ARM Instruction Set Simulation on Multi-Core x86 Hardware

Prepared by: Lee Wang Hao
a1148903
Supervisor: Dr. Bradley Alexander

A thesis submitted for partial fulfillment
to the requirements for an Honours Degree in Computer Science

The University of Adelaide
School of Computer Science

June 19, 2009

Abstract

Dynamic translation is a popular technique to build Instruction Set Simulators (ISS). The dynamic translation process must be fast or the translation overhead will outweigh the benefits of reusing code previously translated. In real hardware, decoding and execution are performed by separate units in parallel. With the arrival of multi-core x86 desktops, it is now possible to do translation and execution of the target processor's instructions in parallel so that Dynamic Binary translation can be much faster. This thesis reports on a Multi-processing approach used for an ARM ISS called SimIt-ARM v3.0 [10]. The simulator is used in exploring different techniques to improve the performance of parallel dynamic translation. Experiments show that Multi-processing provides significant improvement in simulation performance. Further speedup can be realised with more extensive optimisation of code and pre-translation of target instructions by slave processors.

Contents

0.1 Acknowledgments . . . 3

1 Introduction **4**
1.1 Thesis Organisation . . . 5
1.2 Notations and terminology . . . 5

2 Literature Review **7**
2.1 Types of Simulation . . . 7
2.1.1 Cycle Accurate Simulation (CAS) . . . 7
2.1.2 Functional Instruction Set Simulation (ISS) . . . 8
2.2 Techniques for Designing Simulators . . . 8
2.2.1 Interpretive Simulation . . . 8
2.2.2 Binary Translation . . . 10
2.2.3 Static Binary Translation . . . 10
2.2.4 Dynamic Binary Translation . . . 11
2.3 Compiled Simulation . . . 12
2.4 Generating a Simulator . . . 13

3 SimIt-ARM-3.0 - A Multiprocessing ARM ISS **14**
3.1 Interpretive and Dynamic-compiled simulation . . . 14
3.2 System and User Level Simulation . . . 15
3.3 Translation Caching . . . 15
3.3.1 Persistent Caching . . . 17
3.3.2 Block Linking . . . 17
3.4 Floating Point Simulation . . . 18
3.5 Self-Modifying Code . . . 18
3.6 Multiprocessing . . . 19
3.7 Devices . . . 19
3.8 Debugger . . . 19
3.9 Simulation synthesizer with MADL . . . 19

4 Methodology **21**
4.1 Overview . . . 21
4.2 Investigating Parallel Programming Models . . . 21

4.2.1 POSIX Threads (Pthreads) . . . 22
4.2.2 OpenMP . . . 22
4.3 Improving Quality of Translated Code . . . 22
4.3.1 Stitching Translated Blocks . . . 23
4.3.2 Using Higher Optimisation Levels . . . 24
4.3.3 Idle-Time re-optimisation . . . 25
4.4 Improving Locality of Translations . . . 27
4.4.1 RAM Disks . . . 27
4.4.2 Producing Code Images . . . 27
4.5 Reducing Time Spent in Interpretation . . . 27
4.5.1 Speculative Translation . . . 27
4.5.2 Varying Translation Threshold . . . 28
4.6 Test Setup . . . 29
4.7 Metrics . . . 30

5 Results and Discussion **31**
5.1 Original Simulator Results . . . 31
5.2 Results on different threading models . . . 32
5.3 Results with optimising code quality . . . 35
5.3.1 Results on translating at higher Optimisations . . . 35
5.3.2 Results on Idle-Time optimisation . . . 37
5.3.3 Results on Block-Stitching . . . 39
5.4 Results of Locality . . . 40
5.4.1 Producing Object Code images . . . 40
5.4.2 Results of using a RAM-Disk . . . 42
5.5 Results on doing less interpretation . . . 43
5.5.1 Reducing Translation Threshold . . . 43
5.5.2 Results of speculative translation . . . 44
5.6 Other Experiments . . . 47
5.6.1 Persistent Caching . . . 47
5.6.2 Persistent Caching benefits . . . 48
5.6.3 System Level Emulation . . . 49

6 Conclusion and Future Work **50**
6.1 Investigate more optimisation engines and Passes . . . 50
6.2 Proactive Compilation strategies . . . 51
6.2.1 Incorporating the Block Graph analysis . . . 51
6.3 Using Linked Basic blocks . . . 52
6.4 Summary of Findings . . . 52

Appendices **65**

List of Figures

2.1 The typical interpretive simulation template algorithm 9
2.2 FacSim's [9] seperates Functional simulation and Cycle Accurate tracing for parallelism 9
2.3 Binary translation and optimisation process 10

3.1 Overall Simulator Organisation 15
3.2 Instruction-set translation, compilation and optimisation process in SimIt-ARM [10] 16
3.3 Switching from interpretive mode and back 18

4.1 Disjoint translated blocks 23
4.2 Idle time Re-Optimisation 26

5.1 Speedup in original configuration 31
5.2 Open MP with 2 processors 32
5.3 OpenMP version of translation job queue, note the absence of the `wait` construct 34
5.4 Ratio of time in translation to code execution for the (test) h264 benchmark 36
5.5 Translating with 2 processors at various optimisation levels 37
5.6 Effects of Optimisation on short runs after lowering translation threshold to 1m 38
5.7 Expected gain from Idle-time Re-optimisation 38
5.8 Simple function that returns an integer 41
5.9 Simple function image that returns an integer 41
5.10 Use of a RAM-Disk 42
5.11 Potential benefits of speculative translation 45

1 Partial code body of thread doing idle-time reoptimisation 55
2 Pthread version of slave thread body 56
3 OpenMP version of slave thread body 57
4 Pthreads parallel Pi program 58
5 OpenMP parallel Pi program 59
6 Code that calls the function image 60
7 Varying translation thresholds on the 473.astar benchmark 60

8 Varying translation thresholds on the 401.bzip benchmark 61
9 Varying translation thresholds on the 445.gobmk benchmark 61
10 Varying translation thresholds on the 464.h264 benchmark 62
11 Varying translation thresholds with -O3 on the 473.astar benchmark 62
12 Varying translation thresholds with -O3 on the 401.bzip benchmark 63
13 Varying translation thresholds with -O3 on the 445.gobmk benchmark . . . 63
14 Varying translation thresholds with -O3 on the 464.h264 benchmark 64

0.1 Acknowledgments

I would like to thank my wonderful supervisor, Dr. Bradley Alexander for his great amount of help and support rendered in the course of this project even when we're both overseas. I would also like to thank David Knight for his expertise on the Instruction-Sets and buffer-cache, you've answered one of my burning questions. Thanks! Special thanks also go to Francis Vaughan, Geoff Randall, Rod Whitby and Travis Olds from ASTC (Australian Semiconductor Technology Company) for all the grisly low-level technical help. Thank you Andrew Jeffrey for sharing your QEMU translation speeds in your project as I cannot do things *in parallel* :). I would also like to thank Debby Coleman-George from the Writing Centre for the help in proof reading this project's proposal and your informative seminars on thesis and academic writing. Thanks also go to Donna Velliaris from CLPD (Centre for Learning and Professional Development) for the informative and useful seminars to perfect my writing skills.

Chapter 1

Introduction

Designing complex electronic devices such as a new mobile phone or an MP3 player has become a very challenging task. Competition from rival manufacturers make time-to-market a critical determinant of the new products success in competing for a substantial market share.

To speed up the design process of the new device, hardware and software development has to proceed simultaneously. Ideally, the first prototype of the hardware is completed together with the software so testing can start at the earliest possible stage. This means that fast and accurate simulation of the hardware has to be available to the device software developers as they are developing the product without actual hardware.

Portable electronic devices these days are often based upon specialised embedded RISC processors. The main problem faced in creating RISC Simulators is that the simulators execution speed is typically much slower than the actual hardware. Simulation environments typically have to run on a software engineer's workstation, and integrate with the tools used for compilation and debugging. The most common environment for the software team are desktop computers that use an x86 processor. Thus, simulation is limited by the speed at which an x86 processor can emulate a RISC.

In order to improve the execution speed of Instruction Set Simulators, aggressive techniques are used. One such technique is Static Binary Translation where blocks of instructions are translated (in some cases compiled to native code) before execution so no additional effort is required when the target ARM code is needed again. On standard von

Neumann Architectures however, data and code reside in the same memory space, making this process difficult due to the problems [1] such as dynamic linking and self-modifying code. Hence there is now a requirement for performing Dynamic Binary Translation which translates along with program execution. This however, introduces a new problem; Dynamic translation must be fast otherwise the translation overhead will be higher than simply interpreting the translated code prior to caching.

With the arrival of multi-core x86 desktops, it is now possible to do translation and execution of the target processor's instructions in parallel so that Dynamic Binary translation can be potentially faster.

This thesis presents work on exploring the use of readily available parallelism to improve a Multi-processing ARM Instruction Set Simulator, SimIt-ARM. SimIt-ARM utilises a technique called Dynamic Compiled Simulation, which first starts off with interpretive simulation and translates frequently interpreted/executed instruction blocks into C++ dynamically linking libraries (DLLs). Due to the surfeit of processor cores available now, new ways of managing the activities performed by each core during simulation can be explored.

1.1 Thesis Organisation

This thesis is organised as follows, first is a historical overview of the styles and techniques in Instruction Set Simulation in chapter 2. Next, the Multi-processing Simulator, SimIt-ARM is described in some detail in chapter 3. Then the experimental work performed on SimIt-ARM is outlined in chapter 4. The report then documents the experiment results in chapter 5 along with the discussion about the experimental outcomes and the problems that were encountered. Finally the last chapter concludes the thesis by providing suggestions for future work and and summarising the findings.

1.2 Notations and terminology

- 'Target code' or 'Target Instructions' refer to the instructions of the guest architecture that is being simulated; in this work, it refers to the ARM processor instructions; 'host' refers to the machine the simulator is running on i.e. an x86 in this case.

- The terms ISS (Instruction-Set Simulator), 'simulator' and 'emulator' mean the same thing, are often used and are interchangeable.
- The term cache unless otherwise specified, refer to the simulator's translation cache.
- The term 'Block' usually, unless specifically stated refers to a fixed-size chunk of the ARM memory space containing instructions that has been cached.
- A 'Basic block' is different from a 'block'. A basic block of instructions is one with which there is only one entry point, no branch/jump instructions in between and terminated by a jump or branch
- The terms 'Block' and 'DLL' all refer to a unit of cached code in the simulator's translation cache and are interchangeable.

Chapter 2

Literature Review

There are many ways to create an Instruction-set Simulator. Sometimes, speed is not the only definitive characteristic of a quality emulator. Performance also conflicts with flexibility and accuracy. Depending on the usage of the Simulator, engineers use them to emulate new architectures or develop firmware or software for a new architecture that is not released. In such a case, fine-grained simulation of various aspects of the new hardware have to be catered for. Overall it is a cumbersome problem to develop fast emulators while ensuring its accuracy at the same time. In this chapter, the types of simulation are briefly described, then the various techniques of simulation are discussed and finally related work on sequential and parallel simulators are briefly examined.

2.1 Types of Simulation

Broadly, there are two main types of architecture simulators: *Cycle Accurate Simulators* and *Functional Instruction Set Simulators.*

2.1.1 Cycle Accurate Simulation (CAS)

A Cycle Accurate Simulator simulates new or present micro-architecture designs and includes timing information and pipeline simulation. This type of accuracy is useful for time dependent applications such as process control but is usually much slower than a functional Instruction Set Simulator. Timing information may not be completely accurate but

a tolerance level of ±7% is quite acceptable. These types of simulators are usually used for examining properties of new processor micro-architecture designs and low-level software verification.

FacSim [9] is an example of a cycle-accurate simulator. It uses multi-processing to simulate the ARM9E-S processor core and ARM926EJ-S processor's memory subsystem. FaCSim exploits the functional model/timing-model decoupling [9], and is divided into a functional front-end and a cycle-accurate back-end that can be run in parallel.

2.1.2 Functional Instruction Set Simulation (ISS)

An Instruction Set Simulator mimics the *behavior* of a micro-controller by examining target instructions that maintain variables that correspond to the state of the simulated processor. It loses the timing accuracy of a CAS but does ensure that target code runs correctly on a host machine. As it does not simulate minor details of the hardware, an ISS is usually much faster than CAS. It is usually used to verify the correct *functionality* of a program written for the new target hardware before it is available.

Most of the literature survey done in this project focuses on Functional Instruction-set Simulators.

2.2 Techniques for Designing Simulators

There are various strategies to design an architecture simulator, some provide better speed or accuracy than others, each has its uses and merits.

2.2.1 Interpretive Simulation

This is the simplest way to design an ISS or CAS. The approach is to interpret each instruction and execute them step by step using a big switch statement on the host machine operating over the simulated state of the target hardware. Figure 2.1 shows an outline algorithm of this approach. This is a very flexible approach and because it is easy to implement, work has been done to generate (or synthesize) such types of simulators quickly from Architecture Description Languages (ADLs). However, despite its flexibility and

```
while(running){
        next_instr = get_PC();
        instr = decode(next_instr);

        switch(instr){
                      ....
             add: perform_add(op_1, op_2);
                      ...
        }
}
```

Figure 2.1: The typical interpretive simulation template algorithm

ease of implementation, interpretive simulators are also by far the slowest in simulation performance.

