SC instructions only at the beginning and at the end of the execution. In this case, the workload kernel does not contain any synchronisation, and thus is not affected by the schemes.

Counter-intuitively, on average the CAS scheme performs slightly worse than HTM and PTC, but the difference is negligible, and we attribute this minor performance difference to indirect effects, such as the dynamic memory layout of the simulator.

Similar to the Multibench suite, Figure 4.7 shows that PARSEC benchmarks do not heavily use LL/SC instructions. However, the use of regular store instructions is much more frequent. This results in the PARSEC suite showing a significant performance improvement by using any other scheme compared to the Naïve scheme. Since the Naïve scheme incurs a synchronisation overhead for all regular stores, its performance degrades compared to the other schemes that do not synchronise independent regular stores. The performance of the other schemes is comparable, with the PTC scheme achieving a speedup of 1.18× over Naïve on average.

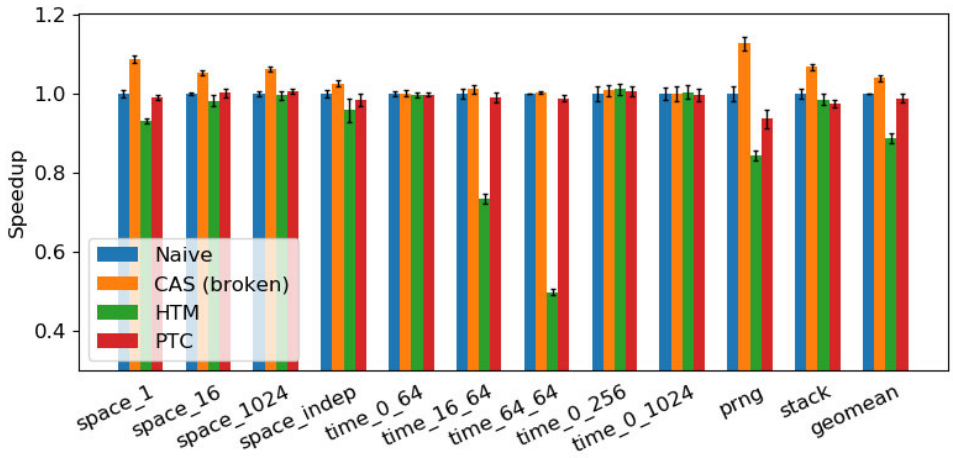


Figure 4.8: Micro-benchmarks with 1 core.

4.3.2 Drilling Down: Micro-Benchmarks

These benchmarks exhibit more frequent use of LL/SC instructions and emphasise performance implications of the different emulation schemes.

We evaluate the implementation overhead of each scheme by running a single-core version of the benchmarks. There is no congestion, as there are no concurrent cores. As a result, all SC instructions succeed and no LL/SC sequence is repeated. The single-core performance results are shown in Figure 4.8.

The CAS scheme results in the best performance with the average speed-up of 1.04× over the naïve scheme. This is due to implementation simplicity and lack of explicit synchronisation. However, this scheme does not preserve the LL/SC semantics. The performance of the PTC scheme is on par with the naïve scheme. This indicates that the additional cache invalidation and lookup present in the PTC scheme incur negligible overhead. The HTM scheme shows performance comparable to the other schemes only if the transactions are small, i.e. there are few instructions between LL and SC. In some benchmarks, there can be around 200 guest instructions between LL and SC, and we typically execute around 300 host instructions per guest instruction. This quickly leads to large transaction sizes on the host machine (e.g. in the order of 60,000 instructions), noticeably increasing the chances of an abort. This causes a significant drop in performance for benchmarks that perform substantial work inside an LL/SC sequence.

Next, we evaluate the performance of the proposed schemes in a multicore context with eight cores. Here, each core must synchronise and communicate the LL/SC linkage, resulting in different LL/SC success rates for different schemes. The efficiency of

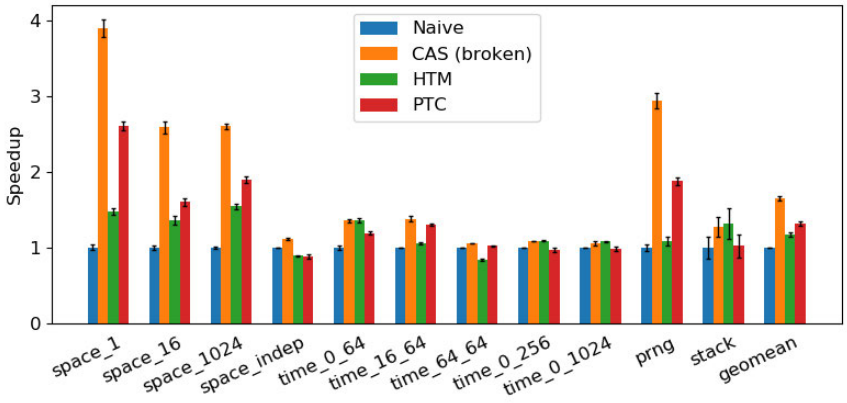


Figure 4.9: Micro-benchmarks using 8 cores.

linkage communication affects the overall performance beyond the single-core overhead. The results are shown in Figure 4.9.

In the case of high space congestion (`space_1`), PTC outperforms the naïve scheme by 2.6×. The broken CAS scheme results in the best performance. Fast LL/SC handling lowers the chances of interleaving concurrent LL/SC pairs, resulting in lower SC failure rates. Using the CAS scheme, each SC instruction fails three times on average before succeeding, while the PTC scheme results in six failures for each successful SC instruction.

Without any congestion, when all LL/SC pairs update independent memory locations (`space_indep`), the naïve scheme outperforms both PTC and HTM. Since the benchmark has few regular stores, the naïve scheme does not suffer the overhead of unnecessary synchronisation of all regular stores. But the naïve scheme emulates LL instructions much more efficiently, as it does not need to handle the PTC invalidation.

In the case of time congestion (`time_64_64`), large frequent LL/SC sections result in low HTM performance. The poor HTM performance in this case has also been observed for single-core execution. If the LL/SC sections are small (`time_0_64`), the HTM scheme outperforms the PTC scheme with a speed-up over Naïve of 1.36× for HTM and 1.19× for PTC. This is because HTM can rely on transactional semantics to avoid data races and thus, if most transactions succeed, it can avoid explicit synchronisation with other cores.

4.3.3 Lock Granularity

To implement critical sections in the proposed schemes, we use standard host-side locks. For correct LL/SC emulation, it is necessary to use the same lock for matching

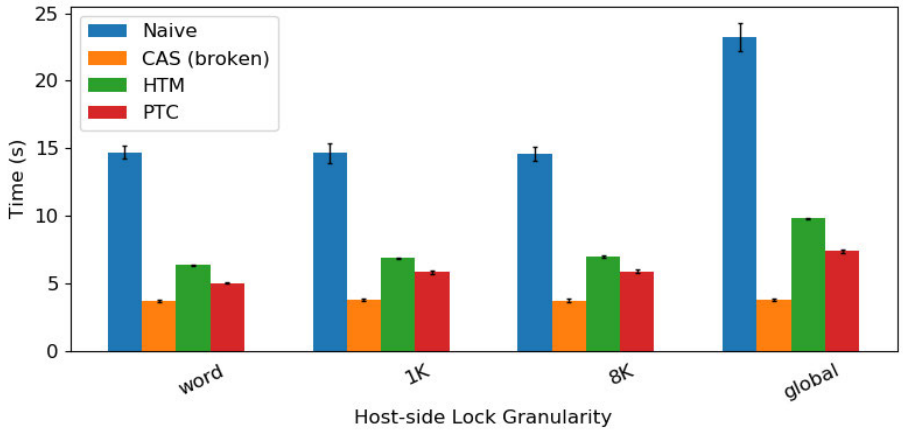


Figure 4.10: Micro-benchmark `space_1024` in 8-core configuration with varying host-side lock granularity.

target memory addresses of concurrent LL/SC instructions. Utilising multiple locks enables different LL/SC targets to proceed independently.

To achieve this, we create a hash-map of host locks. By masking the target of an LL/SC instruction, we obtain an index into the hash-map, which contains the lock to be used for the critical section. Manipulating the size of the mask allows us to control the granularity of the host-side lock. We have experimented with several mask sizes using the `space_1024` micro-benchmark, and the results are shown in Figure 4.10.

The experimental data shows that using a single global lock results in poor performance. Note, the CAS scheme does not use host-side locking at all, and therefore is not affected by the lock granularity. Although correct, using a single lock to facilitate critical sections even for independent LL/SC instructions is an obvious performance bottleneck. We find that using even slightly finer-grained locking improves performance, and results in the level of congestion being predominantly controlled by the guest-side (as opposed to host-side) lock granularity.

In practice, there can only be as many independent LL/SC targets in flight at the same time as is the number of cores being simulated. For example, in an 8-core configuration, any lock granularity that allows for eight random LL/SC targets to map to independent host locks can already achieve optimal performance. Therefore, in all other experiments, we use a word-sized mask, and a hash-map big enough to render collisions for (e.g. eight) random targets improbable.

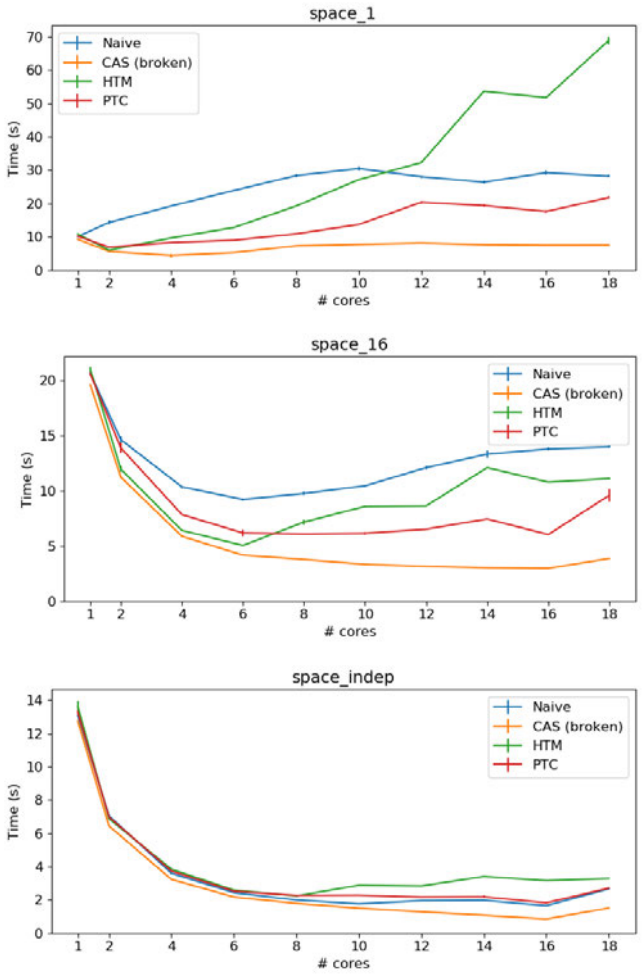


Figure 4.11: Scalability of space micro-benchmarks. High-congestion benchmark at the top.

4.3.4 Scalability

We evaluate the effect of varying the number of cores on the performance of the schemes. We vary the number of cores from one to eighteen in two core increments. The upper number of cores is selected such that all simulation threads, i.e. virtual cores, can be scheduled at the same time, on the same processor. This configuration results in the greatest level of parallelism without creating any thread scheduling conflicts.

The total work performed by micro-benchmarks is the same for all core configurations. The benchmarks split the same workload evenly between all available cores. We expect to see two types of ideal scaling characteristics. First, data parallel benchmarks with a low level of congestion (such as `space_indep`) should exhibit runtimes inversely proportional to the number of cores. Second, high-congestion benchmarks (such as `space_1`) need to sequentialise almost all execution. In this case, adding more cores should result in only marginal runtime improvements, i.e. the runtime is expected

to remain constant for all number of cores.

Without much congestion (`space_indep`), the schemes scale almost ideally. For a greater number of cores, the performance degrades most significantly for the HTM scheme. However, CAS exhibits little performance degradation even for high number of cores. This is because CAS is a long-established synchronisation mechanism that has been highly optimised by the hardware designers, whereas hardware transaction technology is relatively new and implementations haven't been optimised to the same degree.

High space-congestion benchmarks (`space_1`) exhibit near ideal scaling for CAS. HTM scales poorly and above ten cores becomes the worst performing scheme. For this many cores, very few transactions are able to complete without conflicts, resulting in almost all LL/SC instructions taking the fallback path. The PTC scheme scales significantly better than the Naïve scheme, especially at the medium number of cores (up to ten). The Naïve scheme shows no improvement even when moving from one-core to two-core execution. We attribute this scalability improvement to the optimisations introduced by PTC, as discussed in Section 4.2.3.2. In particular, using a lock to handle both LL and SC instructions forces the execution (in the case of Naïve) to sequentialise unnecessarily.

The time-congestion benchmarks show similar behaviour to the space-congestion benchmarks. CAS scales almost ideally for all levels of congestion. The PTC scheme scales similarly to CAS, and outperforms all other correct schemes. The HTM scheme shows poor and unstable performance. For a low core counts, the performance is sensitive to the hardware implementation, which affects the efficiency of conflict detection, and transaction failures. For higher numbers of cores, few LL/SC sequences succeed on the transactional path, resulting in most executions taking the fallback path.

4.4 Summary and Conclusions

We have shown that existing ISS systems implement an approximate version of *load-link/store-conditional* instructions that fails to fully capture their semantics. In particular, we showed how these implementations can cause bugs in real world applications, by causing the ABA problem to appear in e.g. lock-free data structures. We presented software-only and HTM assisted schemes that correctly implement LL/SC semantics in the commercial Synopsys DesignWare® ARC® nSIM ISS system. We evaluated our schemes and showed that we can maintain high simulation throughput for application

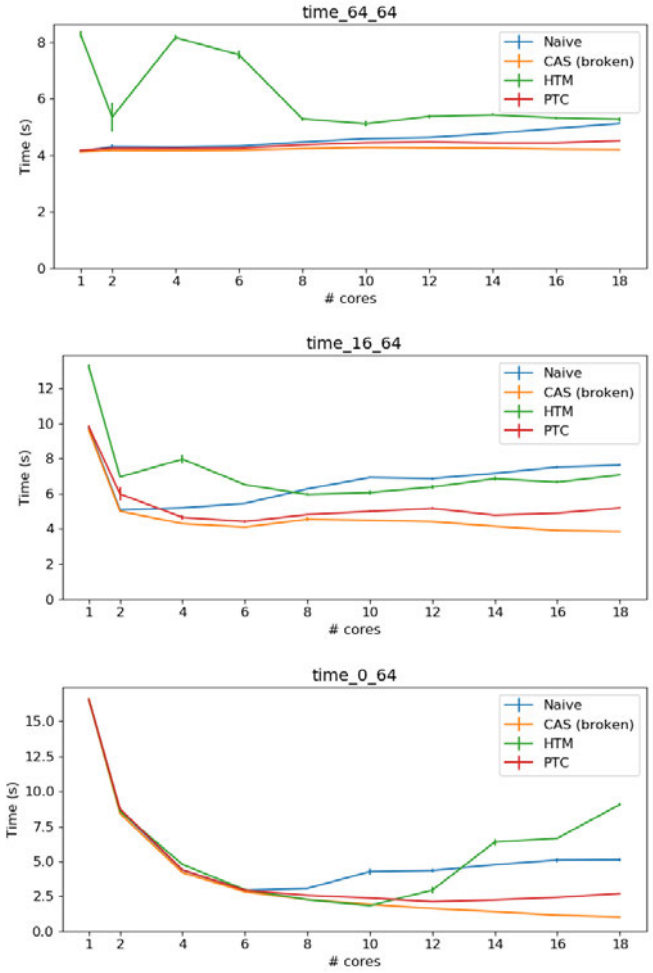


Figure 4.12: Scalability of time micro-benchmarks. High-congestion benchmark at the top.

benchmarks for provably correct LL/SC implementations.

Achieving correctness is a crucial first step in enabling efficient cross-architecture simulation. However, simulating multiple guest cores using multiple host threads must face further inefficiencies at the hypervisor level. For example, hypervisors data structures are naïvely duplicated for each guest core without regard for scalability to many core systems. The next chapter tackles the problem of memory duplication in a decode cache used to optimise interpretive emulation.

Chapter 5

Memory Efficient Decode Caching

There is a range of strategies that can be used to implement an ISS, the most basic being a simple interpreter [90], and the more complex ones involving a DBT [52]. In general, interpreter-based ISSs cannot achieve the performance of native execution. However, they still offer notable advantages that secure them a firm place in a developers' toolbox. Interpreters are relatively easy to implement, making rapid development or modifications of new or existing architectures feasible. More importantly, since interpretation happens inline with the execution of individual guest instructions, instrumentation, profiling, or debugging of the guest code becomes straightforward. Step-by-step interpreted execution enables interpreted ISSs to interface with e.g. detailed pipeline or cache models [69, 37]. In contrast, there is no tool that integrates DBT based multi- or many-core ISSs with accurate cycle modelling. Finally, interpreted ISSs boast a low memory footprint, and low execution startup cost as guest code execution begins immediately, making them ideal candidates for guest code regions that are not re-used enough to recuperate the initial cost of profiling and JIT compilation in a DBT.

