From High Level Architecture Descriptions to Fast Instruction Set Simulators

Harry Wagstaff

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2015

Abstract

As computer systems become increasingly complex and diverse, so too do the architectures they implement. This leads to an increase in complexity in the tools used to design new hardware and software. One particularly important tool in hardware and software design is the Instruction Set Simulator, which is used to prototype new architectures and hardware features, verify hardware, and test and debug software. Many Architecture Description Languages exist which facilitate the description of new architectural or hardware features, and generate a tools such as simulators. However, these typically suffer from poor performance, are difficult to test effectively, and may be limited in functionality.

This thesis considers three objectives when developing Instruction Set Simulators: performance, correctness, and completeness, and presents techniques which contribute to each of these. Performance is obtained by combining Dynamic Binary Translation techniques with a novel analysis of high level architecture descriptions. This makes use of partial evaluation techniques in order to both improve the translation system, and to improve the quality of the translated code, leading a performance improvement of over 2.5x compared to a naïve implementation.

This thesis also presents techniques which contribute to the correctness objective. Each possible behaviour of each described instruction is used to guide the generation of a test case. Constraint satisfaction techniques are used to determine the necessary instruction encoding and context for each behaviour to be produced. It is shown that this is a significant improvement over benchmark-driven testing, and this technique has led to the discovery of several bugs and inconsistencies in multiple state of the art instruction set simulators.

Finally, several challenges in ‘Full System’ simulation are addressed, contributing to both the performance and completeness objectives. Full System simulation generally carries significant performance costs compared with other simulation strategies. Crucially, instructions which access memory require virtual to physical address translation and can now cause exceptions. Both of these processes must be correctly and efficiently handled by the simulator. This thesis presents novel techniques to address this issue which provide up to a 1.65x speedup over a state of the art solution.

Lay Summary

In the modern world, computers are everywhere - from Desktop PCs, to Smartphones, to fridges and microwaves. In the last decade, the number of different types of computer processor has exploded. However, designing a new computer processor is extremely complex. One tool typically used is the 'Instruction Set Simulator', which allows programs designed for one type of system (such as smartphones) to be run on a different type of system (such as a Desktop PC). Usually these systems are known as the 'Guest' and 'Host', respectively. These simulators can be created from scratch, or a description of the desired processor (an 'Architecture Description') can be used to generate a simulator. However, these generated simulators are usually slow, are difficult to test, and may not include useful simulation features.

This thesis presents techniques which allow improved simulators to be generated. These simulators provide improved performance, are easier to test, and provide enhanced functinoality versus what might otherwise be available. Improved performance is obtained using new DBT (Dynamic Binary Translation) techniques. DBT involves translating the simulated program into a new program which can run directly on the host. DBT is already well known, but usually generated simulators do not use DBT or use it extremely inefficiently. This thesis presents techniques which can be used to perform DBT much more efficiently, in order to generate a much faster simulator. Also presented is a novel technique to create tests for the Architecture Description and the generated simulator. Each possible behaviour of each instruction in the description is analysed and used to create a test. This is shown to be a very effective testing method, and several bugs are discovered in popular simulators. Finally, this thesis presents a new technique to speed up 'Full System' simulation. This style of simulation can be very slow, but by handling memory reads and writes more efficiently, the performance of a simulator can be significantly improved.

Acknowledgements

I would like to thank my PhD. supervisor, Dr. Björn Franke, for his continual support and guidance throughout my research, as well as invaluable proof-reading and feedback. Thanks also go to my second supervisor, Prof. Nigel Topham, for the many valuable discussions, and encouragement, that he has provided.

I would also like to acknowledge the many friends and colleagues I have encountered during my time at the University of Edinburgh, both as an undergraduate and during my time at ICSA and in the PASTA office. These include (but are in no way limited to) Oscar Almer, Matthew Bielby, Igor Böhm, Tobias Edler von Koch, Marco Elver, Miles Gould, Stephen Kyle, Dan Powell, Volker Seeker, Tom Spink, and Christopher Thompson. I'm sure that no matter where our careers and travels might take us, working together has been a singular experience.

Finally, I would like to thank my family for the endless support and encouragement that they have provided, and for instilling in me the ambition to aim high, and the drive and ability to get there.

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 papers:

- **H. Wagstaff**, M. Gould, B. Franke, and N. Topham, "Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated From a High-Level Architecture Description", in *Proceedings of the 50th Annual Design Automation Conference* (DAC'50), 2013
- T. Spink, **H. Wagstaff**, B. Franke, and N. Topham, "Efficient Code Generation in a Region-Based Dynamic Binary Translator", in *Proceedings of the 2014 ACM SIGPLAN/SIGBED conference on Languages, Compilers, Tools and Theory for Embedded Systems* (LCTES'14), 2014
- **H. Wagstaff**, T. Spink, and B. Franke, "Automated ISA Branch Coverage Analysis and Test Case Generation for Retargetable Instruction Set Simulators", in *Proceedings of the 2014 International Conference on Compilers, Architectures and Synthesis of Embedded Systems* (CASES'14), 2014
- T. Spink, **H. Wagstaff**, B. Franke, and N. Topham, "Efficient Asynchronous Interrupt Handling in a Full-System Instruction Set Simulator", under review for the *2015 ACM SIGPLAN/SIGBED conference on Languages, Compilers, Tools and Theory for Embedded Systems* (LCTES'15), 2015
- T. Spink, **H. Wagstaff**, B. Franke, and N. Topham, "Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator", under review for the *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation* (SAMOS'15), 2015

(*Harry Wagstaff*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In today's connected world, rapid prototyping and development are becoming ever more important in the race to bring embedded products to market before competitors. Not only are systems becoming larger, as multi-core technologies continue to grow in the embedded space, but they are also becoming more heterogeneous, both in terms of heterogeneity within a single system (such as the ARM big.LITTLE [43] platforms) and, as custom ASICs and accelerators continue to grow in popularity, across systems.

Effective tools are a vital part of the infrastructure required to design and implement new computer systems and components. Much research has been done in improving compilers, designing new programming languages, and improving run time and operating systems for embedded platforms.

However, as embedded systems have become more complex, the effort required to debug these systems, as well as the software built on top of them, has increased exponentially. When developing a new embedded product or platform, it is no longer feasible to wait until silicon is available before beginning software development and debugging. RTL simulation and FPGA prototyping is possible, but these require that system development be complete or almost complete, and have significant time and resource costs. Additionally, these techniques often do not allow improved observability of the system compared with debugging directly on the target platform.

Simulation tools are able to bridge this gap between hardware specification and implementation, allowing both improved hardware prototyping, such as experimenting with new instruction set extensions and application specific accelerators, and also allowing software development and debugging to begin sooner. They also frequently offer greater visibility than FPGA or silicon implementations of the same platform, as event counters and debugging features can be inserted into the simulator arbitrarily, easily, and often with little runtime performance cost.

These tools are used in many areas in academic computer science, such as in architectural and microarchitectural design space exploration [52, 45, 103] and design of new cache coherence protocols [33]. However, the market for simulation tools is also growing in the industrial sector. Most CPU vendors provide some kind of simulation platform, such as ARM's DS-5 [39], but simulation is also increasingly important in real-world software development [98, 89], system verification [91], lower-power system design [28], and many other areas.

From a high level perspective, a simulator is a software model of a computer system. The objective is to examine the behaviour of software running on a *guest* system, using a model on a *host* system. The *guest software* may be a single application, or a full operating system, and the simulation may vary in detail - *Functional* simulation seeks to replicate the expected behaviour of the *guest*, while *Cycle-Accurate* simulation provides a detailed microarchitectural model in order to predict performance or energy usage.

Instruction Set Simulation is a subset of this field, mainly geared around simulating processing units such as CPUs, microcontrollers, and specialised processing units such as Digital Signal Processors (DSPs). This is in contrast to e.g. circuit simulations (which might simulate the complete system as an electronic circuit) or simulation of complex accelerators such as the fixed-function portions of Graphics Processing Units (GPUs) (although many accelerators can be modelled as processing units). This thesis is mainly addressing problems around Instruction Set Simulators, rather than physical circuit or accelerator simulation.

Simulation platforms are clearly useful tools in embedded system development. However, they also have unique requirements and pose additional challenges. The three main objectives when developing a simulator - *completeness*, *correctness*, and *performance* - are often in tension and must be traded against each other. Simulators also require specific implementation skills and a breadth of knowledge covering both low level architectural detail and high level software engineering.

This thesis presents methods for separating out these knowledge and skill requirements, by separating the description of the system to be simulated, from the implementation of the simulator. The system designer provides a description of the system to be simulated, and receives an instrumentable and high performance simulator.

1.2 Motivation

Instruction set simulation technologies have advanced significantly in recent years. In order to boost performance, several technologies such as DBT (Dynamic Binary Translation) have been integrated into many simulators. DBT enables improved simulation performance, by selectively translating parts of the target program into instructions which can be executed directly on the host machine. DBT has also been used in runtime environments, such as Java and Microsoft's CLR, and similar technologies are used in modern web browsers in order to accelerate the execution of JavaScript.

While Dynamic Binary Translation is one of the key technologies used to accelerate simulation, it is also one of the most challenging to implement correctly and efficiently. Simulation platforms such as QEMU, which support DBT, provide high simulation speeds, but retargeting these platforms to new guest architectures is a significant challenge. Conversely, systems such as LISA and ArchC provide easy retargetability but compromise on performance, producing a simulator which is of limited use when large or long running applications are to be simulated.

It would be desirable to have a simulator framework which is easily retargetable, but which is still competitive with a hand tuned simulator.

1.3 Contributions

This thesis seeks to outline techniques which contribute to each of our three simulation objectives (*correctness*, *completeness*, and *performance*). Three such techniques are presented.

First, this thesis covers the generation of many of the simulator components from a high level description. This contributes primarily to our *performance* objective, as many optimisations can only be efficiently performed algorithmically, and from a high level description. Rather than require that the user hand-write complex Dynamic Binary Translation (DBT) components, a method is presented which allows a high-level architectural description to be processed into a high-speed DBT frontend using Partial

Evaluation techniques. Also presented are novel contributions in DBT code generation, particularly in control flow handling and memory accesses. This thesis then demonstrates the performance gains obtained using these techniques, by implementing them on top of the state of the art Arcsim simulation platform, compared with both a naïve approach, and against the high speed QEMU instruction set simulator.

Secondly, this thesis presents a technique for generating tests for generated simulators. First, it is demonstrated that even large and complex benchmark workloads are insufficient for testing the correctness of instruction set simulators as they leave a majority of the instruction space uncovered. Then the high level description is analysed, and the extracted information used to generate targeted and complete tests for the described architecture. This contributes to our *correctness* objective, as we are able to use these tests to determine whether the architectural behaviour of our system is functionally correct at an instruction-by-instruction level. These techniques are further applied to the architectural model used elsewhere in this thesis in order to show its correctness, as well as two modern simulation platforms, and discover several bugs in each when compared against a reference hardware platform.

Finally, a novel technique for performing address translations in the context of a virtual memory system is introduced. Rather than using a cache-based approach, as is common, small machine code fragments which implement the functionality of the MMU for each virtual page, including address translation and permission/privilege checking, are generated. This contributes to both our *completeness* objective, as it allows us to perform ‘full-system’ simulation (where both user and kernel mode execution is simulated), and the *performance* objective, as it improves the efficiency of address translations, which typically contribute significantly to simulation runtime.

1.4 Thesis Structure

This thesis is organised as follows:

Chapter 2 describes simulation techniques such as DBT in more detail. The various components of Instruction Set Simulation are covered, including Dynamic Binary Translation, and the existing work in these fields is discussed. Several existing approaches to simulator generation are discussed, as well as the testing of such simulators.

Chapter 3 examines the existing artefacts used by this thesis, as well as the performance evaluation methodology used in Chapters 4 and 6. Artefacts include the Arcsim simulator and the LLVM compiler framework on which the presented contributions are built, and the SPEC and EEMBC benchmark suites used for evaluation.

Chapter 4 looks at techniques for generating high performance DBT systems from high level Architecture Descriptions. An overview of the GenC ADL, as well as how the various simulation components are generated, is presented. The presented Partial Evaluation technique for DBT module generation provides a large performance improvement compared to a naïve implementation. This chapter is based on material published in [107].

Chapter 5 shows techniques for confirming the accuracy of high level Architecture Descriptions. By analysing the possible control flow paths through each instruction description, a comprehensive test suite can be generated. By running this test suite in simulation, and on a reference platform, the accuracy of the high level Architecture Description can be confirmed. This chapter is based on material published in [106].

Chapter 6 extends the previous techniques into ‘full-system’ simulation, and presents methods for tackling many of the additional challenges which this poses. A novel technique for performing memory translations and accesses in a full system context is presented, leading to a significant speedup over state of the art techniques.

Finally, Chapter 7 concludes this thesis, summarising contributions, analysing the thesis, and proposing future work.

Chapter 2

Background & Related Work

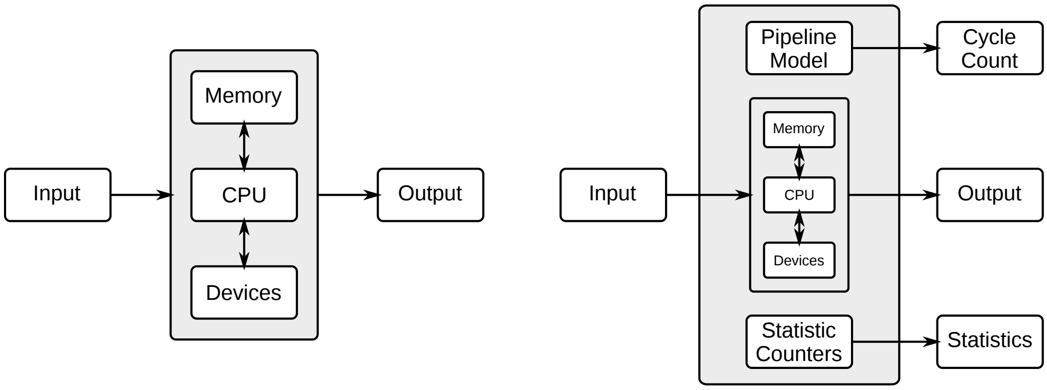
2.1 Introduction

This chapter seeks to introduce the reader to the many techniques used to construct modern Instruction Set Simulators. In particular, many of the techniques discussed later in this thesis are introduced. The ‘architectural’ view of a simulator (that is, what is happening from the perspective of the user) is briefly introduced. Methods to actually perform the simulation, and improve simulation performance, are then described.

The existing literature on each field of Instruction Set Simulation relevant to this thesis is also covered, particularly work on general simulation, Dynamic Binary Translation, and automatic generation of simulators and simulator components. Some work in related fields is also covered where relevant. There is significant overlap in the fields of Dynamic Binary Translation, Instruction Set Simulation, Virtualisation, and Managed Language Runtimes.

How this chapter is structured

- An overview of modern techniques for fast instruction set simulation
- A description of how simulators can be generated using a high level ADL (Architecture Description Language)
- Existing literature and related work on both of the above topics



(a) (b)

Figure 2.1: In (a), a computer system takes input (including a program), executes instructions in one or more CPUs, communicates with memory and external devices, and produces output. In simulation (b), the same input and output are processed. The simulator may also provide additional features such as detailed performance models or collect statistics/profiling information.

2.2 Overview of Simulation

As stated above, a simulator is a model of the *guest system*, designed to run on a *host system*. That model usually includes, at the very least, the expected functional behaviour of the *guest system*. An Instruction Set Simulator typically replicates the expected functional behaviour of the system, and may also include features to predict the running time of an application on the guest system. An Instruction Set Simulator might also include instrumentation or debugging features in order to aid software development (see Figure 2.1).

Instruction Set Simulators have a wide variety of uses. The most direct use of such simulators is in hardware and software development, both commercially and academically. Many hardware features such as new pipeline designs, cache layouts, cache coherence protocols etc. are first prototyped using simulators. Simulation is also frequently used to assess the effectiveness of new software techniques such as compiler optimisations, as simulators typically allow for greater observability than real hardware.

A common use of simulation is in Design Space Exploration. Simulations of a large number of configurations of a particular system are performed, and the results used to guide further research and development. The large number of configurations examined typically means that it would be infeasible to physically construct all of the tested systems or to use FPGAs to test all of the configurations. A large number of simulations can be quickly set up and executed on a compute cluster.

Simulation has also found extensive use in the video games industry in order to provide backwards compatibility. Many video games for consoles make extensive use of ‘tricks’ and specific timing behaviours in order to maximise performance, meaning that highly detailed models are required in order to obtain correct behaviour. A notable example of this is the backwards compatibility features present in some models of Sony’s PlayStation 3 console. While early versions of the Playstation 3 supported previous-generation PlayStation 2 games using direct hardware support (i.e., the PlayStation 3 consoles contained an almost complete PlayStation 2 chipset), later versions switched to a software solution in order to save costs. However, the simulation is not perfect, meaning that many games produce bugs, suffer from poor performance, or simply do not work.

A more successful example of backwards compatibility using simulation technologies is Apple’s Rosetta [2]. Based on QuickTransit by Transitive [105], Rosetta allowed older PowerPC Mac applications to run on newer x86 based machines. In contrast with Sony’s PlayStation 2 emulation, Rosetta performs so-called ‘user-mode’ Dynamic Binary Translation, meaning that many of the behaviours of the *guest* system, such as precise exceptions and virtual memory translations, do not have to be faithfully reproduced, greatly aiding performance.

At a bare minimum, an Instruction Set Simulator must be capable of decoding and executing instructions. Although a wide range of techniques exist for performing these actions, some may not be suitable or applicable to particular pairs of *guest* and *host*, or may not work well with other techniques in use. The rest of this chapter will discuss these techniques, giving a brief overview of what is involved, and how they have been explored and examined in the literature.

2.3 Instruction Decoding

At a basic level, Instruction Decoding is the process of taking a region of memory representing one or more *guest* instructions, and determining what kind of instructions they

```
01 c2 : addl %eax, %edx
```

(a) An example x86 instruction with no prefixes.

```
66 41 01 c2 : addw %ax, %r9w
```

(b) The same x86 instruction again, but with several prefixes. The prefixes have changed the destination register of the instruction, as well as the data width of the operation. This type of ‘stateful’ instruction complicates the decode process.

```
e5912010 : ldr r2, [r1, #16]
```

(c) Here a single ARM instruction can be seen.

```
2010 : movs r0, #16
e591 : b -0x4da
```

(d) If the binary representation of the above ARM instruction is decoded in Thumb mode, two completely different instructions are produced.

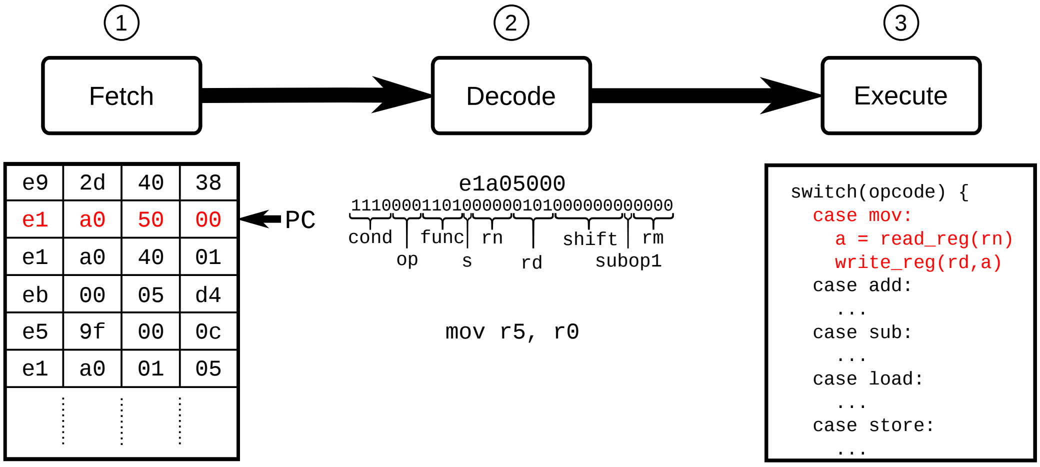
Figure 2.2: Examples of difficulties in decoding instructions. How an instruction is decoded can depend on context, such as with instruction prefixes ((a) and (b)) or system state ((c) and (d)).

are, and what their operands are. This can typically be achieved using bit manipulation operations and lookup tables.

Some care needs to be taken when decoding instructions which are in any way stateful. A classic example of an Instruction Set containing stateful instructions is x86: an instruction can consists of one or more prefixes (which may change the nature of the instruction, or the nature of its operands), followed by an opcode, a ‘ModR/M’ byte, a ‘SIB’ byte, and finally any immediates or offsets, which vary in length according to the instruction. Many of these components of an instruction are optional and their presence depends on the nature of previous parts of the instruction (see Figures 2.2a and 2.2b).

A simpler but still important example might be the ARM and Thumb instruction sets. Modern processors implementing the ARMv5 instruction set and above are able to switch into the Thumb (and later, Thumb-2) instruction set, which provides a more compact encoding for situations where code size is critical. In this case, the decoding of a particular region of memory depends on the state of the CPU at the time when it is decoded (see Figures 2.2c and 2.2d).

For orthogonal, non-stateful instruction decoders, automatically generated decision trees are typically used. This has been investigated by Fournel [36] and Qin [82]. Decoding of x86 instructions has been discussed by Krishna [56], who assume that much



e9	2d	40	38
e1	a0	50	00
e1	a0	40	01
eb	00	05	d4
e5	9f	00	0c
e1	a0	01	05

```
switch(opcode) {
  case mov:
    a = read_reg(rn)
    write_reg(rd,a)
  case add:
    ...
  case sub:
    ...
  case load:
    ...
  case store:
    ...
```

Figure 2.3: Interpretation is an almost direct implementation of the Fetch-Decode-Execute cycle. Instructions are fetched one at a time from memory ①, decoded ②, and then executed ③, with control flow instructions updating the PC, and thus the address of the next instruction to be fetched.

of the ‘stateful’ nature has already been parsed out of the instruction, and that the many possible x86 instruction prefixes are collapsed into a single byte.

The Gem5 simulation framework includes a partially generated x86 instruction decoder [110]. Here, a state machine is used first to filter entire instructions out of the instruction stream. The instructions are then classified according to their general type (i.e., whether they are microcoded). Parts of the decoder are instruction implementation are generated from an ISA description. However, significant amounts of the generation flow are tailored to x86 and it is not clear whether these would be able to support other mixed length instruction sets.

GDSL [96] provides a complete x86 decoder implementation, describing the instruction set using an abstract grammar. Monads are used to provide a grammar which is stateless, but which decodes the stateful x86 instruction set.

2.4 Interpretation

Interpretation is the simplest execution model for a simulator. After instructions are fetched and decoded, the opcode of the instruction is looked up in a switch statement or jump table and the instruction implementation is executed directly, as shown in Figure

2.3. The next instruction is then decoded and fetched, and the process repeats until the simulation ends.

While the simplicity of an interpreter is something of an advantage during implementation, it also means that performance is typically limited. Each executed *guest* instruction must be fetched and decoded (although caches can be used to accelerate this process) individually, and then the correct behaviour for the instruction must be selected and executed. This presents no opportunity for intra- or inter-instruction optimisations (as will be discussed below) and also gives poor *host* instruction cache performance, as the instruction implementations are scattered in memory.

Some work has been done on accelerating interpreter performance using techniques such as interpreter threading and specialisation or partial evaluation. However, threading is typically only suitable for bytecode-based systems (which have instructions with few or no explicit operands). A high-profile example of a simulation platform featuring threaded interpretation is SimIcs [67], which also makes use of various automated generation techniques.

Instruction specialisation involves generating multiple implementations of each instruction to be interpreted, with each one tuned to support a particular execution path through the interpreted instruction. For example, in the ARM ISA, separate versions of arithmetic instructions might be generated for the flag-setting and non-flag-setting versions of each instruction.

Although this can improve simulation throughput by reducing the amount of hard-to-predict control flow, the large amount of extra host machine code exacerbates the *host* instruction cache pressure. Additional interpreter cases can also increase decode complexity, as the correct specialisation must be selected, although this can usually be amortized using decode caching (such as in [104]). In an interpretive simulation environment, instructions are fetched and decoded one at a time. However, the same instruction might be fetched and decoded many times if it is part of a hot loop or frequently called function. Decode caching seeks to reduce the costs associated with decoding an instruction (including determining the type of the instruction and extracting the interesting fields from the instruction) by caching the results of this operation.

2.5 Static Binary Translation

Static Binary Translation (also known as ‘Compiled Instruction Set Simulation’) attempts to convert a binary program compiled for the *guest* instruction set, into one which will execute directly in the *host* environment. Some analysis is done in order to extract control flow information from the binary program, in order to permit optimisations on the translated binary.

There are several major problems facing Static Binary Translation. Firstly, arbitrary and indirect control flow (such as jump tables and ‘return’ statements, which are typically implemented as indirect branches) mean that an accurate mapping between each *guest* instruction, and their *host* equivalents, must be maintained. Secondly, any program making use of self modifying code, or which generates and executes new code at runtime, cannot be supported purely by Static Binary Translation.

One of the major concerns with static binary translation is the amount of time spent translating and compiling the target binary. This can be a significant overhead, especially considering that the entire binary must be fully translated and compiled before any simulation can begin (in contrast with interpretive or Dynamic Binary Translation-based simulation). Obsim [29] seeks to reduce these compilation time overheads by supporting partial and incremental compilation, rather than requiring that the full target binary be translated each time a small change is made. Some performance enhancements are also described, including the optimisation of various control flow structures.

Reshadi [87] seeks to improve the flexibility of Static Binary Translation by introducing elements typically found in ‘dynamic’ (Interpretive and Dynamic Binary Translation-based) simulators. The key idea is to generate a simulator module which is tailored to the target binary or binaries. The instruction types present in the target binary are identified statically, and an interpreter is generated which is specialised to the binaries to be simulated.

A more frequent approach than full Static Binary Translation is to combine some static analysis with either interpreted simulation or a Dynamic Binary Translation approach. This is typically combined with some performance modelling or prediction in order to produce a high speed cycle-approximate simulation. Performance modelling simulators will be discussed below, in Section 2.8.

Bansal [8] presents a superoptimisation-based technique. First, sequences of *guest* instructions are automatically extracted from target programs. Then, functionally equivalent sequences of *host* instructions are automatically generated using a superoptimiser. Although the presented work is implemented in a static binary translator, the paper claims that transferring the techniques to a DBT system would be straightforward.

2.6 Dynamic Binary Translation

While Static Binary Translation seeks to translate *guest* instructions to *host* instructions in an offline context, Dynamic Binary Translation does this translation in an *online* context, that is, at runtime. This translation is typically achieved using JIT Compilation techniques. Dynamic Binary Translation is able to achieve very high *guest* instruction throughput, making it very useful for high-speed simulation.

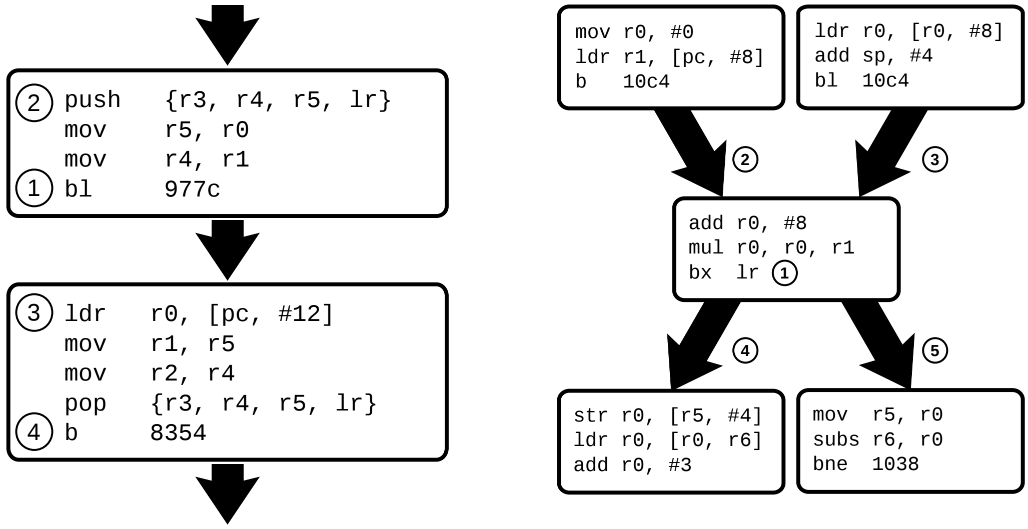
There are several main approaches to Dynamic Binary Translation. One of the most popular is to translate straight-line sections of *guest* instructions, either basic blocks or traces. However, this presents limited opportunities for control flow optimisation and loop optimisations. Region based translation involves translating large regions of *guest* instructions are translated, including complex control flow structures. This provides improved simulation speed, although translation throughput typically suffers.

Once a translation is performed, the translated *host* instructions are stored for future use, typically in a one- or two-level Translation Cache. In the event that one or more *guest* instructions are modified, any translations which cover those instructions will have to be abandoned.

2.6.1 Block Based Translation

The simplest DBT approach typically used is to translate one basic block at a time from the *guest* ISA to the *host* ISA. A basic block is defined as a code region with a single entry and a single exit (Figure 2.4a). A block may have multiple predecessors and successors (Figure 2.4b). Basic Block Translation can be done with very little overhead, as discovering the extents of a basic block can be done by simply looking for control flow instructions.

Once the beginning and end of the basic block is identified, the instructions are translated one at a time. The method for translating the instructions varies, but usually involves emitting some form of Intermediate Representation (IR) such as LLVM bitcode, or QEMU's Tiny Code Generator IR. Once the block is fully translated into IR,



```
② push   {r3, r4, r5, lr}
  mov    r5, r0
  mov    r4, r1
① bl     977c
```

```
③ ldr   r0, [pc, #12]
  mov   r1, r5
  mov   r2, r4
  pop   {r3, r4, r5, lr}
④ b     8354
```

```
mov r0, #0
ldr r1, [pc, #8]
b   10c4
```

```
ldr r0, [r0, #8]
add sp, #4
bl  10c4
```

```
add r0, #8
mul r0, r0, r1
bx  lr ①
```

```
str r0, [r5, #4]
ldr r0, [r0, r6]
add r0, #3
```

```
mov  r5, r0
subs r6, r0
bne  1038
```

(a) Two basic blocks connected by a control flow instruction ①. Each block has a single entry point ②, ③ and exit point ①, ④.

(b) Basic blocks can have multiple predecessors ②, ③. Indirect branches ① can also cause a block to have multiple successors ④, ⑤.

Figure 2.4: Basic blocks are one of the fundamental structures used when analysing the control flow graph for a program.

some optimisations may be performed, and then the IR is translated into *host* instructions.

A Block Based Translation system can be improved in several ways, such as by *chaining* blocks together, so that *host* control flow is able to move directly from one translated block to the next, without performing a lookup in the Translation Cache. This technique is used in QEMU [14]. HQEMU extends QEMU to use LLVM as a JIT compiler [48]. HQEMU also performs translation in parallel, performing costly optimisation operations away from the critical path of the simulator.

Brandner et al. [19] also use LLVM as a JIT compiler. However, instead of extending QEMU, *guest* basic blocks are translated directly into individual LLVM functions. The simulator is cycle accurate (although no accuracy figures are given) and the LLVM translation system is generated from an architectural description.

2.6.2 Trace Based Translation

While Block Based Translation can be effective, basic blocks are typically quite small (For example, SPEC2000 has an average block length of 4.5 [44] when compiled for

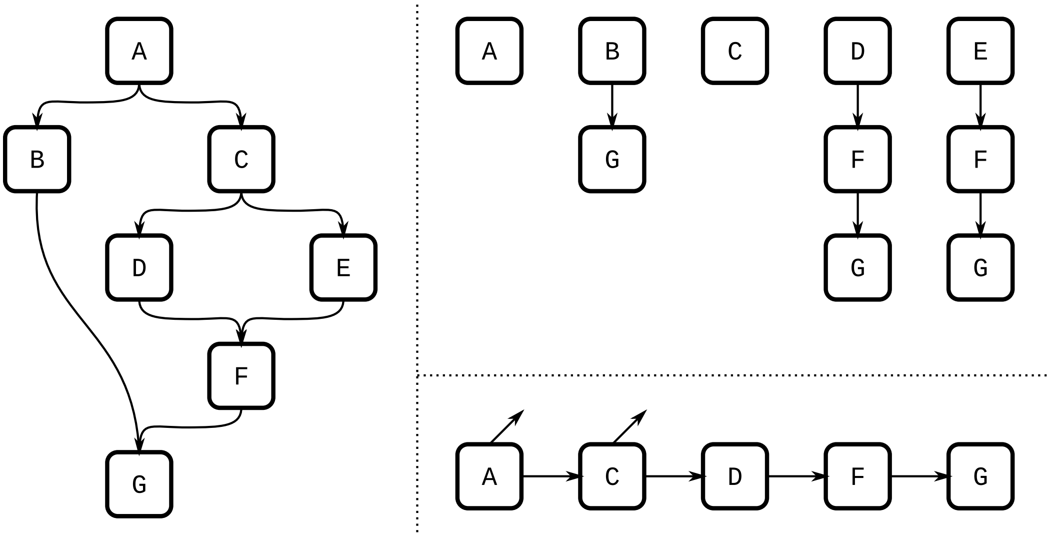


Figure 2.5: On the left a simple control flow graph can be seen. On the top right, the extracted simple traces. Notice how several basic blocks appear in multiple traces. At the bottom right there is an example of a more complex trace with multiple exit points. Progression through the trace at `A` and `C` depends on each branch taking the 'correct' path.

the ARM architecture). This severely inhibits opportunities for optimisation, and also means that translation lookup may become a bottleneck if it is not implemented efficiently. Trace Based Translation seeks to solve these problems. A trace is defined similarly to a block, except that a trace may have multiple exit points (Figure 2.5).

The most primitive form of trace is one which groups together multiple basic blocks, connected by unconditional control flow (i.e., each block, except for the first and last, has only one predecessor and one successor). More complex schemes attempt to include blocks which end in conditional control flow. When encountering conditional or indirect control flow, some profiling may be performed in order to evaluate which branch target should be included in the trace.

Trace Based Translation exposes many more opportunities for optimisation than a block based approach. However, it is not without drawbacks. In particular, it can impose a large memory cost, as each *guest* basic block may be included in many traces. As with Block Based Translation, traces may be chained together in order to provide a performance benefit. DynamoSim is an example of a trace based instruction set simulator [72].

Several optimisations are described in [86], including grouping, ordering and optimising groups of contiguous basic blocks. Here, multiple basic blocks are translated into a single trace provided that there are only direct, statically calculable branches

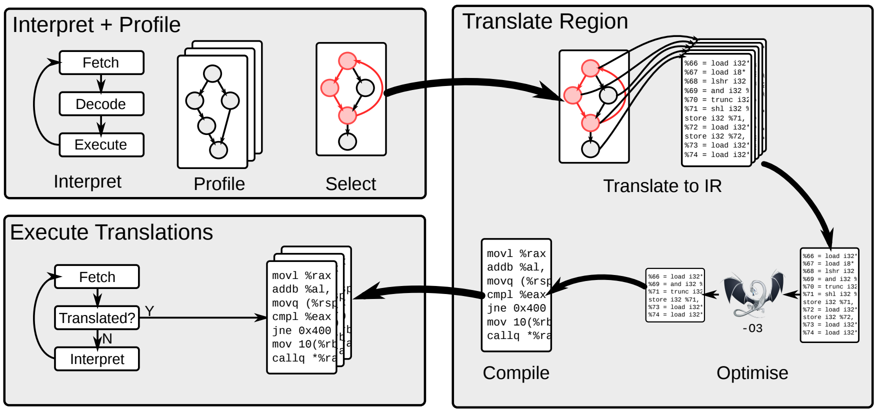


Figure 2.6: Region Based Translation translates entire regions, typically *guest* pages. The regions are first executed using a tracing interpreter, in order to gather direct and indirect control flow information. The regions are then translated, along with a dispatcher which allows execution to begin at the correct position in the region.

between each block. Previously translated blocks may also be retranslated into, and optimised as, a single trace. This enables more aggressive optimisations to be applied.

2.6.3 Region Based Translation

In contrast with Block Based- and Trace Based Translation, which translate only straight line sections of code, Region Based Translation involves extracting ‘regions’ of the *guest* program in order to form a control flow graph, and then translating these regions, including their loops and other complex control flow structures. This approach has the immediate advantage that cross-basic-block and loop optimisations can be used in order to improve the run time performance of the translated *host* code.

However, these additional optimisation opportunities come at a cost. The region profiling step, which includes forming control flow graphs, can be expensive. Executing the additional loop optimisations also comes at a cost, as they can be expensive to perform. It is particularly important to balance the cost of each optimisation against the performance benefit it provides.

Unlike Block Based and Trace Based Translation systems, translated Regions are typically not individually chained when simulating a 32-bit *guest* system. This is because there are a fixed number of Regions in the *guest* address space. For example, if Regions are specified to be of size 4 Kilobytes, then there are $2^{20} = 1048576$ possible Region start addresses. Since the number of possible Region start addresses is much smaller than the number of possible Block or Trace start addresses (which may be several billion), translated regions can be stored in a flat table.

Although region based translation is not as popular as trace or basic block based translation (mainly due to the additional complexity of implementation), it is becoming more popular as compilation infrastructures such as LLVM are increasingly able to do the ‘heavy lifting’ of JIT compilation. Notable region based simulators include Arcsim (discussed in Section 3.2, which performs control flow analysis and optimisation, and Simit-ARM [83], which attempts to translate full guest pages at a time.