Again, FacSim [9] is a cycle-accurate simulator that interprets instructions for accuracy; as it separates the functional front-end from its cycle-accurate back-end via a non-deterministic queue so it is possible to use a multi-processing approach to speedup cycle-accurate simulation (see figure 2.2). Furthermore the high number of hardware components to simulate in a CAS justifies the use of more processors for produce better speedup [15].

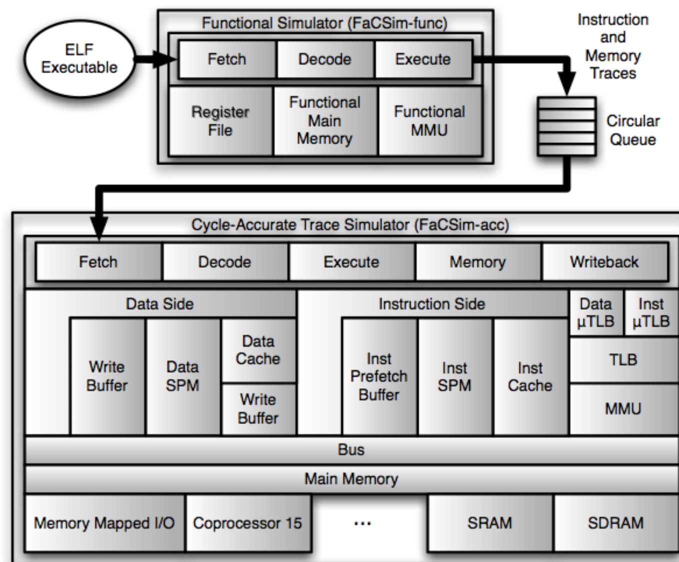


Figure 2.2: FacSim's [9] seperates Functional simulation and Cycle Accurate tracing for parallelism

2.2.2 Binary Translation

This technique is common in adapting a legacy piece of program to run on a newer platform. E.g. code for a PowerPC Mac on an Intel i386. Examples of systems that use the binary translation technique include VMware[2], Bochs [3] and UQBT[4]. The approach is slightly more complicated. The legacy program's binary code is translated directly (or through an intermediate representation in some cases like in figure 2.3) and the program operates on the host machine's devices. This process is usually much faster than the interpretive approach as only the instruction's sematics are different (i.e. an `add` instruction on RISC vs an `ADD` x86 instruction, exception being use of flags on an x86). Figure 2.3 shows the typical process of simulators that use the Binary translation approach. On the other hand, it is unable to emulate the state of the target hardware and is usually unsuitable for verification of new design of hardware devices such as in a CAS. There are two main ways of doing Binary Translation, *static* and *dynamic*.

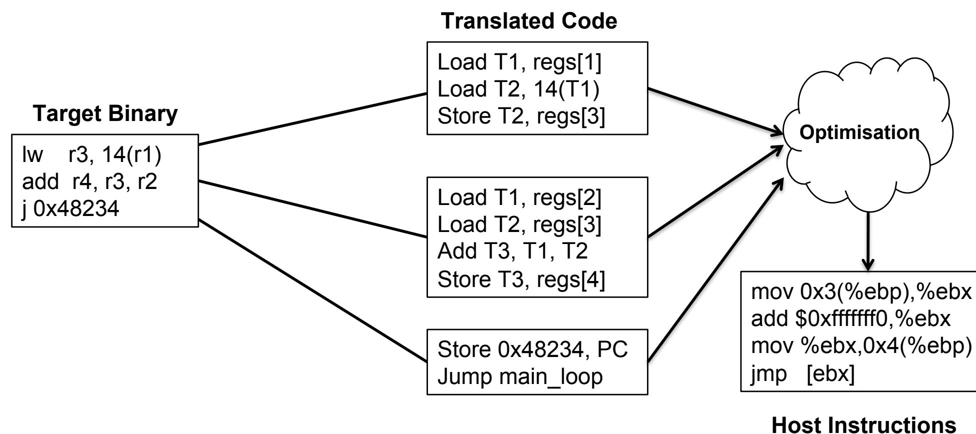


Figure 2.3: Binary translation and optimisation process

2.2.3 Static Binary Translation

Binary translation can be done statically by translating the whole target binary into host binary. This process is very much like compilers or language byte-code translators. The resulting program then runs as usual on the host machine.

The main advantage of this approach is that it runs the translation process only once. The resulting code can be run quickly for as many times as desired (until there is a change

in the source code or binary). The resulting code is also fast as various types of optimisation passes can be done at compile/translation time.

On the other hand, this approach is not flexible. On standard von Neumann Architectures, data and code reside in the same memory space, making this process difficult due to the problems such as dynamic linking and self-modifying code[1]; a technique used by boot-loaders and encryption software. If the program is large, a lot of time will be spent waiting for code to be fully-translated. This is also not beneficial on a developer's point of view as changes to source code can are made and are optimised very frequently. The entire program has to be re-translated on every change, be it small or large.

2.2.4 Dynamic Binary Translation

Dynamic Binary Translation translates along with program execution. Unlike the static approach, dynamic binary translation usually only translates (typically basic blocks) code when it is needed. Translated code are also cached in case they are needed again and there are usually rules for evicting cached entries much like an operating system manages its virtual memory.

This approach has some advantages over static translation. Firstly, there is the benefit of flexibility. It is possible to simulate self-modifying code and dynamic linking. Secondly, there is also no start-up latency for translation before program simulation. Redundant code not executed during a specific test run will not be translated. Saving some storage space and potentially reducing overall translation effort.

However, dynamic translators are slower to execute due to the fact that translation happens along with execution. If the translation effort is large, the program is short or cached entries are not frequently used (poor instruction locality) the translation overhead will be higher than cost of running the translated code.

Shade [6] was one of the first Processor simulators that used dynamic binary translation and caching. It is used to trace SPEC benchmarks on SPARC systems to help build better SPARC hardware and software.

In [7], Embra simulated a MIPS R3000/4000 running SPEC92 benchmarks on a Sili-

con Graphics IRIX 5.3 at 3 to 9 times slower than native execution of the workload. It demonstrated the capabilities of dynamic instruction translation of the simulated hardware.

To facilitate target-adaptability, researchers from University of Queensland and Sun Microsystems proposed a machine-adaptable dynamic binary translation approach in 1999 [4]. However, porting to another host machine requires extensive assembly debugging and testing.

QEMU [5] is a very fast machine simulator used to run simulate full machines with their target OS images on different host architectures. However, it is not easily portable and target-adaptable. Hence, its purpose is mainly used to run foreign Operating Systems in virtual environments.

As very little[1] time is spent on translating code in binary translators like QEMU. It is not of much worth performing translation in parallel. As a programming language is not usually used for binary translators, it is more difficult to optimise translated code.

2.3 Compiled Simulation

Compiled Simulation is very similar to binary translation; except for the fact that a programming language is often used as the intermediate representation for translation. This is to provide portability of the simulator. In other words, the simulator can be run on various types of host hardware. Like binary translation, compiled simulation also can be done statically or dynamically.

The main advantage of this approach is that the simulator can be ported with little effort to another architecture. This is beneficial if the organisation deploys various platforms of computers that will be used for design and simulation.

The main disadvantage is the additional overhead introduced by translating from target Instructions to the High-Level language (i.e. C++) and compiling the consequent C++ code to host instruction set. This is especially so if it is implemented dynamically. Some

[1]QEMU spends on average only 1.7 seconds on translation in 4 benchmarks tested for another project underway by Andrew Jeffery. This is a small value compared to several seconds to a few minutes in SimIt-ARM

JIT interpretive-compiled simulation techniques such as SimIt-ARM [10], EHS [14] and the Java Programming Language use an interpretive mode at the beginning to profile code before starting translation so the performance of short programs is limited by the speed of the interpretive simulator. It also suffers from the initial latency when running slightly longer user-interactive programs.

Zhu et. al. [8] is an example of a static compiled, just-in-time simulation technique for fast and flexible simulation. Their approach is quite different to the overall simulation technique used by SimIt-ARM as it uses a hardware independent register allocation interface together with a standard C compiler for code generation.

As compiled simulation uses a programming language as the intermediate form, it can potentially take a very long time to translate code as speed is dependent on the decoder and the compiler of the language. Hence it is worthwhile to do translation in parallel for a simulator that uses the compiled simulation method.

SimIt-ARM is described in the upcoming chapter.

2.4 Generating a Simulator

Due to the need for modularity and target-adaptibility, simulators are sometimes not hand-written. An interpretive simulator is easy to implement so it is usually generated. As a compiler's code generator engine is used in compiled simulators to produce host machine code, there is also better modularity and simplicity in implementation. Hence, like interpretive simulators, many compiled simulators are also generated instead of being hand-written. In the literature, [8], [10], [11] and [14] are examples of compiled simulators synthesised with (Architecture Description Languages) ADLs. Usually, an architecture's instruction set, (i.e. fields, op-codes and operands) are described in a file. Then a decoder for the instruction set is generated; after which the peripherals and devices are described. Finally, the ISS is generated from these input files. On some implementations like in [19], a target program for the guest machine is also used one of the input to examine the behavior and generate a simulator.

Chapter 3

SimIt-ARM-3.0 - A Multiprocessing ARM ISS

SimIt-Arm is a series of Instruction set Simulators done by Wei Qin et. al [10]. It is the main tool used in this project to explore threaded simulation on x86 chip-multiprocessors. It runs both system-level and (statically-linked) user-level ARM programs. It currently does not support Thumb(R) instructions. SimIt-ARM employs two styles of simulation: interpretation and dynamic-compiled simulation. It is developed for the IA32 Linux platform but since it uses dynamic-compiled simulation, it should in principle, work for other platforms as well. SimIt-ARM is free, open-source software under the GNU General Public License. This section introduces the internals of SimIt-ARM v3.0.

3.1 Interpretive and Dynamic-compiled simulation

SimIt-ARM starts off by interpreting the ARM instructions using its interpretive mode. When it discovers that a block of instructions are executed for a certain threshold amount of times, it translates that block of ARM instructions to a C++ function. It then uses GCC to compile the said C++ function to a linux shared library which is then linked into the simulator engine. Figure 3.1 shows the overall design of SimIt-ARM.

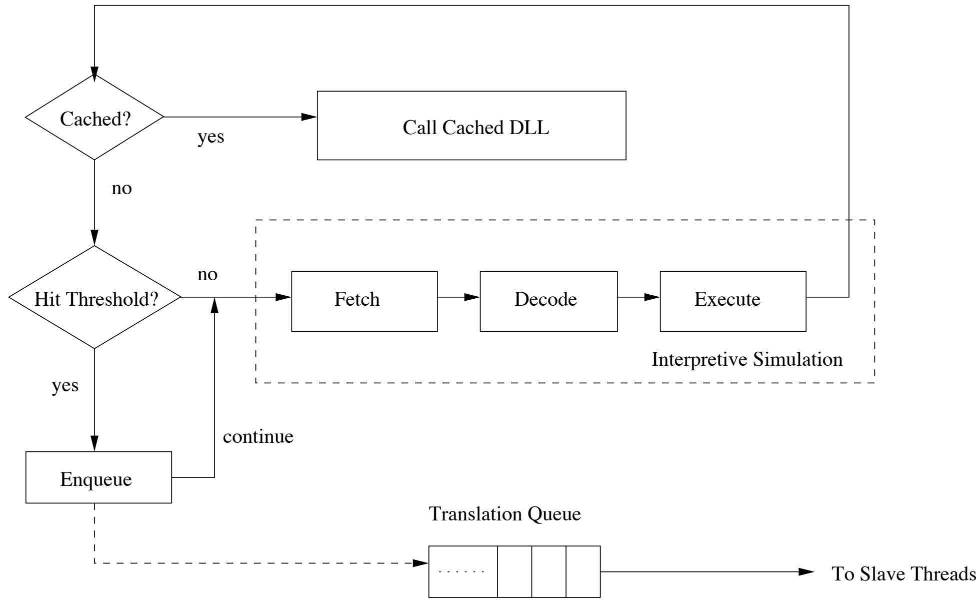


Figure 3.1: Overall Simulator Organisation

3.2 System and User Level Simulation

SimIt-ARM supports the simulation of User-space programs as well as System-level emulation. For system emulation, it is capable of booting up an ARMLinux image on a simulated ARM device operating over the host machine. The latter is possible as the flexibility of dynamic-compiled approach makes it possible to detect and simulate self-modifying code.

There are different variations of compiled simulation. In many cases, C++ code is used as the intermediate representation of translation. The simulation engine simulates several instructions in each calling of a compiled routine. Some compiled simulation approaches such as IS-CS [8] simulates one instruction in a call to the cached routine. In contrast, SimIt-ARM uses a 512-word block of instructions so that one call to the cached routine simulates many instructions at once. Figure 3.2 shows the process of translation in SimIt-ARM.