Interpreted ISSs with a *fetch-decode-execute* simulation loop, are faced with a fundamental performance challenge. In particular, it is the repeated fetching and decoding of instructions that contribute substantially to overall simulator execution time [19]. To ameliorate this runtime cost, ISSs employ caching strategies to accelerate this stage by means of a *decode cache*[1]. Such caches are typically indexed by the guest virtual PC, so that pre-decoded instructions can be obtained quickly and repetitive costs for bit-level instruction decoding can be avoided after an initial warm-up phase [112]. Note,

[1] While conceptually similar, the fetch and decode caches in ISSs are separate entities from the micro-architectural fetch and decode caches of the simulated guest architecture, which may also be simulated for accurate cycle modelling.

decode cache is a hypervisor data structure typically implemented in software. Despite being often organised similarly to hardware caches (fixed size, set-associative), it is functionally more similar to a hashmap, as it is a software key-value data structure. Similarly to a hashmap, decode cache comprises *value* buckets accessed/indexed with a hash code computed from a *key*. A notable difference to a hashmap is that decode cache is statically allocated with fixed total size and the number of items in each bucket.

We observe that the commonly used PC based caching scheme for instructions and their decoding is far from optimal, both for single- and multicore ISSs. We start with two fundamental observations: (1) The number of distinct PC values in a typical program execution is far greater than the number of distinct instruction encodings, leading to inefficient cache utilisation, and (2) as the number of simulated guest cores increases, the memory footprint for decode caches used in many ISSs increases linearly, resulting in poor scalability.

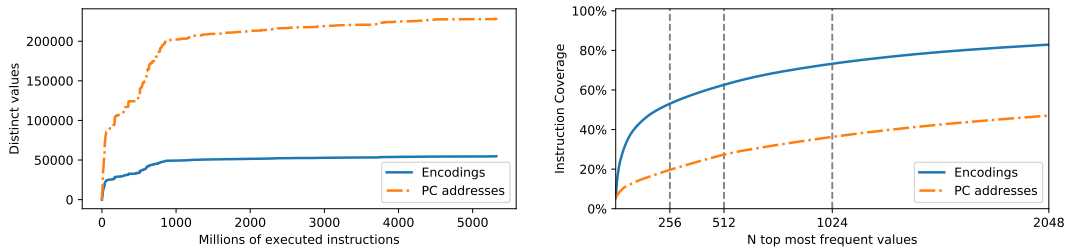
In this chapter, we revisit the instruction fetch and decode cache architecture in ISSs, and introduce a scheme that combines a number of novelties: (a) we use instruction encodings for cache indexing instead of the program counter for higher cache efficiency, (b) we introduce separate instruction fetch and decode caches instead of the traditional unified cache, and we (c) introduce a tiered cache architecture comprising private and global caches for use in simulation of multicore guest architectures for higher cache utilisation, greater scalability and increased efficiency, respectively.

5.1 Motivating Example

Consider the `gcc` benchmark from SPEC 2006 shown in Figure 5.1. The number of distinct PC values encountered during the execution (i.e. the *PC space*) is far greater than the number of distinct instruction encodings (i.e. the *encoding space*). This is because the same few instructions are used repeatedly across multiple PC locations[2]. In particular, out of more than 5 billion executed instructions, we count approximately 229,000 distinct PC values, while counting only 54,000 distinct instruction encodings. Therefore, the PC space is about $4\times$ greater than the encoding space.

Furthermore, we observe that a few encodings account for a large proportion of executed instructions (Figure 5.1b). In contrast, many more PC addresses are required

[2]We consider any two instructions that have different binary representations, not just different opcodes.



(a) Total number of distinct discovered encoding and PC values.

(b) Cumulative coverage of the most frequent encodings and PC values.

Figure 5.1: Encoding and PC distributions in `gcc` SPEC 2006 benchmark. The encoding space is just a fraction of the PC space. Furthermore, a few of the most frequent encodings account for a much larger number of executed instructions.

to account for the same proportion of the dynamic instruction workload.

These observations indicate that many instructions are duplicated in the PC space, and thus we can design a more efficient instruction storage scheme for the decode cache by indexing via instruction encoding instead of the PC. For example, using a fixed *PC*-indexed decode cache of the most frequent 256 PC values results in a hit rate of 19.62%, while using an *encoding*-indexed decode cache of the most frequent 256 instruction encodings results in a hit rate of 53.01%. In fact, the *encoding*-indexed cache can achieve a greater hit rate than a twice as large *PC*-indexed cache.

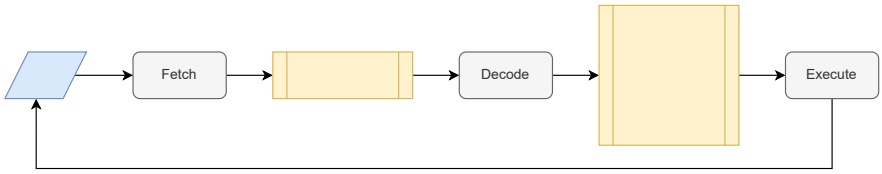


Figure 5.2: A standard interpreter loop. The PC is used to *fetch* the *raw instruction encoding*, which is then passed to the *decode* stage, which produces a *decoded instruction object*. This object is used by the *execute* stage to perform the actual operation of the instruction.

5.2 Background

Interpreter-based ISSs update the guest architectural state through software equivalent of the typical pipeline stages of the guest hardware (Figure 5.2). Given a guest PC, the simulator *fetches* the raw encoding of a single guest instruction from the simulated guest memory. Then, the simulator *decodes* the instruction data to produce a *decoded*

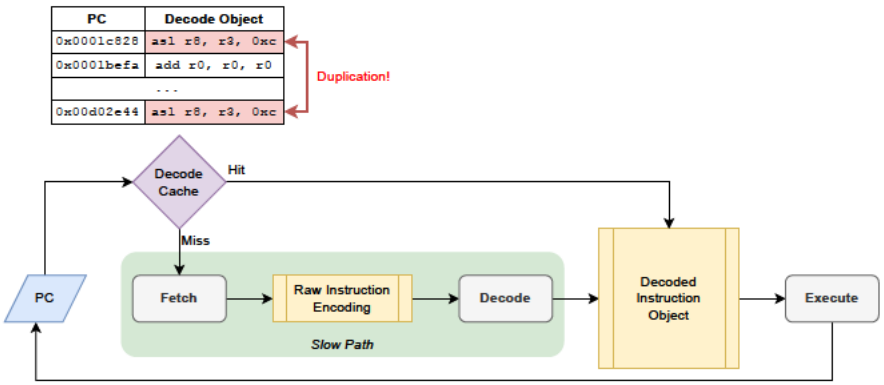


Figure 5.3: A standard interpreter loop with a decode cache. The PC is used to index the decode cache. A hit results in immediate access to the *decoded instruction object*. A miss follows the usual slow path, but with the resulting *decode object* inserted into the cache for later use. A PC-indexed cache results in duplication of *decode objects*.

instruction object (or *decode object*), which contains information such as the opcode of the instruction, source and destination registers, and any immediate values. The *execute* stage uses this representation to update the guest state in accordance with the semantics of the instruction being emulated.

The *fetch* and *decode* stages can take on average up to 66% of the instruction emulation time. In the *fetch* stage, the simulator consults the guest MMU to check executable permissions, perform the guest-virtual to guest-physical address translation, and traverse the simulated memory to actually retrieve the raw instruction encoding. In the *decode* stage, the simulator performs fine-grained bit manipulations to break down the compact and complex instruction encoding into an object amenable to execution, In contrast with real hardware instruction decoders, this stage cannot be easily parallelised in software. As a result, the overhead of these two stages is a major contributing factor to the low performance of interpreter-based simulators.

The most effective technique to reduce the overhead of the *fetch* and *decode* stages is by caching the *decode objects* (Figure 5.3). A *decode cache* enables the interpreter loop to retrieve the *decode object* using only a single lookup based on the current guest PC. If the corresponding entry is *not* found in the cache, the simulator has to perform the *fetch* and *decode* stages as usual, and insert the resultant *decode object* into the cache. With this strategy, the execution quickly reaches a steady state, with most instructions found in the cache.

However, the decode cache is not an unlimited resource. *Decode objects* are designed to enable high simulation performance of the *execute* stage, requiring many

individual fields for representing the instruction opcode, source and destination registers, various offsets and immediate values. As a result, *decode objects* can be quite large (150–200 bytes in some simulators), making the decode cache a memory intensive data structure. To fit within a reasonable memory budget, a typical decode cache contains around 1024 entries, which in many cases is sufficient to hold the most frequently executed instructions.

5.3 Methodology

There exists significant duplication of *decode objects* in the decode cache of a standard interpreter-based ISS. In particular, many PC-indexed cache entries hold identical *decode objects*, i.e. instructions with the same encoding but different PC locations. The **key idea** is to remove this duplication by re-organising the *decode cache* to be indexed by the raw instruction encoding, rather than PC values.

While a decode cache that is indexed by encoding eliminates *decode object* duplication, the cache can be queried only after a *fetch* operation has been performed, and the raw instruction encoding is available. As we described previously, the *fetch* stage also contributes to a significant portion of the interpreter execution pipeline, therefore we maintain an additional PC-indexed cache (the *fetch cache*) that caches the result of the *fetch* operation, and other PC-specific information that is not encoded in the instruction's binary representation. Like the original decode cache, each entry implicitly caches a successful memory permission check for executable code. Since the corresponding *fetch cache* entry is relatively small, we tolerate cache entry duplication for the benefit of improved overall performance.

In the following sections, a range of novel caching schemes that tackle the issue of memory utilisation and performance, in the context of single-core and multicore simulation modes, is to be presented.

5.3.1 Single-core Scenario

This section explores two novel caching schemes that have different organisations of their respective *fetch* and *decode* caches. In both schemes, an individual *fetch* cache is present for user and kernel PC space. An encoding-indexed *decode* cache is shared for both user and kernel execution mode.

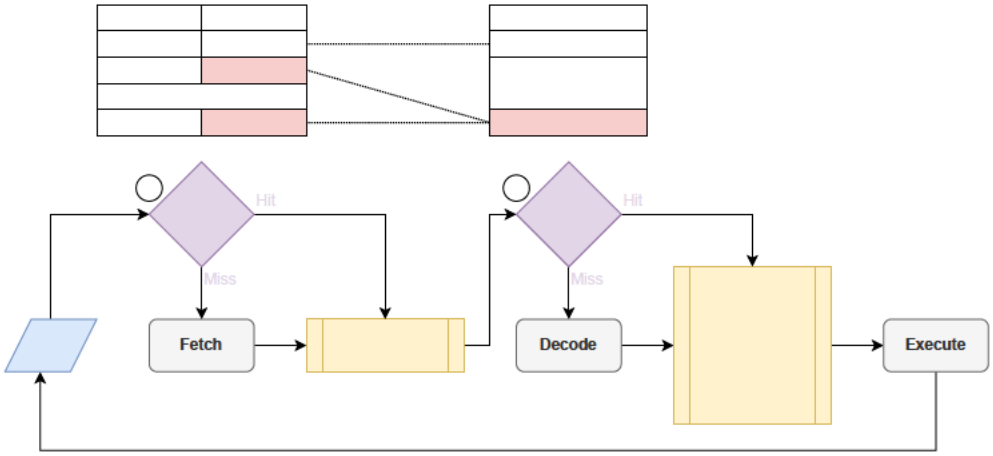


Figure 5.4: The *Dual-step* scheme. At ①, the PC is used to index a *fetch* cache that stores raw instruction encodings, which at ② are then used to index a *decode* cache that stores *decoded instruction objects*. Whilst the *fetch* cache can contain duplicate entries, they would each correspond to a unique entry in the *decode* cache.

5.3.1.1 Dual-step Scheme

The *Dual-step* scheme allocates two independent caches: (1) a PC-indexed *fetch cache* for storing raw instructions encodings, and (2) an encoding-indexed *decode cache* for storing *decoded instruction objects*. Figure 5.4 shows the flow of operations in this scheme. First, the PC is used to index the *fetch cache*, and acquire the raw instruction encoding. If the cache misses, a normal *fetch* operation is performed, with the appropriate entry being inserted into the cache for next time. Then, the *raw instruction encoding* is used to index the *decode cache*. A miss in this cache causes the *decode* operation to be performed, and the new *decoded instruction object* to be inserted. It is important to note that in the decode cache, there are no duplicate entries.

Unfortunately, however, indexing by instruction encoding can result in worse performance than indexing by PC due to various effects of the host micro-architecture. Whilst PC values in an instruction stream are mostly sequential, the values of the corresponding instruction encoding are pseudo-random. Therefore, accesses into an encoding-indexed cache do not benefit from spatial locality, and suffer from worse host data cache performance.

Additionally, although our scheme improves memory utilisation, two cache lookups incur a non-negligible computational overhead. Furthermore, the index computation for a PC-indexed cache is trivial; it is simply a mask applied to the PC value. However, the index computation for an encoding-indexed cache is not so straightforward.

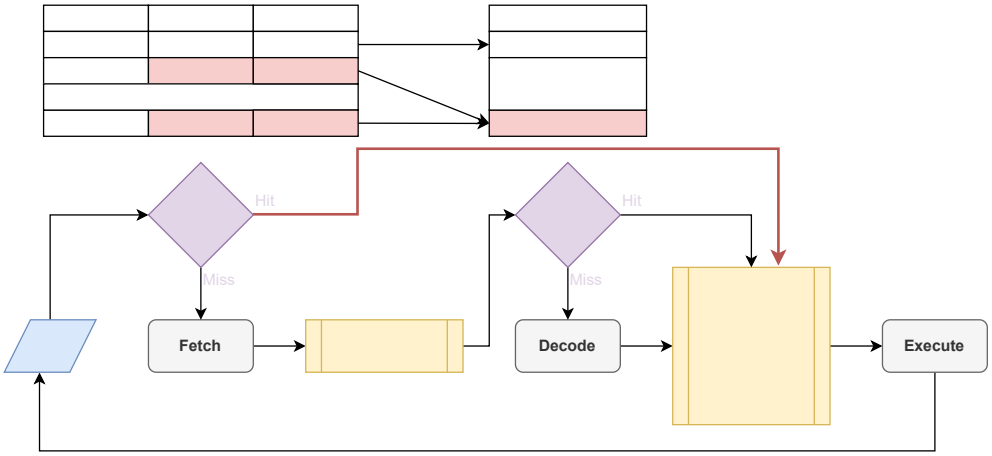


Figure 5.5: The *Indirect* scheme. Similar to the *Dual-step* scheme, a *fetch* cache and a *decode* cache are employed, but instead of performing an additional lookup in the *decode* cache after a *fetch* cache hit, an index field is used as a pointer offset to bypass the *decode* cache lookup completely, and directly access the *decode object*.

This is because observed PCs naturally avoid cache line conflicts due to the sequential nature of instruction execution, however instruction encodings do not have this inherent property, so we need to choose an *indexing function* that optimises cache line selection. The choice of this function can affect the rate of conflicts (i.e. two different encodings mapping into the same cache entry) in the cache, so an indexing function needs to be selected that balances efficient placement with computational cost. The indexing function will be inherently guest-ISA specific, as it depends on knowledge about the underlying structure of the raw instruction encoding.

5.3.1.2 Indirect Scheme

To mitigate the overheads of the *Dual-step* scheme, the *Indirect* scheme introduces a level of indirection. This is achieved by storing an index value next to the raw encoding in each *fetch* cache entry. If the lookup in the *fetch* cache hits, the index value is used to directly access the corresponding *decoded instruction object* without the need to perform a lookup in the *decode* cache. Figure 5.5 shows how the flow of operations are similar to the *Dual-step* scheme, except a hit in the *fetch* cache results in a pointer directly to the valid *decoded instruction object*.

Since multiple *fetch* entries can point to the same *decode* entry, we have to introduce an invalidation mechanism in case the corresponding *decode* entry is evicted. This effectively defines an inclusive caching policy, where the *decode* cache is inclusive of

the *fetch* cache.

We opted for a lazy invalidation approach, where a validity check in the *decode* cache is performed after hitting in the *fetch* cache. This is effectively a tag-check, where we compare the raw encoding from the *fetch* entry, with the raw encoding of the *decode* entry (which is stored as part of the *decoded instruction object*). The alternative is an eager invalidation approach, which would be triggered when a decode object is evicted, however this would require a full scan of the *fetch* cache, to identify entries that need to be invalidated.

As a result of linking the two caches using the index pointer, there are always some additional *fetch* cache misses compared to the *Dual-step* scheme with the same cache size configuration. These misses can be classified as *invalidation* misses. Interestingly, although the *Indirect* scheme results in worse cache performance than the *Dual-step* scheme, the lower computational overhead achieved by avoiding an additional cache lookup results in better runtime performance.

5.3.2 Multicore Scenario

Another level of *decode object* duplication is observed in multicore simulation. This duplication is a result of the *original* scheme instantiating a private *decode* cache for each simulated core. In the case of data parallel guest applications, simulated cores would execute the same code and thus contain the same instructions in their own private *decode* caches. In the case of task parallel guest applications, even though the simulated cores might not execute the same code, they are still likely to use similar, if not identical, instructions (in terms of raw instruction encodings). Therefore, sharing *decode objects* among multiple simulated cores can reduce object duplication to an even greater extent than the novel single-core schemes alone.

Furthermore, the improvements in memory utilisation increase as more cores are simulated. This is because the more cores are simulated, the more likely they are to use the same instructions, thus sharing the *decode objects* avoids more and more duplication.