Arcsim is a full system simulation environment for the ARC architecture. Arcsim supports both user-mode and full system simulation, and can perform cycle accurate simulations for several ARC implementations (including branch predictors and complex cache hierarchies). High simulation speed is achieved using a region based DBT and aggressive LLVM based optimisations and JIT compilation. Simulation occurs in two phases: first, an interpreter executes instructions and forms a control flow graph for the binary code under simulation. After a specified interval (usually several thousand basic blocks), the control flow graph is analysed and ‘hot’ regions (regions containing code which has been executed a large number of times) are dispatched for translation to native (*host*) code. Translation occurs in separate threads to execution, and multiple translations can be in progress simultaneously. Once the translation of a region is complete, that translation will be used the next time simulated control flow returns to that region. Arcsim’s cycle accurate simulation implementation is discussed below.

Simit-ARM [83] is a full system simulator for the ARM instruction set. Simit-ARM supports region based DBT, by translating instructions to C code and compiling using GCC. Only basic profiling is performed, in order to determine which pages of code are ‘hot’ and should be compiled. Simit-ARM also supports parallel JIT compilation, either via threads or via network sockets.

2.7 Full System Simulation

While user mode simulators are capable of dealing with many simulation tasks, they are not capable of hosting a full operating system, nor simulating systems which do not include an operating system. In these cases, full system simulators must be used instead. These simulators include virtual memory models, exception and interrupt models, external devices, etc. These additional requirements typically mean that full system simulators are much more complex than user mode simulators, and they typically have a considerable performance penalty due to the increased complexity of memory accesses.

There have been many schemes developed to accelerate these memory translations. For example, EMBRA, presented by Witchell [109] keeps a 'relocation array' which acts as a cache of translated page addresses. Each entry in this array contains the physical address of the mapped virtual page, as well as any page protection bits. Lookups in this array are then inlined using EMBRA's DBT system.

QEMU also supports full system simulation. The approach taken in QEMU is to keep a small cache of translations, similar to EMBRA. QEMU extends this by indexing DBT translations by physical address. This means that these translations no longer need to be flushed when MMU mappings change (which is extremely frequently in a full-system environment, where a *guest* operating system is frequently context switching).

Koju [55] also seeks to improve translated code performance in a full system simulation context, by optimising indirect branches. Indirect branch targets are not known at DBT compile time, and so address translations for these branches must be performed as part of virtual memory emulation. Koju develops an intra-page offset calculation optimisation, but does not offer an efficient solution for virtual page address translation and relies on standard approaches.

Arcsim, described above, also supports full system simulation. In a paper by Topham et al. [104], a variant on the caching approach taken elsewhere is presented. Here, multiple caches are kept each for reads, writes, and instruction fetches. Each cache is a direct mapped software cache, indexed by low-order target virtual address bits. While improving on approaches taken elsewhere, this technique does not significantly differ from standard translation caching.

Hardware-accelerated approaches have also been examined by Argollo et al. [3]. Here, a proprietary AMD extension, SimNow, is used to accelerate virtual memory simulation. However, this technique is only applied to same-ISA simulation, and has not been applied to cross-architectural simulation. Hardware support for memory translations has also been developed by Transmeta [1], by using a single TLB containing both *host* and *guest* entries. This avoids costly TLB flushes when context switching between executing the *host* and *guest* processes. Transmeta have also described a speculative address translation system, where *host* memory segmentation features are used to speculate on the *host* virtual address corresponding to the *guest* virtual address in question [13].

Such hardware support provides very efficient memory translations, but might not be suitable if the *host* memory page size is larger than the *guest* page size. In this situation, the *host* machine cannot provide native memory protection which matches that of the *guest*, and so a software or combined hardware/ software solution must be used instead. One such solution is presented in [24], where translated code speculates on a particular *guest* page size for each memory access. If the speculation is frequently incorrect, then the *guest* code is retranslated.

Much research has focused on efficient implementation of dynamic binary optimisation (DBO) and instrumentation (DBI). DBO and DBI present many similar problems to simulation, but typically require that the *guest* and *host* have the same architecture. Examples include DynamoRio [20] and Pin [64]. However, these systems operate on individual user applications within a single address space, and so do not face the address translation challenges of full-system simulation. Other cross-architectural DBT systems, e.g. [73], are often limited to application-level simulation, but cannot host an OS due to missing full-system simulation support.

PinOs [22], which is built on the Xen [10] virtualisation platform, uses Intel's Vt technology [51] to perform full system instrumentation. Another prototype cross-architectural virtualisation platform, MagiXen has been presented [25]. This system is a virtual machine monitor with an integrated binary translator, which is capable of hosting an IA-32 virtual machine on an Itanium platform. While performance is good for numerical benchmarks, memory intensive workloads perform poorly. Similar work has been done by Baraz [9].

The efficient handling of dynamically generated or self-modifying *guest* code is also an important factor in simulation performance. A technique is presented in [7] for extending the *host* MMU with a 'T' bit which tracks *host* memory pages containing translated *guest* instructions. Writes to *host* memory pages with this T bit set cause an exception. The *host* pages are further subdivided so that modifying data (rather than instructions) on such a page does not trigger the translation to be discarded.

Other problems in full system simulation include efficient interrupt and exception handling. A checkpoint-and-rollback technique is presented in [59], in order to allow the reordering of *guest* operations during translation, permitting a much more aggressive optimisation strategy. If an exception or interrupt is generated during a section of re-ordered code, the section is re-executed using the original instruction ordering.

A different technique for efficiently handling interrupts and exceptions, presented in [85]. Here, each *guest* register is mapped to two *host* registers or memory locations. During execution, the mapping is alternated between the two *host* registers, providing an efficient checkpoint-and-restore system, should an exception take place during the block.

2.8 Performance/Power Modelling

While issues of performance or power modelling are not addressed in this thesis, it is important to note that many Instruction Set Simulation platforms support these in some form. Performance modelling (sometimes known as 'Cycle Accurate Simulation' or 'Cycle Approximate Simulation') seeks to predict how many cycles (and thus how much 'wall-clock' time) a system will consume in order to perform a particular operation. Power modelling seeks to predict how much dynamic energy may be consumed. These technologies typically require highly detailed pipeline, interconnect and memory models in order to obtain accurate results, and evaluating these models typically dominates simulation execution time.

Some work has been done in improving both the evaluation costs and accuracies of performance and power models. Techniques for simulation vary significantly, as simulators for different configurations benefit from different optimisations. For example, Arcsim [18] focuses on high-speed simulation of an in-order core with a relatively simple memory system by compiling pipeline updates into DBT-generated code. An extension [103] presents techniques for fast and accurate simulation of cache-incoherent multi-core systems, by only invoking the interconnect model when required by non-cached memory accesses.

Other simulators such as Gem5 [15] and MARSS [79] aim for highly accurate simulation of complex microarchitectures. Gem5 seeks to provide a detailed structural simulation of the CPU microarchitecture, including complex memory systems and memory delay simulations. The Gem5 infrastructure is a combination of the M5 core simulator and Gems memory system simulator. Multiple architectures and microarchitectural models are supported, and many parts of a processor model can be automatically generated. However, this involves a macro/template based description of the model source code rather than a direct description of the simulated system (as in ‘true’ ADLs). In addition, the default release of Gem5 supports only a limited selection of microarchitectural models with an interface defined between the architectural and microarchitectural components of the simulator, rather than specific and separate models for each simulated microarchitecture. For example, a simulation of an ARM system must be performed as a simulation of the ARM architecture, tied to a simulation of the included Alpha 21264 microarchitecture ‘tweaked’ to behave more similarly to the ARM system.

PtlSim [112] provides a highly configurable and detailed microarchitectural simulation of processors implementing the x86 ISA, including caches and SMT support. High accuracy is achieved by simulating the micro-op decoding process present in most modern x86 implementations, as well as the complex x86 memory management unit and page fault handling mechanisms. The Xen hypervisor platform is used to support full system and multicore simulation, and there is support for ‘fast-forwarding’ by switching between executing natively using Xen, or using the detailed model provided by PtlSim.

MARSS [79] is a hybrid approach, combining the DBT-based functionality of QEMU with the microarchitectural modelling of PtlSim. Support for newer architectural extensions (such as MMX) has also been introduced, as well as an improved execution model for complex instructions in order to improve accuracy. A simple memory system simulation has also been introduced, replacing the constant delays used in the original PtlSim.

McPat [61] provides timing, power, and area estimates and is based on the Cacti [74] cache modelling framework. Cactiaims to estimate cache power and area usage using a complex physical model of the electrical properties of cache structures and the wires used to connect them together and to other structures. McPat extends this model, adding support for core microarchitectural features and for processor power modes, interconnects, etc.

Strazdins et al. [101] model the UltraSPARC III CPU efficiently using a variety of time-saving techniques. In particular, a detailed microarchitectural model is not maintained, and instead only approximate models of important microarchitectural features are used. Although speedups are reported over highly detailed simulations, evaluations are only using small kernels rather than realistic workloads. Additionally, this simulation technique relies on knowing which architectural features are 'important', which would not necessarily be known when prototyping a new microarchitecture.

Schnarr and Larus [93] use memoisation techniques (also known as 'dynamic programming') to accelerate cycle accurate simulation of a MIPS R10000-like microarchitecture. Microarchitectural configurations are cached, alongside the actions taken to advance those configurations, meaning that when similar microarchitectural states are repeatedly encountered (such as when executing loops), the simulator can perform the cached actions rather than re-evaluating the microarchitectural model. Not all microarchitectural features are memoised. For example, caches are not memoised as these typically have very data-driven behaviour which is not effectively captured by the memoisation technique.

Arcsim performs cycle accurate simulation of a range of in-order single issue ARC microcontrollers and microprocessors. Simulation is highly accurate and includes the correct behaviour of pseudo-random cache replacement policies and branch prediction units. Very high accuracy is possible partly due to the simple nature of the system under simulation (usually single-core, in-order single issue processors), and high simulation speed is achieved using a 'functional-first' simulation scheme, where the functional behaviour of the system is simulated for each instruction, and the microarchitectural state is then updated to reflect the correct behaviour. This can cause some inaccuracies in rare circumstances (for example, instruction cache misses along speculated branch paths are not modelled). Arcsim is discussed more in Section 3.2.

Due to the high cost of evaluating high-accuracy architectural models at runtime, some work has been done on using machine learning and static analysis techniques to extract some timing information from application source or binary code statically. This does not allow the simulation of self modifying or dynamically generated code. However, an alternative approach is to build the model dynamically i.e. at runtime, so that when new code is generated or code is modified, a new or updated model can be built. Once the model has been built, it can be evaluated more cheaply than the full microarchitectural model. Some microarchitectural features which have complex behaviours such as caches and branch predictors may still be fully modelled in order to improve the accuracy of the simulation.

Schnerr et al. [94] first perform static timing analysis of each basic block of a target binary. Once a timing model is extracted, the original C source code of the executable to be simulated is translated into a SystemC TLM model, and annotated with the timing model. Structures such as caches and branch predictors are still evaluated dynamically, as well as any other data-dependent microarchitectural structures such as load/store stalls. Good simulation performance is obtained, although method is evaluated only on a few short-running benchmarks, making it difficult to assess the simulation accuracy obtained by this approach.

Ottlik et al. [76] perform offline static analysis in order to produce context based timing information. Possible control flow paths through the binary program are analysed in terms of 'execution contexts'. Use of execution contexts rather than explicit control flow paths greatly improves the efficiency of the model both in terms of required memory and lookup time. The paper reports good results for both simulation performance and accuracy, although complex microarchitectural features such as caches and branch predictors are not covered.

Powell et al. [81] use a continuous and online machine learning based approach, enabling accurate modelling of dynamically generated and self modifying code. Their approach is capable of adapting itself, depending on the confidence of each prediction, and predictions include effects caused by caches and branch prediction units. A high speedup over pure cycle-accurate simulation is reported with a relatively low degradation in accuracy.

2.9 Retargetability

As customisable architectures and instruction set extensions have become more prevalent, the extensibility and retargetability of Instruction Set Simulators has become more important. However, implementing these new features can be challenging, especially when high performance is desired.

Generally, using a retargetable simulation system involves creating a description of the desired architecture in a domain specific ADL (Architecture Description Language). This description is then passed through a tool to produce a module, which can then be 'plugged' into the simulation infrastructure. While many different systems exist for this, most with their own specific ADL, they can be broken down generally into two categories: high-level abstract descriptions, and low level structural descriptions.

2.9.1 High Level Descriptions

High level ADLs typically seek to describe the desired architecture at an abstract level. They typically deal directly with *guest* instructions, and often ignore much of the underlying microarchitecture. This style of ADL is particularly amenable to high level analysis and optimisation, and are typically capable of producing high speed simulators.

However, performance and power modelling simulators can not be easily generated from this type of description, as the description lacks microarchitectural information such as pipeline structure. A high level description could be coupled with a microarchitectural model in order to provide both high simulation throughput and performance modelling but this would require producing and synchronizing two different descriptions of the same system which presents maintenance and validation problems.

Due to its popularity, a large body of work regarding SystemC and Transaction Level Modelling (TLM) also exists [40, 77]. TLM involves building models which abstract away low level details of a system and instead focus on *transactions* within the simulated system. This approach avoids the overheads of having a detailed structural model of the entire simulated system, while still allowing the mixing of highly detailed component models with less detailed models which are faster to evaluate.

ArchC [90, 5] is a SystemC based instruction set simulation platform, essentially providing an interface for easily describing microprocessors at the instruction level, as well as a simulation infrastructure. ArchC can be used for functional simulation, or can have pipeline information included in order to enable cycle accurate simulation. However, the instruction and cycle accurate descriptions are typically very different, meaning that these two models must be kept in sync if both forms of simulation are desired for a particular architecture.

Blanqui et al. [16] present a method for generating an architectural model directly from the reference documentation provided by the ISA vendor. The PDF format documents are parsed and ISA syntax and semantic descriptions are extracted from the instruction encoding tables and implementation pseudocode contained within the document. However, reference document formatting is not consistent between different ISA vendors, meaning that different parsing and extraction strategies would be needed for each model.

Simit-ARM [83] is a simulator partially generated from a high level description. An abstract instruction decoding scheme is used to implement an efficient jump-table based instruction decoder, and snippets of C are attached to particular instruction fields (to allow for ARM's multiple immediate encoding formats and addressing modes), and to the leaf instructions. Although flexible, this mixes instruction syntax and semantics in a way which can make following the flow of execution of an instruction's semantic action quite difficult.

A 'Generic Instruction Model' is presented in [88]. A flexible method for describing the syntax and semantics of the instruction set is provided, as well as a detailed description of their method for automatically generating an instruction decoder. However, only a straightforward interpreter implementation is described.

SLED, a Specification Language for Encoding and Decoding, is presented in [84]. SLED is flexible enough to describe both fixed length RISC and variable length CISC instructions, and can be used to generate various binary utilities and code analysis tools. However, no method for describing instruction semantics is included in the language.

The SSL language [30] takes the opposite approach, allowing the description of instruction semantics but not bitwise instruction representations. SSL is also capable of describing complex 'higher order' instructions (such as delay-slot instructions from SPARC and repeat string instructions from x86).

The nML [38] machine description language uses a top-down approach. The description forms a grammar, for which each valid derivation represents one valid machine instruction. So, the structure of the grammar reflects the structure of the instruction set. As with SSL, complex and high level machine behaviours such as delay slots can be described fairly succinctly.

EXPRESSION [45] is another machine description language taking an abstract, high level view of the machine. EXPRESSION is designed to support generation of both retargetable simulators and compilers, while still supporting the simulation of microarchitectural details such as memory hierarchy and instruction pipelines. This system is designed to support efficient design space exploration, so it is easy to modify microarchitectural details such as the number of execution units. Although EXPRESSION supports the generation of compiler backends, this is only possible if the user provides a mapping back from compiler IR structures to architectural instructions.

Pydgin [63] uses a set of libraries embedded into RPython (also used by the PyPy Python implementation) to allow for the implementation of fast instruction set simulators using the Python programming language. Several specific optimisations are presented in order to improve the performance of the generated simulators.

2.9.2 Low Level Descriptions

This style of ADL describes systems at a very low level, typically involving direct descriptions of the hardware under simulation. This style of ADL is typically more flexible than a high-level ADL, as it is capable of describing systems at both high and low levels of abstraction depending on the detail contained within the provided description.

Low level descriptions are particularly amenable to providing simulators with accurate timing or power models. These simulators often operate on a cycle-by-cycle basis, where the simulation runs on each described architectural or microarchitectural component one cycle at a time. Simulation therefore involves, in each cycle, reading information from ‘input’ registers, processing it some way, and placing the processed data in ‘output’ registers. This closely matches the behaviour of the real system, although internal details are often abstracted away.

However, generating high performance simulators is much more difficult when using this style of ADL, as such descriptions lack many of the high level details which makes this possible in high level description languages. Less detailed descriptions can provide greater performance (for example, a description where the processor completes exactly one instruction per cycle), but the simulation rate is still held back by the cycle-by-cycle model, which is intrinsic to many low level simulation platforms.

While LISA [115, 80] still operates at the level of instructions, it provides a detailed method for describing the encoding, scheduling, and stage-by-stage pipeline behaviour of the system. This allows for highly detailed and performance modelling simulations. The descriptions are tied to a generic machine model which includes superscalar instruction scheduling logic, and which can be ‘tweaked’ to behave similarly to the modelled machine.

HARMLESS [53] is similar to LISA in that instructions are treated as an abstract concept. However, HARMLESS allows the model to separately describe the ISA syntax, the semantic behaviours, and the microarchitecture. For functional simulations, only the syntax and semantic descriptions are used and the microarchitectural description is ignored. This separation of descriptions also means that for certain architectures, a single architectural description can be compiled against multiple microarchitectural descriptions to produce performance modelling simulators for multiple processors without rewriting the common ISA components. For example, in the paper the PowerPC behavioural model is extended with two microarchitectural models, for the e200z1 and e200z6 embedded cores.

SystemC is also a popular platform for cycle accurate simulation [35, 21, 26]. SystemC is widely used in both structural and so-called TLM forms, depending on the desired accuracy of the simulation. The structural form of SystemC supports many features present in hardware description languages such as four-value logic.

2.9.3 Correctness

Once a processor model has been constructed, it must be tested in order to check the *correctness* objective. The scheme for checking the correctness of a model depends on the type and structure of the model under test: a low level structural model requires a very different testing approach than a high level abstract model. For low level and cycle accurate models it is also often desirable to test against real hardware. This can be used either to verify the correctness of the model, or if the model is taken to be a *golden reference*, to verify the correctness of the hardware.

The testing of an instruction set simulator might be done by treating it as a general piece of software and using generic software testing methodologies. Methods for test generation on restricted languages (e.g. [42]) might be applied to ADLs, or more general techniques might be used. These typically include so-called 'Concolic' testing, where parts of the program are executed symbolically and some 'concretely'. This mixed-mode execution allows us to formally reason about large parts of the program without requiring that the entire program be formalisable. This allows us to generate tests for each possible execution path through a program or subprogram. Such an approach is taken for general programs by Sen et al. [95] and Burnim et al.[23].

Ma et al. [66, 65] generate a simulator and tests from the x86 ISA manual, including tests for the complex addressing modes available in that architecture. These tests are then run against a 'hardware oracle' to confirm the behaviour of the simulator. However, the test generation is rather ad-hoc and requires a large number of test cases and different testing methodologies. They also examine the existence of ambiguity or inaccuracy in the ISA reference manuals, such as where particular behaviours are defined to be unpredictable but actually produce consistent results.

Randomized testing, known as 'fuzzing' is another approach common when verifying both processor hardware and processor simulators. Fully randomized testing is often difficult, as a truly randomly generated instruction may attempt to access an arbitrary memory location which may be mapped to a device, or simply be unmapped. The space of possible instruction encodings is also fairly large (2^{32} for a 32-bit instruction word) but the space of possible tests is even larger when input register and memory contexts are considered. Of course, many of these configurations are redundant, such as differences in memory space for a non-memory instruction, and differences in registers which are unused by the instruction under test. Martignoni et al. attempt to tackle this in [68]. They present both ‘naive’ and ‘optimised’ approaches to test generation in order to attempt to cover a large portion of the ‘interesting’ space of instructions without executing an excessive number of tests.

Another approach again is to show that the model under test is equivalent to another formal model, as in [71]. However, this requires that a second model be produced and maintained, and that both models are in a form suitable for this kind of formal analysis.

Alternatively, benchmark-driven testing seeks to use standard benchmark programs as comprehensive tests, reasoning that if a model is able to correctly execute a large and complex application, then it can be reasonably believed to be correct. Glamm et al. [41] build upon this by taking a hardware reference and comparing, after each instruction, the current state of both the simulated and reference register files. Although thorough, this approach is fairly slow, as communication between the reference hardware and the simulation host must be done via a debugging device.

The topic of testing hardware CPU implementations against an ISS reference has also received considerable research interest. In these cases, the microarchitectural state is also often compared, rather than just the architecturally visible effects. For example, Yang [111] tests a hardware implementation directly against an ‘oracle’ ISS. Analysis of test coverage for complex circuits is a related problem, e.g. Ho [47]. Generation of test cases for verification of hardware designs has also been examined by Mathaikutty [69] and Kodakara [54], where microarchitectural details of the system under test are compared against a low-level reference model. Test cases are generated by examining an architectural description and using a constraint satisfier. Although testing hardware is a related problem to testing simulators, techniques applied to hardware testing are often not applicable to software testing and so they will not be further examined.

2.10 Conclusion

This chapter has provided a basic description of Instruction Set Simulation, as well as some of the key technologies used to provide improved performance in Instruction Set Simulators such as Decode Caching and Static and Dynamic Binary Translation. Additionally, existing work relating to Instruction Set Simulation, and the Automatic Generation of Simulators, has been reviewed.

While a lot of work exists on generating simulators from high level descriptions, such simulators tend to suffer from poor performance. Separately, the performance of instruction set simulators has been studied extensively, with techniques ranging from novel DBT code generation techniques, to improved handling of control flow instructions. However, many of the techniques rely on hand tuning or specific architectural features, or have simply not been demonstrated in the context of a generated simulator.

This chapter has also discussed the testing and verification of simulators. Software testing is a huge and complex field, with many general testing methodologies, and techniques for generating tests for general software. However, these techniques may not be as effective or as efficient as a possible domain specific method which is designed to operate on simulators or on high level architectural descriptions. Additionally, many of the techniques developed for the verification of hardware are not applicable to high level models since these techniques tend to focus on cycle by cycle behaviour, or on hardware structures which do not exist in a high level software model.

Chapter 3

Infrastructure

3.1 Introduction

This chapter outlines the various software artefacts referred to by other chapters in this thesis. While the contributions presented in this thesis are of course the author's own work, building the infrastructure required to demonstrate and evaluate these contributions from the ground up would be an impossible task for a single individual. This chapter seeks to provide information on these artefacts in order to provide context for their use elsewhere in this thesis.

These are Arcsim, which is used as a base for the simulation framework presented in Chapter 4, LLVM, which is used as a JIT compiler backend, and CVC4, which is a constraint solver used for test generation in Chapter 5. The SPEC and EEMBC benchmark suites are also used, as they provide realistic computational workloads. QEMU is also used for comparison purposes, as it represents a hand-tuned, state of the art instruction set simulator.

How this chapter is structured

- We begin by outlining the major artefacts used to demonstrate the contributions presented in this thesis.
- We then briefly describe the benchmark suites used for the evaluation of these contributions

3.2 Arcsim

Arcsim is a high speed instruction set simulator developed at the University of Edinburgh. It was originally developed to act as a ‘golden-reference’ model for the EnCore microprocessor, also developed in Edinburgh. Arcsim makes use of cutting edge simulation technologies in order to provide high speed, true cycle-accurate simulations for the ARC architecture [102]. Arcsim makes use of LLVM to perform binary translation, which is briefly discussed in the section below (Section 3.3).

Arcsim integrates many novel techniques for high speed DBT [104, 18], high speed cycle accurate simulation [17, 81, 37], and high speed multi-core simulation [57]. It has also been used to evaluate interconnect designs for many-core systems-on-chips [103], as well as in assessing the feasibility of auto-parallelisation [32]. In this thesis Arcsim is extended to accept automatically generated simulator modules, in order to simulate the described architectures instead of the hard-coded ARC architecture.

Many of the techniques presented and discussed in this thesis have been applied to a heavily modified version of Arcsim, known as GenSim. Although GenSimis originally based on Arcsim, very little of the original code now remains. However, GenSim includes many of the techniques used by Arcsim to obtain high simulation performance, including Arcsim’s signature region-based, parallel, asynchronous DBT.

3.3 LLVM

LLVM [58] is a modern, object-orientated compiler framework designed to provide a high level, abstract representation of machine code, and to provide the means to analyse and optimise this code, and finally lower it to machine code. The core LLVM framework provides a means of constructing and analysing LLVM ‘bitcode’ (called LLVM IR), which consists of abstract Single-Static Assignment form (SSA-form) instructions representing a wide range of machine operations.

LLVM has seen widespread adoption both academically and commercially, and a large number of powerful optimisations have been developed, as well as back-ends for many different architectures. It has been used to develop a number of static analysis tools such as AddressSanitizer [97] and MemorySanitizer [100], as well as the Clang C/C++ compiler. LLVM also provides components optimised for JIT compilation, making it especially useful in the context of DBT.

3.4 CVC4

CVC4 [11] (Cooperating Validity Checker, version 4) is a SMT (Satisfiability Modulo Theories) solver, developed as a joint project between New York University and the University of Iowa. In the context of this thesis, CVC4 is used as a constraint solver in Chapter 5, in order to generate tests which exercise specific control flow paths. As this thesis is not specifically related to the large and complex areas of constraint satisfaction and theorem proving, this thesis treats CVC4 essentially as a ‘black box’.

3.5 QEMU

QEMU is a high speed instruction set simulator supporting a wide variety of architectures and platforms, originally developed by Fabrice Bellard [14]. QEMU is capable of simulating in both user mode and full system modes, and can simulate x86, MIPS, ARM, Power, and many more architectures and platforms. A high speed DBT code generator, TCG, is used. Each guest architecture is implemented primarily as a CPU data structure and TCG frontend which must be hand-written. As well as its cross-architecture DBT mode, QEMU also supports accelerated x86-on-x86 using hardware virtualisation extensions.

QEMU has been used throughout the literature as a base for more complex simulation setups, including cycle-accurate and cycle-approximate simulation [76, 108, 79, 26], parallel multicore simulation [31], software testing [12], and many other areas. As QEMU is primarily a performance-focused application (i.e., for performance reasons it is not designed to directly support any manner of microarchitectural simulation, instruction observability, automatic retargetability etc.) it is used in this thesis as a performance comparison.

3.6 Evaluation

Evaluating the performance characteristics of instruction set simulation technologies typically involves executing a known, deterministic workload on both a baseline simulator, and on the simulator incorporating the novel technology. Rather than using workloads specifically designed to test the simulation technologies directly, standard

Vendor & Model	DELL™POWEREDGE™ R610
Processor Type	2× Intel©Xeon™ X5660
Number of cores	2×6
Hyperthreading	Disabled
Clock Frequency	2.8 GHz
L1-Cache	$2 \times 6\times$ 32K Instruction/Data
L2-Cache	$2 \times 6\times$ 256K
L3-Cache	2× 12 MB
Memory	36 GB
Operating System	Linux version 2.6.32 (x86-64)

Table 3.1: DBT Host Configuration.

benchmark suites are used, as these are more representative of the kinds of workloads simulated in practice.

3.6.1 Equipment

In all cases, experiments have been carried out on the machine described in Table 3.1. This is a fairly powerful workstation-class computer with many CPU cores and plenty of memory. The large number of CPU cores is particularly important in our case since the ArcSim simulator (upon which most of the work presented in this thesis is based) makes use of an asynchronous, parallel DBT system which scales well across multicore systems [18].

3.6.2 Methodology

This thesis presents several techniques for improving the performance of instruction set simulators. In this case, a reduction in the total run time of a benchmark or other application is considered to be an improvement in performance. Results will typically be given as speedup against some baseline, where speedup is considered to be:

$$Speedup = \frac{Runtime_{old}}{Runtime_{new}}$$

Speedup will typically be presented for each benchmark in a suite, as well as a "Total" value. This total speedup is computed as:

$$Total\ Speedup = \frac{\sum Runtime_{old}}{\sum Runtime_{new}}$$

Such aggregate measures should typically be taken with a grain of salt, since they inevitably weight some benchmarks more heavily than others. For this reason, individual results are always provided alongside aggregate results in this thesis. Direct measurements of performance, such as run time and MIPS (Millions of Instructions per Second) are not given, since these are only relevant to a particular simulator implementation running on a particular *host* machine.

3.6.3 Artefacts

In this thesis the integer benchmarks of the SPEC CPU2006 suite are used. These represent large, complex workloads typical of high-end embedded and ‘desktop’ applications (such as compilers, compression algorithms, video decoding, etc.), and the EEMBC benchmark suite, which contains a wide range of small benchmark kernels typical of more deeply embedded systems, such as automotive applications, audio and text processing, etc.

3.6.3.1 SPEC

SPEC (The Standard Performance Evaluation Corporation) is an organisation which produces and maintains a wide range of benchmark suites for a variety of computer platforms and tasks. SPEC provide benchmark suites for computer systems ranging from Java-based server applications to Virtualisation technologies. In this thesis, evaluations are primarily based on results from the SPEC CPU benchmark suite, which contains a range of benchmarks testing integer and floating point performance on a variety of realistic applications. This thesis also omits the floating point portion of the benchmark suite. This is to remove the requirement to implement a bit-accurate floating point model in any architecture descriptions used for evaluations in this thesis, since this would represent an extremely large time investment. The benchmarks could be run in a so-called soft-float configuration (i.e., with the floating point instructions replaced by sequences of integer instructions), however this would only produce additional coverage of the integer instructions, which are already well exercised by the integer benchmarks.

These benchmarks, outlined in Table 3.2 are considered representative of a large array of computation workloads, and are commonly used when evaluating simulation technologies and improvements in computer architecture and microarchitecture.

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

Table 3.2: A summary of the SPEC CPU2006 Integer benchmark suite. All of the SPEC benchmarks are based on slightly modified ‘real world’ applications.

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

Table 3.3: A summary of the categories of benchmarks included in the EEMBC v1.1 Embedded benchmark suite. Most of the benchmarks are small kernels executed in a benchmark harness.

3.6.3.2 EEMBC

EEMBC (The Embedded Microprocessor Benchmark Consortium) [34, 60] develop and maintain an industry standard benchmark suite for embedded systems. In contrast with the SPEC benchmark suite discussed above, EEMBC benchmarks tend to be small, single-kernel benchmarks capable of operating on a ‘bare-metal’ system. As there are over 30 individual benchmarks, they are not listed individually here. However, they fall into the categories outlined in Table 3.3.

3.6.3.3 The Linux Kernel

The Linux kernel is one of the most well-known and influential open source projects, and certainly the most popular open-source operating system kernel. The first version of the kernel was released by Linus Torvalds in 1991, and it has grown to be the operating system of choice on servers, forms the basis for many desktop operating systems (known as ‘Linux Distributions’, and including Ubuntu, Fedora, and ArchLinux), and is also extremely popular in the mobile and embedded world.

For full-system simulation, the simulated (*guest*) operating system is ArchLinux for ARM, with benchmark binaries compiled for Linux. When performing full system simulation, all operating system operations and system calls are fully simulated. Experiments performed with user-mode simulation also use Linux system calls, but these system calls are performed via an emulation layer on the *host* machine.

Chapter 4

Efficient Simulator Generation

4.1 Introduction

As modern computer architectures become increasingly complex, simulating these architectures becomes increasingly difficult. At the same time, constant advances in simulation technologies intended to keep pace with the desire for increasing simulation speeds makes it ever more difficult to implement a fast, modern simulator. Automatically generating a simulator module, rather than hand-writing a simulator each time a new architecture or simulation technology is developed, seems a reasonable way of tackling this problem: new architecture descriptions can be written in order to cover new architectures, and new simulation technologies can be implemented on the simulator generation platform, allowing all generated simulators to use these improvements.

This chapter presents a new ADL (Architecture Description Language), GenC, and describes how the language is used to describe architectures and generate high speed Instruction Set Simulators. In particular, this chapter presents a novel *Partial Evaluation* technique for generating high speed DBT modules, which provide a significant speedup over the state of the art despite being generated from a high level description.

How this chapter is structured

- First, GenC, an ArchC based architecture description language is discussed.
- A novel *partial evaluation* technique for generating fast simulators is presented.
- Finally, this technique is integrated into the ArcSim simulator, producing GenSim, and this is compared against the state of the art.

4.2 The GenC Architecture Description Language

In this section, the design and implementation of the GenC Architecture Description Language will be presented and discussed. This language was developed in order to investigate the potential for the generation of high speed simulators, but still remains flexible enough to form the base for other analyses (in particular, the test generation performed in Chapter 5) and to produce other tools.

4.2.1 Existing Architecture Description Languages

There are several existing architecture description languages. These are typically divided into three types, each with different objectives and focused on a different part of system design. First, there are the *structural* languages. These are the languages which provide a low-level, detailed description of the system, typically described in terms of hardware structures. These might be used for verification and hardware generation. These include true hardware description languages such as Verilog and VHDL. There also exist *behavioural* descriptions (such as ArchC), which seek to describe the system in an abstract sense. Although these might be capable of modelling performance characteristics of the described system, they typically abstract away the implementation details of the system. So, for example, the processor is described in terms of instructions, and their pipeline behaviour, rather than describing the pipeline directly. These *behavioural* languages are typically used for high speed simulation and the generation of tools such as compilers, linkers, assemblers, etc. Lastly, *mixed-mode* languages incorporate features of both behavioural and structural descriptions. A *mixed-mode* description might be used for various purposes, but are typically too high level for hardware generation and too detailed for efficient simulation. Typically a single description can be used to generate either a fast, or a performance-modelling, instruction set simulator. SystemC is an example of such as language as it can be used to produce both abstract and cycle-accurate models.

Another popular option for simulation development is to hand-write portions of the simulator (such as logic for specialised instructions, external devices, or system events such as exceptions), but generate the rest from a high level description. This is often the case in full-system behavioural simulators since it is often more efficient to develop system components in a general purpose high level language such as C or C++, rather

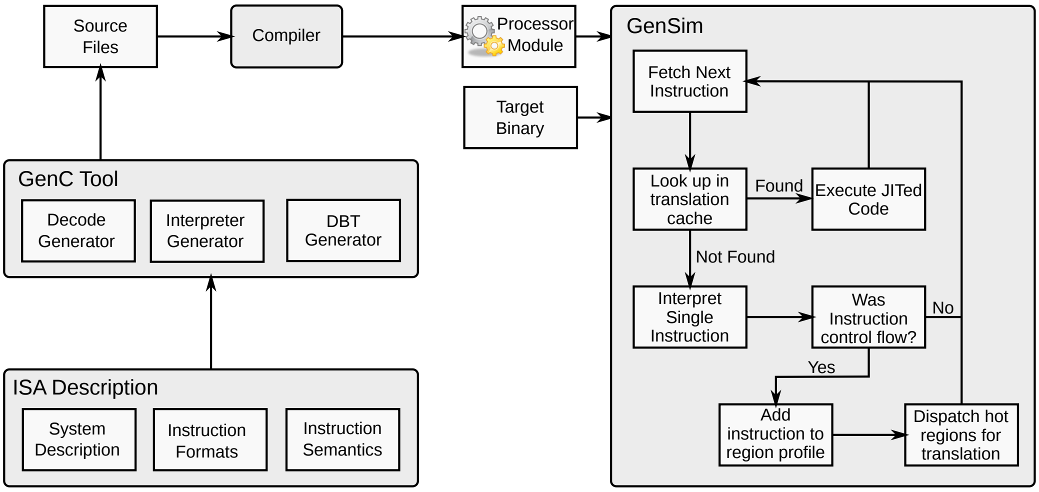


Figure 4.1: Diagram showing the process for simulating a binary application using our ADL based simulation framework. Details on the region profiling and translation system can be found in [18].

than in an ADL. Other times, the ADL may have been designed with a particular type of system in mind and it may be difficult to efficiently describe unusual operations or instructions. Examples of both of these situations can be found in the Simit-ARM simulator [83], which has a description and auto-generation system for the architectural portion of the simulator, (i.e., the syntax and semantics of instructions) but implements certain instructions, and the system model, directly in C++.

4.2.2 Our ADL

Our ADL, GenC, is based on a modified ArchC [90, 5]. While ArchC is implemented as a set of classes and templates on top of SystemC, GenC produces modules for a standalone simulation platform, GenSim. GenC models are first processed using a simulator generation tool (also called GenC), which outputs several C++ source files. This files can then be compiled to produce a module, which is then dynamically linked into our simulator framework, GenSim. The overall flow of this process can be seen in Figure 4.1. GenC can be considered to be a *behavioural* ADL, as it deliberately lacks many of the features required to accurately describe microarchitectural details (although some work has been done to extend the language in this direction).

The general philosophy of the design and implementation of the language is that it should be *intuitive*, so that the user is able to write the model in a way in which they feel

```
AC_ARCH(armv5e)
{
    // General Purpose Registers
    ac_regbank<uint32> RB:16;          ①

    // General Flags
    ac_reg<uint8> C;                   ②
    ac_reg<uint8> V;
    ac_reg<uint8> Z;
    ac_reg<uint8> N;

    ac_wordsize 32;                    ③

    ARCH_CTOR(armv5e)
    {
        ac_isa("armv5e_isa.ac");       ④
        ac_isa("armv5e_thumb_isa.ac");
    };
};
```

Figure 4.2: Example of an ARMv5 system description for user-mode simulation. The general purpose register bank ① and each status flag ② are individually described, as well as the system word size ③. Finally, the two instruction sets supported (ARM and Thumb) are included in the model by referring to the files in which they are described ④.

comfortable, *compact*, so each object in a description has only a single definition, aiding maintainability, and *efficient*, both in terms of the tools which process descriptions, and the artefacts which they generate.

