High Speed Simulation of Microprocessor Systems using LTU Dynamic Binary Translation

Daniel Jones

Doctor of Philosophy

Institute for Computing Systems Architecture

School of Informatics

University of Edinburgh

2010

Abstract

This thesis presents new simulation techniques designed to speed up the simulation of microprocessor systems. The advanced simulation techniques may be applied to the simulator class which employs dynamic binary translation as its underlying technology. This research supports the hypothesis that faster simulation speeds can be realized by translating larger sections of the target program at runtime. The primary motivation for this research was to help facilitate comprehensive design-space exploration and hardware/software co-design of novel processor architectures by reducing the time required to run simulations.

Instruction set simulators are used to design and to verify new system architectures, and to develop software in parallel with hardware. However, compromises must often be made when performing these tasks due to time constraints. This is particularly true in the embedded systems domain where there is a short time-to-market. The processing demands placed on simulation platforms are exacerbated further by the need to simulate the increasingly complex, multi-core processors of tomorrow. High speed simulators are therefore essential to reducing the time required to design and test advanced microprocessors, enabling new systems to be released ahead of the competition.

Dynamic binary translation based simulators typically translate small sections of the target program at runtime. This research considers the translation of larger units of code in order to increase simulation speed. The new simulation techniques identify large sections of program code suitable for translation after analyzing a profile of the target program's execution path built-up during simulation.

The average instruction level simulation speed for the EEMBC benchmark suite is shown to be at least 63% faster for the new simulation techniques than for basic block dynamic binary translation based simulation and 14.8 times faster than interpretive simulation. The average cycle-approximate simulation speed is shown to be at least 32% faster for the new simulation techniques than for basic block dynamic binary translation based simulation and 8.37 times faster than cycle-accurate interpretive simulation.

Acknowledgements

I wish to thank my adviser Professor Nigel Topham for his support and advice, and for sharing his knowledge of microprocessor architectures during my research.

I would also like to thank all my colleagues within the Institute for Computing Systems Architecture at the University of Edinburgh for their support and encouragement, especially Dr Björn Franke, Professor Mike O'Boyle and Dr Marcelo Cintra.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification. Some of the material presented in this thesis has been published in the following papers:

> D. Jones and N. Topham, "High Speed CPU Simulation using LTU Dynamic Binary Translation", in *HiPEAC '09: Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, Paphos, Cyprus, 2009.

> N. Topham and D. Jones, "High Speed CPU Simulation using JIT Binary Translation", in *MoBS '07: Proceedings of the 3rd AnnualWorkshop on Modeling, Benchmarking and Simulation*, San Diego, CA, USA, 2007.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The research presented in this thesis investigates some novel, high speed simulation techniques which were developed to help facilitate comprehensive design-space exploration (DSE) and hardware/software co-design of microprocessor architectures. Thorough exploration of the design-space is often not performed in situations where a new system must be designed within a limited period of time. As a consequence sub-optimal system designs may be selected for manufacture. This problem is particularly acute in the embedded processor domain where companies work within tight schedules to release new systems to market. Faster simulators allow the design-space to be explored in more detail within the time available. They are therefore very important in the microprocessor design process as they enable the best system architectures to be discovered.

Simulators are used to accurately predict the performance characteristics, such as speed and power consumption, of new processor architectures so that the most efficient design can be selected for fabrication. Simulators are also used to test experimental instructions set architectures (ISAs), for hardware/software co-design and verification, and in the development and debugging of new compilers and applications. Simulation platforms are vital to industry because they enable the engineering tasks involved in the development of a new microprocessor to be performed in parallel, thus reducing the time to release.

The design-space for a new microprocessor architecture is typically very large. Its size will depend on a number of different factors such as the ISA, pipeline design, register file size, functional unit type and quantity, number of processor cores, on-chip network, cache hierarchy and configuration, and application domain. In most embedded system research and design groups, not only are the design goals very exact, in terms of the performance criteria that must be met, there is also limited time available in which to design, test and fabricate a system. Time constraints may necessitate a reduction in the size of the design-space in order that a new system can be released on schedule. System designers may be required to make assumptions about individual micro-architecture parameters or to chose less representative applications with which to test the system. However, a reduced design-space is less likely to contain the best design point.

The design-space may be reduced by decreasing the number of micro-architecture parameters and configuration values to be explored for the target system. Whilst this will cut the overall simulation time it may result in the selection of a sub-optimal design. The design-space may also be reduced by decreasing the number and size of the programs simulated. Small benchmarks may be run instead of real-world programs in an attempt to replicate real application behaviour and at the same time reduce the overall simulation time. However, benchmarks can not imitate real-world programs perfectly and this may also result in the selection of a sub-optimal system design.

The need to model new, advanced system designs has increased the processing demands placed on simulators. In addition to accurately modelling increasingly complex, multi-core systems, simulators must also maintain statistics on a range of system indicators. The degree of modelling detail, accuracy and instrumentation may also be reduced in order to speed up simulation, but this will generate less reliable data from which to select the best design.

Comprehensive design-space exploration therefore involves testing every design point with the simulation of real-world programs. However, rigorous design-space exploration conflicts with the manufacturers natural desire to ship leading-edge systems ahead of the competition. As outlined, simulators play an important role in determining the optimal system design. The simulation speed directly affects the time required to design and to test new system designs and hence it indirectly affects the accuracy of the predicted performance results. For this reason, state-of-the-art high speed simulation techniques remain an active field of research.

In order to perform comprehensive design-space exploration and verification a simulator needs not only to be fast, it must also satisfy the requirements of a research simulator. This means that the simulator be flexible in both its configuration and operation. It should be capable of modelling any target system architecture and offer a number of different simulation modes.

A flexible research simulator should fulfil the following criteria:

- **High Speed Simulation**. The design-space needs to be comprehensively explored if the optimal system design is to be found. A linear increase in the number of micro-architecture design parameters results in exponential expansion of the design-space. Vast design spaces, coupled with the need to accurately model complex systems and the desire to run real-world programs demands a lot of processing power. Therefore, high speed simulators are required to explore the design-space as thoroughly as possible, in as short a time as possible.

- **Accurate Modelling**. The simulator must be capable of modelling the target system at the required level of abstraction and with the desired degree of accuracy in order to confidently predict the behaviour of the real system. It should incorporate instruction level and cycle-accurate modes of operation to facilitate high and low level DSE.

- **Instrumentation**. The performance of the target system can only be evaluated if the simulator is instrumented to return statistics on the system indicators of interest. For example, the simulator may be instrumented to provide instruction counts, program execution cycles, instruction execution profiles, L1 hits and misses, and power consumption figures.

- **State Observability**. The ability to capture all architecturally-visible CPU state changes at instruction commit is necessary in order to support hardware/software co-design and verification.

- **System Simulation**. To accurately model the target system's behaviour, the simulation environment should be setup to precisely mirror that anticipated for the real system. A research simulator should therefore possess the ability to simulate standalone applications and full operating systems (OSs) or embedded system runtimes. User level simulation requires emulation of all system calls, whilst system level simulation requires comprehensive modelling of the system's hardware and I/O devices.

- **Simulation Flexibility**. Simulators which possess a variety of simulation modes offer greater flexibility for performing DSE. The appropriate level of abstraction, speed and accuracy of simulation may be chosen by designers so as to satisfy the optimal exploration strategy for a particular project. Exploration may be focused on specific regions of code that are of interest by employing a mixture of fast-forwarding and sampling.

- **Target System Configuration**. To be of use, the simulator must be capable of modelling all of the target systems present in the design-space. It should possess a straightforward method of building complex system models and facilitate detailed configuration of all micro-architecture parameters.

- **Retargetable**. The simulator needs to support different target ISAs and the extension of ISAs so that the benefits of experimental ISAs can be investigated. It should employ a standardized means of defining the target instruction set and the pipeline model.

Instruction set simulators (ISSs) are important software tools which are used to design the advanced processor architectures of tomorrow. They enable the trade-offs between different micro-architecture models to be explored so that the best system design can be selected for production. They also facilitate the development, verification and debugging of hardware and software.

The new simulation techniques detailed in this thesis are applicable to Dynamic Binary Translation (DBT) based simulators. Dynamic binary translation is a high speed emulation technique [Altm 00, Altm 01] which has applications across many domains. Dynamic binary translation based simulators typically run programs up to four times faster than the corresponding field-programmable gate array (FPGA) setup.

This research is concerned with high speed research simulators which can be used to carry out DSE and hardware/software co-design and verification of novel processor architectures. In order to perform these tasks the simulator must be capable of high speed instruction level and cycle-accurate simulation, and support observable modelling of the processor state. A glossary of the main simulation terms used throughout this thesis are listed in appendix A.

1.1 The High Speed Simulation Problem

Intense competition amongst embedded system designers has led to more leading-edge processors being released to market more frequently. Modern processors incorporate many novel technologies which are designed to increase execution speed and to reduce the power consumption and thermal output. As the complexity and number of cores in future systems increase, the time necessary to run simulations of such systems also increases, at the same time deadlines are becoming tighter.

Superscalar processors incorporate a number of advanced micro-architecture technologies which increase performance. Processors may employ long pipelines, wide instruction issue, out-of-order processing, speculative execution or a trace cache. A simulator must model all of these novel components in addition to emulating complex events such as interrupts and exceptions. Even embedded processors incorporate Memory Management Units (MMUs) to support multi-tasking operating systems.

Chip Multi-Processors (CMPs) are rapidly becoming the preferred processor model as manufacturers strive to make the most effective use out of the ever increasing silicon area available to them [Oluk 96]. Dual-core (AMD Athlon X2; IBM POWER6; Intel Itanium, Core 2 Duo) and quad-core (AMD Phenom X4, Opteron; Intel Core 2 Quad, Core i7) processors are already in mainstream use and the number of on-chip cores is set to increase many fold in the near future. A CMP may contain heterogeneous or homogeneous cores. IBM's CELL processor [Kahl 05] is a heterogeneous CMP designed for the games market and consists of a single general-purpose PowerPC core and 8 special-purpose digital signal processing cores.

Sun Microsystems' UltraSPARC T1 processor [Kong 05] is a homogeneous Chip Multi-

Threaded (CMT) processor which has 8 cores, each of which is 4-way multi-threaded, providing a total of 32 hardware thread contexts or logical processors. The processing demand on simulators rises dramatically as the number of processor cores is increased. Simulators will soon be required to emulate systems with hundreds - if not thousands - of cores as well as model the associated on-chip network, cache hierarchy, coherency protocols and I/O.

In order to discover the optimal system design, the effect of all interesting micro-architecture parameters should be evaluated during DSE. However, the consideration of a large number of parameters increases the size of the design-space and thus requires significantly more time to explore. To accurately predict the behaviour of the real system, the simulation environment must be setup to reflect the real system environment. This may involve running real-world programs and operating systems which also take time to simulate.

Table 1.1 provides definitions of the main simulator classes. Most interpretive simulators achieve modelling accuracy by operating at the register transfer level (RTL), but such simulators are very slow. However, compiled simulators, which are many orders of magnitude faster than interpretive simulators, do not provide processor state observability. They can also only be used in situations where the binary code to be simulated is known in advance and are therefore unable to emulate self-modifying code. Sampling simulators on the other hand can perform cycle-accurate simulation at speed, but their functional simulation speed is similar to that of interpretive simulators.

Dynamic binary translation based simulators, which perform a mixture of interpretation and compilation at runtime, are both fast and flexible. Dynamic binary translation is the process of translating source code, which performs a specific task, into the equivalent binary code which runs on the host machine. When executed, the translated code performs the same task but at much greater speed. Dynamic binary translation based simulators are also capable of simulating self-modifying programs and support observability of the processor state.

Dynamic binary translation based systems are very flexible as they enable extensive runtime control over program modification. They are able to react to unforeseen events by generating code which is translated on-the-fly to deal with a new situation. Software employing DBT technology is used across many different application domains such as

Simulator Class	Definition	High Speed	Self-modif.	State Observ.
Interpretive	The target program is simulated by repeatedly fetching, decoding and emulating the next instruction in the execution stream. Interpretive simulators are flexible and support the running of self-modifying code. Decode caches are used to reduce the overhead of instruction decoding, but interpretive simulators are still comparatively slow.	No	Yes	Yes
Compiled	A statically compiled simulator is generated by translating the target binary into an executable which when run simulates the target program. This class of simulator is optimized for speed but lacks flexibility and typically can not run self-modifying code.	Yes	No	No
DBT	The simulator switches between interpretive simulation and ‘translated’ simulation in which host code functions are called to emulate sections of the target program at high speed. Whilst interpreting instructions, sections of the program are identified for translation into host code functions. If the simulator detects that code has been modified it discards the corresponding translations. Dynamic binary translation based simulators are both fast and flexible.	Yes	Yes	Yes
Sampling	Sampling is used to perform fast cycle-approximate simulation. The simulator alternates between functional simulation and cycle-accurate simulation. Cycle-approximate simulation is fast when the fast-forwarding interval is many times larger than the sampling interval. The data gathered in each sampling interval is used to generate the simulation statistics. However, state observability is typically not maintained during the fast-forwarding intervals.	No	Yes	No

Table 1.1 Simulator Classes. This table lists the main simulator classes and shows whether they feature high speed instruction level simulation, emulation of self-modifying code and state observability.

resource virtualisation, system resilience, network security, dynamic code patching and optimization, and system emulation.

Dynamic binary translation based simulation is a high speed simulation technique in which target instructions, or more typically blocks of instructions, are translated at runtime into equivalent host code functions (translated functions). The translated functions are then called to emulated the basic blocks at high speed within the simulated microprocessor model. In most simulations, the overhead of performing translation is more than offset by the time saved through faster simulation. Figure 1.1 provides an overview of the tasks involved in DBT based simulation.

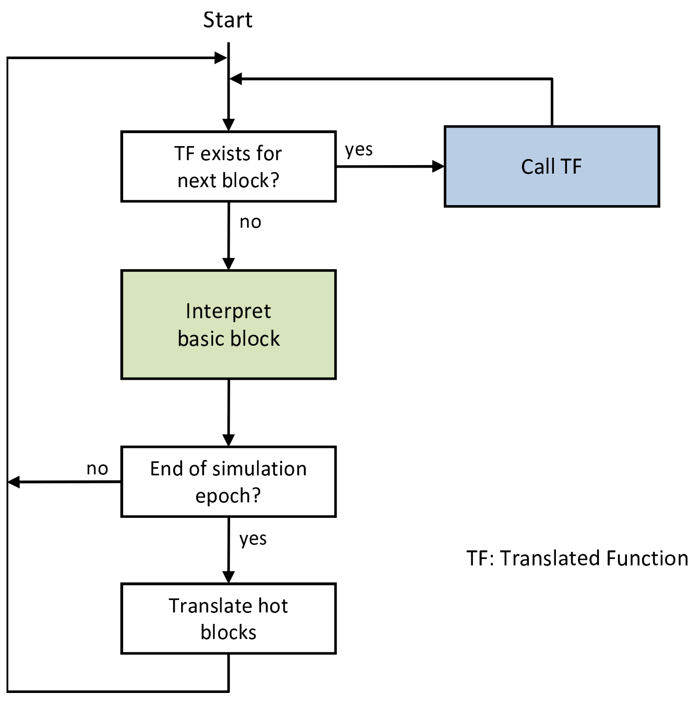


Figure 1.1 DBT Simulation Overview. Basic blocks are interpreted for a defined period (simulation epoch). At the end of each simulation epoch the frequently executed blocks are translated into host code functions (translated functions). If the next block to be simulated has previously been translated, the translated function for the block is called, else the block is interpreted as usual.

High speed cycle-accurate simulation is required in order to perform low level design-space exploration. However, because DBT based simulators perform instruction level simulation very efficiently, there exists little scope for increasing the speed of cycle-accurate simulation through optimizations in areas other than cycle-accurate modelling. Therefore significant speed-ups in DBT based cycle-accurate simulation may only be realized by deploying simplified models of target system components.

There are limitations associated with all of the main simulation techniques. Whilst interpretive simulators are flexible, they are also slow. Compiled simulators on the other hand are fast but place restrictions on the type of programs that may be simulated. Sampling simulators can perform cycle-approximate simulation at high speed but do not provide state observability. Dynamic binary translation based simulators feature high speed instruction level simulation, emulation of self-modifying code and processor state observability.

High speed ISSs contribute towards reducing the time needed to carry out DSE. As outlined previously, the time required to perform DSE is dictated by the size of the design-space, the complexity of the system to be modelled, the need to simulate real-world programs and the scheduled system release date. The processing demands placed on simulators are set to increase substantially in the near future as designers seek to model advanced new multi-core CPUs. The central challenge for today's system designers, of achieving high speed simulation whilst retaining absolute modelling accuracy, is therefore becoming increasingly difficult to satisfy.

The main simulation issues which are addressed in this thesis are:

- **High Speed Instruction Level Simulation**. Simulators must be capable of performing high speed instruction-level simulation in order to facilitate comprehensive high level DSE.

- **High Speed Cycle Accurate Simulation**. Simulators must be capable of performing high speed cycle-accurate simulation in order to facilitate comprehensive low level DSE. Cycle-approximate models of the target system are typically used to speed-up simulation at the expense of introducing small degrees of error into the simulation results.

- **State Observability**. The processor state must be accessible at all observation points so that high and low level hardware/software co-design and verification can be performed. The hardware model is validated against the golden reference model using the simulator's co-simulation API.

- **Realistic Simulation**. Simulators must be capable of running real-world programs within a realistic simulation environment if they are to accurately predict system behaviour. This implies that the simulator should support the running of stand-alone applications, self-modifying programs and operating systems.

1.2 Large Translation Unit Solution

This thesis proposes that the simulation speed of DBT based simulators can be increased by identifying large translation units at runtime. By profiling the target program's execution path during simulation it is possible to identify large sections of code which span multiple basic blocks and which are suitable for translation. If the translator has a larger section of target code to analyze it will be better able to optimize the translated code produced for speed of execution.

Large translation units not only contain more target instructions they also have more branches and jumps to instructions within the same translation unit. Indeed, entire loops, even nested loops may be contained within a single translation unit. This means that more instructions will be simulated per translated function call and control will be returned to the main simulation loop less frequently. This results in an overall increase in simulation speed as less time is spent in the slower main loop.

This thesis investigates the performance benefits of translating three different types of large translation unit, or LTU. The different LTUs are based on the standard computer software objects listed below.

SCC	:	Strongly Connected Component
CFG	:	Control Flow Graph
Page	:	Physical Page

The DBT simulation process is divided into a number of simulation epochs. During each simulation epoch the simulator builds up a profile of the target program's execution path. At the end of each epoch the execution path profile is analyzed to identify the LTUs and to determine which LTUs should be translated. In subsequent epochs, large sections of the target program may be simulated at high speed by calling the corresponding translated function.

1.3 Research Contributions

The Edinburgh High Speed (EHS) simulator, developed at the University of Edinburgh, is a high speed DBT based simulator and is the platform on which all of the simulations were performed for the research presented in this thesis.

This thesis contributes to the knowledge of high speed DBT based simulation as follows:

- **Novel High Speed Simulation Techniques**
  This thesis shows that LTUs can be deployed to increase the simulation speed of DBT based simulators. The techniques used to profile target programs and to identify and translate LTUs at runtime are unique to the Edinburgh High Speed simulator and are outlined in chapter 6.

  The EHS simulator was designed as a research simulator suitable for performing DSE and hardware/software co-design of novel microprocessor architectures. The processor state is updated after each instruction is emulated and is observable at every translation unit boundary. The simulator also incorporates advanced management of translations so that self-modifying code can be simulated.

- **Quantitative Analysis of Simulation Techniques**
  The performance of the different LTU DBT simulation modes are analyzed in chapters 7 and 8 and provide an insight into their effectiveness and future potential as high speed simulation techniques. Many aspects of LTU DBT based simulation are investigated in detail, including analysis of the number of instructions emulated by translations on the first

and second simulation runs; the time spent performing the different simulation tasks; the size and number of translation units generated; the size and frequency of the translated functions called; the factors which cause translation unit fragmentation and the effects of varying the size of the simulation epoch.

- **Instruction Level Performance Analysis**
  The instruction level simulation performance of the different LTU DBT simulation modes are compared in chapter 7. The results show that all of the LTU DBT simulation modes are on average at least 1.63 times faster than basic block DBT based simulation.

- **Cycle Approximate Performance and Accuracy Analysis**
  The cycle-approximate simulation performance of the different LTU DBT simulation modes are compared in chapter 8. The results - using simplified models for the target pipeline and memory sub-system - show that all of the LTU DBT simulation modes are on average at least 1.32 times faster than basic block DBT based simulation. The simplified system model is shown to introduce an average error of 2.4% into the cycle count.

- **Comparison with State-of-the-Art Simulators**
  Chapter 7 demonstrates that, in addition to being a flexible research simulator, the EHS simulator is capable of performing instruction level simulation at speeds comparable with other state-of-the-art simulators which were designed purely for speed. The EHS simulator completed simulation of a set of benchmarks on average 1.07 times quicker than Simit-ARM and 1.26 times slower than QEMU (ARM).

1.4 Thesis Outline

This thesis is divided into nine chapters. Chapter 2 provides an overview of the main simulation techniques used in simulating microprocessor systems. Chapter 3 summarizes related work in the field of static and dynamic binary translation based simulators and Chapter 4 describes the design and operation of the Edinburgh High Speed simulator used in this research.

Chapter 5 outlines the benchmarking methodology used to asses the performance of the different DBT simulation modes and Chapter 6 describes the different types of Large Translation Unit and details how they are identified and translated at runtime.

Chapters 7 and 8 analyze the performance of the instruction level and cycle-accurate simulation modes respectively. The simulation speed of each LTU DBT mode is presented and compared with that of basic block DBT based simulation. Chapter 7 also analyzes the characteristics of the different LTUs generated and compares the performance of the EHS simulator with two start-of-the-art functional simulators. Finally, chapter 9 presents the conclusions and outlines future work which naturally follows on from this research.

Chapter 2

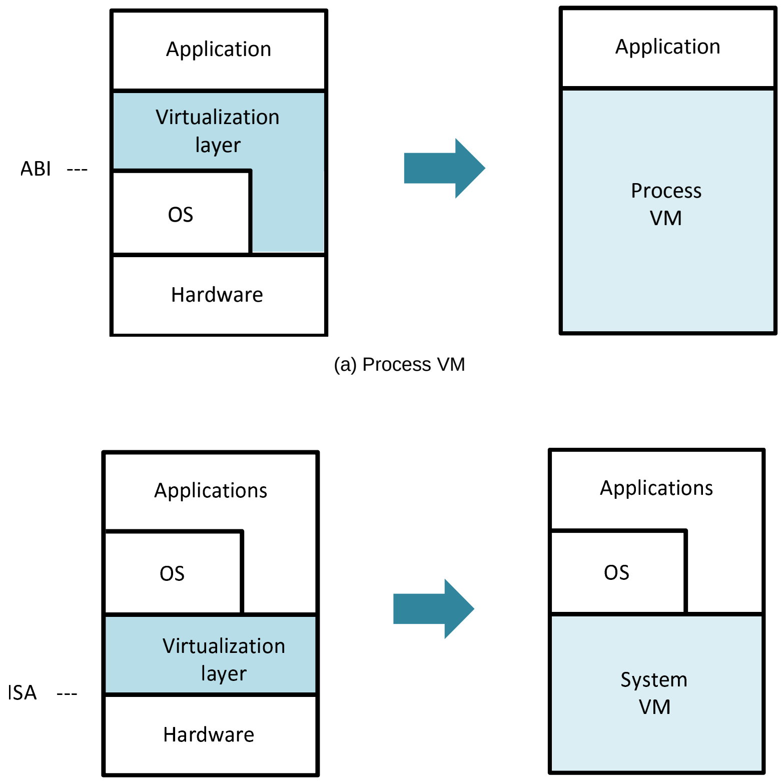
Simulation Techniques

This chapter describes the simulation techniques employed in instruction set simulators which can be used to perform design-space exploration of microprocessor systems. The different simulation techniques used define a simulator's strengths and weaknesses, and therefore its application domain. Whilst slower simulators tend to provide flexibility of operation, the fastest simulators are restricted in their use. Hybrid simulators on the other hand, which employ a combination of simulation techniques, have the potential to be both fast and flexible and are therefore ideal for carrying out research.

2.1 Overview

Simulators simulate programs by emulating each target instruction within a model of the target system running on the host machine. The simulation environment must model all of the lower-level components present in the target program's native execution environment. Therefore, all simulators form a virtualisation layer [Gold 73, Pope 74, Smit 05a, Smit 05b] between the simulated application and the host platform.