In this section, we present three novel caching schemes suitable for multicore simulation that remove duplication of *decode objects* between simulated cores. This is achieved by utilising a shared memory pool of *decoded instruction objects*, with additional low-overhead synchronisation mechanisms to prevent data races.

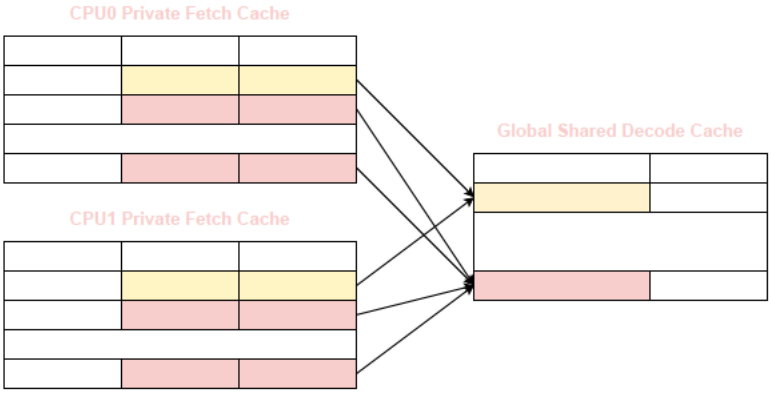


Figure 5.6: Cache organisation for the *Exclusive* scheme. Each simulated core has its own private *fetch* cache, which references entries in a global shared *decode* cache.

5.3.2.1 Exclusive Scheme

The *Exclusive* scheme builds on top of the *Indirect* scheme, to enable sharing *decode objects* among cores. In fact, the interpreter loop is identical to Figure 5.5, and it is the organization of the caches that is different. In this scheme, each simulated core gets a private *fetch* cache, and there is one global shared *decode* cache. Figure 5.6 demonstrates how each private *fetch* entry's index points into the shared *decode* cache.

However, sharing *decode objects* between multiple cores raises the possibility of concurrent modification, specifically when one core replaces a *decode* entry in the shared cache while another core has a pointer to the same entry in its private *fetch* cache. Since the replacement can happen concurrently, the invalidation mechanism from the *Indirect* scheme is not sufficient on its own to guarantee correct execution. To protect against concurrent modifications, a lock must be acquired for each *decode* entry before the corresponding *decode object* is used in the *execute* stage.

We use a spinlock over a 1-byte lock variable to guarantee mutual exclusion. This reduces the memory footprint of the synchronisation mechanism, as well as achieving the best performance relative to other synchronisation options (e.g. a pthread mutex takes up 40 bytes of memory, and in our experiments was always slower than a spinlock, with up to 40% longer execution times). The outstanding performance of a spinlock is due to the small size of the critical section, i.e. the execution time of the *execute* stage is always shorter than the cost of the two context switches, required to service a sleeping lock.

Unfortunately, other than protecting against concurrent modifications, the use of a lock also forces mutual exclusion in the case of executing the same instruction. Se-

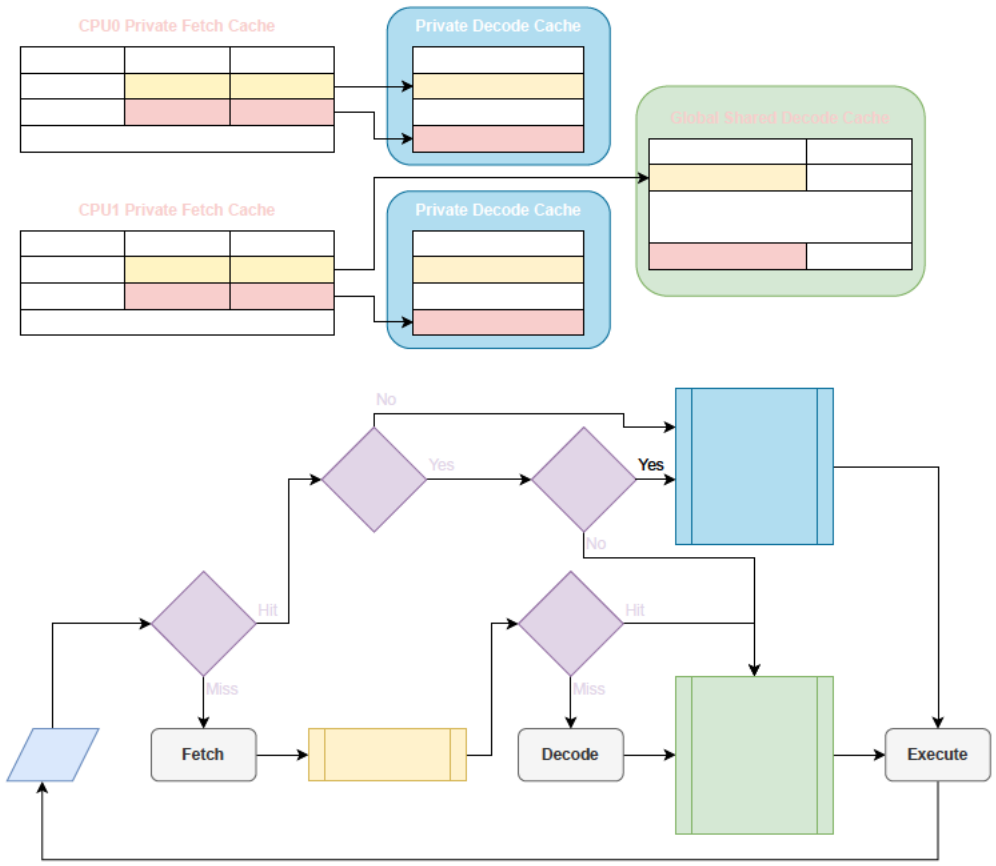


Figure 5.7: Cache organisation for the *Mixed* scheme. Private *fetch* entries reference either private or shared *decode* entries. Shared *decode* pointers are indicated by a bit tag in the index field of each *fetch* entry.

quentialising the execution of the same instruction can become a bottleneck, especially for data-parallel applications with small workload kernels. For example, a 16-core simulation with a workload kernel of just 16 distinct raw encodings is guaranteed to have at least two cores executing the same instruction encoding at any one time, and therefore needing to sequentialise, resulting in an overall slowdown of the application.

5.3.2.2 Mixed Scheme

To prevent multiple cores from having to sequentialise their execution in case of using the same raw instruction encodings, we introduce a mixed private and shared *decode* cache, as shown in Figure 5.7. This *Mixed* scheme builds on top of the *Exclusive* scheme by adding a small core-private *decode* cache. All *decode objects* first start in the shared cache. Then, if a core tries to acquire a lock during future *fetch* cache hits, and the lock variable is already acquired, the *decode object* is copied into the

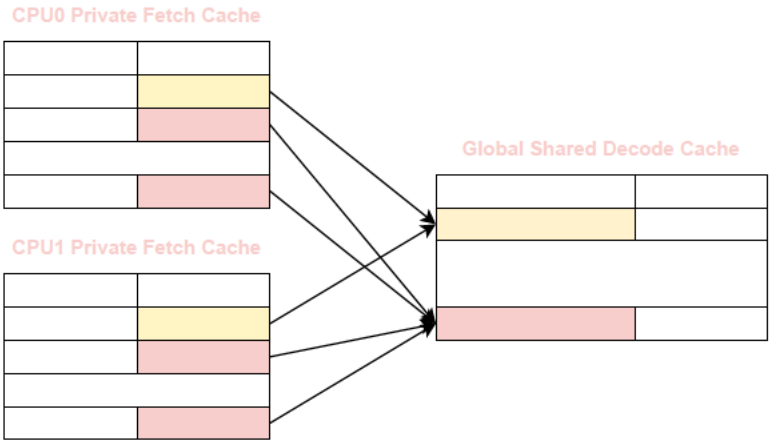


Figure 5.8: Cache organisation for the *Parallel* scheme. Core-private fetch entries only need an index pointer, as no *decode* cache invalidations are allowed.

core's private *decode* cache. By creating a private copy of the *decode object*, the core no longer needs to synchronise with others, as concurrent modification of the private *decode* entries does not happen.

To support this, the corresponding index in the *fetch* cache entry has to be patched to indicate that the corresponding *decode object* resides in the private cache. A shared *decode object* is indicated by setting the most significant bit of the 2 byte index field. We ensure the cache sizes are never big enough to require this bit, and thus we can use it for index/pointer tagging. After the entry index has been patched, all future *fetch* cache hits can use the private *decode object* without any synchronisation. Note, replacements in the private *decode* caches are still possible, therefore a validity check of the entry (using the *raw instruction encoding*) is still required.

5.3.2.3 Parallel Scheme

Fully concurrent access to shared *decode objects*, with a negligible amount of synchronisation, can be achieved if the cache is designed such that *decode objects* are never replaced during instruction emulation - concurrently or otherwise. Under this invariant, the validity check is no longer required, so we can remove the raw instruction encoding from the *fetch* cache entry. Furthermore, the lock variable in the *decode* cache entry can be removed, as no synchronisation is required. This arrangement, which allows for all simulated cores to execute with the maximum speed possible, is shown in Figure 5.8.

The only modification to the shared *decode* cache is allowed in the case of a com-

pulsory miss. If any core tries to execute an instruction encoding that is not already present in the *decode* cache, a global lock is acquired, and the corresponding cache entry is added to the *decode* cache. Note, this only applies for compulsory misses to the *global shared decode cache*. If a core experiences a compulsory miss in its *private fetch cache*, but the corresponding *decode object* is found in the *shared decode* cache, no synchronisation is necessary. This is because searching in the *decode* cache can be done without any data races, even in the case of a concurrent addition of a new *decode* cache entry.

Since the *decode* cache used in this scheme is an append-only data structure with a statically known maximum size, we organise it as an array-allocated AVL binary search tree. This results in logarithmic look-up times, with respect to the size of the cache.

In case the *decode* cache becomes full, we discard the entire content of the cache and start filling it anew. This approach, whilst quite brutal, is very similar to QEMU's handling of translated code caches [13]. To guarantee the aforementioned invariant, all cores must be stopped before the *decode* cache invalidation is performed. Furthermore, the *fetch* caches of all cores are also invalidated to eliminate any stale pointers. After the reset operation, cores experience compulsory misses that result from filling up the *decode* cache again.

The major drawback of this scheme is the high cost of the *Stop-the-World* cache reset. This can become a significant portion of the simulation if the memory allocated to the shared *decode* cache is small relative to the application workload size. However, as more and more cores are simulated, more and more memory is saved by having a shared *decode* cache, allowing for a larger *decode* cache size, resulting in less frequent resets.

5.4 Evaluation

To evaluate the effectiveness of our novel caching schemes, we extended the commercial Synopsys ARC© nSIM ISS simulator, and run our experiments in two primary simulator configurations: *single-core*, and *multicore*. All experiments are run on the host machine described in Table 5.1.

CPU Model	Intel Xeon E5-2640 v4
Cores	10
Frequency	2.4 GHz
L1 cache (private)	32 KB
L2 cache (private)	256 KB
L3 cache (shared)	25600 KB
Operating System	CentOS 6.6

Table 5.1: Evaluation machine description

5.4.1 Single-core Scenario

To evaluate the benefits of an encoding indexed decode cache, we compare cache performance and runtime performance of the *Dual-step* and *Indirect* schemes against the *Original* caching scheme (i.e. the existing caching scheme in the simulator).

5.4.1.1 Experimental Setup

To achieve a fair comparison, single-core schemes use the same total amount of memory to hold their various data structures. We choose a memory budget based on the memory consumption of the *Original* scheme, using 256 entries each for kernel mode and user mode (512 entries in total). This gives a total memory budget of 94 KB, based on a *decode object* size of 184 bytes.

The small size of the fetch entry in the *Dual-step* and *Indirect* schemes allows for doubling the number of entries at the fetch stage to a total of 1024 entries. The remaining memory budget was allocated to decode entries that were shared for both kernel and user space. The exact memory allocations are given in Table 5.2.

Note, each scheme could employ an arbitrary memory allocation strategy. Our choice was motivated by keeping the number of fetch entries a power of two for faster lookup operations. Furthermore, by doubling the number of fetch entries, and sharing the decode entries between kernel and user space, the new schemes can cover a higher percentage of the PC-space as well as the encoding-space.

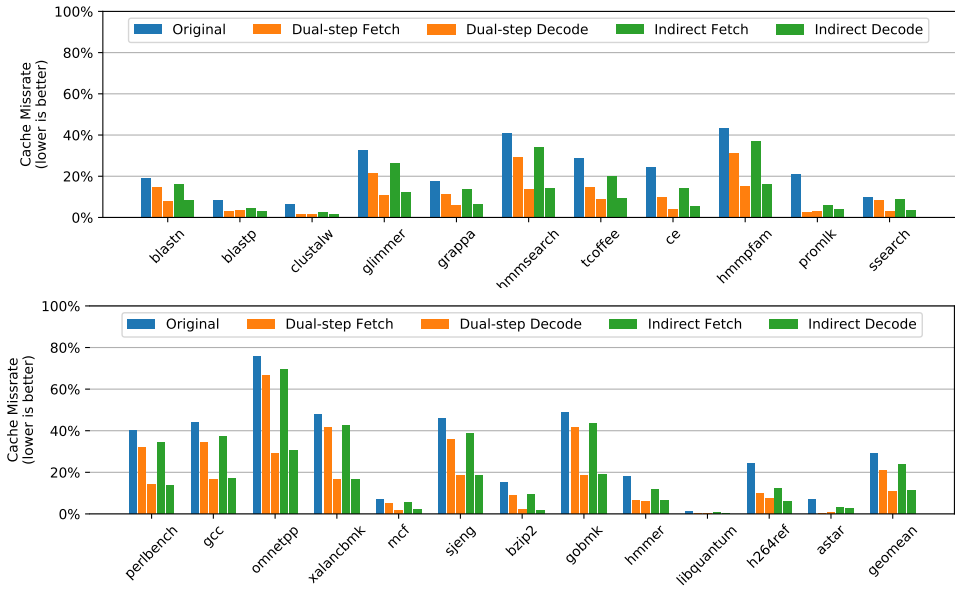
We compiled benchmarks from the SPEC CPU 2006 Integer [48], and the Bioperf [9] benchmark suites for bare-metal single-core execution.

5.4.1.2 Key Results

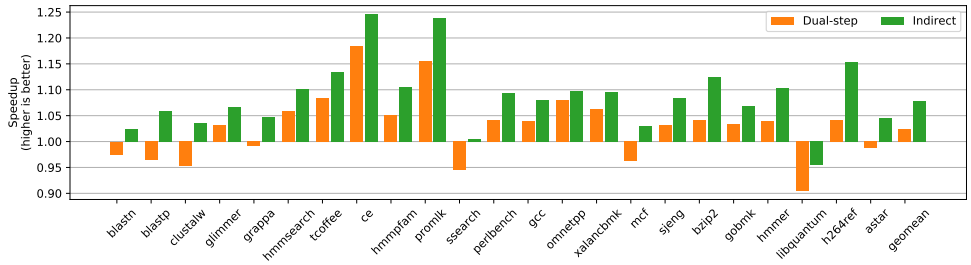
The memory efficiency gained by using our novel caching schemes result in improved cache performance (Figure 5.9a). Both the *Dual-step* and *Indirect* schemes result in

Scheme	Original	Dual-step	Indirect
# of fetch entries	512	1024	1024
Fetch entry size	188 B	12 B	14 B
Total fetch memory	94 KB	12 KB	14 KB
# of decode entries	-	456	445
Decode entry size	-	184 B	184 B
Total decode memory	-	82 KB	80 KB

Table 5.2: Allocation of the 94 KB memory budget.



(a) Cache performance of different schemes. *Original* scheme has a unified cache; a miss necessitates a fetch and a decode step. *Dual-step* and *Indirect* schemes have a fetch and a decode cache each; a miss necessitates only the corresponding step to be performed.



(b) Speedup of *Dual-step* and *Indirect* schemes relative to the *Original* scheme.

Figure 5.9: Single-core schemes evaluation.

significantly better cache performance than the *Original* scheme, whilst using the same amount of memory. In the case of the *Dual-step* scheme, the number of instructions for which a fetch operation has to be performed (i.e. the fetch cache miss rate) is reduced by a factor of 1.4× on average relative to the *Original* scheme. Moreover, the number of instructions requiring a decode operation (i.e. the decode cache miss rate) is reduced by a factor of 2.7×. In the case of the *Indirect* scheme, the fetch cache miss rate is reduced by a factor of 1.23×, and the decode cache miss rate is reduced by a factor of 2.59× on average.

The *Indirect* scheme results in a slightly higher miss rate than the *Dual-step* scheme for both the fetch and decode caches. The reduction in fetch cache performance is due to the invalidation mechanism required by the *Indirect* scheme. Since the index pointer in the fetch cache directly links with the decode cache entries, any decode entry replacement invalidates potentially several fetch cache entries, which account for the additional invalidation misses in the fetch cache of the *Indirect* scheme.

The reduction in decode cache performance is mainly due to the smaller size of the cache, compared to the decode cache of the *Dual-step* scheme (Table 5.2).

Figure 5.9b depicts speed-up relative to the *Original* scheme. The *Dual-step* scheme performs similarly to the *Original* scheme, with an average speedup of 1.02×. Although some benchmarks benefit from sharing *decode objects*—with up to a 1.18× speedup (`ce` in BioPerf)—the computational overhead of two cache lookups in the *Dual-step* scheme reduces execution performance significantly.