Architecture descriptions in our ADL have three main components:

- A system description, outlining the basic architectural features such as the register file.
- An ISA Syntax description, describing how instructions are encoded.
- An ISA Semantic description, describing how instructions are executed.

GenC also supports architectures which have multiple ISA Modes, for example ARM/Thumb and MIPS/MIPS16e. In these cases, there is one ISA Syntax and Semantic description per ISA mode.

4.2.2.1 System Description

The system description portion of the architecture description contains basic information about the architecture such as the native word length, the register file and status flags, and the endianness. This part of the description also links to each of the required ISA Syntax descriptions. An example system description for a simple ARM user mode simulation model can be seen in Figure 4.2.

```
AC_ISA(arm)
{
    ac_format Type_DPI1    = "%cond:4 %op!:3 %func1!:4 %s:1 %rn:4 %rd:4 %shift_amt:5 \
                             %shift_type:2 %subop1!:1 %rm:4";
    ac_format Type_MBXBLX = "%cond:4 %op!:3 %func1!:4 %s:1 0xfff:12 %subop2!:1      \
                              %func2!:2 %subop1!:1 %rm:4";
    ...
    ac_instr<Type_DPI1> and1, eor1, sub1 ... ;
    ac_instr<Type_MBXBLX> bx, blx2;
    ...

    ISA_CTOR(armv5e) {

    and1.set_decoder(op=0, subop1=0, func1=0);                               ①
    and1.set_behaviour(and1);
    and1.set_asm("and%[cond]%sf %reg, %reg, %reg", cond, s, rd, rn, rm, ...);  ②


    bx.set_decoder(op=0,subop1=1,subop2=0,func1=0x9,s=0,func2=0);
    bx.set_behaviour(bx);
    bx.set_asm("bx%[cond] %reg", cond, rm);
    bx.set_end_of_block();                                                    ③
    bx.set_variable_jump();

    ...
    }
}
```

Figure 4.3: Example snippets of ISA syntax description using GenC, our ArchC-based ADL. Here two instruction formats (Type_DPI1 and Type_MBXBLX) are described. Several instructions associated with these formats, and the encoding and assembly of each instruction, are also described. Each instruction is also associated with a behaviour, and any branching behaviour is specified.

MBXBLX template

cond	op	func1	s	1 1 1 1 1 1 1 1 1 1 1 1	subop1	func2	subop1	rm

BX constraints

- - - -	0 0 0	1 0 0 1	0	- - - - - - - - - - - -	0	0 0	1	- - - -

Final Bitstring

X X X X	0 0 0	1 0 0 1	0	1 1 1 1 1 1 1 1 1 1 1 1	0	0 0	1	X X X X

Figure 4.4: Diagram showing how an instruction format, combined with decode constraints for multiple instructions, gives the final bit string used to perform instruction decoding. ‘X’s in the final bitstring represent unconstrained bits, meaning that the cond and rm fields can carry arbitrary values in a BX instruction.

4.2.2.2 ISA Syntax

GenC ISA Syntax descriptions are very similar to the original ArchC descriptions. In fact, an unmodified ArchC description can be used provided that only the interpretive execution mode of our simulator is used (i.e., no DBT is performed).

Figure 4.3 shows a stripped-down example of an ISA description. Here, two instruction formats are described, one for data processing instructions (DPI), and the other for branching instructions (MBXBLX). Several instructions associated with each of these formats are also described. Then, in the ISA_CTOR portion of the description, the properties of each instruction are given. For example, ① lists 'decode constraints', which, when combined with the instruction format string, provide the instruction template bit string inserted into the decode tree. Figure 4.4 shows how instruction templates and constraints are combined in to decode bitstrings to be inserted into the decode tree. ② describes how to disassemble the instruction for debugging purposes, using a printf-like format string. Mappings from numerical values to strings, described elsewhere in the syntax description, provide a flexible way of describing these. For example, the `cond`, `sf`, and `reg` formats used within the assembly format string are all defined elsewhere in the syntax description. Finally, ③ provides additional information on the nature of branching instructions, used by the DBT system to generate faster code.

ISA Syntax descriptions provide a clear separation between the available instruction formats, the set of actual instructions available, and the mapping between instructions and formats. An instruction format describes the set of bitfields contained within an instruction encoding. Instruction formats are not directly used when decoding instructions and may overlap with each other partially or fully.

Once instruction formats are described, the set of actual instructions is provided. Each instruction is assigned to a format, so there is a many to one relationship between instructions and formats. The remainder of the description contains information on how each instruction is actually encoded, how it should be assembled/disassembled (the original ArchC implementation supports the generation of binary utilities including assemblers, GenC uses this information to disassemble instructions when producing debugging output from our simulator), and some information used by our DBT system which is not in the original ArchC and which will be covered in Section 4.4.

```
uint32 pc_check(uint8 reg_index) ...
uint32 decode_imm(uint8 type, uint8 shft, uint32 val, uint8 c_i, uint8 &c_o) ...    ①
void update_ZN_flags(uint32 value) ...

execute(and1)
{
    uint32 val;                                                                      ②
    uint32 imm32;
    uint32 decode_input = read_register_bank(RB, inst.rm) + pc_check(inst.rm);       ③
    uint8 carry_in = read_register(C);
    uint8 c;
    imm32 = decode_imm(inst.shift_type, inst.shift_amt, decode_input, carry_in, c);④
    uint32 src1m = read_register_bank(RB, inst.rn) + pc_check(inst.rn);
    val = src1m & imm32;
    if(inst.s)                                                                       ⑤
    {
            update_ZN_flags(val);
                write_register(C, c);
    }
    write_register_bank(RB, inst.rd, val);                                           ⑥
}
```

Figure 4.5: Example snippet of ISA semantic description. ISA semantic actions are implemented using a high-level but restricted C-like language.

4.2.2.3 ISA Semantics

The original ArchC implementation has instruction semantics described using SystemC functions. These functions are then called, in order to execute the simulated instructions. This is essentially an interpretive model of execution. SystemC is a complex language, and the fact that arbitrary language constructs could be used inside instruction semantics makes analysing these descriptions difficult. So, GenC removes the reliance on SystemC and instead implements a simpler language for describing instruction semantics.

This instruction semantic description language is C-like, although there are some significant differences which reduce the power of the language (in order to aid analysis). These limitations do not adversely affect the description of instruction semantics, since individual instructions tend to have simple behaviours which do not require the full power of a language such as C++ to describe. For example, the instruction semantic description language does not allow the use of arbitrary pointers or pointer types. *Host* memory cannot be directly accessed, and *Guest* memory can only be accessed via special memory access intrinsic functions. Most operations on the guest machine are performed via calls to intrinsic functions.

Figure 4.5 shows an example snippet of our ARMv5 ISA description. Here the and1 instruction (for which the syntax was defined in Figure 4.3) is described. First ①, some helper functions are defined. The decode_imm function implements ARM's sophisticated shifter unit. Note that the `c_o` argument is defined with an ampersand, meaning that it will be passed by reference.

The instruction behaviour begins by defining some variables ②. The language supports a full set of signed and unsigned integer data types. At ③, the `read_register_bank` intrinsic is used, in order to obtain a value from the register file. A helper function is also used. ④ shows a call to the shifter function. Note that the variable `c` is passed by reference as `c_o`. At ⑤, the instruction is inspected to determine whether status flags should be updated. Inlining this as control flow rather than as a separate instruction type (as is done in several other simulators) means that the ISA description is much more concise. Finally, at ⑥, the output is written to the destination register.

4.2.3 Implementation of the GenC Tool

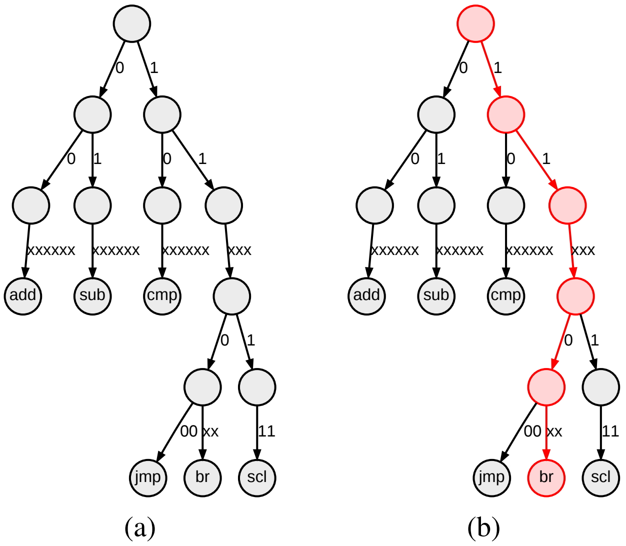
The GenC tool is a modular C++ application used to analyse GenC descriptions and produce useful artefacts such as simulation modules (described in this chapter) and test suites (described in Chapter 5. A set of ANTLR [78] generated parsers are used to process the descriptions into an ANTLR AST, which is then transformed into a set of GenC-specific data structures. These data structures contain information such as the structure of the register file and information on instruction formats.

Instruction semantic descriptions are further processed into an SSA form, where simulation-specific actions (such as accessing *guest* registers and memory) have dedicated SSA node types. This SSA representation can then be walked in order to generate interpreters, DBT modules, and to perform the test generation described in Chapter 5.

4.3 Generating a Simulator Module

High level functional simulators are typically constructed from a number of individual components. While some simulators may merge or mix these components (e.g., QEMU mixes instruction decoding and implementation), these components are kept separate in GenC in order to allow for reuse of components in other contexts (such as the same instruction decoding structure being used in both the simulator, and in a binary disassembler), and to allow for easy swapping of implementations to test new ideas and implementations.

GenC typically generates four types of component for simulation:



(a) (b)

Figure 4.6: A decode tree for a simple 8-bit instruction set. Figure (b) shows the path taken through the tree in order to decode a br instruction such as `0b11101011`. xs represent 'don't-care' bits. These may carry important information about the instruction (e.g. source or destination registers) but do not contribute to decoding the instruction type.

- An instruction decoder,
- A disassembler (for debugging/tracing purposes),
- An interpreter,
- A DBT frontend.

Of these, the decoder, interpreter and DBT frontend are the most interesting, and there are discussed in the section below.

4.3.1 Instruction Decoding

Instruction Decoding is one of the first steps in executing any instruction, whether in hardware or simulation, and whether interpreted or via a DBT system. There are multiple possible approaches to decoding instructions in simulation, and the suitability of each of these depends on the ISA under simulation. For example, when decoding x86 instructions, a finite state machine is necessary due to the possibility of instruction prefixes and the complex encodings of register and memory accesses. VLIW instructions also require a different strategy, due to 'bundling'. Our ADL primarily targets RISC instruction sets which do not require these complex features, but they must be kept in mind to ensure that compatibility could be added in future if desired.

The approach taken by GenC when generating instruction decoders is to build and optimise a decision tree for the ISA. At each node of the tree, one or more bits of the input instruction encoding are examined and compared against several possibilities. If no available possibility matches the values of the bits, then the tree backtracks and moves along the last encountered ‘don’t care’ edge.

Special care must be taken when decoding variable length instructions sets such as Arcompact and Thumb-2. Instruction formats in these architectures can contain fields which cross the boundaries between each fetched unit of the instruction which means that the complete instruction word must be rearranged relative to how it is stored in memory in order to decode the instruction correctly. For this reason, GenC handles such fields by splitting the field for the purposes of constructing the decision tree.

4.3.1.1 Constructing the Tree

In order to construct the decode tree, GenC first starts with an empty tree which contains a single ‘invalid’ node. A ternary bit representation of each instruction is then generated, and these are processed in turn in order to generate the full tree. The ternary instruction representation contains ones, zeros and ‘don’t care’ bits which represent variable fields in the instruction encoding (such as register indices, immediate values, etc.). This bit representation is then split into ‘care’ and ‘don’t care’ sections, and these sections are added to the tree ‘left to right’ (i.e., MSB to LSB). An example tree can be seen in Figure 4.6.

Constructing the tree in this way means that each node has at most one ‘don’t care’ edge, which is important as it eliminates ambiguity in the tree. This allows us to implement these fall throughs as ‘default’ cases in switch statements rather than requiring a more sophisticated backtracking system. Individual edges may also encode multiple bits, so the leaf nodes may not all have the same tree height. This is usually the case with ISAs with mixed instruction lengths such as Thumb-2 - the section of the decode tree for the 32-bit instructions will have a much greater height than that for the 16-bit instructions.

4.3.1.2 Optimising the Tree

Once the full tree is constructed, it is then optimised. Each node of the tree is considered in turn. If a node has edges with similar prefixes, these edges are merged and a new node

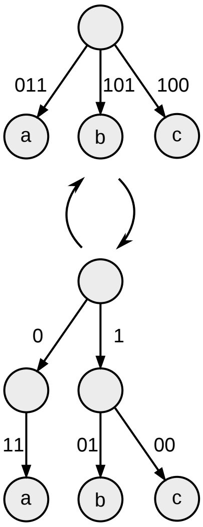


Figure 4.7: Optimising nodes in the decode tree. Nodes in the tree may be merged or split, based on some user-tuned heuristics.

is produced. On the other hand, if a node *A* has too few outgoing edges then these edges are moved to the parent node *B* and node *A* is removed from the tree. This is essentially the same operation happening forwards and in reverse (Figure 4.7). Currently, user-tuned parameters determine when the optimisation is applied, and in which direction, since the best set of values differs by *host* machine and *guest* architecture.

4.3.1.3 Generating the Decoder

Once a decode tree is constructed, the final step in generating a decoder is to produce the code actually implementing the decode logic. There are several possible methods for doing this, including as a switch statement tree (which is the implementation technique used by GenC) or as a set of nested jump tables (used by Simit-Arm [83]).

Although determining the instruction type is an important part of the decode process, the data fields, e.g. the source and destination registers, immediate values, etc., must also be extracted. This can be done eagerly (where the value is extracted from the instruction and stored in a data structure) or lazily (where the shifting and masking operations are done each time the value is accessed). GenC uses an eager decoder, in order to avoid the overhead of shifting and masking the fields each time they are accessed. This means that our data structure for storing a decoded instruction is somewhat larger (since it must include space for each instruction field). However, these structures can be efficiently cached [104], so this does not significantly impact simulation performance.

```
...
add r0, r1, #3
cmp r0, r4
beq 400
...
```

(a) A simple example fragment of ARM assembly

```
...
  call void @add(%struct.cpu* %0, [instruction fields]);
  call void @cmp(%struct.cpu* %0, [instruction fields]);
  %pred = call i1 @pred_eq(%struct.cpu* %0);
  br i1 %pred, label %beq_taken, label %beq_not_taken;

beq_taken:
  call void @b(%struct.cpu* %0, [instruction fields]);
  br label control_flow_handler

beq_not_taken:
  br label control_flow_handler
```

(b) The LLVM which might be emitted for that fragment

Figure 4.8: LLVM-based DBT using a function-call based instruction translation method. While this is straightforward to implement, it provides very poor runtime performance.

4.3.2 Interpretation

Generating an interpreter from a GenC model is simple since the ISA Semantic descriptions are a strict subset of C++. The instruction semantic functions can be inserted directly into a C++ source file, provided that the required intrinsic functions for accessing memory, registers etc. are available. For most RISC architectures, the flow of instruction execution - Fetch, Decode, Execute - is general enough that a generic interpreter structure can be provided which performs each of these functions in a loop, with the GenC model filling in the actual decode and instruction execution behaviour.

However, it is also possible to generate an interpreter from the parsed AST of the ISA semantic descriptions. This allows much greater flexibility in the implementation of the interpreter. For example, this feature is used in Chapter 5 in order to instrument the flow of execution through each instruction implementation and profile instruction block and path coverage. This could also be used to generate other special features in interpreters, or be used to generate optimised interpreters using e.g. instruction specialisation.

Although the *partial evaluation* techniques discussed later in this chapter could be applied to interpreters, this is not generally necessary as the code size and compile time reductions are not as important when compiling an interpreter (which is done in an Ahead-Of-Time context) when compared with DBT translations (which are performed in a Just-In-Time context).

4.3.3 A Naïve DBT

The simplest approach to generating a LLVM-based DBT system for an instruction set simulator is to compile each instruction implementation into a function, and then translate each instruction into a call to its associated function (Figure 4.8). LLVM can then be used to inline and optimise these functions, before finally generating native *host* machine code. Note that this is not particular to the context in which an instruction is translated (i.e., block-based, trace-based, region-based etc.), but instead is concerned with translating individual instructions.

So, GenC is able to generate a naïve LLVM based DBT by using Clang (a C/C++ compiler based on the LLVM framework, which can be configured to emit LLVM bitcode) to compile each ISA semantic instruction implementation into a separate LLVM bitcode function. The bitcode for these functions can then be packaged into the generated processor module, and at translation time, GenSim can generate calls to each of these functions.

Note that the reason this is considered to be a 'naïve' implementation is that it gives very poor performance. LLVM is quite poor at performing optimisations on functions containing a very large number of basic blocks, and since each GenC ISA semantic description can contain arbitrary control flow structures such as loops, switch statements and if-then statements, the compiled LLVM bitcode function for each instruction can consist of many basic blocks. The LLVM optimisation for inlining this large number of basic blocks (which consists of cloning the data structures for each block) is therefore expensive, in addition to the increased cost of performing further optimisations on these functions. In addition, many of these basic blocks are 'dead', that is to say they are never encountered at runtime. Although they may be removed by a dead code elimination optimisation pass, this means that time is spent constructing the data structures representing these blocks, analysing them to determine if they are dead, and then, assuming that they are dead, tearing them down again.

4.4 High Speed Dynamic Binary Translation

Although the naïve DBT system described above provides an improvement over simple interpretation, it has a major drawback: the reliance on high-power LLVM optimisations causes the translation speed to be extremely poor (see Section 4.5.1). This causes a large

```
%1 = sub i32 32, i32 %ror
%2 = shl i32 %imm, i32 %1
%3 = shr i32 %imm, i32 %ror
%4 = or  i32 %2, i32 %3
%5 = load i32* %r0_ptr
%6 = sub i32 %5, i32 %4
store i32* %r0_ptr, i32 %6
```

(a) LLVM bitcode emitted by a Naïve JIT. The LLVM bitcode must be optimised using expensive analysis and transformation passes, otherwise the rotated immediate is computed each time the instruction executes.

```
uint32_t imm_l_shift = 32 - rotate;
uint32_t imm_l = imm << imm_l_shift;
uint32_t imm_r = imm >> rotate;
uint32_t imm_val = imm_l | imm_r;
```

(b) A *partial evaluation* JIT knows to compute the value at JIT time...

```
%5 = load %r0_ptr
%6 = sub %5, 0xa000000a
store %r0_ptr, %6
```

(c) ... and then emit LLVM code using the computed value

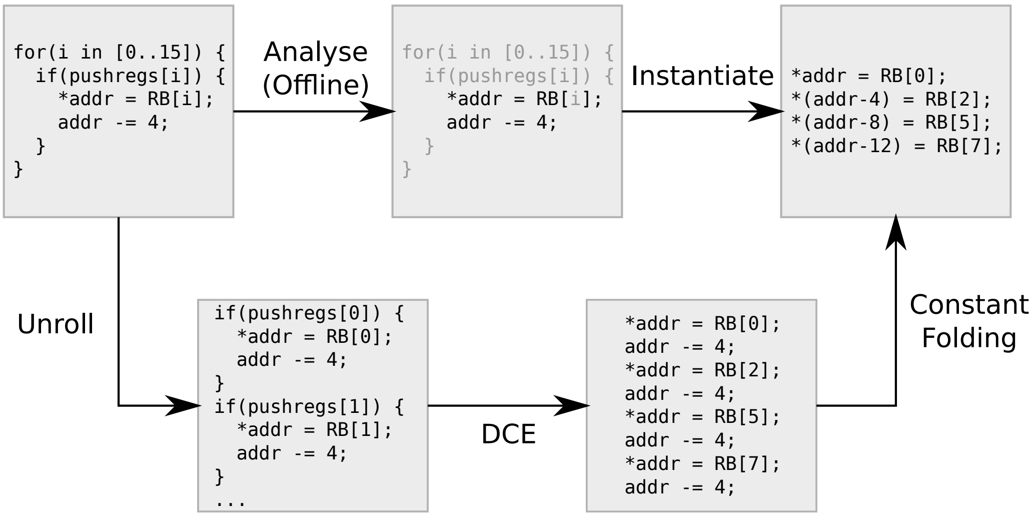
Figure 4.9: Consider the ARM instruction `sub r0, #0xa000000a`. The 32-bit immediate value is encoded using only 12 bits. While a naïve DBT might emit the LLVM instructions required to decode the value, and then (hopefully) optimise them away, a partial evaluation based approach calculates the value in advance, and emits only the final decoded constant value.

‘warm-up’ time in the simulator and means that simulation of short-running programs performs no-better than when using a simple interpretive simulator. Additionally, the complex and large code regions produced are not particularly suitable for analysis and so overall simulation speed also suffers.

In order to solve these problems, a partial-evaluation optimisation on the generated DBT (presented in [107]) is performed. Rather than simply passing our ISA semantic descriptions to Clang, to be compiled into LLVM instructions, GenC produces a LLVM bitcode generator directly. This should provide us with much better ‘warm-up’ time (only the required code is generated and optimised), as well as giving better overall simulator performance (since the generated LLVM bitcode is much more amenable to complex optimisation passes).

While a naïve DBT might generate IR, and then optimise it afterwards, the key idea presented here is to generate optimised IR in the first instance, by *identifying* opportunities for optimisations at generation time, and *exploiting* these opportunities at JIT time. For example, while the naïve DBT described above would generate LLVM bitcode to perform an immediate decoding calculation (which may or may not be optimised by LLVM), our partial-evaluation DBT would perform this immediate decoding calculation at JIT time, then use the result at runtime (Figure 4.9).

In order to generate this DBT, it must be determined what can be calculated at JIT time and what must be put off until runtime. If a computation can be completed at JIT time, it is referred to as being ‘fixed’. Analysing the ISA semantic description to



```
for(i in [0..15]) {
  if(pushregs[i]) {
    *addr = RB[i];
    addr -= 4;
  }
}
```

```
for(i in [0..15]) {
  if(pushregs[i]) {
    *addr = RB[i];
    addr -= 4;
  }
}
```

```
*addr = RB[0];
*(addr-4) = RB[2];
*(addr-8) = RB[5];
*(addr-12) = RB[7];
```

```
if(pushregs[0]) {
  *addr = RB[0];
  addr -= 4;
}
if(pushregs[1]) {
  *addr = RB[1];
  addr -= 4;
}
...
```

```
*addr = RB[0];
addr -= 4;
*addr = RB[2];
addr -= 4;
*addr = RB[5];
addr -= 4;
*addr = RB[7];
addr -= 4;
```

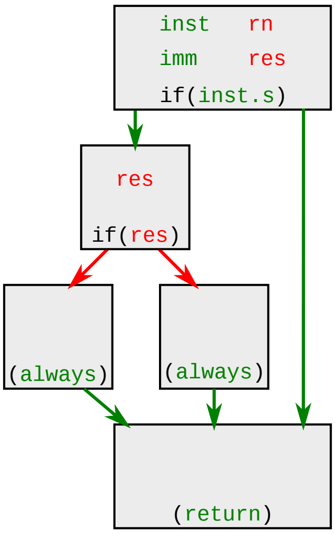
Figure 4.10: Comparison of Partial Evaluation (top) against traditional optimisation techniques (bottom), applied to a snippet of pseudocode. The Partial-Evaluation based system analyses the code at *generation* time, and at JIT time immediately produces optimised IR. In contrast, the traditional scheme must apply expensive loop unrolling, dead code elimination, and constant folding transformations, all at JIT time.

determine which computations are ‘fixed’ is referred to as ‘Fixedness Analysis’. An example of a ‘fixed’ computation would be the computation of an immediate value which depends only on constant numbers and instruction fields. An example of a non-fixed computation would be one which includes one or more values read from registers or from memory.

Control flow in the ISA semantic description is also determined to be fixed or non-fixed. An example of a fixed control flow statement would be an if statement which has as its condition a fixed computation. A basic block in our ISA semantic description which has only fixed incoming control flow edges is itself fixed. Importantly, a fixed basic block can contain non-fixed computations, and vice-versa.

Many of the transformations performed by the partial evaluation process could be accomplished using traditional constant propagation, dead code elimination, and loop unrolling techniques. However, traditional techniques require that *a*) both analysis and transformation phases of the optimisation are performed at JIT time, and *b*) the full, non-optimised IR is built up prior to analysis (which might itself be expensive). In contrast, use of partial evaluation techniques means that the analysis and much of the transformation can be performed at generation time, and optimised IR is produced in the

```
uint32 imm = ror(inst.imm, inst.rotate);
uint32 rn = read_register(inst.rn)
uint32 res = rn + imm;
write_register(inst.rd, res);
if(inst.s) {
    if(res) write_Z(0);
    else write_Z(1);
}
```



(a) A simple instruction semantic description for an add immediate instruction

(b) Control Flow Graph for this description

Figure 4.11: Figure (a) shows an example ISA semantic description. The GenC user writes a C-like description as normal. Figure (b) shows how GenC has analysed this description to identify fixed (green) and non-fixed (red) variables and control flow elements.

first instance, rather than constructing non-optimised IR and then optimising it. Figure 4.10 shows a high-level comparison of these two approaches.

4.4.1 Fixedness Analysis

In order to compute the fixedness of each statement in the SSA-form representation of the ISA semantic description, each of the SSA statements has a fixedness attached to it. For some types of statement, the fixedness can be easily evaluated. For example, constant values are always fixed, and values read from registers or memory are never fixed. Arithmetic statements are fixed if and only if all of their inputs are fixed. More complex analysis is required for determining the fixedness of statements which read from variables, as they are fixed if and only if all writes dominating the read statement are fixed. This process of computing whether these variable read statements are fixed is known as ‘fixedness analysis’.

Fixedness analysis can be considered to be similar to liveness analysis. Liveness analysis concerns itself with determining which values are ‘live’ throughout the control flow graph of a function. A value is ‘live’ at the point where it is assigned to a variable, and is ‘killed’ at the point where it is overwritten by a new value. This analysis can be

```
function InsnImplFixedness($action$)
    for all $b \leftarrow BB \in action$ do
        $b.dyn\_in \leftarrow []$
        $b.dyn\_out \leftarrow []$
        $b.ctrlflow \leftarrow invalid$
        $b.mark\_variable\_accesses\_as\_fixed()$
    $wl \leftarrow [action.entry\_block]$
    while $wl$ is not empty do
        $b \leftarrow wl.pop\_front()$
        $result \leftarrow$ BBFixedness(b)
        if $result = False$ then
            $wl.insert(b.successors)$
function BBFixedness($block$)
    for all $p \leftarrow BB \in block.predecessors$ do
        if $p.ctrlflow = dynamic \vee \neg p.final\_stmt.is\_fixed$ then
            $block.ctrlflow \leftarrow dynamic$
        $block.dyn\_in \leftarrow block.dyn\_in \cup p.dyn\_out$
    $dyn\_now \leftarrow block.dyn\_in$
    for all $s \leftarrow Statement \in block.statements$ do
        if $s$ writes a dynamic value to a variable $v$ then
            $dynamic\_now \leftarrow v$
        if $s$ reads a variable in $dyn\_now$ then
            mark $s$ as dynamic
    $block.dyn\_out \leftarrow dyn\_now$
    if $block.ctrlflow$ changed $||dyn\_now \neq dyn\_in$ then
        return $False$
    return $True$
```

Figure 4.12: Computing ‘fixedness’ of variable modifying statements in an instruction implementation.

done in a single pass for functions with simple control flow structures (i.e., no loops). However, when loops are involved a more complex approach must be taken to ensure that the liveness of each value is computed correctly.

Fixedness analysis differs in several important ways. Basic statements such as constant values and instruction fields are considered to be fixed, as well as expressions containing these values. A read from a variable is considered to be fixed if all dominating writes to that variable are fixed. If a non-fixed value is written into a variable, that variable is considered to be non-fixed until a fixed value overwrites it. A simple example can be seen in Figure 4.11.

The fixedness analysis algorithm can be seen in Figure 4.12, and operates on a SSA-form representation of a ISA semantic description. In order to maximise the possible savings in terms of bitcode generation and optimisation, all non-intrinsic functions are inlined at each call site in each ISA semantic description. Certain complex functions (such as those used to implement exception and interrupt handling logic) are marked as 'noinline' - in this case the return value of the function is always considered to be non-fixed.

The function INSNIMPLFIXEDNESS considers the semantic description for a single instruction. First (Line 2), data structures for each basic block are initialised. These include dyn_in, which is a set of variables which carry dynamic (non fixed) values at the start of the block, dyn_out, which is a set of dynamic variables at the end of the block, and ctrlflow, which states whether the block has fixed or dynamic incoming control flow. The mark_variable_accesses_as_fixed function marks each read and write to a variable as fixed.

Once initialisation is complete (Line 7), a work list wl is initialised to contain the entry block of the instruction semantic. Then, each entry in the work list is processed using the BBFIXEDNESS function, until the work list becomes empty. If the BBFIXEDNESS function returns false (Line 11), this indicates that some element (variable or control flow) has changed from fixed to dynamic, and so each successor of the block currently being processed should be reprocessed (since their incoming control flow or dyn_in may have changed).

The BBFIXEDNESS function begins (Line 13) by assessing the control flow of this block, and calculating the current dyn_in set. The control flow for a block is dynamic if any predecessor of that block has dynamic control flow, or if the terminating statement of that block is not fixed (Line 14). For example, a branch which has a condition relying only on fixed variables is considered to be fixed. A branch which has a condition whose calculation includes dynamic variables is itself dynamic. The current block's `dyn_in` set is computed to be the union of its predecessors' `dyn_out` sets (Line 17).

Each statement s in the block is then considered in turn (Line 19). If s writes a dynamic value to a variable v (e.g. a read from a register or from memory) then v is considered to be dynamic and is added to the `dyn_now` set. If s reads a dynamic variable, then any values produced by s are considered to be dynamic. Finally (Line 24), the block's `dyn_out` set is set to `dyn_now`. If new dynamic variables have been discovered, or if the control flow status of this block has changed, then BBFIXEDNESS returns false, indicating that the block's successors should be added to the work list. All blocks start with 'invalid' control flow, so all blocks will be processed at least once.

4.4.2 Generating LLVM Bitcode For An Instruction

Once the fixedness of each basic block and statement in the SSA-form representation of an instruction has been evaluated, a function to generate LLVM bitcode for this instruction is then generated. This function takes a *guest* machine instruction as input, and outputs the LLVM bitcode required to execute that instruction in simulation. A single *guest* instruction is likely to require many LLVM bitcode instructions, and potentially several LLVM basic blocks, so generating this bitcode is not a trivial operation. Care must also be taken to generate code which is efficient, taking into account the strengths and limitations of LLVM and of the information which can be quickly extracted from the SSA-form ISA semantic description.

4.4.2.1 Instruction Predication

One major feature of the ARM architecture which must be addressed is instruction predication. Instruction predication allows individual instructions to be conditional, reducing the number of very small basic blocks and reducing the requirement of a large, accurate branch predictor for high performance. The ARMv5 ISA allows most instructions to be predicated, including branches, arithmetic instructions, and loads and stores. The Thumb-2 ISA takes this a step further by providing *context-based* predication. Rather than the instruction predicate being stored in a constant instruction field, special instructions are used (known as 'If-Then' or `IT` instructions) to load values into a special register. This special register is then checked on each instruction to determine whether the instruction should be predicated.

While ARMv5's 'static' predication could potentially be addressed in an ad-hoc manner in an ISA description (i.e., by wrapping each instruction implementation in an if-then statement), Thumb-2's more complex 'dynamic' predication cannot be efficiently implemented in this way. The fact that instruction predication information is read from a register means that no predicated instruction body can be 'fixed', meaning that much of the benefit of the partial evaluation techniques described above is lost.

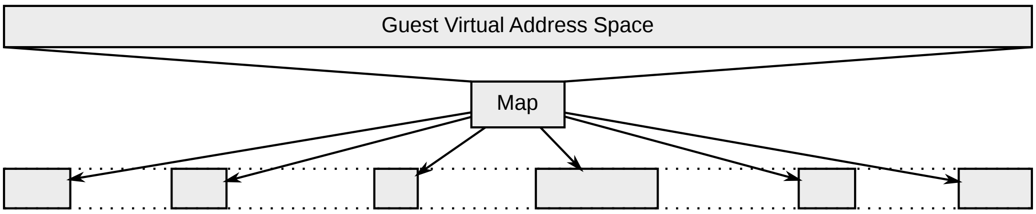
So, in order to implement these predication features, predicated instructions are identified during interpretation. During translation, predication logic is emitted for only these instructions. This takes advantage of the fact that branching into or out of a predicated block is specified to produce Undefined behaviour in the Thumb-2 instruction set. Efficiently handling predication on an architecture where such an operation was permitted would be more complex.

4.4.2.2 Register Accesses

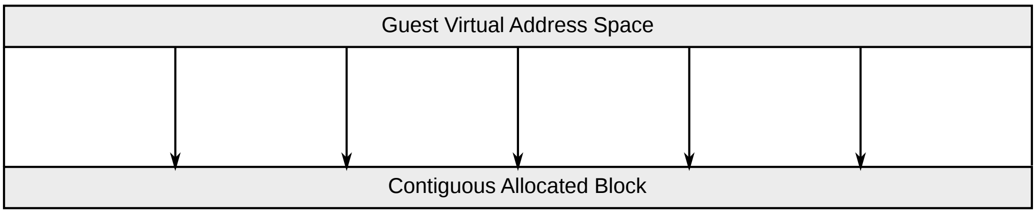
Certainly the most common operation for an instruction to perform is a register access. Most instructions, across most architectures, perform at least two register accesses, and instructions such as ARM's `ldm` and `stm` instructions allow a large number of register (and memory) accesses to be performed in a single instruction. For a high performance simulator, register accesses must therefore be highly optimised, both in interpretation and DBT execution modes.

GenC implements the register file for each described architecture as a structure in memory consisting of 'Registers' and 'Register Banks'. An individual Register might be an architectural status register (such as ARM's CPSR), or, for performance reasons, an individual field in such a register (such as the C (carry), V (overflow), Z (zero), and N (negative) fields of the CPSR). Splitting these fields out in to separate 'Registers' allows for reads and writes to be performed on these registers with a single memory access, rather than requiring that the fields be masked out of a composite register.

Register Banks then typically consist of e.g., ARM and MIPS's General Purpose registers. These are laid out as flat arrays, meaning that accesses can be performed using Base+Offset memory access instructions on the host machine. Aliasing registers (such as the various access modes to x86 registers) are not currently directly supported by the



(a) Demand-paged memory



(b) Contiguous memory

Figure 4.13: Diagram showing contiguous vs demand-paged simulated memory. Demand-paging requires no up-front allocation but memory accesses must be performed via a map or cache of a map, reducing performance.

language, meaning that they require ad-hoc implementation. This must be addressed in future work if a high performance x86 model is to be produced. Multiple register banks are permitted, although the bank accessed by a particular instruction is fixed and so bank-switching operations (such as those produced by an ARM mode change) require backing up and restoring register values.

4.4.2.3 Memory Accesses

High speed memory access is an important factor for high performance simulation. This section focusses only on user-mode memory accesses, since Full System memory access techniques are discussed in Chapter 6.

While memory allocation can be considered a different issue to memory access, the manner in which memory is allocated can influence the possible memory access schemes. In previous generations of simulators, user mode memory was demand-paged by the simulator itself, meaning that whenever the simulator attempted to access a memory page which it had permission to use, but which had not yet been allocated, the simulator would step in and allocate the page. This provides reasonable performance with low overheads compared to more naïve schemes such as using direct maps.

Such a demand paged system is usually implemented using a page table and cache (e.g. [104]). The generated LLVM bitcode for this typically involves either a table or cache look up, including control flow structures since the page may or may not be allocated, and may or may not be present in the cache.

However, a demand paging scheme implemented in a simulator is simply doing the job of the operating system in managing memory. A faster, and less complex approach to allocating memory is to allocate a flat block of memory ‘up-front’ at the start of simulation, and treat this flat virtual memory block as the simulated system’s memory. This provides high performance, as only an addition needs to be performed in order to translate from guest to host addresses. However, allocating a full address space for e.g. a 32-bit guest on a 32-bit simulation host obviously presents difficulties: the full address range cannot be mapped while still leaving memory available for the simulation infrastructure. This is less of an issue when simulating a 32-bit guest on a 64-bit host, which is a fairly common use case. Figure 4.13 compares contiguous memory allocation and demand paging.

Endianness also becomes an issue when simulating e.g. a big-endian guest on a little-endian host (or vice-versa). While the endianness of a user-mode application is typically fixed, many architectures support selecting endianness at boot time, dynamically at run time, or even selecting endianness on a page-by-page basis.

4.4.2.4 Other Operations

In order to perform I/O, user mode programs typically use syscall instructions. In hardware, and in full-system simulation, these instructions cause a synchronous exception, which is then handled by the processor, and the desired operation is performed. This interface is typically well defined, and so it is possible for a simulated system to intercept these syscall instructions in the guest, determine the desired operation, and perform this operation on the host machine. This allows simulated user-mode applications to communicate with the outside world, by reading/writing files, reading and manipulating system state, etc. Trapping these syscalls can be done by having the instruction perform a function call on the host, into an emulation layer tailored to the guest environment (i.e., a MIPS-Linux user mode application will require a different emulation layer to an ARM-Windows CE application).