The two types of virtualisation layer are shown in figure 2.1. Process virtualisation supports the execution of a single process, or single-threaded application, by abstracting the Application Binary Interface (ABI). Process virtual machines (VMs) emulate user-level instructions and operating system calls, and are initiated when a process is created and destroyed when the process terminates. System virtualisation provides a complete system environment which supports the running of operating systems and their processes by abstracting the ISA interface. System VMs map a virtual guest system onto a real host system by emulating hardware components such as processors, memory and I/O devices.



(a) Process VM

(b) System VM

Figure 2.1 **Virtual Machines** The two figures show the virtualisation layers used to simulate a) a process and b) an entire system.

2.2 Interpretation

Traditional interpretive simulators, such as SimpleScalar [Burg 96, Burg 97, Aust 02] and Bochs [Lawt 96], start by loading the target binary into simulated target memory. The simulator then fetches, decodes and emulates the next instruction in the execution path [Half 94]. The fetching, decoding and execution tasks are usually performed in a monolithic function. After fetching and decoding the next instruction opcode from memory, the simulator calls an instruction specific function which emulates the instruction's behaviour. The function carries out the instruction operation within the processor model, updating the processor state, general purpose registers and main memory as required. Instrumentation functions may also be called to gather data on indicators such as the instruction count, total execution cycles and power usage statistics.

Figure 2.2 shows how a basic interpretive simulator might be implemented in C code. The main simulation loop is represented by the `while` statement, with the `switch` statement directing control to the next instruction to be emulated. The next instruction opcode is fetched and decoded by calling the `decode_opcode` function. This function will search a decode cache in order to return previously decoded instructions as quickly as possible. The decoded instruction is then matched with the corresponding `case` statement which emulates the instruction and updates the program counter (PC). The `break` statement marks the end of each instruction and transfers control back to the main simulation loop.

Hardware decoders are fast, but instruction opcode decoding in software is a very time consuming process. This is because each opcode must be bit tested in order to ascertain the instruction operation, addressing mode, source and destination operands, data size (16/32 bit), indexing mode and any conditional execution flags to be tested. Interpretive simulators typically execute between 10 and 100 host instructions per target instruction [May 87]. However, interpretive simulation is flexible and enables accurate modelling of the target processor, albeit at relatively slow speeds.

2.3 Binary Translation

Binary translation [Cifu 96] is a technique used to convert binary code (target), which has been compiled [Aho 86, Torc 07] to run on one processor architecture, into binary code (host) which can be run on a different - or the same - processor architecture. When executed, the host binary reproduces the behaviour of the target binary within the simulated target environment. The host binary generated is capable of emulating the target

```
while (!end_of_simulation) {

    inst = decode_opcode(PC);

    switch (inst) {
        case ADD:
                *a = *b + *c;
                PC++;
                cycles++;
                stats[ADD]++;
                break;
        case MPY:
                *a = *b x *c;
                PC++;
                cycles+=2;
                stats[MPY]++;
                break;
        case J:
                PC = *c;
                cycles++;
                stats[J]++;
                break;
        ...
    }
}
```

Figure 2.2 Interpretive Simulator Code. The next target instruction is fetched from memory address PC and decoded by calling the decode_opcode function within the main simulation loop. The decoded instruction opcode is then matched with an instruction `case` statement which emulates the instruction by updating the simulation environment. Variables `a`,`b` and `c` are pointers to general purpose registers which are assigned at instruction decode. The execution cycle count is maintained in the `cycles` variable and instruction profiling is achieved using the `stats` variable.

program up to 11 times faster than is possible with interpretive simulation. Chapter 3 covers the work carried out by others into binary translation based simulation.

There a two main types of binary translation: static binary translation and dynamic binary translation.

- **Static Binary Translation (Compiled)**. The target binary is parsed by a translator which analyzes it to discover all possible execution paths and then generates the simulator executable. The simulator is then run on the host machine to simulate the target program at high speed. Some compiled simulators incorporate a fallback interpreter to deal with instructions which were not identified during compilation.

- **Dynamic Binary Translation**. The target binary opcodes are fetched, decoded, emulated and profiled by the simulator. Frequently emulated sections of the target binary are then translated at runtime into host code functions. The host code functions are then called to emulate the same sections of program code at high speed. Although not as fast as static binary translation based simulators, DBT based simulators are capable of simulating any target program including self-modifying applications.

Simulators which employ binary translation have been used to port legacy applications across to new systems with minimal effort. This has enabled individuals to continue to benefit from their software investment. Rebuilding or possibly rewriting applications can be very time consuming and may require in-depth knowledge of the compilation process, assuming one even has access to the source code. However, whilst binary translation based simulators are many times faster than interpretive simulators, native compilation of the source code remains the fastest way to run a program. This is primarily because the native compiler can view the target program in its entirety and at a higher level of abstraction. This enables the compiler to better optimize the program executable for speed.

2.3.1 Static Binary Translation

The processes involved in static binary translation are shown in figure 2.3. The front-end is responsible for loading and decoding the target binary. The decoded instructions are then translated into an optimized intermediate representation (IR) which is computer and operating system independent. The back-end compiles the intermediate code to generate the simulator executable. The simulator produced is a self-contained executable which when run simulates the target binary.

A compiled instruction set simulator spends most of its time emulating target instructions and is consequently much faster than an interpretive simulator. One straightforward compiled simulator design uses in-line macro expansion [Mill 91] present in

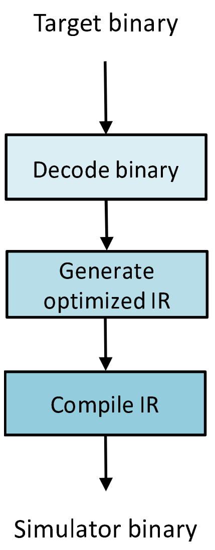


Figure 2.3 Processes involved in Static Binary Translation Static binary translation is performed prior to simulation.

many programming languages such as C. The target target binary is statically translated into a host binary which is then run directly.

A macro is created for each target assembly language instruction. The macro defines the high-level emulation function for each target instruction. For example, macros for the add (`ADD`), branch on equal to zero (`BEQ`) and jump (`J`) instructions may be defined as:

```
#define ADD(a,b,c)  (a) = (b) + (c); cycles++; stats[ADD]++;
#define BEQ(disp)   PC += (disp);    cycles++; stats[BEQ]++;
#define J(target)   PC = (target);   cycles++; stats[J]++;
```

Control instructions, such as direct branches and jumps, with destination addresses that can be computed statically may be modelled using `GOTO` statements and address labels placed before the target instructions. However, control instructions with destination addresses which are computed at runtime can not use such a method. Indirect branch and jump instructions, as well as returns from subroutine calls fall into this category. A `switch` statement can be used to model the execution path at runtime if each instruction macro is defined as a `case` statement, where the `case` statement value is equal to the instruction address. The example instruction macros now look as follows:

```
#define ADD(addr,a,b,c)  case (addr): (a) = (b) + (c); \
                                      cycles++; \
                                      stats[ADD]++;

#define BEQ(addr,disp)   case (addr): cycles++; \
                                      stats[BEQ]++; \
                                      if (status_flag(ZERO)) { \
                                          PC = (addr) + ((disp); \
                                          break; \
                                      }

#define J(addr,target)   case (addr): PC = (target); \
                                      cycles++; \
                                      stats[J]++; \
                                      break;
```

Figure 2.4 shows example C code for a statically compiled simulator. Each target instruction is represented by an instruction macro placed within the main simulation loop (`while` statement). The `switch` statement controls the next instruction to be emulated based on the value of the PC. The figure shows that the first instruction in the target program is an `ADD` instruction at address 0x1000 which adds together source registers `r2` and `r3`, and places the result in destination register `r1`.

```
while (!end_of_simulation) {
    switch (PC) {
            ADD(0x1000,r1,r2,r3);
            ADD(0x1004,r5,r4,r1);
            BEQ(0x1008,0x0008);
            MPY(0x100C,r7,r5,r6);
            ADD(0x1010,r5,r1,r7);
            J(0x1014,r5);
            ...
    }
}
```

Figure 2.4 Compiled Simulator Code. This figure shows the target instruction macros placed within the `switch` statement.

After a non-control instruction, simulation passes on to the next sequential instruction (following `case` statement) as non-control instructions do not end with a `break` statement. The PC is updated with the target address for control instructions which are taken, a `break` statement then forces control back to the main simulation loop. The PC is not incremented after non-control instructions, or not taken control instructions, in order to increase the simulation speed. The `switch` statement is compiled by `gcc` into a set of indexed indirect jumps (jump table) which point to the different `case` statements. This is an efficient way to reference target instructions and enables changes in the control flow to be simulated at speed.

It is possible for large target programs to exceed the maximum code size (compiler dependent limit) allowed within a `switch` statement. If this is the case. the target program can be broken up into smaller sections, with each section being placed within a separate `switch` statement as shown in Figure 2.5.

```
while (!end_of_simulation) {
    switch (PC) {
            ADD(0x1000,r1,r2,r3);
            ...
            PC = 0x1234;
            default: break;
    }
    switch (PC) {
            MPY(0x100C,r7,r5,r6);
            ...
            PC = 0x2FDC;
            default: break;
    }
    switch (PC) {
            ADD(0x1010,r5,r1,r7);
            ...
            HALT;
            default: break;
    }
}
```

Figure 2.5 Compiled Simulator Code for Large Programs. Multiple `switch` statements are used to overcome compiler dependent `switch` size limits. The last instruction within a `switch` statement sets the PC value to equal the instruction address of the next consecutive instruction (first instruction within the following `switch` statement).

The PC is set to the next instruction address after the last instruction within a `switch` statement has been emulated. If the previous instruction was a non-control instruction, simulation continues with the first instruction in the following `switch` statement.

Control instructions are emulated as before, although they may now have to traverse a number of switch blocks before finding a matching target address. The overhead of searching across `switch` blocks for a target address increases with program size, but the associated performance degradation is negligible.

If the same target program is to be simulated many times, which is often the case, then compiled simulation is much faster than interpretive simulation. The initial cost of translating the target program is more than offset by the increased simulation speed. However, compiled simulators do not normally model a processor's internal state, including the PC, as accurately as an interpretive simulator for performance reasons.

Compiled simulators can only be used if all of the program code to be simulated can be identified at the time translation is performed. In other words, the target program code must be statically discoverable in order for it to be successfully simulated. This pre-condition excludes simulation of target programs which are self-modifying. Multi-tasking OSs can not be simulated as different processes may occupy the same address space. Operating system simulation is further complicated by the need to model asynchronous events such as interrupts.

Most embedded systems rely on some form of OS to schedule workloads across multiple processor cores and to control peripheral devices. Although compiled instruction set simulators are much faster than interpretive simulators, their use is restricted to stand-alone programs, which is not sufficient to model the complex hardware/software interfaces present in modern embedded systems.

The process of static binary translation is complicated by the existence of instructions and data within the same address space, and by the presence of indirect branches. Control-flow and register analysis are issues which static binary translation based simulators have to address in much the same way as disassemblers and compilers. The initial parsing of a target executable may not be able to resolve all instructions and data during translation. Hybrid static binary translation based simulators overcome any restrictions by calling a fallback interpreter to emulate target instructions, which for whatever reason, were not previously identified during translation or have been modified.

2.3.2 Dynamic Binary Translation

The processes involved in dynamic binary translation are shown in figure 2.6. The front-end is invoked at runtime to decode regions of target code which have not previously been translated. The decoded code is then optimized and translated into an intermediate representation. The back-end compiles the intermediate code into host code functions which are called to emulate the code sections. It may be discovered during emulation that certain host code functions lie on a critical path. In this case, the corresponding sections of code may be re-translated using a more aggressive optimization policy.

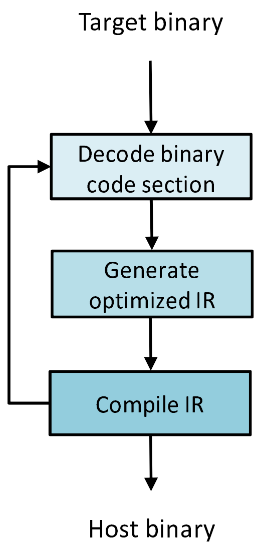


Figure 2.6 Processes involved in Dynamic Binary Translation Dynamic binary translation is performed at runtime.

Simulators which employ dynamic binary translation can emulate any type of application, including full operating systems, and are almost as fast as static binary translation based simulators. A DBT based simulator translates frequently executed sections of the target code - typically basic blocks - into code which when executed on the host machine emulates the same instructions within the simulation environment. High speed simulation is achieved by combining DBT with translation caching. If self-modifying code is detected at runtime, any translations which emulate the modified program region are discarded. Binary translation is a processor intensive task which can slow down simulation significantly on the first simulation run. To reduce the translation overhead some simulators perform emulation and translation in parallel.

Figure 2.7 shows a basic implementation of the main simulation loop for a DBT based simulator. The simulator calls the `fetch_translation` function which searches the translation cache to see whether a translation exists for a basic block with start address equal to the PC. If a translation exists, the pointer to the translated function (`trans_func`) is returned to the main loop. The translated function is then called to emulate the block at high speed. A pointer to the processor state is passed to the translated function so that it can update any status flags and registers whilst emulating the instructions within the block.

```c
while (!end_of_simulation) {

    trans_func = fetch_translation(cpu_state->PC);

    if (trans_func)
        (*trans_func)(cpu_state);
    else
        interpret_block(cpu_state);

    if (end_of_epoch)
        perform_translation();

}
```

Figure 2.7 DBT Simulator Code. This figure shows the C code skeleton for the main loop of a basic block DBT based simulator.

If a translation does not exist for the basic block, the `fetch_translation` function returns a `NULL` pointer. The simulator then calls the `interpret_block` function which interprets the basic block and maintains a profile of how many times the block has been emulated. The simulator continues emulating consecutive basic blocks in this manner for a fixed number of blocks - the simulation epoch. At the end of each simulation epoch, the `perform_translation` function is called which scans the basic block profiles to identify those blocks which were frequently executed. After translating the hot blocks, the function then adds the newly created translated functions to the translation cache.

A DBT based simulator can also gather profiling information (control-flow, register contents) on the target program whilst simulating it, something which is not possible with a static binary translation based simulator. This means that frequently executed code regions may be sent for highly optimized translation. Dynamic binary translation is lazy. This is an advantage as it guards against translating sections of the target binary which are never executed or which contain only data.

2.4 Sampling

Sampling is a technique which is used to speed up cycle-accurate simulation. A sampling based simulator collects accurate simulation data for a small subset (sample) of the entire benchmark simulation period (population), fast-forwarding through the remainder of the benchmark. Statistical analysis is then performed on the data collected to produce approximate figures for the simulation. During fast-forwarding, full observability is typically not supported, therefore sampling is not suitable for performing hardware/software co-design.

The Sampling Microarchitecture Simulation (SMARTS) framework [Wund 03] uses statistical sampling. It has been shown to speed up the simulation of 8-way and 16-way out-of-order processors by a factor of 35 and 60 times respectively compared to full cycle-accurate simulation. SMARTS can calculate the clock cycles per instruction (CPI) to within $\pm3\%$ for 41 of the SPEC2000 benchmarks. SMARTS applies statistical sampling theory to work out the optimal sampling strategy that will capture a programs' variability and produce results with the required degree of accuracy. The sampling strategy requires taking a large number of small samples from the population. By selecting a minimal, but representative sample, the nature of a particular benchmark can be accurately modelled.

SMARTS samples a tiny fraction of a benchmark's execution stream using detailed cycle-accurate simulation. The rest of the time it fast-forwards through the benchmark using functional simulation. The desired micro-architectural data is collected during sampling, whereas only the program-visible architectural state is updated during fast-forwarding. Systematic sampling is used where the samples, which consist of a relatively small number of consecutive instructions, are separated by sampling intervals, which consist of a large number of consecutive instructions.

Whilst SMARTS maintains the processor state between samples with functional simulation, the state of the system micro-architecture is left to become stale. If the micro-architecture state is not up-to-date prior to sampling then large errors appear in the detailed data collected. To combat this, the micro-architecture state is updated by the inclusion of a cycle-accurate warm-up period just prior to sampling. However, it is difficult to know how long to make the warm-up phase as some micro-architecture states may require many simulation cycles before they are representative of the true cycle-accurate states.

Another sampling technique, SimPoint [Sher 02], can summarize the large-scale behaviour of programs relatively quickly. It achieves this by offline analysis of the basic blocks within large representative sample traces - 100 million instructions - taken from the program trace. The assumption is that samples with matching dynamic basic block profiles exhibit similar behaviours. However, SimPoint does not provide a formal method for quantifying the accuracy of the results returned.

2.5 Summary

This chapter outlines the main simulation techniques used in instruction level and cycle-accurate simulation of microprocessor systems. Whilst interpretive simulation is flexible, in that it provides observability and can simulate any target binary, it is very slow at performing instruction level simulation compared to the different binary translation based simulation techniques. Dynamic binary translation is the best simulation technique for performing instruction level simulation as it is not only very fast, it also provides state observability and can simulate self-modifying programs. This makes DBT based simulation ideal for carrying out high level DSE and for performing hardware/software co-design and verification.

Sampling based simulation techniques are superior for performing cycle-approximate simulation. Sampling based simulation is many times faster than interpretive or binary translation based simulation techniques as it fast forwards through the majority of the simulation, needing only to simulate very small sections of the target program in detail. The inaccuracies introduced are small and can in some cases be quantified which makes sampling ideal for carrying out low level DSE in situations where there is a very large design-space. However, DBT based simulation remains the best simulation technique for performing cycle-accurate hardware/software co-design and verification of microprocessor architectures.

Chapter 3

Related Work

This chapter describes the simulation techniques used in previous work which are relevant to the field of high speed binary translation based simulation.

3.1 Binary Translation Simulators

This section looks at the translation techniques employed in static and dynamic binary translation based simulators [Cifu 96].

3.1.1 Static Binary Translation

The first static binary translation simulators were used to port legacy software across to newer, faster RISC based systems [Patt 85, Stal 90]. Static translators operate like compilers, translating the target binary into an equivalent host code binary image. Compiled simulators spend most of their time emulating target instructions and are consequently much faster than interpretive simulators. Interpretive simulators are slow because they spend most of their time, in the main simulation loop, fetching and decoding each instruction. Even if interpretive simulators employ a decode cache, the emulation of instructions is still slow.

3.1.1.1 HP Object Code Translator

When Hewlett Packard released its MPE XL operating system for its new HP Precision Architecture (RISC) series of computers it incorporated a Compatibility Mode (CM) environment [Berg 87]. The CM environment enabled program binaries from the previous family of HP 3000 computers (stack-orientated CISC, MPE V operating system) to run on the Precision Architecture platform. The CM environment uses two subsystems: the HP 3000 emulator and the static binary translator, called the HP 3000 Object Code Translator (OCT).

The emulator is capable of running HP 3000 binary code on HP Precision Architecture platforms without modification. However, the OCT first translates the HP 3000 binary code into native code which is then executed. The OCT binary translator can simulate HP 3000 programs up to five times faster than the emulator.

The OCT translates HP 3000 binary code segments into native code modules. The translator also tries to discover all of the node points within the program code and creates a node mapping table. The node mapping table holds the translated code addresses, within the modules, which correspond to the node addresses within the code segments. When a branch target address can not be statically determined it is looked up at runtime in the node mapping table. If a target address is not found within the node mapping table the emulator is invoked until the PC value equals a module entry-point at which point execution is returned to the translated code.

3.1.1.2 Hunter Systems DOS to Unix Translator

Hunter Systems used object code translation to port MS-DOS binaries (8086) into executable files which run on UNIX systems [Hunt 89, Wirb 88]. A number of different translator back-ends made translation to different host architectures possible. However, the program analyser required manual intervention in order to deal with complex code, such as self-modifying code, and to compute indirect jump target addresses.

3.1.1.3 Tandem Accelerator Object Code Translator

Tandem wanted an easy way to migrate software from its proprietary TNS CISC machines to its new TNS/R RISC machines based on the MIPS processor. The OCT developed by Tandem, called the Accelerator [Andr 92], enabled all existing TNS software to be run immediately and at high speed on the TNS/R machines. The Accelerator was also used to translate Tandem's Guardian 90 operating system and produce the first RISC release. This contributed to bringing Tandem's new RISC machines to market many years earlier than would otherwise have been possible.

The Accelerator emulates TNS CISC binary programs on TNS/R RISC machines by using a combination of translation and interpretation. It augments the target binary with translated code sections and a PMap table, which is a map of CISC to RISC instruction addresses, in advance of simulation. The Accelerator acts like any optimizing compiler except that it tightly controls TNS/R register and stack frame usage so that it can easily switch between accelerated and interpreted simulation modes.

After disassembling the CISC (TNS) target binary the Accelerator performs static control-flow analysis in which it attempts to identify all of the branch paths. Jumps through pointer variables or calculated addresses are explicitly marked and if the target address is unknown at runtime - not found in a PMap table - a switch is made to the interpreter. The Accelerator translates the CISC instructions within each basic block, on a per CISC subroutine basis, into a preliminary sequence of RISC instructions. CISC subroutine calls lookup the target address in a jump table which is replaced by a direct jump into translated RISC code. Returns back to the caller must also be looked up in the PMap table. Standard optimization techniques are then applied to the translated code within and across the basic blocks, including reordering the instructions within each block to minimize pipeline stalls.

Four different programs (TAL compiler, TAL-coded Dhrystone, Axcel and ET1) were used as benchmarks to measure the performance of the Accelerator. The benchmarks were run natively on a NonStop Cyclone, 22.3MHz superscalar CISC machine and compared with OCT emulation of the same benchmarks on a NonStop Cyclone/R, 25 MHz machine. The average benchmark simulation speed was 78% of the average native execution speed, and the simulation speed for the Axcel benchmark was 8%

faster than native execution.

It was shown that translated code ran 5 to 8 times faster than interpretation and that interpretation accounted for less than 1% of the emulation time. On average, the number of RISC instructions generated per CISC instruction was 1.6, and the accelerated code file (CISC binary plus translated code plus PMap) was 5 times larger than the original CISC binary.

3.1.1.4 Digital VEST Binary Translator

In 1988, Digital wanted to run legacy code which had previously been executed on its VAX machines [Brun 91] on its latest Alpha AXP processor [Site 93a, Site 95], but it was not simply a case of recompiling the applications for the new architecture. Large and complex applications typically rely on a spectrum of different OS libraries and services, and the time required to rebuild everything from scratch would have been prohibitive. It was therefore necessary to run as much as possible in the old environment, with system calls being redirected to the newly ported OpenVMS AXP [Kron 93] operating system. The Alpha AXP team decided to use static binary translation to enable not just their existing VAX code base, but also their MIPS code [Kane 88] code base, to be run on the Alpha processor.

Digital developed the VAX Environment Software Translator (VEST) binary translator to translate an OpenVMS VAX binary image into a OpenVMS AXP binary image [Site 93b]. VEST disassembles the VAX code starting at standard entry points, such as global sub-routines, and traces the program building-up a control-flow graph of basic blocks. After analysing the CFG, VEST generates an optimized host binary. The mapping between architectures is simplified by the fact that the AXP processor has more registers than the VAX processor. VAX condition codes, which are not implemented in the AXP processor, are mapped on to spare AXP registers. Each VAX instruction gets translated into zero or more AXP instructions.

VEST inserts jump instructions within the host binary to emulate direct branches and jumps. However, in order to emulate branches and subroutine calls to unknown target addresses, VEST inserts calls to a runtime look-up routine. The routine uses a look-up table which maps VAX instruction addresses to the corresponding translated Alpha

AXP instruction addresses. If the destination address is found in the look-up table then control is passed to the corresponding address in the host code. If it is not found, control is passed back to the runtime environment.

The Translated Image Environment (TIE) is the runtime environment which executes the translated image. The TIE employs open-ended translation and emulates the OpenVMS VAX environment by using wrappers to map library and system calls to the corresponding OpenVMS AXP calls. Target binary instructions which were either not discovered, or which did not exist at translation (self-modifying code), are caught and then simulated by TIE's built-in interpreter.

Digital used binary translation as an interim solution to enable users to run existing VAX/MIPS binaries on the Alpha processor with minimal effort. Over time, all legacy applications and dependent libraries were ported over to the new platform. By utilizing binary translation, Digital were able to run translated applications on Alpha AXP systems as fast, or faster, than the original applications ran on VAX systems.