3.3 Translation Caching

The basic translation unit is a block of ARM instructions with a predefined size. For user-level programs, a 512-word translation unit is used as it can be readily aligned at a

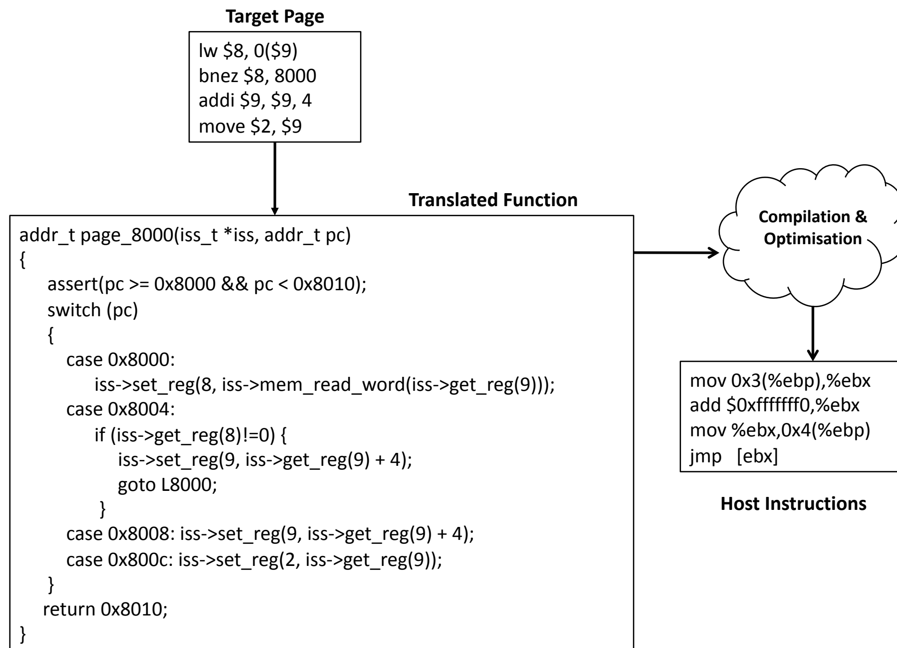


Figure 3.2: Instruction-set translation, compilation and optimisation process in SimIt-ARM [10]

2KB boundary. For system mode emulation, a 256-word translation unit is used as it has to support ARM devices' *Tiny* page sizes and a code block has to be aligned according to these specifications.

In the source-code, it is deduced that the translation cache indexes up to 2^{21} locations of 512-word blocks when running in user-mode and 2^{22} locations of 256-word pages in system emulation to map physical-pages on the target ARM device.

A 512 or 256 word block of translated code is aligned at physical address boundaries. This is to facilitate the computation of instruction block address mapping to the simulator's physical memory space [10]. In user-mode simulation, no eviction strategy is used for removing stale cache entries as it is generally assumed (by inspection of the source code) that the size of the Translation cache look-up table indexes $2^{21} \times 512_words$; a size big enough to index the largest target ARM program. In system mode when self-modifying

code is detected, the cache entry affected by the over-written code is then removed from the cache look-up array. The cache related files on disk are not removed so that subsequent runs of the virtual environment can be faster due to persistent caching (discussed in the next sub-section).

3.3.1 Persistent Caching

SimIt-ARM uses a hidden folder in the user's home directory as a base for its translation cache. Because it is stored on the disk. Cached code can be re-used for subsequent runs of the same program. The translation cache consists of the following:

- **C++ files** that contain the C++ function that represents the block of translated code.
- **A Dictionary** that contains the file index of the C++ functions and their corresponding checksum value. The checksum is obtained by adding all bytes of the instruction words in a block. A single read to the dictionary file will reveal if the block of code is familiar to the simulator engine. It is used in conjunction with the raw binary files to do matching as checksums may collide at a low probability with various combinations of the same instructions.
- **Raw ARM binary block files** that corresponds to 512/256 (depending on mode) words of the ARM code that has been translated. They are used to match if the block of instructions in memory matches that in the file.
- **DLL files** that contain the compiled C++ functions hat corresponds to the translated block of ARM instructions. These files are loaded into the simulator engine when it finds the same code on a subsequent run.

3.3.2 Block Linking

The translated code blocks are linked together via a loop and during jumps and conditional branches to a destination outside of the block, it is checked that if the next executing block is also cached, it immediately jumps into the next cached block; instead of returning to the interpretive dispatch loop. Otherwise, it exits the cache loop and continues with

interpretive simulation. Figure 3.3 shows the cached vs interpretive dispatch loop.

```
while(running)
{
        //cached part
        while(PC_value = cached_entries[PC%blockSize])
        {
                call_cache(&cached_entries[PC%blockSize]);
        }

       next_instr = get_PC();
       instr = decode(next_instr);

       //interpretive part
       switch(instr)
       {
              ....
            add: perform_add();
              ...
       }
}
```

Figure 3.3: Switching from interpretive mode and back

3.4 Floating Point Simulation

Currently, the simulator does not support Floating-point code. If it encounters a piece of floating point instruction, it calls the host OS implementation of floating-point emulation library (usually NetWinder) or other Soft-float techniques to simulate the instruction.

3.5 Self-Modifying Code

SimIt-ARM simulates self-modifying code by using the `setjmp` or `longjmp` routines as part of the ANSI C library to throw an exception. All memory operations are monitored, if a memory write occurs to a block that has been compiled/translated, an exception is

triggered to interrupt simulation. The translated shared library that corresponds to the modified block is then unloaded and simulation resumes.

3.6 Multiprocessing

Compared to the direct translation approach in QEMU [5] or UQBT [4], the GCC-based approach is easier to implement and is highly portable. However, this comes at the cost of translation speed. To reduce translation delay, SimIt-ARM distributes translation tasks to other CPUs/cores or workstations via either pthreads on a single node with shared memory chip-multiprocessors or sockets on distributed memory clusters respectively. The authors tested SimIt-ARM 3.0 on a cluster of 6 commodity PCs but official figures for chip-multiprocessor runs are not available.

3.7 Devices

SimIt-ARM has classes for a co-processor, an MMU and a soft-UART controller for a terminal device. A TCP/IP stack is available on the packaged ARM-Linux Images but a simulated Ethernet device is not yet available.

3.8 Debugger

In SimIt-ARM, there is a light-weight debugging interface that allows the programmer to step through the program and dump of status register and memory values.

3.9 Simulation synthesizer with MADL

SimIt-ARM's binary decoder and Simulation engine are synthesized using the Mescal Architecture Description Language (MADL) [11]. This approach is able to easily develop and test new architectures by just describing them using the MADL and its novel concurrency Operation State Machine (OSM) model. The description language can also be used to generate fast and highly accurate decoders for the another described architecture (i.e. the DLX machine in our school). SimIt-ARM-3.0 is largely generated using MADL. The only hand-coded parts of the simulator are the sockets and threading model used in doing

multi-processing simulation. To date, the current MADL implementation is currently able to generate only interpretive and single threaded compiled simulators.

Chapter 4

Methodology

4.1 Overview

SimIt-ARM shall be built and tested with default parameters on 1-4 cores/processors. Following which the many ways to exploit parallel execution of a simulator to improve simulation performance are explored. Among them are the usage of different programming models (various shared memory threads), improving utilisation of slave processors/threads; various methods of improving utilisation include online code reoptimisation, translating target code at higher initial optimisation levels and pre-prediction of frequently executed blocks. Attempts to improve locality of translated code is also made in the form of copying the whole operation of the Simulator into a portion of the host ramdisk and producing executable images instead of the heavier DLLs. The details of each experiments and problems encountered are described.

4.2 Investigating Parallel Programming Models

There are various parallel programming models available today. Some used the shared memory architecture and another paradigm uses the distributed memory model.The perhaps most popular distributed memory model is MPI (Message Passing Interface). The ones relevant to the problem at hand, which is to investigate running efficiency on chip multiprocessors (a kind of shared momory machine ubiquitous today) are OpenMP and Pthreads. Results from comparing the threading models are in section 5.2.

4.2.1 POSIX Threads (Pthreads)

Pthreads or POSIX threads is a Unix standard for creating threads on most flavours of Unix based Operating Systems including Linux and Mac OS X. Programmers can use Pthreads to create, manage and manipulate threads with synchronisation primitives like `wait`, `signal` and `mutexes`. It has been used for many years on the Unix platform and with the advent of Mult-core x86 Unix systems, porting a program to exploit the parallelism available using pthreads is readily convenient.

Pthreads is also the original threading model used by the authors of SimIt-ARM to do parallel simulation on one desktop node.

4.2.2 OpenMP

The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

4.3 Improving Quality of Translated Code

In SimIt-ARM, target code is initially interpreted by an interpretive simulator. As the interpreter runs for a while, commonly executed code is earmarked for translation from ARM Binary to a C++ function, which is then compiled to a shared library object of the host platform. This translation process in the original implementation of SimIt-ARM-3.0 is making use of `g++` as its intermediate translation engine.

It is hypothesized that there is not a lot of time spent in translation since it takes 0.9 seconds to translate a DLL [10] but this largely depends on the nature, size and the locality of the program. Thus one of the opportunity that arises from the use of multi-procesing is that time can be spent to improve the quality of the code in parallel since the locality of programs can mostly be deterministic (i.e. 80-90% of the code is spent executing 10-20% of the code).

There are 3 main experiments that are conducted. First, the translated (or cached) blocks are examined for stitching opportunities to improve simulation performance. Next, various optimisation level passes shall be performed on translated code and finally more optimisations can be done at runtime to the already translated code to improve code quality when slave processors are idle. The results of these experiments are discussed in section 5.3.

4.3.1 Stitching Translated Blocks

As blocks of program code are translated, they can form disjoint portions of the original program. Since SimIt-ARM uses a cache that is directly mapped, the process of running code from the cache can be improved slightly. Figure 4.1 shows how cached code blocks can be linked to reduce the overhead of frequently jumping from the main interpretive simulation mode to compiled simulation mode and vice-versa.

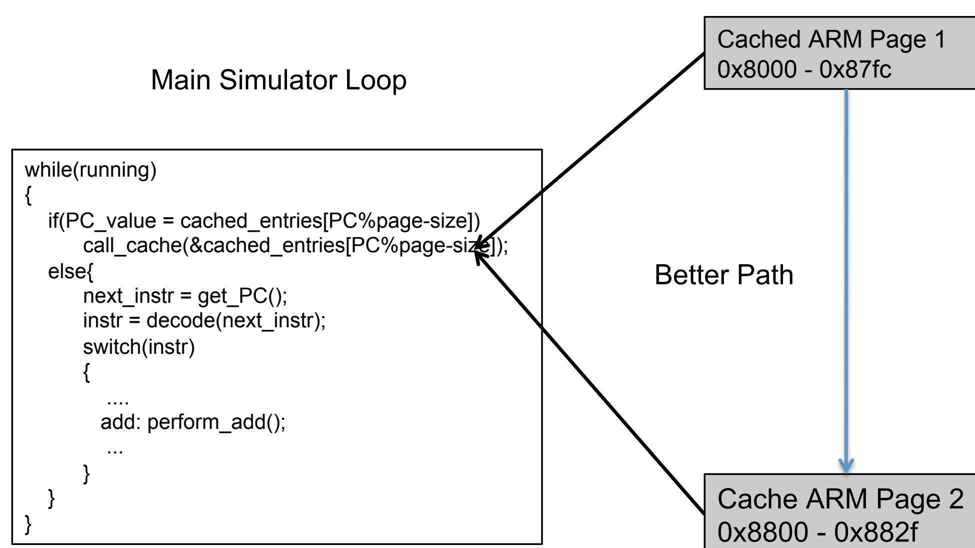


Figure 4.1: Disjoint translated blocks

Two main experiments are conducted namely:

- How many times does the simulator switch from compiled simulation to interpretive simulation?
- Increasing the size of the translation unit for user-mode emulation

The results of stitching translated blocks are described in section 5.3.3.

4.3.2 Using Higher Optimisation Levels

As an optimising compiler, `g++` has mainly 3 levels of optimisation available. Namely `-O1`, `-O2` and `-O3`. Some optimisation passes in these options are useful to an ISS; especially those related to the code generation phase. They are briefly described below.

- **-O1** - The useful optimisation passes at this level are: `-finline-small-functions` and `-fomit-frame-pointer`. There are many small functions in SimIt-ARM's simulator engine. Inlining them would reduce the amount of host assembly code performing call setup clean up and linkup for only a small, short function call. The latter pass, `-fomit-frame-pointer` however has to be flagged manually as `g++` do not use it by default on x86 hosts.

- **-O2** - The `-fpeephole2` optimisation from `-O2` can do more optimisation on small units of host instructions. This can be useful for smaller block sizes like in System-level simulation.

- **-O3** - This switch turns on many optimisations that increase code size and compile time. One of the useful optimisations are `-fpredictive-commoning` that reuses computations like memory loads and stores performed in previous iterations of loops. Compiling at `-O3` also inlines larger functions. The flag `-finline-limit` is also set to 5000 so that a function simulating a block of translated code can fit in its inlining heuristic.

There are also many machine specific options such as `-mtune` and `-march` that tunes code to specific CPU types i.e. SSE2 instructions for Pentium(R) 4 processors.

As an initial experimentation effort by the designers of SimIt-ARM, it was assumed that many blocks will be translated and they will have to be translated as quickly as possible. Hence the initial design decision in SimIt-ARM [10] was to cap the translation effort to level `-O1` in `g++`. This ensures that the code translated will be fairly efficient and translation time is at the same time minimised.

On the other hand, code that is heavily utilised can be translated at a higher optimisation level since most of the time, the program will run disproportionately at a few specific regions. Hence, a few more experiments are proposed:

- Identify amount of time spent in translation

- Translation of target code in parallel using `-O2` and `-O3` on `g++`
- Use no optimisations

The results of using high Optimisation levels are revealed in section 5.3.1.