Finally, while memory accesses are often used to manipulate memory mapped devices, this is not a concern in user mode applications since these devices cannot usually be manipulated in user space and must instead be accessed via OS syscalls. This is also partially true of coprocessors, although floating point units and vector processing units are usually available directly to user mode applications and so require special handling.

4.5 Evaluation

In this section, the overall performance of a *partial evaluation* based DBT module will be assessed. The *partial evaluation* technique will first be compared against a call-based naïve technique using the SPEC and EEMBC benchmark suites. Overall performance will be assessed, and the performance gains obtained by the *partial evaluation* based module will be analysed. The evaluation will be done by executing the SPEC and EEMBC benchmark suites, compiled for ARM, on an x86 *host* machine, using GenSim.

The *partial evaluation* based DBT module will then be augmented with the optimisations outlined in [99], and compared against the state of the art QEMU instruction set simulator. This will demonstrate that despite the GenC DBT module having been generated from a high level description, rather than being hand-written, performance is still extremely competitive. Furthermore, both *Partial Evaluation* modules were generated from the same high-level GenC description, reinforcing the original statement that GenC descriptions can benefit from improvements in simulation technology without requiring modification.

4.5.1 Comparison Against Naïve

In this section the *Partial Evaluation* based DBT module will be compared directly against a Naïve implementation, both in terms of overall performance on the SPEC and EEMBC benchmark suites, and in terms of quantity of generated LLVM bitcode (representing total compilation/optimisation time in the DBT engine).

Figure 4.14 shows a comparison of the performance of both DBT modules when executing the SPEC benchmark suite. A large speedup of at least 2x can be observed on each benchmark, with a 2.5x speedup over a run of the entire benchmark suite. The 462.libquantum and 471.omnetpp benchmarks display a particularly high speedup

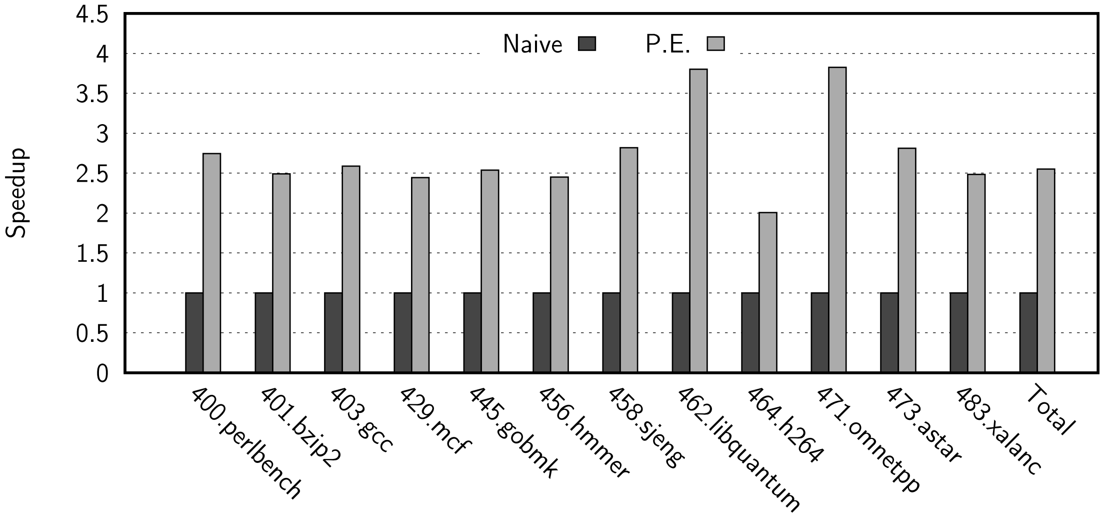


Figure 4.14: Graph showing the performance of the *Partial Evaluation* based DBT module, compared against that of a Naïve call based module. Using the *Partial Evaluation* based module results in a large speedup across all SPEC benchmarks.

of over over 3.5x. The `464.h264ref` benchmark shows a reduced but still significant speedup of approximately 2.0x.

Figure 4.15 shows a comparison of both DBT modules when executing the EEMBC benchmark suite. Although large speedups are observed on most of the EEMBC benchmarks, the range of speedups is much wider. This is likely to be due to the much simpler nature of the EEMBC benchmarks (which are typically small kernels) when compared against the SPEC benchmark suite (which are large applications). Speedups of at least 2x can be observed on almost all of the EEMBC benchmarks, with an overall speedup of approximately 3x.

Considerably less LLVM bitcode is generated when using the *Partial Evaluation* based module when running both the SPEC and EEMBC benchmark suites, as can be seen in Figure 4.16. A large reduction can be seen in the amount of bitcode after light JIT-time optimisations have been applied (O1 bars). This is due to the partial evaluation process eliminating much dead code ‘in advance’ of actual code generation. This is particularly pronounced on the EEMBC benchmark suite: light O1 optimisations when using the *Partial Evaluation* module already result in less LLVM bitcode than when using the naïve module with much heavier O3 optimisations.

Interestingly a significant reduction can also be seen in the bitcode present after both modules have applied heavier (O3) optimisations. This suggests that the bitcode

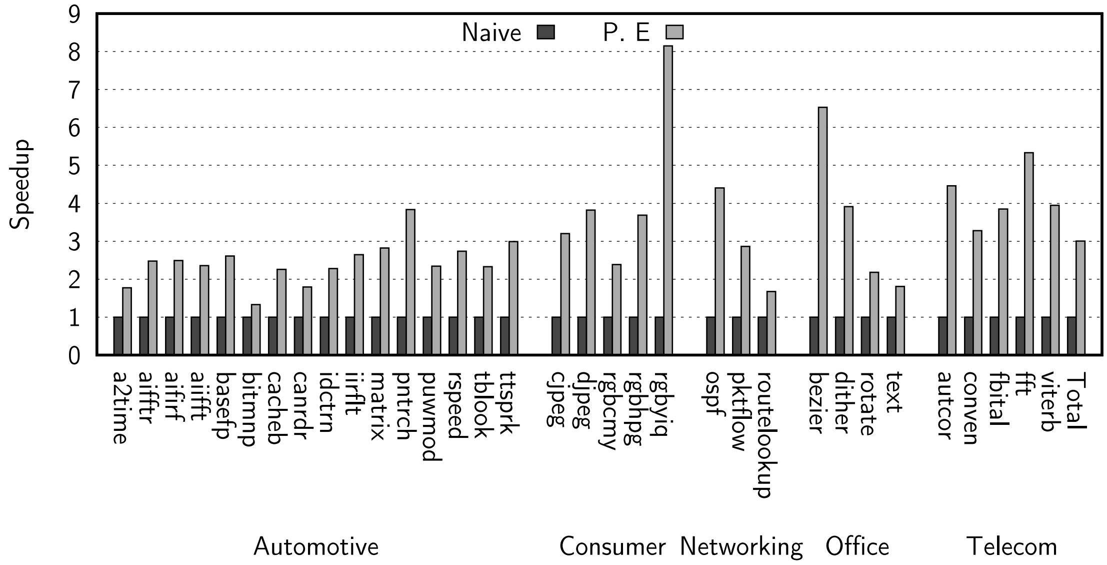
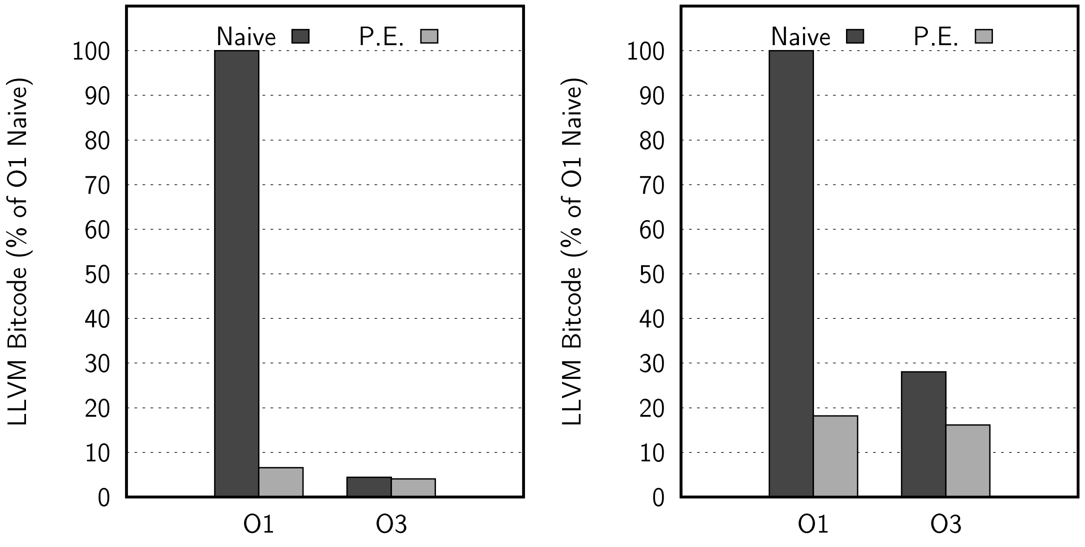


Figure 4.15: Graph showing the performance of the *Partial Evaluation* based DBT module, compared against that of a Naïve call based module. Using the *Partial Evaluation* based module results in a large speedup across all EEMBC benchmarks.



(a) SPEC benchmark suite

(b) EEMBC benchmark suite

Figure 4.16: Graphs showing generated LLVM bitcode size for Naïve and *Partial Evaluation* based DBT modules, using both standard -O1 and -O3 DBT-time optimisation levels. Using the *Partial Evaluation* based module results in a large reduction in the number of generated bitcodes initially generated, and a small reduction in the number once strong optimisations have been applied.

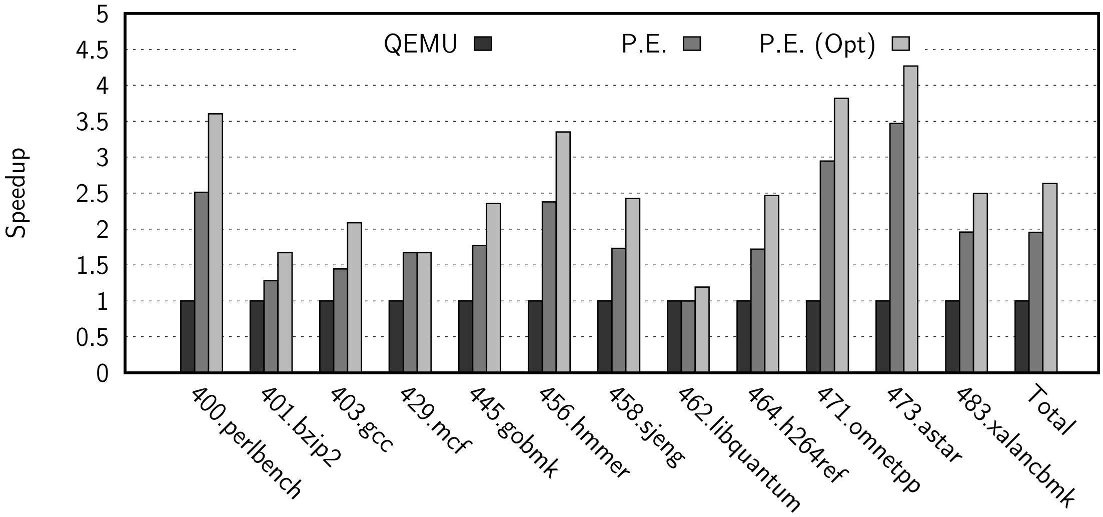


Figure 4.17: Comparison of performance of QEMU and both Novel DBT modules on SPEC.

presented by the *Partial Evaluation* module is more amenable to optimisation than the ‘messier’ bitcode presented by the naïve module.

4.5.2 Comparison Against QEMU

In this section, the novel *Partial Evaluation* based DBT module will be compared against the state of the art QEMU instruction set simulator. The *Partial Evaluation* based module will be considered both in isolation, and with the optimisations described in [99] applied. This will show that although the baseline performance of the *Partial Evaluation* based module is good, when combined with suitable optimisations it can outperform even a hand-tuned simulator such as QEMU.

Figure 4.17 compares the performance of each simulator across the SPEC benchmark suite. Both the baseline and optimised *Partial Evaluation* DBT modules give a significant speedup over QEMU over most of the SPEC benchmarks. The baseline module gives an overall speedup of around 1.9x over QEMU, while the optimised module gives a speedup of 2.7x.

This shows that an automatically generated simulator module, making use of *Partial Evaluation* techniques, can outperform even a hand-written and carefully tuned simulator such as QEMU by almost 2x. Once more aggressive optimisations are introduced (i.e., the custom alias analysis and control flow handling techniques described in [99]), this speedup increases significantly with no change to the ISA description, demonstrating the flexibility of automatically generating a simulator.

One of the frequently described weaknesses of region based DBT is that the translation latency is much higher than that of block based systems. Building up the data structures describing a page of *guest* code, and then analysing, optimising, and compiling that code, is intuitively much more costly than doing the same for a single basic block. However, using the described *Partial Evaluation* techniques, a region based DBT is able to outperform a block based DBT even on highly phase-orientated workloads such as 403.gcc.

4.6 Conclusion

This chapter presented a methodology for generating high speed functional instruction set simulator modules, including a description of how each component of the module can be produced, and a comparison of differing implementations and techniques for the most performance-critical component: the DBT module.

By leveraging existing DBT techniques, and combining them with novel partial-evaluation based code generation, it is possible to produce simulator modules which are easy to describe, but which are competitive with hand written and tuned simulation platforms such as QEMU across a range of large and complex benchmark workloads, especially when combined with the domain-specific alias analysis and control flow handling techniques described in [99]. It has also been shown that this analysis is necessary for high performance, i.e. that a naïve approach results in a large slowdown due to the extreme pressure placed on optimisations provided by the DBT backend.

While the presented *Partial Evaluation* techniques have been applied in the context of instruction set simulation, there are also applications in the implementation of virtual machines and execution environments in general, and in the implementation of bytecode-based domain specific languages. Both of these currently require that the runtime implementation for each bytecode be implemented by hand, which can be complex and error prone, especially when generating native host code. Using the partial evaluation techniques presented above, descriptions of each bytecode could be provided and automatically implemented in an efficient manner.

Chapter 5

Automated Test Generation

5.1 Introduction

While the previous chapter covered the *performance* objective, there is no way of knowing whether the generated simulator is *correct*. Software testing is an area of active academic and commercial research. The cost of releasing broken hardware or software cannot be overstated, both in terms of the immediate financial cost of repairing the problems, and in terms of long term damage to the reputation of an organisation.

In this chapter the importance of automated testing in the context of instruction set simulators is demonstrated. In particular, it is shown that ad-hoc functional testing is not sufficient, covering only 20% of possible ISA behaviours. Instead, a more thorough technique for generating test suites based on ADL descriptions, known as **GenTest**, is demonstrated. Finally, strengths and weaknesses of this technique are identified, and possible improvements that might be made in future work are suggested.

How this chapter is structured

- A motivation for automated testing in instruction set simulators
- A novel approach to automated testing for high level ADL models
- Strengths of this approach and how any weaknesses might be addressed

```
; Base in %rax
; Index in %rdi
; Destination is %ecx

movl 0x20(%rax,%rdi,4), %ecx
```

```
; Base in r0
; Index in r1
; Destination is r2

add r0, #0x20
ldr r2, [r0, r1 lsl 2]
```

```
; Base in $a0
; Index in $a1
; Destination is $v0

sll $a1, $a1, 2
add $a0, $a0, $a1
lw $v0, 0x20($a0)
```

Figure 5.1: An example instruction sequence in x86, ARM, and MIPS assembly. A complex addressing operation can be achieved in one instruction when using the x86 ISA. However, when using ARM or MIPS, two or three instructions are required, respectively. Clearly, architectures incorporating instructions with complex addressing modes require more complex descriptions.

```
; compare the values in r0 and r1
cmp r0, r1
; if comparison failed, add to PC
addne r0, pc, #0x3fc
```

(a) Arm v5 Assembly snippet making use of an addne instruction

```
if(condition_passed(inst.cond))
{
 uint32 imm = ROR(inst.imm, inst.rot);
 uint32 rnval = read_register(inst.rn);
 uint32 rdval = rnval + imm;
 write_register(inst.rd, rdval);
}
```

(b) Actual fragment of semantic action exercised by the addne instruction from Figure (a)

```
if(condition_passed(inst.cond))
{
 uint32 imm = ROR(inst.imm, inst.rot);
 uint32 rnval = read_register(inst.rn);
 uint32 rdval = rnval + imm;
 write_register(inst.rd, rdval);

 if(inst.S)
 {
  write_flag(N, !!(rdval & 0x80000000));
  write_flag(Z, rdval == 0);
  write_flag(C, carry_from(rnval + imm));
  write_flag(V, ovrflw_from(rnval + imm));
 }
}
```

(c) Simplified high-level GenC description of the behaviour of an add instruction in Arm v5

Figure 5.2: Figure (a) shows a snippet of ARM assembly. The addne instruction is predicated, and does not write back to the status flags. When this instruction is considered given the generic add semantic description in Figure (c), the result is the specialised semantic action shown in Figure (b).

5.2 Motivation

As computer architectures become more complex, so too must descriptions of these architectures. Many modern instruction sets feature complex immediate encoding methods and addressing modes. For example, the ARM architecture allows for the second operand of most arithmetic instructions to be bit shifted in various ways, and also supports a large number of addressing modes on memory instructions, including auto-incrementing, indexing, etc. An example of a computation being performed using three different architectures, ARM, MIPS, and x86 can be seen in Figure 5.1.

These complex behaviours increase the chance that errors are introduced into architectural descriptions, in the form of misinterpreted or ambiguous specification documents, edge cases, and logic errors. Small errors in uncommon instructions or instruction paths can easily lead to a description which appears correct, and which will correctly execute many programs, but which fails in certain cases.

In an architecture description, these complex behaviours are described in terms of control and data flow structures. For example, selecting an addressing mode from several possibilities may be done using a switch statement. A particular instantiation of an instruction will only ever exercise a single case of the switch statement as it can only encode a single addressing mode. Optional behaviours may be described using `if` statements, as seen in Figure 5.2.

Many simulators are advertised as being capable of executing complex benchmark suites such as SPEC and EEMBC. These are made up of ‘real-world’ code and consist of applications such as compilers, industrial control and automation, graphics manipulation, etc. While many of these applications are large and highly complex, it can be easily shown that they still do not provide good coverage of the overall instruction space under test (see Figure 5.5). What’s more, many of these applications can easily appear to succeed, even in the presence of serious simulation bugs. For example, the Simit-Arm simulator has been used several times in the literature[37, 46, 75], despite containing several bugs as seen in Section 5.5.2.3.

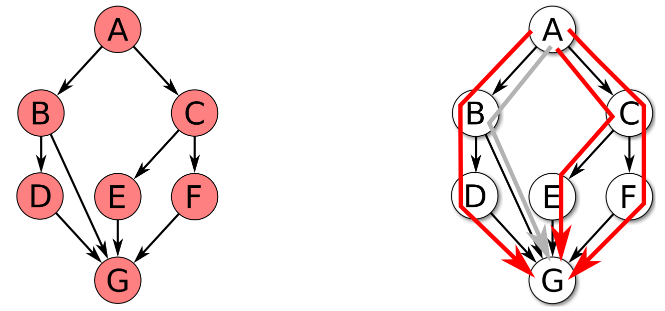
The popular QEMU simulator has a variety of test suites and tools available. As much QEMU development is distributed, the testing of QEMU guests is typically done by third parties. For example, QEMU’s ARM guest is tested by the Linaro project [62] using an automatically generated test suite, produced by a tool known as *risu* [70]. However, this tool requires that the architecture syntax is separately described (and so must be kept in sync with QEMU development), and has no knowledge of what an instruction might do, or what interesting edge cases might exist in the architecture.

In order to be useful, an architecture description must be correct. However, 'correctness' can mean different things in different contexts. For example, if the architecture specification is taken to be a reference, then a correct simulator must behave exactly according to spec (ignoring any behaviours explicitly specified to be unpredictable or undefined). However, if a particular hardware implementation of the architecture is used, then a 'correct' simulator might also include any bugs present in that implementation. A decision must also be made about simulating behaviours which are considered incorrect or undefined. The outcome of this decision will likely depend on the intended use of the simulator. If the simulator is to be used for debugging software under development, then it is important to reproduce bugs caused by incorrect usage of architectural features. However, if the simulator is to be used for design space exploration and will only run programs reasonably believed to be correct then accurately reproducing incorrect behaviours becomes less important.

5.3 Coverage Analysis

Although there are many possible methods for testing correctness of an ISA implementation, many are either inadequate (such as ensuring that one instruction of each type is executed, which may leave certain features or edge cases in each instruction untested) or are impractical (such as enumerating and testing all possible instructions with all possible inputs, which for a single 2-input instruction gives at least 2^{32} times 2^{32} possible inputs). The problem is compounded by the fact that instruction behaviours may be modified by runtime information: a simple example of this is that an instruction may or may not execute depending on its predicate and the current condition flags. A more complex example might be the large variety of shifting operations available to most ARM instructions and their various different edge-case behaviours. This context sensitivity means that such exhaustive testing is completely infeasible.

A useful method for testing correctness must adequately cover all instruction behaviours. It must also be straightforward to perform the tests, i.e. to execute and check the results of each test. Two possible control flow orientated instruction test cover-



(a) Block Coverage (b) Path Coverage

Figure 5.3: A comparison of block and path coverage on an instruction with a simple control flow graph. The paths *ACEG*, *ACFG*, and *ABDG* have been taken. Figure (a) shows block coverage, where the taken blocks have been shaded red. Figure (b) shows path coverage, where the taken paths have been shaded red. Although every block has been touched, there are still a path (*ABG*) which has not been covered, showing the limitation of block coverage versus path coverage.

age metrics can be considered: *Basic Block Coverage* (discussed in terms of general software testing in [49, 50], and *Path Coverage* (discussed in [114]).

While such metrics are well known in existing software testing methodologies, here they are applied specifically to high level ADLs. This provides a different enough context that the best solution for general software testing may not necessarily be the best solution for ADL testing. For example, while a general function under test may have restricted and well structured data as input and output, individual machine instructions have the entire machine state as their context, including multiple registers and register banks, multiple machine modes, and if the instruction accesses memory, typically the entire address space. Conversely, most instructions do not read and write from the same memory locations, so modelling an entire memory system is not generally necessary.

5.3.1 Basic Block Coverage

One of the simpler general-purpose coverage metrics available is *basic block coverage*. Here it is applied to individual instruction behaviours, which may contain branching or looping control flow. Basic block coverage information is collected by instrumenting the high-level instruction descriptions with block profiling code. During execution of each test case, the entry of each block is then recorded, and any blocks which are not executed (and therefore not tested) can be identified for each instruction.

This coverage metric is most useful for testing a generated ISS, rather than for verification of hardware, since the hardware structures involved in the execution of a particular instruction are unlikely to correspond with the control flow structure of the same instruction in an ISS. A block based analysis of an instruction can be seen in Figure 5.3a.

5.3.2 Path Coverage

While block coverage is extremely straightforward to implement, one of the biggest disadvantages is that it does not cover interactions between blocks. The behaviour inside and control flow out of a particular block may be affected by control or dataflow occurring in a previous block. Thus, it would be useful to examine not just the blocks covered in an instruction, but also the paths through it (see Figure 5.3b). These paths can then be used either for further analysis (such as identifying complex instructions), or for generating tests as will be examined later.

One important caveat here is that many modern ISAs include instructions designed to either block-copy data or efficiently manipulate the stack. For example, the ARM ISA includes `ldm` and `stm` instructions, which loop over a bit vector representing the register file in order to determine which registers should be saved or restored. An intuitive way of implementing this instruction in an ISS is using a `for` loop. The ARM `ldm`/`stm` instructions are also described in this way in the ARM Architecture Reference Manual [4]. However, considering all of the possible paths through such a loop can produce an explosion in the number of possible paths with little actual benefit (see Figure 5.4). Care must be taken when testing these instructions as some registers have special purposes. For example, while the ARM `ldm` instruction could be seen as a straightforward memory manipulation instruction, it is also capable of writing into the PC register and thus affecting program control flow.

In order to tackle this problem, each path containing a loop is ‘collapsed’ into multiple paths, each containing the same prologue/epilogue, but with only one iteration through the loop. For example, consider the control flow graph in Figure 5.4b. Suppose the path $ABCEFBCDFBG$ is encountered, which involves two iterations of the loop. This path contains the prologue A, the epilogue G, and the loop iterations $BCEF$ and $BCDF$. Two new paths, $ABCDFBG$ and $ABCEFBG$ (as well as the degenerate path ABG), can be formed by combining the prologue, loop iterations, and epilogue. This

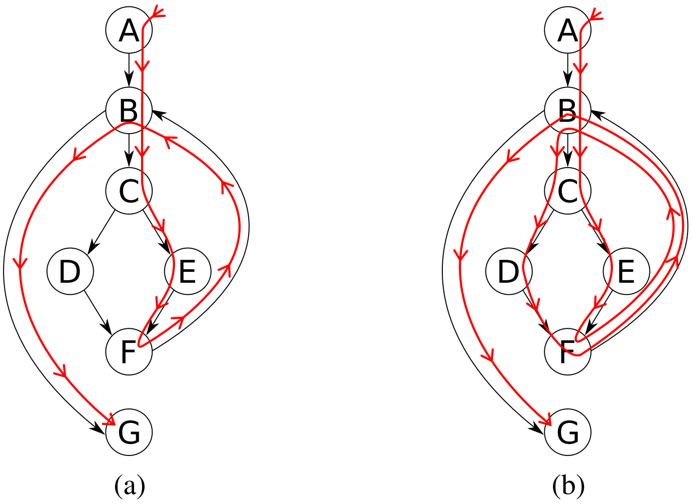


Figure 5.4: An example control flow graph containing a loop. In Figure (a), the path *ABCEFBG* (i.e., one loop iteration) is taken. In Figure (b), the path *ABCEFBCDFBG* (i.e. two loop iterations) is taken. In order to simplify analysis, these two paths can be transformed into *ABCDFBG* and *ABCEFBG*, removing paths with multiple loop iterations. Without this analysis, the ARM ldm and stm instructions would require around 2^{15} test vectors (one for each valid combination of registers), making efficient testing impossible.

method can also be scaled to handle multiple or nested loops although these are rare within real-world instructions. Although this does not capture e.g. all possible combinations of registers for ldm/stm instructions, it hugely reduces the space of instructions required to be tested while still maintaining good coverage from an ISS perspective.

In terms of real instructions, this means that an instruction such as the ARM instruction pop {r4, r5, pc} might be translated into two separate instructions pop r0 (which exercises popping a stack entry into a general purpose register) and pop pc (which would exercise popping a stack entry into the PC and performing a branch). Each of these two behaviours can then then be analysed and tested separately.

An example path-based analysis can be seen in Figure 5.3b. It is possible that ‘impossible’ paths might be discovered, which place conflicting constraints on variables. For example, it is impossible to take a path which includes control flow requiring both X and $\neg X$. In the case of ARM, these might represent edge cases or particular combinations of shift operations and input values which are invalid or which are incapable of setting certain flags. However, this kind of control flow is not problematic and can be easily detected, either by analysis done directly by GenTest, or later on by CVC4. These paths can then be discarded.

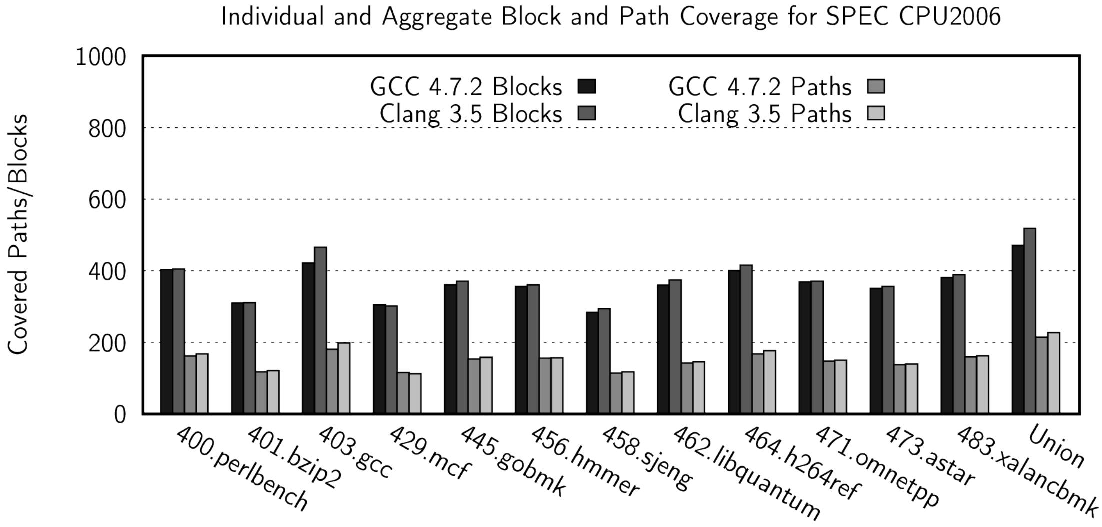


Figure 5.5: Basic block and path coverage of SPEC CPU2006 benchmarks compiled with GCC 4.7.2 and Clang 3.5. While each benchmark may individually cover up to 18% of testable instruction paths, large groups of these overlap, and so only 228 of the 1122 possible paths (20.3%) are covered.

5.3.3 Coverage Results

By instrumenting the ARM model and simulator described in Chapter 4, the path and block coverage across each instruction during the execution of a program can be measured. Figure 5.5 shows the percentages of blocks and paths covered by executing the SPEC benchmark suite.

This instrumentation was performed automatically by modifying the GenC tool to track instruction entry and instruction semantic block execution events. This stream of events was then processed in order to determine which paths had been taken through each instruction during the execution of a particular application.

Despite the large number of diverse benchmarks evaluated, a large portion of the ARM ISA model are untouched by the SPEC benchmark suite. In fact, most benchmarks cover less than 20% of the instruction path space, meaning that even a simulator capable of correctly executing these complex applications might be hiding bugs in 80% of the possible execution paths. In total, 1122 paths were discovered in the ARM model (which includes only the user mode and integer portion of the instruction set).

This poor block and path coverage shows that simply testing a simulator against a commonly used benchmark suite is inadequate for ensuring that it is correct. While an instruction set simulator capable of correctly executing these complex programs might

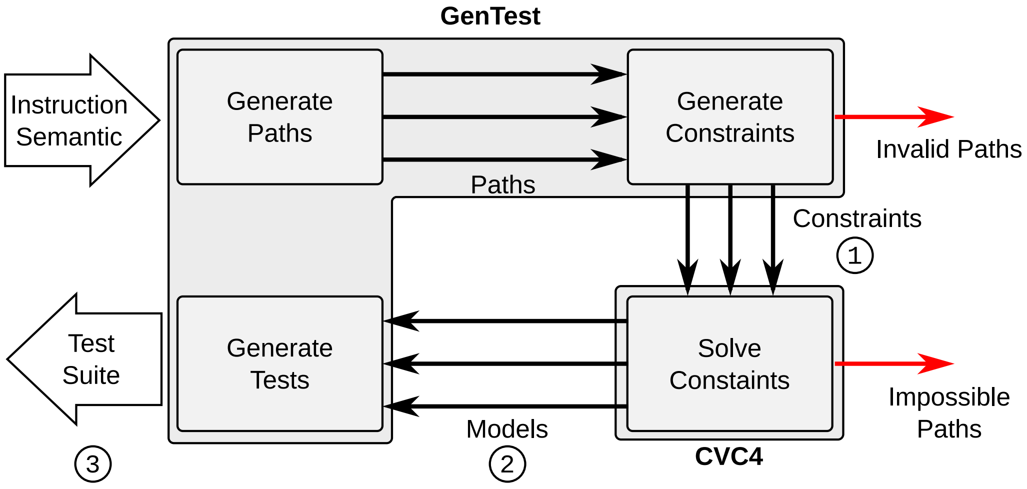


Figure 5.6: High level flow diagram of the operation of **GenTest**. For our example instruction in Figure 5.2, ① can be seen in Figure 5.9a, ② in Figure 5.9b, and ③ in Figure 5.11.

be naïvely seen to be reasonably correct, this poor path and block coverage shows that a more comprehensive testing method (such as that described in Section 5.4) is required.

5.4 Test Generation

By examining the paths not covered by a particular test suite (or by simply considering all available paths through an instruction), new test cases can be constructed. This may require that both static factors (i.e., the precise type of the instruction), and dynamic factors (such as any data which is read from the register file or from memory) are taken into account. For example, overflow is typically calculated using slightly different methods for addition and subtraction (a static factor), and a processor may switch ISA mode (e.g. between ARM and Thumb modes) depending on a value read from a register (a dynamic factor).

The remainder of this chapter describes and evaluates a test generation methodology for high level ADL descriptions, known as **GenTest**. For each individual instruction to be tested, GenTest generates tests in three steps (also outlined in Figure 5.6):

1. Generate a suitable set of constraints for the execution of the particular path through the instruction, which may include constraints on instruction field values, and values written to/from registers and memory.
2. Solving those constraints to generate valid instruction field and initial register values.

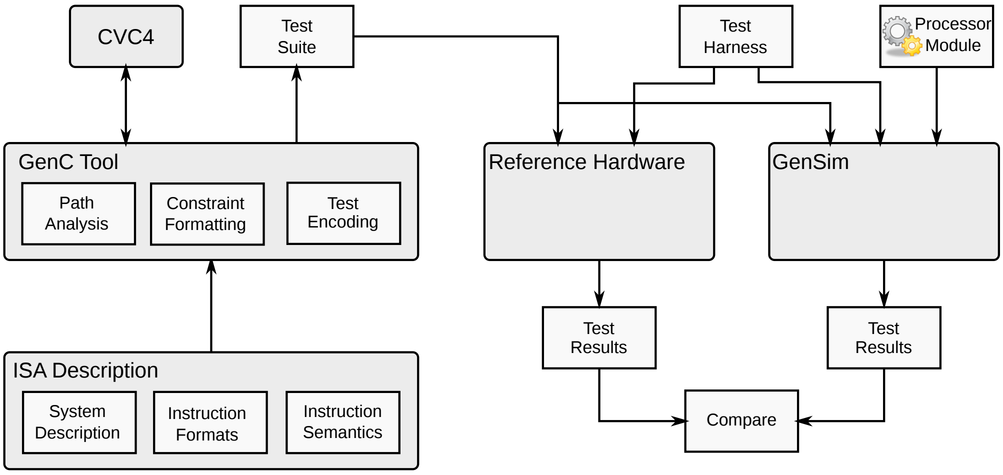


Figure 5.7: Overall flow diagram for generating and executing tests. Rather than generate a Gensim Processor Module, the ISA description is analysed, and tests are created. These tests are then executed using a test harness on both GenSim and reference hardware, and the results are compared.

3. Using the generated instruction field values to encode an instantiation of the instruction to test, and the generated register values to produce a context in which that instruction should be executed.

Once these tests are generated, they can be executed using a test harness application on both GenSim and matching reference hardware. The results of each test (i.e., the state of the register file) can then be collected and compared in order to detect inconsistencies between executing the test using GenSim, or using the reference hardware platform. This is shown in Figure 5.7.

5.4.1 Constraint Generation

In order to ensure that the execution through the instruction under test takes a specific path, constraints can be applied to expressions used by control flow statements in the instruction. For example, given the instruction and path shown in Figure 5.2, the following constraints might be applied:

$$(condition\_passed(inst.cond) = 1) \land (inst.S = 1)$$

This would ensure that the generated instruction takes the missing control flow path outlined in Figure 5.2.

However, there are several problems remaining. Imagine testing a variation on the above instruction, where the destination register is the PC, i.e. the instruction generated is a branch. In this case, the test generation system must ensure that any branches are to a safe region, so that the generated instruction successfully executes when tested (for example, by using a large `nop` slide). This can be addressed by introducing additional constraints: if the destination register is the PC, then the value written must be within a specific range. Although memory protection features could also be used to detect where a branch lands (by intercepting the exception generated when executing code marked ‘no-execute’), this requires hardware support (which is not present in e.g. ARMv5), and still requires that we constrain the instruction branch target to avoid e.g. a self branch, or branching into test harness code.

Since here an ARM model is tested, which supports Thumb interworking, constraints that the simulated core will not switch between ARM and Thumb during a test are also added. These interworking features could still be tested using this test generation technique, but would require special handling, to ensure that a branch into Thumb code lands on a Thumb safe region, and a branch into ARM code lands on an ARM safe region.

Similar constraints must be applied to memory accesses, to ensure that the test does not read or write outside of a safe area. The test generation methodology must also ensure that memory accesses are correctly aligned when testing for architectures which do not support unaligned memory accesses. A safe memory region could be used such that the contents of each memory word in this region is equal to the address of the end of the `nop` slide described above. This ensures that an instruction, which loads a value from memory and then writes it into the PC can be successfully generated.

Crucially, GenTest does not need to model the memory system in detail. Very few instructions make repeated accesses to the same memory location, so unlike in conventional unit test generation frameworks such as CUTE [95], which must track accesses to memory in order to reason correctly about the program under test, GenTest does not need to track values as they change in memory. This greatly simplifies the implementation of GenTest as it does not need to build and use a memory graph nor do any pointer alias analysis.

Generating constraints for each of the instruction control flow paths produces a number of constraint sets. In order to avoid the constraint solver wasting time, constraint sets which are obviously unsolvable are filtered out, e.g. constraint sets including constraints such as $N \neq N$, or pairs of constraints $(A = B) \wedge (A \neq B)$. Although this is not guaranteed to catch all impossible instruction paths, it significantly reduces the number of paths passed to the more expensive constraint solver.