The overhead of the *Indirect* scheme on the hit path is significantly smaller (only a validity check), resulting in much better performance. Using the same amount of memory, the *Indirect* scheme can speed up execution by up to 1.25×, and by 1.08× on average.

5.4.1.3 Micro-benchmarks

In order to evaluate the best and worst case performance of the new schemes under different load conditions, we developed a suite of micro-benchmarks to exemplify the two extremes of the PC *vs.* Encoding space relation.

Same-instruction benchmarks execute *identical* instruction encodings at *multiple* PC locations, i.e. $|PC| \gg |ENC| \approx 1$. On the other hand, *Unique-instruction* benchmarks execute *different* instruction encoding at *multiple* PC locations, i.e. $|PC| \approx |ENC| \gg 1$. The program template used for micro-benchmark construction is given in Figure 5.10.

```
1   mov r0, ITER_COUNT
2 loop_start:
3   add r1,r2,r3
4   add r1,r2,r3
5   add r1,r2,r3
6   ...
7   add r1,r2,r3
8 loop_check:
9   sub r0,r0,#1
10  bnz r0,loop_start
```

workload size

(a) Same-instruction benchmark template.

```
1   mov r0, ITER_COUNT
2 loop_start:
3   add r1,r1,r1
4   add r1,r1,r2
5   add r1,r1,r3
6   ...
7   add r9,r9,r9
8 loop_check:
9   sub r0,r0,#1
10  bnz r0,loop_start
```

workload size

(b) Unique-instruction benchmark template.

Figure 5.10: Implementation of micro-benchmarks.

Scheme	***Same*-instruction**			***Unique*-instruction**		
	S	**M**	**L**	**S**	**M**	**L**
Original	0%	100%	100%	0%	100%	100%
DS fetch	0%	0%	100%	0%	0%	100%
DS decode	0%	0%	0%	0%	29.8%	100%
IN fetch	0%	0%	100%	0%	38.21%	100%
IN decode	0%	0%	0%	0%	38.21%	100%

Table 5.3: Micro-benchmark cache miss rate at various workload sizes.

Each type of benchmark is varied in the size of the PC space, such that guest instructions exhibit varied miss rates in the corresponding caches. The novel schemes are evaluated across *small* (always hit), *medium*, and *large* (always miss) benchmark size configurations. The miss rates for different caching schemes using each workload size configuration are given in Table 5.3. Note, the presented miss rates correspond to the cache size configuration shown in Table 5.2.

The best-case scenario is represented by the *same*-instruction micro-benchmarks. The *small* workload size, which exercises the hit path of all schemes' caches, exhibits similar performance to the *Original* scheme. The *medium* workload size of the *same*-instruction micro-benchmark only fits in the increased caches of the *Dual-step* and *Indirect* schemes. This benchmark demonstrates the maximum benefit of using the novel schemes. This is achieved by removing *decode object* duplication, resulting in a cache hit in both the fetch and decode caches for every instruction (except a negligible number of compulsory misses). The *large* workload size *same*-instruction micro-

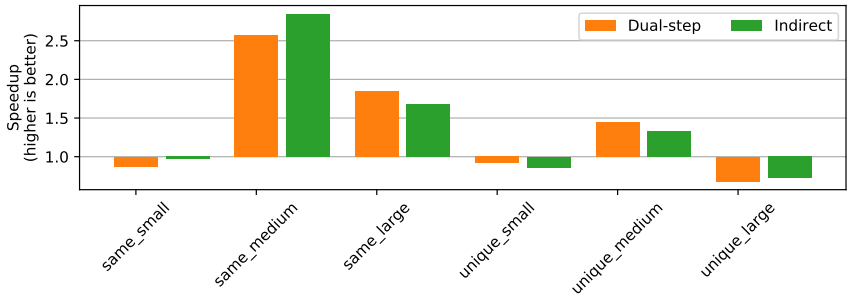


Figure 5.11: Speed-up of the *Dual-step* and *Indirect* schemes relative to the *Original* scheme in the best-case and worst-case micro-benchmarks.

benchmark only manages to cache the decode step while the fetch cache entries are continually being replaced, due to the PC-space being too large relative to the fetch cache size.

The worst-case scenario is evaluated using the *unique*-instruction micro-benchmarks. In this context, cache indexing by instruction encoding does not improve memory efficiency as there is no instruction duplication even in the PC space. Due to the overheads of the increase in complexity of the novel schemes, these micro-benchmarks often exhibit slowdown relative to the *Original* scheme. However, according to our analysis, real applications' encoding space is on average only 30% of the PC space, making them more similar to the *same*-instruction benchmarks.

5.4.2 Multicore Scenario

The previously described single-core schemes can also be used in a multicore simulation by instantiating the corresponding decode cache hierarchy for each simulated core. In this context, all *decode objects* are private to their virtual cores and there is no interaction or sharing amongst the caches. Our novel multicore schemes introduce sharing *decode objects* between cores by creating a globally shared decode cache. In this section, we evaluate the effectiveness of decode cache sharing, along with performance implications that arise due to the required synchronisation. We compare four schemes: *Indirect*, *Exclusive*, *Mixed*, and *Parallel*.

5.4.2.1 Experimental Setup

Multicore evaluation is facilitated by a collection of benchmarks from the SPLASH-2 suite [111]. These benchmarks can be run as multithreaded guest applications; the

number of guest threads can be configured in powers of two. Multithreaded bare-metal execution is achieved by linking with a custom implementation of the `pthread` library. This implementation assigns each guest application thread to a virtual core, which is simulated as a host thread.

In multicore simulation, the *Original* and *Indirect* schemes duplicate their *decode* caches for each simulated core, resulting in memory consumption increasing linearly with the number of simulated cores. However, the *Exclusive*, *Mixed*, and *Parallel* schemes duplicate only the smaller *fetch* cache. Decode entries can be shared in a global memory pool, resulting in improved scalability for many-core simulation—the more cores that participate in cache memory sharing, the more memory is available to increase the size of both the fetch and decode caches. Alternatively, if the cache sizes are fixed, the multicore schemes will require less memory than the *Original* scheme as more cores are simulated.

The multicore schemes were evaluated using two configurations: 8-core, and 128-core. In the 8-core configuration, the multicore schemes match the memory budget of the *Original* scheme. By sharing *decode objects* between 8 cores, the memory budget allowed multicore schemes to allocate 4× more *fetch* entries, and more than 4× more *decode* entries before the memory budget was exhausted. This increased cache size significantly improved cache performance, resulting in most binaries in our evaluation achieving near zero miss rates. As increasing cache sizes further would not significantly improve performance, the 128-core configuration is evaluated using the same cache size configuration as the 8-core simulation. As a result, the multicore schemes require significantly less memory than the *Original* scheme, whilst still achieving significantly better cache performance. The exact memory allocation is given in Table 5.4.

Furthermore, 128-core simulation evaluates the effect of host thread contention. Since the host machine used for experiments provides only 10 physical cores, simulating 128 virtual cores results in up to 13 host threads sharing a physical core.

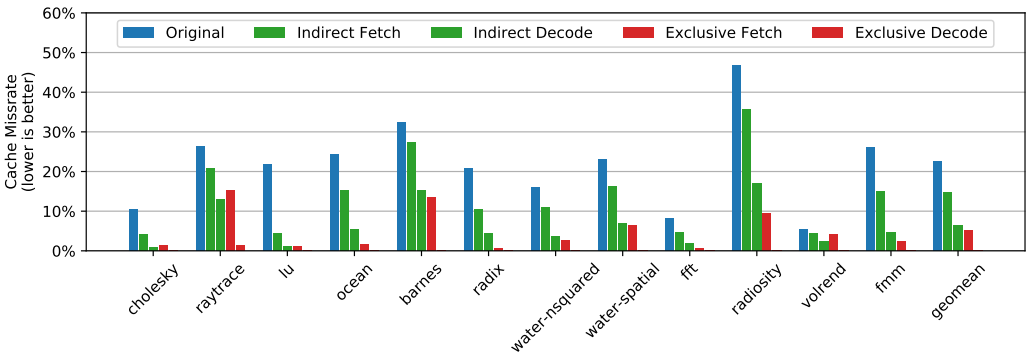
5.4.2.2 Key Results

The novel multicore schemes vastly improve cache performance (Figure 5.12a), reaching average miss rates of 5.07% and 0.15% for fetch and decode caches respectively. In comparison, the *Indirect* scheme resulted in average miss rates of 14.66% and 6.51% for the fetch and decode caches, whilst the *Original* scheme's miss rate of its unified cache was 22.62%.

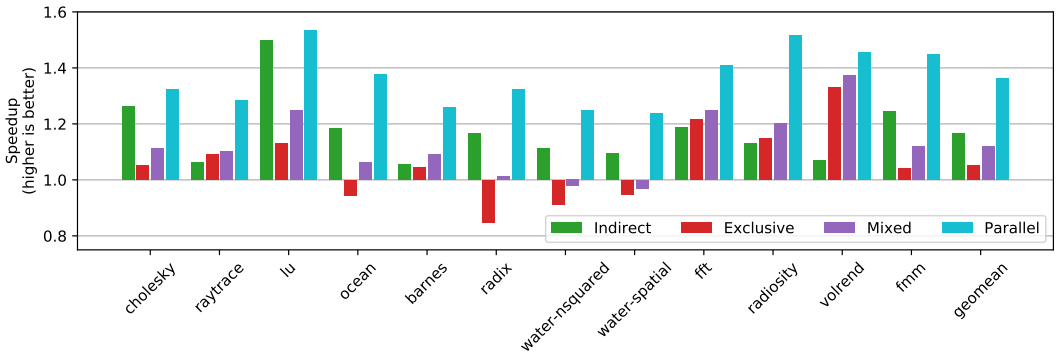
In terms of runtime performance, the *Exclusive* and *Mixed* schemes result in speedups

Scheme	Orig.	Indir.	Excl.	Mixed	Par.
# of fetch entries	512	1024	2048	2048	2048
Fetch entry size	188 B	14 B	14 B	14 B	10 B
# of private entries	-	445	-	32	-
Private entry size	-	184 B	-	184 B	-
# of shared entries	-	-	2900	2650	3150
Shared entry size	-	-	185 B	185 B	191 B
Total (8 cores)	752 KB	752 KB	748 KB	749 KB	748 KB
Total (128 cores)	11.8 MB	11.8 MB	4.0 MB	4.7 MB	3.1 MB
Red. over Orig.	-	0.1%	66%	60%	74%

Table 5.4: Memory budget for multicore simulation



(a) Cache performance in the 8-core configuration. The multicore *Exclusive* scheme reduces the miss rate more effectively than the best single-core *Indirect* scheme.



(b) Speedup of the novel multicore schemes relative to the *original* scheme in the 8-core configuration. In this configuration, there are enough physical host cores to run every simulated core, resulting in no thread contention on the host.

Figure 5.12: Multicore schemes evaluation.

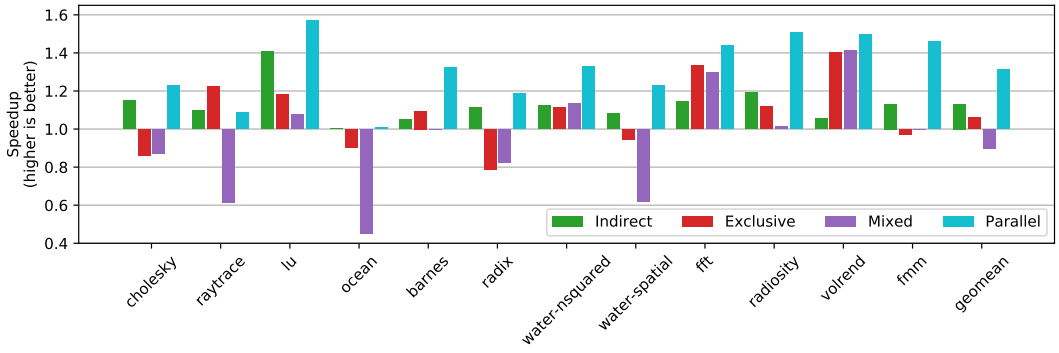


Figure 5.13: Speedup of the novel multicore schemes relative to the *original* scheme, in the 128-core configuration. This configuration simulates guest cores using 128 threads on a 10-core machine, resulting in significant host thread contention.

of 1.05× and 1.12× respectively, for 8-core configuration. Although the schemes are sharing memory between cores, the synchronisation overhead negates the benefits of improved cache performance. In fact, both schemes are often outperformed by the *Indirect* scheme, which does not provide any multicore sharing but does not require any synchronisation.

The *Parallel* scheme achieves the best performance, with speed-ups of up to 1.53×, and an average speedup of 1.36× over the *Original* scheme. This outstanding performance is achieved due to the low synchronisation overhead and high cache performance as a result of multicore *decode object* sharing. In fact, the *Parallel* scheme is the best performing scheme for all benchmarks.

The *Mixed* scheme always outperforms the *Exclusive* scheme, because the *Exclusive* scheme must acquire a lock to execute every instruction, resulting in sequential execution of the most contended instructions. On the other hand, the *Mixed* scheme promotes *decode objects* of contended instructions to a small core-private cache, improving execution parallelism. On average, the *Mixed* scheme results in about 35% of instructions being executed directly from the private decode cache.

In the 128-core configuration, the *Parallel* scheme continues to dominate, with a speed-up over the *Original* scheme of up to 1.57× and 1.31× on average (Figure 5.13). The `raytrace` benchmark is a notable exception to this trend, as the *Exclusive* scheme outperforms the *Parallel* scheme in this instance. Due to the large instruction footprint of `raytrace`, the *Parallel* scheme reaches the memory limit more than 600 times during the execution, and has to perform the costly *stop-the-world* reset. Simulating 128 virtual cores makes this reset particularly costly, resulting in diminished runtime

performance. Note, the *Parallel* scheme still outperforms the *Original* scheme.

Interestingly, the *Mixed* scheme results in worse runtime performance than the *Exclusive* scheme for simulated core counts that exceed the number of physical host cores. This degradation occurs due to increased pressure on the host operating system when scheduling the threads that run the virtual cores. The *Mixed* scheme is designed to keep the simulation running, by reducing contention on the locks that manage access to shared decoded instruction objects. However, in certain workloads guest worker threads use their own synchronisation primitives to wait for the initialization thread to complete, resulting in those worker threads running faster, and creating host OS thread scheduling contention that ultimately slows down the guest initialization thread.

5.4.3 Critical Evaluation

Our novel schemes have showed varied performance benefits for different benchmarks. For example, benchmarks with low instruction footprints do not benefit from improved cache performance, but rather suffer a slowdown due to the additional overheads incurred in the novel schemes. For this class of applications, an adaptive scheme could monitor the current cache performance and switch between decode cache schemes dynamically to choose an optimal caching strategy, given a constrained memory budget. Dynamic scheme switching might be particularly beneficial to applications with phased behaviour, as different application phases are likely to exhibit different instruction pressure.

Similar adaptations can also be explored for multicore simulation. For example, an application can be initialized using the low-synchronisation *Parallel* scheme, and switch to the *Exclusive* scheme dynamically if too many *stop-the-world* resets are observed. Later, if increasing instruction contention is observed, the decode caching scheme can be changed again to the *Mixed* scheme.

Another limitation of the multicore scheme is homogeneity, i.e. each virtual core is exposed to the same shared decode cache hierarchy. This might be suboptimal memory allocation for task-parallel applications, as each task might have different instruction footprints. Even in data-parallel applications, execution often comprises a sequential initialization phase with a high-instruction footprint, and a parallel phase with a smaller kernel. Increasing the memory budget, or making the decode cache completely private for the application thread performing initialization, might significantly speed up the overall application.

5.5 Summary and Conclusions

In this chapter, we have demonstrated that traditional decode cache approaches to interpreter-based ISS under-utilise the memory allocated to them, due to duplication of *decoded instruction objects*. By introducing a novel cache data structure indexed by instruction encoding, we reduce the number of instructions *requiring* decoding (i.e. not cached) by a factor of 2.6× on average. The improved cache performance results in an average runtime speed-up of 1.08×, and up to 1.25×, whilst using the same amount of memory as state of the art decode caches using traditional cache organization.

Another level of duplication is observed in multicore simulation, when different virtual cores use the same instructions and duplicate the *decoded instruction objects* in their respective private decode caches. Our novel multicore schemes improve memory utilisation by removing this duplication using a globally shared decode cache. We propose several schemes with varying trade-offs related to synchronisation of the shared data structure. In an 8-core simulation, our novel schemes improved cache performance significantly, resulting in less than 0.5% of instructions requiring decoding, while using the same amount of memory as the traditional cache. Runtime speedup achieved up to 1.53× and 1.36× on average. In 128-core simulation, speed-ups of up to 1.57× and 1.31× on average were observed, while using only 27% of memory relative to a traditional decode cache.

Although the new schemes allow for efficient caching of *decode instruction objects*, an ISS can observe frequent invalidations of the cache, necessitating re-decoding of guest instructions. For example, in case of a guest application modifying its own instructions, the cached entries no longer correspond to the current guest state and must be invalidated. Unfortunately, this protection mechanism if often triggered without any actual self-modifying code behaviour. In particular, guest applications relying on dynamic code generator might write new code into a memory location adjacent to already executed code. The next chapter introduces techniques to recognise this memory access pattern and avoid spurious code invalidations.