4.3.3 Idle-Time re-optimisation

In SimIt-ARM, target code is first compiled with minimal optimisations so that translation/compilation speed is fast. In idle periods where some threads sleep, a timer interrupt checks if the idle period has exceed the specified time. If it does, it wakes up the threads to perform re-optimisation of the blocks at a much higher optimisation level. The re-compiled DLL is then reloaded when the main thread is not executing the freshly re-optimised block. Figure 5.7 illustrates the process of idle-time re-optimisation.

Optimising the cache during idle periods on slave processors is challenging. This is mainly due to 2 factors:

1. When to do re-optimisation
2. How many blocks to re-optimise at one time.

If a new block in the target program is identified for translation when a slave thread is currently performing a re-optimisation task, the main simulation thread has to perform more interpretation on the newly idenfitied block. As it is a frequently executed block, the performance is affected. This delay is called the *missed opportunity cost*. To address this problem, a timer is placed to check if the slave threads have been idle for a specified interval. This interval of idle period has to be chosen carefully, if it is too long, very few blocks will be reoptimised, if it is too short, it will stand a higher chance of clashing with the arrival of tasks to translate newly identified blocks.

To help choose the appropriate interval, the following formula is used to compile code at a dynamic idle period and back-off once a new DLL has be compiled or an old DLL has been re-optimised:

$$idle_thres = \frac{2.0}{idle_interval + 0.1} \times 5.0$$

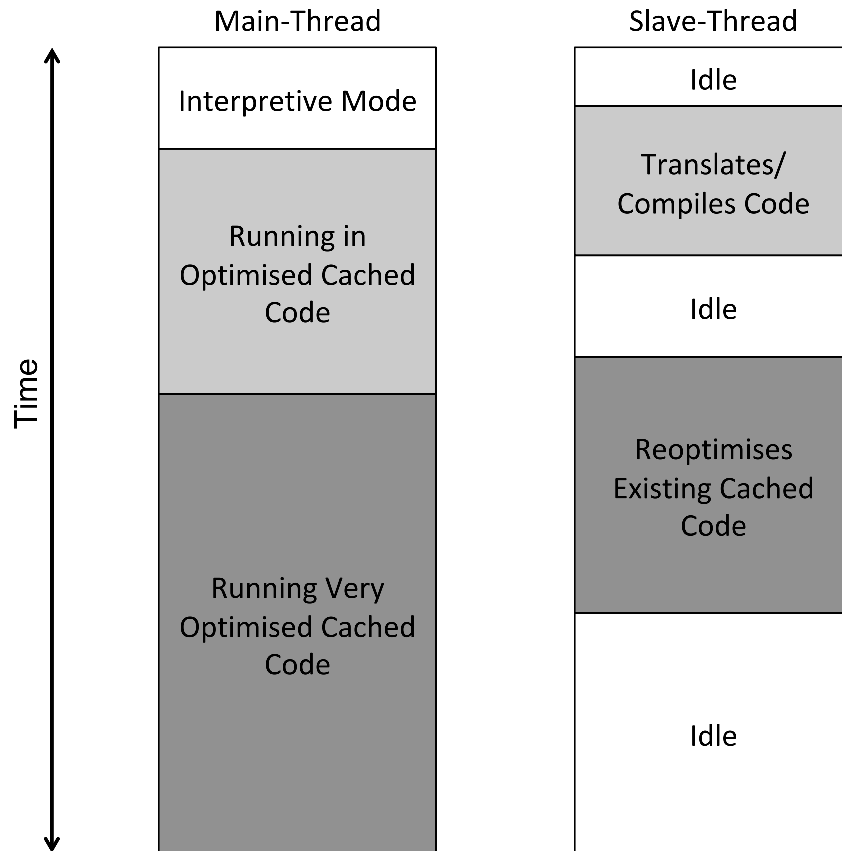


Figure 4.2: Idle time Re-Optimisation

Where *idle_thres* is the idle interval threshold in seconds to issue the re-optimisation wake up signal for the sleeping threads and *idle_interval* is the real amount of time (also in seconds) the slave processors have been idle. The *idle_interval* is feedbacked into the formula to re-adjust the *idle_thres*. As the *idle_interval* grows, the *idle_thres* decreases.

The resulting high level code for the body of a slave thread is shown in figure 1 in the appendix. The results of idle-time re-optimisation is shown in section 4.3.3.

4.4 Improving Locality of Translations

Doing compilation as a form of translation style seems to incur huge I/O costs during simulation. Various ways are proposed to reduce the cost of I/O in compiled simulation.

4.4.1 RAM Disks

Most flavours of Unixes including Linux have a RAM-Disk partition known as `/tmp` or `/dev/shm/`. These portions of the space is actually used by the operating system to store temporary files and user program variables. It is postulated that translating and storing the shared library objects on the RAM-Disk would yield faster lookup and cache writing. The results of using a RAM-Disk is shown in section 5.4.2

4.4.2 Producing Code Images

In the first instance, the aim is to replace the DLL caching scheme with a direct dynamic code generation method into SimIt-ARM. This means changing the part that creates the DLL and the part that loads the DLL. The outcome should be that the simulator ends up running the same code with less overhead than with a DLL. More about the outcomes and problems of this experiment is discussed in section 5.4.1.

4.5 Reducing Time Spent in Interpretation

As described in the section 2.3, the performance of Interpretive simulation is slow. Since both interpretive and compiled simulation schemes are used in this simulator, ideally, it should spend most of the time running cached instructions. However, this is currently not really the case. A few experiments are conducted to see if its possible to reduce the amount of time spent interpreting the instructions.

4.5.1 Speculative Translation

This problem may be solved with a predictive heuristic that speculatively and proactively translates the blocks of target code hopefully before it is executed. Successful predictions can further reduce the number of interpreted instructions.

To determine the potential benefits of doing the translation predictively, a 100% accurate prediction, negligible-time prediction algorithm is mocked up. The procedure is as follows, a benchmark is run normally to discover which blocks are being used in the run. After that, the identified blocks are hard-coded to be pushed onto the queue at the start of the simulation for slave threads to begin their work. The same procedure is repeated for each of the five benchmarks.

There are two ways in which cached blocks are being identified. To simulate a *conservative* heuristic, the threshold can be set to the default value of 16 million so that the mock-up prediction is more careful its selection (fewer but very heavily executed blocks will be identified). A more *aggressive* heuristic can simulated by using a lower threshold so more blocks will be identified and translated.

Discussion on the results of speculative translation is available in section 5.5.2.

4.5.2 Varying Translation Threshold

Currently, the simulator utilises a simple block history heuristic to detect most frequently executed instruction blocks for translation. The threshold is set by the user in values to the power of 2. For system emulation this value is set at a boundary of 2^{16} instructions and for user-mode emulation, 2^{32}. A lower threshold value can mean that more blocks can be selected for compilation/translation and can affect simulation speed. However, more time can be spent translating less-frequently executed blocks and very hot-code may be given less attention. A higher threshold value can mean translation focused on frequently executed code. On the other hand, there is a missed opportunity cost can be incurred through interpreting the block of instruction for too long before it is cached.

Higher translation threshold values can also mean higher utilisation for slave processors. The threshold values being tested are much lower than the default 16m. The interesting values in the original paper [10] are 1m and 2m so these two values are tested. Not all threshold values are tested as it will create more combinations and more time is needed to test all of them.

Results on varying translation threshold are discussed in section 5.5.1.

4.6 Test Setup

SimIt-ARM-3.0 is configured, build and installed and run on a Desktop with an Intel(R) Core(TM)2 Quad CPU Q8200 running at 2.33GHz. The Desktop runs Ubuntu Linux 8.10 with version 2.6.27-11 of the kernel and Native POSIX Threads Library (NPTL).

The simulator is compiled using gcc-4.2 with `-O3 -fomit-frame-pointer -finline-limit=5000`. This is so that the compiler inlines functions sufficiently large for many redundant host assembly call setups and cleanups to be eliminated. Switching on `-fomit-frame-pointer` also removes the stack frame pointers that is only necessary for debugging but will reduce simulation speed.

During the dynamic compiled simulation process, the `g++` flag, `-pipe` is also used so all data in the intermediate compilation process is done in memory.

The simulator is mainly tested with 5 different SPEC INT 2006 benchmarks [16]. For each of the benchmarks, the first reference input provided by SPEC are used. The test input is sometimes also used as experiments for various short program length tests since the reference inputs take a long time to execute. As there are possibly many tests combinations to be run; not all of these combinations are tested. It is also because tests have to be run at least 3 times and the results collected averaged with the geometric mean. The benchmarks are selected based on real world user space usage of a typically complex mobile electronic devices today. The benchmarks are:

- **473.astar** - an A*Star search algorithm for 2D-maps to simulate a program for GPS devices.
- **401.bzip2** - a file compression tool from Julian Seward version 1.0.3, modified to do most work in memory. Simulates opening and making archives on a smart-phone.
- **455.gobmk** - a board game of Go, a simply described but deeply complex game.
- **464.h264** - A reference implementation of H.264/AVC. Mocks up video encoding for digital, video cameras and also smart-devices.
- **458.sjeng** - A highly-ranked chess program that also plays several chess variants.

On top of these benchmarks, some smaller benchmarks are also used to test quick experimental modifications to SimIt-ARM. They are mentioned along-side with the experimental results.

4.7 Metrics

To measure the performance of the simulator, there are more variables to consider than just the wall-clock time. Therefore, a few more parameters can sometimes be considered, they are:

- Utilisation - also expressed in %, this is usually available on most Unix systems' `time` command. A higher utilisation leads to better multi-processing justification.
- Translation Delay - denoted $t(b)$ expressed in seconds is the time from which a block is first interpreted to the time the block is cached.

The *average translation delay* denoted α is an important effectiveness metric for translation schemes (i.e. translation threshold and proactive compilation strategies), it is a measure of how much time on average does a block spend in interpretive mode. Let b be a block and C be the cache lookup table where $\{b : b \in C\}$, the α for a run is calculated by:

$$\alpha = \frac{\sum_{\forall b \in C} t(b)}{|C|}$$

In general, a lower value of α means code is identified and cached early. Lower α values does not mean better performance as fast translation may occur at the cost of optimisation.

Chapter 5

Results and Discussion

Unless specifically stated, all runs are made with the first reference input of the SPEC CINT 2006 benchmarks.

5.1 Original Simulator Results

SimIt-ARM is ran with the default settings to measure overall performance and speed-up.

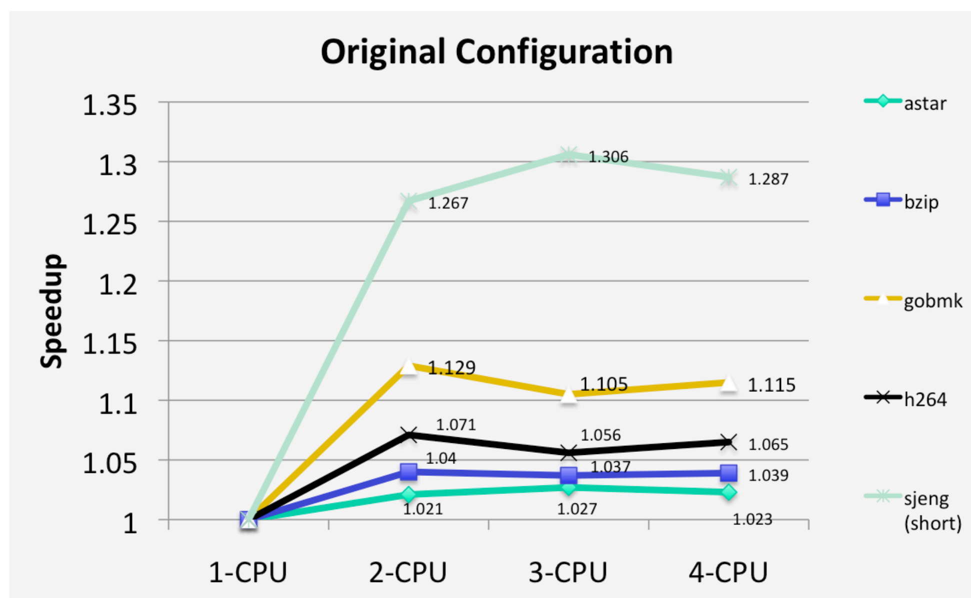


Figure 5.1: Speedup in original configuration

Chip-based Multi-processing offers significant improvements. As shown from figure 5.1, the use of more cores however, does not guarantee further improvements. Note that the

main purpose of multiprocessing in SimIt is to do the most labourous part of simulation (which is translation) along with program execution to reduce dynamic-compiled simulation overhead. Therefore, speedup is proportional to the number of blocks being translated.

Speedup achieved also depends on the length of a program; it varies between 1.01 for long (3 hour) programs and 1.4 for a 100-second short programs (using reference and test workloads of the `sjeng` benchmark respectively) depending on the total run-time of the target program. Note that the test workload of the `sjeng` benchmark is used in figure 5.1 as an indication of a case with higher speedup.