In order to more exhaustively test against hardware, GenTest can also generate additional constraint sets which test control flow and dataflow edge cases. As a practical example, in the case of an ARM shift instruction where the shift amount is encoded as 0, the actual shift amount is 32 in order to allow the encoding of shifts by 32 without requiring an extra bit in the instruction field. By including these extra constraint sets, GenTest can check for edge cases not exposed by the architecture description but which may exist in the reference implementation. These extra constraint sets are also useful for exhaustively testing the correctness of status flags, by attempting to generate constraint sets such that tests are generated which attempt, for each instruction, to set all combinations of flags.

5.4.2 Constraint Satisfaction

Once GenTest has generated a full set of constraints for a given instruction path, those constraints must be solved in order to generate valid instruction fields and register values which explores the particular instruction path.

In order to solve path constraints, GenTest uses CVC4 [11], which is an SMT (Satisfiability Modulo Theories) theorem prover. CVC4 is able to check validity or satisfiability of theories with respect to a number of built in theories (such as theories of linear integer arithmetic or bit vectors). GenTest uses it to solve the constraints over register and instruction field values described above, making use of CVC4's built in theories of integer arithmetic and bit vectors.

While CVC4 is thought to be sound and complete over linear arithmetic and bitvector operations, it may report that it cannot determine the validity or satisfiability of a context with non-linear arithmetic terms. This is not a problem for most instruction semantic descriptions as they do not typically produce different behaviours depending on the result of non-linear arithmetic expressions. However, some instructions such as flag-setting multiplies, divides etc. may have control flow dependent on non-linear arithmetic. GenTest can flag such paths as requiring direct user intervention.

When solving constraints for a particular path P using CVC4, GenTest first converts these constraints into a set of formulae, with each formula representing a constraint or

```
OPTION "produce-models";

%%% Define various flag registers
N : BITVECTOR(1);
Z : BITVECTOR(1);
C : BITVECTOR(1);
V : BITVECTOR(1);

%%% Define general purpose register bank as an array of bitvectors
%%% indexed by another bitvector (the register number)
Reg : ARRAY BITVECTOR(4) OF BITVECTOR(32);

%%% Define instruction fields
insn_cond : BITVECTOR(4);
insn_op : BITVECTOR(3);
...
...

%%% Require that the PC to be the location of the
%%% test instruction location in the test harness
ASSERT Reg[0b1111] = 0hex00008bd0;

%%% Now list the constraints
constraint_1 : BOOLEAN;
ASSERT constraint_1 = ...;
constraint_2 : BOOLEAN;
ASSERT constraint_2 = ...;

%%% Group constraints into a single formula to check
constraintset : BOOLEAN = constraint1 AND constraint2;

%%% Check the satisfiability of the constraints and
%%% produce a satisfying model
CHECKSAT constraintset;
COUNTERMODEL;
```

Figure 5.8: Example of a constraint set formatted as input to CVC4. Registers and instruction fields are represented as bit vectors of varying lengths. Each control flow decision is represented as one or more intermediate formulae. To generate an instruction and register context, CVC4 is instructed to check the satisfiability of these formulae and provide a satisfying model.

```
%%% Input Constraints

constraint_1 : BOOLEAN;
ASSERT constraint_1 = insn_S == 0;

constraintset : BOOLEAN = constraint1;
```

(a) Input constraints for our example in Figure 5.2. In this case, only a single simple constraint is required.

```
%%% Output model

%%% Initial flag values
N : BITVECTOR(1) = 0bin0;
Z : BITVECTOR(1) = 0bin0;
C : BITVECTOR(1) = 0bin0;
V : BITVECTOR(1) = 0bin0;

%%% Initial register file
Reg : ARRAY BITVECTOR(4) OF BITVECTOR(32) = ... ;

%%% Constrained instruction field values
insn_cond : BITVECTOR(4) = 0bin1111;
insn_S : BITVECTOR(1) = 0bin0;

constraint_1 : BOOLEAN = TRUE;
```

(b) The result of using CVC4 on the input in Figure 5.9a.

Figure 5.9: Input constraints (excluding environment) and output from CVC4 when considering the example in Figure 5.2.

set of constraints. These formulae exist in the context of a set of variables, including registers and instruction fields, modelled as fixed-length bit vectors. As mentioned above, the memory system is not modelled in detail. An abstract example of an input file presented to CVC4 can be seen in Figure 5.8.

Once the constraints have been processed, GenTest requests that CVC4 determine the satisfiability of a formula C representing the logical conjunction of each of the constraints. CVC4 is also instructed to generate a model – if the formula C is satisfiable, then this model represents a single example of instruction fields and register values which would produce the desired path through the instruction. These fields and register values are then used to encode a test instruction.

If the formula C is not satisfiable, then the path P cannot be exercised by any instruction. The path may contain conflicting control flow statements or have requirements which cannot be satisfied by the testing environment. A practical example of this can be seen in Figure 5.10. In this example, the path requires that the result of the addition *result* causes both the N and Z flags to be set. However, an integer cannot be both negative and zero, so this path cannot be exercised.

```
int32 a = read_register(inst.Rn);
int32 b = read_register(inst.Rm);
int32 result = a + b;
write_register(inst.Rd, result);
if(result == 0) write_Z(1);                        ①
else write_Z(0);

if(result < 0) write_N(1);                         ②
else write_N(0);
```

(a) An instruction implementation which contains conflicting paths – *Z* and *N* cannot both be set simultaneously. However, an enumeration of all of the paths through this instruction would include a path requiring that both *Z* and *N* are set.

```
%%% Constraint set preamble
...
...

%%% Constraints for conflicting path
constraint1 : BOOLEAN;
ASSERT constraint1 =                               ①
    ((Reg[insn_Rn] + Reg[insn_Rm]) == 0);
constraint2 : BOOLEAN;
ASSERT constraint2 =                               ②
    (((Reg[insn_Rn] + Reg[insn_Rm]) < 0) != 0);

contraintset : BOOLEAN = constraint1 AND constraint2;
CHECKSAT constraintset;
COUNTERMODEL;
```

(b) The CVC4 input generated for Figure (a)

Figure 5.10: These listings show an example instruction implementation which contains a conflicting path, and the constraints generated in order to evaluate the conflicting path. In this case, CVC4 will report that the constraints are unsatisfiable. The `if` statements marked ① and ② in Figure (a) produce the matching constraints in Figure (b).

```
add r3, r9, #60
```

(a) The assembly code for the encoded instruction

Add Instruction Group Template

Cond	0 0	I	0 1 0 0	**S**	Rn	Rd	Shift

Add Register-Immediate Template

Cond	0 0	1	0 1 0 0	**S**	Rn	Rd	Rotate	Immediate

Encoded Instruction

1 1 1 0	0 0	1	0 1 0 0	**0**	1 0 0 1	0 0 1 1	0 0 0 0	0 0 1 1 1 1 0 0

(b) The bitwise encoding of the instruction. Fields with values selected by the constraint satisfaction process are in **bold**. Fields randomly selected are in blue.

Figure 5.11: The resultant instruction, given the constraint set solution in Figure 5.9b

5.4.3 Instruction Encoding

Once a set of constraints for a particular path have been generated and solved, the model provided by CVC4 can be used to encode an instruction and register file context. The instruction can then be executed with the given context in order to exercise the original path to be tested. The instruction encoding process is relatively straight-forward and simply involves inserting the values provided by CVC4 into the appropriate fields in the instruction format for the instruction under test. Once an instruction and context have been generated, they can be stored for later use as part of a test suite.

For the motivating example (Figure 5.2), a set of test cases can be produced, including register context, which will exercise the paths through the instruction not tested so far. For example, the instruction given in Figure 5.11 can be generated given the constraint set solution in Figure 5.9b. Note that the constraint set solution does not select source or destination registers, or an immediate value, since these do not influence control flow through the instruction and so can be selected to have default values (such as zeroes) or at random (as in the example). However, the instruction is guaranteed to exercise the non-flag-setting behaviour of the `add` instruction due to the correct selection of the ‘s’ bit of the instruction encoding. Although this example is quite simple (requiring only one field be assigned a value), paths which make use of the ARM shifting functionality, or which have control flow which is dependent on run-time information, can result in much more complex constraint systems and final instruction encodings.

5.5 Evaluation

This section demonstrates the effectiveness of the test generation techniques used by GenTest, as compared with testing using only standard benchmark suites (as outlined in Section 5.3.3). It is also demonstrated that GenTest is useful in practice, i.e. that it is capable of identifying errors and inconsistencies in the model when compared against a reference platform.

First, the evaluation methodology is described, before the ISA coverage for the generated tests is compared to that of the SPEC and EEMBC benchmark suites. GenTest is then applied to an ARMv5 model, and the results of executing the tests on GenSim, the state of the art QEMU simulator, and the Simit-ARM simulator, using a Raspberry Pi as a reference hardware platform, are presented.

5.5.1 Empirical Methodology

The GenTest testing methodology has been applied to most instructions of the ARMv5 ISA model first described in Chapter 4, excluding those which have special behaviours such as system and coprocessor control instructions. While these instructions could be covered by GenTest, their complex behaviours mean that they produce extremely large numbers of possible paths and thus skew the results. Section 5.5.2.1 compares the path and block coverage of the generated tests against two standard benchmark suites – EEMBC 1.1 and SPEC CPU2006 (integer).

Each test consists of a single instruction encoding, alongside general-purpose register and flag values providing context for the instruction. The output for each test is the general-purpose register file and flag values of the processor after the instruction has been executed. To evaluate GenTest, a test harness was constructed, which loads each test, executes the test, and then records the result. This harness was then used to execute tests generated by GenTest, using the GenSim simulator platform, the state of the art QEMU simulator, and Simit-ARM [83], a fast simulator for the XScale architecture. Although not as well known as QEMU, Simit-ARM has seen significant use in academia, such as in [37, 46, 92].

Since there is plenty of ARM reference hardware available, the same application and test set is also executed, with no modification, on a reference hardware platform (in this case a Raspberry Pi), and the final register and status flag values are recorded.

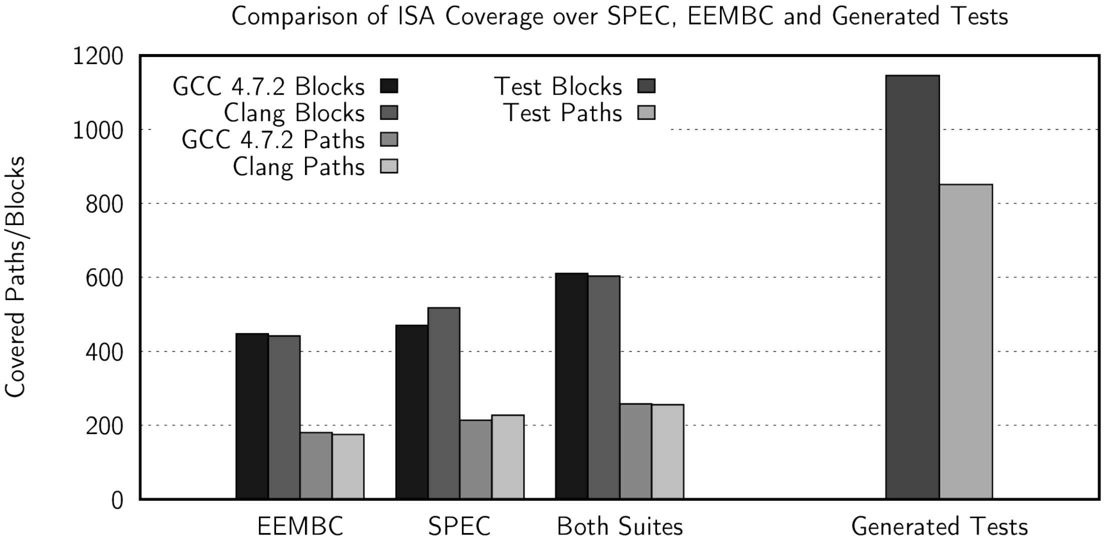


Figure 5.12: Aggregate path and block coverage for EEMBC and SPEC benchmark suites compiled with GCC 4.7.2 and Clang 3.5 for ARMv5T, compared with coverage obtained using GenTest's generated tests. Even when testing with both EEMBC and SPEC benchmark suites, only a small subset of paths are covered, meaning a large number of paths are untested and may produce simulation bugs. In total, 1122 paths exist in the ARMv5T model.

These final register and status flag values can then be compared in order to detect faults in the ARMv5 GenSim model.

This approach to testing is quite heavyweight and requires that some basic functionality in the simulator already exists, as the loading, execution and recording of results is all done in the context of a guest program. In particular, this requires that some instructions are known to be correctly implemented. In situations where this is not the case, test execution could instead be done using debug interfaces to the simulator and reference hardware.

In the case where no such reference hardware is available, such as when prototyping a new architecture or a new extension to an existing architecture, the user would need to manually check the results of each test. However, since the user can be sure that any given path is covered, a minimal test set can be generated, reducing the number of tests results which need to be examined.

Instruction	Reason for Exclusion
bkpt	System-level instruction
cdp	Coprocessor access instruction
ldc	Coprocessor access instruction
mcr	Coprocessor access instruction
mrc	Coprocessor access instruction
mrs	System-level instruction
msr	System-level instruction
stc	Coprocessor access instruction
swi	System-level instruction
clz	Operation does not exist in constraint system
ldm	Looping control flow
stm	Looping control flow

Table 5.1: Instructions excluded from testing. The top group consists of system instructions which can have effects not visible to the architectural model. The bottom group consists of instructions which have not been included for implementation reasons.

5.5.2 Key Results

5.5.2.1 Test Coverage

Once GenTest is applied, tests which cover a much wider range of blocks and paths compared to standard benchmark suites are generated, as can be seen in Figure 5.12. GenTest more than triples the path coverage over the entire SPEC benchmark suite. For this vastly improved coverage, test cases for 851 paths are required as opposed to over 26 trillion dynamic instructions for a full run of SPEC.

However, GenTest is not yet able to handle the complete looping control flow in the `ldm` and `stm` ARM v5T instructions (which account for 56 paths, including paths which produce conflicts and thus cannot be tested). The ‘count leading zeroes’ operation is also not currently supported (the `clz` instruction contains a single path). The remaining non-tested but valid instruction paths consist of system instructions and instructions which always produce an ISA mode switch (which are not tested here). The full list of excluded instructions can be seen in Table 5.1.

Note also that the testing of memory accessing instructions is limited to only their defined architectural behaviours in a user-mode context. The testing of these instructions in a full system context might also include the testing of the exception system and of memory mapped I/O devices. Similarly, care must be taken when testing integer divide or floating point instructions, since these have the capability to produce division by zero and general floating point exceptions. The base ARM architecture does not include such instructions, so this is not a problem in this case.

There are also 214 invalid instruction paths, half of which are detected and discarded by a basic conflict detection system. The rest are passed to, and rejected by, CVC4. Although this could be caused by flaws in the constraint solver, manual inspection of these paths reveals that they do indeed contain conflicting control flow constraints (mainly produced by combinations of edge cases in the ARM shifting operations).

So, in all, we have 851 tested paths (75.8%), 57 valid but untested paths (5.1%), and 214 paths detected as invalid (19.1%) giving us a total of 1122 considered paths. The ISA model analysis and test generation takes approximately 12 seconds. The tests themselves execute in simulation in around 6 seconds. Even without taking into account repeated use of the same test suite, this is much faster than a multi-hour run of the SPEC suite.

5.5.2.2 Generated Tests

Table B.1 shows statistics about the tests which GenTest was able to generate. Although the actual architecture description separates many instruction types by their operand encoding mode (e.g., register-immediate, register-register shifted by immediate, register-register shifted by register), Table B.1 groups these into instruction types in order to declutter the table. The instruction type name is listed in the left-most column. The **Tested Paths** column displays the number of paths for which a test was successfully generated. **Cvc4 Rejected Paths** refers to the paths where the constraint set was unsolvable by Cvc4. The **Average Path Length** and **Average Constraints** columns refer to the average length of each successfully tested path, and the average number of constraints in solvable constraint systems.

So, on average, the `add` instruction has the longest paths to be tested. Other arithmetic instructions also had long path lengths, probably due to the presence of the complex ARM operand shifting behaviours. The instructions with the most complex paths (i.e., those with the largest number of constraints) were memory access instructions such as `ldr` and `str` (and their variants). In these cases, GenTest has introduced additional constraints on the range of addresses allowed to be accessed, which has increased the constraint count compared to the arithmetic instructions. These instructions also have multiple addressing modes (such as post- and pre-increment, immediate and register

Platform	Type	Native Architecture	Faults
Raspberry Pi	Hardware Platform	ARMv6T2	–
GenSim	Generated Simulator	ARMv5T	3
QEMU	Hand-Written Simulator	ARMv7	2
SimIt-Arm	Partially Generated Simulator	ARMv5T	10

Table 5.2: A summary of the reference and test platforms. Although QEMU and the reference hardware platform implement later versions of the ARM architecture, the architecture is fully backwards compatible, so test instructions run on an ARMv6 platform will have the same outputs as those run on an ARMv5T platform.

indexed etc.), each of which requires a specific set of instruction field values (and thus, additional constraints) to select.

On the other hand, instructions which do not have access to the advanced ARM operand shifting functions have considerably fewer paths. For example, the smull and `smlal` (Signed Multiply Long and Signed Multiply Accumulate Long) instructions have only two paths each: one path for a flag-setting behaviour, and one for a non-flag-setting behaviour. However, as will see in Section 5.5.2.3, the low number of possible paths through these instructions does not preclude them from being incorrectly implemented.

5.5.2.3 Test Results

Of course, automated test generation is not useful unless the tests are capable of detecting faults in the ISA model or in its implementation. As mentioned above, the generated tests were run on a reference hardware platform and several simulators (including GenSim). A summary of these test platforms can be seen in Table 5.2.

After generating and running a full set of tests on each platforms, it can be seen that GenSim and QEMU performed well. Although several faults were detected, these were related to behaviours which are specified to be UNDEFINED or UNPREDICTABLE in the ARM Architecture Reference Manual [4]. Specifically, these faults were related to the setting of flags in 64 bit multiply instructions when the instruction attempts to write both halves of the result to the same register.

While faults relating to undefined behaviour are not problematic when executing ‘correct’ programs, there may be issues when perfect simulation is required, such as when verifying hardware or observing the behaviour of malware. In these cases, the correct simulation of even Unpredictable and Undefined operations is extremely im-

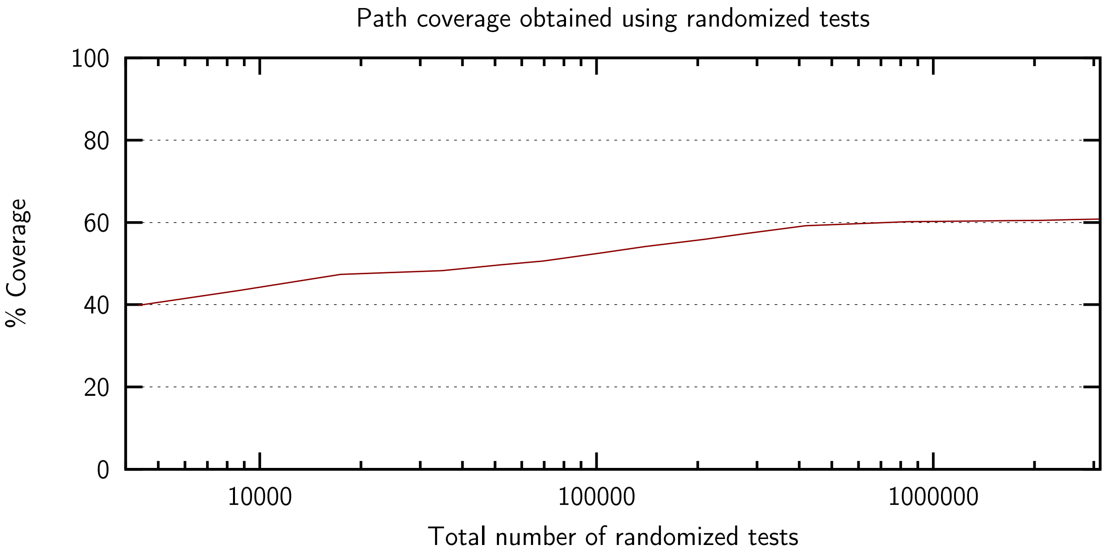


Figure 5.13: Graph showing the number of tests required to obtain a given path coverage by randomised and path-based test generation systems. The result is given as a percentage of paths covered by the path-based test generator. Even when using large numbers of randomised tests (>1,000,000), the path coverage obtained is poor.

portant. In particular, malware may be able to detect that it is running in a simulated environment by attempting these undefined operations and examining the results.

On the other hand, several bugs were discovered in Simit-ARM. Of the 10 faults detected, 4 were related to the setting of the carry flag under certain circumstances, and 1 was related to incorrect implementation of signed multiply instructions. These behaviours are architecturally incorrect, meaning that even correct and 'well behaved' programs would behave incorrectly if these instructions were executed. The 5 remaining faults were due to UNDEFINED behaviour (as with QEMU and GenSim).

The incorrect carry flag behaviour in Simit-ARM is caused by an edge case in the shift-by-register operand mode in the implementation of certain ARM arithmetic instructions. The affected instructions are `bic`, `orr`, `eor`, and `teq`. In these instructions, a left shift of 1 by 32 should result in the carry flag being set (as can be seen on page 450 of [4]). However, executing such an instruction in Simit-ARM does not result in the carry flag being set.

5.5.3 Comparison

In this section GenTest is compared to a simple fuzzing test approach, whereby random but valid instructions, and register contexts for those instructions, are generated and evaluated. Although this scheme is simple to implement, it is incapable of generating tests for memory-accessing instructions, since it cannot guarantee that such an access will be to a mapped or valid region of memory. It is also incapable of generating branching instructions as the branch target cannot be guaranteed to be safe. The fuzz tester used was generated from the same ADL description as used in the rest of this chapter.

Fuzz testing is typically used when testing software libraries and APIs (as in [27]), and systems with well-structured inputs (e.g. [113]). Inputs may be generated completely from scratch, or based on known-correct inputs which have been mutated in some way. In this case (i.e., the testing of an instruction set simulator), the well-structured input is considered to be a single instruction and register context. In order to ensure that valid instructions are generated, the instruction decode and disassembly information provided in the ArchC architecture description is used to guide values for instruction fields. This gives 86 instruction ’templates’ to be tested (excluding branch and memory instructions, as described above). The fuzz tester then generates a number of randomised instantiations of these templates.

In order to compare the randomised test suite against GenTest, the number of instruction paths covered by the randomised test suite using different quantities of template instantiations is recorded. This is presented in Figure 5.13 as a proportion of paths covered by GenTest.

The randomised test suite was also executed on the Raspberry Pi reference platform and the results of the tests were compared. The randomised test suite did not cover any additional paths, nor discover any additional bugs, when compared to the test suite generated by GenTest. As can be seen in Figure 5.13, even large numbers of randomised tests are unable to cover more than 60% of possible paths compared to the path-based test generation. Such large numbers of tests also present a significant problem if reference hardware is resource limited or unavailable, or if simulation speed is slow such as in cycle accurate simulation.

5.5.4 Strengths and Limitations

While GenTest has been useful during the development and testing of the GenC processor models, there are several limitations. The main limitation is that the system cannot test anything which has not been described. For example, it would be impossible for this system to test the behaviour of an edge case which exists in hardware but which has not been described in the processor model, as the test generation system does not know that the edge case exists.

This is true of all test generation systems which rely on analysing either an abstract processor model, or directly analysing the source code of an ISS. While comprehensive randomised testing can (eventually) detect such behaviours, randomised testing is not a practical solution to the problem, as shown in Section 5.5.3.

This system also requires that the instruction decoding and disassembly information provided by the model is correct and reasonably complete. However, this portion of the architecture description is much simpler to exhaustively test than the instruction semantics. Very little work has been done in this specific area and instruction decode systems are usually assumed to be correct. This does cause problems in architectures such as ARM, where 'gaps' in the instruction encoding are gradually filled by newer architecture versions and instruction decode becomes increasingly complex. For example, the 'Media Instruction Space' is provided for multimedia acceleration instructions. This space was empty in ARMv5, and has been populated by a considerable number of instructions in ARMv6 and above.

GenTest is also unable to model instructions which make multiple accesses to the same memory location. Instructions which atomically access memory (such as swap and compare-and-exchange instructions) are usually implemented in this way and so GenTest is unable to test these kinds of instructions. This is not a significant issue in practise as this presents a minority of instructions. Two such instructions are present in ARMv5 (`swp` and `swpb`) and these instructions are deprecated in newer versions of the architecture, which uses a Load-Linked/ Store-Conditional synchronisation model rather than a Compare-and-Swap model [4].

Instructions which modify system state in some way (e.g., switching between ARM and Thumb mode, raising a software interrupt, breakpoint instructions etc.) are also not practically testable by this system although the state-changing nature of this instructions often mean that they cannot be effectively tested in isolation and thus require larger and more detailed test suites. This type of instruction also typically interacts with components of the system (such as syscall emulation models, system components, etc.) which are not included in the GenC simulation model and are instead implemented using e.g. C++ classes.

The current implementation of this system also does not support generating tests for instructions which contain loops. This is an implementation problem rather than a problem with the concept itself. Since tests are generated only for paths which contain one loop iteration, many of the issues surrounding tests for such structures do not apply here. However, this is not a significant concern as these kinds of instructions are relatively uncommon in RISC architectures, other than in specific cases such as the ARM `ldm` and `stm` instructions.

5.6 Conclusion

This chapter has demonstrated that an ad-hoc, benchmark-driven approach to testing for instruction set simulators is not sufficient to obtain good test coverage or to detect many bugs, by using novel intra-instruction path profiling techniques. The coverage of both the SPEC 2006 and EEMBC benchmark suites was examined, and it was demonstrated that despite the wide range of complex workloads, neither suite produced good ISA coverage and thus are both unsuitable for use as comprehensive test cases.

A novel test generation infrastructure for testing instruction semantic descriptions was presented, which operates by enumerating all paths through an instruction, formulating a set of constraints based on the instruction and context leading to this path, and then solving those constraints using a standard constraint satisfaction package. These generated tests are able to obtain improved coverage of the instruction space compared with ad-hoc testing, and several bugs in two popular instruction set simulators were detected, as well as in the architecture description described in Chapter 4.

Although this chapter has shown the presented test generation methodology to be effective, several key limitations based on dataflow bugs and ISA syntactic bugs were identified, and solutions to these problems were suggested. Finally, GenTest was compared against a 'fuzzing' based tester, and it was demonstrated that even large numbers of randomised tests are unable to obtain path coverage comparable to the targeted test suites produced by GenTest.

Both the demonstrated coverage analysis and test generation techniques contribute significantly to the *correctness* objective, as it is now possible to determine the effectiveness of test cases applied to the GenSim simulator, and to generate new, useful test cases to cover any gaps.

Chapter 6

Efficient Full-System Simulation

6.1 Introduction

Up until now we have concerned ourselves mainly with so-called ‘user-mode’ simulation, where OS related operations are trapped and handled by our simulator infrastructure. In this simulation mode, no interrupts or exceptions are handled by the simulated processor. For all intents and purposes, the simulated program ‘believes’ itself to be running in a typical Linux environment.

Although this mode of simulation can be useful for testing and debugging user-mode software, it is impossible to simulate a full operating system or bare metal runtime environment. For this we need ‘full-system’ simulation, where a full *guest* machine is simulated, including memory mapped I/O devices, a Memory Management Unit (if one exists in the real *guest* system), and exception and interrupt mechanisms. Full system simulation is much more capable than user mode simulation, since it allows for the simulation of a full operating system rather than just individual applications, but typically comes with a performance penalty, mainly due to the increased complexity of memory accesses due to the presence of an MMU.

How this chapter is structured

- We begin by discussing the additional requirements for full system simulation.
- We then discuss improvements to interrupt handling in functional simulation.
- Finally, we discuss techniques for improving the performance of memory accesses in the context of a simulated MMU.

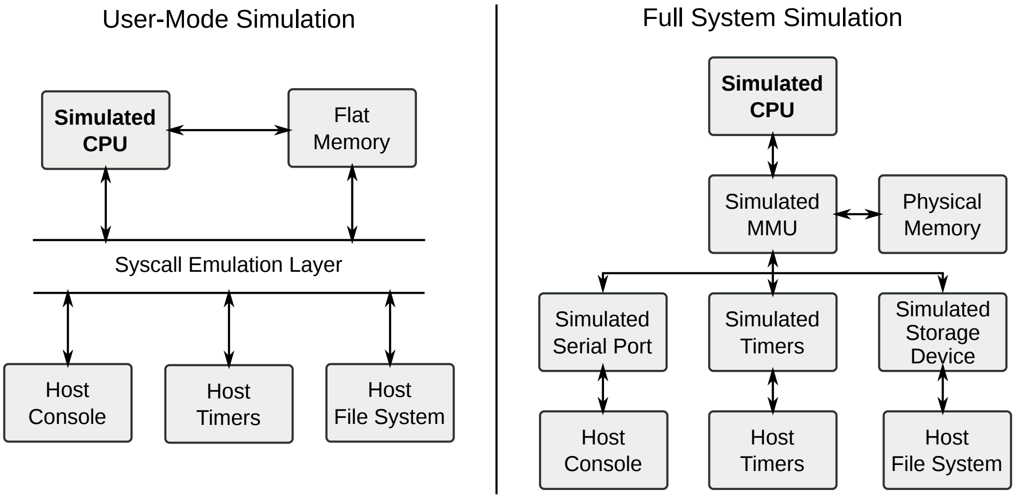


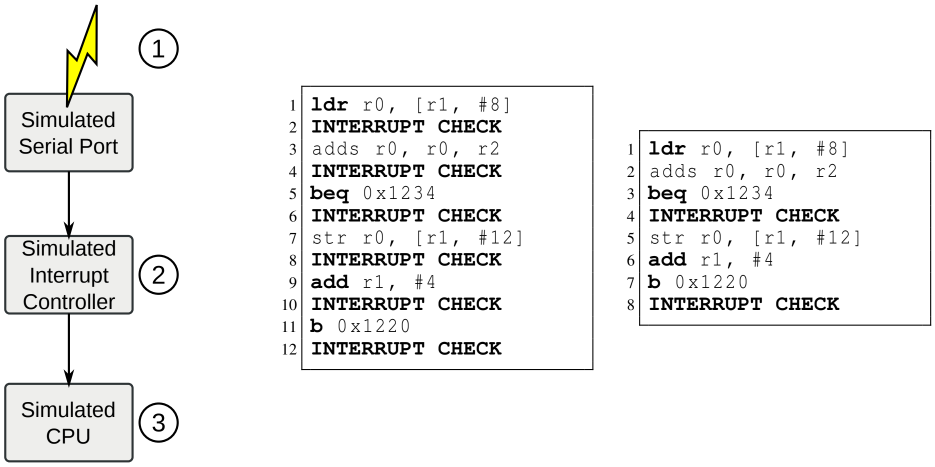
Figure 6.1: Diagram comparing user-mode simulation with full-system simulation. While user mode simulation involves using a system call emulation layer to abstract away details of device accesses and memory management, full system simulation must address these issues directly.

6.2 Full System Simulation

Simulation of a user-mode program is analogous to running a program using a managed runtime environment such as Java or the Microsoft CLR. The program is encoded in a form which must be interpreted or translated prior to execution, and a well-defined interface is used to communicate between the program and the 'host' environment. The program begins executing at a defined entry point, and the system usually has a defined termination condition (i.e., some kind of 'exit' syscall).

However, full-system simulation is much more complex. We must now worry about the *guest* system architecture, such as the interrupt and exception model, the virtual memory model, and external devices which now provide the interface between the simulated system and the host. Execution typically begins with the system taking a 'reset' exception, and there may be no explicit termination condition for the system. Full System simulation also requires that we simulate the individual hardware devices in a system, rather than providing a system call interface, as can be seen in Figure 6.1.

Additionally, while most instruction set architectures have a conceptually similar instruction execution model (i.e., most architectures fetch, decode, and execute instructions which perform arithmetic and/or memory operations), system architectures are much more diverse. System configuration might be done via auxiliary registers (as with ARC), via a system coprocessor (as with most ARM architectures), or via a memory



```
ldr r0, [r1, #8]
INTERRUPT CHECK
adds r0, r0, r2
INTERRUPT CHECK
beq 0x1234
INTERRUPT CHECK
str r0, [r1, #12]
INTERRUPT CHECK
add r1, #4
INTERRUPT CHECK
b 0x1220
INTERRUPT CHECK
```

```
ldr r0, [r1, #8]
adds r0, r0, r2
beq 0x1234
INTERRUPT CHECK
str r0, [r1, #12]
add r1, #4
b 0x1220
INTERRUPT CHECK
```

(a) The flow of interrupts through a simulated system

(b) Interrupt checking after each instruction

(c) Interrupt checking after each branch instruction

Figure 6.2: Interrupts must be propagated through the simulated system (Figure (a)). Once they reach the simulated CPU, there is some leeway in when they are handled (Figures (b) and (c)).

mapped configuration device (as with ARM Cortex-M series devices). Virtual memory models are similarly diverse: the ARC architecture directly exposes its TLB, requiring that the operating system manage it directly, while x86 and ARM systems manage the TLB as a cache and perform page table lookups automatically. This image is further complicated by the many differences between individual versions of the same architecture: early versions of the x86 architecture such as 286 share very few similarities with modern x86-64 architectures.

6.3 Interrupt Handling

As we are simulating a full system, we must now concern ourselves with devices in our simulated system which are external to the processor, such as timers, serial communications devices, graphics devices etc. These devices are typically attached to a bus and accessed by manipulating memory mapped hardware registers. External devices may also raise interrupts which must then be serviced by the processor. Interrupts are typically handled via an interrupt controller, which may itself be an external device with memory mapped registers.

In an example system (Figure 6.2a), some data might be received by the serial port device ①. This might cause an interrupt to be generated, and propagate to the interrupt controller ②. Depending on the configuration of the interrupt controller, the interrupt might be masked or manipulated in some way, and eventually an interrupt line into the processor itself may be raised ③. The processor will check for interrupts at instruction boundaries, provided that the processor is in a state where interrupts can be taken, and the interrupt will eventually be serviced.

In simulation, the architectural behaviours involved in handling interrupts must be faithfully reproduced, both at the system level and in the processor itself. The internal behaviour of the interrupt controller must be modelled as much as is required to obtain architecturally correct behaviour. However, due to the asynchronous nature of external interrupts, there exists some leeway in *when* an interrupt is serviced when performing functional simulation. Since no precise timing behaviour is considered in the simulation systems outlined in this thesis (i.e., GenSim is not intended to be a cycle accurate simulator), the handling of interrupts can be slightly delayed, provided any interrupts which arrive at the core are eventually serviced. This means that we do not need to check for interrupts at every instruction boundary (Figure 6.2b). The common approach is to check for interrupts approximately at each basic block (Figure 6.2c) or trace boundary (depending on how the simulator is implemented). This is critical in region based DBT systems such as GenSim, since these rely heavily on inter-instruction optimisations, which are inhibited by frequent interrupt checks.

6.4 Memory Management

One of the largest costs when simulating a full system is that of accurately modelling the potentially complex behaviours of the *guest* Memory Management Unit (MMU). The MMU is the primary hardware structure required to implement virtual memory in the *guest* system. Different architectures tend to have vastly different virtual memory architectures, and thus they require different strategies for MMU simulation. A particularly notable example is the x86 virtual memory system, which has been shown to be Turing complete independently of the CPU core itself [6].

In this Section, we will briefly introduce Virtual Memory, and the complexities it presents in a simulation environment, before discussing various techniques for ef-

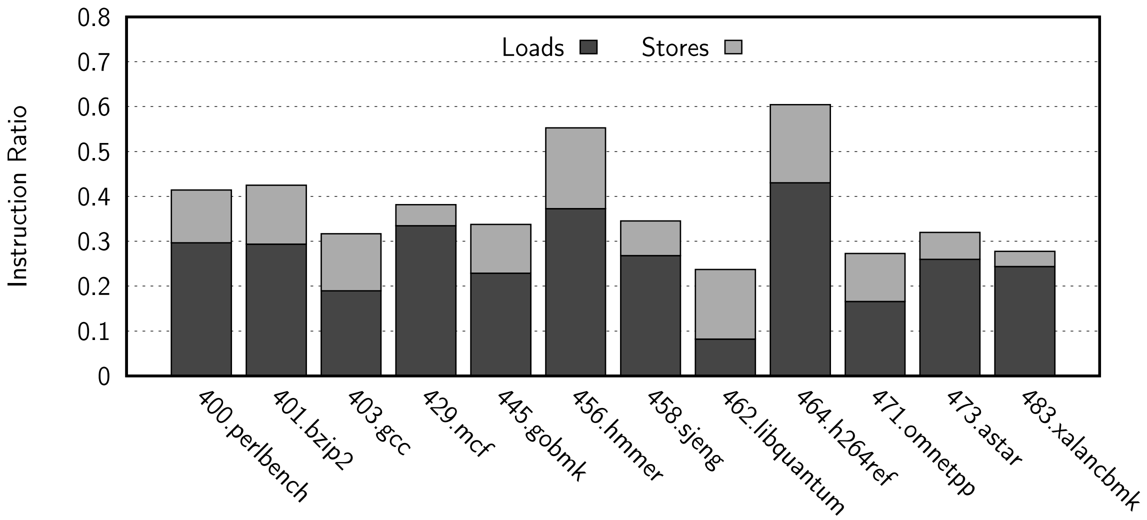


Figure 6.3: Proportion of memory instructions in SPEC benchmark suite. Instructions are counted only once, even if they cause multiple memory accesses, such as with the `ldm` and `stm` instructions.

ficiently handling memory translations and virtual memory systems. Management of memory, and efficiency of memory access, is extremely important for the overall performance of a simulator. Memory accesses typically constitute 30% - 55% of all instructions (see Figure 6.3), and so simulated virtual memory translations, and the conversion of a *guest* physical address to a *host* virtual address must be performed as efficiently as possible if a high speed simulation is desired.