Chapter 6

Improved Detection of Self-Modifying Code

Detecting and handling of SMC is a significant challenge for ISSs (see Section 2.4.1). Since most guest applications never modify their instructions, optimisation techniques employed by ISSs assume that SMC is rare. Relying on guest code being static allows hypervisors to cache and optimise guest instructions, provided there is a mechanism to detect a rare code modification event (to be resolved by invalidating the hypervisor caches). The detection mechanism is typically integrated with the guest memory simulation model to monitor code and data in guest memory with little overhead. Each guest page containing code can be protected against write access, while accessing data pages remains unrestricted. This design enables high performance simulation of executables generated by static compilers (that place code and data on distinct pages). However, JIT compilation runtimes experience frequent false-positive SMC detection and suffer from an excessive overhead of SMC handling as a result.

In particular, DGC guest applications necessitate both writing to and executing from the same pages, as writes into code pages occur at least during code generation, which is often immediately followed by execution of the newly generated code. Furthermore, some DGC runtimes interleave metadata with dynamically generated code (e.g. V8 JavaScript Engine by Google [43]) resulting in ongoing writes into code pages. This access pattern is misidentified as SMC by the standard page-protection mechanism, leading to an excessive number of code invalidations.

Intuitively, SMC can be understood as an application modifying its own instructions while it is executing. Besides correctly handling this use-case by a hypervisor, further consideration must be given to avoid executing potentially stale guest instruc-

tions from a code cache. For example, in case of a code cache indexed by guest virtual addresses, guest MMU changes must be intercepted to invalidate the corresponding entries in the code cache. For the purpose of readability, we repeat the definition of SMC from the hypervisor perspective introduced in Section 2.4.1.

Definition 6.1. *Self-Modifying Code* is observed if two fetch operations from the same memory location (virtual PC) return different fetch values

Although SMC detection mechanisms relying on protecting guest memory at page granularity are efficient for many applications, DGC applications are prone to suffer from *false-sharing*. This happens when guest code fetches and data writes are performed within the same page, but do not result in SMC according to Definition 6.1.

This chapter proposes optimisations aimed at avoiding unnecessary code invalidations during simulation of DGC guest applications due to misidentification of SMC. The **key idea** is to precisely track memory locations containing code within each page to differentiate true SMC from false sharing.

6.1 Motivating Example

A microbenchmark highlighting the extreme case of DGC can be constructed from a few assembly instructions that copy themselves to a subsequent memory location, and then jumping to the new copy, as exemplified in Listing 6.1. The code copies itself using the loop between lines 6-8. After the code bytes are copied to the memory location corresponding to line 10, execution continues using the new copy of the loop on line 10.

```
1  start:
2    mov r0, loop   # copy source
3    mov r1, new    # copy destination
4    mov r2, new    # until source reaches here
5  loop:
6    ld.ab  r3, [r0, 4]    # also increment address [r0]
7    st.ab  r3, [r1, 4]    # also increment address [r1]
8    bne r0, r2, loop
9    mov r2, r1
10 new:
```

Listing 6.1: Selfgrow copies itself and continues executing the new copy.

Selfgrow is not self-modifying code, according to Definition 6.1. Although the effective write address stored in `r1` to be used during code generation (line 7) corresponds to the same page as the code itself, the address itself has never been fetched from before. Nevertheless, executing this application in an ISS results in repeated code invalidations caused by SMC protection mechanisms, undoing most optimisations employed by the hypervisor. In fact, disabling the corresponding invalidation yields a speedup of up to 38×.

6.2 Background

Detection of SMC in most DBTs is based on protection of guest memory. The main idea is to artificially protect code pages against write access. As a result, any write operation into the protected page resulting from simulating the guest application triggers a page-fault. During fault handling, the hypervisor determines if the fault is real (i.e. the guest write violated the memory access permissions) or spurious (the page was artificially protected by the hypervisor). Real faults are propagated to the guest, while spurious faults constitute SMC detection, resulting in all code corresponding to the faulting page being invalidated.

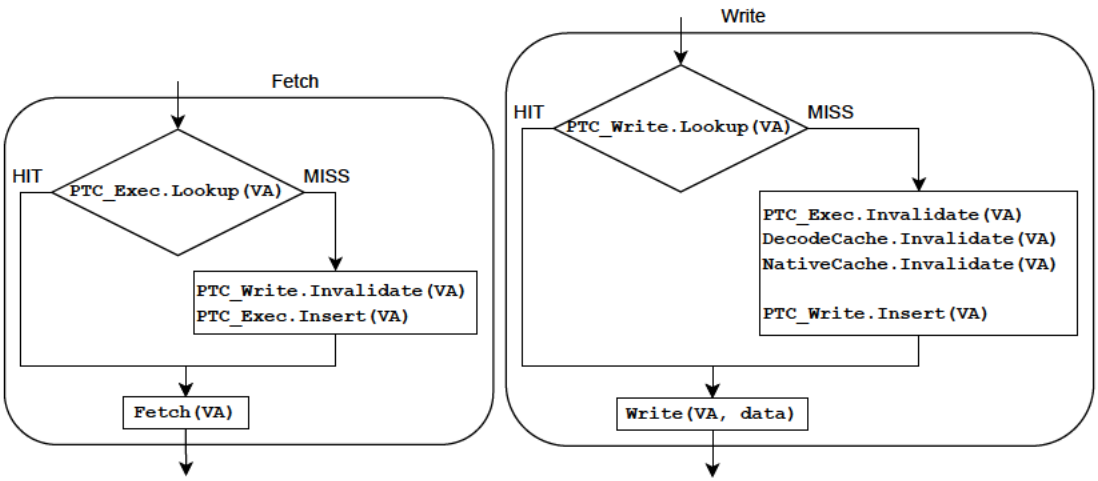
In particular, DynamoRio marks code pages as read-only using OS page protection mechanism [20]. QEMU full system simulation uses a software MMU module to translate all memory addresses from the guest address space to the host address space [13]. As part of memory address translation using the SoftMMU, writes into pages marked as read-only warrant further handling as potential SMC.

Our binary translator, nSIM, employs a software approach to guest memory translation based on the *Page Translation Cache*. nSIM instantiates separate PTCs for *Exec/Read/Write* access types, private to each simulated core. For a correct detection of SMC, it is crucial to maintain the following invariant:

Invariant 1. *A page cannot be cached by both, the Exec and Write, Page Translation Caches at the same time.*

In other words, guest memory translation can be optimised to a single lookup in a PTC for either a fetch operation or a write operation, but not for both at the same time.

Maintaining this invariant ensures writes to code pages cause a Write PTC miss. Whilst handling the write miss, any cached code corresponding to the same page is invalidated, as depicted by Figure 6.1.



(a) Fetch (b) Write

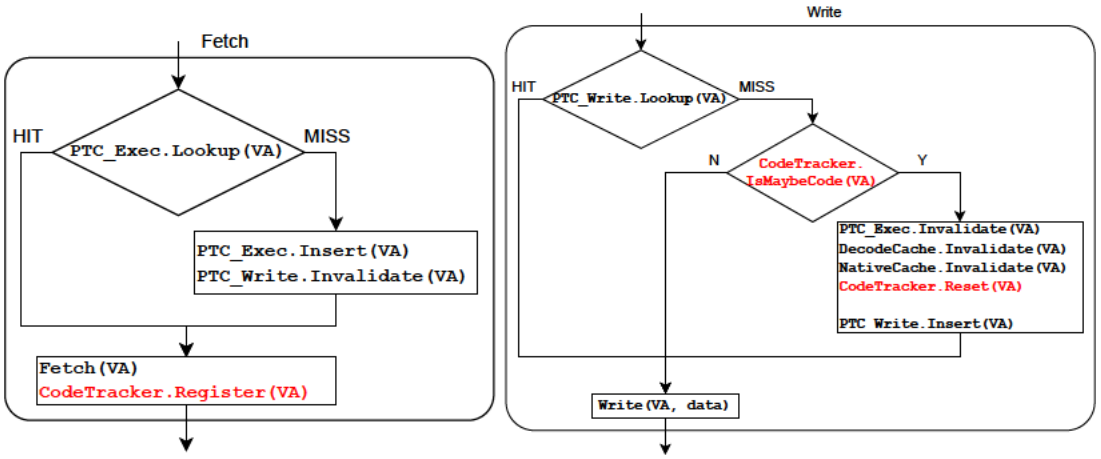
Figure 6.1: Detection and handling of self-modifying code in nSIM. A page cannot be cached in both, Exec and Write Page Translation Cache entries. This forces memory writes into code pages to *fault*. Handling a page fault during a write operation invalidates any code contained on the same page.

6.3 Methodology

Preliminary trace analysis of simulating QEMU [13] running `502.gcc` workload from SPEC CPU17 benchmark suite [65] shows that SMC protection mechanism at page granularity results in many unnecessary code invalidations. In fact, up to 95% of code invalidations were the result of false sharing of code and data on the same page.

We propose a new code tracking module to maintain a record of guest memory corresponding to executed instructions. This allows the code tracker to more accurately differentiate write targets modifying previously executed code from write targets modifying data adjacent to code. The code tracker provides a `Register(VA)` function to update its view of memory with every new instruction located at a particular virtual address. Then, the `IsMaybeCode(VA)` function can be used to query if a particular virtual address may potentially contain an instruction. Such a non-definitive query still enables precise tracking but also allows for approximate tracking with smaller implementation overhead. The code tracker also provides a `Reset(VA)` function that starts instruction tracking for a particular page anew.

By inserting the code tracker into PTC slow-path routines, we can avoid code invalidations due to false sharing while still achieving correctness (see Figure 6.2). It is crucial to maintain Invariant 1 so that writes into pages containing cached code always



(a) Fetch (b) Write

Figure 6.2: Improved detection and handling of self-modifying code in nSIM in single-core. Each code instruction is registered with a code tracker during its first fetch. Subsequent write misses query the write location against registered code to filter out false-sharing writes, for which, code invalidation can be omitted.

result in taking the PTC slow-path. Note, the PTC fast-path implies that no additional checks are required, and memory can be accessed without any side effects.

Every guest instruction being fetched for the first time (i.e. cold fetches) *must* register with the code tracker. In a mixed interpretive and native execution, every instruction is first interpreted before it manifests in a DBT code region. Therefore, registering instructions only during interpretation ensures all instructions are registered. Note, many instructions might still end up being registered several times, resulting in a significant registration overhead during interpretation but no additional registration cost after DBT code becomes available. The registration overhead can be further reduced by only registering instructions upon a decode cache miss.

During instruction fetch, pages containing code are removed from PTC_Write to ensure future writes take the PTC slow-path. At a later execution point, a write into the same page queries the code tracker to compare the write target against previously fetched instructions. If the write target overlaps with any fetched instruction, code invalidation is performed. Firstly, the PTC_Exec entry for the page is removed and a PTC_Write entry is created, maintaining Invariant 1. Since future writes into the page will take the PTC fast-path, writes will no longer be tracked with respect to potential code modification, thus the whole page of code must be invalidated from all hypervisor caches. Finally, the write target is written to.

This design prioritises fetch operations at the cost of slowing down writes targeting code pages (note, data-only pages are unaffected and data writes to them do not experience any additional overhead). Since writes to code pages in DGC applications are much less frequent than fetching from those pages, slowdown of these writes is preferred to a slowdown of fetch operations.

6.3.1 Code Tracking Schemes

We experiment with several implementations of the code tracker. To achieve correctness, a conservative approach, that considers every address as code, is sufficient, i.e. `IsMaybeCode(VA)` returning *true* is always safe. Such a trivial code tracker can be used as a *Baseline* scheme for comparison of other schemes (i.e. it corresponds to the non-existent code tracking in the original nSIM). Although safe, this scheme results in false-positive detections of SMC in presence of *false-sharing*. Another trivial code tracker is *Optimistic*, indicating no memory location corresponds to code. This scheme is too optimistic, indicating false-negative SMC detections even in the presence of true SMC. As a result, this scheme does not ever invalidate modified code, and thus is unsafe, but demonstrates the performance upper bound.

6.3.1.1 Watermark Scheme

The *Watermark* code tracker keeps a simple memory pointer per page indicating the highest memory location ever fetched from. If a write address is greater than this watermark, the memory location is guaranteed to not contain previously fetched code. If a write address is smaller or equal to the watermark pointer, the write can potentially modify an instruction and the whole page worth of code is invalidated.

This code tracking scheme efficiently detects false-sharing in DGCs that generate new code in subsequent memory locations prior to execution, e.g. QEMU [13]. Such an access pattern is typical in DGC engines that generate code fragments into a flat memory buffer.

6.3.1.2 Bitmap Scheme

A more granular tracking can keep a bitmap per page, indicating for each byte if it corresponds to an instruction. This fine code tracking granularity can differentiate code and data even between adjacent bytes. Since this fine tracking requires 1 bit per each byte of tracked memory, a 4096 byte page requires 4096 bits (512 bytes) for

code tracking. This can be implemented more efficiently if the ISA only has word-sized instructions. In that case, one bit per word is required, resulting in 128 bytes of tracking data per page.

6.3.1.3 Regions Scheme

Regions code tracker stores a set of memory address ranges corresponding to code per page. Such a fine tracking granularity achieves precise differentiation of code and data on the same page, even when interleaved. This access pattern is observed in Google JavaScript engine V8 [43], which interleaves metadata with dynamically generated functions.

Core tracking at region granularity requires a more complex data structure capable of tracking a list of region start and end addresses. Additionally, it ought to provide the option to extend and/or merge existing regions dynamically as more instructions are being registered during interpretation. After initial cost of registration of instructions executed from a particular page, the region-based code tracker requires less memory than the *Bitmap* tracking on average, while providing the same precision in differentiating code and data. This can be particularly effective when guest pages contain only a few large contiguous regions of code.

6.3.1.4 Working Example

To demonstrate the granularity and memory consumption of the three non-trivial code tracking schemes, Table 6.1 shows an execution trace of a single program. In each step, data tracking of each scheme is shown.

Each code tracking scheme can be found to be the best performing scheme, depending on the page size, guest ISA characteristics, and guest application memory access patterns. The *Watermark* scheme incurs the smallest computational and memory overheads, but cannot precisely capture more complex guest behaviour. The *Regions* and the *Bitmap* schemes are more precise, with more fine-grained tracking. While code registration and checking of the *Bitmap* scheme is computationally simpler than the *Regions* based tracking, it can potentially require more memory, putting more pressure on the host micro-architecture.

```
1 0x00:            nop
2 0x04:            nop
3 0x08:            nop
4 0x0c:            j 0x20
5 0x10 <start>:    nop
6 0x14:            nop
7 0x18:            j 0x00
8 0x1c:
9 0x20:            nop
```

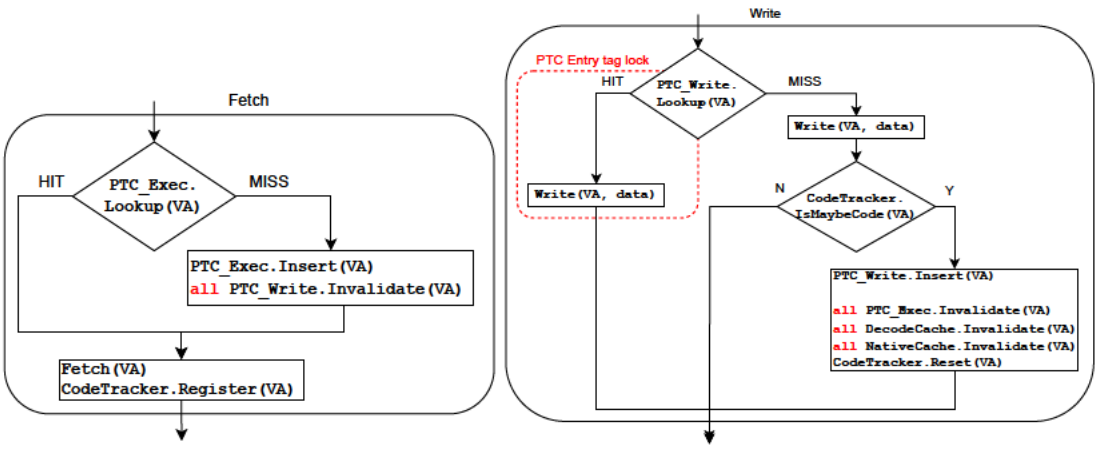
Time	PC	Watermark	Bitmap	Regions
1	0x10:	0x10	0000 1000 0	0x10-0x14
2	0x14:	0x14	0000 1100 0	0x10-0x18
3	0x18:	0x18	0000 1110 0	0x10-0x1c
4	0x00:	0x18	1000 1110 0	0x00-0x04, 0x10-0x1c
5	0x04:	0x18	1100 1110 0	0x00-0x08, 0x10-0x1c
6	0x08:	0x18	1110 1110 0	0x00-0x0c, 0x10-0x1c
7	0x0c:	0x18	1111 1110 0	0x00-0x1c
8	0x20:	0x20	1111 1110 1	0x00-0x1c, 0x20-0x24

Table 6.1: Trace of code tracking scheme's data during execution of a simple program.

6.3.2 Avoiding False-Sharing in Multicore Scenario

Simulating multiple cores poses extra challenges for correct code tracking and handling of SMC. Firstly, code tracking schemes must be extended to support thread-safety, so that multiple cores can register code addresses and query the state of the tracked memory without risking data races. A core-private code tracking schemes would incur no synchronisation overhead during registration, but system-wide data-vs-code differentiation would require checking multiple code trackers across the system for each data write targeting a code page. Instead, a shared code tracker is used. This requires that all updates to code tracker data are properly synchronised. However, after initial registration, querying the code tracker is a read operation that can be easily parallelised. Costly synchronisation is required only if the code tracker has to be restarted due to true SMC being detected, which is already costly because of the necessary code invalidation.