5.2 Results on different threading models

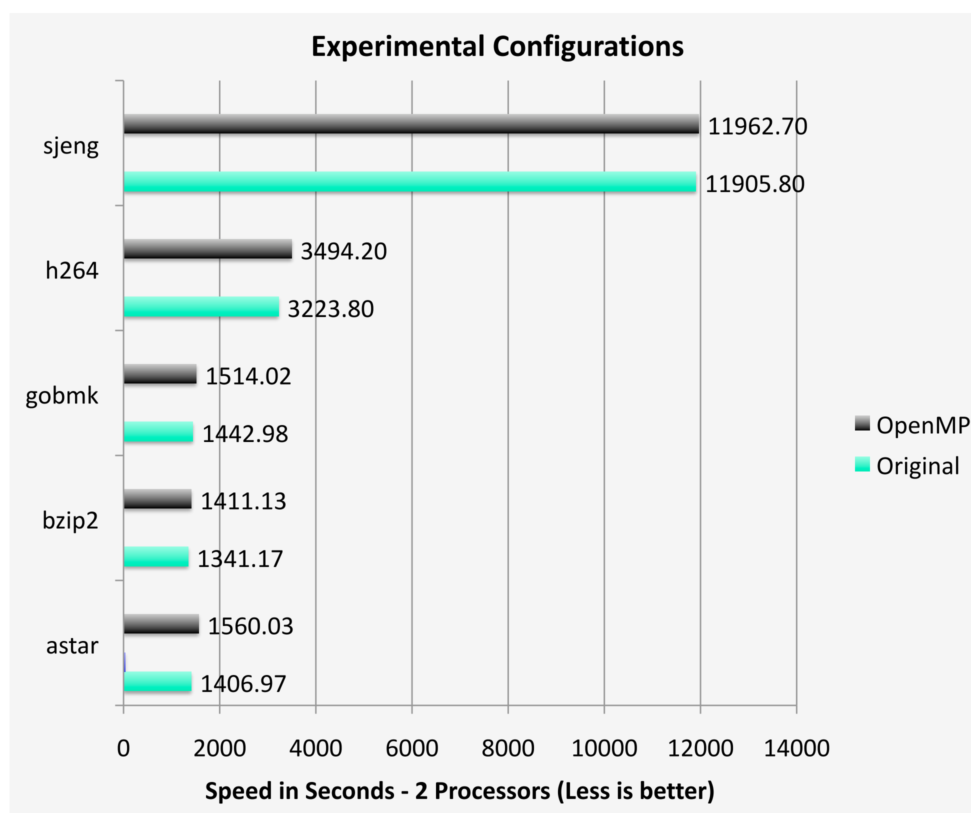


Figure 5.2: Open MP with 2 processors

OpenMP, has no wait and signal primitives; threads have to poll to check if there is translation work one at a time (see figure 5.3). Comparatively, Pthreads have a func-

tion called `pthread_cond_wait` that makes it more suitable for the task at hand. So far OpenMP is perhaps best used in algorithms with which the amount of unit of load is known in advance. With the dynamic nature of code translation, units of work arrive at stochastic intervals.

As busy wait wastes processor cycles in an environment with only 2-4 PEs, the use of OpenMP results in poorer performance (refer to figure 5.2) . Busy waiting is only useful where there are a lot of other computation tasks for many processors. However, there is only one queue (or pool of tasks) at any one time. Without the primitive contruct `wait` that allows threads to sleep while waiting for the queue to be non-empty, the problem become accentuated.

The effects of False sharing is also magnified in this scenario. In this simulator, threads share many variables that might be stored in different cache lines on the host system. The effect is not propagated when threads sleep when there are no translation tasks.

When programming Pthreads in a busy-wait fashion, the same slowdown effect is also replicated. Figure 2 and 3 in the appendix shows the pthread and OpenMP implementation of the slave thread code bodies for performing translation tasks respectively.

It is recommended that the OpenMP team should make available a few more critical constructs `wait` and `signal` to software engineers so that programs that deal with real-time work load arrival and dispatch is more smoothly dealt with on machines with less processors. Unfortunately, it is not available yet at the current upcoming version 3.0.

```
void get_block_to_compile(unsigned *dll, unsigned *pblk)
{
      //PID_LIST are translation requests from the main thread
      omp_set_lock(&pid_mut);
      if (PID_LIST->empty() && !running)
      {
           omp_unset_lock(&pid_mut);
           return false;
      }

     *pblk = PID_LIST->front();
       PID_LIST->pop_front();
     *dll = dll_ind++;
      omp_unset_lock(&pid_mut);
      return true;
}
```

Figure 5.3: OpenMP version of translation job queue, note the absence of the `wait` construct

On the other hand, algorithms with simple work sharing schemes often perform much better in OpenMP and are also very easy to code. A classic simple example is one with the following program that calculates an approximation of the value of pi π using the following equation with $N = 730000000$:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx = \sum_{i=0}^{N-1} \frac{4}{1+(\frac{i+0.5}{N})^2} \times \frac{1}{N}$$

Figure 4 and 5 in the appendix shows the OpenMP and Pthreads' pi calculation program respectively.

# Procs.	OpenMP	Pthreads
2	3.47s	6.513s
3	2.35s	5.071s
4	1.81s	4.721s

Table 5.1: Performance of OpenMP and Pthreads doing coarse grained π computation

In short, OpenMP works well at a fairly coarse-grained level (see table 5.1). It is masterly in its ability to perform data decomposition on loops, assigning tasks to individual threads, and other high-level operations. However, the design of an Instruction-Set Simulator needs to perform fine grained control of threads making OpenMP is less suitable [13] than native API sets like Pthreads.

In a different simulation approach such as with predictive translation, OpenMP will be able to know the workload at program start. The size of an executable is usually fixed unless there is self-modifying code. Hence it would be interesting to see how the OpenMP model will perform when placed into such contexts of simulation.

5.3 Results with optimising code quality

Doing translation without optimisation results in poorer performance. A test workload of the 429.mcf benchmark takes 27 seconds to run in `-O` c.f. 83 seconds without any optimisation. In general, optimising code in parallel pays off; as the overhead in translation is moved to another processor, more powerful optimisation passes can be used without affecting performance too much. This section shows the results of the various experiments on optimising the translated code.

5.3.1 Results on translating at higher Optimisations

To reveal the costs of compiled translation at high optimisation levels, as mentioned in section 4.3.2, the amount of time spent on translation is first investigated.

Amount of time spent on translation

As shown in figure 5.4, much of the time is spent in executing the translated code. Each translation takes about 1.2s on `-O` and 2.9s `-O3` on average; hence in the time spent trans-

```
...ng@qcore1: ~/benchmarks/h264_ref — ssh — 80x24
[a1148903@uss ~/SimIt-ARM-3.0]$ cd ../SimIt-ARM-3.0_ORIGINAL/build/bin/
[a1148903@uss bin]$ bash
bash-3.2$ cat out
out                          output_h264reopt.err
output_en_reopt.err          output.out
output.err                   output_reopt.err
output_gobmk_gobmkreopt.err  output_sjeng_en2.err
output_go.err
bash-3.2$ cat output_h264reopt.err
Total user time  : 660.734 sec.
Total system time: 4.053 sec.
Simulation user time  : 574.521 sec.
Simulation system time: 0.273 sec.
Caching user time  : 86.214 sec.
Caching system time: 3.780 sec.
Simulation speed : 3.511e+08 inst/sec.
Total instructions : 233377997308 (233G), including 99.25% compiled
Requested blocks  : 75.
Cache-hit count : 0.
Connected servers/threads : 2 of 2.
Compilation count :
  thread 0: 38
  thread 1: 37
bash-3.2$
```

Figure 5.4: Ratio of time in translation to code execution for the (test) h264 benchmark

lating in the h264 test load benchmark shown in figure 5.4 is only about 90 seconds if done in `-O` and 142.5 seconds in `-O3`. Note that, the longest amount of time spent in translation with respect to overall runtime is the h264 benchmark because of the poor locality of instruction execution.

From this, the proportion of time spent in translation depends on a few parameters:

- Threshold value - A lower threshold value, means more time is spent in translating
- Optimisations - As mentioned earlier, doing higher levels of optimisation not only increases code size but also gives longer compilation times.
- Locality of the target program - If the program spends most of the time executing only 10% of the code, only a few blocks need translation. If this is the case, optimising that 10% will provide a disproportionate amount of improvement.

Adding more optimisations during compilation will further increase the translation time and add on to α, the translation delay.

Note that all these benchmarks execute for at close to at least 20mins. For all these runs, higher optimisations seem like a better value (refer to figure 5.5), on the other hand,

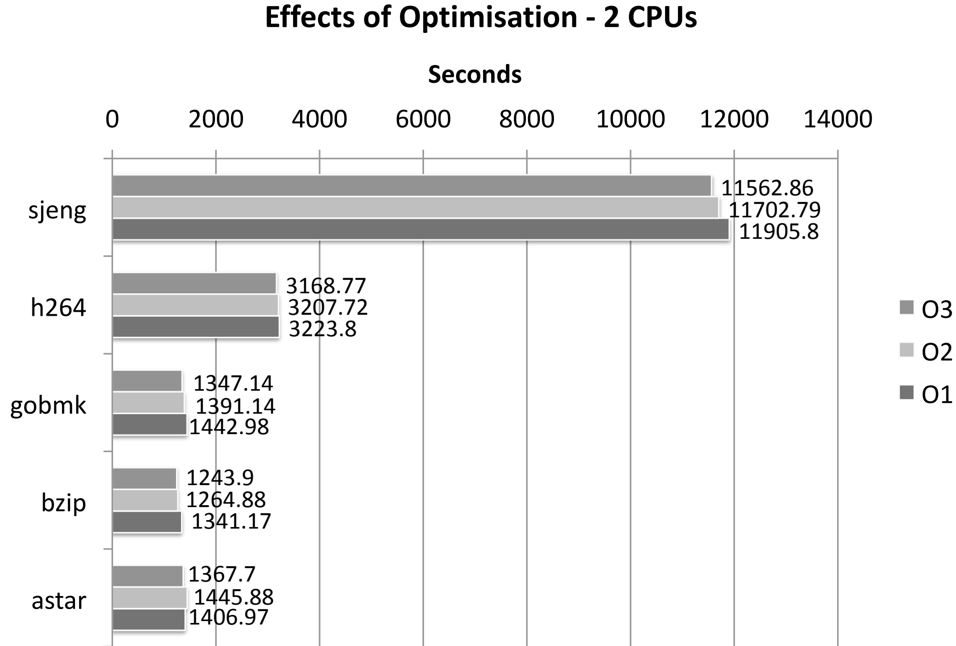


Figure 5.5: Translating with 2 processors at various optimisation levels

for short programs that only run for 1-2 mins, the optimisation cost outweighs the benefits. For e.g. in a parallel (2 processor) run of the bzip2 test workload, it took 73 seconds with `-O` and 118 seconds with `-O3`. This speed imbalance can be reduced by using more processors and lowering the translation threshold so that more instructions can be translated and simulation spends more time in translated code. At figure 5.6, doing more optimisation at a translation threshold of 1m shows little speed penalty with 4 processors. Also, with multi-processing, the effort for function inlining in `-O3` starts to pay off compared to `O2`. Therefore, translating at higher optimisations appears to be useful for future platforms with more cores/CPUs.

More results on tuning the translation threshold are discussed in section 5.5.1.

5.3.2 Results on Idle-Time optimisation

Although re-optimisation promises to bestow a trade-off between similar translation overhead for short programs and better performance for longer running programs, a more counter-intuitive outcome is observed. Figure 5.7 shows the average potential gain in performance by executing re-optimised code at `-O3` with respect to the total amount of code translated.

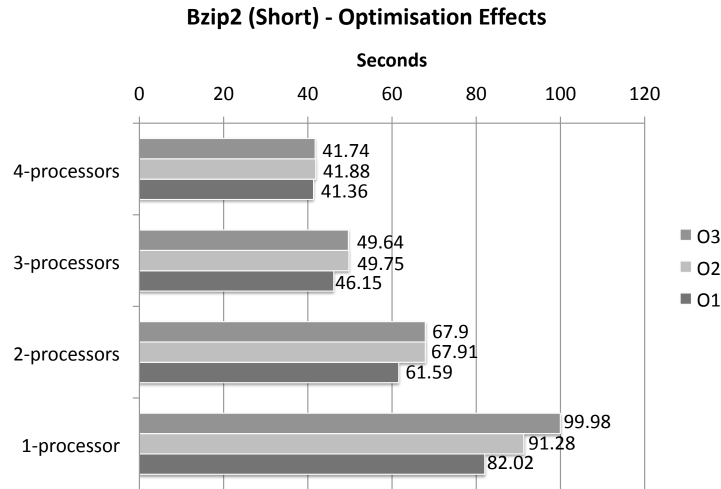


Figure 5.6: Effects of Optimisation on short runs after lowering translation threshold to 1m

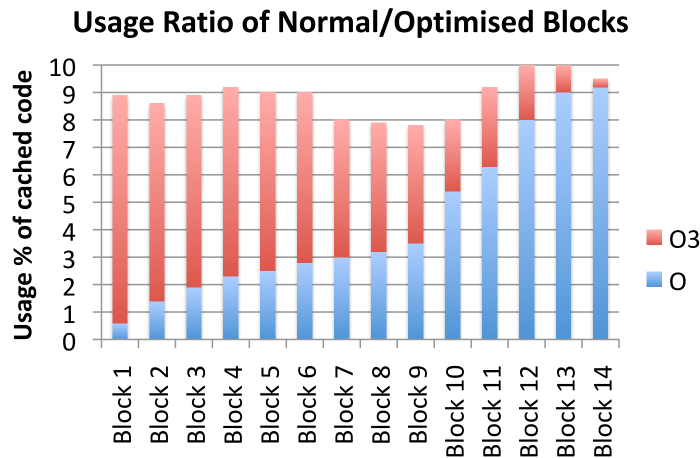


Figure 5.7: Expected gain from Idle-time Re-optimisation

The graph pictured is just a rough guide but reflects possible gain by spending 50% of its time executing in re-optimised code (although it depends on how much code is re-optimised on the whole). Consider an example where the bzip2 benchmark in figure 5.5 is ran this way. Observe that there was only a 7.11% of gain from the benchmark where 99% of the code is executed from the cached code optimised at the beginning with `-O3`. Therefore, if 50% of the time is spent executing reoptimised code. There will be about

3.56% improvement in performance. So in practice, it should provide slightly better performance for longer programs and similar performance (compared to default settings) for short programs; this is also largely dependent on the time-slot where the re-optimisation is performed.