3.1.1.5 Digital FreePort Translator

FreePort Express [Free 95] is a free program developed at Digital Research which translates SunOS 4.1.x user-mode binaries into executable files which can be run on DEC Unix 3.0 and later systems. It was the first translator from Digital which converted binaries from a non-Digital OS platform. FreePort Express translates the target binary prior to execution and incorporates a fallback interpreter. It was first demonstrated at SunWorld '95 where translated SunOS applications were shown to run as fast, or faster, on an AlphaStation 400 4/233 system than natively on a SPARC 20/71 system.

3.1.1.6 Digital FX!32 Emulator

Digital developed the FX!32 emulator [Thom 96, Hook 97a, Cher 98] in order to increase the popularity of its Alpha RISC platform by ensuring that a large number of applications would be available to run on it. FX!32 enabled all x86 32-bit Windows NT 4.0 programs to be run on the Alpha Windows NT 4.0 system. The FX!32 was the first emulator to use a combination of interpretation and profile-directed translation to provide fast simulation [Hook 97b] of x86 programs on the Alpha platform. The translation of x86 code into native Alpha code is performed in the background. Parallel translation means that the translated code can be optimized for speed without affecting the simulation speed.

The FX!32 runtime is started automatically whenever an x86 executable is run. The runtime loads the x86 image into memory and then calls the emulator which interprets the whole program the first time it is run. At the same time the emulator generates profile data on CALL instruction target addresses, and source and target address pairs for indirect jumps, which it stores in a database for use later by the translator.

The translator uses the execution profile information gathered during emulation to translate the target binary into a collection of native code images. The unit of translation is the assembly code routine. The translator divides the target image into separate routines that have entry points at each call target address. The routines are created using the control-flow profile information which includes known target addresses for indirect jumps. A routine is a collection of one or more regions which consist of a contiguous set of instructions. Direct entry is permitted to any region within a routine.

A hash table is generated which maps target binary addresses to entry points within the translated routines. If the emulator finds that the next instruction address is mapped to an entry point, the corresponding translated routine is called. As it is generally impossible to statically analyse all program execution paths, the emulator is invoked as a backup when no translated routine mapping exists for a target address.

Digital's FX!32 emulator transparently emulates x86 binaries on the Alpha platform at high speed. The performance of a set of x86 benchmarks running on a 200MHz Intel Pentium Pro and a 500MHz Alpha system under FX!32 were compared. The results showed that the x86 applications ran as fast on the Alpha (second emulation run) as they did on the Intel machine.

3.1.1.7 Ultra-fast Instruction Set Simulator

A number of research groups are now developing retargetable instruction set simulators. The Ultra-fast Instruction Set Simulator [Zhu 99, Zhu 02] improves the performance of statically compiled simulation by aggressively utilizing low level machine resources to take full advantage of the host architecture. The low level simulation techniques were shown to increase the simulation speed by a factor of 2.7 on average over traditional compiled simulation techniques which generate C code.

3.1.1.8 Static Scheduling Simulator

The static scheduling simulation technique [Brau 01] applies static compilation to instruction decoding and instruction scheduling in retargetable simulators. Whilst static instruction scheduling increases the simulation speed of cycle-accurate simulators it also restricts flexibility of operation. Compiled simulators were generated from model descriptions of TI's TMS320C54x processor (cycle accurate model) and the ARM7 processor (functional model). A FIR filter was used to benchmark both processor models running on an 800MHz Athlon PC. The simulation results for the TMS320C54X processor, showed that static scheduling led to an increase in speed by almost a factor of 4 compared to dynamically scheduled simulation. Static scheduling resulted in a speed-up by a factor of 7 for the ARM7 processor, from 5 MIPS to 35.5 MIPS.

3.1.1.9 JIT-CCS Simulator

Just-In-Time Cache Compiled Simulation (JIT-CCS) [Nohl 02, Brau 04] can be used to create retargetable functional and cycle-accurate simulators. The JIT-CCS simulator references an array of built-in, pre-compiled instruction functions which emulate the behaviour of the different target instructions. When a target instruction is decoded a reference to the corresponding compiled instruction function is stored in the translation cache. If the simulator finds a matching translation cache entry for the next target instruction it calls the corresponding instruction function. If the simulator detects that an instruction has been modified, it decodes it and then calls the corresponding compiled instruction function.

Simulation results for cycle-accurate simulation of an ARM7 processor showed that JIT-CCS simulation is four time faster than interpretive simulation and only 5% slower than compiled simulation. The simulation performance results for the jpeg200 codec benchmark were: compiled simulation 7.2 MIPS; JIT-CCS simulation 7.0 MIPS; interpretive simulation 1.8 MIPS.

3.1.1.10 IS-CS Simulator

The Instruction Set Compiled Simulation (IS-CS) simulator [Resh 03, Resh 09] was developed as a fast and flexible functional simulator. In order to achieve high speed simulation, the time consuming instruction decode process is performed during the static compilation stage. The simulation engine checks to see whether the next instruction is valid before it calls the corresponding translated instruction function. If an instruction has been modified the binary code at the PC address is decoded and the instruction interpreted by a generic emulation function. As the number of instructions modified in most simulations is very small the slowdown in simulation speed is minimal. Performance is further increased by a technique called instruction abstraction which produces aggressively optimized decoded instructions.

Simulations of the adpcm and jpeg benchmarks were run on a model of the ARM7 processor. IS-CS was able to simulate adpcm and jpeg at speeds of 11.2 MIPS when running on a 1GHz P3 host machine.

3.1.2 Dynamic Binary Translation

The poor performance of interpretive simulators and the lack of flexibility inherent in compiled simulators has led to active research in the field of DBT based simulation. Dynamic binary translation based simulation takes advantage of the fact that programs typically spend 90% of their execution time in only 10% of the code. This means that the cost of compilation can be amortized over the duration of a simulation - even on the first run - by caching the translations. The latest DBT emulation techniques are outlined in the following sections.

3.1.2.1 MIMIC Simulator

The MIMIC simulator [May 87] simulates IBM System/370 instructions on the IBM RT PC RISC machine. The MIMIC simulator was developed so that important programs, written mainly in System/370 assembly, could be run on an RT PC workstation with minimal effort. A process VM was created on the RT PC to emulate the System/370 application environment. To increase performance all system calls are mapped to native OS calls. However, instructions which invoke OS services directly, such as the Supervisor Call, are translated to call a host code wrapper which then calls the equivalent native OS service.

MIMIC takes a group of target instructions, called a code block, and translate them as a unit. A code block consists of one or more connected basic blocks which may be contiguous or disjoint. If the code block is larger than a basic block, complex flow control analysis may be necessary. Each code block is analysed and translated just before it is executed.

The MIMIC translator works in two stages. The first stage analyses each target code block and the second stage generates the RT PC host code for the code block. MIMIC utilizes three different data structures during simulation. The S/370 binary is loaded into the Source Memory data structure and the translated host code blocks are stored in the Target Memory data structure. The Intermediate Memory data structure maintains a mapping between the target program addresses in Source Memory and the corresponding translated code blocks in Target Memory. Each Intermediate Memory address therefore passes simulation control over to the associated translated code block or to the translator when no translated code exists.

Each translated code block consists of one or more prologs, a main body and an epilog. A prolog exists at the entry point to a code block, This enables control to be transferred to the next target instruction within the translated code block. The epilog exits the code block and jumps to the pointer target in Intermediate Memory for the next instruction address. This will either call the next translated code block or the translator.

Performance results for MIMIC are from the simulation of two large S/370 programs, EXEC 2 and CIPHER. The quality of the translated code was judged by the expansion factor, which was 4.25 for EXEC 2 and 2.7 for CIPHER.

3.1.2.2 Shade Simulator

Shade [Cmel 94, Hsu 89] is a fast instruction set simulation tool which includes a flexible trace generation facility for the analysis, design and tuning of hardware and software systems. Whilst statically translated code can simulate and trace programs at high speed, it is incapable of tracing self-modifying or dynamically linked code. Shade achieves both high performance and comprehensive tracing by dynamically translating code which simulates and instruments the target programs. Shade runs on Sun SPARC systems and can simulate SPARC (V8 and V9) and MIPS I ISAs.

Shade dynamically translates target instructions up to the next control instruction. The host code fragments generated emulate the target block and perform any profiling when called. The translated code fragments are chained together (direct branches only) so that control can pass from one block directly to the next without needing to return to the main simulation loop. Any memory references are replaced with calls to the target memory model.

Each target instruction address is mapped to the corresponding translated code fragment by a Translation Look-aside Buffer (TLB). The simulator performs a lookup, first in a fast partial TLB, and then in the full TLB to see whether a translated fragment exists for a current PC address. If both lookups fail, the translator is invoked to create a translated fragment for the current block which is then stored in the translation cache. If the tracing strategy is changed, or the program code modified during simulation, the TLB and translation cache are flushed.

The performance figures presented are for a subset of the SPEC89 benchmark suite running on a SunOS 4.x, SPARC V8 platform. On average, Shade simulates V8 integer and floating-point binaries 6.2 and 2.3 times slower respectively than they run natively. SPARC V9 integer and floating-point binaries were simulated 12.2 and 4 times slower respectively than they run natively.

3.1.2.3 Embra Simulator

Embra [Witc 96], which runs as a subsystem in the SimOS [Rose 95] simulation environment, accurately models MIPS R3000 and R4000 uni-processor and multiprocessor systems. Embra is a flexible high speed simulator which can be used for research and development into operating systems and computer architectures. It deploys DBT to generate code sequences which simulate the workload, model system components and gather simulation statistics.

Embra translates each basic block it encounters into a host code segment which it then executes to emulate the target instructions within the block. New host code segments are stored in a translation cache and references to them maintained in a hash table. If the next PC address hits in the hash table, the corresponding host code function is called to emulate the block. If the next PC address misses, the translator is invoked to translate the target block. Consecutive basic blocks are chained together to avoid returning to the main simulation loop. Embra supports self-modifying code by flushing the translation cache and hash table on detecting a write to a previously translated page.

Embra can customize the translated code generated to model different machines (memory configurations). The detail and type of profiling information captured may be changed during simulation. This enables fast-forwarding through uninteresting parts of the workload and is useful when simulating large applications.

Embra is a system simulator which models the R3000's MMU used to translate virtual addresses to physical addresses. It supports multiple virtual address spaces so that operating systems and multiple processes may be simulated. Embra can be operated in one of three different simulation modes: Base mode is the fastest mode and uses 6000 cycle processor interleaving; Cache mode accurately models the target memory hierarchy and uses 80 cycle processor interleaving; Parallel mode simulates each target processor on a different host processor.

An SGI Challenge, 150MHz four processor (MIPS R4400) machine running IRIX 5.3 was used to evaluate the performance of Embra. A subset of the SPEC92 [Dixi 92] benchmark suite running under IRIX 5.3 was simulated to ascertain the uni-processor simulation performance. The average slowdown in simulation, compared to native execution, was 5.8 for Base mode and 11.4 for Cache mode. In Base mode, the simulation speeds ranged from 11.1 to 20 MIPS, with floating-point benchmarks executing faster than integer benchmarks. A subset of the SPLASH-2 [Woo 95] benchmark suite running under IRIX 5.3 was used to ascertain the simulation speed of a four processor, shared memory, multiprocessor system. The average slowdown in simulation, compared to native execution, was 13 for Base mode, 99.2 for Cache mode and 6 for Parallel mode.

3.1.2.4 Simics Simulator

Simics [Magn 98, Magn 02] is a commercial, user level and full system simulator. It is capable of modelling target system behaviour at two levels of abstraction, functional and timing approximate. Timing approximate simulation is achieved by interfacing Simics with more detailed hardware models [Wall 05]. Simics supports the development and testing of both hardware systems and software. The target system is defined using objects to represent components such as processors, memory, network cards, graphics cards and disks. The system state can be inspected by single stepping through the simulation or by setting breakpoints. Simics can also model a range of different multi-processors, including out-of-order cores, and run a number of different operating systems.

A range of different operating systems were booted-up under Simics - running on an Intel P-III, 933 MHz host platform - to test its performance. The simulation speeds ranged from 2.1 MIPS for the boot-up of Windows XP running on an x86 P-II target system, to 9.3 MIPS for booting-up Linux running on a PowerPC-750 target system. Simulations were also performed for a multi-processor target system with Simics running on an UltraSparc III, 750 MHz host platform. The results for the boot-up of Solaris 8 on an Ultra II Enterprise server target system show that the MIPS/CPU decreased from 6.62 for a single processor down to 1.25 for a 30 processor system.

3.1.2.5 QEMU Simulator

QEMU [Bell 05, Bart 06] is a fast, instruction level simulator which can model a range of different target processors and perform system and process level simulation. QEMU is straightforward to port between different host machines as all necessary compilation is performed at the time the simulator is built. The simulator forms a translated block for each basic block it encounters and places it in a 16MB translation cache which is simply flushed when full. It maintains a page cache which records which physical pages are write-protected. This enables QEMU to simulate self-modifying code. On detecting a write to a read-only page, QEMU invalidates all translations for the page and enables write access.

QEMU uses an original dynamic translator. Each target instruction is divided into a simple sequence of micro operations which are implemented in C code. The set of micro operations are pre-compiled by `gcc` offline - at build time - and then placed into an object file. During simulation, the code generator accesses the object file and concatenates micro operations to form a host function. When called, the host code function emulates the target instructions within the block. To increase the simulation speed, translated blocks which are known to follow one another are directly linked together. However, these links must be reset when the MMU address mappings change.

User level simulation of the Linux BYTEmark benchmarks resulted in a slowdown by a factor of 4 for integer code and by a factor of 10 for floating point code over native execution. System level simulation resulted in a slowdown by a factor of 2.

3.1.2.6 Simit-ARM Simulator

Simit-ARM [Qin 06, DErr 06] is a fast, instruction level DBT based simulator which distributes the tasks associated with translation across multiple processors. The simulator offloads the translation process to the other cores and continues to emulate the application interpretively. This means that no delays are experienced when running interactive applications such as operating systems.

Whilst interpreting target instructions the simulator identifies frequently executed pages for translation. A page is defined as a contiguous block of 512 words. When the simulation count for a page exceeds a predefined threshold, the program code for the page is translated into a C++ page function. The page function is then compiled by `gcc` into a shared library and linked with the simulation engine at runtime. If the next PC address is within the address range of a translated page, the corresponding host page function is called with the PC passed as an argument. The host page function emulates the target instructions within the page starting at the PC address. Simulation is controlled within a page function via a `switch` statement until the execution flow exits the page. A host page function may emulate many thousands of target instructions in a single call. Simit-ARM can perform process and system level simulation as well as emulate self-modifying code.

Simulation Results for the SPEC CINT2000 benchmark suite, running on a four processor, 2.8GHz P4 machine, show an average simulation speed of 197 MIPS for the

MIPS32 ISA (Simit-MIPS simulator) and 133 MIPS for the ARM v4 ISA (Simit-ARM simulator).

3.2 Summary

Most of the simulation techniques covered in this chapter translate either small sections of the target program (typically individual instructions or blocks of instructions) dynamically or larger sections statically. In addition, very few simulators support cycle-accurate simulation which is needed to perform low level DSE. Whilst Simics and Embra DBT based simulators can be used to perform DSE of hardware systems, they are relatively slow simulators.

Simit-ARM uses a page as its translation unit with entry being permitted to any instruction address within a host page function. This means that the compiler is restricted in the optimizations that it can perform across basic blocks. Also, pages which contain mostly data, or in which only a small region of code is executed, may get translated.

QEMU, Shade and Embra chain together translated basic blocks which they know follow one another. However, the basic blocks are still translated separately and the compiler is not presented with the opportunity to optimize the code generated for speed across multiple blocks. In the case of QEMU, the code produced is not even optimized across a single block as the instruction micro operations are pre-compiled and then combined at runtime.

The research presented in later chapters looks at techniques which identify and translate larger sections of the target program at runtime in order to increase the simulation speed.

Chapter 4

Edinburgh High Speed Simulator

This chapter presents the Edinburgh High Speed simulator and details its modes of operation, capabilities and performance enhancing structures. The advanced LTU DBT simulation techniques investigated in this research and incorporated into the simulator are described in later chapters.

4.1 Overview

The Edinburgh High Speed (EHS) simulator [Toph 07] is a high performance research simulator developed at the Institute for Computing Systems Architecture at the University of Edinburgh. The simulator can perform user-level (emulated system calls) and system-level simulation. It can be run in either instruction level or cycle-accurate simulation modes and is capable of switching between these two simulation modes at runtime. The simulator is target-adaptable and currently models the ARC 700$^{\text{TM}}$ processor which implements the ARCompact instruction set architecture [ARCo].

The simulator operates in either interpretive or DBT based simulation modes. Dynamic binary translation based simulation is a hybrid form of simulation in which the simulator alternates between performing interpretation and DBT based simulation. Interpretive simulation provides precise observability of the processor state after each instruction and DBT based simulation provides precise observability at translation unit boundaries.

The EHS simulator models a complete computer system including the processor, its memory sub-system and sufficient interrupt-driven peripherals to simulate the boot-up and interactive operation of a Linux based operating system. In contrast to other high speed instruction level simulators a precise view of the target processor state is maintained. This allows the simulator to be used as a software development platform as well as a tool for functional verification of customized processors derived from the ARC 700 baseline processor.

In common with conventional interpretive simulators, such as SimpleScalar, the interpretive simulation mode repeatedly fetches, decodes and then emulates successive instructions in the execution path. Registers, memory and the context of I/O devices are updated as instructions commit in order to maintain a precise view of the target system.

The DBT simulation mode combines the speed of compiled simulation with the flexibility of interpretive simulation. This means that all binaries can be simulated and at high speed. When running in this mode the simulator initially operates interpretively, discovering and profiling basic blocks as they are emulated. The simulator periodically examines the target program's execution profile looking for frequently executed basic blocks which are then marked for binary translation. Once a basic block has been translated, it will from that moment on be emulated by calling the corresponding translation.

The underlying simulator components which handle memory access, I/O, interrupts and exceptions are the same whether the simulator is operating in interpretive or DBT based simulation mode. This facilitates seamless switching between the different simulation modes at runtime. Dynamic binary translation based emulation of a basic block may be terminated on any instruction and simulation restarted at the current program counter. This enables translated blocks to raise exceptions part-way through, after which the remaining instructions in the block will be interpreted.

The EHS simulator is written in C and C++ and incorporates a number of standard performance enhancing structures such as instruction decode and translation caches. The simulator can retain the translations generated during simulation of a given binary for reuse when simulating the same executable. The maximum simulation speed is observed when all target instructions emulated have been translated in previous simulation runs. Persistent translations enable a library of application translations to be built up for future use.

4.2 Simulator Features

The EHS simulator was designed for the purpose of researching novel micro-processor architectures. In order to be able to perform design-space exploration and verification effectively, the simulator must not only be fast, it must also provide flexibility of operation.

The key features of the simulator which make it suitable for design-space exploration are outlined below:

- **Fast Simulation**. The EHS simulator is capable of high speed instruction level simulation. Running in DBT simulation mode the simulator is as fast as other state-of-the-art simulators designed purely for speed. Faster simulation reduces the time scales for software and hardware design, testing and verification.

- **Instruction Level Simulation**. In instruction level simulation mode the simulator emulates programs at high speed and returns the instruction count. The simulator also incorporates an interface to connect it to hardware description language (HDL) generated simulators in order to carry out co-simulation.

- **Cycle-accurate Simulation**. In cycle-accurate mode the simulator returns the instruction count, the number of execution cycles and the number of hits and misses for each level of the memory hierarchy. This information is required to map the design space when searching for the most efficient processor and memory sub-system designs. The simulator is able to switch dynamically between instruction level and cycle-accurate simulation. This flexibility of operation enables different techniques, such as fast-forwarding and sampling (cycle-approximate), to be used to collect data during simulation. Testing new processor designs using cycle-approximate simulation reduces the time required to run individual simulations which facilitates detailed exploration of large design spaces.

- **Application and System VMs**. The simulator is capable of providing application and system level simulation. Applications can be run standalone with the simulator emulating Linux system calls, or an operating system can be run with the simulator modelling the standard hardware peripherals. System simulation enables realistic testing of embedded applications which typically run on top of some form of cut-down operating system.

- **State Observability**. The simulator maintains processor state observability enabling it to support hardware/software co-design, verification and debugging tasks. The state of the processor is updated as each instruction is emulated.

- **Target System Definition**. The simulator provides for comprehensive definition of the target system architecture. Target system configuration parameters include the core processor type; system clock speed; main memory and closely coupled memory address ranges; type, level, size, block size, associativity and replacement policy for caches; memory, closely coupled memory, cache and cpu data path widths and latencies; and branch predictor unit type.

- **ISA Configuration**. The number of cycles required to execute each target instruction can be configured in an ISA file. The simulator also provides for extension of the ISA through the addition of new instructions loaded in shared libraries.

- **Target Adaptable**. The modular design of the simulator means that it is relatively straightforward to swap one ISA for another.

- **Command Line Interface**. The simulator incorporates a command line interface which allows a simulation to be paused so that breakpoints can be set, instruction tracing activated, simulation check points created or the processor state displayed.

4.3 Normal Simulation Mode

In ‘normal’ interpretive simulation mode, the EHS simulator’s main loop fetches the next instruction opcode from memory, decodes it and then emulates the instruction updating the processor state. The average instruction level simulation speed is 30 MIPS and the average cycle-accurate speed is 12.5 MIPS (2.66 GHz Intel Core 2 Duo). Figure 4.1 outlines the operation of the EHS simulator’s interpretive simulation loop.

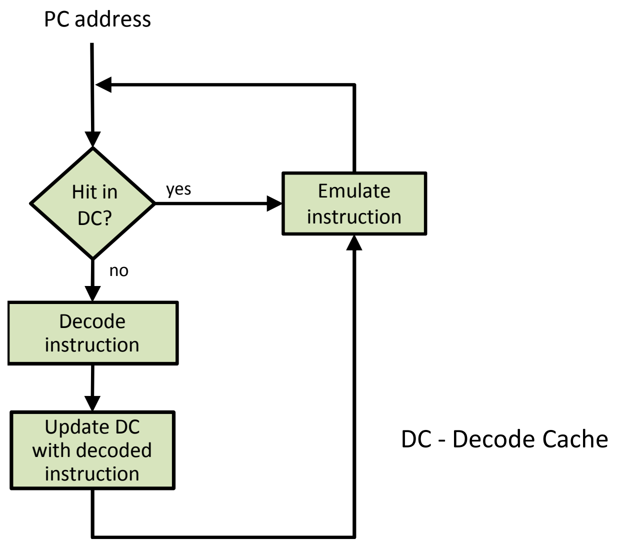


Figure 4.1 Interpretive Simulation Loop. This flow chart shows the EHS simulator’s interpretive simulation loop.

After each instruction is fetched it must be decoded so that its operation can be emulated. The overhead of instruction decoding is high accounting for over 90% of the total simulation time. This is due to the complex instruction encoding schemes employed in modern processors which are designed to hold a range of information. An instruction opcode may have encoded the instruction size (16/32-bit), instruction operation, instruction operands, address mode, data size and sign and whether another fetch is required to load immediate data. In order to streamline instruction decoding the EHS simulator checks each opcode fetched against the most frequently executed instructions first in an attempt to perform decoding as fast as possible.

Decode caches are widely deployed in simulators to minimize the cost of instruction decoding by storing previously decoded instruction information. Decode caches increase the simulation speed significantly as most programs are highly repetitive in nature during execution.

The EHS simulator also sets pointers within the decoded instruction object which directly references the instruction operands so that they can be accessed at speed when emulating the instruction. For example, the decoded information for the instruction `ADD r0,r30,r31` includes three pointers, two for the source operands (`r30`, `r31`) and one for the destination operand (`r0`). On decoding an instruction the pointers for each operand are set to reference the corresponding registers in the model of the processor register file. When the instruction is emulated the values of its operands can obtained and updated quickly by simply dereferencing the operand pointers (`*r0 = *r30 + *r31`).

4.3.1 The Decode Cache

When an instruction is decoded, the information (including any long immediate operand values) is stored in the decode cache. The decode cache in the EHS simulator is configurable in size (number of decoded entries) and associativity. By default the EHS decode cache is configured as a 2-way, 8K entry cache.

Before fetching the next instruction from memory the instruction address is looked-up in the decode cache. If the next PC address hits in the decode cache the previously decoded instruction information is returned with minimal delay. If the next PC address misses in the decode cache the instruction is fetched and decoded in the usual manner and the decoded instruction information stored in the decode cache. If the simulator detects self-modifying code all of the data stored in the decode cache is simply invalidated. The EHS simulator's decode cache typically experiences a hit rate above 98%.