PTCs also have to be extended to correctly detect SMC in the multicore context. Since PTC data structures are core-private, Invariant 1 is extended to ensure concurrent



(a) Fetch (b) Write

Figure 6.3: Detection and handling of self-modifying code in multicore. PTC invalidation applies across all simulated cores. PTC_Write invalidation is synchronised to wait until each simulated core finished executing Write fast-path.

cores cannot modify code without also invalidating caches in ALL simulated cores. Note, the new Invariant 2 is enforced only outside of the memory access simulation section. While executing Fetch or Write access simulation, it is permitted for the invariant to be temporarily violated.

Invariant 2. *A page can be cached either by Exec-type or Write-type PTC across the whole simulated system.*

To ensure Invariant 2 is maintained, PTC invalidation has to apply across the whole system. Since each PTC is core-private and each core is simulated by a corresponding host thread, additional synchronisation issues must be addressed. SMC detection and handling for multicore is presented in Figure 6.3.

The implementation must be safe when a Fetch operation is performed concurrently with a Write operation. By maintaining Invariant 2, Fetch and Write for a particular memory location never both execute the fast path of the respective operation. Therefore, there are three important combinations of the Fetch and Write operations' paths that could result in a data race. During correctness analysis, it is important to consider the case when a Write actually modifies an instruction to be Fetched. Note, for Writes modifying data, performance (not correctness) is the primary concern.

Core 1 - Fetch	Core 2 - Write
Insert C1.PTC_Exec	Insert C2.PTC_Write
Invalidate C2.PTC_Write	Invalidate C1.PTC_Exec

Table 6.2: PTC insertion and invalidation between two concurrent cores. There is no interleaving of the concurrent operations that would end up in a final state of the PTCs violating Invariant 2.

6.3.2.1 Fetch miss vs Write miss

If both Fetch and Write operations execute the slow-path, the actual `write` is performed as the first step of the Write operation. This guarantees the new instruction value is picked up by the actual `fetch`, as this is the last step of the Fetch operation. Note, this relies on the strong memory order of the host machine preventing the `fetch` read from being reordered before the PTC manipulations on the Fetch slow-path. Similarly, the memory model prevents the `write` to be reordered after the PTC manipulation on the Write slow-path.

The more important consideration is the PTC manipulation resulting in Invariant 2 being maintained. Consider a simplified example of PTC interaction of two concurrent cores in table 6.2. Although the accessed page can temporarily be inserted in both PTC_Exec and PTC_Write, for all allowed reordering of the PTC operations from the two concurrent cores (under a TSO memory model), the page cannot end up being cached by both PTCs. Thus, Invariant 2 is maintained after the Fetch and Write operations are completed.

6.3.2.2 Fetch miss vs Write hit

Fast-path of a Write operation could potentially be missed if not properly synchronised with Fetch operation. In particular, an interleaving depicted in Table 6.3 can result in a code modification going unnoticed. Although Invariant 2 is maintained in this interleaving, as the Fetch operation invalidates PTC_Write, a write of the new instruction (step t5) happens after the instruction is fetched (step t4). Since the Write operation executed the fast path, the fetched instruction might never be invalidated and re-fetched, i.e. the instruction modifying Write has been missed.

To prevent such interleaving, entering Write fast-path locks the corresponding PTC entry. Then, the entry cannot be invalidated by concurrent Fetch slow-path until the lock is released, i.e. after the `write` has completed. This is equivalent to Load-link and

Time	Core 1 - Fetch	Core 2 - Write
t_1		C2.PTC_Write lookup **hits**
t_2	Insert C1.PTC_Exec	
t_3	Invalidate C2.PTC_Write	
t_4	Fetch original instruction	
t_5		Write new instruction

Table 6.3: A particular interleaving of operations that leads to a race-condition resulting in a missed code modification.

Write operation synchronisation from Section 4.2.3. Note, the PTC entry lock is core-private, and as such is contended only during concurrent invalidation, i.e. concurrent regular Write operations can proceed in parallel.

6.3.2.3 Fetch hit vs Write miss

Fetch operation fast-path does not need to be perfectly synchronised with a concurrent Write operation. Fetching a stale instruction is safe in the absence of explicit code synchronisation instructions (e.g. the guest performing an instruction cache invalidation). This is equivalent to the case when the concurrent Write operation would be performed at a later point in time. A similar instruction consistency scenario takes place if the concurrent Write operation happens in-between a Fetch and Execute operation of the modified instruction.

Although the Fetch operation reads stale instruction data, the modifying Write has **not** been missed, as the corresponding PTC_Exec entry is invalidated and future execution has to re-Fetch the modified instruction.

6.4 Evaluation

The code tracking schemes are implemented in the commercial Synopsys ARC© nSIM ISS simulator and evaluated using various workloads. Firstly, the overheads of the different novel schemes are measured using benchmarks without any false-sharing of code and data. Then, the key benefits of employing precise code tracking with DGC workloads are presented. Finally, micro-benchmarks are used to stress various patterns of data and code sharing to demonstrate the code tracking power of the novel schemes.

The impact of using all novel code tracking schemes is compared to the performance of the scheme that does not track code within code pages. This *Baseline* scheme

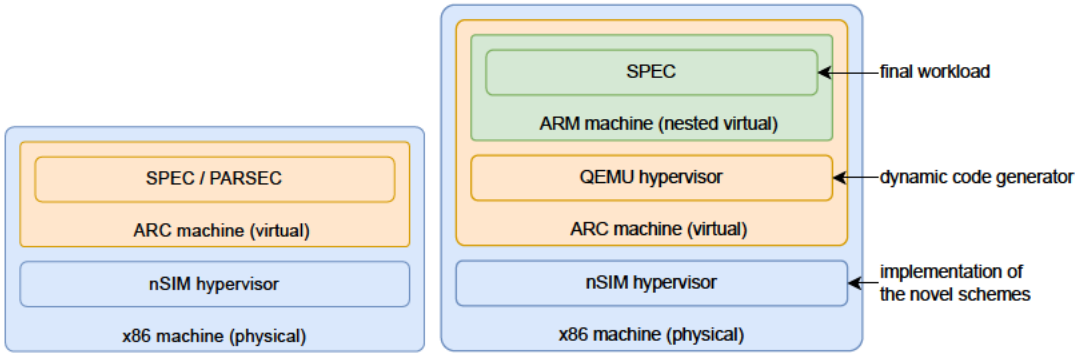
Host	
Processor	Intel Xeon E5-2640 v4
Frequency	2.4 GHz
Cores	10
L1 cache (private)	32K
L2 cache (private)	256K
L3 cache (shared)	25600K
Operating System	CentOS 7.3
Hypervisor	
Emulation mode	mixed JIT+interpreter
Decode Cache Type	PC-indexed
Decode Cache Size	8096 entries
JIT	asynchronous
JIT threads	3

Table 6.4: Host machine and Hypervisor Configuration

is still correct, but only "tracks" code at page granularity and cannot distinguish false-sharing behaviour of code and data within a memory page from true SMC. Therefore, for every write operation targeting a page containing code, all the code corresponding to the page is invalidated.

6.4.1 Experimental Setup

Benchmarks are evaluated using the configuration in Table 6.4. nSIM is run in an optimising mode, employing a large decode cache for interpretive execution. First, the overheads involved in code tracking are evaluated using workloads without any DGC behaviour. Overhead data is collected using only interpretive execution, as this is the execution mode that stresses the code tracking the most (i.e. native execution avoids further code registration with the code trackers). Key results evaluation employs an interpreter as well as a JIT engine with 3 compilation worker threads for native execution of guest code. Such a configuration can be typically found in an ISS fully optimised for simulation speed. Micro-benchmarks are evaluated using only interpretive execution. This is sufficient to demonstrate the targeted effect of avoiding code invalidations due to false-sharing. Furthermore, some micro-benchmarks are designed to continually generate new guest code, and thus do not reuse code enough to benefit



(a) Benchmark workload running inside ARC virtual machine.

(b) Nested hypervisors for Dynamically Generated Code evaluation.

Figure 6.4: Virtual Machines setup for overhead evaluation and for evaluation with DGC workloads.

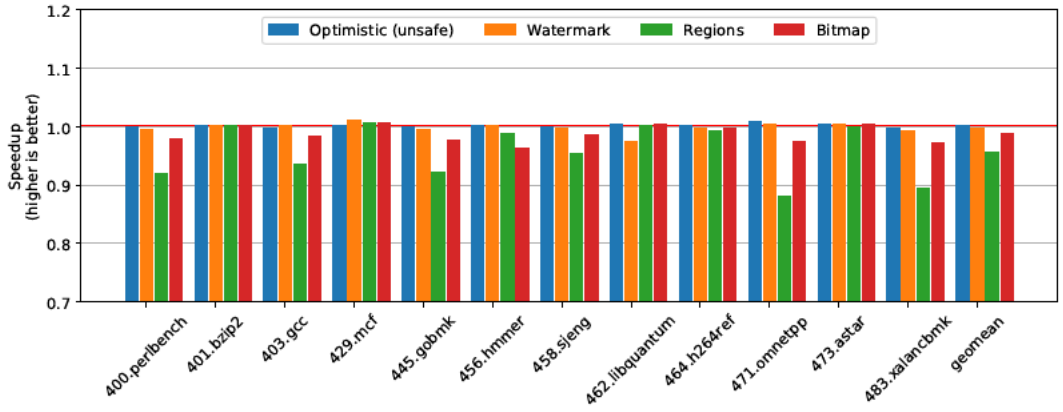
from employing JIT compilation.

6.4.2 Overheads

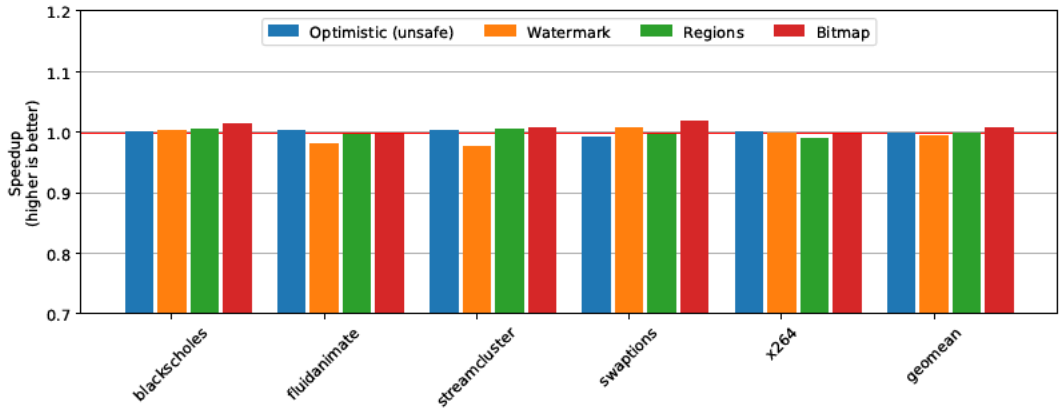
Evaluating benchmarks that do not exhibit any code and data sharing on the same memory pages measures the implementation overhead of the novel code tracking schemes. These benchmarks run directly inside the ARC virtual machine created by nSIM, as depicted in Figure 6.4a. Even though the SMC detection mechanism is never triggered, all guest instructions have to be registered with the code tracker. This puts stress on the data structures used for code tracking, as well as creates synchronisation overheads for multicore code tracking schemes.

Note, each guest instruction *needs* to be registered with a code tracker only once (unless the corresponding page has been modified). Registering an instruction multiple times is always safe, but can result in unnecessary performance overhead. To avoid excessive registration, an instruction is registered only if there is a miss in the interpretive decode cache. Superfluous registrations can still occur in case of conflict-type decode cache misses.

Figure 6.5 depicts the overheads of employing the novel code tracking schemes. In an optimised ISS using hypervisor caches to memoise guest instructions, the code tracking overhead constitutes a relatively small portion of the overall runtime of the benchmarks. Single-core benchmarks (Figure 6.5a) experience the registration overhead more severely than multicore benchmarks (Figure 6.5b). This is caused by single-core benchmarks comprising larger workload kernels, resulting in larger instruction



(a) Single-core: SPEC06 Int



(b) Multicore: PARSEC with 10 cores.

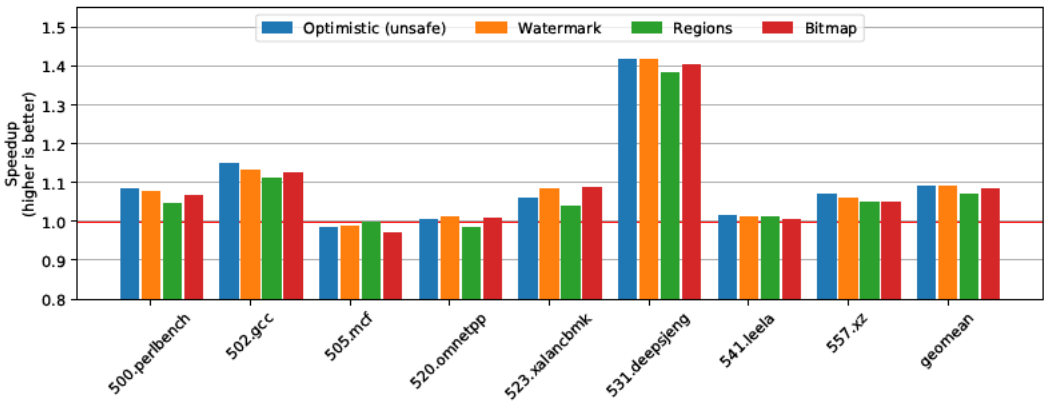
Figure 6.5: Overhead of the novel schemes relative to the *Baseline* scheme in interpretive simulation

footprint and typically more registration operations.

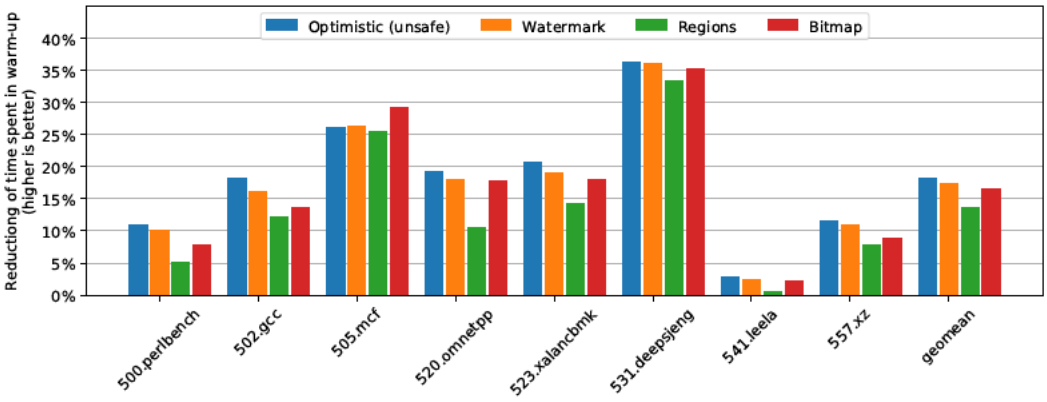
The greatest overhead is observed for the *Regions* code tracking scheme. On overage, the scheme results in a 0.96× slowdown and up to 0.88× slowdown for `471.omnetpp` benchmark. On the other hand, the *Watermark* and the *Bitmap* schemes require very little computation when guest instructions are being registered, with average slowdown for single-core benchmarks of 0.998× and 0.987×, respectively.

6.4.3 Key Results

The impact of code tracking on the ability to avoid unnecessary code invalidation in the presence of false sharing is evaluated using a DGC workload. The direct workload target for nSIM is QEMU ported for the ARC ISA. As described in more detail in Section 3.2, QEMU employs a JIT engine to compile guest code at basic block gran-



(a) Speedup of end-to-end runtime.



(b) Percentage reduction of the warm-up phase in a mixed-mode ISS.

Figure 6.6: Performance of code tracking schemes compared to the *Baseline* scheme in mixed-mode simulation with 3 JIT workers. SPEC17 Int benchmark suite is compiled for ARM and is running in nested QEMU-nSIM hypervisor.

ularity. The compilation is triggered whenever undiscovered guest code is reached. A new code section of the host code (ARC ISA) is generated into a contiguous memory buffer, followed by immediate native execution.

During QEMU's discovery of new guest code, the host code generation and its subsequent execution exhibits intense false sharing behaviour. This, however, is only a transitory period, as after the initial code discovery, the guest application can execute from the native cache without any further code generation. SPEC CPU17 Int benchmark suite [65] compiled for the ARM ISA is used as a workload for QEMU. This stresses QEMU's DGC in different ways depending on the benchmark's code size, discovery patterns, and proportion of the initial code generation phase. The nested virtualisation setup is depicted in Figure 6.4b.

The total runtime speedup relative to the *Baseline* scheme is depicted in Figure 6.6a. For some benchmarks, employing an accurate code tracking schemes significantly improves performance. The best performance improvement is observed for the *Watermark* scheme, with up to 1.41× speedup (*531.deepsjeng* benchmark) and 1.09× speedup on average. Note, the average speedup of the *Optimistic* unsafe scheme, that assumes code is never modified, is identical, at 1.09×. The excellent performance of the *Watermark* scheme can be attributed to its low computational overhead (i.e. incrementing a single value per page) and the ability to capture the behaviour of the workload application (i.e. QEMU's code generation pattern).