Unfortunately, implementation is more challenging than anticipated. The modified code is unusable as a re-optimised block is often reloaded at the same time it is called by the main simulator thread. To solve this race-condition, the obvious solution seems to use a mutex lock on pointers to the DLL. Unfortunately, locking the pointer with a mutex results in slower performance as the mutex has to be locked and unlocked for each execution of a cached block.

Further study is needed to examine safe ways of reloading a DLL (possibly) in use before the trade-off can have a chance at being realised.

5.3.3 Results on Block-Stitching

In a short benchmark (429.mcf) that ran for 27 seconds, the amount of switches from interpretive to compiled simulation exceeded a million. These additional instructions executed per mode switch to do call-setup, link-up and clean up on the host machine's stack to call the translated C++ functions add significant overhead.

Therefore, block stitching could provide potential improvement to overall simulation speed. As larger blocks are being translated, more scope is available for optimisations by the compiler. Basic blocks spanning 2 blocks can also be quickly executed. The simulator would be doing less jump returns to the main execution loop before going to the next cached block.

SimIt-ARM is designed to align blocks at 2KB boundaries so it could not run many benchmarks properly with a block size of 2^{10}. For one benchmark (445.gobmk) that worked the benefits are shown in table 5.2:

It shows that if permitted to run with larger block sizes, performance would potentially be better. However, the translation delay is also increased as it takes more time to compile a block twice as big.

Configuration	Performance (s)	α Delay
512 word	6.73	3.82
1024 word	8.52	4.79s
512 word (persistent)	3.80	n/a
1024 word (persistent)	3.66	n/a

Table 5.2: Block stitching effects on first and subsequent runs

Since an optimisation level of -O3 inlines functions at our overwritten `-finline-limit=5000`, the translated function fits into the compiler inlining heuristic's overwritten prescribed limit.

5.4 Results of Locality

There are two experiments done to improve the locality of translations; namely the use of a RAM disk and the use of Code-Images. Only the former has results. The other one did not produce runnable code for the simulator. This section describes the outcomes and problems.

5.4.1 Producing Object Code images

In section 5.4.2, the aim is to replace the DLL caching scheme with a direct dynamic code generation method into SimIt-ARM; in the hope that changing the part that creates the DLL and the part that loads the DLL would remove the overhead of loading and creating DLLs hence decreasing the overhead of translation and improving performance.

Shown in figure 5.8 is a program that as a start creates the equivalent of a simple function in the x86 architecture that returns an integer value 42. It is a simple example on trying to generate assembly, store it in an image, then load and run it. The same principle is used to translate ARM target instructions and generate the equivalent x86 machine code at runtime and execute it. Figure 5.9 shows how the code image in figure 5.8 is created and stored in a .bin file.

```
        int func(void) { return 42; }

x86 assembly:
--------------------------------------------
        00000000 <_func>:
0: 55                    push   %ebp
1: 89 e5                 mov    %esp,%ebp
3: b8 2a 00 00 00        mov    $0x2a,%eax
8: 5d                    pop    %ebp
9: c3                    ret
```

Figure 5.8: Simple function that returns an integer

A code example to use the image created in figure 5.9 is shown in Figure 6 at the appendix. The program works fine without linking and resolving symbols by relocation. However, for the complex, tightly coupled nature of the translated functions in SimIt-ARM, it is not a straightforward task to resolve the symbols programatically. Doing so would just be the same as doing dynamic linking, hence the introduction of DLLs or shared libraries. By just running the function in pure object file format, a segmentation fault occurs.

```
1) od -Ad -x 42.o | less

outputs:
...
0000048 0009 0006 8955 b8e5 002a 0000 c35d 0000
...

2) cat 42.o | head -c 62 | tail -c 10 | od -Ad -x | less

outputs:

0000000 8955 b8e5 002a 0000 c35d
0000010

3)cat 42.o | head -c 62 | tail -c 10 > image.bin
```

Figure 5.9: Simple function image that returns an integer

5.4.2 Results of using a RAM-Disk

Usually RAM-Disks are made available on various Linux or Unix based machines to keep temporary files and variables. An effort has been made to use it to reduce the overhead of file I/O.

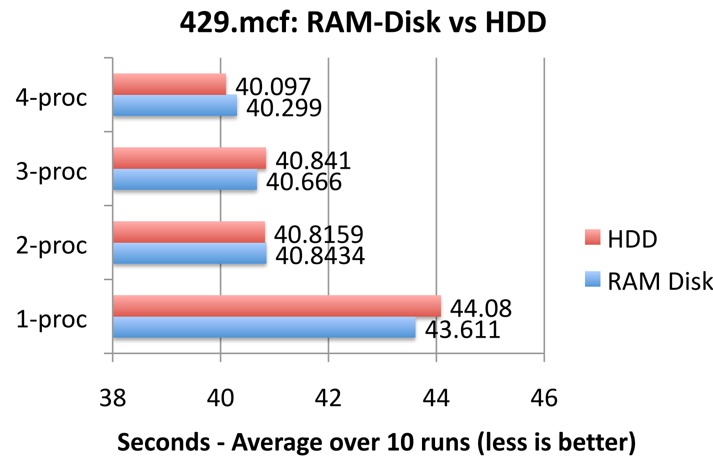


Figure 5.10: Use of a RAM-Disk

As shown in figure 5.10, the results show no difference in the speed of the simulation. The counter-intuitive observation is basically based on the fact that most Unix/Linux systems (or even most Windows systems) have a buffer cache that does a write-back to the disk at idle periods. A Write-back is more efficient than write-through (which immediately writes to the destination disk). On the other hand, using a RAM-DIsk is also more prone to errors. For instance, if the machine crashes, or the power is cut at a bad moment, or the floppy is removed from the disk drive before the data in the cache waiting to be written gets written, the changes in the cache are usually lost.

This is also the main reason why the power should not be abruptly switched off without using a proper shutdown procedure, or remove a floppy disk from the disk drive until it has been unmounted. These systems of file-buffering reduces the read/write times to disk for many situations that are not limited to just these examples:

- Results from a Relational Database Management System (RDBMS)

- Files on a Dynamic Servlet/CGI form in a Web Server
- Running a program just compiled from source, e.g. a program for a programming assignment
- Freshly downloaded files or executables

Hence the phenomenon occurs in the case of producing a C++ function that is immediately linked to the simulator engine after compilation. The buffer cache is also automatically managed by the operating system. Neither intervention is required nor special settings are needed to use it. Furthermore, the size of the buffer cache is typically the remaining amount of main memory on the system. The size of the buffer cache is automatically reduced when processes need more memory.

Before the project started, it is feared that many multi-core PCs today have 2-4 cores on chip but only 1 hard-disk. Hence the write/reads to the DLL will be serialised. Fortunately there is the file-buffer cache and it is Concurrently accessible so further speedups in parallel are actually possible.

5.5 Results on doing less interpretation

On the whole, the interpretative simulation runs more than 4.5 times slower running cached code in compiled simulation. Table 5.3 shows the MIPS observed for Interpretive and cached code on all benchmarks ran. This section shows the results of spending as little time as possible in interpretive mode.

Simulation Mode	**Average Speed (MIPS)**
Interpretive	52.3
Persistent-Cached	238.16

Table 5.3: Magnitude of improvements in Persistent caching vs. total interpretive simulation

5.5.1 Reducing Translation Threshold

Reducing the translation threshold means increasing the number of instructions in the program that will be selected for translation. This has the effect on performance and

utilisation. Overall chip multiprocessing benefits from lower threshold values along with better optimisation levels. In the original paper [10], a value of 1m to 2m as the threshold is the best for 4-6 processors on a cluster; similarly, for a chip-multiprocessor with 4 cores, it is indeed no different. Figures 7-14 in the appendix show the results of the translation threshold experiments. The reader is encouraged to compare the results against the speculative translation results in figure 5.11.

The 445.gobmk benchmark executed normally but thrashes at the end of the run at a translation threshold of 1m. Therefore, its results for 1m thresholds were omitted. Translation thresholds below 1m also causes more thrashing [1] upon finishing the run in some other benchmark programs. While it is running, more time is spent translating and lower emulation performance is obtained.

Despite that, results still produce the fact that if the threshold value is too low (i.e. at 1m), it increases α and will affect emulation speed on shorter programs being simulated with higher optimisation levels, involves many cached blocks and use fewer processors.

Lowering translation threshold and increasing the optimisation levels suits machines with more processor cores on chip. This is because there is more compilation and optimisation work can be done in parallel, thereby increasing simulation performance. Lowering the translation threshold brings the values much closer to the 'modest speculative translation' technique (cross refer to figure 5.11).

5.5.2 Results of speculative translation

This section presents the results of mocking up an ideal-case prediction strategy, i.e. one that takes a negligible time to execute in parallel and detects with 100% accuracy the most frequently executed blocks.

[1] The thrashing phenomenon is totally unexpected as the ARM target instruction block sizes are aligned at a fixed boundary. At a threshold of 1 (in other words, translate all executed code), the target program thrashed, froze and the amount of blocks translated is continuing to grow far beyond the size of the target program. The original paper did not report similar issues with threshold values of 1m. That can mean a defect in the simulator that requires careful examination of the block to physical-page interaction. SimIt-ARM uses a 512-word code region scheme from by [19], which in the author's opinion is not suitable to be used dynamically since code is often mixed with data in physical memory. This could also be the reason why SimIt-ARM can only execute statically linked programs. The issue still requires further investigation.

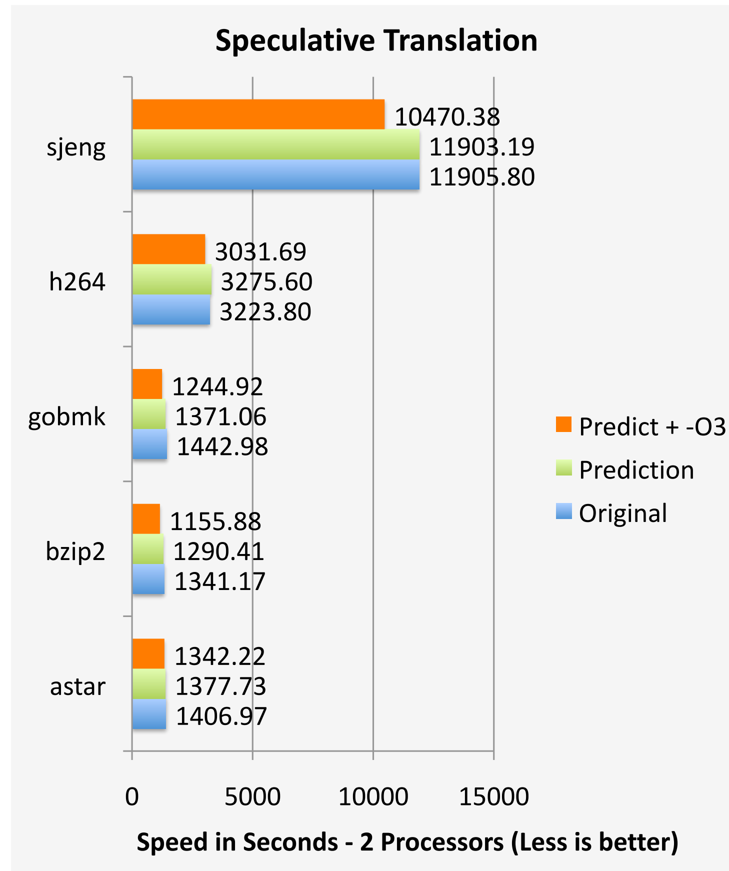


Figure 5.11: Potential benefits of speculative translation

Speculative translation on the whole usually gives a significant improvement in simulation performance. The only exception is the h264 benchmark. This is due to the video encoder's poor locality. Each block of code takes a long time to be executed 16m times. There are blocks which are slightly less frequently executed being hit more times before a particular block is executed 16 million times. In short, the working set of the h264 program is large as there are many frames in flight.

Hence, when the translation heuristic is conservative in its selection of blocks, the performance is affected. The default 16 million threshold is used to mock up a cautious speculation heuristic (refer to section 4.5.1 on how mocking up is done) and figure 5.11 shows the general results of doing the cautious prediction strategy. For the case of h264, when a more generous predictive heuristic (mocked up using a 1m threshold) is used, the number of blocks identified increases only by 15%, but the ordering of blocks changed

dramatically. Upon reducing the translation threshold to mock-up the heuristic, the performance is thence improved.