4.4 Fast Simulation Mode

The EHS simulator may also be operated in one of its high performance or ‘fast’ DBT based simulation modes. Frequently executed groups of target instructions are translated into native code functions which when called emulate the same instructions at high speed. Dynamic binary translation based instruction level simulation is typically more than 10 times faster than interpretive simulation. The default unit of translation for the EHS simulator is the basic block. However, it may be configured to use larger translation units consisting of multiple basic blocks. LTUs and their implementation within the EHS simulator are described in detailed in chapter 6.

Figure 4.2 outlines the operation of basic block DBT based simulation as implemented in the EHS simulator. The simulator interprets a fixed number of blocks (1000 blocks by default) at the same time building up a profile of which blocks were executed and the number of times they were emulated. At the end of this period, called the simulation epoch, the block profile is analyzed in order to ascertain those blocks which were frequently executed. Blocks which were emulated more times than the translation threshold, a fixed number beyond which a block is considered hot, are marked for translation.

The hot blocks discovered are then translated into host code functions which can be executed directly on the host machine. The EHS simulator first generates C code functions to emulate the instructions within each of the hot blocks. It then invokes `gcc` to compile the C code functions and create a shared library containing the host code functions which it loads. All of these actions are performed during actual simulation of the target program.

Before the simulator starts to emulate the next block it checks to see whether it has previously been translated. If it has, the simulator simply calls the corresponding translated function which emulates the block directly. If not, the block is interpreted and profiled as normal. Whilst the cost of dynamic binary translation is substantial, it is amortized through the use of a translation cache which stores and then returns previously translated functions at high speed.

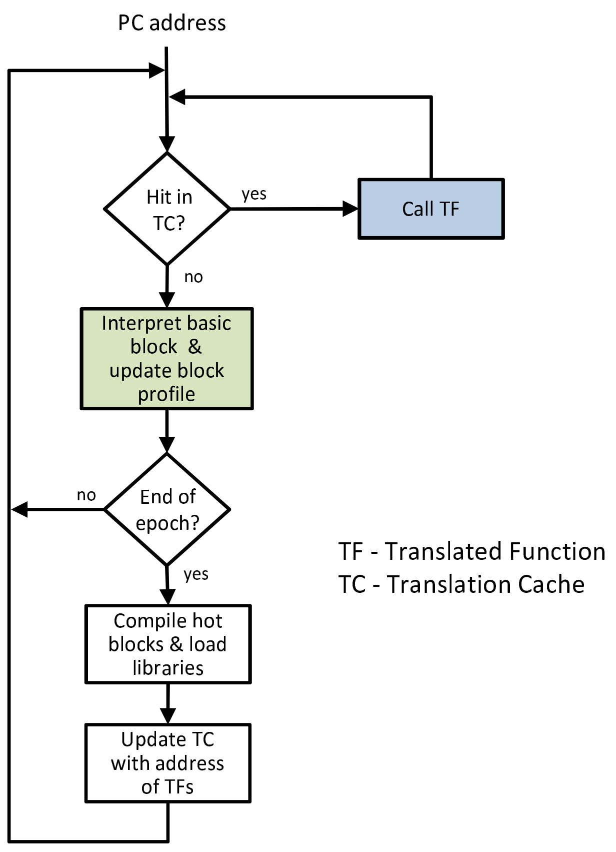

PC address
Hit in TC?
yes
Call TF
no
Interpret basic block & update block profile
no
End of epoch?
yes
Compile hot blocks & load libraries
Update TC with address of TFs
TF - Translated Function
TC - Translation Cache

Figure 4.2 DBT Simulation Loop. This flow chart shows a simplified version of the EHS simulator's DBT simulation loop.

4.4.1 The Translation Cache

In order to emulate the next basic block, the corresponding translated function needs to be found, if one exists. However, as it would be very time consuming to search of all the translations within the shared libraries looking for a match, all DBT based simulators incorporate some form of translation cache. A translation cache is used to locate the translated function for a given block with minimum delay. By default the EHS translation cache is configured as a direct, 8K entry cache.

Before checking the decode cache for the next instruction, the next PC address is looked-up in the translation cache. If the next PC address hits in the translation cache, the previously translated function for the basic block with that start address is returned to the simulator and then called. If the next PC address misses in the translation cache, the basic block is interpreted as normal. If the simulator detects self-modifying code all of the entries stored in the translation cache are flushed and the translations affected discarded. The EHS simulator's translation cache typically experiences a hit rate greater than 99%.

4.5 Instruction Level Simulation

The instruction level and cycle-accurate simulation models have been developed as separate components within the EHS simulator. This facilitates switching between these two modes at runtime. The instruction level simulation mode emulates programs at high speed and returns the instruction count.

4.6 Cycle Accurate Simulation

Cycle-accurate simulation models the processor pipeline [Henn 96] and memory subsystem in detail returning the execution cycles and cache hit and miss statistics.

4.6.1 The Pipeline Model

The EHS simulator currently models a 7-stage pipeline based on the ARC 700 32-bit processor as shown in figure 4.3. The pipeline model is called after emulating each instruction and it updates the cycle time of all the pipeline stages at this point.

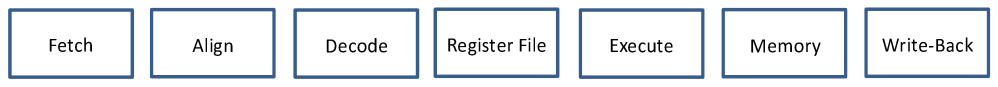


Figure 4.3 Processor Pipeline. This figure shows the 7 stages of the ARC 700 based processor pipeline.

The processor's pipeline consists of the following stages:

1. **Fetch**

   Fetches the next 32-bit word into the instruction buffer from memory. May stall the pipeline whilst fetching data.

2. **Align**

   Extracts the next instruction word (instructions are aligned on 16-bit boundaries) and performs some pre-decoding of register operand addresses and instruction size.

3. **Decode**

   Decodes the instruction opcode, identifying the instruction operation and any operands.

4. **Register File**

   Returns any register values used by the instruction from the register file. May stall the pipeline whilst updating source register values.

5. **Execute**

   Performs the instruction operation updating the processor state and any destination register values. The pipeline may be stalled during instruction execution.

6. **Memory**

   Performs load or store of register values from/to a memory address. May stall the pipeline whilst loading data from memory.

7. **Write-Back**

   Instruction commit.

4.6.2 The Memory Model

The cycle-accurate memory model returns the number of cycles taken for each instruction fetch and data load from memory. The memory model also returns the number of hits and misses for all levels of the memory hierarchy. In order to speed up cycle-accurate simulation the memory model incorporates a L1 front-end cache.

4.6.2.1 L1 Front-End Cache

The L1 front-end cache is a small software cache which is placed in front of the cycle-accurate L1 target cache [Hand 98] models (see figure 4.4) in order to speed up the simulation of instruction fetches and data accesses to and from memory. Cycle-accurate simulation of read and write requests involves a significant amount of processing, which has a negative effect on the simulation speed. This is because L1 caches are fairly complex to model, involving searching for specific blocks of data. The L1 front-end cache is a simple structure which reduces the overhead of modelling the L1 target cache/s by returning cycle-accurate information at greater speed. The L1 front-end cache typically provides a speed-up of 1.16 for interpretive simulation and 1.32 for DBT based simulation.

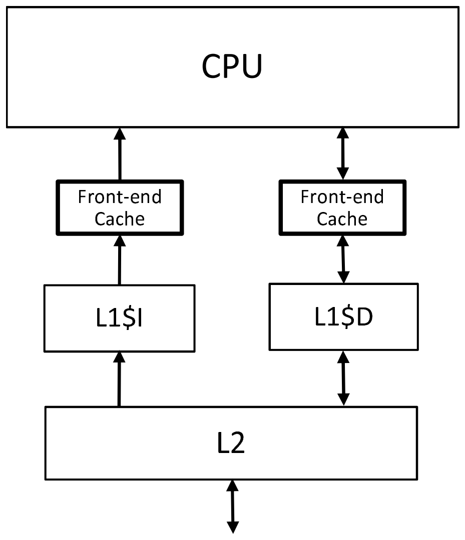


Figure 4.4 L1 Front-End Cache. Figure shows the logical placement of the L1 front-end caches between models for the target processor and L1 caches.

A cycle-accurate simulator must model all of the data accesses in detail, updating the hit and miss statistics for all caches and calculating the latencies of all read and write operations between the different memory levels. It has to take into account the cache block size, associativity, replacement policy and write strategy (write-through or copy-back) at every level.

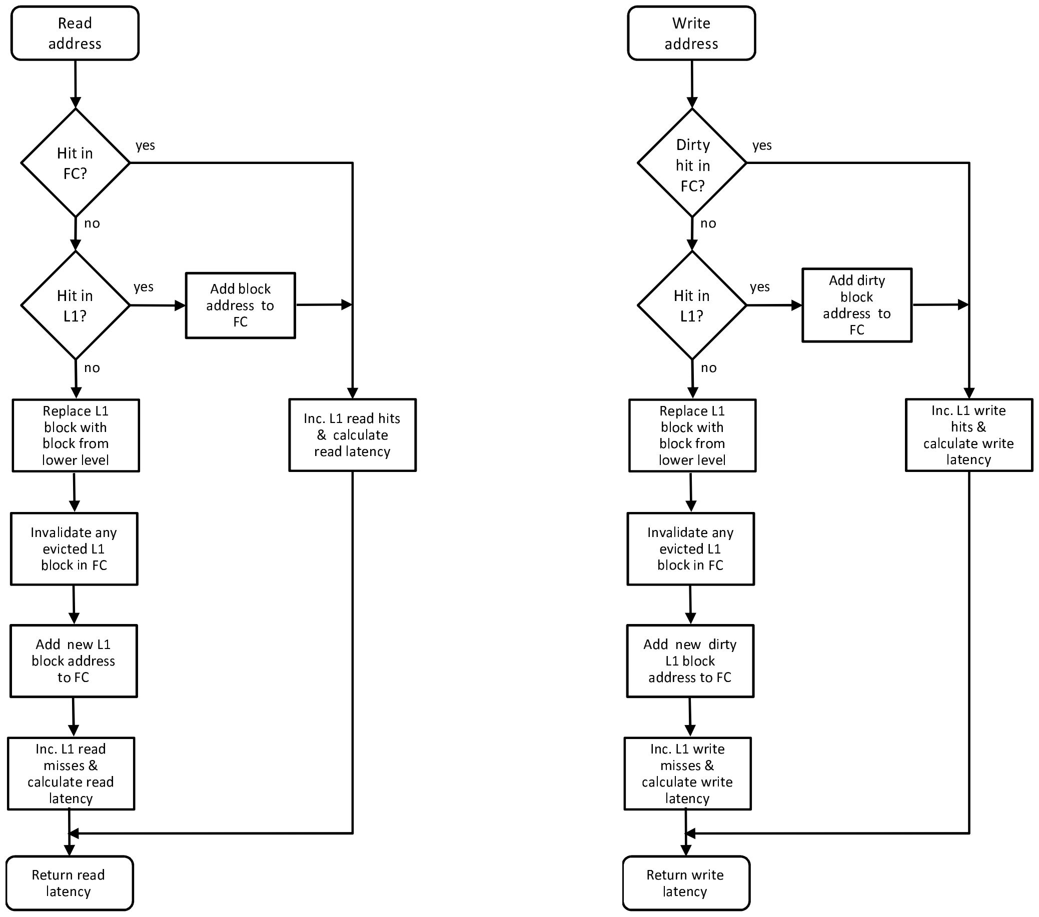
As programs exhibit a high degree of memory locality during execution the hit rate observed for L1 caches is typically very high. Most of the time spent performing cycle-accurate simulation of the memory hierarchy will be devoted to emulating L1 data accesses. Therefore, increasing the speed of the L1 cache models will result in faster simulation overall. The L1 front-end cache is designed to speed up return of L1 read/write latencies and updating of L1 hit statistics. In other words, front-end caches speed up the emulation of hits in the L1 target cache/s.

The front-end caches sit in front of the L1 target cache models and intercept the read and write requests from the CPU. Hits in the front-end cache are processed locally, whereas requests that miss are forwarded on to the L1 cache model. The front-end caches are small, direct-mapped, inclusive caches which operate at speed. The flow-charts in figure 4.5 show how read and write requests are processed by the front-end cache.

All read requests made by the CPU are intercepted and the data block address is looked up in the front-end cache. If there is a hit, the cycle-accurate model returns the read latency (cycles) and updates the number of L1 read hits. If the read request misses in the front-end cache, the request is passed on to the underlying L1 target cache model. If the read request hits in the L1 cache its block address is added to the front-end cache and the read latency returned.

If the read request misses in the L1 cache, it is forwarded on to the next lower memory level which processes it as usual. Once the request has been fulfilled with a block from a lower memory level it is stored in the L1 cache. Any block which is evicted from the L1 cache by the new block is also invalidated in the front-end cache. The new block address is then added to the front-end cache and the read latency returned.

All write requests made by the CPU are intercepted and the data block address is looked up in the front-end cache. If there is a 'dirty hit', in which there is a match for the PC block address and the block's dirty bit is set, the cycle-accurate model returns the write latency and updates the number of L1 write hits. If the write request misses in the front-end cache, the request is passed on to the underlying L1 target cache model. If the write request hits in the L1 cache the dirty bit is set, if not already set, and its block address is added to the front-end cache, the dirty bit set and the write latency returned.


Read address
Hit in FC?
yes
no
Hit in L1?
yes
Add block address to FC
no
Replace L1 block with block from lower level
Inc. L1 read hits & calculate read latency
Invalidate any evicted L1 block in FC
Add new L1 block address to FC
Inc. L1 read misses & calculate read latency
Return read latency
Write address
Dirty hit in FC?
yes
no
Hit in L1?
yes
Add dirty block address to FC
no
Replace L1 block with block from lower level
Inc. L1 write hits & calculate write latency
Invalidate any evicted L1 block in FC
Add new dirty L1 block address to FC
Inc. L1 write misses & calculate write latency
Return write latency

(a) Read Request (b) Write Request

Figure 4.5 L1 Front-End Cache Operation. The flow charts above show how Read and Write requests are processed respectively by the front-end cache.

If the write request misses in the L1 cache, it is forwarded on to the next lower memory level which processes it as usual. Once the request has been fulfilled with a block from a lower memory level it is stored in the L1 cache with its dirty bit set. Any block which is evicted from the L1 cache by the new block is also invalidated in the front-end cache. The new block address is then added to the front-end cache, its dirty bit set and the write latency returned.

4.7 System Simulation

The EHS simulator supports full system simulation by modelling the underlying hardware so that operating systems can be emulated. Modelling memory access for systems which incorporate an MMU (Memory Management Unit) [Henn 02] is a processor intensive task which slows down simulation. Memoization techniques are therefore employed to speed up simulation of read and write requests whilst accurately modelling system memory. All memory exceptions: misalignment errors, memory access violations and TLB misses, must occur in the same manner and at the same point in the simulated program as they would in the real system.

The translation lookaside buffer (TLB) translates a target virtual address to the corresponding target physical address or it raises a TLB miss. The EHS simulator deploys Page Translation Caches (PTCs) to cache target virtual page to host physical page address mappings which in turn model the target physical pages. Three different direct-mapped PTC caches indicate whether a page accepts Read, Write and Execute accesses. The PTCs speed up MMU simulation by bypassing TLB translation and by directly referencing the data in the host physical pages. Figure 4.6 shows the PTCs location within the MMU.

Each entry within a PTC holds the host physical page address mapping for a given target virtual page address and is valid if and only if:

- The target virtual page address is currently mapped in the TLB.
- The current process has permission to access the page.
- The target physical page is in normal external memory with no read or write side effects.

The read PTC enables the simulator to trap writes requests to read-only pages at the same time allowing full speed read and execute accesses to read-only pages. Self-modifying code is also trapped by identifying write requests to target physical pages referenced in the fetch PTC. On detecting self-modifying code the entry in the fetch PTC is removed and the decode and translation caches are flushed. Fetch requests to target physical pages referenced in the write PTC are also trapped and their entries

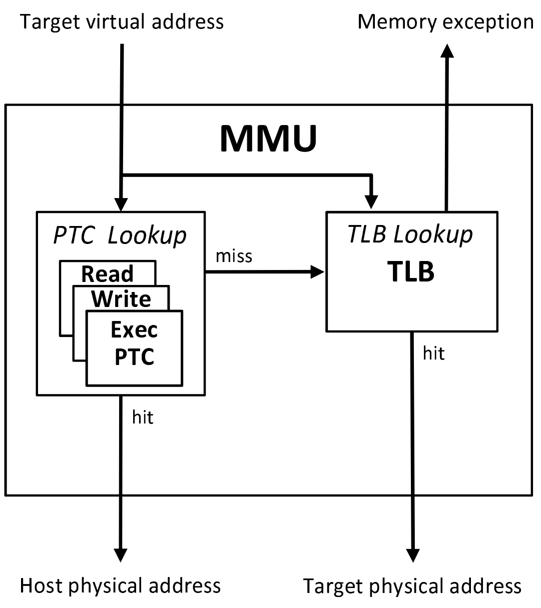


Figure 4.6 MMU Page Translation Caches. This figure shows the read, write and fetch PTCs within the MMU model. A hit in a PTC provides direct access to the data in host memory (host physical address), whereas a hit in the TLB returns the target physical address which is then used in a call to the memory model.

removed from the write PTC. This enables processes with different privileges to access the same physical page and avoid virtual aliasing.

4.8 Future Development

The EHS simulator will continue to be developed to increase its speed and effectiveness as a tool for design-space exploration.

Development is planned in the following areas:

- **Parallel Translation**. The simulator currently waits for the translation process to finish before continuing to simulate the target binary. The simulator will be updated so that translation is performed in parallel to simulation. This will result in faster simulation speeds on first runs, noticeably reducing the latencies experienced when running interactive applications for the first time.

- **CMP Simulation**. The simulator will be developed to support high speed simulation of processors incorporating multiple cores as intensive research is ongoing in this area. It must be capable of modelling homogeneous and heterogeneous processors as well as any on-chip networks and coherence protocols.

- **Retargetable**. In order to test new processor architectures and ISAs the simulator will be made fully retargetable. This will require implementation of an architecture description language (ADL) which is capable of defining instruction semantics and the architectural model.

- **Power Model**. The power consumption of new systems is of utmost importance to manufacturers, particularly in the embedded market where battery life is vital. A power model will be integrated into the simulator to provide detailed power performance figures for system components.

4.9 Summary

This chapter details the design and operation of the Edinburgh High Speed simulator. It outlines the internal components which make it a fast and flexible simulator suitable for performing research into computing system architectures. The EHS simulator can be used for both high and low level design-space exploration. However, increasing the instruction level and cycle-accurate simulation speed of the simulator remains a priority so that it can better fulfil its DSE role.

Chapter 5

Evaluation Methodology

This chapter describes the benchmarking infrastructure and the methodology used to evaluate the performance of the novel DBT simulation techniques presented in this thesis.

5.1 Target System

The target system used for this research is based on the ARC 700$^{\text{TM}}$ processor and configured as shown in table 5.1. The results from running a subset of the EEMBC benchmark suite [EEMB] on the EHS simulator operating in basic block DBT mode were used as a baseline measure of the simulator's performance. The simulation results for the new LTU DBT simulation modes researched in this thesis were then compared with those for basic block DBT based simulation. This enabled any increases or decreases in the simulation speed from deployment of the new simulation techniques to be scientifically quantified.

The 20 EEMBC lite benchmarks simulated are listed in table 5.2, four benchmarks were selected from each of the five categories. All benchmarks were compiled for the ARC 700 architecture using `gcc` version 4.2.1 with `-O2` optimization and linked against `uClibc`. The EEMBC lite benchmarks were run for the default number of iterations and the simulator operated in user-level mode to eliminate the non-deterministic behaviour of a simulated operating system.

Component	Configuration
Processor	ARC 700 uni-processor
L1 inst. cache	8KB 2-way set associative 16 Byte cache block Random replacement policy
L1 data cache	8KB 2-way set associative 16 Byte cache block Random replacement policy

Table 5.1 Target System Architecture

Category	Benchmark	Iterations
Automotive	`aifftr01`	200
	`bitmnp01`	3000
	`idctrn01`	1500
	`matrix01`	110
Consumer	`cjpeg`	1000
	`djpeg`	1000
	`rgbhpg01`	100
	`rgbyiq01`	100
Networking	`ospf`	100
	`pktflowb4m`	100
	`pktflowb512k`	100
	`routelookup`	100
Office	`bezier01fixed`	1000
	`dither01`	1000
	`rotate01`	1000
	`text01`	1000
Telecoms	`autcor00data_3`	5000
	`fbital00data_2`	5000
	`fft00data_1`	1000
	`viterb00data_3`	3000

Table 5.2 EEMBC Lite Default Iterations

5.2 Simulation Environment

All simulations were performed on a 2.66 GHz Intel Core 2 Duo workstation (see table 5.3) running Fedora Core 7 (kernel 2.6.23) under conditions of minimal system load. The EHS simulator was configured to use a simulation epoch of 1000 blocks and a translation threshold of 1 (see table 5.4). The EHS simulator was compiled, and the translated functions dynamically compiled, with `gcc` version 4.1.2 and `-O3` optimization.

Entity	Description
Model	Dell OptiPlex
Processor	1 x Intel Core 2 Duo 6700
CPU frequency	2.66 GHz
L1 caches	32KB I & D caches
L2 cache	4MB per dual-core
FSB frequency	1066 MHz
RAM	2GB, 800MHz, DDRII
OS	Fedora Core 7

Table 5.3 Simulation Host Machine

Entity	Configuration
Simulation epoch	1000 blocks
Translation threshold	1
Decode cache	8K entry 2-way set associative
Translation cache	8K entry Direct-mapped cache
Physical page	8KB

Table 5.4 EHS Simulator Configuration

5.3 Performance Metrics

The EHS simulator was used to simulate each EEMBC benchmark (excluding test harness) and then return the simulation speed in MIPS. All of the benchmarks were simulated on the EHS simulator running in both instruction level and cycle-approximate simulation modes and operating in each of the different DBT simulation modes.

The simulator maintains a count of the instructions simulated and the real simulation time was calculated from readings taken from the host machine's hardware clock. In order to minimize the effect of any variation in the simulation time - caused by the underlying OS and hardware - each benchmark was simulated 10 times and the average simulation speed calculated[1].

The geometric mean speed-up in simulation speed was calculated for each of the new LTU DBT simulation modes relative to basic block DBT based simulation so that the performance of each of the LTU DBT modes could be compared with one another. The geometric standard deviation in the geometric mean speed-up results provides an indication of the variation in the speed-up across all of the benchmarks.

The **Geometric Mean** is defined as:

$$\mu_g = \sqrt[n]{x_1 x_2 \ldots x_n}$$

where x_i represents the value of element i in set X of speed-up values for each benchmark.

The **Geometric Standard Deviation** is defined as:

$$\sigma_g = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\ln x_i - \ln \mu_g)^2}\right)$$

The relative mean absolute error (RMAE) in the cycle count was calculated to measure the cycle count accuracy of the timing-approximate simulator modes relative to cycle-accurate simulation. The standard deviation in the RMAE provides an indication of the spread of cycle count errors across all of the benchmarks.

[1]The average speed for a benchmark is calculated by dividing the total instruction count for all simulation runs by the total simulation time.

The **Relative Mean Absolute Error** is defined as:

$$RMAE = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{|f_i - y_i|}{y_i} \right)$$

where f_i represents the value of element i in set F of cycle-approximate values, and y_i represents the value of element i in set Y of cycle-accurate values for each benchmark.

The **Mean** is defined as:

$$\mu_a = \frac{1}{n} \sum_{i=1}^{n} x_i$$

where x_i represents the value of element i in set X.

The **Standard Deviation** is defined as:

$$\sigma_a = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu_a)^2}$$

Chapter 6

Large Translation Units

This chapter presents the new simulation techniques which were developed during the course of this research. These innovative techniques for the generation of large translation units were designed with the goal of speeding up DBT based simulation. The methods used to profile the target program, identify the LTUs and to perform binary translation at runtime are covered in detail.

6.1 Overview

In DBT based simulation, sections of the target program are discovered at runtime and considered as possible candidates for translation. The translator then translates the sections of code which have been frequently emulated into host code functions. When executed the host code functions emulate the corresponding target instructions within the simulated processor model at much higher speeds than is possible with interpretation.

In DBT based simulators the unit of translation is typically either a target instruction or a basic block [Hech 77]. This thesis explores the hypothesis that significant increases in the simulation speed can be achieved by identifying larger units for translation at runtime.