6.4.1 Introduction to Virtual Memory

Virtual Memory is a technique implemented in hardware and software used to enforce separation of user programs and to allow paging techniques to be used. This is achieved by using a Memory Management Unit to translate *virtual* memory addresses into *physical* memory addresses on a page-by-page basis. Contiguous *virtual* memory pages do not need to map to contiguous *physical* memory pages, and paging techniques mean that a given *virtual* memory page need not be present *physical* memory at all. This allows a system to give each process its own complete view of a flat and contiguous address space (see Figure 6.4).

In order to describe how *virtual* addresses should map to *physical* addresses, a *page table* is used. This is a data structure stored in memory, which contains a complete description of how the address for each memory page should be translated. Often a

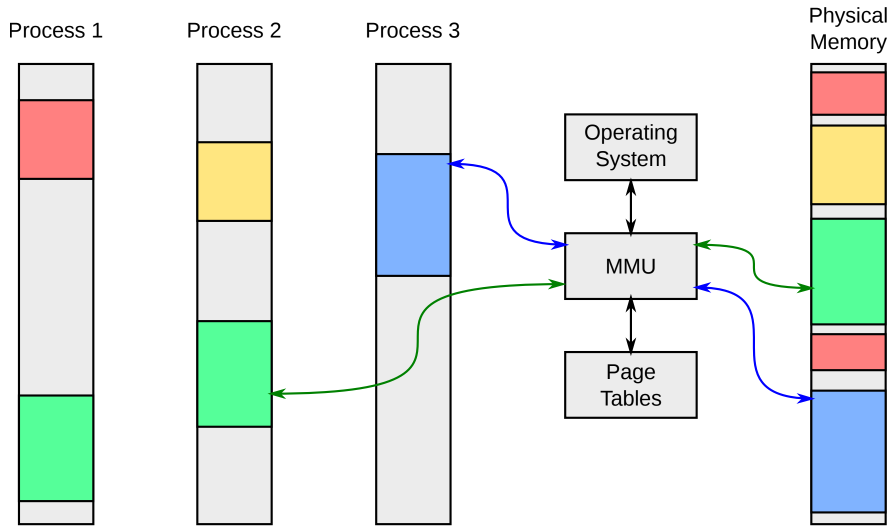


Figure 6.4: When using Virtual Memory, each process has an isolated memory image. Physical regions can be mapped into multiple virtual regions (Green), and contiguous virtual pages do not need to map to contiguous physical pages (Red).

multi-level page table is used in order to save memory space. A page table can also contain additional information, such as describing the *permissions* required to access a particular page of virtual memory. A page of memory may be marked as readable, writeable, executable, or any combination of these. If an invalid operation is attempted on a page of virtual memory (for example, writing to a page which is not marked as writeable), then a memory system exception (usually called a *page fault* or *permission fault*) is produced. This is typically implemented as a synchronous exception on the processor, which, in contrast with external interrupts, must be handled immediately.

When encountering an instruction which should access memory, a processor which has virtual memory enabled will first compute the virtual address to be accessed, and then use the MMU to convert this virtual address into a physical address. The access permissions described by the page table are checked, and then this physical address is then used to access the memory system. Since doing a page translation via the MMU might require several memory accesses in order to traverse the page table, a Translation Lookaside Buffer (TLB) is often used as a cache of page translations.

Although virtual memory is often used in conjunction with several other techniques such as swapping, where increased memory pressure causes some pages of memory to be temporarily moved out of memory and onto a backing store, these are implemented at the OS layer rather than in hardware, so they should operate correctly in our simu-

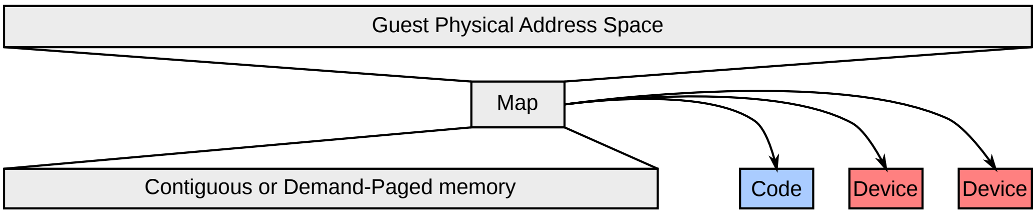

Guest Physical Address Space
Map
Contiguous or Demand-Paged memory
Code
Device
Device

Figure 6.5: When simulating a full system, memory mapped devices and self modifying code must be correctly dealt with. Although a scheme such as those outlined in Figure 4.13 can be used to allocate *guest* physical memory, device and code regions must still be tracked.

lator provided that we correctly implement the required hardware features for virtual memory.

6.4.2 Memory in Simulators

Several approaches exist for the implementation of memory systems in instruction set simulators. The most popular, and most flexible, is to demand-allocate pages of *guest*-physical memory as they are required by the guest. This has several advantages over alternative approaches such as allocating the full guest address space up-front, which typically requires a *host* system with a larger address space than the *guest* system. In particular, this up-front allocation scheme can be difficult to implement when simulating 64-bit *guest* systems as there is not enough *host* address space available to hold even a reasonably restricted 64-bit *guest* address space.

However, in user space simulation of a 32-bit *guest* system, a demand-paged implementation introduces unnecessary overheads, as each *guest* memory translation must be done via a simulated page table. Using an up-front allocation scheme avoids these overheads, instead relying on the *host* OS to manage memory effectively. Flatly allocating the 2-4GB of memory required in this approach is not typically a problem, as the *host* OS will itself only reserve these pages, and only allocate them to the simulator as they are required.

The situation becomes more complex when full system simulation is introduced. The simulated system now might include memory mapped devices and virtual memory systems. All *guest* memory accesses must now have their addresses translated from *guest virtual* to *guest physical* addresses, and these physical addresses might point to a memory mapped device rather than a memory page. When using a DBT system, care

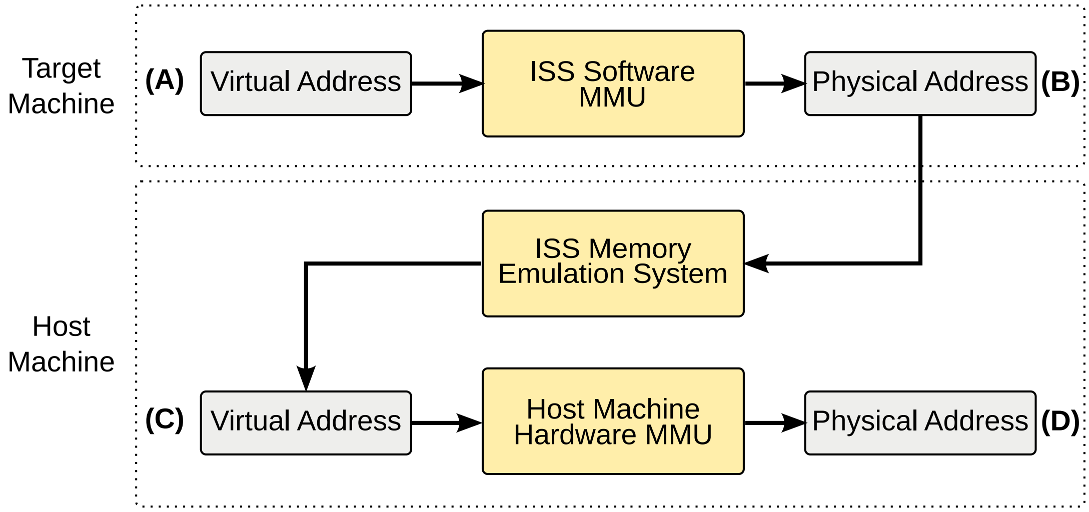


Figure 6.6: A diagram showing the progression of a *guest virtual* address into a *host physical* address. Initially a memory access is made by a guest application with a *guest virtual* address **(A)**. The ISS software MMU uses the guest page tables to translate this address into a *guest physical* address **(B)**. This address is then further translated into a *host virtual* address **(C)**, representing the actual storage location on the host machine before being served by the underlying host machine hardware, and translated into a
host physical memory address **(D)**

must be taken to invalidate any translated code pages which are modified. Figure 6.5 gives an outline of full system memory simulation.

Due to the large number of memory accessing instructions encountered in most applications (as well as the instruction fetch operations which are also present on real hardware), it is extremely important to handle *guest* memory accesses as efficiently as possible in order to obtain good simulation performance.

6.4.3 Virtual Memory In Simulation

When running natively, virtual memory addresses are translated into physical addresses using the Memory Management Unit. In simulation, an additional step is present, as *guest* physical addresses must then be further translated into *host* virtual addresses. This process is shown in Figure 6.6.

To perform *guest* virtual to *guest* physical address translation, each of the structures required for hardware virtual memory must be modelled. For cycle accurate simulation, this includes detailed models of the MMU (including any state machine it implements in order to perform page table lookups), as well as a detailed model of the TLB, which may have a complex or pseudo-random replacement policy. In functional simulation, the TLB is often not accurately modelled, and can often be omitted entirely, provided that the architectural behaviour of the system does not depend on the presence of the TLB.

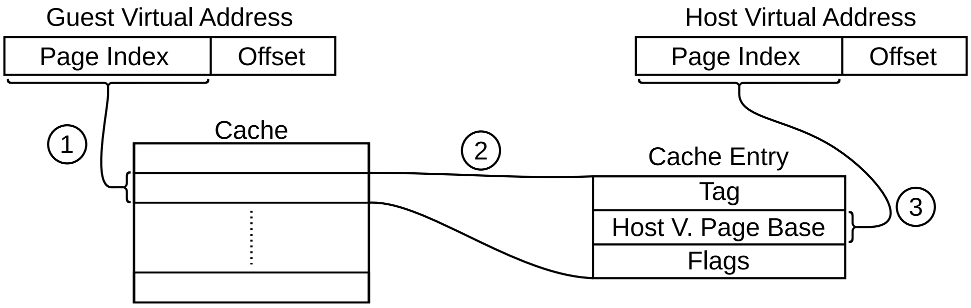
Somewhat surprisingly, omitting the TLB in simulation comes with a cost: traversing the *guest* page table for each memory access is very expensive, especially if a lookup is required for instruction fetches. Just as the TLB is used in hardware to amortise expensive page table lookups, a similar structure can be used to accelerate simulated memory accesses. Other techniques can also be used to eliminate or reduce translation overheads.

Virtual memory simulation can also be extended to allow efficient detection of self-modifying code. Detection of self-modifying code in a simulator using Dynamic Binary Translation has similar costs to support for virtual memory as each *guest* memory write must be checked to see if it modifies a translated region of the *guest* application. Self modifying code is not particularly prevalent in ‘normal’ software outside of dynamic language runtimes such as Java, Microsoft’s CLR, and the JavaScript runtimes embedded in modern web browsers. However, when simulating a full operating system, application code can be arbitrarily loaded and moved by the operating system and so support for this becomes critical.

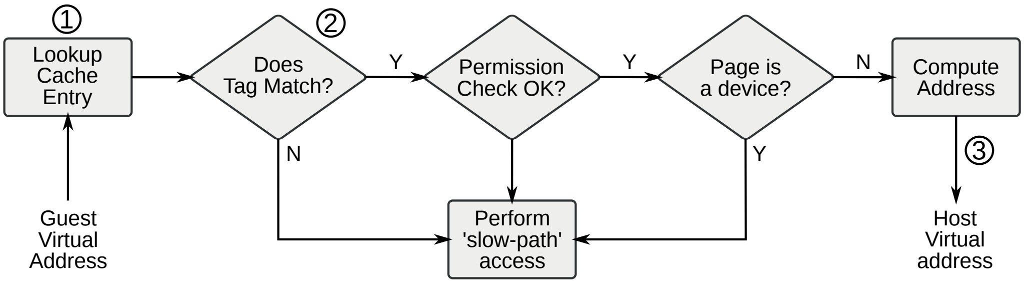
6.4.4 Software Cache Based Approaches

The obvious way to avoid the cost of a full MMU translation, including a page table walk, on each *guest* memory access, is to implement a simulated structure similar to a TLB. On systems where the contents of the TLB is not architecturally visible (which includes most commonly used architectures such as x86 and ARM), the structure of this cache does not need to match that of a real TLB. A software cache can be made much larger than would be feasible for a hardware TLB, although typically the associativity is much lower. Hardware TLBs are typically highly associative, which can be implemented efficiently in hardware but such parallel lookups are expensive to perform in software.

One disadvantage with using this cache based approach is that memory mapped devices, as well as the potentially complex permission systems must be correctly handled. When implemented in a DBT, this means that a large amount of code, including several additional basic blocks, must be emitted for each memory accessing instruction.



(a) Data structures/data flow



(b) Control flow

Figure 6.7: An overview of the data and control flow when using Cache based memory translation. Figure (a) shows a high level overview of the data structures and data flow involved, while Figure (b) shows a flow diagram of control flow for a memory access.

The typical execution flow when evaluating the cache can be seen in Figure 6.7. In ①, the page index is extracted from the *guest virtual* address. This is converted to a cache index using a modulo operation. Once the correct entry is found, the entry's tag is compared against the page index ②. If the tag does not match, a 'slow-path' translation is performed and the entry replaced. If the tag does match, a permission check must be performed using the flags stored in the cache entry. If this check fails, a memory fault is triggered and the access is aborted. Finally, the *host virtual* page base is combined with the *guest virtual* page offset to form the final *host virtual* address ③.

Several caches, each intended for different classes of memory access (e.g., reads, writes and fetches) or privilege levels (user/kernel mode) might be kept (such as the 'Page Translation Caches' in [104]), which reduces but does not eliminate the problem of checking permissions. Keeping additional caches also increases data cache pressure on the *host* machine, means that pages which are both read and written must be loaded into the cache twice, and increases the cost of invalidations.

Several operations require the invalidation of some or all of this cache. *Guest* TLB manipulation operations (such as the TLB flush often required when a *guest* OS context switch occurs) require that some or all of the cache is invalidated. Similarly, the cache may require invalidation when new regions of code are translated into *host* machine code, in order to ensure that any *guest* self-modifying code is correctly detected and those translations invalidated.

6.4.4.1 Generated Code

Figure 6.8 shows an example of the LLVM code required to implement a memory access when using the cache-based approach to memory translations. A fairly considerable amount of code is required, since we must:

1. Look up the correct entry in the cache
2. Check that the cache entry is valid and compare the tags
3. Perform any necessary permission checks
4. Determine if this page points to a memory mapped device
5. Perform a 'slow path' lookup if the cache entry is not valid
6. Translate the *guest* virtual address to a *host* virtual address if the memory access should succeed
7. Raise a memory exception if the access should fail

As can be seen from Figure 6.8, this requires a considerable number of additional basic blocks. Although this code can be extracted into a separate function, the frequency of memory accesses means that it should be inlined. This places a large amount of pressure on both the applied LLVM optimisations, and on the back-end code generator.

6.4.5 Efficiently Handling Invalidations

There are several techniques possible for handling partial and full invalidations of the structures required for memory translation. These invalidations are required in a variety of contexts, including *guest* TLB invalidations and page table modifications, as well as when required by the simulation infrastructure, such as when DBT translations occur.

We will discuss three possible schemes for invalidation:

- A Naïve scheme, where each entry in the structure is overwritten by default entries.
- A Generational scheme, where each entry carries a generation which is used to determine if it is valid.
- A Lazy memory protection based scheme, where *host* memory protection is used.

```
  ...
  %reg_val = load %rb0_r11
  %addr = add %59, -32
  %addr_shift = lshr %addr, 12
  %addr_index = and %addr_shift, 2047
  %addr_tag = and %addr, -4096
  %tag_ptr = getelementptr %tlb, 0, %addr_index, 0      ①
  %tag = load %tag_ptr
  %tag_matches = icmp eq %addr_tag, %tag                ②

  br %tag_matches, label %permission_check_block, label %not_matched

permission_check_block:
  %perms_ptr = getelementptr %tlb, 0, %addr_index, 4    ③
  %perms = load %perms_ptr
  %cpu_mode_ptr = getelementptr %cpu, 0, %mode_index
  %cpu_mode = load %cpu_mode
  %allowed = icmp gte %cpu_mode, %perms

  br %allowed, label %device_check_block, label %exit

device_check_block:
  %flags_ptr = getelementptr %tlb, 0, %addr_index, 3    ④
  %flags = load %flags_ptr
  %8 = and %flags, 1
  %9 = icmp eq %8, 0
  br %9, label %memory_access, label %device_access

not_matched:                                            ⑤
  %rslt = call @cpuRead32(%ctx, %addr, %value)
  %fail = icmp ne %rslt, 0
  br %fail, label %exit, label %continue

device_access:
  br label %exit

memory_access:
  %base_ptr = getelementptr %tlb, 0, %addr_index, 2     ⑥
  %base = load %base_ptr
  %aligned = lshr %addr, 2
  %offset = and %aligned, 1023
  %host_virt_addr = getelementptr %base, %offset
  %tmp = load %host_virt_addr
  store %tmp, %value
  br label %continue

exit:                                                   ⑦
  call memory_exception(%cpu)
  ret 0

continue:
  store %value, %rb0_r0
  ...
```

Figure 6.8: Example LLVM assembly for performing a memory translation using the cache-based technique. A large number of basic blocks and branching instructions, as well as memory accesses, are required on every memory access.

6.4.5.1 Naïve Scheme

This scheme involves simply overwriting each entry of the structure (be it a cache or translation function table) with a default, invalid entry. This is the simplest to implement, and block memory accesses can be performed fairly efficiently on modern processors. However, the large number of invalidations required, combined with the potentially large size of the structure being invalidated, mean that memory bandwidth can become a problem in this situation. This scheme has an extremely large invalidation cost for reasonably sized caches relative to the other schemes we will consider, so we will not consider it further.

6.4.5.2 Generational Scheme

In this scheme, a generation is attached to each entry, and a global generation counter is kept. On invalidation, the global generation counter is simply incremented, making invalidations extremely cheap. When an entry is inserted into the structure, a copy of the current global generation counter is attached. When an access is performed, the current global generation counter is compared against the local generation count. If they differ, then an invalidation must have been performed since the entry was inserted, and so the entry is treated as invalid. So, although invalidations are very cheap, this check must be performed on every single access to the structure.

Care must be taken since it is possible over a reasonably long simulation for the generation number to overflow, which might cause problems if a short data type is used to represent the generation number and some pages are infrequently accessed. For this reason, we perform a complete flush (similarly to the Naïve scheme) when the generation counter overflows. This happens extremely infrequently since a 32-bit unsigned counter is used.

6.4.5.3 Lazy Scheme

This scheme uses *host* memory protection features in order to detect accesses to invalidated or stale data. When the structure is invalidated, each *host* page of the structure is set to be inaccessible using *host* memory protection features (such as `mprotect`). When a page of the structure is later accessed, the access generates a memory protection fault (known in Linux as a 'segfault'). This fault can be trapped, and in the fault handler the page protections are reset and the page is individually invalidated. This reduces the cost of a full invalidation compared to the Naïve scheme while avoiding the per-access cost of the generational scheme. However, it does rely on efficient memory protection and fault handling in the *host* operating system, and is also more complex to implement.

Note that crucially, the *guest* page size is not relevant to this technique, as the memory being protected is e.g. the cache structures described above (Section 6.4.4), or the Memory Translation Function table described below (Section 6.4.6.

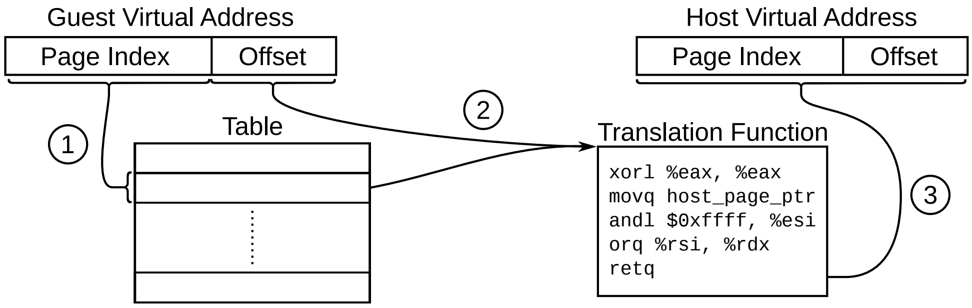
6.4.6 Memory Translation Functions

This thesis presents a novel approach for accelerating memory translations. Rather than keeping maintaining a TLB data structure, small memory translation functions for each page of virtual memory are generated. A table is kept with an entry for each page of *guest* virtual memory. Each entry in the table points to a function tailored to translating addresses for that *guest* virtual page directly into a *host* virtual address. At startup, each entry in the table points to a 'default' handler which will perform a full *guest* MMU translation, and then generate a tailored function. The entry in the table is then updated to point to this newly generated function, so that future memory accesses to the same *guest* virtual page are able to directly translate the *guest* virtual address into a *host* virtual address. If an access should succeed, it returns a *host* virtual address, otherwise it returns an error code indicating why the access failed.

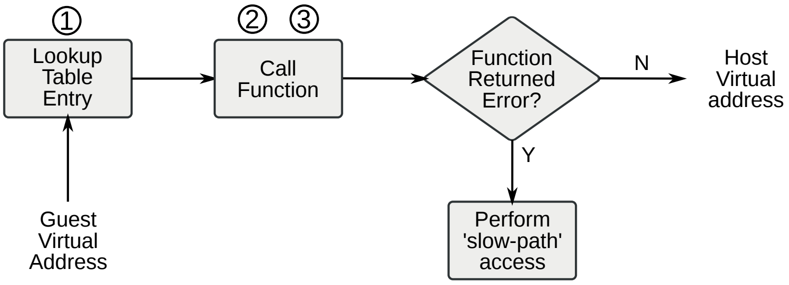
Figure 6.9 shows the process for performing a *guest* memory access when using this memory translation approach. First ①, the *guest virtual* page index is used to look up the correct memory translation function in a table. The table has one entry per possible virtual page, so no modulus operation is required. This value is guaranteed to be a function pointer: once it has been looked up, it is called directly, taking the page offset as a parameter. In the common case, the called function performs any necessary permission checks ②, and then combines the *host virtual* page base with the page offset, to form the *host virtual* address ③.

This memory translation function approach to virtual memory translations has a key advantage over a cache-based approach: the generated function can be tailored to the virtual page which is being accessed. A different function 'template' can be used for each of the outcomes of a memory translation, which include:

1. The virtual address is valid and can be translated to a page of physical memory (the common case)



(a) Data structures/data flow



(b) Control flow

Figure 6.9: An overview of the data and control flow when using Function based memory translation. Figure (a) shows a high level overview of the data structures and data flow involved, while Figure (b) shows a flow diagram of control flow for a memory access.

2. The virtual address is valid and points to a memory mapped device
3. The virtual address is valid but the running *guest* process does not have permission to access it
4. The virtual address is not valid

The first situation, where the memory translation should succeed, is the common case, and so this should be made to be the most efficient. In this case, we can generate a simple function which will mask off the bits of the address corresponding with the page base, so that we are left with the bits representing the page offset for the memory access. We then combine these bits with the pre-translated address of the base address of the correct *host virtual* page. We can then return this address and the memory access can proceed.

In other situations, we have several options. A key aspect of these memory translation functions is that they do not perform the memory access themselves. If they did, we would need to generate a large variety of functions (one for each possible memory access length, and versions for reading and writing) which would negatively impact *host* instruction cache performance. This means that for situation ②, we cannot simply perform the device access directly in the translation function.

Similarly, when differentiating between situations ① and ③, a choice can be made. For example, permissions checks could be performed directly in the translated function. This would increase the size of the translation functions and slightly slow down translations. Alternatively, these checks could be performed once when the function is generated, in which case the function table must be invalidated or switched whenever the *guest* system changes privilege level. One additional benefit of the second approach is that it is easy to port to new architectures. If we need to check permissions in the memory translation function, then the function generator needs to know the details of how permissions are implemented on the *guest* system. If we instead perform the permissions check at function generation time, then the permission check needs only to be implemented in the *guest* MMU model, rather than in both the MMU model and in the function generator. In practice, the first approach tends to be more efficient, since the majority of memory accesses are to unprivileged pages, and privilege level changes tend to happen fairly frequently (e.g. for system calls).

So, for these four different situations, we generate two types of function. The first kind, used in situation ①, actually performs a translation from a *guest* virtual address into a *host* virtual address. The second type of function, used in situations ②, ③, and ④, immediately returns an error code, indicating that the memory access should be attempted using a method which can correctly handle the presence of devices and translation faults. In this way we separate our memory accesses into an optimised 'fast-path', where the memory translations succeeds and the access can be immediately performed, and a conservative 'slow-path', where the additional complex behaviours required for accurate memory access handling can be correctly produced.

6.4.6.1 Generated Code

In this section we will briefly outline the actual generated x86 machine code used in several of the memory translation function configurations. Figure 6.10 shows the general template for the code for a successful memory translation. First, if the Generational Invalidation Scheme is in use, the current generation is checked. If the function is still valid, any permission checks are performed. If the permission check fails, then the function returns an error code. Otherwise, the correct *host virtual* address is computed

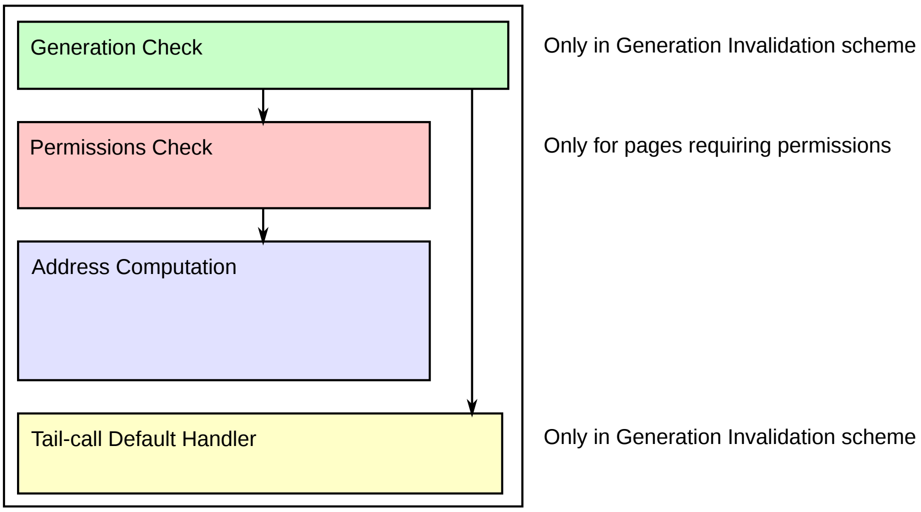

Generation Check
Permissions Check
Address Computation
Tail-call Default Handler
Only in Generation Invalidation scheme
Only for pages requiring permissions
Only in Generation Invalidation scheme

Figure 6.10: Outline of code generated for a successful memory translation. Some sections may be omitted depending on if privilege checks must be performed, or if the Generational Invalidation scheme is in use.

```
; Incoming guest virtual address in %esi, Guest CPU state struct pointer in %rdi
mem_txln_fun:
        xorl %eax, %eax                 ; Zero out the %eax register
        movq host_page_ptr(%rip), %rdx  ; Load the host virtual page pointer
        andl $0xfff, %esi               ; Mask off the page offset
        orq %rsi, %rdx                  ; Combine host page base and guest offset
        retq                            ; Return the translated address
host_page_ptr:
        .long 0x7f46a81b000
```

Figure 6.11: A simple example of a memory translation function using the Naïve or Lazy invalidation technique. No privileges are required to access the page.

and returned. If the Generational Scheme is in use and the function is found to have been invalidated, then a tail call to the default handler is performed, which will cause a new function to be generated. The colours in this diagram match those used in the assembly listings in Figures 6.11, 6.12, and 6.14,.

In Figure 6.11, we have example machine code generated for a function translating writes, using the Naïve or Lazy invalidation scheme. In this example, no special permissions are required to access the page.

In Figure 6.12, we show machine code generated for translating accesses when using the Generational invalidation scheme. Here, an additional basic block is used. If the generation in which the translation function is produced (in this case, '3') differs from the current generation (read from the CPU state data structure), then a new function is generated by tail-calling the `DefaultReadHandler` function. Although this invali-

```
; Incoming guest virtual address in %esi, Guest CPU state struct pointer in %rdi
mem_txln_fun:
        cmpl $3, 0x24(%rdi)              ; Compare the 'generation' entry of the CPU
        jne 1f                           ; struct with the generation of this function
        xorl %eax, %eax                  ; Zero out the %eax register
        movq host_page_ptr(%rip), %rdx   ; Load the host virtual page pointer
        andl $0xfff, %esi                ; Mask off the page offset
        orq %rsi, %rdx                   ; Combine host page base and guest offset
        retq                             ; Return the translated address
1:
        movq DefaultReadHandler, %rax    ; Load the address of the default handler
        jmp *%rax                          ; and perform a tail-call
host_page_ptr:
        .long 0x7f46a81b000
```

Figure 6.12: An example of a memory translation function for translating addresses for reads, using the Generational invalidation technique. Again, no special privileges are required to access the page.

```
; Incoming guest virtual address in %esi, Guest CPU state struct pointer in %rdi
mem_txln_fun:
        movl $0x1, %eax                  ; Load an error code into the %eax register
        retq                             ; Return the error code
```

Figure 6.13: When a translation fails for any reason (e.g., the virtual address is not mapped), the generated memory translation function immediately returns an error code.

```
; Incoming guest virtual address in %esi, Guest CPU state struct pointer in %rdi
mem_txln_fun:
        xorl %eax, %eax                  ; Zero out the %eax register
        cmpl $1, 0x20(%rdi)              ; Compare the 'mode' entry of the CPU struct
        adcl %eax, %eax                  ; with the value '1' (kernel mode)
        movq host_page_ptr(%rip), %rdx   ; Load the host virtual page pointer
        andl $0xfff, %esi                ; Mask off the page offset
        orq %rsi, %rdx                   ; Combine host page base and guest offset
        retq                             ; Return the translated address
host_page_ptr:
        .long 0x7f46a81b000
```

Figure 6.14: Permission checks can also be performed as part of the translation function. Here, if the CPU is not in Kernel mode, the access fails. Note that even if the permission check fails, the translated address is still returned, and no additional branching is performed in this function.

```
  %regval = load %rb0_r11
  %addr = add %105, -32
  %page_index = lshr %addr, 12
  %fn_ptr = getelementptr %txln_funcs, %page_index ①
  %fn = load %fn_ptr
  %txln = call %fn(%ctx, %addr)                    ②
  %result = extractvalue %txln, 0
  %fail = icmp ne %114, 0                          ③
  br %fail, label %exit, label %continue

continue:                                          ④
  %host_virt_addr = extractvalue %txln, 1
  %value = load %host_virt_addr
  store %value, %rb0_r0
  br label %next_instruction

exit:
  ret 0                                            ⑤
```

Figure 6.15: Example Function Based LLVM assembly for the ARM instruction `ldr r0, [r11, #-28]`

dation technique requires additional control flow compared to the others, the branch is trivially predictable not-taken.

Figure 6.13 shows the code generated when the page translation fails. This immediately returns an error code, which then signals to the simulation infrastructure that a memory exception should be triggered.

Figure 6.14 demonstrates code similar to the first example, Figure 6.11, except that in this case Kernel privileges are required to complete the memory access. The current privilege level is read from the CPU state data structure and compared against the value ‘1’ (representing the Kernel privilege level). The error code is then set using an assembly trick (using the `adc` instruction). Notice that no additional control flow is required to implement this privilege check.

Figure 6.15 shows example LLVM assembly code for actually calling a memory translation function. The correct function pointer is first looked up in the table ①, before it is called ②. The error code returned by the function is checked ③. If the error code is 0, the access succeeds and we perform a read from the pointer returned ④. Otherwise, we signal an exception ⑤.

6.5 Evaluation

In order to evaluate each of the described memory access systems, as well as each of the invalidation schemes, they have been added to GenSim. Due to the large number of combinations of results, as well as the long run time of a full run of the SPEC

DBT Parameter	**Setting**
Target architecture	ARMv5T
Host architecture	x86-64
Translation/Execution Model	Asynch. Mixed-Mode
Tracing Scheme	Region-based [18, 99]
Tracing Interval	30000 blocks
JIT compiler	LLVM 3.5
No. of JIT Compilation Threads	11
JIT Optimisation	`-O3` & Part. Eval. [107]
Initial JIT Threshold	20
Dynamic JIT Threshold	Adaptive [18]

Table 6.1: DBT System Configuration.

Name	**Translation Model**	**Invalidation Scheme**
Naïve	Naïve	Lazy
Cache	Cache-based	Lazy
Function	Function-based	Generational

Table 6.2: The 'Key' configurations selected for detailed study

benchmark suite, we present complete results for several key configurations, outlined in Table 6.2. The 'Naïve' translation model is the same as the Cache model, except that a function call is involved and the accesses cannot be optimised by LLVM. Additionally, Table 6.1 shows the GenSim configuration used during evaluation. The host machine used is described in Table 3.1.

Each SPEC benchmark has been run with its reference input (shorter input sets have been used when the reference runtime is excessively long). Where multiple data sets make up the complete reference input, these have all been executed and the total run time of all data sets is presented. For example, the 473.astar benchmark has two input sets, BigLakes2048 and Rivers. So, the run time presented for the 473.astar benchmark is the sum of the run time for the BigLakes2048 data set and the run time for the Rivers data set. We also present a Total run time, which represents the sum of the run times of all reference data sets for all tested benchmarks.

6.5.1 Key Results

Figure 6.16 presents the overall performance of each of our memory access translation configurations in terms of speedup versus the slowest. In all cases, the Cache and Function configurations outperform the Naïve configuration, typically providing at least

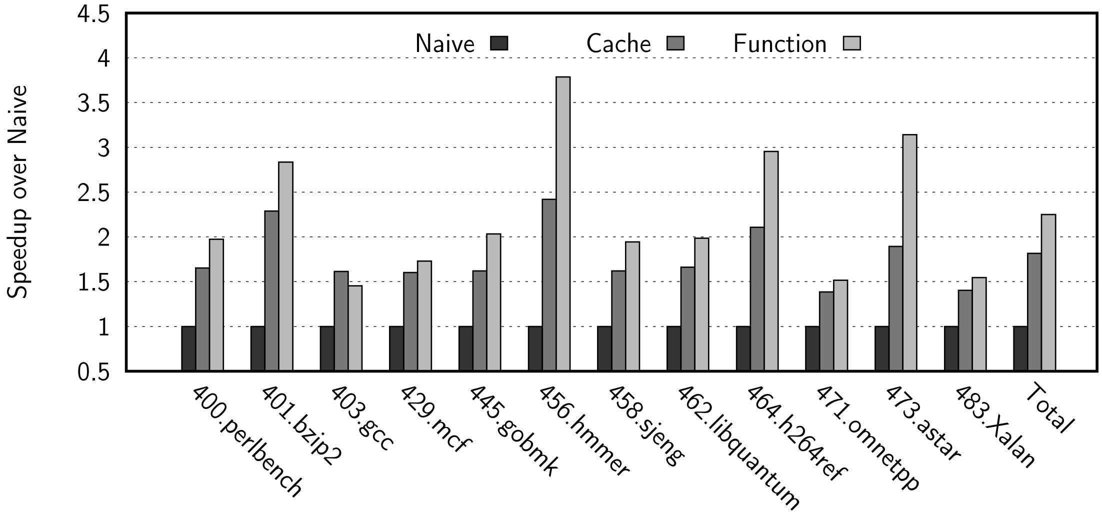


Figure 6.16: A graph showing speedups obtained using each 'Key' configuration, treating the slowest configuration as a base line. The Function configuration outperforms both other configurations on all benchmarks except for Gcc, and delivers an overall speedup of 2.25x when compared against the Naïve scheme.

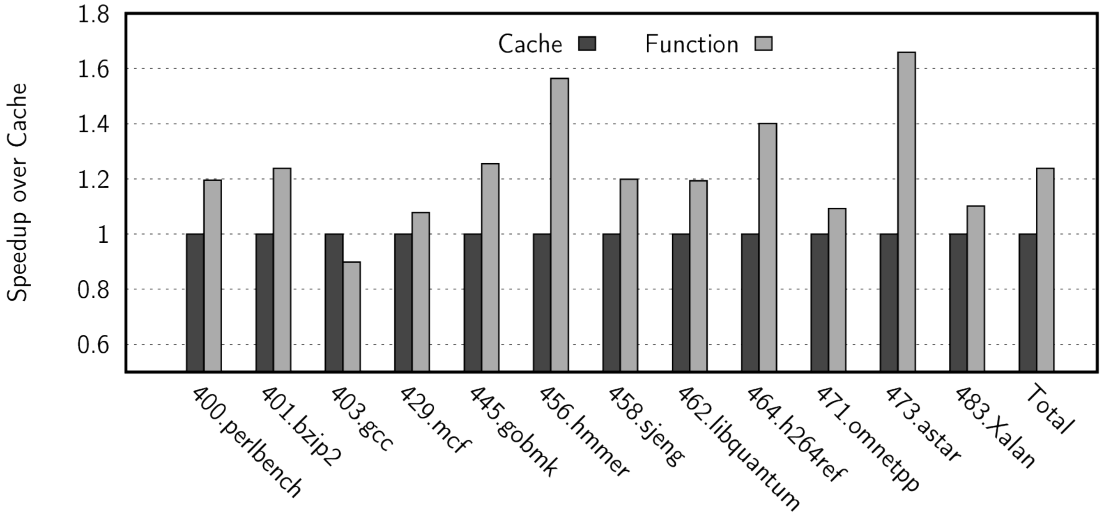


Figure 6.17: Graph comparing Function configuration against Cache configuration. The Function configuration delivers a 1.2x speedup over a full run of the benchmark suite, with a maximum speedup of 1.65x on the 473.astar benchmark.