Opt. Level	**1m - Threshold**	**1m - Speculative**
-O	3329.31s	3275.6s
-O3	3076.35s	3079.79s

Table 5.4: Results of H264 on 2-cores with generous speculative or low threshold modes

The other way to prevent code usage ordering anomalies is to optimise the translated code. Shown in figure 5.11, using optimisations on the same conservative heuristic improves performance of the h264 video encoding benchmark.

Another interesting observation in table 5.4 that generous speculative prediction on the h264 benchmark do not perform better than just lowering the threshold. This is due to the fact that less urgent code that has to be translated is given an unfairly more attention compared to code that is really frequently used. It would be, as a interesting further study, examine the generous heuristic on more benchmarks.

Optimisations on a speculative heuristic

For all benchmarks, a speculative heuristic benefits from the use of higher optimisations. This is due to the fact that spare processors handle the work in parallel and more time is spent in executing optimised code.

However, when the generosity of the heuristic is increased with the optimisation levels. It does not necessarily improve the performance. This is demonstrated again in the h264 benchmark (see table 5.4) as truly hot-code is not given the necessary attention to be translated first, causing a slight performance penalty from missed opportunity cost.

Importance of ordering in speculative translation

When a heuristic is mocked up as mentioned in section 4.5.1, blocks are pushed onto the queue according to the order in when they are cached. When ordered using the block's location in the virtual address space, the performance may not be ideal. Hence, when

designing such a real predictive heuristic, it is important to do some profiling. An example of such a possible profiling heuristic is seen in [17].

5.6 Other Experiments

Some miscellaneous experiment results useful to product engineers are highlighted below.

5.6.1 Persistent Caching

Persistence caching is a useful feature for product engineers or software programmers writing the system software for a new platform. As code can sometimes be altered during debug cycles, it is useful not to flush the cache as only some pieces of code have been changed or added.

In the following use cases on the test workload of the SPEC INT combinatorial optimisation (429.mcf) benchmark, SimIt-ARM performed the following:

Code Position Changed	Block Invalidations
Start	changed 1 block
Middle	changed 2 blocks
End	changed whole program

Table 5.5: Persistent caching and its effects on small changes on code debug cycles

Sometimes, changes to code involves adding more code, hence the effects of adding more code on the persistent caching mechanism is also tested. Table 5.6 shows the effects:

Code Position Added	Block Invalidations
Start	changed all
Middle	changed 1
End	changed 1

Table 5.6: Persistent caching and its effects on adding code in debug cycles

Note that in 5.6 changing code in the middle of the program can invalidate more than

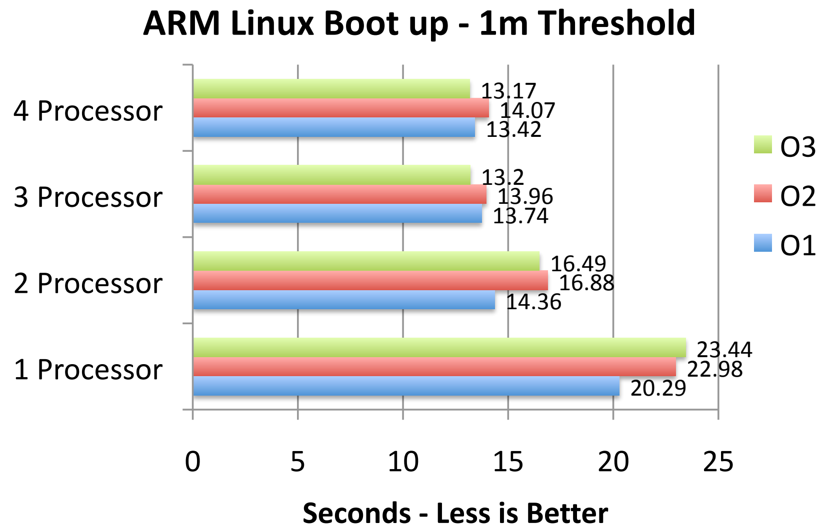
1 block of translated code in the cache. This depends on the number of instructions previously in the modified block (a block usually has < 512 instructions in it).

5.6.2 Persistent Caching benefits

The current implementation of Dynamic compiled simulation and persistent caching at the disk level demonstrates the capability of reducing more translations happening in short debug cycles. As code is being reused for more than one run of the program. Persistent caching makes it even more worthwhile to do more optimisations as part of the overhead of translation from target instructions to a programming language.

5.6.3 System Level Emulation

A compressed ARM-Linux zImage is booted up using SimIt-ARM to briefly test system level emulation performance. All tests are performed on 1m threshold, 1-4 processor cores and varying optimisation levels.



Chip-multiprocessing benefits system level simulation just as user-level simulation did. It is also worth doing more optimisations when multiprocessing systems when more than 2 processing cores are used. The overall user keyboard response on using the shell in the bash ARM Linux terminal is also faster with multi-processing as the bash shell code is being translated in parallel at boot time.

Chapter 6

Conclusion and Future Work

Parallel Instruction-Set Simulation is a very new technique to improve emulation performance. There is a lot of work left to be done. This section briefly summarises some possible future research work related to parallel ISS process improvement and the main findings of this project.

6.1 Investigate more optimisation engines and Passes

Not all possibilities for tuning the simulator are used. Using a compiler as a translation tool opens up many possibilities for tuning to specific architectures. In fact, GCC can with the help of the `-mtune` and `-march` option to tune the translations using architecture specific instructions i.e. SSE2 for Pentium 4 processors during the instruction selection phase to improve simulation performance. There are also options for register allocation optimisation that had to be flagged at compile time normally. Examples of such an option is the `-fira-algorithm=algorithm` where the argument `algorithm` can be `priority` or `CB`. These tune the register allocations using the Chow's Priority colouring and the Chaitin-Briggs' colouring algorithms respectively.

It is also possible to further investigate LLVM as a compiler infrastructure [12] to experiment advanced optimisation passes. It offers optimisations at all stages of a program, namely at compilation or AST and SSA states, machine language stage (where LLVM comments are used to track possible profile guided optimisation opportunities) and offline reoptimisation.

According to the authors of the compiler infrastructure, LLVM offers a significantly better code quality and optimisation performance when compared to the standard `gcc/g++`. As compilation and translation work is distributed to slave processors/threads, a lot more optimisation can be done in the background.

At the moment, LLVM is unable to be used by SimIt-ARM due to a front-end issue that conflicts between the use of C99 `_extern_inline` function libaries. More time is needed to resolve this issue as it potentially could have used up too much of this project's time.

6.2 Proactive Compilation strategies

Before work can begin on a speculative compilation heuristic, further study must be done on the tolerance of mis-predictions such as with a very generous heuristic. The experiment on h264 in section 5.5.2 with a generous speculation can be expanded to all benchmarks to see if it is really worth doing such a heuristic. Nevertheless, such a proactive strategy still has to rely on some dynamic profiling of the target code to be able to accurately detect hot-code. Alternatively, it is also possible to use a dynamic threshold for individual blocks of code instead of using a general threshold value for all blocks of instructions. Growth rates of block hits can be analysed for faster detection of hot-code.

6.2.1 Incorporating the Block Graph analysis

As one possible way to construct a proactive heuristic, a target executable can be viewed as a graph of code blocks. Each block has several jumps, conditional and unconditional branches to other blocks in the program. To predict which block will be hottest (i.e. most frequently executed), the following policies or hypotheses can be formulated:

- Hot blocks are often the targets of jumps/branches from other blocks
- Hot blocks also have many backward jumps due to nested loops and `if` statements

With this, the slave threads can begin analysing the program after being loaded into memory. The algorithm looks out for the characteristics mentioned above in the program, marks the location and translates them immediately. After translating a block, it continues analysing from the saved location. The memory space of the program is divided equally among processors so that threads can work with mutual exclusion. As mentioned in Section 5.2 OpenMP might be tried out on this coarse grained approach to parallel simulation.

6.3 Using Linked Basic blocks

Instead of translating blocks of memory, it would be more efficient, comprehensible and stable to implement and translate at the basic block level. These basic blocks can be stitched to form the cache in one shared library. This will probably improve the efficiency of translation as the actual amount code that is actually translated may not be that much. It will also be possible to do basic block level persistent caching that is flexible and do not invalid that many parts of the cache when code is changed or added. Compiling a long stringed basic block of instructions will not only be more efficient but will also give the compiler more scope for optimisation.

The basic block level of translation can also use branch prediction techniques (as opposed to profiling techniques) to accurately translate code before they are executed. Accurate branch prediction techniques are prevalent in the literature for JIT compilers, language translators and modern processor pipelining mechanisms. They should be fairly effective and easier to design.

A similar concept of this paradigm is demonstrated by Daniel et. al. [14]. The EHS JIT simulator uses a design strikingly similar to SimIt-ARM but caches at the basic block level and stitches them to form Strongly Connected Components (SCC), Control Flow Graphs (CFGs) and pages. In this way, it can benefit from higher levels of optimisation and achieve just the correct amount of translations needed. Doing this type of translation in parallel is one of the future work as well.

6.4 Summary of Findings

Chip-based Multi-processing offers significant improvements and convenience for the average product engineer working on consumer grade multi-core PCs. Translation time is high in the compiled simulation strategies but it also provides the balance between portability and target-adaptibility. Hence, being able to do the decoding and compilation in parallel improves simulation performance and makes compiled simulation techniques more popular than before for product engineer users. This is because more processing cores can provide the additional processing power to translate and cache more target code as well as perform more code optimisation in parallel.

Translating at high optimisation levels generally yields better results as there is usually much more time spent in running compiled code. To improve the performance of shorter running programs, it is better to use more CPUs to help in doing the translation and optimisation. Larger blocks of translation units are more likely to benefit from compiler's optimisations. Therefore future work also has to focus on creating larger translation units for optimisation.

At best, a prediction strategy should make significant improvements in simulation speed. However, good prediction techniques are not easy to design and might not be much better than just lowering the threshold, increasing the number of compute nodes and do more optimisation. More work is needed to investigate this possibility.

Threading schemes that provide fine threading control are currently better for multiprocessing simulators. More work can investigate the use of coarse grained program profiling techniques to translate target code so that simple threading models like OpenMP can be used to its potential.

Appendices

```
    while (running)
    {
          unsigned ind;
          unsigned pblk; // physical address index

          //threads sleep in the below function if the queue is empty
          get_block_to_compile(&ind, &pblk);

           /*
              if its an existing DLL and the thread is idle for more than
              <idle_thres> amount of time, optimise the cache
           */

           if(cache[pblk].dll_pointer && idle_interval > idle_thres)
           {
                   char filename[1024];
                   char funname[1024];

                   sprintf(filename, "%s/libXcompiled_%u.so", cache_dir, ind);
                   //DLL's function name
                   sprintf(funname, "compiled_%u", ind);

                   reoptimise_dll(ind);

                   void* handle = dlopen(filename, RTLD_LAZY);

                   fptr tt = (fptr) dlsym(handle, funname);

                   while(pblk == curr_blk)
                           continue;

                   cache[pblk].dll_pointer = tt;

          }
          else //its a newly identified block
          {
                  compile_block(buf, ind))
                  update_dll_dict(ind, crc);
                  load_lib(pblk, ind));
          }
  }
```

Figure 1: Partial code body of thread doing idle-time reoptimisation

```
while (running)
{
        unsigned ind; // DLL number
        unsigned pblk; // physical address index

        get_block_to_compile(&ind, &pblk);

        if (!running) pthread_exit(NULL);

        mem->read_block(buf, pblk << shiftval, bufsize);

        unsigned crc;
        unsigned dres = lookup_dict(buf, &crc);

        //if dictionary has existing translated code, load it

        if (dres != -1u) load_lib(pblk, dres);

        else if (compile_block(buf, ind))
        {
             pthread_mutex_lock(&srv_mut);
             update_dll_dict(ind, crc);
             pthread_mutex_unlock(&srv_mut);

             load_lib(pblk, ind);
        }

 }
```

Figure 2: Pthread version of slave thread body

```
while (running)
{
          unsigned ind; // DLL number
          unsigned pblk; // physical address index
          bool has_a_block=false;//is there a block to compile?
          has_a_block = get_block_to_compile(&ind, &pblk);

          if (!running) break;

          if(has_a_block)
          {
                 if (!running) pthread_exit(NULL);

                 mem->read_block(buf, pblk << shiftval, bufsize);

                 unsigned crc;
                 unsigned dres = lookup_dict(buf, &crc);

                 //if dictionary has existing translated code, load it

                if (dres != -1u) load_lib(pblk, dres);

                else if (compile_block(buf, ind))
                {
                       omp_set_lock(&srv_mut);
                       update_dll_dict(ind, crc);
                       omp_unset_lock(&srv_mut);
                       load_lib(pblk, ind);
                }

          }
 }
```

Figure 3: OpenMP version of slave thread body

```
void *PIworker(void *arg)
{
    int i, myid;
    double s, x, mypi;
    myid = *(int *)arg;
    s = 0.0;
    for (i=myid+1; i<=n; i+=num_threads) {
        x = (i-0.5)*d;
        s += 4.0/(1.0+x*x);
    }

    mypi = d*s;
    pi += mypi;
    pthread_exit(0);
}

/****************  in main() *******************/

  int i;
  int *id;
  n = atoi(argv[1]);  num_threads = atoi(argv[2]);
  d = 1.0/n;
  pi = 0.0;
  id = (int *) calloc(n,sizeof(int));
  tid = (pthread_t *) calloc(num_threads, sizeof(pthread_t));

  for (i=0; i<num_threads; i++) {
       id[i] = i;
       if(pthread_create(&tid[i], NULL, PIworker, (void *)&id[i])) {
              exit(1);
       }
  }

  for (i=0; i<num_threads; i++)
    pthread_join(tid[i],NULL);

 printf("pi=%.15f\n", pi);
```