If this hypothesis is correct, the increase in simulation speed will be attributable to two main factors. Firstly, LTUs provide the translator with greater scope for optimization for speed because they are larger, consisting of multiple blocks rather than just a single basic block. And secondly, larger sections of the target program will on average be emulated within each translated function (TF), where a TF is the translated host code function which when called emulates the target instructions in a translation unit. This results in fewer returns to the outer simulation loop in order to seek the next TF to call.

6.2 Translation Unit Types

This research investigates three different types of LTU [Jone 09], in addition to the basic block translation unit. An LTU, in the context of program simulation, is a group of target basic blocks which are connected by control-flow arcs and which may have one or more entry and exit points. The LTUs selected for use in this research are based on standard objects which have traditionally been used by computer scientists to understand the structure and behaviour of programs.

This research investigates four different ways of constructing translation units based on the following object types:

BB	:	Basic Block translation unit
SCC	:	Strongly Connected Component LTU
CFG	:	Control Flow Graph LTU
Page	:	Physical Page LTU

In contrast to most other DBT based simulators, the EHS simulator profiles the target program's execution in order to discover hot paths rather than to identify hot blocks or pages, parts of which may be infrequently executed or which may contain mostly data. The target program is profiled and the translation units created on a per physical page (target) basis. Grouping translations by physical page aids the simulator in its translation management tasks. All of the translations for a physical page are simply discarded when the simulator detects changes to the data within the page. This can be as a result of self-modifying code or page swapping.

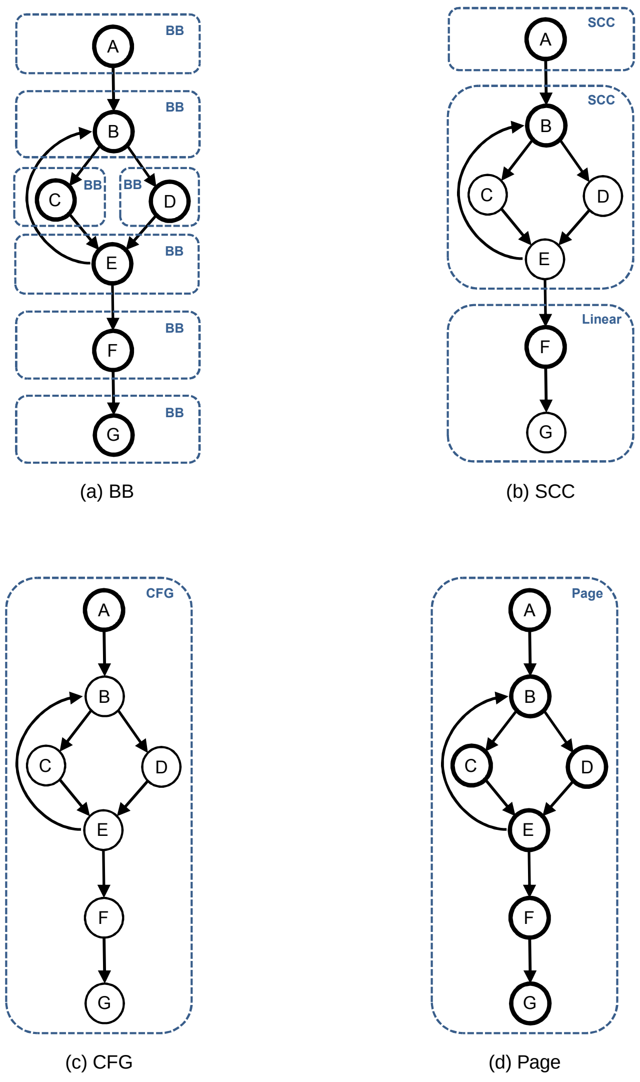


Figure 6.1 Translation Units. The figures above show an example target-program CFG divided into BB, SCC, CFG and Page based translation units respectively. Dotted lines outline the different translation units and the thick-edged circles indicate the possible entry points (basic blocks).

Figure 6.1 shows the different translation unit types and the associated entry points. The example program CFGs have been divided into separate translation units in accordance with the DBT mode. The entry points to blocks within the translation units are dependent upon the DBT mode. Entry is always to the first instruction within a basic block.

In BB based DBT, basic blocks which are frequently executed at simulation time are identified and scheduled for binary translation. When the PC value subsequently matches the start address of a previously translated basic block, the translated code associated with that basic block is called to emulate the block at high speed.

In SCC based DBT, the program execution path is analysed at runtime in order to discover SCC (strongly connected blocks) and linear block region LTUs. The frequently executed SCC, and linear region, LTUs are then marked for translation. When the PC value subsequently matches the root block address of a previously translated SCC LTU, the translated code associated with that SCC is called.

In CFG based DBT, the program execution path is analysed at runtime in order to discover CFG LTUs. The frequently executed CFG LTUs are then marked for translation. When the PC value subsequently matches the root block address of a previously translated CFG LTU, the translated code associated with that CFG is called.

In Page based DBT, the program execution path is analysed at runtime in order to discover all of the CFGs within the physical page. The Page LTU is then translated as a whole. When the PC value subsequently matches the start address of any block within a previously translated Page LTU, the translated code associated with that block within the Page LTU is called.

6.3 Runtime Profiling

Simulation time is partitioned into epochs, where each epoch is defined as the interval between two successive binary translations. The simulator generates a profile of the target program's execution path for those basic blocks interpreted in the current simulation epoch. The end of a simulation epoch is reached when the number of basic blocks interpreted equals the translation threshold, a predefined value. During each simulation epoch, new translation units may be interpreted; previously seen but not translated translation-units may be re-interpreted; translated translation-units may be discarded (e.g. self-modifying code); and translated translation-units may be executed.

In each simulation epoch, execution path profiles for the target program are built-up for each physical page. For BB DBT, this involves simply maintaining a count of the number of times individual basic blocks have been interpreted. In LTU DBT (SCC, CFG or Page) based simulation, a page-CFG [Much 97] is generated for each physical page. Execution path profiles are built-up by adding the next block interpreted in a page to the page-CFG, in addition to incrementing the block's execution count. The EHS simulator models a default physical page size of 8KB.

Figure 6.2 shows examples of the different types of page-CFG that may be created during simulation. A page-CFG may consist of a single CFG or multiple CFGs, in which case they may be separate, combined or a mixture of both types. In order to prevent the generation of 'broken' page-CFGs caused by interrupts and exceptions, block sequences for the different processor interrupt levels are independently traced.

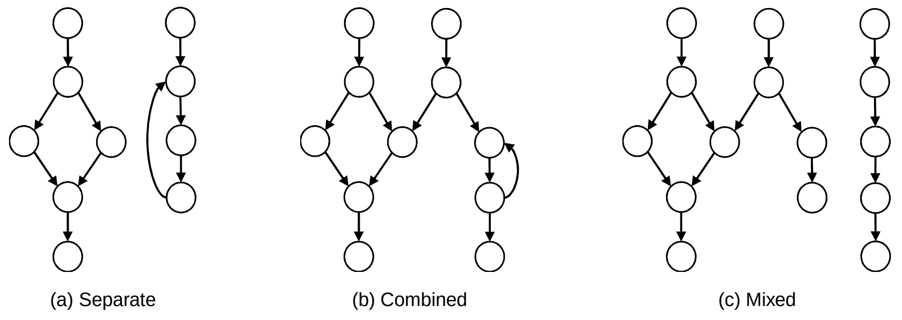


Figure 6.2 Page-CFG Configurations. A page-CFG may contain any number of (a) separate CFGs, (b) combined CFGs or (c) a mixture of both.

At the end of each simulation epoch the page-CFGs are analysed in order to retrieve the constituent translation units. In the case of SCC DBT, Tarjan's algorithm [Tarj 72] is applied to each CFG in order to extract the SCC translation units. Regions of linear basic blocks are also identified as another translation unit. In CFG DBT, the translation units are extracted by tracing the CFG paths starting at each of the root nodes. No further processing is required for Page DBT as the translation unit is the page-CFG itself.

6.4 Program Simulation

The main simulation loop of the EHS simulator is outlined in the flow chart of figure 6.3. The simulator looks up the next instruction to be emulated in the Translation Cache (TC) which is used to return translations at speed. The TC, which is indexed by target instruction address, contains a pointer to the Translated Function (TF) of the corresponding translation unit.

If the next PC address hits in the TC, the corresponding TF (host code function) is called which emulates the target instructions within the translation unit at high speed. If the next PC address misses in the TC, the target instruction is looked up in the Translation Map (TM). The TM contains an entry for every translation unit which has previously been translated. The TM is indexed by target instruction physical-address and contains a pointer to the TF of the corresponding translation unit. If the instruction address hits in the TM the corresponding TF pointer is cached in the TC and the TF called.

If the next instruction address misses in the TM, this indicates that a TF with this entry address has not yet been generated. The basic block starting at the next PC address must therefore be interpreted and profiled in the usual manner. In the case of BB DBT, an entry for the basic block is cached in the Epoch Block Cache (EBC) which records the blocks interpreted during the current epoch. In the case of LTU DBT, the basic block is added to the Epoch CFG Cache (ECC) which is used to create a page-CFG for the target program for the current epoch. Instances of the EBC and ECC caches exist for each physical page.

At the end of each simulation epoch a profiling analysis phase is initiated prior to binary translation. In the case of BB DBT, the EBCs are scanned for frequently executed blocks. In SCC and CFG DBT, the page-CFGs cached in the ECCs are searched for frequently executed LTUs. Page DBT does not require any profiling analysis as Page LTUs are always translated.

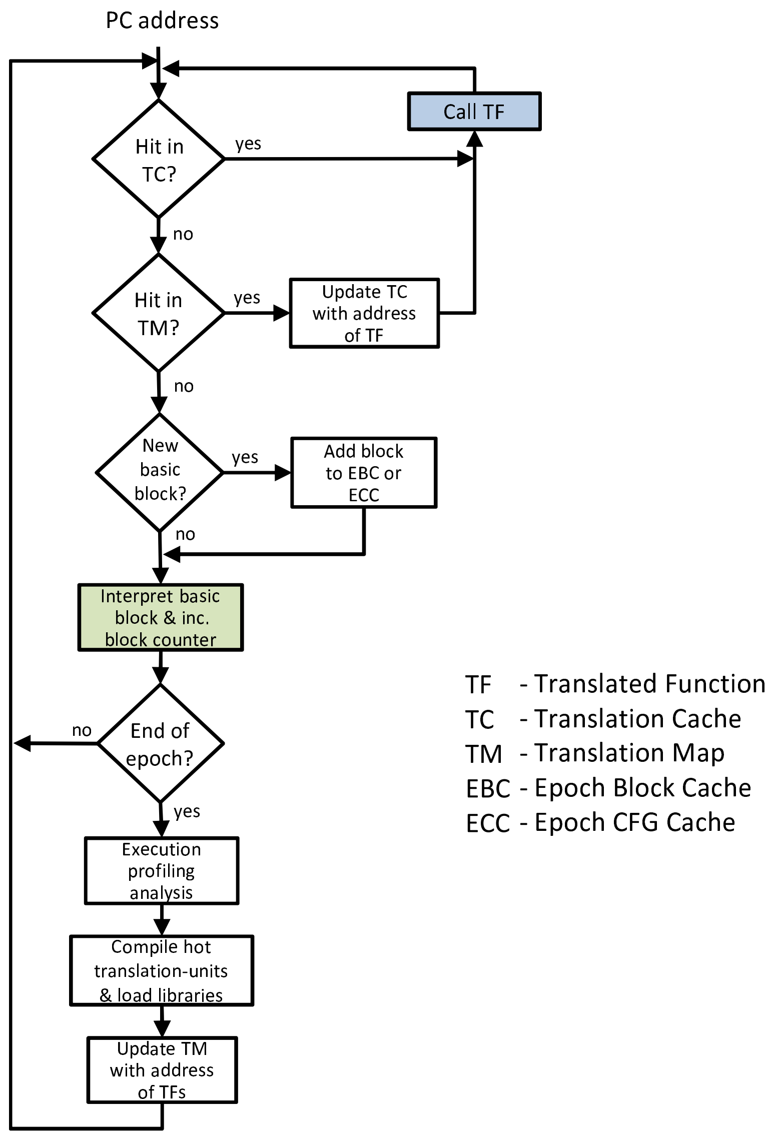


Figure 6.3 LTU DBT Simulation Loop. This flow chart shows the EHS simulator's LTU DBT simulation loop.

6.4.1 Dynamic Binary Translation

Those translation units which were interpreted at least as many times as the translation threshold during the previous simulation epoch are marked for translation. The metric used to determine whether a translation unit is considered hot depends on the DBT simulation mode as follows:

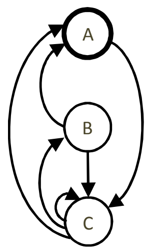
BB	: Number of executions.
SCC	: Number of root node executions.
CFG	: Number of root node executions.
Page	: Always translate.

After all the hot translation units have been identified, the binary translation phase begins. The hot translation units are translated in batches consisting of translation units belonging to the same physical page.

The translation units are first converted into C code functions which emulate the target instructions within the processor model. The C code functions are then compiled using `gcc` [Stal 01] into shared libraries which are loaded by the dynamic linker. Finally, the TM is updated with pointers to the newly generated TFs. If the next instruction to be simulated corresponds to an entry point in a recently translated TF, its instruction address will hit in the TM, a pointer to the TF will also be added to the TC and the TF called.

Each target instruction within a basic block is translated into C code which emulates the target instruction's operation within the processor model. The simulated processor state is updated during emulation of each instruction with the exception of the PC which is updated at the end of each block or on encountering an exception. Translated functions exit immediately on detecting an exception or at the end of the current basic block if pending interrupts exist. Control is returned to the main simulation loop where the exception or interrupts can be serviced. The edges connecting basic blocks are recorded in the page-CFGs during profiling so that the control flow can be replicated within the C code functions using `GOTO` statements.

Figure 6.4 shows an example CFG translation unit and the corresponding outline C code function. The C code functions for Page DBT based simulation contain a jump table at the beginning which enables emulation to commence from any block within the translation unit.



(a) CFG LTU

```
void L_00010098 (cpuState *s)
{

Block_0x00010098:

    /* C code to emulate
       target instructions
       within block A */

    s->pc = next_pc;

    if (pending_interrupts) return;

    /* Unconditional Direct Control Transfer */
    goto Block_0x000100c4;

Block_0x000100bc:

    /* C code to emulate
       target instructions
       within block B */

    s->pc = next_pc;

    if (pending_interrupts) return;

    /* Conditional Direct Control Transfer */
    if (s->pc == 0x00010098) goto Block_0x00010098;
    goto Block_0x000100c4;

Block_0x000100c4:

    /* C code to emulate
       target instructions
       within block C */

    s->pc = next_pc;

    if (pending_interrupts) return;

    /* Indirect Control Transfer */
    if (s->pc == 0x00010098) goto Block_0x00010098;
    if (s->pc == 0x000100bc) goto Block_0x000100bc;
    if (s->pc == 0x000100c4) goto Block_0x000100c4;
    return;
}
```

(b) C Code Function

Figure 6.4 LTU Translated Function. When the TF is called execution starts at the root node (block A, start address 0x00010098). All target instructions within the block are emulated before the PC is updated. A check is then performed to see if any interrupts need servicing before simulating the next block or exiting the TF.

6.5 Cycle Accurate Simulation

Large translation unit based DBT can be applied to instruction level and cycle-accurate simulation of microprocessor systems. However, because a high proportion of the overall simulation time is spent performing cycle-accurate modelling in a DBT based simulator, the scope for speeding up cycle-accurate simulation using LTU DBT is greatly reduced.

If DBT techniques are to realize significant increases in cycle-accurate simulation speeds, the amount of time spent accurately modelling the system must be reduced. The application of DBT simulation techniques will therefore prove to be much more effective when applied to a cycle-approximate simulator. Chapters 7 and 8 provide quantitative evaluation of these new techniques when applied to instruction level and cycle-approximate simulation respectively.

Chapter 7

Instruction Level Simulation

This chapter investigates high speed DBT based instruction level simulation and evaluates the research hypothesis by comparing the simulation speeds of the novel LTU DBT simulation techniques. The simulation performance and the characteristics of the translation units generated and the translated functions called are analyzed for each LTU DBT mode. The simulation speed of the EHS simulator is also compared with two state-of-the-art functional DBT based simulators.

7.1 Overview

Instruction level simulators enable target programs to be simulated at high speed as they need only model the order of the processor state changes accurately and not the precise moments in time at which they occur. They support high level DSE, hardware/software co-design, verification and debugging. They also play an important role in the development of software and compilers for new systems which are being developed in parallel.

When operated in instruction level simulation mode the EHS simulator updates the target processor state after each instruction is emulated. However, when the simulator is running in fast (DBT based simulation) mode, the PC is updated at the end of each basic block and the instruction count updated on exiting each TF in order to maximize performance.

7.2 Instruction Level Simulation Analysis

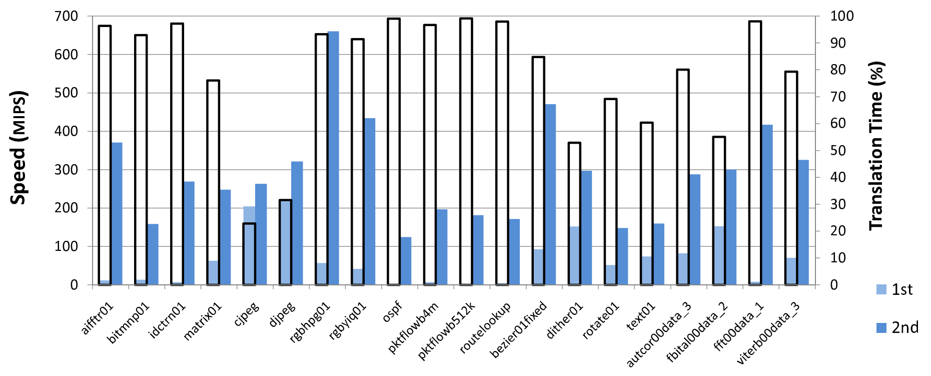
In order to test the research hypothesis the same set of benchmarks were run on the simulator operating in each of the different DBT simulation modes. This enabled the instruction level simulation performance for the three LTU DBT modes and for the basic block DBT mode to be quantitatively compared with each other. This section presents the results from simulating a subset of the EEMBC benchmark suite [EEMB] on the Edinburgh High Speed simulator. The simulation speeds reported are in native MIPS: millions of target instructions simulated per real-time (host) second. Unless explicitly stated otherwise, the simulation epoch was set at 1000 blocks and the translation threshold set to 1.

7.2.1 Performance

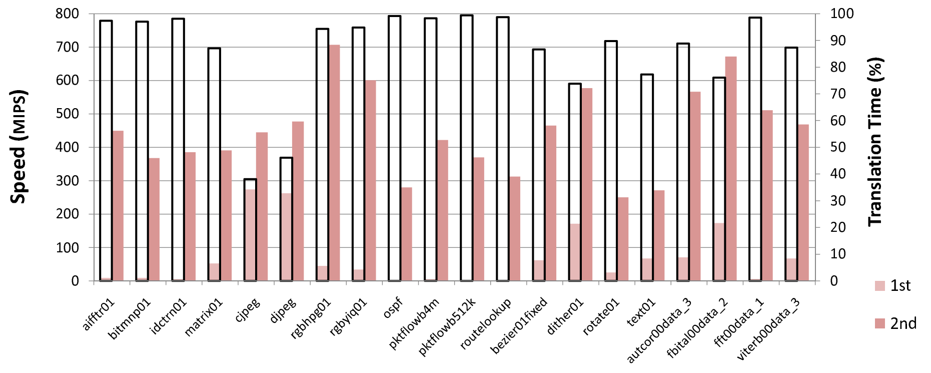
Figure 7.1 shows the simulation speed and the proportion of the total simulation time spent performing translation when the simulator is operated in BB, SCC, CFG and Page DBT simulation modes. Each benchmark was simulated twice, with the second simulation run loading the translations generated by the first simulation run. The simulation speeds were calculated using the overall simulation time which includes the time spent performing translation and the time spent loading translations, in addition to the time taken to emulate the target instructions (interpretively or using TFs).

The results show a large jump in simulation speed from the first to the second simulation run for most of the benchmarks in all DBT based simulation modes. For example, the simulation speed for the bezierfixed benchmark goes from 98 MIPS on the first run, to 710 MIPS on the second run when the simulator is operating in Page DBT mode.

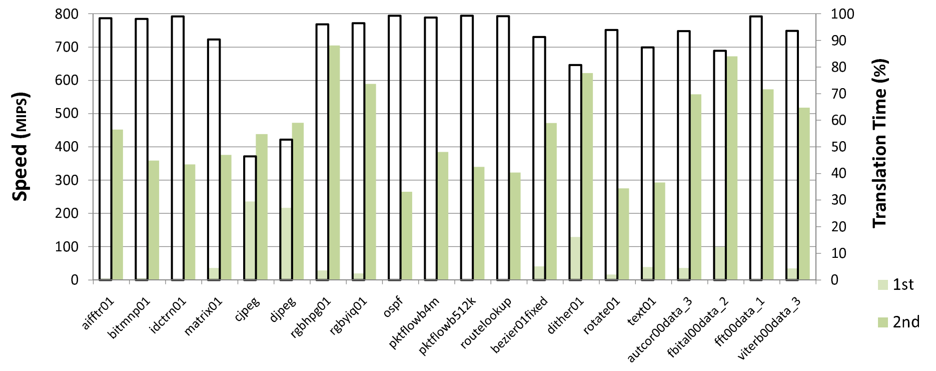
The second simulation run represents the maximum simulation speed attainable for a given benchmark using a particular DBT simulation mode. No translation is performed on the second run as all of the target instructions are emulated by TFs which were generated on the first run. As many of the benchmarks run for short periods of time, the proportion of the total simulation time spent performing translation can be significant on the first run. For benchmarks which exhibit longer simulation times, the amount of time spent performing translation will tend towards zero on the first run. If binary



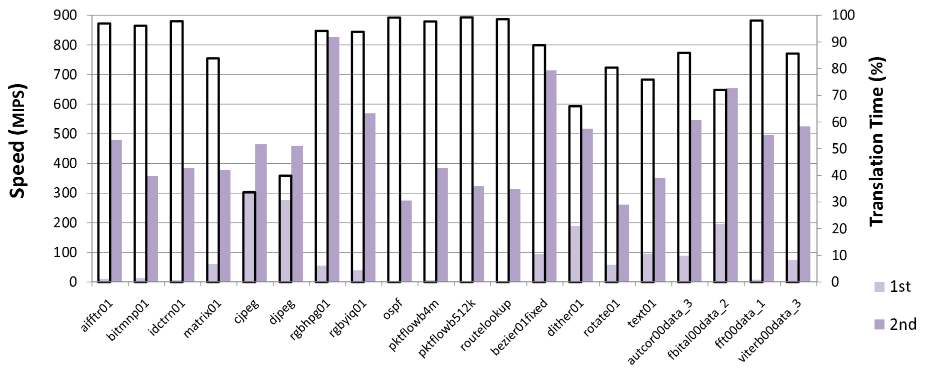
(a) BB



(b) SCC



(c) CFG



(d) Page

Figure 7.1 Instruction Level Simulation Profile. The figures show the simulation speed and the proportion of total simulation time spent performing translation (outlined bars) for two consecutive runs of each benchmark in each of the DBT modes.

translation were performed in parallel to simulation, the minimum simulation speed experienced would be equal to that of interpretive simulation.

Overall, the three LTU DBT simulation modes perform significantly better than the BB DBT simulation mode. Figure 7.2 shows the simulation speed for each benchmark running on the simulator operating in each of the different DBT simulation modes. A summary of the simulation performance statistics is provided in table 7.1. The LTU DBT simulation speeds range from a low of 233 MIPS for the rotate benchmark in SCC mode, to a high of 826 MIPS for the rgbhpg benchmark in Page mode. For BB DBT, the slowest simulation speed is 124 MIPS for the ospf benchmark and the fastest is 660 MIPS for the rgbhpg benchmark. The average simulation speed across all benchmarks is 283 MIPS for BB DBT based simulation, 460 MIPS for SCC DBT, 455 MIPS for CFG DBT and 466 MIPS for Page DBT.

	Interpretive	BB	SCC	CFG	Page
SPEED (MIPS)					
Slowest	24	124	233	270	259
Fastest	33	660	706	705	826
Median	29	278	446	445	461
Average	**30**	**283**	**460**	**455**	**466**
SPEED-UP					
Geo. Mean	**0.11**	**1**	**1.63**	**1.64**	**1.67**
Geo. S.D.	0.046	0	0.397	0.376	0.336

Table 7.1 Instruction Level Simulation Performance Summary. The geometric mean speed-up for each DBT simulation mode is relative to the basic block DBT simulation speed.