Some benchmarks exhibit little to no performance difference regardless of the code tracking scheme used (e.g. `505.mcf` and `520.omnetppp`). These are long-running benchmarks for which the code generation transitory period is small relative to the total runtime of the benchmark. Since this transitory period necessitates interpretive execution, measuring the overall time spent interpreting (as opposed to native execution) better reveals the benefits of the novel schemes for the initial application phase of a DGC engine. For example, the long-running `505.mcf` benchmark has near identical end-to-end runtime regardless of the used code tracking scheme. However, the time spent in interpretive execution can be reduced by more than 25% when employing the novel schemes relative to *Baseline*. This performance improvement in the warm-up period can benefit applications relying on fast startup or responsiveness.

6.4.4 Micro-benchmarks

A suite of micro-benchmarks was designed to stress the false-sharing memory access pattern, and thus none of the micro-benchmarks exhibits true SMC. The benchmarks can be categorised by using either *moving* or *stable* memory. *Moving* benchmarks continuously keep generating and executing new code sections. This represents a typical DGC access pattern employed by JIT engines, e.g. QEMU [13]. On the other hand, *stable* benchmarks repeatedly access the same memory as either data or instructions. This represents access pattern of interleaving data with code, as observed in V8 JavaScript engine [43].

The first set of micro-benchmarks is designed to execute as single-core applications, stressing the code tracking schemes themselves. Then, the micro-benchmarks are extended for multicore. This also stresses the synchronisation during the interactions of the core-private *Page Translation Caches*. For multicore benchmarks, multi-

core variants of the code tracking schemes are used.

Selfgrow is a micro-benchmark used as a motivating example in Section 6.1. It is constructed in *ARCompact* assembly as a short sequence of instructions copying themselves, and then executing the new copy, repeating the process indefinitely.

Increment micro-benchmark imitates a JIT engine by repeatedly generating a small function for incrementing a data pointer and then executing the function. Although *increment* resembles *selfgrow* benchmark (i.e. it is also a *moving* benchmark), it has distinct code generation and code execution phases that alternate. Note, the data pointer being incremented does not reside on the same page as the function incrementing it, i.e. the false-sharing writes are observed only during code generation, not during execution of the data increment function.

Parallel Increment is a multicore version of the *increment* benchmark. Multiple guest cores repeat the same sequence of code generation into a shared buffer, followed by execution of the generated function. The code generation is guarded by mutual exclusion, while the code execution proceeds in parallel. This benchmark stresses concurrent writes into code pages being executed, i.e. concurrent false-sharing.

The **Random** micro-benchmark repeatedly accesses a random location in a page comprising a sequence of `return` instructions. It alternates data access with execution access by first reading and writing back a value at a random location on the page, and then jumping to a random location on the page. This benchmark exhibits a chaotic access pattern. Note, although the write back address accesses a previously fetched instruction, this is not true self-modifying code as defined by Definition 6.1, since the written value is exactly the same as the original instruction.

Parallel Random employs multiple cores, all exercising the random access pattern using a single shared memory page.

```
1 void fibonacci(int* data) {
2   int n = 300, temp = 1;
3   *data = 1;
4   for (int i = 2; i < n; i++) {
5     int last_data = *data;
6     *data = *data + temp;
7     temp = last_data;
8   }
9 }
```

Listing 6.2: Fibonacci function accessing input data pointer

Fibonacci function kernel is a building block for several micro-benchmarks (see Listing 6.2). The function computes the first 300 numbers in the Fibonacci sequence using a memory location for intermediate results provided as a *data* pointer argument. Several micro-benchmarks are constructed by controlling the position of the *data* relative to the *Fibonacci* function in memory. Since the benchmark kernel does not generate any new code, all micro-benchmarks based on the `fibonacci` kernel are *stable*.

Fibonacci after benchmark positions the `data` pointer right after the `fibonacci` function. On the other hand, **fibonacci before** positions the `data` pointer before the function in memory. The main difference between the two variants is the ability of *Watermark* code tracking scheme to differentiate the false-sharing behaviour. In case of *fibonacci before* variant, *Watermark* code tracker identifies each data write as potentially modifying code, i.e. it is unable to recognise the false-sharing behaviour.

Multicore micro-benchmarks using the *fibonacci* kernel are constructed by varying the location of the `data` pointer used by different cores.

Parallel Fibonacci Alternating configures all cores to execute the same *fibonacci* function and to use one of two `data` pointer, with every other core alternating using a `data` pointer either *after* or *before* the function.

In **Parallel Fibonacci Consecutive**, each core uses its own copy of the *fibonacci* function followed by a `data` pointer. Both code and data for multiple cores are allocated consecutively. As a result, the memory layout of this benchmark is a multi-copy of *Fibonacci After*, with all copies residing on the same memory page.

Parallel Fibonacci Independent similarly copies *Fibonacci After* memory layout but places each copy on its own dedicated page.

The performance comparison of various code tracking schemes for micro-benchmarks is depicted in Figure 6.7. Since the *Baseline* scheme does not provide any code tracking, its performance is poor due to frequent code invalidations. The resulting slowdown is particularly pronounced in micro-benchmarks with intense false-sharing. Due to the massive performance benefit of employing code tracking schemes, the speedups are plotted on a logarithmic scale.

Single-core evaluation manifests the greatest speedup for `selfgrow` benchmark. This is also the benchmark that exhibits the most intense false-sharing behaviour. The *Watermark* scheme achieves a 36.8× speedup, compared to the unsafe *Optimistic* scheme of 38.6×. For other benchmarks, the *Watermark* scheme performs the best among the correct schemes only as long as it fits the access pattern of the particular benchmark. This is **not** the case for `fibonacci before` and `random` benchmarks.

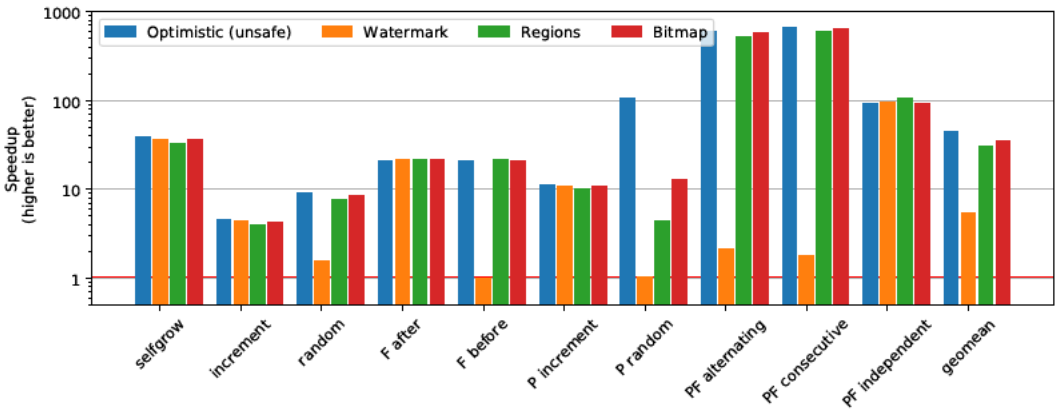


Figure 6.7: Speedup relative to *Baseline* in interpretive simulation. Parallel benchmarks were evaluated in 8-core configuration.

`Fibonacci before`, in particular, exhibits an access pattern exactly opposite to the watermark-based incremental code tracking, as the data access targets a lower memory location than the executed function. As a result, the *Watermark* scheme performs the same as the *Baseline* scheme without any speedup. The effect is less pronounced for `random` benchmark, as some random writes fit the watermark-based tracking, hence code invalidation can be avoided sometimes. For this benchmark, the *Watermark* scheme achieves a speedup of 1.56×.

Multicore benchmarks typically experience greater benefits of the novel schemes than their single-core variants. For example, the *Watermark* scheme shows a speedup for `parallel increment` benchmark of 10.9×, while only 4.44× for a single core `increment` variant. The speedup difference can be explained by a greater frequency of code invalidations in an 8-core system, i.e. any core's code is being invalidated as a result of other concurrent false-sharing writes, while in a single core system, code invalidations happen only as a result of the core's own false-sharing write.

`Parallel fibonacci consecutive` benchmark presents an interesting case. Although, the benchmark's memory layout is just a consecutively duplicated layout of `fibonacci after`, the *Watermark* scheme is unable to differentiate code and data when a global shared memory is accessed by multiple cores. As a result, its speedup is only 1.82× over the *Baseline* scheme. Note, a single-core version of the *Watermark* scheme, which tracks code and data only from the individual core's perspective, achieves a speedup similar to the unsafe *Optimistic* scheme of 674.5×.

Another interesting micro-benchmark is `parallel random`. Among the correct scheme, the best performing is the *Bitmap* scheme with a speedup of 13.1×, due to

its fine-grained code tracking and low computational overhead. On the other hand, the unsafe *Optimistic* scheme achieves a speedup of 106.5×. The correct schemes could potentially achieve similar performance if they tracked instructions' values as well as their location. However, modifying an instruction with the same instruction value (i.e. the instruction remains unchanged) is rare in practice.

6.5 Summary and Conclusions

In this chapter, we have demonstrated that guest applications exhibiting false-sharing of code and data in adjacent memory are excessively triggering the self-modifying code protection mechanism in hypervisors. As a result, hypervisors must often invalidate translated and cached guest code even in absence of true self-modifying code.

We propose integrating a code tracking mechanisms for affected pages of simulated guest memory. Leveraging *Page Translation Cache* infrastructure used for guest memory simulation, efficient memory monitoring of pages containing both code and data can be implemented across a multicore system. Within these pages, precise code tracking can be used to differentiate if any write operation targets a previously executed guest instruction or is a benign (i.e. data) write.

By avoiding unnecessary code invalidations, guest execution is allowed to be optimised by the hypervisor's caches. This is a particularly frequent use case for guest applications exhibiting dynamic code generation, such as a JIT-based runtime.

Chapter 7

Conclusion

This thesis demonstrated techniques for improving cross-architecture simulation of multicore systems in the areas of *correctness*, *memory efficiency*, and *performance*. In summary, the technical chapters presented the following contributions:

1. **Emulation of Atomic Synchronisation Instructions**: correct and performant emulation of Load-Linked/Store-Conditional atomic guest instructions on CISC based hosts

2. **Memory efficient decode cache**: efficient storage of Decode objects during interpretive guest execution with object sharing among multiple cores

3. **Improved Detection of Self-Modifying Code**: reduction of superfluous code invalidations by self-modifying-code protection mechanism in presence of dynamic code generation

7.1 Contributions

7.1.1 Emulation of Atomic Synchronisation Instructions

Chapter 4 presents techniques for handling LL/SC atomic guest instructions commonly used by RISC based architectures. These atomic instructions are notoriously hard to translate onto CISC based host architecture atomicity model. In contrast to LL/SC instructions providing atomic behaviour as a collaboration of two distinct instructions, CISC hosts typically offer a single *read-modify-write* atomic instruction, e.g. *Compare-And-Swap* (CAS). However, simulating multicore guests by concurrent

execution (i.e. each guest core executed by a dedicated host thread) on multicore hosts requires faithful support for the guest atomicity model.

Several techniques for handling LL/SC emulation are provided. An approximate approach relying on CAS instruction is scalable and simple to implement, but does not maintain the full semantics of the emulated LL/SC instructions. Instead, a software emulation approach can be implemented efficiently by relying on the guest memory translation infrastructure. The presented approach exhibits scalability and performance similar to the CAS approximation. Furthermore, it is formally proven to be correctly emulating the full semantics of LL/SC instructions, i.e. execution inside a simulator using the novel approach behaves identically to execution on a real hardware for all possible guest programs. Another correct scheme leverages hardware support (if available) by mapping emulation of LL/SC pair onto hardware transactional memory. This approach works well in short LL/SC sequences with low contention but suffers significant performance overhead in case of intense interactions of concurrent LL/SC accesses due to many resultant transaction aborts.

7.1.2 Memory efficient decode cache

Chapter 5 describes a novel organisation of Decode objects in a hypervisor cache used to store the results of the Decode stage during interpretation of frequently executed guest instructions. The major benefit of the novel organisation is the removal of entry duplication originally present when different PC locations contained identical guest instructions. By sharing Decode objects across different PC locations, less memory is required to cache the same amount of guest instructions, or conversely, more guest instructions can be cached with the same total memory usage.

Furthermore, the chapter extends the idea of sharing Decode objects to multicore sharing, where several cores can reuse the same Decode objects in a global decode cache. In this scenario, the memory requirement for storing Decode objects remains constant regardless of the number of simulated code. This approach exhibits great scalability to many-core simulation. In fact, the memory saving/efficiency becomes better the more cores are simulated.

7.1.3 Improved Detection of Self-Modifying Code

Chapter 6 addresses inefficiencies of SMC protection mechanism when faced with dynamic code generation. Typically, hypervisors translate and cache guest code in several

different formats. When self-modifying code (i.e. the guest execution re-writes its own instructions) is detected, the cached code in the hypervisors' data structures must be invalidated. However, the common invalidation approach is overly conservative. In particular, guests relying on dynamic code generation often trigger the code invalidation spuriously.

Proposed solutions improve the granularity of tracking memory locations corresponding to have-been-executed guest instructions in order to distinguish benign writes from true self-modifying behaviour. This greatly benefits guest memory access patterns observed in applications employing JIT compilation. Furthermore, the improved detection of self-modifying code enables efficient guest execution, even in case when data is interleaved in code sections and is continuously accessed.

7.2 Critical Analysis

7.2.1 Emulation of Atomic Synchronisation Instructions

Software-only approach using *Page Translation Cache* is generally recommended if the guest behaviour is unknown and the guest semantics should be fully supported. With high contention behaviour, the CAS approximation might be preferred. However, this approach will only support a particular usage of guest's LL/SC atomic instructions that rely on atomic changes of values (this is enough for a spinlock implementation). However, if the guest relies on full semantics of LL/SC instructions (i.e. SC can succeed only if there was no modification of the target memory locations since the last LL instruction), the CAS approximation results in divergent behaviour and potential corruption of the guest execution due to the ABA problem (i.e. the memory value changed from A to B and later back to A).

In case of low contention and short LL/SC sequences, a Hardware-Transactional-Memory scheme performs well by leveraging hardware support to track data races to detect interleaving writes. A major drawback of this scheme is the availability of this technology on the host. Furthermore, its performance can depend on the specific hardware implementation of HTM, in particular the granularity of tracking memory accesses, supported sizes of transactions, spurious transaction aborts, and the cost of speculative execution in case of aborts.

7.2.2 Memory efficient decode cache

Our novel techniques provide benefits in memory constrained environments. For a guest application with a small instruction footprint (i.e. the size of code corresponding to most of the execution), a simpler decode cache might result in better performance. In such a cache design, decode objects can be further specialised by their PC locations and particular context of each instruction. In multicore execution, decode objects can specialise to the core context. Furthermore, PC-indexed objects can exhibit better spacial locality, as consecutive instructions would likely be stored consecutively in the cache data structure, resulting in better performance of the host micro-architecture.

Multicore object sharing must address the consistency of each shared object in use, i.e. a decode object being used by one core cannot be invalidated by a concurrent core. Requiring exclusive access to instruction objects necessitates additional synchronisation and can create a sequentialisation point of many guest cores. For a many core applications, it is recommended to use a decode cache without invalidation. Such a design might require more memory for storing the shared decode objects, but can still result in a memory saving by employing a single big shared global cache instead of many smaller core-private caches. However, if not enough memory is allocated to the global cache relative to the application instruction footprint, the frequent stop-the-world invalidation of all decode objects can significantly hinder performance.

7.2.3 Improved Detection of Self-Modifying Code

Code tracking schemes greatly alleviate the problem of spurious code invalidations due to false-sharing of code and data on the same memory page. This is especially beneficial to applications employing dynamic code generation, e.g. a JIT engine. These applications typically generate new code into a memory buffer incrementally, and then execute the newly generated functions. For this memory access pattern, the *Watermark* code tracking scheme results in the best performance while requiring negligible memory and computational overheads. However, in these applications, employing a code tracking scheme only benefits the transient period during initial code generation. For long-running applications, the runtime becomes dominated by the period of executing the generated code, during which no more code is being generated. Nevertheless, the speedup of the initial transient code-generation period is important, as evidenced by a huge body of literature focusing on latency and warm-up period of JIT-based runtimes.

Alternatively, some applications interleave data and code on the same memory

pages. This results in continuous false-sharing, which impacts even the steady state period of JIT-based runtimes, when no more code is being generated. In such applications, it is recommended to utilise a more advanced code tracking system. The *Bitmask* code tracker is simple to implement and results in low computational overhead. However, *Region* based tracking has more parameters for fine-tuning to a particular application memory access pattern. For example, if an application generates only a single big function followed by a memory block of data, *Region* code tracker can capture the page layout at lower memory cost than the *Bitmask* code tracker.

7.3 Applicability to Other Instruction Set Simulators

7.3.1 Emulation of Atomic Synchronisation Instructions

Software-only approach to emulation of LL/SC instructions can be adopted by simulators that make use of software caches for guest memory address translation. For example, Simics' [70] Simulator Translation Cache can be repurposed to enforce slow-path handling for writes that target addresses conflicting with ongoing LL/SC sequences. Similarly, QEMU's SoftMMU [13] can be configured to perform additional checks only for write addresses of interest. The contribution presented in this thesis provides a method for scalable and race-free implementation of these data structures.