Figure 4: Pthreads parallel Pi program

```
#define N 100000000

 int main()
{
    double t, pi=0.0, w;
    long i;
    w = 1.0/N;
    #pragma omp parallel num_threads(2) private(t)
          #pragma omp for reduction(+:pi)
          for(i=0;i<N;i++)
          {
               t = (i + 0.5) * w;
               pi = pi + 4.0/ (1.0 + t*t);
          }
     printf("pi is %f\n", pi*w);
 }
```

Figure 5: OpenMP parallel Pi program

```
#include<stdio.h>
#include <sys/mman.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

typedef int (*entry_func)(void);

void* alloc_code_buffer(int size, int fd){
  return mmap(NULL,size,PROT_EXEC | PROT_READ, MAP_PRIVATE, fd ,0);
 }

 int main()
 {
       int fd = open("./image.bin",O_RDONLY);
       void *funcptr = alloc_code_buffer(10,fd);
       entry_func func = (entry_func) funcptr;
       printf("Result is %d", func());
       close(fd);
 }

 ----------------------------------------
 Output of above: "Result is 42"
```

Figure 6: Code that calls the function image

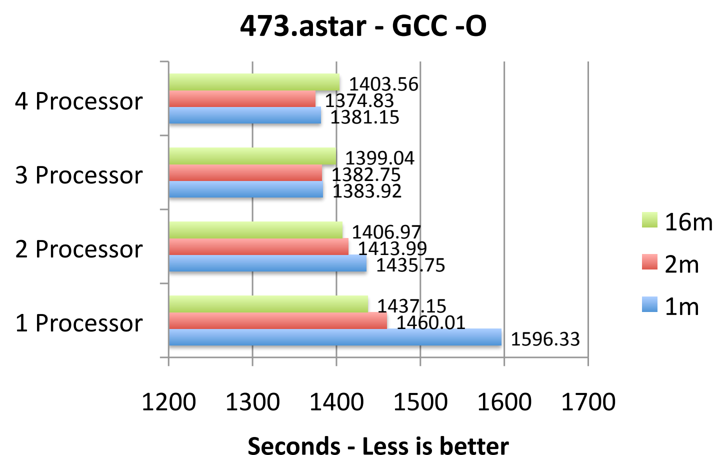


Figure 7: Varying translation thresholds on the 473.astar benchmark

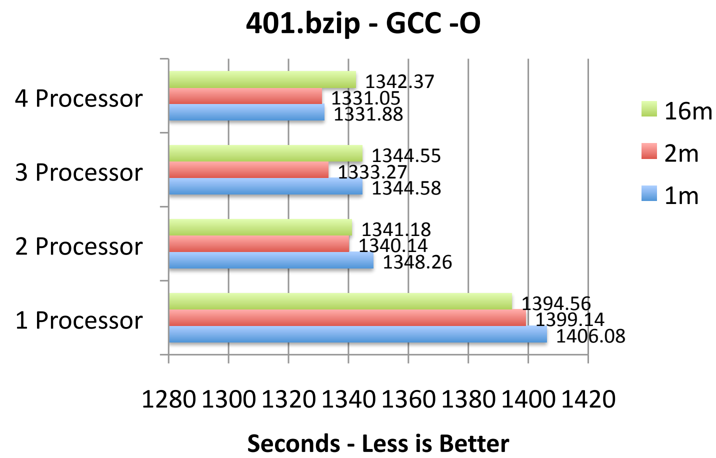


Figure 8: Varying translation thresholds on the 401.bzip benchmark

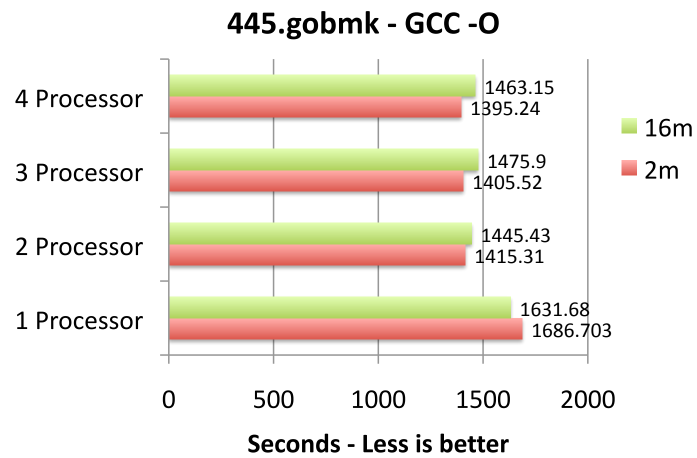


Figure 9: Varying translation thresholds on the 445.gobmk benchmark

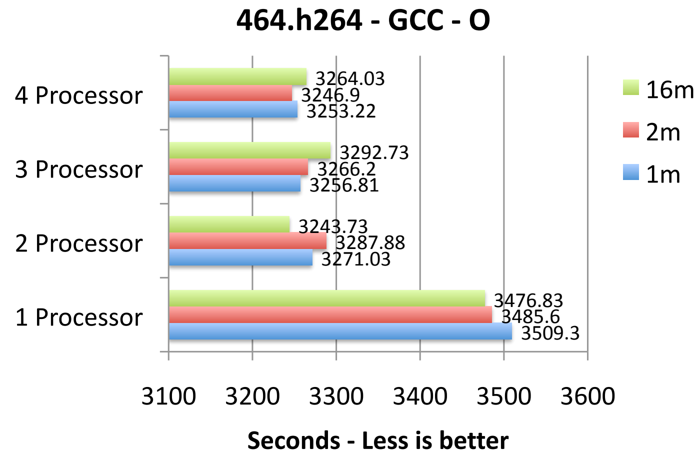


Figure 10: Varying translation thresholds on the 464.h264 benchmark

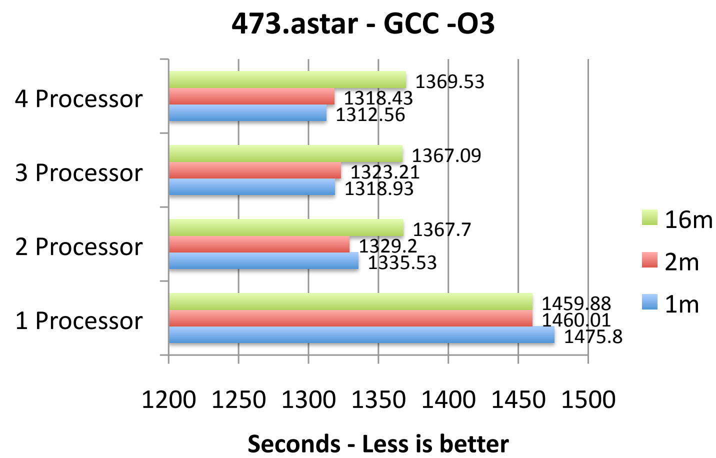


Figure 11: Varying translation thresholds with -O3 on the 473.astar benchmark

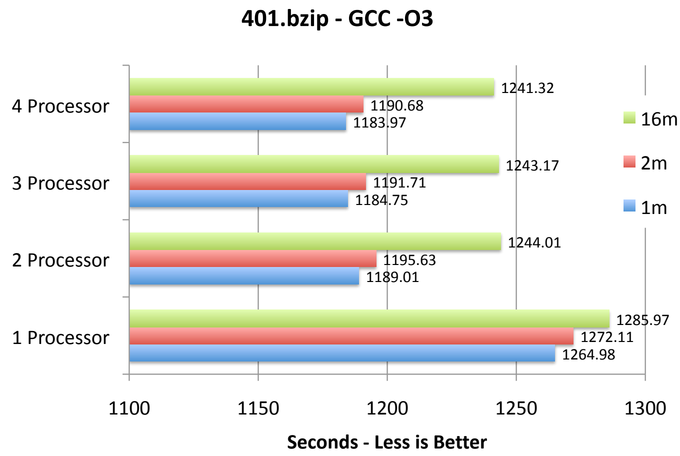


Figure 12: Varying translation thresholds with -O3 on the 401.bzip benchmark

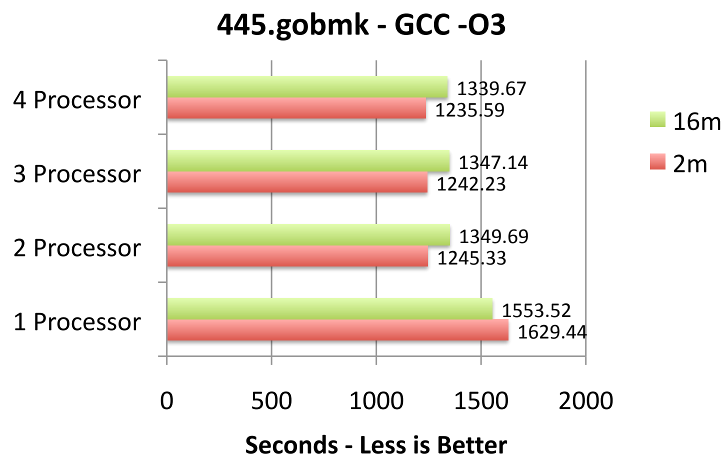


Figure 13: Varying translation thresholds with -O3 on the 445.gobmk benchmark

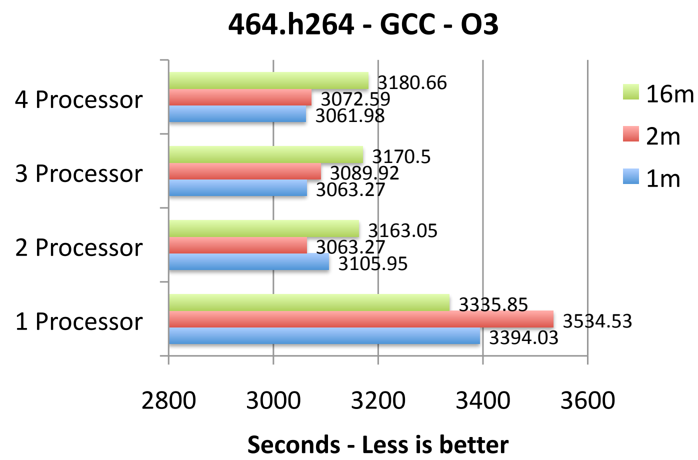


Figure 14: Varying translation thresholds with -O3 on the 464.h264 benchmark

Bibliography

[1] M Probst; *Dynamic Binary Translation*;
UKUUG Linux Developers Conference, 2002.

[2] VMWare Inc.; http://www.vmware.com; VMWare.

[3] Bochs; http://bochs.sourceforge.net/.

[4] C Cifuentes, M Van Emmerik; *UQBT: adaptable binary translation at low cost*; IEEE Computer, Vol. 33, Issue 3, pp. 60-66.

[5] F Bellard, *A Fast and Portable Dynamic Translator*; Proceedings of the USENIX Annual Technical Conference, 2005, usenix.org.

[6] R F Cmelik, D Keppel; *Shade: A Fast Instruction Set Simulator for Execution Profiling*; ACM SIGMETRICS Performance Evaluation Review, Vol. 22, Issue 1, pp. 128-137, May 1994.

[7] E Witchel, M Rosenblum; *Embra: fast and flexible machine simulation*; ACM SIGMETRICS Performance Evaluation Review, Vol. 24, Issue 1, pp. 68-79, May 1996.

[8] J Zhu, D Gajski; *A retargetable, ultra-fast instruction set simulator*; Proceedings of the Design Automation and Test Conference, Munich, Germany, Article No. 62, 1999.

[9] J Lee et. al; *FaCSim: a fast and cycle-accurate architecture simulator for embedded systems*; Proceedings of the ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems, pp 89-100, 2008.

[10] W Qin, J D Errico, X Zhu; *A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation*; Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, pp. 193-198, 2006.

[11] W Qin, S Rajagopalan, S Malik; *A formal concurrency model based architecture description language for synthesis of software development tools*; Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pp 47-56, 2004.

[12] C Lattner, V Adve; *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*; Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, Page 75, 2004.

[13] A Binstock; *Choosing between OpenMP and Explicit Threading Methods*; Intel(R) Software Network, http://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods/.

[14] D Jones, N Topham; *High Speed CPU Simulation Using LTU Dynamic Binary Translation*; Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, pp. 50-64, 2008.

[15] V Sagdeo, *The complete Verilog Book*; Springer, page 372,1998.

[16] SPEC Corp. Standard Performance Evaluation Corporation, http://www.spec.org/cpu2006/CINT2006/.

[17] E Duesterwald, Vansanth Bala; *Software profiling for hot path prediction: less is more*; ACM SIGARCH Computer Architecture News, Vol. 28, Issue 5, pp. 202-211, December 2000.

[18] GNU Compiler Collection Online Documentation - Description of Optimisations, http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[19] M Bartholomeu et. al; *Optimizations for Compiled Simulation Using Instruction Type Information*; IEEE Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, pp. 74 - 81, 2004.