The increase in simulation speed for each benchmark and LTU DBT mode, compared to BB DBT based simulation, is shown in figure 7.3. LTU DBT simulation outperforms BB DBT based simulation for all benchmarks with the exception of the bezierfixed benchmark, where BB DBT simulation outperforms SCC DBT simulation by a small margin. Overall, the LTU DBT simulation modes exhibit a mean speed-up of at least 1.63 compared to BB DBT based simulation. The smallest and largest simulation speed-ups were observed when the simulator was running in SCC DBT mode. The smallest speed-up was 0.95 for the bezierfixed benchmark and the fastest speed-up was 2.32 for the bitmnp benchmark. Page DBT based simulation performs the best across all benchmarks with a mean speed-up of 1.67 and standard deviation of 0.336. However, SCC DBT simulation exhibits the fastest simulation speed in 9 out of 20 of the benchmarks, compared to 7 out of 20 for Page DBT.

7.2.2 Instruction Emulation

The proportion of instructions simulated by TFs (‘translated’ simulation) is greater than 99% on the first simulation run for all benchmarks and DBT modes. The benchmarks and DBT simulation modes which exhibit the highest degree of interpretation on the first run are, aifftr with 0.17% of instructions interpreted when running in BB and Page DBT mode, and pktflowb512k with 0.52% of instructions interpreted in SCC DBT mode and 0.40% in CFG DBT mode. The proportion of total simulated instructions emulated by TFs is dependent primarily on the application behaviour.

The results demonstrate the repetitive nature of the benchmarks and the manner in which of all the DBT simulation modes benefit from this, even when benchmark run times are very short. All of the DBT simulation modes perform similarly with just under 100% of instructions being emulated by TFs on the first run, increasing to 100% on the second run.

7.2.3 Dynamic Binary Translation

The proportion of the total simulation time spent performing translation is dependent upon the simulation period, benchmark behaviour, simulation epoch, translation threshold, simulation run and the DBT simulation mode (see figure 7.1). A significant proportion of the simulation time is spent performing translation on the first run with no translation taking part on the second run for all benchmarks and all DBT modes. All of the target instructions are translated on the first simulation run with no translation occurring on successive runs. Further translation only occurs on successive runs if the program execution path changes between simulation runs (SCC and CFG DBT modes only) or if the code is self-modifying.

The proportion of time spent performing translation follows a similar pattern for all DBT simulation modes with three-quarters or more of the benchmarks spending over

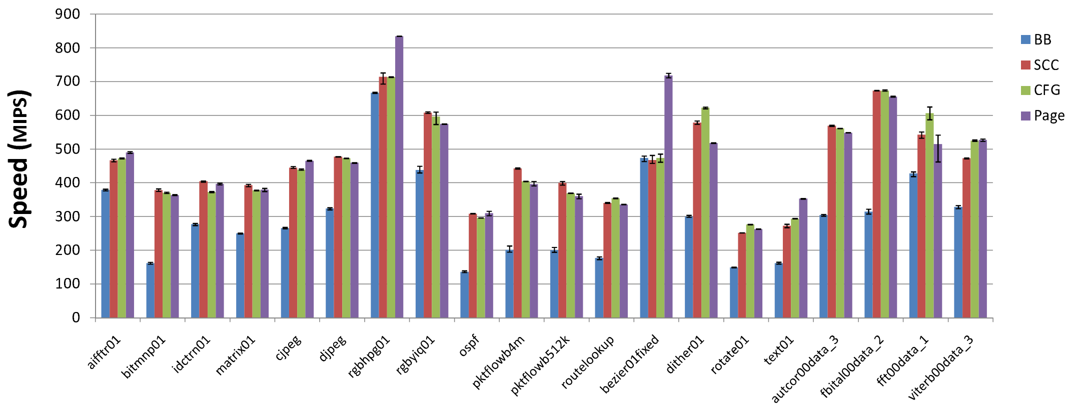


Figure 7.2 DBT Instruction Level Speed. These figures show the simulation speed for each benchmark using DBT simulation modes. The simulation speeds presented are for the main simulation loop. The speeds shown are the average of 10 simulation runs in which all target instructions had previously been translated.

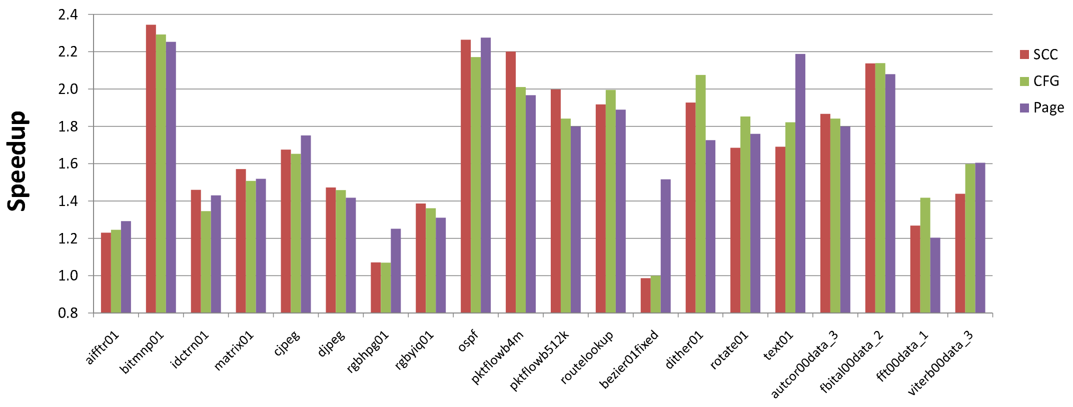


Figure 7.3 LTU DBT Instruction Level Speedup. These figures show the simulation speed-ups for each benchmark for each LTU DBT mode relative to basic block DBT based simulation. The speed-ups shown are calculated from the average of 10 simulation runs in which all target instructions had previously been translated.

70% of the time performing translation on the first run. The proportion of the total simulation time spent performing translation for BB DBT based simulation ranges from 99%, for benchmarks such as ospf and pktflowb512k down to 23% for cjpeg. The proportion of time spent performing translation for SCC DBT based simulation is slightly higher than for BB DBT simulation, ranging from 99% for ospf and pktflowb512k down to 38% for cjpeg. And the proportion of time spent performing translation for CFG DBT based simulation is slightly higher than for SCC DBT simulation, ranging from 99% for ospf and pktflowb512k down to 46% for cjpeg. Page DBT based simulation is very similar to SCC DBT simulation, with the proportion of time spent performing translation ranging from 99% for ospf and pktflowb512k benchmarks down to 34% for cjpeg.

The simulation times for the benchmarks ospf, pktflowb4m, pktflowb512k and routelookup (EEMBC networking category) are the shortest. This explains why these benchmarks spend a high proportion of the total simulation time performing translation on the first run for all DBT modes. Conversely, the cjpeg and djpeg benchmarks have the longest simulation times and therefore spend a much smaller proportion of the simulation time performing translation on the first run for all DBT modes.

The proportion of the total simulation time spent performing translation on the first run for each benchmark is very similar across all DBT simulation modes. In general, the percentage of time spent performing translation is slightly higher for SCC DBT than for BB DBT, and slightly higher for CFG DBT than for SCC DBT, whilst that for Page DBT is very similar to SCC DBT. The differences in the percentage of simulation time spent translating across DBT modes reflects the number and size of the translation units which are translated, with larger translation units taking longer to construct and to compile. These results reflect the fact that CFG LTUs are larger than SCC LTUs which are in turn larger than BB LTUs.

7.2.4 Simulator Tasks

This section investigates what proportion of the overall simulation time is spent performing each of the main simulator tasks. It therefore highlights those tasks which have a predominant affect on the simulation speed during each run.

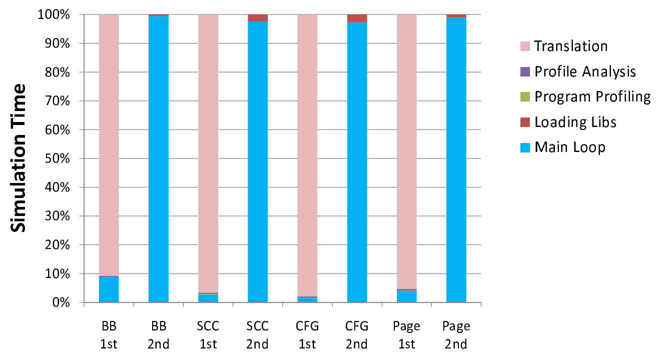
The EHS simulator performs five main simulator tasks:

- **Main Simulation Loop**: function which emulates target program instructions either interpretively or by calling TFs.
- **Loading Libraries**: function which loads the shared libraries containing the TFs.
- **Program Profiling**: function which adds interpreted blocks to the page-CFG target program execution profile (LTU DBT modes only).
- **Profile Analysis**: function which analyzes the page-CFGs in the ECCs, or blocks in the EBCs, in order to identify hot translation units.
- **Translation**: function which translates the hot translation units creating shared libraries which hold the TFs.

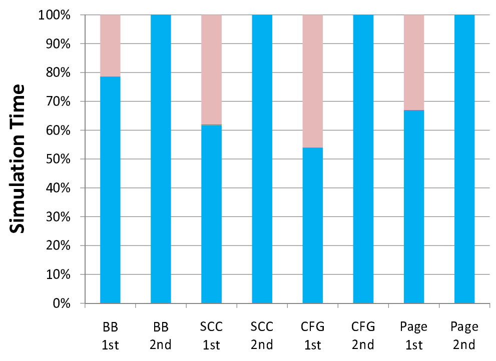
The proportion of the total simulation time spent performing each task for each DBT mode is shown for five benchmarks in figure 7.4. Each benchmark was simulated twice, with the second simulation using the translations generated by the first simulation. The figure shows that a high proportion of the total simulation time is spent performing translation during the first run which is reduced to zero by the second run. The precise pattern depends on the benchmark and on the DBT simulation mode.

On the first run, 78% - 99% of the total simulation time is spent performing translation for 4 out of 5 of the benchmarks (bitmnp, ospf, bezierfixed, viterb). The cjpeg benchmark is the exception which spends between 22% and 46% of the time performing translation. This is because the cjpeg benchmark runs for a much longer period of time than the other benchmarks and consequently spends a much smaller proportion of the overall time performing translation. CFG DBT based simulation spends the largest proportion of simulation time performing translation followed by SCC DBT and Page DBT simulation and lastly BB DBT based simulation. The majority of the remaining simulation time is spent in the main simulation loop emulating target instructions.

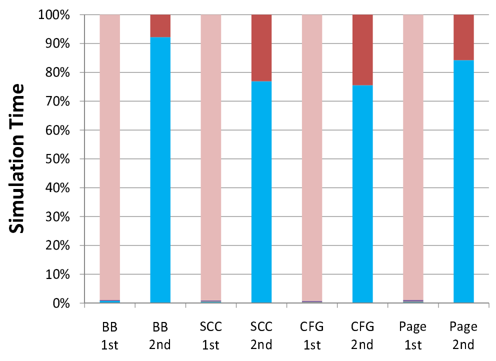
On the second run, just under 100% of the total simulation time is spent within the main simulation loop for 4 out of 5 of the benchmarks. The time spent performing



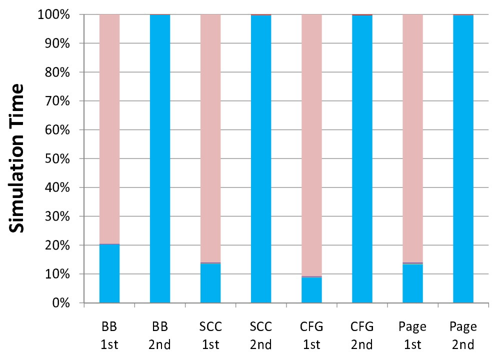
(a) bitmnp



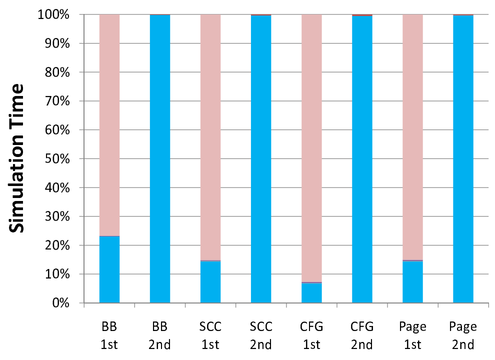
(b) cjpeg



(c) ospf



(d) bezierfixed



(e) viterb

Figure 7.4 EHS Simulation Tasks. These figures show the proportion of the total simulation time spent performing each of the main simulation tasks. One benchmark from each EEMBC category was simulated for two runs using different DBT simulation modes.

other tasks is largely insignificant. The exception is the ospf benchmark where the proportion of time spent loading the dynamic libraries varies from 8% for BB DBT based simulation to 24% for CFG DBT based simulation. This is because the ospf benchmark runs for a very short period of time compared to the other benchmarks and the proportion of the overall simulation time spent loading libraries is therefore markedly higher.

Basic block DBT based simulation outperforms SCC DBT based simulation when emulating the bezierfixed benchmark. This confirmed in the bar charts for bezierfixed which show that less time is spent in the main simulation loop for SCC DBT based simulation than for BB DBT based simulation. On average BB DBT based simulation spent 1.19 seconds in the main simulation loop whereas SCC DBT based simulation spent 1.22 seconds. Although one might expect SCC DBT to perform better than BB DBT, this result highlights the complex interactions which take place between benchmark, simulator and host hardware.

7.2.5 Translated Functions

Table 7.2 shows the average and largest, static and dynamic, TF block sizes broken down by benchmark and LTU DBT simulation mode. The static size of a TF is equal to the number of target blocks contained within the TF, whilst the dynamic size of a TF is equal to the number of target blocks emulated by a TF when it is called, this may vary for a given TF each time it is called.

During each simulation epoch, the simulator profiles a fixed number of basic blocks (interpreted). The size of the TFs generated will be depend on both the size of the simulation epoch and the type of translation unit used. As expected, the average number of blocks within a TF is greater for Page DBT than for CFG DBT, which is in turn greater than for SCC DBT, across all benchmarks.

The average dynamic TF size provides an indication of the overhead which the simulator experiences from returning to the main simulation loop. The larger the average dynamic TF size, the fewer the number of times the simulator had to return to the slow main loop in order to search for the next TF to call. An increase in the average dynamic TF size should therefore correspond with an increase in simulation speed as

Benchmark	Avg TF Size			Avg Dynamic TF Size			Largest TF			Largest Dynamic TF		
	SCC	CFG	Page	SCC	CFG	Page	SCC	CFG	Page	SCC	CFG	Page
`aifftr`	3.4	7.4	16.4	4.4	6.0	5.1	87	126	134	240127	240129	5120
`bitmnp`	5.0	10.6	33.1	10.3	14.4	10.8	54	79	116	10111	10113	10113
`idctrn`	3.0	12.9	25.4	11.5	123.9	9.2	34	97	114	8193	8801	8193
`matrix`	3.5	6.5	28.7	5.7	11.8	9.3	45	97	174	3419	3419	3419
`cjpeg`	3.6	6.5	20.8	12.6	42.1	40.5	51	72	108	69129	69129	69129
`djpeg`	3.7	6.2	20.0	48.8	80.5	77.4	43	94	121	64327	64327	64327
`rgbhpg`	3.7	6.3	25.3	39.7	59.4	26.5	43	87	125	75921	75925	75925
`rgbyiq`	3.9	6.6	31.0	3612.9	4921.7	4272.4	43	97	133	4607999	4608002	4608002
`ospf`	4.4	8.7	26.0	426.3	484.6	497.3	64	105	148	189599	189602	189602
`pktflowb4m`	4.4	7.4	31.3	2485.1	17.8	17.8	39	90	149	1171544	1171548	1171548
`pktflowb512k`	4.4	7.6	31.7	325.9	397.5	17.3	39	90	149	154794	154798	154798
`routelookup`	4.3	7.5	27.7	667.6	712.8	37.0	41	105	105	56453	56457	56457
`bezierfixed`	4.0	7.2	26.2	1029.9	7163.2	1057.4	43	105	135	187799	187802	187802
`dither`	4.0	6.0	28.6	33.6	30332.6	33.6	44	97	130	163838	290986	163841
`rotate`	8.3	16.5	34.5	34.5	121.9	4.7	100	105	134	16644	16648	16648
`text`	4.3	10.4	28.1	4.3	10.2	9.4	43	105	136	2590	3523	2605
`autcor`	3.4	7.2	29.6	4350.9	5323.8	4518.1	41	86	127	15567	15573	15573
`fbital`	4.0	9.5	25.7	148.1	508.4	508.0	34	73	111	10752	10754	10754
`fft`	3.8	7.9	31.3	32.8	102.0	69.6	41	99	123	10495	10498	10498
`viterb`	4.3	10.7	22.1	1498.2	2617.6	1520.5	34	77	108	22475	22511	22511
Average	**4.2**	**8.5**	**27.2**	**739.1**	**2652.6**	**637.1**	**48.2**	**94.3**	**128.9**	**354088.8**	**360527.3**	**342343.3**

Table 7.2 Static and Dynamic Translated Functions. This table shows the average and largest, static and dynamic TF block sizes for each benchmark for the different LTU DBT simulation modes.

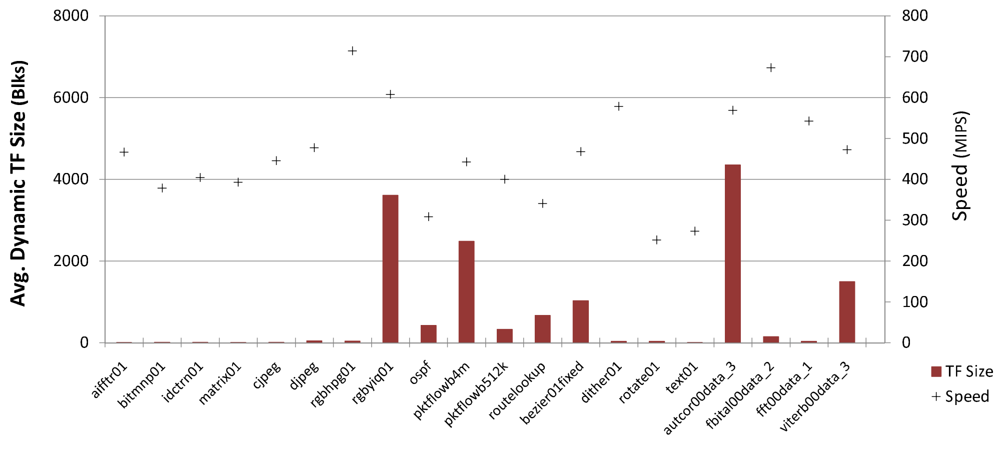
more instructions are emulated per TF call. For most benchmarks, the average number of blocks simulated per TF call is greatest for CFG DBT, followed by Page DBT and then SCC DBT.

Figure 7.5 shows graphs of the average dynamic TF size and the maximum simulation speed for all benchmarks and LTU DBT simulation modes. The dynamic TF size for each benchmark follows a similar pattern across the different LTU DBT modes with the largest dynamic TF sizes corresponding to higher than average simulation speeds. One exception however is the rgbhpg benchmark which exhibits the fastest simulation speed across all DBT modes whilst possessing a small average dynamic TF size.

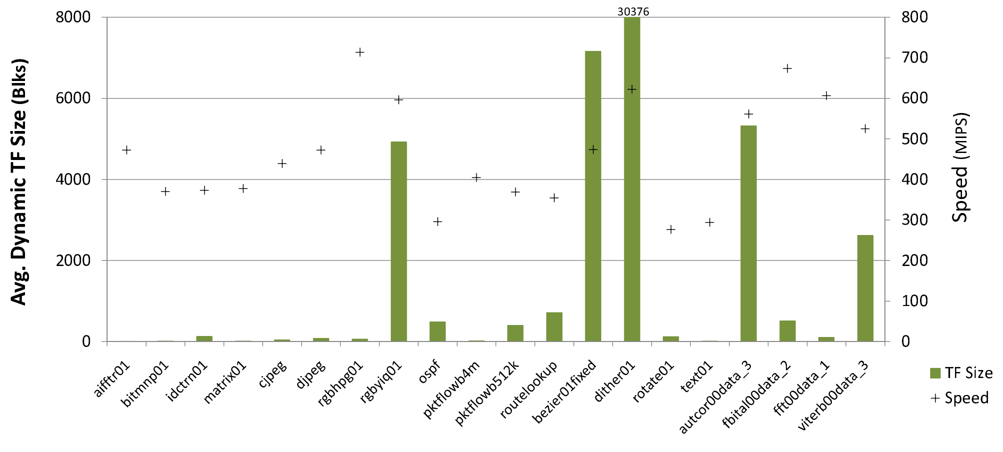
Figure 7.6 shows the distribution of static TFs for the pktflowb4m benchmark. The average static TF size for Page DBT based simulation is greater than for CFG DBT which is in turn greater than for SCC DBT. The graphs show that both CFG and SCC DBT based simulation generate a lot of TFs (53 and 79 respectively) which contain only a single block, whereas Page DBT based simulation generates just two 2 TFs containing 2 blocks. Page DBT based simulation identified the largest translated unit (149 blocks) followed CFG DBT (90 blocks) and lastly SCC DBT (39 blocks). In total Page DBT based simulation discovered 28 TFs, CFG DBT 210 TFs and SCC DBT 237 TFs.

Figure 7.7 shows the distribution of dynamic TF calls for the pktflowb4m benchmark. SCC DBT based simulation exhibits a higher average dynamic TF size than either CFG or Page DBT based simulation. This is due to the large number of TFs called in both CFG and Page DBT based simulation which simulate only a small number of basic blocks. It can be observed that SCC DBT based simulation called only 1,182 TFs which executed a single block, where as CFG DBT simulation called 283,038 TFs which executed a single block and Page DBT simulation called 282,816 TFs which executed three blocks. In addition, SCC DBT based simulation called TFs which emulated 87545 blocks 100 times, the same number of times as the default number of iterations for the benchmark. In the case of the pktflowb4m benchmark, the LTU DBT mode which exhibits the highest average dynamic TF size (SCC DBT) also simulates the benchmark the fastest (443 MIPS).