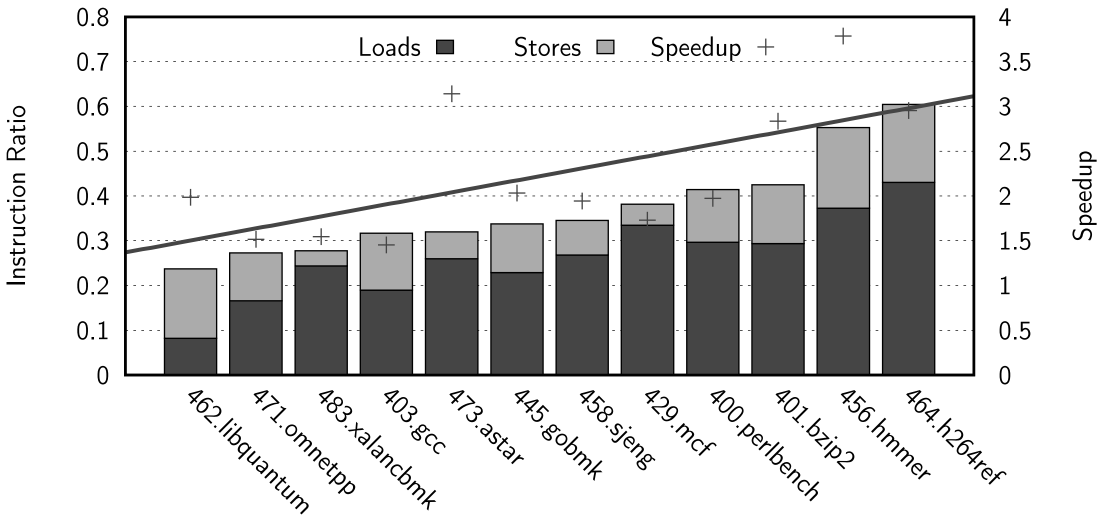


Figure 6.18: This graph plots the ratio of memory access instructions for each SPEC benchmark, and a linear regression on the speedup obtained using the Function model versus the Naïve model on that benchmark. The benchmarks are sorted by ratio of memory instructions. The line shows the speedup trend. It can be clearly seen that as the ratio of memory access instructions increases, speedup trends upwards.

a 1.5x speedup. When considering total run time, the Cache configuration provides a 1.8x speedup, and the Function configuration a 2.25x speedup, when compared to the Naïve configuration.

Furthermore, the Function configuration outperforms the Cache configuration in all cases except for on the 403.gcc benchmark (the slowdown on this benchmark is examined in Section 6.5.2). Figure 6.17 shows the speedup of the Function configuration when compared to the Cache configuration for each benchmark. Over the full benchmark suite, a speedup of 1.23x can be observed when using the Function configuration, compared to the Cache configuration.

6.5.2 Analysis

Figure 6.18 shows a graph comparing the ratio of memory access instructions (split into loads and stores) against the speedup obtained using the Function configuration compared to the Naïve configuration, presented as a linear regression. As the ratio of memory accesses increases (i.e., as more memory accesses are performed), the speedup trends upwards significantly.

Two interesting outliers are present when examining these results. Firstly, that a slowdown is observed on the 403.gcc benchmark when comparing the Function con-

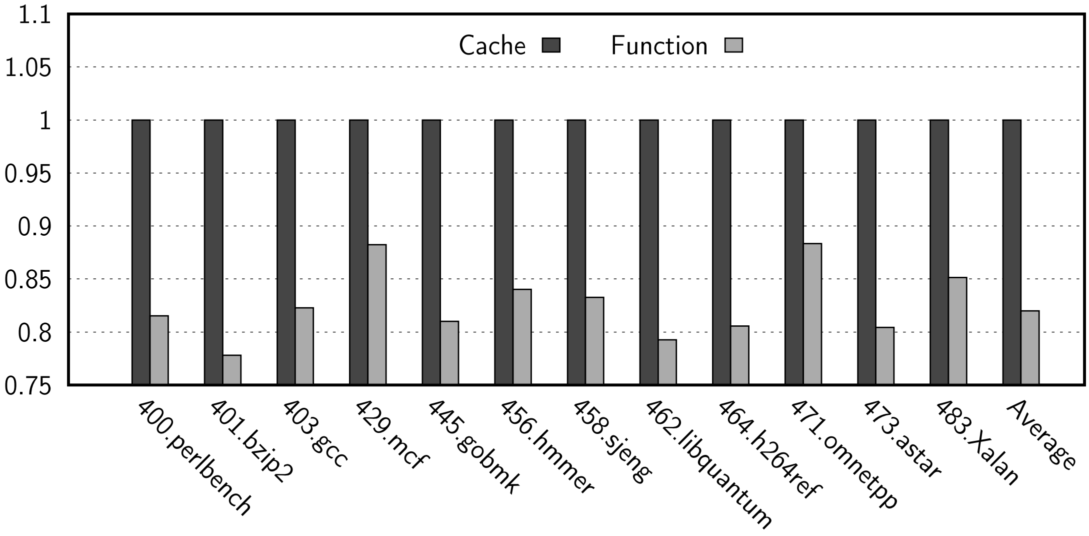


Figure 6.19: This graph compares the total size of code produced for each SPEC benchmark, when using the Cache and Function configurations, normalized against the Cache configuration. A reduction in code size is observed on every SPEC benchmark, with an overall improvement of 18%.

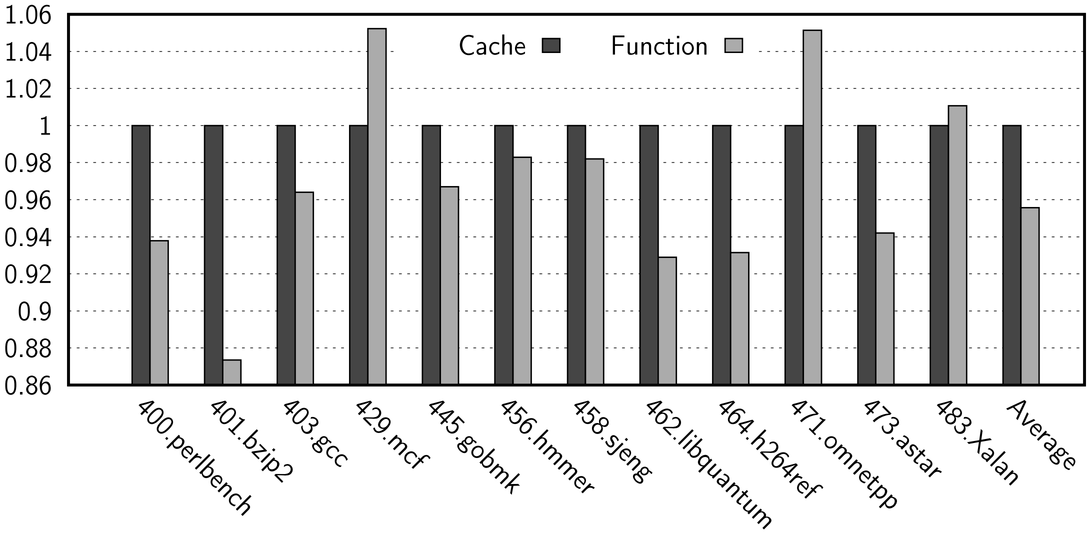


Figure 6.20: This graph compares the total sequential code generation time for each SPEC benchmark, when using the Cache and Function configurations, normalized against the Cache configuration. An improvement is observed on all but two SPEC benchmarks, with an overall improvement of 4%.

figuration against the Cache configuration, and secondly, that the 473.astar benchmark obtains such a high speedup despite having a low ratio of memory access instructions (see Figures 6.16 and 6.18).

The large speedup obtained by the 473.astar benchmark may be due to reduced compilation latency of a critical section of code. Due to the asynchronous nature of the DBT system used, reducing compilation latency can have unpredictable (but generally positive) effects on performance, as pages will be translated in a different order.

Figures 6.19 and 6.20 show the overall translated code size and the total code generation time (i.e., the time spent by LLVM optimising IR, and then translating it into native x86 binary code) for each of the SPEC benchmarks, normalised against the Cache configuration. Code size improvements are seen on all benchmarks, and code generation speed improvements are seen on all but three benchmarks when using the Function configuration.

Three benchmarks (429.mccf, 471.omnetpp and 483.Xalan) show an increased code generation time when using the Function configuration when compared against the Cache configuration. This could be due to code produced in the Function configuration being smaller, but more complex to analyse and generate. By examining Figures 6.19 and 6.20, a correlation can be observed between code generation time and total code size. On most benchmarks, code size is reduced enough to balance out the complexity of analysis. However, in these three cases, code size is not reduced by enough (due to some property of these three benchmarks) and so code generation time is increased.

Code generation speed is critical when executing highly phase-orientated workloads, since new regions of code are constantly becoming hot and requiring compilation. Over the full SPEC suite, a generated code size improvement of around 18% is observed, as well as a 4% improvement in code generation speed.

In order to more accurately assess the various costs associated with each memory access configuration, three microbenchmarks were executed using each configuration:

1. Access Cost: A microbenchmark which repeatedly access the same memory location (Figure 6.22).
2. Invalidation Cost: A microbenchmark which repeatedly accesses a memory location, and then performs a TLB flush (Figure 6.23).
3. Generation Cost: A microbenchmark which accesses one word each from a large number of memory pages (Figure 6.24).

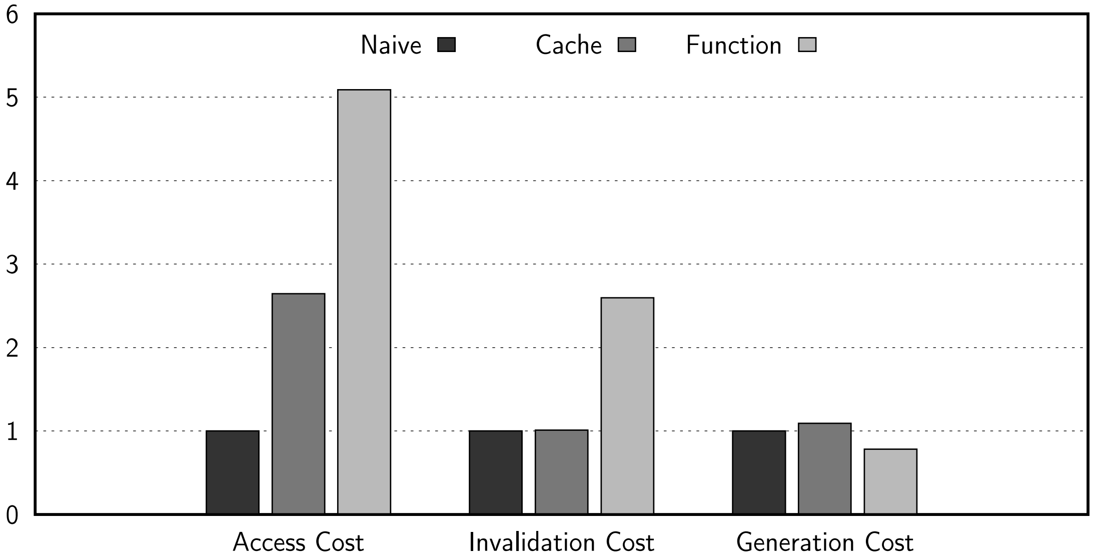


Figure 6.21: This graph shows the performance of each memory access configuration, on three microbenchmarks, in terms of speedup over the Naïve configuration.

```
void bmark() {
    // Get a pointer to a mapped memory location
    volatile char *ptr = (volatile char*)0x60000000;

    // Flush all TLB structures
    flush_tlb();
    for(int i = 0; i < 1000000; ++i) {
        *ptr;
    }
}

void main() {
    // Set up page tables, exception vectors, etc.
    setup_environment();

    // Call the benchmark function in a loop
    for(int i = 0; i < 20000; ++i) {
        bmark();
    }
}
```

Figure 6.22: Overview of the Access Cost microbenchmark. Over the run of the benchmark, approximately 20 thousand TLB flushes and 20 billion memory accesses are performed. Each memory access touches the same virtual page. The benchmark function is called a large number of times in order to amortise the cost of the environment setup.

```
void bmark() {
    // Get a pointer to a mapped memory location
    volatile char *ptr = (volatile char*)0x60000000;

    // Perform a memory access in order to ensure TLB structures are 'dirty'
    *ptr;

    // Perform a TLB flush
    flush_tlb();
}

void main() {
    // Set up page tables, exception vectors, etc.
    setup_environment();

    // Call the benchmark function in a loop
    for(int i = 0; i < 10000000; ++i) {
        bmark();
    }
}
```

Figure 6.23: Overview of the Invalidation Cost microbenchmark. Over the run of the benchmark, approximately 10 million TLB flushes and 10 million memory accesses are performed. Each memory access touches the same virtual page. The benchmark function is called a large number of times in order to amortise the cost of the environment setup.

```
void bmark() {
    // Get a pointer to a mapped memory location
    volatile char *ptr = (volatile char*)0x60000000;

    // Perform a TLB Flush
    flush_tlb();

    // Touch a large number of pages
    for(i = 0; i < 4096; ++i) {
        *p;
        p += 4096;
    }
}

void main() {
    // Set up page tables, exception vectors, etc.
    setup_environment();

    // Call the benchmark function in a loop
    for(int i = 0; i < 50000; ++i) {
        bmark();
    }
}
```

Figure 6.24: Overview of the Generation Cost microbenchmark. Over the run of the benchmark, approximately 50 thousand TLB flushes and 200 million memory accesses are performed. 4096 different pages are touched. The benchmark function is called a large number of times in order to amortise the cost of the environment setup.

These three microbenchmarks are designed to assess the costs associated with each of the major events relating to memory emulation. The Access Cost microbenchmark simply assesses the time taken to access a memory location which is already in the cache/has a generated function (i.e., it has already ‘warmed up’). The Invalidation Cost microbenchmark assesses the cost of performing a TLB flush. A memory location is first accessed on each iteration as TLB flush commands are ignored by each memory model if the TLB is not ‘dirty’. Finally, the Generation Cost microbenchmark assesses the cost of a ‘cache miss’ for each benchmark. This is performed by accessing one word from each page of a large memory region, so that the cost of inserting an entry into the cache or generating a function is paid once, but that cost is not amortized over multiple memory accesses.

Each microbenchmark was executed a large number of times with each memory access configuration, and the total run time for each was recorded. Figure 6.21 shows the speedup obtained using each memory access configuration, when compared to the Naïve configuration. The Function configuration obtains a speedup of around 5x compared to the Naïve configuration, and around 2x compared to the Cache configuration, on the memory access microbenchmark. This shows that it is indeed significantly faster when performing individual memory accesses. We can also see that the invalidation costs for the Function configuration are significantly lower than for the Naïve and Cache configurations. However, the Generation Cost is somewhat higher for the Function configuration than for the Cache or Naïve configurations, reflected in a small slowdown on the Generation Cost microbenchmark. This is likely due to the increased complexity of generating a memory translation function versus filling in a cache entry: memory must be allocated from a dedicated memory zone, and the *host* Data/Instruction cache coherence operations must occur, on top of the cost of performing the MMU lookup (which is constant across all three configurations) and emitting the function binary code itself (which is fairly cheap).

6.5.3 Analysis of Invalidation

The high function generation cost when compared with inserting entries into the cache suggests that these costs may have something to do with the slowdown observed on the `403.gcc` benchmark. Figure 6.25 shows a graph of the number of invalidations (i.e., TLB flushes) performed per memory accessing instruction. While most of the benchmarks

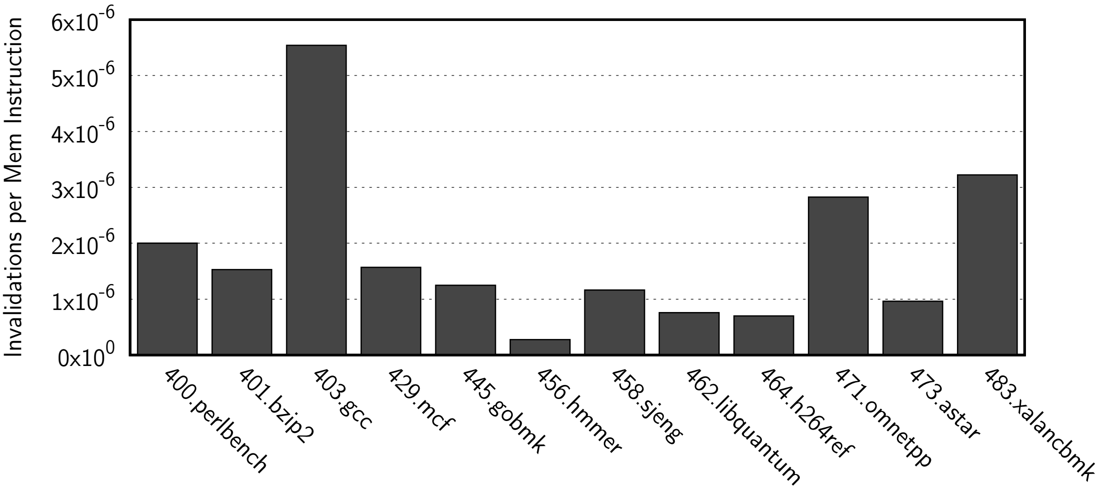


Figure 6.25: Graph showing invalidations per memory accessing instruction for each SPEC benchmark. The 403.gcc benchmark produces TLB/memory model invalidations much more frequently than any other benchmark.

perform such an invalidation much less frequently than once every 500,000 memory instructions (i.e., less than 2×10^{-6} invalidations per memory instruction), 403.gcc performs invalidations much, much more frequently (approximately once per 180,000 memory instructions). This invalidation, and the consequent re-generation of memory access functions, is likely to be what contributes to the slowdown on this benchmark when using the Function configuration. Note that these invalidations are architectural (e.g. TLB flushes) and do not depend on the memory translation model in use.

Many of the SPEC benchmarks have multiple input data sets making up their full workload. 403.gcc is one such benchmark, and the full ‘Reference’ data set is composed of 8 individual inputs. As might be guessed from the name, the 403.gcc benchmark is essentially a packaged version of the GCC compiler. The inputs are preprocessed C source files, which are then compiled into x86 assembly. Figure 6.26 shows a graph of the number of invalidations performed per memory instruction for each of the datasets for this benchmark. The s04 dataset causes a much larger number of invalidations to occur than any other dataset.

Finally, Figure 6.27 shows the speedup obtained when using the Function configuration versus the Cache configuration, against the frequency of invalidations (expressed as invalidations per memory instruction). A clear trend can be seen: the more frequently memory invalidations occur, the lower the speedup. The s04 dataset can be seen circled at the bottom right of the graph. This dataset performs invalidations much more

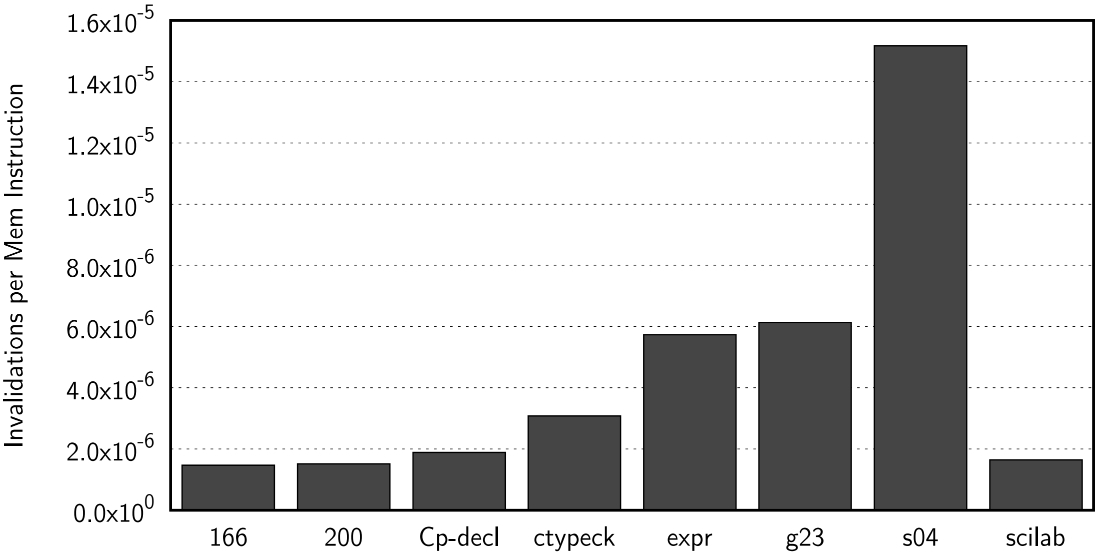


Figure 6.26: Graph showing invalidations per memory accessing instruction for each dataset of the 403.gcc benchmark. The s04 dataset produces invalidations much more frequently than any other dataset.

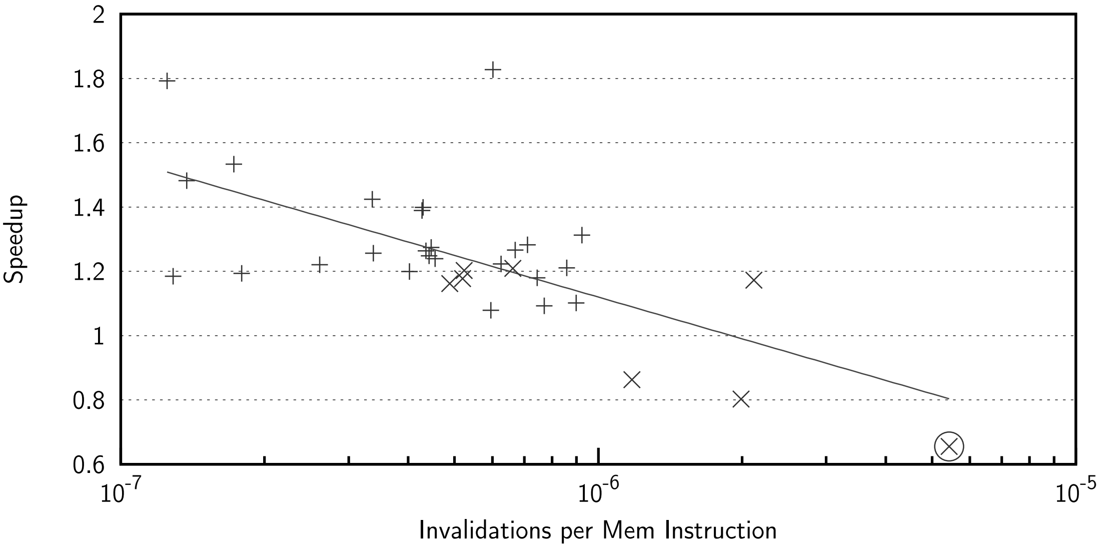


Figure 6.27: Graph showing the speedup of each dataset of each SPEC benchmark when using the Function configuration, relative to the Cache configuration, plotted against the frequency of invalidations. 403.gcc's data sets are plotted with ×s, and 403.gcc's s04 dataset is circled.

frequently than any other SPEC dataset. In addition, the only three points which fall below the 1x speedup line are data sets from the `403.gcc` benchmark.

6.6 Conclusion

This chapter has presented a novel method for performing high speed memory translations in a full system simulator. A variety of memory translation configurations were evaluated, and the novel *Memory Translation Function* based approach provided a large speedup over all measured SPEC benchmarks when compared with a naïve implementation, and a significant speedup when compared against a state of the art approach on all but one measured SPEC benchmark.

Memory translation and access speed is critical to the overall performance of an instruction set simulator, due to the large proportion of memory accessing instructions. While the state of the art approach, i.e. to use a cache similar to the TLB used in hardware, provides good performance, this chapter has shown that more can be done to accelerate memory accesses, mainly by improving the technique used to translate a *guest* virtual address into a *host* virtual address, and to develop new techniques to invalidate any cache structures used. This chapter has also investigated the costs of invalidating these structures, and shown that although the novel Memory Translation Function approach provides improved memory access time, function generation is somewhat more expensive than refilling a cache entry. Future work might focus on reducing function generation costs in order to provide improved all-round performance.

The presented techniques contribute significantly to our *performance* objective, by providing a significant speedup when compared against state of the art techniques, and also our *completeness* objective, by providing a flexible method for implementing memory translations in simulation.

Chapter 7

Conclusion

7.1 Introduction

This thesis has attempted to address three key objectives for Instruction Set Simulators: *performance*, *correctness*, and *completeness*. This chapter will first assess each of the contributions presented in this thesis in terms of these objectives (Section 7.2), as well as being critically analysed, particularly in terms of trade-offs made in order to make the evaluation of each contribution feasible, in Section 7.3. Finally, in Section 7.4, future work and extensions to each contribution will be suggested, motivated again by each of the original three objectives.

How this chapter is structured

- This chapter begins with a summary of the main contributions of this thesis
- Secondly, these contributions are critically analysed
- Finally, possible future work to extend these contributions is discussed

7.2 Contributions

This thesis has attempted to demonstrate solutions to a variety of relevant and difficult problems in the field of high speed instruction set simulation. Three key objectives were originally proposed: *performance*, *completeness*, and *correctness*, and techniques have been presented which address each of these objectives. First, in Chapter 4, a *Partial Evaluation*-based analysis was performed in order to enable the generation of a high-speed DBT system from abstract ADL descriptions, addressing the *performance* objective. Then, in Chapter 5, the *correctness* objective was tackled: ADL descriptions were used to evaluate a benchmark driven ad-hoc testing scheme, showing it to be insufficient, and then to generate a much more complete test suite based on constraint solving techniques. Finally, in Chapter 6, challenges relating to full-system simulation were addressed, and improvements to both interrupt handling and efficient memory address translation were presented, contributing to both the *performance* and *completeness* objectives.

7.2.1 Efficient Simulator Generation

The main contribution presented in Chapter 4 is a technique to produce a high speed DBT module from a high level ADL description, integrated into our GenC ADL. While several fully or partially generated simulation frameworks have generated DBT modules in the literature, GenC is the first to perform a partial-evaluation driven analysis in order to improve DBT performance, both in terms of compilation time, and runtime performance, and obtains a 2.5x speedup over a naïve approach, and a similar speedup over a state of the art simulator when domain specific optimisations are applied.

Although initially aimed only at improving simulation performance, the ADL parsing and analysis techniques developed for this contribution were then reused for the implementation of the test generation techniques described in Chapter 5. In Section 7.4, further uses for the information provided by this analysis are also suggested.

7.2.2 Automated Test Generation

Chapter 5 began by evaluating the effectiveness of benchmark-driven testing. Although not designed with simulator testing or ISA coverage in mind, many modern benchmark suites find themselves used as simulator test suites, or as measures of the correctness or completeness of a simulator. However, Section 5.3.3 shows that these benchmark suites provide very poor instruction set coverage. Rather than relying on these benchmark suites to show correctness, a more thorough testing methodology is required.

Section 5.4 presents such a methodology. By identifying possible paths through each instruction semantic implementation, and identifying the constraints which these paths place on the instruction and context in which the instruction is executed, constraint solving techniques can be used to generate a test suite which provides almost complete path coverage for most instructions. This contributes to our *correctness* objective, since we are now able to show that our architecture description is correct with much more confidence.

7.2.3 Efficient Full-System Simulation

Although interrupt handling is a major part of any full system simulation model, one of the most major components in terms of performance is the method by which *guest virtual* addresses are translated first into *guest physical*, and then into *host virtual* addresses. It was shown in Section 6.4 that memory instructions constitute between 30% and 55% of the dynamic instruction mix when executing the SPEC benchmark suite. In Section 6.4.6, a novel technique for accelerating these memory translations, namely to generate fast *memory translation functions*, gave a significant speedup of up to 1.65x over state of the art techniques, with a 1.2x speedup observed over a full run of the SPEC benchmark suite. Although a slowdown was observed on one benchmark, the cause of this slowdown has been investigated and some suggestions have been made on how to further improve performance.

7.3 Critical Analysis

7.3.1 Efficient Simulator Generation

Although models for a number of architectures exist for the original ArchC language (upon which GenC is based), so far the only complete GenC model is for the ARM architecture. This is partly due to a lack of development time, but also partly due to a lack of flexibility in the GenC language. The original ArchC language extends SystemC directly, meaning that many features can be described which would not be possible in the current version of GenC. However, this flexibility is traded off against performance, as the current ArchC implementation does not support DBT-based simulation. In any case, GenC must be made much more flexible before many new architectures are describable.

7.3.2 Automated Test Generation

The constraint generation and satisfaction model presented in Chapter 5 is capable of handling most instructions which contain non looping control flow. However, it is not capable of handling instructions which do contain looping control flow, such as the `ldm` and `stm` instructions in the ARM architecture. These instructions therefore currently require special handling, i.e. by hand-writing tests rather than generating them automatically. Test generation systems for general purpose programming languages e.g. CUTE [95] do support the generation of test for programs which contain looping control flow, so this is clearly not a limitation of the concept of path-based testing, rather it is a limitation of the presented implementation.

The presented constraint satisfaction and generation system also does not support the generation of tests for system instructions, such as ARM's `mcr`/`mrc` instructions which handle coprocessor communication. These instructions have effects beyond the architectural state of the CPU, which makes them challenging to test, as their effects must be described in a generic manner (for example, in our full-system ARM model the `mcr` and `mrc` instructions use special intrinsic functions to perform communication with simulated coprocessors), and must be comparable between implementations, i.e. it must be possible to compare the behaviour of the simulator against a reference model.

Finally, the testing framework presented has a number of limitations: it does not support the testing of the 'full-system' effects of instructions (such as memory instructions causing memory protection or page faults), and since it runs as a user-mode program, it requires that a significant number of instructions be known to work. A more flexible testing framework might use a debug interface to perform testing, meaning that instructions can be run totally in isolation, meaning that exceptions, updates to system registers, etc. can be observed.

7.3.3 Efficient Full-System Simulation

In Chapter 6, a novel technique for accelerating memory accesses under full system simulation was presented. This technique gives improved performance when compared with the state of the art, and is highly flexible. However, generating such a memory access function is somewhat more time consuming than inserting an entry into a software cache. In most instances, the improved memory access speed balanced this, but some improvements should be made to the function generation cost.

While the contributions presented in Chapters 4 and 5 rely on the presence of an architecture description, and an ADL-based simulator generation framework, the use of memory translation functions can be easily applied to other functional simulators such as QEMU. Some work would need to be performed to generate these functions in a portable manner, since QEMU supports a variety of *host* and *guest* platforms.

7.4 Future Work

Although some analysis of instruction semantics described using the GenC ADL is performed, the analysis is restricted to identifying opportunities for optimisation in the DBT module at quite a low level. For example, although the partial evaluation framework is capable of performing limited constant folding and dead code elimination effectively 'in advance', it cannot currently identify properties of instructions useful for higher level optimisations. There are several instances in current descriptions where this information could be extracted from instruction semantics, but must instead be provided separately by the user. For example, if an instruction can read the PC, it must be marked as such. Control flow instructions must also be marked up, including information about the type of jump (i.e., if it is a direct or indirect branch), in order to support the region forming algorithm used by the DBT system. This places a significant burden on the user, as this information must be correct in order to obtain correct simulation behaviour, and incorrect information can result in subtle and confusing simulation bugs.

While the base GenC ADL has proven flexible enough to model the base ARM ISA in full, as well as supporting models for MIPS, Power, the Texas Instruments C6x DSP series, as well as several microcontrollers, certain features in these architectures (such as many of the DSP features, and branch delay slots in MIPS) have required direct modification of the simulator framework. The main issue is that the GenC ADL describes only behaviours at the level of an individual instruction, and so behaviours which involve multiple instructions cannot be efficiently handled. A possible extension of the GenC ADL would be to support a high level system description, effectively involving a description of the fetch-execute cycle of the described architecture. A description at this level would be able to handle many of the advanced features of modern architectures, such as delay slot branches, VLIW instruction bundling, multiple ISA modes, instruction prefixes in x86 architectures, etc. Careful analysis of the description could potentially allow for high levels of flexibility, allowing not just multiple CPU architectures, but also potentially allowing for the description of DSP and GPU architectures, while maintaining high performance.

GenTest, the test methodology outlined in Chapter 5 has been proven to be useful, and has discovered several bugs in several popular simulators. However, a critical limitation with our evaluation of GenTest is that the methodology was tested only in a user-mode simulation context. Full system simulation presents many additional complex behaviours which must be properly handled for correct simulation, such as multiple types of memory fault, exception return behaviours, register banking, mode and privilege switching, etc. Although generating such tests is likely to be straightforward (as test generation is performed strictly in terms of intra-instruction control flow, which is in no way different or ‘special’ in a full-system context), actually executing the tests and collecting their results is much more complex. It may also be desirable to place limits on the scope of the tests. For example, in the ARM architecture, a memory fault has both architectural effects (including a privilege change, a mode and register bank switch, and potentially an instruction set switch if working in Thumb mode) and system-level effects, causing changes to coprocessor registers and other state outside of the CPU itself. In such cases, it is difficult to assess what should be included directly in the architectural description (and thus exposed to the test generation infrastructure) and what should be compared when executing the tests against a reference model.

A further limitation which has been placed on our test generation methodology is that we have mainly applied it to simple integer and bitwise arithmetic instructions. However, the most complex part of modern architectures tends to be in floating point, vector, cryptographic, and other media instructions. Floating point instructions present a particular problem as relying on the *host* FPU to perform *guest* floating point operations may not produce bit-accurate results due to differences in the underlying floating point model. On the other hand, directly implementing a bit-accurate floating point model can be extremely complex. For this reason, evaluating the test generation methodology outlined in Chapter 5 in the context of more complex instructions would be a true test of the effectiveness of the methodology.

Finally, fast full system simulation poses many challenges in addition to those of user mode only simulation. The efficient implementation of the guest virtual memory system certainly has one of the largest impacts on simulation performance in a full system context. The memory translation function technique presented in Chapter 6 goes some way to addressing this problem, but full system simulation still experiences a significant slowdown when compared to user mode simulation. Most modern architectures now include extensions intended to allow same-ISA virtualisation. In particular, AMD's AMD-V and Intel's VT-x extensions enable efficient x86-on-x86 virtualisation with very low overheads. It is possible that these extensions could be used instead to allow simulation of an arbitrary architecture on top of an x86 machine, provided that some basic similarities exist between the system models. In particular, the page table caching structures described in Chapter 6 and accelerated by the use of memory translation functions could instead be implemented directly using a virtualised page table and performed by the host system's MMU, essentially reducing the overhead of a guest memory access to zero.

7.5 Final Remarks

This thesis has presented several techniques for improving the performance, correctness, and completeness of simulators generated from high level descriptions. It has been shown that such generated simulators can obtain performance exceeding that of hand-written simulators by using novel DBT techniques, without requiring significant additional user effort. Such descriptions have also been shown to be highly testable in an automated fashion, greatly aiding the model debugging process. Techniques for improving full system simulation have also been covered. In particular, this thesis has presented novel techniques for accelerating memory accesses in a system including a virtual memory model.

However, these techniques barely touch the surface of what is possible - and necessary - when constructing a high speed simulator. As the performance gap closes between high end 'embedded' systems such as modern smartphones, and the full-power workstation machines used to simulate these systems, the shortfall in simulation performance has increased. While the simulation techniques discussed in this thesis might aid in improving the performance of functional simulation, these techniques address just some of the problems in one small section of the large field of system simulation. New simulation techniques will need to be developed across the field if the simulation of interesting workloads on interesting systems is to remain feasible.

Appendix A

Detailed Results - Efficient Simulator Generation

Benchmark	Dataset	Naïve	Dynamic
400.perlbench	diffmail.pl	5316.00	2344.29
400.perlbench	splitmail.pl	5355.00	1987.03
401.bzip2	input.source	3278.50	724.98
401.bzip2	chicken.jpg	1326.24	234.02
401.bzip2	liberty.jpg	2139.45	339.39
401.bzip2	input.program	4260.00	793.20
401.bzip2	text.html	4626.00	900.50
401.bzip2	input.combined	2689.16	569.44
403.gcc	166	1059.44	529.85
403.gcc	200	1954.93	912.13
403.gcc	ctypeck	1861.22	960.57
403.gcc	cp-decl	1249.32	635.47
403.gcc	expr	1358.63	701.47
403.gcc	g23	2068.19	1038.71
403.gcc	s04	1894.40	954.89
403.gcc	scilab	784.72	374.32
429.mcf	(reference)	2462.53	633.14
445.gobmk	13x13	3703.00	1191.02
445.gobmk	nngs	9638.00	2949.48
445.gobmk	score2	4297.00	1310.95
445.gobmk	trevorc	3727.00	1191.35
445.gobmk	trevord	5134.00	1554.18
456.hmmer	nph3	8973.00	1330.27
456.hmmer	retro	24038.00	4285.00
458.sjeng	(train)	6289.00	1795.55
462.libquantum	(reference)	20872.00	6591.00
464.h264ref	foreman baseline	5070.00	1490.81
464.h264ref	foremain main	3397.91	1177.59
464.h264ref	sss main	30262.00	9969.00
471.omnetpp	(reference)	17914.00	8684.00
473.astar	BigLakes2048	2705.22	770.89
473.astar	rivers	5455.00	1365.97
483.xalancbmk	(reference)	15784.00	7445.00

Table A.1: Time taken, in seconds, to execute SPEC benchmarks in user mode simulation with Naïve and Partial-Evaluation based DBT frontends when using -O3 JIT-time optimisation.