7.3.2 Memory efficient decode cache

Redesigning the decode cache can be beneficial to many interpreter-based ISSs [68, 26, 25, 2, 53]. Traditionally, most ISSs use PC-indexed decode cache, resulting the the memory inefficiencies due to duplication of decode instructions. Applying the contributions from Chapter 5 can vastly improve cache performance and/or memory utilisation of these systems. A notable case is gem5 [15], that employs an unbounded hashmap in the decode stage that is indexed by instruction encoding. In this context, the contribution of this thesis can be used to cache the fetch stage by extending the fetch entries with a link to the decode entries, akin to the introduced *Indirect* scheme. Furthermore, other ISSs can benefit from employing schemes that share decoded instruction objects by multiple cores.

7.3.3 Improved Detection of Self-Modifying Code

Detection of SMC is improved by reducing the number of false-positives through employing precise code tracking techniques. These techniques focus on avoiding unnecessary code invalidations, and thus, are orthogonal to the ways code invalidation is actually handled. Therefore, the techniques are applicable to systems protecting guest memory using either OS page protection mechanism (e.g. DynamoRio [20], QEMU user-space [13]) or software-based memory translation cache (e.g. Simics [70], QEMU full-system). Code tracking schemes can also be employed by both interpreters and JIT-base ISSs, although the latter will suffer smaller overhead of code registration, as each instruction is ideally JIT-compiled only once. Interpreters can reduce the registration overhead by employing a decode cache and only registering guest instructions with the code tracker during a cache miss.

7.4 Future Work

As more devices become multicore, the importance of efficient tools of multicore cross-architecture simulation will only increase. Furthermore, modern systems are becoming increasingly heterogeneous, leveraging specialised hardware for narrow workloads to provide performance improvements, as benefiting from the Moore's scaling is no longer feasible.

Several improvements can be investigated in the area of the decode cache design. For example, variable size cache could result in an optimal memory consumption adapting to the current working set of the guest application. Adjusting the cache size dynamically could prevent allocating an unnecessarily large cache in case of applications with small instruction footprints. On the other hand, applications with large instruction footprint could up-size the decode cache to prevent cache conflict misses.

This is particularly relevant for multicore simulation, which is susceptible to synchronisation overheads. A dynamically resizing cache could always allocate enough cache entries for multiple cores to share decoded instruction objects without any object invalidations. Another research direction could investigate asymmetric cache designs, which would optimally allocate private and shared decode caches based on the execution pattern of individual cores. For example, data parallel core threads could utilise a shared cache for memory efficiency, while a task parallel guest core could be allocated a private cache for maximum performance.

With increasing performance of embedded and mobile devices, more demanding applications can be supported. Already today, we see mobile devices running complex operating systems and language runtimes, all relying on dynamic code generation. In simulating these platforms, future research can focus on enabling hypervisors to designate special memory regions where data and code can both be accessed efficiently. This could be done through hardware/software co-design leveraging tagged pointers or extending the MMU to indicate special access permissions for guest pages. Furthermore, hardware support could efficiently track instruction addresses in dedicated pages to efficiently differentiate between benign writes and true code modification.

Cross-architecture simulation should also be able to simulate a wildly heterogeneous platform. Modern systems utilise several asymmetric processors, GPUs, DSP chips, and other accelerators. Simulating a full platform with varying hardware components poses a significant challenge to mapping the guest computation faithfully onto the available host hardware. Future work can investigate the topics of computation segmentation and scheduling, heterogeneous processor simulation, and hw/sw co-design. For example, a host platform could configure an available FPGA accelerator to speed up a particular guest component that would be too laborious to simulate in software.

Bibliography

[1] [n. d.]. Liblfds. https://www.liblfds.org

[2] [n. d.]. OpenRISC 1000 Instruction Set Simulator (or1kiss). https://github.com/janweinstock/or1kiss.

[3] 2007. SimIt-ARM. https://simit-arm.sourceforge.net/.

[4] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[5] Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *computer* 29, 12 (1996), 66–76.

[6] L. Albertsson. 2006. Holistic Debugging – Enabling Instruction Set Simulation for Software Quality Assurance. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*. 96–103. https://doi.org/10.1109/MASCOTS.2006.26

[7] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.

[8] Apple. 2020. Apple announces Mac transition to Apple silicon. https://www.apple.com/uk/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon/. [Online; accessed 2-Oct-2023].

[9] David A Bader, Yue Li, Tao Li, and Vipin Sachdeva. 2005. BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE, 163–173.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 1–12.

[11] Tomás Balderas-Contreras. 2000. Interpretive and Non-interpretive Techniques for Instruction-Set Simulation. In *Proceedings of the Sixth Conference on Electrical Engineering (CIE00)*.

[12] Robert C. Bedichek. 1995. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Ottawa, Ontario, Canada) *(SIGMETRICS '95/PERFORMANCE '95)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/223587.223589

[13] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[15] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[16] Igor Bohm. 2013. Speeding up dynamic compilation: concurrent and parallel dynamic compilation. (2013).

[17] Igor Böhm, Tobias JK Edler von Koch, Stephen C Kyle, Björn Franke, and Nigel Topham. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. *ACM SIGPLAN Notices* 46, 6 (2011), 74–85.

[18] Igor Böhm, Björn Franke, and Nigel Topham. 2010. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 1–10.

[19] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. 2004. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 12 (2004), 1625–1639. https://doi.org/10.1109/TCAD.2004.836734

[20] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[21] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. 2006. Thread-shared software code caches. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp.

[22] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for cross-isa system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 117–128.

[23] Wei Chen, Zhiying Wang, Hongyi Lu, Li Shen, Nong Xiao, and Zhong Zheng. 2009. A Hardware Approach for Reducing Interpretation Overhead. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, Vol. 1. 98–103. https://doi.org/10.1109/CIT.2009.104

[24] Ming-Chao Chiang, Tse-Chen Yeh, and Guo-Fu Tseng. 2011. A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 593–606.

[25] Bill Clarke, Adam Czezowski, Peter Strazdins, et al. 2002. Implementation aspects of a SPARC V9 complete machine simulator. (2002).

[26] Bob Cmelik and David Keppel. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. 128–137.

[27] Intel Corporation. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf. [Online; accessed 8-Aug-2024].

[28] Emilio G Cota, Paolo Bonzini, Alex Bennée, and Luca P Carloni. 2017. Cross-ISA machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 210–220.

[29] Emilio G Cota and Luca P Carloni. 2019. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 74–87.

[30] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 33–48.

[31] D. Dechev. 2011. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*. 158–167. https://doi.org/10.4108/icst.collaboratecom.2011.247161

[32] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 166–180.

[33] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (2002), 255–271.

[34] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. 2011. PQEMU: A parallel system emulator based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 276–283.

[35] Tobias JK Edler von Koch and Björn Franke. 2013. Limits of region-based dynamic binary parallelization. *ACM SIGPLAN Notices* 48, 7 (2013), 13–22.

[36] EEMBC. 2016. 1.0 Multicore Benchmark Software.

[37] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. 2007. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture* 53, 8 (2007), 501–510. https://doi.org/10.1016/j.sysarc.2006.11.003 Architectures, Modeling, and Simulation for Embedded Processors.

[38] Steven Feldman, Pierre Laborde, and Damian Dechev. 2015. A Wait-Free Multi-Word Compare-and-Swap Operation. *Int. J. Parallel Program.* 43, 4 (Aug. 2015), 572–596. https://doi.org/10.1007/s10766-014-0308-7

[39] Vinicius Fulber-Garcia. 2023. Differences Between Simulation and Emulation. https://www.baeldung.com/cs/simulation-vs-emulation. [Online; accessed 15-Nov-2023].

[40] Hui Gao, Yan Fu, and Wim H. Hesselink. 2009. Practical Lock-Free Implementation of LL/SC Using Only Pointer-Size CAS. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering (ICISE '09)*. IEEE Computer Society, Washington, DC, USA, 320–323. https://doi.org/10.1109/ICISE.2009.841

[41] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto. 1995. A hardware-software co-simulator for embedded system design and debugging. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*. 155–164. https://doi.org/10.1109/ASPDAC.1995.486217

[42] Parallels International GmbH. 2023. Parallels Desktop for Mac. https://www.parallels.com/.

[43] Google. 2008. V8 JavaScript. https://v8.dev/.

[44] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.

[45] Irfan Habib. 2008. Virtualization with KVM. *Linux Journal* 2008, 166 (2008), 8.

[46] Sarah Harris. 2023. Emulator vs. Simulator: Demystifying the Differences. `https://sofy.ai/blog/emulator-vs-simulator-simulator-vs-emulator/`. [Online; accessed 15-Nov-2023].

[47] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing binary translation of dynamically generated code. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 68–78.

[48] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[49] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 104–113.

[50] SPARC International Inc and David L Weaver. 1994. *The SPARC architecture manual*. Prentice-Hall Englewood Cliffs, NJ, USA.

[51] Xiao-Wu Jiang, Xiang-Lan Chen, Huang Wang, and Hua-Ping Chen. 2014. A Parallel Full-System Emulator for RISC Architure Host. In *Advances in Computer Science and its Applications*, Hwa Young Jeong, Mohammad S. Obaidat, Neil Y. Yen, and James J. (Jong Hyuk) Park (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1045–1052.

[52] Daniel Jones and Nigel Topham. 2009. High speed CPU simulation using LTU dynamic binary translation. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 50–64.

[53] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet, and Guillaume Savaton. 2009. Instruction set simulator generation using harmless, a new hardware architecture description language. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*. 1–9.

[54] Marco Kaufmann and Rainer G Spallek. 2013. 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*. 33–40.

[55] Chad D Kersey, Arun Rodrigues, and Sudhakar Yalamanchili. 2012. A universal parallel front-end for execution driven microarchitecture simulation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. 25–32.

[56] Taj Muhammad Khan. 2011. *Processor design-space exploration through fast simulation. (Exploration de l'espace de conception de processeurs via simulation accélérée)*. Ph. D. Dissertation. University of Paris-Sud, Orsay, France. `https://tel.archives-ouvertes.fr/tel-00691175`

[57] Frederic Konrad. 2014. [Qemu-devel] Atomic instruction. `https://lists.nongnu.org/archive/html/qemu-devel/2014-07/msg00827.html`. [Online; accessed 5-Oct-2023].

[58] Frederic Konrad. 2015. [Qemu-devel] [RFC 00/10] MultiThread TCG. `https://lists.nongnu.org/archive/html/qemu-devel/2015-01/msg01954.htmlhttps://lists.nongnu.org/archive/html/qemu-devel/2015-01/msg01954.html`. [Online; accessed 5-Oct-2023].

[59] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 1–32.

[60] Martin Kristien, Tom Spink, Brian Campbell, Susmit Sarkar, Ian Stark, Björn Franke, Igor Böhm, and Nigel Topham. 2020. Fast and correct load-link/store-conditional instruction handling in DBT systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3544–3554.

[61] Stephen Kyle, Igor Böhm, Björn Franke, Hugh Leather, and Nigel Topham. 2012. Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation. *ACM SIGPLAN Notices* 47, 5 (2012), 21–30.

[62] Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers* 100, 9 (1979), 690–691.

[63] Leslie Lamport. 2019. A new solution of Dijkstra's concurrent programming problem. In *Concurrency: the works of leslie lamport*. 171–178.

[64] R Lantz. 2008. Fast functional simulation with parallel Embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*. Citeseer.

[65] Ankur Limaye and Tosiron Adegbija. 2018. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 149–158.

[66] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[67] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 388–400.

[68] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu. 2008. ARMISS: An Instruction Set Simulator for the ARM Architecture. In *2008 International Conference on Embedded Software and Systems*. 548–555. https://doi.org/10.1109/ICESS.2008.73

[69] P.S. Magnusson. 1997. Efficient Instruction Cache Simulation And Execution Profiling With A Threaded-code Interpreter. In *Winter Simulation Conference Proceedings,*. 1093–1100. https://doi.org/10.1145/268437.268745

[70] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[71] Victor Matos. 2009. Android environment emulator. *Cleveland State University* 20, 11 (2009).

[72] Cathy May, Ed Silha, Rick Simpson, Hank Warren, and CORPORATE International Business Machines, Inc. 1994. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc.

[73] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[74] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004).

[75] Maged M. Michael. 2004. Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS. In *Distributed Computing*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–158.

[76] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[77] Wai Sum Mong and Jianwen Zhu. 2004. DynamoSim: a trace-based dynamically compiled instruction set simulator. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* IEEE, 131–136.

[78] Daniel Mueller-Gritschneder, Martin Dittrich, Marc Greim, Keerthikumara Devarajegowda, Wolfgang Ecker, and Ulf Schlichtmann. 2017. The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping. In *2017 International Symposium on Rapid System Prototyping (RSP)*. 79–84.

[79] Koh M. Nakagawa. 2021. Project Champollion. `https://ffri.github.io/ProjectChampollion/`. [Online; accessed 15-June-2024].

[80] Ragavendra Natarajan and Antonia Zhai. 2015. Leveraging Transactional Execution for Memory Consistency Model Emulation. *ACM Trans. Archit. Code Optim.* 12, 3, Article 29 (Aug. 2015), 24 pages. https://doi.org/10.1145/2786980

[81] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[82] Oracle. 2023. VirtualBox. https://www.virtualbox.org/.

[83] Sebastian Ottlik, Stefan Stattelmann, Alexander Viehl, Wolfgang Rosenstiel, and Oliver Bringmann. 2014. Context-sensitive timing simulation of binary embedded software. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 1–10.

[84] Andrew Over, Bill Clarke, and Peter Strazdins. 2007. A comparison of two approaches to parallel simulation of multiprocessors. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 12–22.

[85] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*.

[86] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. 1050–1055.

[87] Daniel Christopher Powell and Björn Franke. 2009. Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. 315–324.

[88] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[89] Ravi Rajwar and James R Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 294–305.

[90] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and N. Dutt. 2003. An efficient retargetable framework for instruction-set simulation. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*. 13–18. https://doi.org/10.1109/CODESS.2003.1275249

[91] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. 2003. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of the 40th Annual Design Automation Conference* (Anaheim, CA, USA) *(DAC '03)*. Association for Computing Machinery, New York, NY, USA, 758–763. https://doi.org/10.1145/775832.776026

[92] Alvise Rigo, Alexander Spyridakis, and Daniel Raho. 2016. Atomic Instruction Translation Towards A Multi-Threaded QEMU.. In *Proceedings 30th European Conference on Modelling and Simulation*. European Council for Modeling and Simulation, United Kingdom, 587–595.

[93] Tom Spink, Harry Wagstaff, and Björn Franke. 2016. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–25.

[94] StackOverflow. [n. d.]. Simulator or Emulator? What is the difference?, howpublished = "https://stackoverflow.com/questions/1584617/simulator-or-emulator-what-is-the-difference", year = 2023, note = "[Online; accessed 15-Nov-2023]".

[95] Timo Stripf, Ralf Koenig, and Juergen Becker. 2012. A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 21–26.

[96] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2003. 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*. 312–323.

[97] Synopsys. 2023. Synopsys Processor Solutions. https://www.synopsys.com/designware-ip/processor-solutions.html.

[98] Synopsys. 2023. Virtual Prototyping. https://www.synopsys.com/verification/virtual-prototyping.html.

[99] Synopsys. 2023. Virtual Prototyping. https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html.

[100] Christopher Thompson, Miles Gould, and Nigel Topham. 2013. High speed cycle approximate simulation for cache-incoherent MPSoCs. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 88–95.

[101] Nigel Topham and Daniel Jones. 2007. High speed CPU simulation using JIT binary translation. In *Workshop on Modeling, Benchmarking and Simulation (MOBS)*. Citeseer.

[102] Apple (via Web Archive). 2011. Rosetta. https://web.archive.org/web/20110107211041/http://www.apple.com/rosetta/. [Online; accessed 2-Oct-2023].

[103] Advanced Micro Devices Inc (via Web Archive). 2018. Virtualization Solutions. https://web.archive.org/web/20180110195706/http://www.amd.com/en-us/solutions/servers/virtualization. [Online; accessed 5-Oct-2023].

[104] Intel Corporation (via Web Archive). 2019. Intel Virtualization Technology. https://web.archive.org/web/20190808112711/http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html. [Online; accessed 5-Oct-2023].

[105] VMware. 2023. Server Virtualization and Consolidation. https://www.vmware.com/uk/solutions/consolidation.html.

[106] Huang Wang, Chao Wang, and Huaping Chen. 2015. XEMU: A cross-ISA Full-system Emulator on Multiple Processor Architectures. *Int. J. High Perform. Syst. Archit.* 5, 4 (Nov. 2015), 228–239. https://doi.org/10.1504/IJHPSA.2015.072853

[107] Kun Wang, Yu Zhang, Huayong Wang, and Xiaowei Shen. 2008. Parallelization of IBM mambo system simulator in functional modes. *ACM SIGOPS Operating Systems Review* 42, 1 (2008), 71–76.

[108] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. 2015. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. *ACM SIGPLAN Notices* 50, 7 (2015), 53–64.

[109] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 213–222.

[110] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual. Volume I: User-Level ISA*. RISC-V Foundation. `https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view` Document Version 20191214-draft..

[111] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.

[112] Quan Xiao, Jian Wu, Zhuo Lin, Li Kong, Jing Liu, Lei Deng, Shuyu Li, Tao Zhang, and Fangquan Lin. 2012. The study of binary decoding cache for instruction-set simulation. In *International Conference on Automatic Control and Artificial Intelligence (ACAI 2012)*. 2187–2190. `https://doi.org/10.1049/cp.2012.1433`

[113] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V Kalé. 2004. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 78.