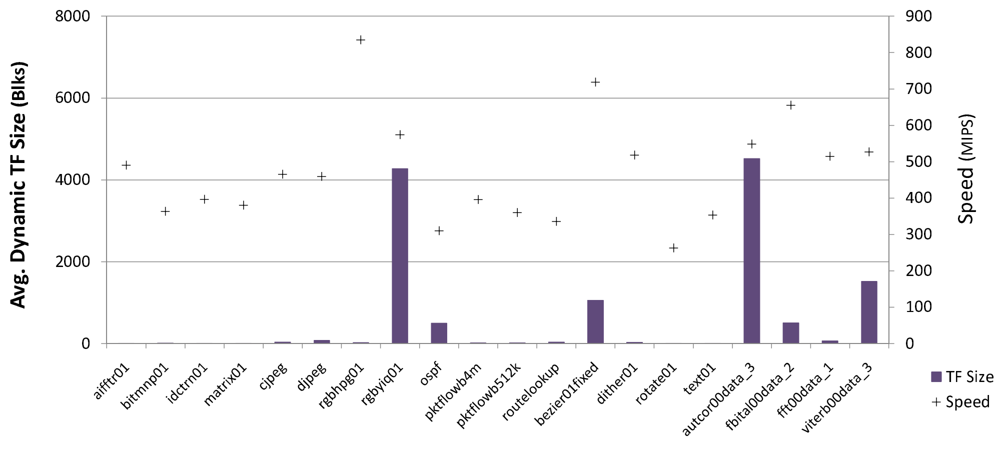
Basic blocks which are emulated by TFs cease to be profiled in all DBT simulation modes. This is an issue for LTU DBT based simulation as it prevents accurate genera-



(a) SCC

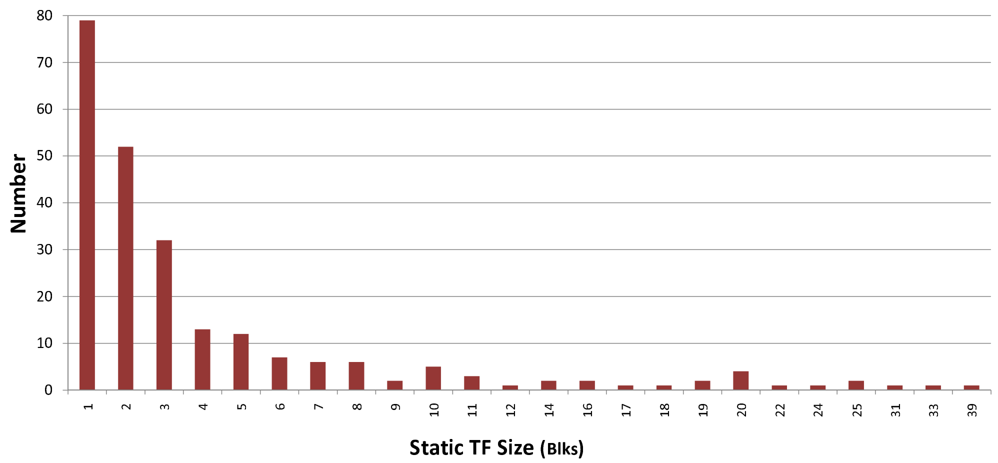


(b) CFG

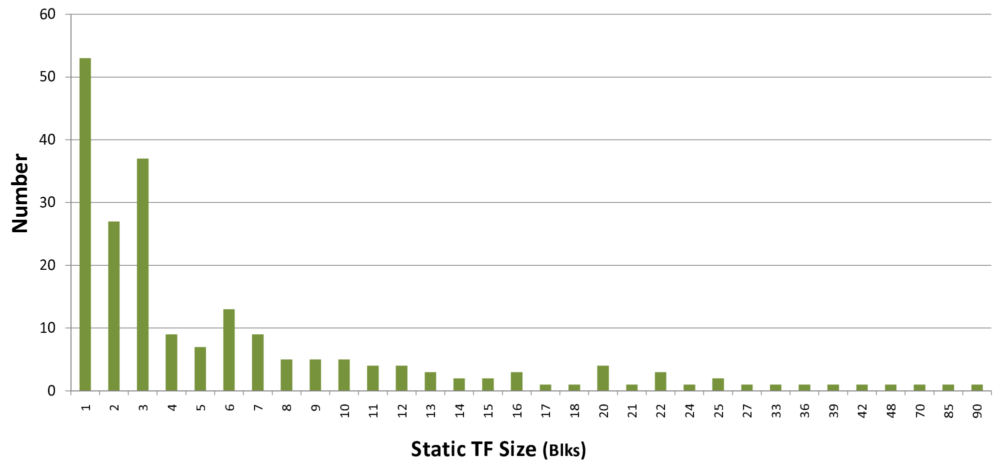


(c) Page

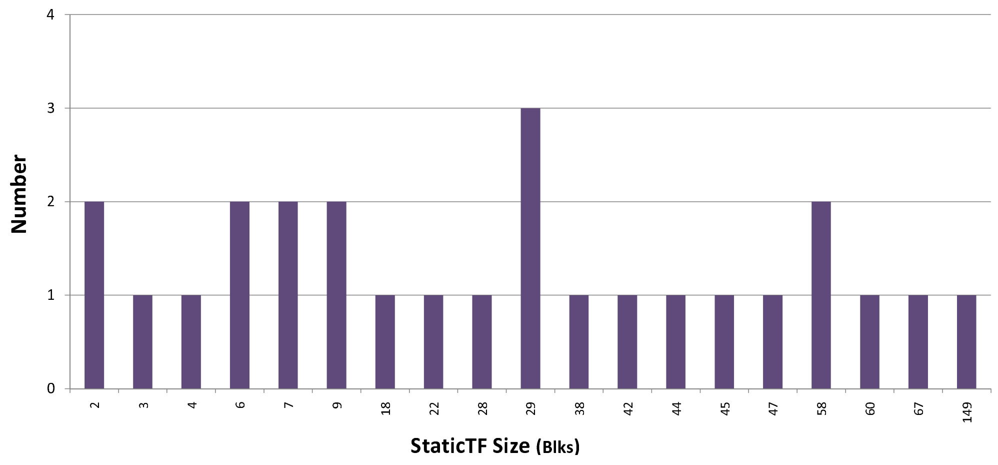
Figure 7.5 LTU Dynamic TF Size and Simulation Speed. The figures show the average dynamic TF size and simulation speed for each benchmark for SCC, CFG and Page DBT based simulation.



(a) SCC

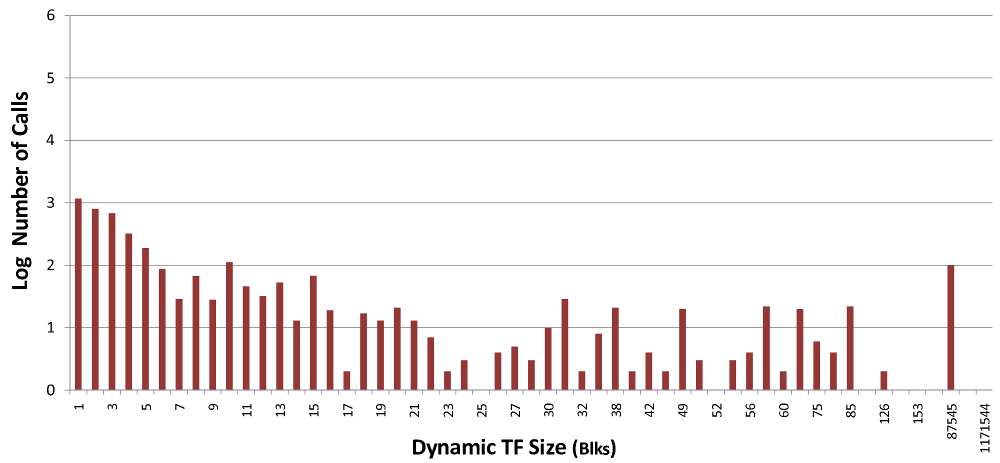


(b) CFG

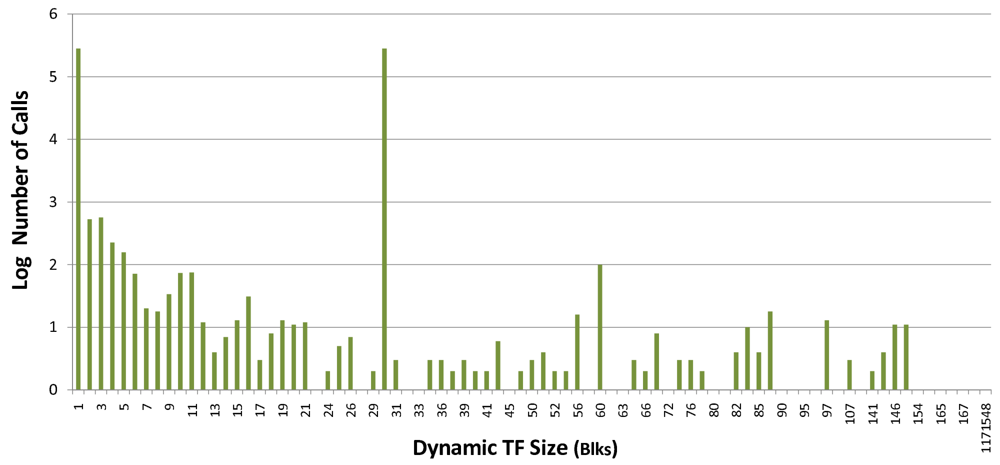


(c) Page

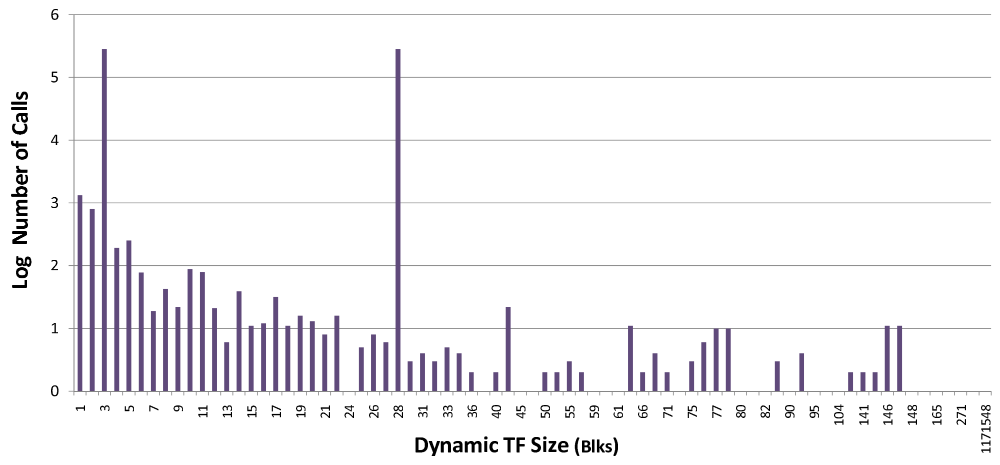
Figure 7.6 Static TF Distribution. The figures show the number of static TFs of a given size generated for the pktflowb4m benchmark for SCC, CFG and Page DBT based simulation.



(a) SCC



(b) CFG



(c) Page

Figure 7.7 Dynamic TF Distribution. The figures show the number of times that dynamic TFs of a given size were called for the pktflowb4m benchmark for SCC, CFG and Page DBT based simulation.

tion of page-CFGs. It may cause individual blocks, or small groups of blocks, which lie on the execution path to be profiled as isolated code segments. This results in the formation of fragmented translation units (refer to section 7.5). Severe translation unit fragmentation is in evidence during CFG and SCC DBT based simulation of the pkt-flowb4m benchmark in which large numbers of static TFs are generated consisting of just one block.

Changes in a program's execution path (phase change) may also affect simulation performance. Target program execution path information is only gathered during the current simulation epoch and only for blocks which are interpreted. Profiling information for a section of code may therefore be incomplete as the simulator has no future (or past epoch) knowledge of indirect branches between or within translation units. If there is a change in the program path (new hot path), it could be that the blocks which lie on the new path are contained within different TFs. The simulator must therefore call multiple TFs in order to simulate the instructions in the current path.

Translation unit fragmentation lowers the average dynamic TF size and explains why the average dynamic TF size for Page DBT based simulation is less than that for CFG DBT based simulation (except for the ospf benchmark) even though it exhibits a higher average static TF size. This can be observed in Page DBT based simulation of the pk-tflowb4m benchmark in which the smallest static TF size is 2, but where there are over 1,000 calls to dynamic TFs of size 1. Translation unit fragmentation and program phase changes lower the average dynamic TF size resulting in slower simulation speeds.

For a given benchmark and LTU DBT simulation mode there exists a TF, or multiple TFs, which represent the the simulation of the target program's main loop, or part thereof. In the case of the rgbhpg benchmark, TFs with a dynamic block size of 75,921 blocks were called a total of 100 times, the same number of times as the default number of iterations for the benchmark. This is true for all of the LTU DBT simulation modes, suggesting that the main loop, or part thereof, of the rgbhpg benchmark is contained within a strongly connected component.

Whilst those simulations which exhibit very large average dynamic TF sizes do experience faster than average simulation speeds, the average dynamic TF size does not on its own explain the simulation speeds achieved relative to other benchmarks or DBT simulation modes. For example, whilst CFG DBT based simulation of the bezierfixed benchmark exhibits the largest average dynamic TF size (7163 blocks compared to 1057 blocks for Page and 1029 blocks for SCC DBT) by a factor of 6 or more, it possesses the slowest simulation speed (467 MIPS), slower even than BB DBT based simulation.

7.3 Translation Cache Size

Whilst the size of the translation cache has an affect on simulation performance it does not favour one DBT simulation mode over another as it holds a fixed number of TFs. Increasing the size of the translation cache may in theory improve performance as a larger number of TFs should be accessible at greater speed. However, it is the size of the L1 cache on the host machine which has an overriding affect on simulation speed. Maximum simulation speed will be attained if the translation cache, or the hot part of it, fits into the L1 cache.

7.4 Workload Sensitivity

It was observed that translation is only performed on the first simulation run for all DBT simulation modes when simulating the EEMBC benchmarks. This shows that the translations generated in the first run provide all of the TFs necessary to emulate the complete benchmark on consecutive simulation runs. However, it should be remembered that this is only the case if the benchmarks are rerun with the same workload. If the same benchmark is run with a different workload then changes to the program's execution path, and possibly to the program code itself (self-modifying), are likely to occur.

If the program execution path changes with different workloads then new translations may be generated on consecutive simulation runs. This can affect the performance of SCC and CFG DBT based simulation as entry to each TF is only permitted via the root node and new translations may therefore need to be generated. Changes to the workload will not initiate further translation in BB and Page DBT based simulation as both modes allow direct entry to any block within a TF. It will however adversely affect the performance of Page DBT based simulation as a change to the program path will very likely straddle many more TFs.

7.5 Translation Unit Fragmentation

Translation unit fragmentation is the generation of multiple, smaller translation units resulting from disruption to the profiling process during simulation. Incomplete profiling of the target program path is caused by a number of factors including the physical page size, the simulation epoch and translated functions.

The target program's execution path is profiled at runtime in order to discover the translation units. However, only those blocks which are interpreted are profiled, blocks emulated by TFs (previously translated translation units) do not get profiled. No profiling is performed within TFs so that sections of the target program can be emulated at as fast a speed as possible. Adding profiling code to TFs would slow down the overall simulation speed considerably.

Figure 7.8 shows how fragmented translation units are formed for CFG DBT based simulation. Figure 7.8a shows the control flow graph for an example target program and figure 7.8b shows the segment simulated during the first simulation epoch (shown in orange). The translation unit (CFG DBT mode) which might normally be identified at the end of the first simulation epoch is shown to the right in black.

Figure 7.8c shows what actually happens during the first simulation epoch. Any physical page boundaries disrupt the profiling process as the translation units are created on a per physical page basis and so any control arcs going from one page to another are lost (dashed orange lines). The boundaries between simulation epochs also disrupt profiling as the control arc leading to the next block to be simulated in the following simulation epoch will be lost. The result is that two smaller translation units (shown to the right in black) are formed at the end of the first simulation epoch instead of the single translation unit previously shown.

Figure 7.8d shows the program segment simulated during the second simulation epoch (shown in orange) and the corresponding translation unit normally identified to the

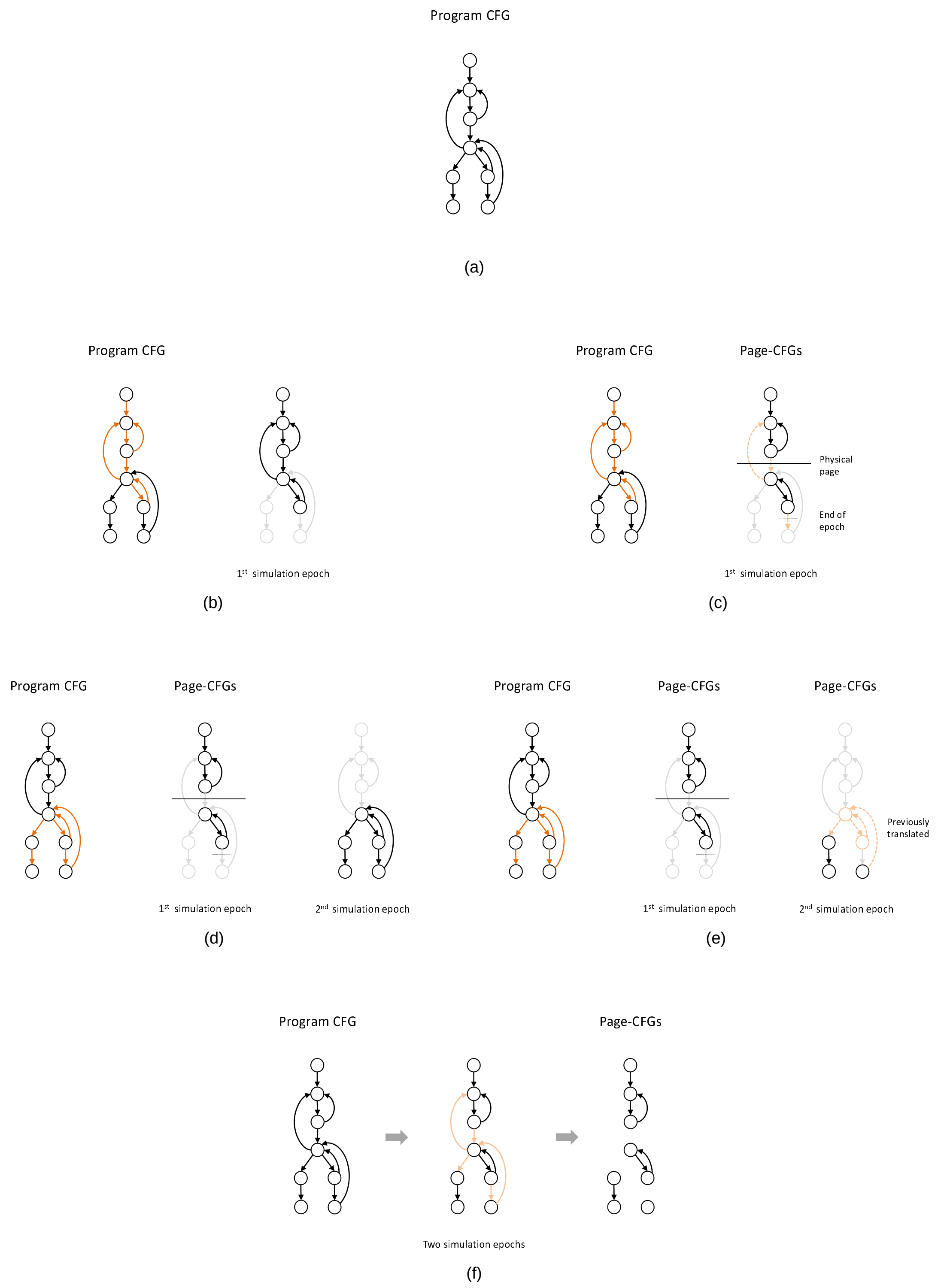


Figure 7.8 Translation Unit Fragmentation. These figures show the control flow graph for an example target program, the CFG LTUs identified in each simulation epoch and how they become fragmented. Orange lines show the program segments emulated in each simulation epoch and the dashed orange lines show the control arcs lost during profiling.

right (shown in black). However, profiling of the target program is further disrupted in the second simulation epoch now that TFs exist. Figure 7.8e shows that those sections of the program emulated by TFs (shown in light orange) divide potential translation units as they do not perform any profiling themselves. The result is the two small translation units shown on the right in black, instead of the single translation unit shown before. Translation unit fragmentation, as a consequence of existing TFs, has the potential to affect Page DBT based simulation more severely than SCC or CFG DBT simulation as Page based TFs are called to emulate any basic block which they contain, not just the root block.

Figure 7.8f shows the situation after two simulation epochs in which five control arcs have been lost (shown in orange) and the control flow graph for the target program has been divided into four small translation units. Translation unit fragmentation is affected by:

- Physical Page Boundaries
- Existing Translated Functions
- Simulation Epoch Size
- Translation Threshold
- Program Phase Changes

Translation unit fragmentation results in a larger number of smaller TFs which has a negative effect on simulation performance. Fewer basic blocks will be emulated within a given TF, requiring the simulator to return to the main simulation loop more frequently in order to find the next TF to call. More time spent searching for TFs in the main simulation loop results in slower simulation speeds.

Figure 7.9 outlines two different dynamic profiling techniques: interpretive and continuous profiling. Figure 7.9b shows the translation units identified using interpretive profiling which only profiles interpreted blocks. This is the method used by the EHS simulator and which is described above. Figure 7.9c shows the translation units identified using continuous profiling which profiles all basic blocks, whether they are interpreted or emulated by a TF. The continuous profiling technique can be used with any LTU DBT mode to reduce the effect of translation unit fragmentation.

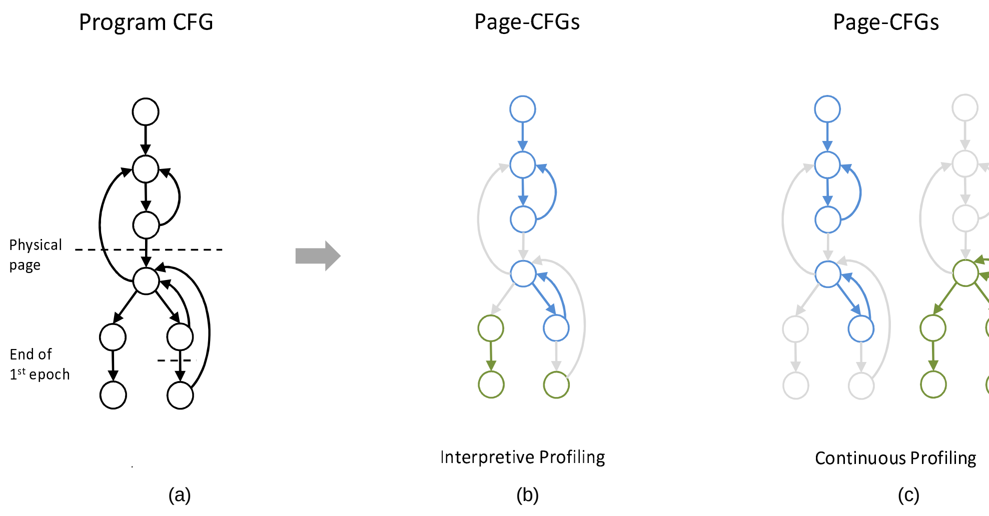


Figure 7.9 Translation Unit Profiling Techniques. Shows the (a) control flow graph for an example target program and the CFG LTUs identified by (b) interpretive only profiling and by (c) continuous profiling (interpretive and TF). The translation units identified in the first simulation epoch are shown in blue and those identified in the second simulation epoch are shown in green.

After two simulation epochs, the interpretive profiling technique identifies four translation units, the largest of which contains three blocks. The continuous profiling technique identifies three translation units, the largest of which contains five blocks. This demonstrates that continuous profiling is more immune to translation unit fragmentation than interpretive profiling. However, physical page and simulation epoch boundaries will still cause fragmentation. It may be that the overhead involved in continuously profiling the target program is less costly than the performance degradation resulting from translation unit fragmentation. An additional advantage of continuous profiling would be the ability to continuously monitor program behaviour and to respond to events such as phase changes.

7.6 Simulation Epoch Size

The effect of varying the size of the simulation epoch (interval between translations) on the average simulation speed and on the average static and dynamic TF sizes is shown in figure 7.10. The average performance statistics, for all of the benchmarks, were calculated using data obtained from the second simulation run in which all instructions are emulated by TFs.

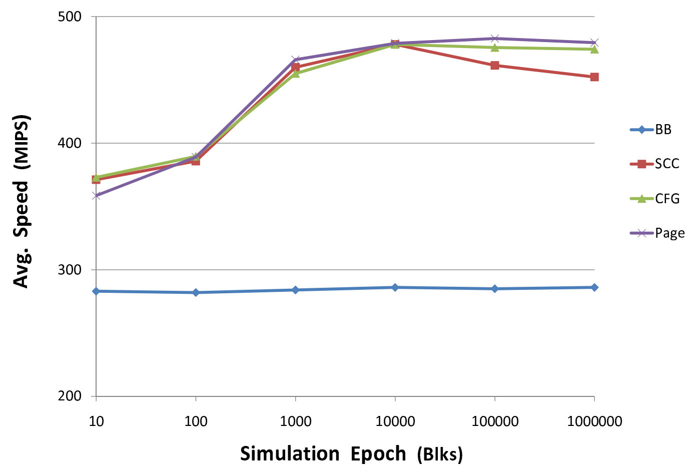
As expected, figure 7.10a shows that the average simulation speed for BB DBT based simulation is significantly below that of the LTU DBT simulation modes. The mean simulation speed for BB DBT remains constant at 283 MIPS for all simulation epochs.

For SCC DBT based simulation, the simulation speed increases from 371 MIPS at 10 blocks to 478 MIPS at 10,000 blocks and then slowly decreases to 452 MIPS at 1000,000 blocks. In CFG DBT based simulation, the simulation speed increases from 373 at 10 blocks to 474 MIPS at 1,000,000 blocks. In Page DBT based simulation, the simulation speed increases from 358 MIPS at 10 blocks to 482 MIPS at 100,000 blocks and above.