Benchmark	Dataset	Speedup
400.perlbench	diffmail.pl	2.267
400.perlbench	splitmail.pl	2.694
401.bzip2	chicken.jpg	5.667
401.bzip2	input.combined	4.722
401.bzip2	input.program	5.370
401.bzip2	input.source	4.522
401.bzip2	liberty.jpg	6.303
401.bzip2	text.html	5.137
403.gcc	166	1.999
403.gcc	200	2.143
403.gcc	cp-decl	1.965
403.gcc	ctypeck	1.937
403.gcc	expr	1.936
403.gcc	g23	1.991
403.gcc	s04	1.983
403.gcc	scilab	2.096
429.mcf	(reference)	3.889
445.gobmk	13x13	3.109
445.gobmk	nngs	3.267
445.gobmk	score2	3.277
445.gobmk	trevorc	3.128
445.gobmk	trevord	3.303
456.hmmer	nph3	6.745
456.hmmer	retro	5.609
458.sjeng	train	3.502
462.libquantum	(reference)	3.166
464.h264ref	foreman baseline	3.400
464.h264ref	foreman main	2.885
464.h264ref	sss main	3.035
471.omnetpp	(reference)	2.062
473.astar	BigLakes2048	3.509
473.astar	rivers	3.993
483.xalancbmk	(reference)	2.120

Table A.2: Speedup when executing SPEC benchmarks in user mode simulation with Partial-Evaluation based DBT frontend compared to Naïve when using -O3 JIT-time optimisation.

Benchmark	Iterations	Naïve-O1	Naïve-O3	Dynamic-O1	Dynamic-O3
a2time01	12000000	664.06	185.41	131.44	53.75
aifftr01	120000	1453.06	481.95	304.43	60.23
aifirf01	20000000	841.29	320.12	164.15	40.72
aiifft01	120000	1391.02	456.99	294.62	55.94
basefp01	4000000	1477.94	443.00	341.68	130.88
bezier01	80000	1050.09	327.40	237.88	65.78
bitmnp01	600000	835.09	238.99	170.84	62.94
cacheb01	200000000	560.90	176.19	159.80	64.74
canrdr01	300000000	654.27	182.91	176.45	66.53
cjpeg	2000	835.55	247.59	174.65	48.78
conven00	1200000	2148.89	549.17	393.33	61.75
dither01	30000	1422.42	371.91	276.49	65.88
djpeg	3000	982.04	296.87	210.15	49.97
fft00	2000000	2531.35	667.81	441.91	55.33
idctrn01	1600000	844.04	278.59	203.77	62.67
iirflt01	12000000	940.89	344.82	203.85	58.00
matrix01	10000	634.51	180.13	153.62	64.65
ospf	1000000	1081.32	284.80	273.29	71.99
pktflow	400000	483.76	150.56	140.81	64.63
pntrch01	2000000	1125.57	322.92	249.22	52.98
puwmod01	300000000	941.21	241.74	215.07	53.80
rgbcmy01	30000	1445.57	443.18	306.22	69.10
rgbhpg01	40000	1553.16	467.07	341.15	63.72
rgbyiq01	25000	2535.08	825.10	508.25	61.10
rotate01	80000	909.08	237.57	194.60	54.35
routelookup	150000	631.75	196.55	172.63	69.88
rspeed01	200000000	773.04	202.67	166.11	52.76
tblook01	4000000	463.99	155.11	126.71	61.93
text01	30000	567.31	161.42	158.21	81.88
ttsprk01	8000000	911.76	272.96	209.49	72.53
viterb00	250000	1523.56	488.77	284.83	61.09

Table A.3: Time taken, in seconds, to execute EEMBC benchmarks in user mode simulation with Naïve and Partial-Evaluation based DBT frontends.

Benchmark	Iterations	Naïve-O1	Naïve-O3	Dynamic-O1	Dynamic-O3
a2time01	12000000	1.000	3.582	5.052	12.355
aifftr01	120000	1.000	3.015	4.773	24.125
aifirf01	20000000	1.000	2.628	5.125	20.660
aiifft01	120000	1.000	3.044	4.721	24.866
basefp01	4000000	1.000	3.336	4.326	11.292
bezier01	80000	1.000	3.207	4.414	15.964
bitmnp01	600000	1.000	3.494	4.888	13.268
cacheb01	200000000	1.000	3.183	3.510	8.664
canrdr01	300000000	1.000	3.577	3.708	9.834
cjpeg	2000	1.000	3.375	4.784	17.129
conven00	1200000	1.000	3.913	5.463	34.800
dither01	30000	1.000	3.825	5.145	21.591
djpeg	3000	1.000	3.308	4.673	19.653
fft00	2000000	1.000	3.791	5.728	45.750
idctrn01	1600000	1.000	3.030	4.142	13.468
iirflt01	12000000	1.000	2.729	4.616	16.222
matrix01	10000	1.000	3.523	4.130	9.815
ospf	1000000	1.000	3.797	3.957	15.020
pktflow	400000	1.000	3.213	3.436	7.485
pntrch01	2000000	1.000	3.486	4.516	21.245
puwmod01	300000000	1.000	3.893	4.376	17.495
rgbcmy01	30000	1.000	3.262	4.721	20.920
rgbhpg01	40000	1.000	3.325	4.553	24.375
rgbyiq01	25000	1.000	3.072	4.988	41.491
rotate01	80000	1.000	3.827	4.672	16.726
routelookup	150000	1.000	3.214	3.660	9.040
rspeed01	200000000	1.000	3.814	4.654	14.652
tblook01	4000000	1.000	2.991	3.662	7.492
text01	30000	1.000	3.514	3.586	6.929
ttsprk01	8000000	1.000	3.340	4.352	12.571
viterb00	250000	1.000	3.117	5.349	24.940

Table A.4: Speedup when executing EEMBC benchmarks in user mode simulation with Naïve and Partial-Evaluation based DBT frontends.

Appendix B

Detailed Results - Automated Test Generation

	Tested Paths	**Cvc4 Rejected Paths**	**Avg. Path Length**	**Avg. Constraints**
adc	46	4	8.52	5.12
add	86	16	10.00	5.96
and	46	4	8.02	5.12
b	1	0	1.00	3.00
bic	46	4	8.02	5.12
bl	1	0	1.00	3.00
blx	1	1	3.00	3.00
bx	1	1	3.00	3.00
cmn	25	2	6.48	3.11
cmp	23	2	6.52	3.12
eor	46	4	8.02	5.12
ldr	15	29	6.73	7.50
ldrb	14	4	5.78	6.50
ldrbt	7	13	6.42	6.50
ldrd	4	0	4.50	8.50
ldrh	4	0	4.50	5.50
ldrsb	4	0	4.50	5.50
ldrsh	4	0	4.50	5.50
ldrt	7	17	6.57	6.66
mla	2	0	2.50	2.00
mov	58	8	8.29	5.39
mul	2	0	2.50	2.00
mvn	46	4	8.02	5.12
orr	46	4	8.02	5.12
rsb	46	4	8.52	5.12
rsc	46	4	8.47	5.12
sbc	46	4	8.52	5.12
smlal	2	0	2.50	3.00
smull	2	0	2.50	3.00
str	14	4	5.78	5.50
strb	14	4	5.78	5.50
strbt	7	11	6.28	5.50
strd	4	0	4.50	6.50
strh	4	0	4.50	4.50
strt	7	11	6.28	5.50
sub	50	4	8.56	5.11
swp	1	0	1.00	3.00
swpb	1	0	1.00	3.00
teq	23	2	6.52	3.12
tst	23	2	6.52	3.12
umlal	2	0	2.50	3.00
umull	2	0	2.50	3.00

Table B.1: Statistics on tests generated using GenTest. Some instructions such as system instructions, `ldm`, and `stm` must have tests written by hand and are not included in this table. A list of these instructions can be found in Table 5.1.

Appendix C

Detailed Results - Efficient Full-System Simulation

Benchmark	Dataset	Naïve	Cache	Function
400.Perlbench	diffmail.pl	9645.63	6170.87	5231.39
400.Perlbench	splitmail.pl	10980.46	6314.80	5215.63
401.bzip2	input.source	5361.91	2475.48	2028.09
401.bzip2	chicken.jpg	1718.92	705.59	550.03
401.bzip2	liberty.jpg	2539.01	1040.98	822.16
401.bzip2	input.program	6366.48	2810.21	2140.96
401.bzip2	text.html	7347.59	3137.15	2648.14
401.bzip2	input.combined	4124.51	1832.02	1497.43
403.gcc	166.i	1523.57	896.18	744.94
403.gcc	200.i	2871.38	1733.32	1491.85
403.gcc	c-typeck.i	2742.23	1507.81	1246.82
403.gcc	cp-decl.i	2261.77	1295.53	1501.23
403.gcc	expr.i	3017.74	1968.82	2453.28
403.gcc	g23.i	3495.06	2036.93	1736.86
403.gcc	s04.i	5259.30	3658.21	5580.91
403.gcc	scilab.i	1131.55	705.22	599.04
429.mcf	(test)	53.92	33.62	31.16
445.gobmk	13x13.tst	5100.22	3160.01	2478.95
445.gobmk	nngs.tst	13121.46	8089.76	6481.77
445.gobmk	score2.tst	5570.73	3371.97	2683.27
445.gobmk	trevorc.tst	5055.43	3149.31	2541.34
445.gobmk	trevord.tst	7050.78	4382.06	3466.45
456.hmmer	nph3.hmm	12566.35	5325.39	2970.98
456.hmmer	retro.hmm	29927.77	12228.95	8253.61
458.sjeng	ref.txt	41435.97	25584.33	21327.69
462.libquantum	(reference)	20607.46	12388.54	10378.41
464.h264ref	foreman baseline	12109.86	6110.50	4290.02
464.h264ref	foreman main	10104.07	4364.67	3141.46
464.h264ref	sss main	82673.48	39266.49	28075.68
471.omnetpp	(reference)	29280.79	21110.07	19314.54
473.astar	BigLakes2048.cfg	5557.92	3423.73	1873.51
473.astar	rivers.cfg	8237.01	3860.14	2517.54
483.xalancbmk	(reference)	20296.02	14468.40	13129.11

Table C.1: Time taken, in seconds, for each SPEC dataset in each of the tested full-system configurations.

Benchmark	Dataset	Naïve	Cache	Function
400.Perlbench	diffmail.pl	1.00	1.56	1.84
400.Perlbench	splitmail.pl	1.00	1.74	2.11
401.bzip2	input.source	1.00	2.17	2.64
401.bzip2	chicken.jpg	1.00	2.44	3.13
401.bzip2	liberty.jpg	1.00	2.44	3.09
401.bzip2	input.program	1.00	2.27	2.97
401.bzip2	text.html	1.00	2.34	2.77
401.bzip2	input.combined	1.00	2.25	2.75
403.gcc	166.i	1.00	1.70	2.05
403.gcc	200.i	1.00	1.66	1.92
403.gcc	c-typeck.i	1.00	1.82	2.20
403.gcc	cp-decl.i	1.00	1.75	1.51
403.gcc	expr.i	1.00	1.53	1.23
403.gcc	g23.i	1.00	1.72	2.01
403.gcc	s04.i	1.00	1.44	0.94
403.gcc	scilab.i	1.00	1.60	1.89
429.mcf	(test)	1.00	1.60	1.73
445.gobmk	13x13.tst	1.00	1.61	2.06
445.gobmk	nngs.tst	1.00	1.62	2.02
445.gobmk	score2.tst	1.00	1.65	2.08
445.gobmk	trevorc.tst	1.00	1.61	1.99
445.gobmk	trevord.tst	1.00	1.61	2.03
456.hmmer	nph3.hmm	1.00	2.36	4.23
456.hmmer	retro.hmm	1.00	2.45	3.63
458.sjeng	ref.txt	1.00	1.62	1.94
462.libquantum	(reference)	1.00	1.66	1.99
464.h264ref	foreman baseline	1.00	1.98	2.82
464.h264ref	foreman main	1.00	2.31	3.22
464.h264ref	sss main	1.00	2.11	2.94
471.omnetpp	(reference)	1.00	1.39	1.52
473.astar	BigLakes2048.cfg	1.00	1.62	2.97
473.astar	rivers.cfg	1.00	2.13	3.27
483.xalancbmk	(reference)	1.00	1.40	1.55

Table C.2: Speedup, compared to Naïve for each SPEC dataset in each of the tested full-system configurations.

Benchmark	Read	Write
`400.perlbench`	`0.297`	`0.117`
`401.bzip2`	`0.294`	`0.131`
`403.gcc`	`0.189`	`0.127`
`429.mcf`	`0.334`	`0.046`
`445.gobmk`	`0.229`	`0.109`
`456.hmmer`	`0.373`	`0.179`
`458.sjeng`	`0.268`	`0.078`
`462.libquantum`	`0.082`	`0.155`
`464.h264ref`	`0.430`	`0.174`
`471.omnetpp`	`0.165`	`0.107`
`473.astar`	`0.259`	`0.059`
`483.xalancbmk`	`0.244`	`0.034`

Table C.3: Proportion of dynamic memory instructions in each SPEC benchmark.

Benchmark	Naïve	Cache	Function
`Access Cost`	`222.21`	`83.99`	`43.65`
`Invalidation Cost`	`121.13`	`119.82`	`46.65`
`Generation Cost`	`136.14`	`124.43`	`173.68`

Table C.4: Time taken, in seconds, for each microbenchmark in each of the tested full-system configurations.

Benchmark	Naïve	Cache	Function
`Access Cost`	`1.00`	`2.64`	`5.09`
`Invalidation Cost`	`1.00`	`1.01`	`2.59`
`Generation Cost`	`1.00`	`1.09`	`0.78`

Table C.5: Speedup for each microbenchmark in each of the tested full-system configurations, compared to Naïve.

Bibliography

[1] H. Peter Anvin. “Method and system for providing hardware support for memory protection and virtual memory address translation for a virtual machine”. Pat. US 7111146 B1. 2006.

[2] *Apple Rosetta (via Web Archive)*. `https://web.archive.org/web/20110107211041/http://www.apple.com/rosetta`. 2011. (Visited on 03/26/2015).

[3] Eduardo Argollo et al. “COTSon: Infrastructure for Full System Simulation”. In: *ACM SIGOPS Operating Systems Review* 43.1 (Jan. 2009), pp. 52–61. ISSN: 0163-5980. DOI: `10.1145/1496909.1496921`.

[4] ARM Limited. *ARMv5 Architecture Reference Manual, DDI0100I*. ARM Limited, 2007.

[5] Rodolfo Azevedo et al. “The ArchC architecture description language and tools”. In: *International Journal of Parallel Programming* 33.5 (Oct. 2005), pp. 453–484. ISSN: 08857458. DOI: `10.1007/s10766-005-7301-0`.

[6] Julian Bangert et al. “The page-fault weird machine: lessons in instruction-less computation”. In: *Presented as part of the 7th ldots*. `https://www.usenix.org/system/files/tech-schedule/woot13-papers-archive.zip`. USENIX Association, 2013.

[7] John Banning et al. “Fine grain translation discrimination”. Pat. US 6363336 B1. 2002.

[8] Sorav Bansal and Alex Aiken. “Binary Translation Using Peephole Superoptimizers”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 177–192. URL: `http://dl.acm.org/citation.cfm?id=1855741.1855754`.

[9] Leonid Baraz et al. “IA-32 Execution Layer: a Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems”. In: *Proceedings of the International Symposium on Microarchitecture*. IEEE, 2003, pp. 191–201. ISBN: 0-7695-2043-X. DOI: `10.1109/MICRO.2003.1253195`.

[10] Paul Barham et al. “Xen and the Art of Virtualization”. In: *Proceedings of the Symposium on Operating Systems Principles*. ACM, 2003, pp. 164–177.

[11] Clark Barrett et al. “CVC4”. In: *Proceedings of the International Conference on Computer Aided Verification*. CAV’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. ISBN: 978-3-642-22109-5.

[12] Markus Becker et al. “XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software”. In: *Proceedings of the International Conference on Embedded Software*. ACM, 2012, pp. 33–42. ISBN: 978-1-4503-1425-1. DOI: `10.1145/2380356.2380368`.

[13] Richard Belgard. “Speculative address translation for processor using segmentation and optical paging”. Pat. US 6430668 B2. 2002.

[14] Fabrice Bellard. “QEMU , a Fast and Portable Dynamic Translator”. In: *USENIX Annual Technical Conference. Proceedings of the 2005 Conference on*. `https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/bellard.html`. USENIX Association, 2005, pp. 41–46.

[15] Nathan Binkert et al. “The gem5 simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011), p. 1. ISSN: 01635964. DOI: `10.1145/2024716.2024718`.

[16] F Blanqui et al. “Designing a CPU model : from a pseudo-formal document to fast code”. In: *Proceedings of the Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2011, pp. 2–7.

[17] Igor Böhm, Björn Franke, and Nigel Topham. “Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator”. In: *Proceedings of the International Conference on Embedded Computer Systems*. IEEE, July 2010, pp. 1–10. DOI: `10.1109/ICSAMOS.2010.5642102`.

[18] Igor Böhm et al. “Generalized Just-In-Time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator”. In: *Proceedings of the Conference on Programming Language Design and Implementation*. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 74–85. ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993508`.

[19] Florian Brandner et al. “Fast and accurate simulation using the llvm compiler framework”. In: *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO*. Vol. 9. Jan. 2009, pp. 1–6.

[20] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. “An infrastructure for adaptive dynamic optimization”. In: *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. CGO ’03. `http://dl.acm.org/citation.cfm?id=776261.776290`. Washington, DC, USA: IEEE, 2003, pp. 265–275. ISBN: 0-7695-1913-X.

[21] Richard Buchmann and Alain Greiner. "A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs". In: *Proceedings of the International Conference on Microelectronics, ICM*. December. IEEE, 2007, pp. 101–104. ISBN: 9781424418473. DOI: `10.1109/ICM.2007.4497671`.

[22] Prashanth P Bungale and Chi-Keung Luk. "PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation". In: *Proceedings of the International Conference on Virtual Execution Environments*. VEE '07. New York, NY, USA: ACM, 2007, pp. 137–147. ISBN: 978-1-59593-630-1. DOI: `10.1145/1254810.1254830`.

[23] J Burnim and K Sen. "Heuristics for Scalable Dynamic Test Generation". In: *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM. 2008, pp. 443–446. ISBN: 978-1-4244-2187-9. DOI: `10.1109/ASE.2008.69`.

[24] Neil Campbell, Geraint North, and Graham Woodward. "Memory Management for a Dynamic Binary Translator". Pat. US 20120117355 A1. 2012.

[25] Matthew Chapman and Dj Magenheimer. *MagiXen: Combining binary translation and virtualization*. Tech. rep. HPL-2007-77. HP Labs, 2007.

[26] Ming Chao Chiang, Tse Chen Yeh, and Guo Fu Tseng. "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 593–606. ISSN: 02780070. DOI: `10.1109/TCAD.2010.2095631`.

[27] Younghan Choi, Hyoungchun Kim, and Dohoon Lee. "An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing". In: *Advanced Communication Technology*. 2008, pp. 473–475. ISBN: 9788955191363.

[28] Calvin Chow. *Using Simulation-based Approach to Ensure Power Integrity and Reliability of SoC and System*. Tech. rep. 2015, pp. 1–5.

[29] Moo-kyoung Chung. "Improvement of Compiled Instruction Set Simulator by Increasing Flexibility and Reducing Compile Time". In: *Proceedings of the International Workshop on Rapid System Prototyping*. Ieee, 2004, pp. 38–44. ISBN: 0-7695-2159-2. DOI: `10.1109/IWRSP.2004.1311093`.

[30] C Cifuentes and S Sendall. "Specifying the Semantics of Machine Instructions". In: *Proceedings of the International Workshop on Program Comprehension*. IWPC '98. `http://dl.acm.org/citation.cfm?id=580914.858217`. Washington, DC, USA: IEEE Computer Society, 1998, p. 126. ISBN: 0-8186-8560-3.

[31] Jiun Hung Ding et al. "PQEMU: A parallel system emulator based on QEMU". In: *Proceedings of the International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 276–283. ISBN: 9780769545769. DOI: `10.1109/ICPADS.2011.102`.

[32] Tobias J.K. Edler von Koch and Björn Franke. “Limits of Region-Based Dynamic Binary Parallelization”. In: *Proceedings of the International Conference on Virtual Execution Environments*. ACM, 2013, p. 13. ISBN: 9781450312660. DOI: `10.1145/2451512.2451518`.

[33] Marco Elver and Vijay Nagarajan. “TSO-CC: Consistency directed cache coherence for TSO”. In: *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 2014, pp. 165–176. ISBN: 9781479930975. DOI: `10.1109/HPCA.2014.6835927`.

[34] Embedded Microprocessor Benchmark Consortium. *EEMBC Benchmark Suite*. `http://www.eembc.org/`. 2008.

[35] L. Formaggio, F. Fummi, and G. Pravadelli. “A Timing-Accurate HW/SW Cosimulation of an ISS with SystemC”. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. IEEE, 2004, pp. 152–157. ISBN: 1-58113-937-3. DOI: `10.1109/CODESS.2004.240910`.

[36] Nicolas Fournel, Luc Michel, and Fr’ed’eric P’etrot. “Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators”. In: *Proceedings of the International Conference on Computer Aided Design*. IEEE, 2013, pp. 739–746. ISBN: 9781479910717.

[37] Björn Franke. “Fast cycle-approximate instruction set simulation”. In: *Proceedings of the 11th international workshop on Software & compilers for embedded systems*. ACM. 2008, pp. 69–78.

[38] M Freericks. *The nML Machine Description Formalism*. Tech. rep. `http://www6.in.tum.de/Main/Publications/Freericks1991a.pdf`. 1991.

[39] *Getting Started with DS-5, DUI0478B*. ARM Limited.

[40] Frank Ghenassia. *Transaction Level Modeling With SystemC*. 2005th ed. `http://link.springer.com/content/pdf/10.1007/b137175.pdf`. Springer, 2005. ISBN: 9780387262321.

[41] B Glamm and D J Lilja. “Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison”. In: *Proceedings of the Annual Simulation Symposium*. IEEE, 2001, pp. 72–77. DOI: `10.1109/SIMSYM.2001.922117`.

[42] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. “Automatic Test Data Generation Using Constraint Solving Techniques”. In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’98. New York, NY, USA: ACM, 1998, pp. 53–62. ISBN: 0-89791-971-8. DOI: `10.1145/271771.271790`.

[43] Peter Greenhalgh. *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. Tech. rep., pp. 1–8.

[44] M.R. Guthaus et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: *Proceedings of the International Workshop on Workload Characterization*. IEEE, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: `10.1109/WWC.2001.990739`.

[45] Ashok Halambi, Peter Grun, and Alex Nicolau. "EXPRESSION : A Language for Architecture Exploration through Compiler / Simulator Retargetability". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, 1999.

[46] Alex Heunhe Han et al. "Virtual ARM Platform for Embedded System Developers". In: *Proceedings of the International Conference on Audio, Language and Image Processing*. IEEE. 2008, pp. 586–592.

[47] R C Ho and M A Horowitz. "Validation Coverage Analysis for Complex Digital Designs". In: *Digest of Technical Papers of the International Conference on Computer-Aided Design*. ICCAD-96. ACM/IEEE, 1996, pp. 146–151. DOI: `10.1109/ICCAD.1996.569537`.

[48] DY Hong et al. "HQEMU: a Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores". In: *Proceedings of the International Symposium on Code Generation and Optimization*. `http://dl.acm.org/citation.cfm?id=2259030`. ACM/IEEE, 2012, pp. 104–113.

[49] J R Horgan and S London. "Data Flow Coverage and the C Language". In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. ACM, 1991, pp. 87–97. ISBN: 0-89791-449-X. DOI: `10.1145/120807.120815`.

[50] Joseph R. Horgan, Saul London, and Michael R. Lyu. "Achieving Software Quality with Testing Coverage Measures". In: *Computer* 27.9 (1994), pp. 60–69. ISSN: 00189162. DOI: `10.1109/2.312032`.

[51] Intel Corporation. *Enabling Intel Virtualization Technology Features and Benefits*. Tech. rep. Intel Corporation, 2010.

[52] AB Kahng et al. "ORION 2.0: a Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. `http://dl.acm.org/citation.cfm?id=1874721`. European Design and Automation Association, 2009, pp. 423–428.

[53] Rola Kassem et al. "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation." In: *Journal of Systems Architecture - Embedded Systems Design* 58.8 (2012), pp. 318–337.

[54] Sreekumar V Kodakara et al. "Model Based Test Generation for Microprocessor Architecture Validation". In: *Proceedings of the International Conference on VLSI Design*. IEEE, 2007, pp. 465–472.

[55] Toshihiko Koju et al. "Optimizing Indirect Branches in a System-level Dynamic Binary Translator". In: *Proceedings of the Annual International Systems and Storage Conference*. SYSTOR '12. New York, NY, USA: ACM, 2012, 5:1–5:12. ISBN: 978-1-4503-1448-0. DOI: `10.1145/2367589.2367599`.

[56] Rajeev Krishna and Todd Austin. "Efficient Software Decoder Design". In: *Workshop on Binary Translation*. 2001.

[57] Stephen Kyle et al. "Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-Based Just-In-Time Dynamic Binary Translation". In: *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 2012, pp. 21–30. ISBN: 9781450312127. DOI: `10.1145/2248418.2248422`.

[58] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". PhD thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[59] Bich-Caue Le. "Emulation system that uses dynamic binary translation and permits the safe speculation of trapping operations". Pat. US 6631514 B1. 2003.

[60] Markus Levy. "EEMBC and the Purposes of Embedded Processor Benchmarking". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software,* IEEE, 2005, p. 1. DOI: `10.1109/ISPASS.2005.1430553`.

[61] Sheng Li Sheng Li et al. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *Proceedings of the International Symposium on Microarchitecture*. IEEE, 2009, pp. 469–480. ISBN: 978-1-60558-798-1. DOI: `10.1145/1669112.1669172`.

[62] *Linaro Project*. `http://www.linaro.org/`. (Visited on 07/07/2015).

[63] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. "Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. 2015.

[64] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the Conference on Programming Language Design and Implementation*. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: `10.1145/1065010.1065034`.

[65] Weiqin Ma, Alessandro Forin, and Jyh-Charn Liu. "Rapid Prototyping and Compact Testing of CPU Emulators". In: *Proceedings of the International Symposium on Rapid System Protyping*. RSP. IEEE, June 2010, pp. 1–7. ISBN: 978-1-4244-7073-0. DOI: `10.1109/RSP.2010.5656339`.

[66] Weiqin Ma, Jyh-Charn Liu, and Alessandro Forin. *Design and Testing of a CPU Emulator*. Tech. rep. Microsoft Research, 2009.

[67] P.S. Magnusson and M. Christensson. “Simics: A full system simulation platform”. In: *Computer* 35.2 (2002), pp. 50–58. ISSN: 00189162. DOI: `10.1109/2.982916`.

[68] Lorenzo Martignoni et al. “Testing CPU emulators”. In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’09. New York, NY, USA: ACM, 2009, pp. 261–272. ISBN: 978-1-60558-338-9. DOI: `10.1145/1572272.1572303`.

[69] Deepak A Mathaikutty et al. “Design Fault Directed Test Generation for Microprocessor Validation”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, 2007, pp. 1–6.

[70] Peter Maydell. *risu - random instruction sequence generator for userspace testing*. `https://wiki.linaro.org/PeterMaydell/Risu`.

[71] Valerio Medeiros and David Deharbe. “Formal Modelling of a Microcontroller Instruction Set in B”. In: *Formal Methods: Foundations and Applications*. Ed. by Marcel Vinicius Medeiros Oliveira and Jim Woodcock. Vol. 5902. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 282–289. ISBN: 978-3-642-10451-0. DOI: `10.1007/978-3-642-10452-7_19`.

[72] WS Mong and J Zhu. “DynamoSim: a trace-based dynamically compiled instruction set simulator”. In: *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. `http://dl.acm.org/citation.cfm?id=1112266`. 2004, pp. 131–136.

[73] Ryan W Moore et al. “Addressing the Challenges of DBT for the ARM Architecture”. In: *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’09. New York, NY, USA: ACM, 2009, pp. 147–156. ISBN: 978-1-60558-356-3. DOI: `10.1145/1542452.1542472`.

[74] Naveen Muralimanohar and Rajeev Balasubramonian. *CACTI 6.0: A tool to understand large caches*. Tech. rep. `http://www.cs.utah.edu/~rajeev/cacti6/cacti6-tr.pdf`. 2009.

[75] A. Muttreja et al. “Hybrid Simulation for Embedded Software Energy Estimation”. In: *Proceedings of the Design Automation Conference*. ACM/IEEE, 2005, pp. 23–26. ISBN: 1-59593-058-2. DOI: `10.1109/DAC.2005.245623`.

[76] Sebastian Ottlik et al. “Context-Sensitive Timing Simulation of Binary Embedded Software”. In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. `http://dl.acm.org/citation.cfm?id=2656117`. 2014. ISBN: 9781450330503.

[77] Preeti Ranjan Panda. “SystemC - A Modeling Platform Supporting Multiple Design Abstractions”. In: *Proceedings of the International Symposium on Systems Synthesis*. IEEE, 2001, pp. 75–80.

[78] Terence J. Parr and Russell W. Quong. “ANTLR: A predicated-LL (k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[79] A Patel et al. “MARSS: a full system simulator for multicore x86 CPUs”. In: *Proceedings of the Design Automation Conference*. `http://dl.acm.org/citation.cfm?id=2024954`. ACM/IEEE, 2011.

[80] Stefan Pees et al. “LISA - Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures”. In: *Proceedings of the Design Automation Conference*. DAC ’99. New York, NY, USA: ACM/IEEE, 1999, pp. 933–938. ISBN: 1-58113-109-7. DOI: `10.1145/309847.310101`.

[81] DC Powell and B Franke. “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 ldots*. `http://dl.acm.org/citation.cfm?id=1629478`. 2009, pp. 315–324.

[82] Wei Qin and Sharad Malik. “Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits”. In: *Proceedings of the Design Automation Conference*. 2003, pp. 764–769.

[83] Wei Qin and Sharad Malik. “Flexible and formal modeling of microprocessors with application to retargetable simulation”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’03. Washington, DC, USA: IEEE, 2003, pp. 556–561. ISBN: 0-7695-1870-2. DOI: `10.1109/DATE.2003.1253667`.

[84] Norman Ramsey and Mary F Fern’andez. “Specifying representations of machine instructions”. In: *ACM Transactions on Programming Languages and Systems* 19.3 (May 1997), pp. 492–524. ISSN: 0164-0925. DOI: `10.1145/256167.256225`.

[85] Alasdair Rawsthorne, John Harold Sandham, and Jason Souloglou. “Exception handling method and apparatus for use in program code conversion”. Pat. US 7353163 B2. 2008.

[86] Alasdair Rawsthorne et al. “Block translation optimizations for program code conversation”. Pat. US 20040255279 A1. 2004.

[87] M Reshadi, P Mishra, and N Dutt. “Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation”. In: *ACM Transactions on Embedded Computing Systems* 8.3 (2009). `http://dl.acm.org/citation.cfm?id=1509292`.

[88] Mehrdad Reshadi et al. “An efficient retargetable framework for instruction-set simulation”. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. New York, New York, USA: IEEE/ACM/IFIP, 2003, pp. 13–18. ISBN: 1581137427. DOI: `10.1109/CODESS.2003.1275249`.

[89] Victor Reyes. *Using Virtual Prototypes to Address the Growing Software Complexity in Automotive*. Tech. rep. November. Synopsys, 2013, pp. 1–18.

[90] Sandro Rigo and G Araujo. “ArchC: A SystemC-Based Architecture Description Language”. In: *Proceedings of the Symposium on Computer Architecture and High Performance Computing*. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1364738`. IEEE, 2004, pp. 66–73.

[91] J.A. Rowson. “Hardware/Software Co-Simulation”. In: *Proceedings of the Design Automation Conference*. ACM/IEEE, 1994, pp. 439–440. ISBN: 0-89791-653-0. DOI: `10.1109/DAC.1994.204143`.

[92] Patrick Schaumont and Ingrid Verbauwhede. “A Component-Based Design Environment For ESL Design”. In: *Design & Test of Computers* 23.5 (2006), pp. 338–347.

[93] Eric Schnarr and James R. Larus. “Fast Out-of-Order Processor Simulation Using Memoization”. In: *Proceedings of the International Conference on Architectural support for Programming Languages and Operating Systems*. New York, New York, USA: ACM, 1998, pp. 283–294. ISBN: 1581131070. DOI: `10.1145/291069.291063`.

[94] J Schnerr and O Bringmann. “High-performance timing simulation of embedded software”. In: *Proceedings of the 45th annual Design Automation Conference*. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4555825`. 2008, pp. 290–295.

[95] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: a concolic unit testing engine for C”. In: *Proceedings of the European software engineering conference*. Ed. by Michel Wermelinger and Harald Gall. ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0.

[96] Alexander Sepp, Julian Kranz, and Axel Simon. “GDSL: A Generic Decoder Specification Language for Interpreting Machine Language”. In: *Electronic Notes in Theoretical Computer Science* 289 (2012), pp. 53–64. ISSN: 15710661. DOI: `10.1016/j.entcs.2012.11.006`.

[97] Konstantin Serebryany and Derek Bruening. “AddressSanitizer: a fast address sanity checker”. In: *Proceedings of the Usenix Annual Technical COnference*. `https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf`. USENIX Association, 2012, p. 28. ISBN: 978-931971-93-5.

[98] Marc Serughetti. *Software Development Using Virtual Hardware Platform*. Tech. rep. CoWare, 2007.

[99] Tom Spink et al. “Efficient code generation in a region-based dynamic binary translator”. In: *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*. ACM, 2014, pp. 3–12.

[100] Evgeniy (Google) Stepanov and Konstantin (Google) Serebryany. “MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. ACM/IEEE, 2015, pp. 46–55.

[101] Peter Strazdins, Bill Clarke, and Andrew Over. “Efficient Cycle-Accurate Simulation of the UltraSPARC III CPU”. In: *Proceedings of the Australasian conference on Computer science*. 2007, pp. 221–228.

[102] Synopsys. *Synopsys DesignWare ARC Processor Cores*. `http://www.synopsys.com/IP/ProcessorIP/ARCProcessors/`. 2015.

[103] Christopher Thompson, Miles Gould, and Nigel Topham. “High Speed Cycle Approximate Simulation for Cache-Incoherent MPSoCs”. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, July 2013, pp. 88–95. ISBN: 978-1-4799-0103-6. DOI: `10.1109/SAMOS.2013.6621110`.

[104] Nigel Topham and Daniel Jones. “High Speed CPU Simulation using JIT Binary Translation”. In: *Proceedings of the Annual Workshop on Modeling, Benchmarking and Simulation*. MOBS’07. Springer-Verlag, 2007, pp. 50–64.

[105] *Transitive technology: The Rosetta Stone for binary translation*. `http://www.cs.manchester.ac.uk/our-research/research-impact/transitive-technology/`. (Visited on 03/26/2015).

[106] Harry Wagstaff, Thomas Spink, and Björn Franke. “Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded System*. `http://dl.acm.org/citation.cfm?id=2656113`. ACM, 2014.

[107] Harry Wagstaff et al. “Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description”. In: *Proceedings of the Design Automation Conference*. DAC ’13. New York, NY, USA: ACM/IEEE, 2013, 21:1–21:6. ISBN: 978-1-4503-2071-9. DOI: `10.1145/2463209.2488760`.

[108] VM Weaver and S McKee. *Are Cycle Accurate Simulations a Waste of Time?* Tech. rep. `http://www.csl.cornell.edu/~vince/papers/wddd08/wddd08_workshop.pdf`. 2008.

[109] Emmett Witchel and Mendel Rosenblum. “Embra: Fast and Flexible Machine Simulation”. In: *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’96. New York, NY, USA: ACM, 1996, pp. 68–79. ISBN: 0-89791-793-6. DOI: `10.1145/233013.233025`.

[110] *X86 Instruction Decoding (Gem5)*. `http://m5sim.org/wiki/index.php/X86_Instruction_decoding`. 2009. (Visited on 02/11/2015).

[111] Hoonmo Yang and Moonkey Lee. “Embedded Processor Validation Environment Using a Cycle-Accurate Retargetable Instruction-Set Simulator”. In: *The Journal of Supercomputing* 33.1-2 (2005), pp. 19–32. ISSN: 0920-8542. DOI: `10.1007/BF02764138`.

[112] Matt T. Yourst. “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4211019`. IEEE, Apr. 2007, pp. 23–34. ISBN: 1-4244-1081-9. DOI: `10.1109/ISPASS.2007.363733`.

[113] Michal Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/`. (Visited on 03/01/2015).

[114] Hong Zhu, Patrick a. V. Hall, and John H. R. May. “Software Unit Test Coverage and Adequacy”. In: *ACM Computing Surveys* 29.4 (1997), pp. 366–427. ISSN: 03600300. DOI: `10.1145/267580.267590`.

[115] Vojin Zivojnovic, S Pees, and Heinrich Meyr. “LISA-Machine Description Language and Generic Machine Model for HW/SW Co-Design”. In: *Proceedings of the Workshop on VLSI Signal Processing*. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=558311`. IEEE, 1996, pp. 127–136. ISBN: 0780331346.