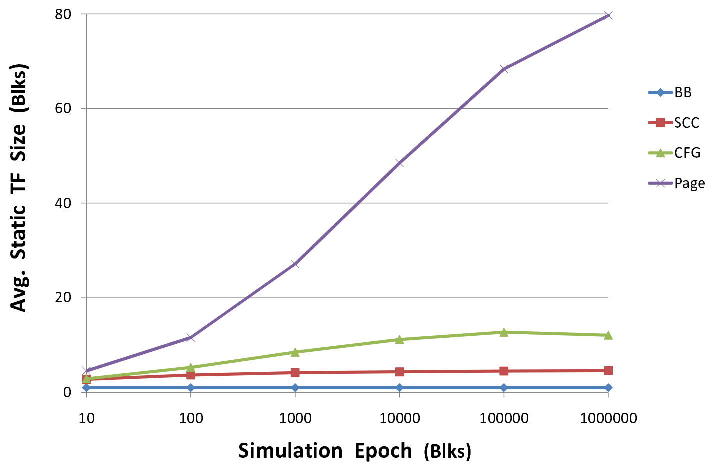
In general, for simulation epochs between 100 to 1,000,000 blocks, Page DBT based simulation is slightly faster than CFG DBT based simulation, which is in turn faster than SCC DBT based simulation. However there are a few exceptions, SCC DBT based simulation is slightly faster than CFG DBT based simulation at 1000 and 10,000 blocks. It is worth noting that the simulation speed for SCC DBT based simulation decreases slightly as the simulation epoch increases beyond 10,000 blocks.

Figure 7.10b shows that the average static TF size (number of basic blocks in a TF) increases for all LTU DBT modes as the size of the simulation epoch increases - the static size of BB TFs is always 1. The average static TF size for Page DBT based simulation increases from 4.5 at a simulation epoch of 10 blocks to 79.7 at a simulation epoch of 1,000,000 blocks. This demonstrates the ability of Page DBT based simulation to benefit from larger simulation epochs. Page DBT based simulation will continue to identify larger translation units until they contain whole physical pages.

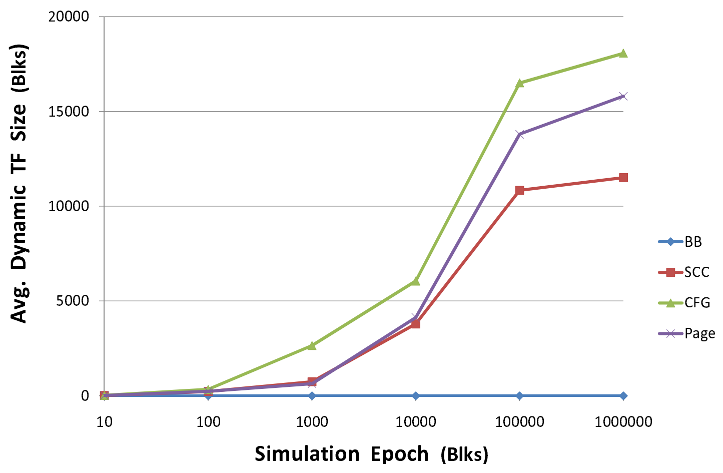
The average static TF size for CFG DBT based simulation increases slowly from an average of 2.9 at a simulation epoch of 10 blocks to 12.7 at a simulation epoch of 100,000 blocks. On average larger static TFs are generated as the simulation epoch is increased up to 100,000 blocks. However, increasing the simulation epoch beyond 100,000 blocks results in a slight decrease in the average static TF size, indicating that all of the CFGs that can be identified have been identified. The maximum average static TF size for CFG DBT based simulation is capped at 12.7 blocks.



(a) Simulation Speed



(b) Static TF Size



(c) Dynamic TF Size

Figure 7.10 Simulation Speed and TF Profiles. The graphs above show the mean (a) simulation speed, (b) static TF size and (c) dynamic TF size for the EEMBC benchmarks for varying simulation epochs.

The average static TF size for SCC DBT based simulation also increases very slowly from an average of 2.7 at a simulation epoch of 10 blocks to 4.6 at a simulation epoch of 100,000 blocks. The graph shows that increasing the simulation epoch beyond 100,000 blocks does not result in a larger average static TF size, indicating that all of the SCCs have been identified. The maximum average static TF size for SCC DBT based simulation is capped at 4.6 blocks.

Figure 7.10c shows that the average dynamic TF size (number of basic blocks emulated by a TF) increases substantially for all LTU DBT modes as the size of the simulation epoch increases - the dynamic size of BB TFs is always 1. The average dynamic TF size for CFG DBT based simulation increases from 24 at a simulation epoch of 10 blocks to 18,071 at a simulation epoch of 1,000,000 blocks. The average dynamic TF size increases from 18 to 15,809 for Page DBT based simulation and from 17 to 11,512 for SCC DBT based simulation.

Whilst Page DBT based simulation exhibits the highest average static TF size (followed by CFG DBT) at all simulation epoch sizes, it is CFG DBT based simulation which exhibits the highest average dynamic TF size at all epochs followed by Page DBT based simulation. A high degree of translation unit fragmentation must therefore be occurring in Page DBT based simulation for this to be the case.

Although CFG DBT based simulation exhibits the highest average dynamic TF size, it is none the less Page DBT based simulation which displays the best average simulation speeds for simulation epochs greater than 100 blocks. SCC DBT based simulation also performs well in so much as it emulates more benchmarks faster than any other DBT simulation mode. These results further illustrate the complex interactions which take place during simulation. The main factors affecting the speed of simulation are:

- **Program Behaviour**: The execution path of a target program may favour one DBT simulation mode over another. Translation unit fragmentation makes it difficult to predict which LTU DBT mode will perform the best given a particular benchmark.

- **Dynamic TF Size**: The greater the number of basic blocks emulated within TFs the faster, in theory, the simulation speed. The key is to keep emulation of the target program confined to as few TFs as possible as they have been optimized for speed of execution. This also reduces the number of times that control has to be passed back to the main simulation loop.

- **Host Hardware**: The system hardware, in particular the cache configuration, may benefit a particular DBT simulation mode more than the others.

In general, the average simulation speed for all LTU DBT simulation modes increases as the average dynamic TF size increases. However, the average simulation speed hits a ceiling of 480 MIPS as the simulation epoch is increased beyond 10,000 blocks (100,000 blocks for Page DBT). This upper limit in the average speed most likely reflects the physical restrictions imposed by the host platform as the dynamic TF size, which has a positive affect on simulation speed, is shown to increase as the epoch size is increased beyond 10,000 blocks. It is therefore reasonable to assume that faster simulation speeds are attainable with a more capable host machine.

7.7 Comparison with State-of-the-Art Simulators

The EHS simulator was compared with two state-of-the-art functional DBT based simulators, Simit-Arm and QEMU. Further details of both simulators, which model the ARM processor [ARMv], can be found in chapter 3. Figures 7.11 and 7.12 show the simulation speed and the time to completion for each benchmark respectively, running on the three different simulators. Table 7.3 provides a summary of the results. The EHS simulator was configured to run in one of its fastest setups for the host platform.

The simulation speeds (MIPS) for Simit-ARM and QEMU can not be directly compared with the EHS simulator as they simulate different ISAs. Not only do the number of instructions emulated using different ISAs differ, the instruction sets and the complexity of each instruction which must be modelled also differ.

The time taken to simulate each benchmark shows that QEMU is the fastest simulator overall. QEMU takes on average 650 milliseconds to emulate a benchmark, followed by the EHS simulator which takes 820 milliseconds, with Simit-ARM last taking 880

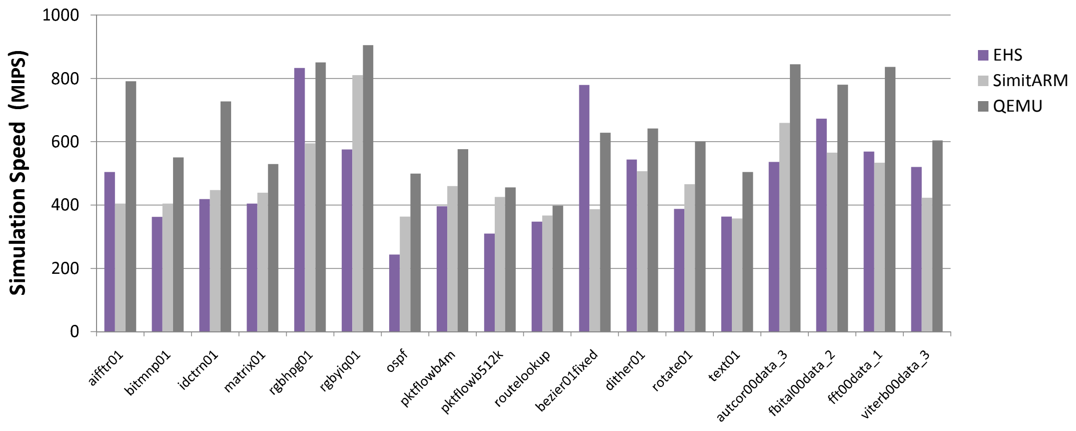


Figure 7.11 EHS, Simit-ARM and QEMU Simulation Speeds. This chart shows the functional simulation speeds of selected EEMBC benchmarks running on the EHS (Page DBT), Simit-ARM and QEMU simulators.

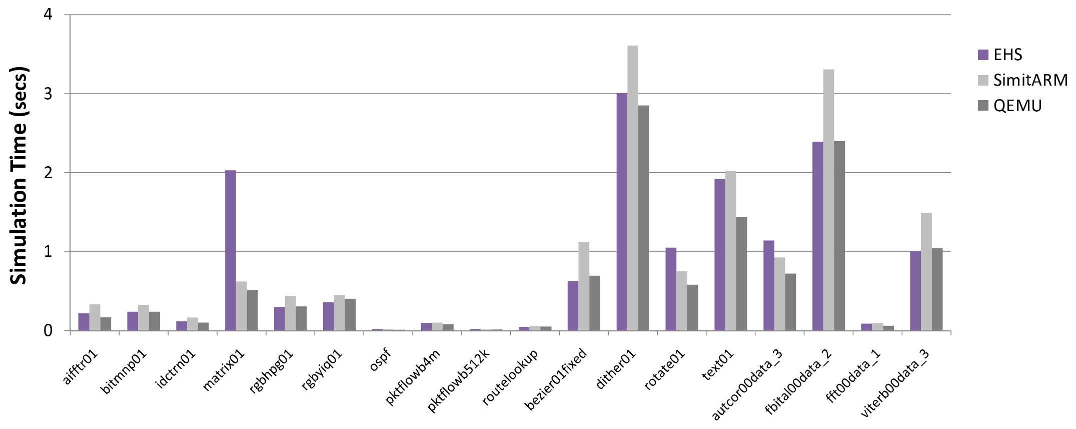


Figure 7.12 EHS, Simit-ARM and QEMU Simulation Times. This chart shows the functional simulation times of selected EEMBC benchmarks running on the EHS (Page DBT), Simit-ARM and QEMU simulators.

	EHS	Simit-ARM	QEMU
SPEED (MIPS)			
Slowest	244	358	398
Fastest	833	810	905
Median	462	443	627
Average	**483**	**477**	**651**
TIME (msecs)			
Shortest	20	150	13
Longest	3010	3610	2852
Median	330	4450	356
Average	**820**	**880**	**650**
INSTRUCTIONS			
Average	**2008**M	**2176**M	**2176**M

Table 7.3 EHS, Simit-ARM and QEMU Performance Summary. Simulation speed, real-time to complete execution and instruction count summary for a subset of the EEMBC benchmark suite running on the EHS, Simit-ARM and QEMU simulators. The EHS simulator was run in Page DBT mode with a 100,000 block simulation epoch and a translation threshold equal to 1.

milliseconds. QEMU simulates 12 out of the 18 benchmarks the fastest, followed by the EHS simulator which simulates 6 benchmarks the fastest. Simit-ARM does not simulate any of the benchmarks the fastest. Simit-ARM emulates 13 of the benchmarks the slowest, followed by the EHS simulator which simulates 5 of the benchmarks the slowest. The QEMU simulator does not simulate any of the benchmarks the slowest.

It is evident from these results that the simulation speed of the EHS simulator is comparable with best in class. The EHS simulator completes simulation of the benchmarks on average 1.07 times faster than Simit-ARM and 1.26 times slower than QEMU. This is impressive considering that the EHS simulator was developed as a flexible research simulator suitable for performing design-space exploration, unlike Simit-ARM and QEMU simulators which were designed purely for functional simulation speed.

7.8 Summary

The results for instruction level simulation across all benchmarks show that the LTU DBT simulation modes are on average at least 1.63 times faster than BB DBT based simulation (using a 1000 block simulation epoch). However, BB DBT based simulation of the bezierfixed benchmark is slightly faster than SCC DBT based simulation. Page DBT based simulation performed the best overall with a mean speed-up of 1.67 and SCC DBT based simulation simulated 9 out of the 20 benchmarks the fastest.

The combination of profiling the target programs' execution path at runtime and dynamic binary translation is shown to be an effective technique for high speed simulation of microprocessor systems. In all DBT simulation modes, less than 1% of the instructions simulated are interpreted on the first run (the rest are emulated by TFs) with all of the instructions being emulated by TFs on the second run.

The relationship between the average static TF size, the average dynamic TF size and the average simulation speed was investigated. It was observed, for all LTU DBT simulation modes, that the average dynamic TF size increased as the size of the simulation epoch increased. This research demonstrates that faster simulation speeds can be attained by increasing the number of instructions emulated on each TF call.

Chapter 8

Cycle Timing Simulation

This chapter investigates high speed DBT based cycle-accurate simulation and evaluates the research hypothesis by comparing the simulation speeds of the novel LTU DBT simulation techniques. Three cycle-approximate simulators designed to increase simulation speed are also explored. The simplified target models used in the cycle-approximate simulators are described in detail and their simulation speed and accuracy analyzed.

8.1 Overview

Cycle-accurate simulators not only emulate the target program, they model the system in sufficient detail so that the execution time of a program can be calculated in clock cycles. Cycle-accurate simulators facilitate low-level design space exploration, enabling a large number of different architectures to be tested with real-world applications. The information returned by a cycle-accurate simulator allows the performance of different system designs to be properly assessed and compared with one another. The system which best fulfils the design criteria can then be selected for fabrication.

Cycle accurate simulation involves modelling the precise behaviour of all system components in simulated time. For a microprocessor system this means accurately modelling the operation of at least the processor pipeline and memory sub-system. The memory latencies for instruction fetches and data reads and writes must be calculated and then inserted into the pipeline at the correct stage. Each stage of the pipeline must also be modelled, maintaining any inter-dependencies between pipeline stages.

Cycle-accurate simulators model the timing events within a system in detail and are therefore slow. Cycle-approximate simulators however, model a system less precisely which means that whilst they are many times faster than cycle-accurate simulators, they also introduce a greater degree of error into the data returned. Three different cycle-approximate versions of the EHS simulator are investigated. The first employs a simplified model of the processor pipeline, the second a simplified model of the memory sub-system and the third incorporates both of these simplified models. Cycle-approximate simulators are often used to test new system designs when the time available is limited.

In order to evaluate the research hypothesis the same set of benchmarks were run on the simulator operating in each of the different DBT simulation modes. The performance of the different DBT modes could then be quantitatively compared for cycle-accurate and cycle-approximate simulation. The following sections present the results from simulating a subset of the EEMBC benchmark suite [EEMB] on the Edinburgh High Speed simulator. The simulation speeds reported are in native MIPS: millions of target instructions simulated per real-time (host) second. Unless explicitly stated otherwise, the simulation epoch was set at 1000 blocks and the translation threshold set to 1.

8.2 Cycle Accurate Simulation Analysis

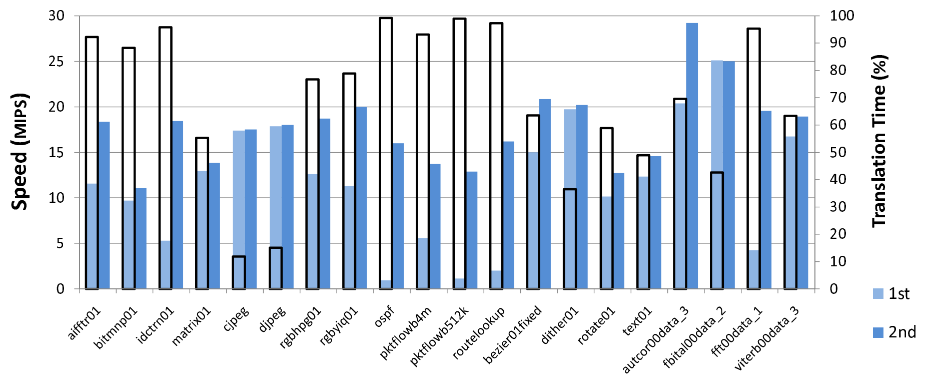
The cycle-accurate version of the EHS simulator models the 7-stage pipeline of the ARC 700 based processor which is detailed in figure 8.1. It shows the inter-stage dependencies and the points at which pipeline stalls can occur. The processor pipeline may stall as a result of memory latencies, experienced when fetching instructions and loading data, instruction execution latencies or source operand dependencies, experienced when waiting for the value of a source operand to be updated by a previous instruction. The pipeline and processor states are updated after each target instruction is simulated. The performance figures obtained from cycle-accurate simulation are used as a baseline to compare the relative performances of the new cycle-approximate models.

```
// Fetch Stage
pipeline[FETCH] += inst_fetch_cycles;
if (pipeline[FETCH] < pipeline[ALIGN])
    pipeline[FETCH] = pipeline[ALIGN];

// Align Stage
pipeline[ALIGN] = pipeline[FETCH] + 1;
if (pipeline[ALIGN] < pipeline[DECODE])
    pipeline[ALIGN] = pipeline[DECODE];

// Decode Stage
pipeline[DECODE] = pipeline[ALIGN] + 1;
if (pipeline[DECODE] < pipeline[REGISTER])
    pipeline[DECODE] = pipeline[REGISTER];

// Register File Stage
pipeline[REGISTER] = pipeline[DECODE] + 1;
pipeline[REGISTER] = max(REGISTER, reg_cycle[src_op1], reg_cycle[src_op2]);
if (pipeline[REGISTER] < pipeline[EXECUTE])
    pipeline[REGISTER] = pipeline[EXECUTE];

// Execute Stage
pipeline[EXECUTE] = pipeline[REGISTER] + inst_exe_cycles;
reg_cycle[dst_op1] = pipeline[EXECUTE];
if (pipeline[EXECUTE] < pipeline[MEMORY])
    pipeline[EXECUTE] = pipeline[MEMORY];

// Memory Stage
pipeline[MEMORY] = pipeline[EXECUTE] + memory_load_cycles;
reg_cycles[dst_op2] = pipeline[MEMORY];
if (pipeline[MEMORY] < pipeline[WRITEBACK])
    pipeline[MEMORY] = pipeline[WRITEBACK];

// Write-Back Stage
pipeline[WRITEBACK] = pipeline[MEMORY] + 1;
```

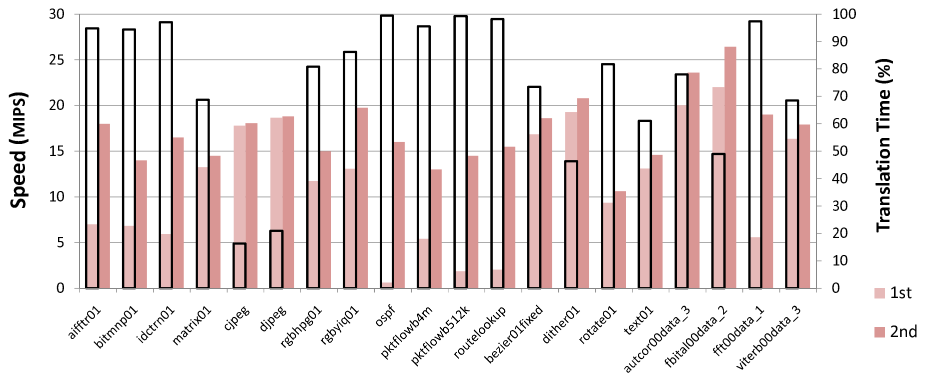
Figure 8.1 Cycle Accurate Pipeline Model. This figure shows the cycle-accurate pipeline model for both interpretive and DBT based simulation modes. The processor and pipeline states are updated after each instruction has been emulated. The `pipeline` structure holds the current state (instruction cycle) for each pipeline stage and the `reg_cycle` structure holds the availability (cycle) for each register.

The results of two consecutive simulation runs for each benchmark and DBT simulation mode are shown in figure 8.2. The simulation speeds for the second run range from 10 to 30 MIPS.

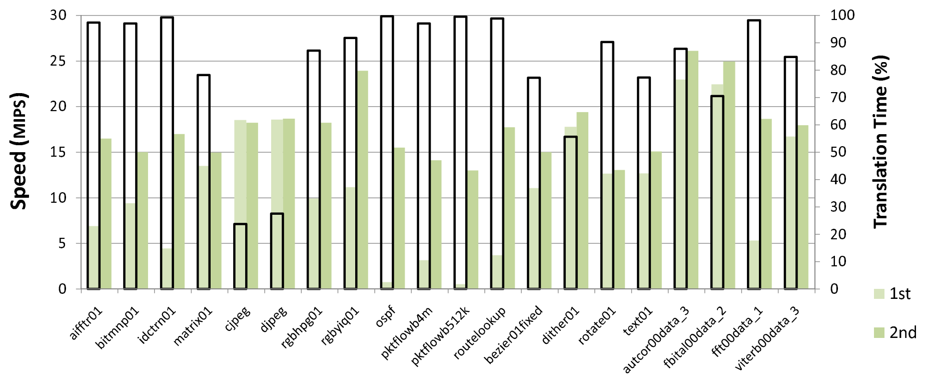
The increase in simulation speed from the first to the second simulation run is small for most benchmarks and all DBT simulation modes. For example, the simulation speed for the dither benchmark goes from 18 MIPS on the first run, to 19 MIPS on the second run when the simulator is operating in the CFG DBT simulation mode. This is in contrast to instruction level DBT based simulation which experiences a much bigger jump in speed from the first to the second run. The minimal speed increase on



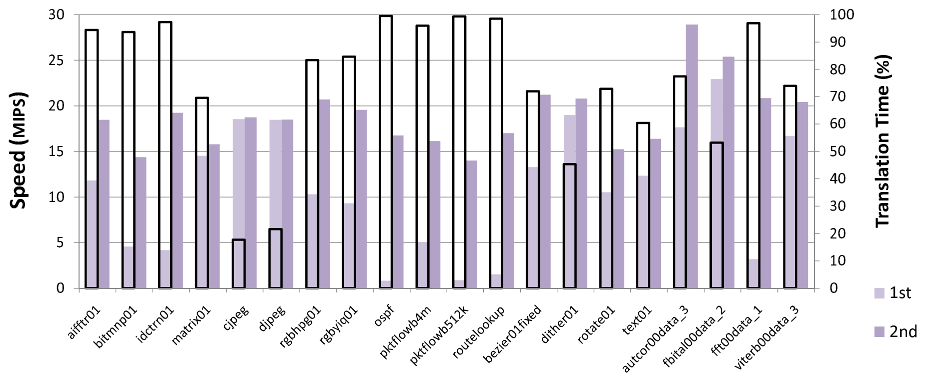
(a) BB



(b) SCC



(c) CFG



(d) Page

Figure 8.2 Cycle Accurate Simulation Profile. The figures show the simulation speed and the proportion of total simulation time spent performing translation (outlined bars) for two consecutive runs of each benchmark in each of the DBT modes.

the second simulation run can be attributed to the large overhead involved in modelling the cycle-accurate behaviour of the system. The pattern of simulation speeds across all benchmarks is similar for each DBT simulation mode and similar to that of instruction level DBT based simulation, albeit at much slower speeds.

As with instruction level DBT based simulation, the proportion of the overall simulation time spent performing translation during the first run is high. The proportion of time spent performing translation is more than 70% for 11 benchmarks running in BB DBT simulation mode, for 13 benchmarks in SCC DBT mode, for 17 benchmarks in CFG DBT mode and for 15 benchmarks in Page DBT simulation mode. The proportion of time spent performing translation for each benchmark is slightly higher in SCC DBT mode than in BB DBT mode, and slightly higher in CFG DBT mode than in SCC DBT mode, with figures for Page DBT mode very similar to SCC DBT mode. This is the same picture which emerged from instruction level DBT based simulation of the benchmarks.

The proportion of the total simulation time spent performing translation is significant for benchmarks which run for short time periods, and markedly less for those benchmarks which run for longer periods. As expected, no translation is performed on the second simulation run as all of the target instructions are translated on the first run. More than 99% of the instructions simulated on the first run are emulated by TFs, with all instructions being emulated by TFs on the second run.

Figure 8.3 shows the maximum cycle-accurate simulation speeds for the different DBT simulation modes and compares them with the interpretive simulation speeds. The corresponding speed-ups for each DBT mode, relative to interpretive simulation, are shown in figure 8.4. The interpretive cycle-accurate simulation speed for each benchmark remains pretty constant at about 12.5 MIPS. All of the benchmarks run faster when the simulator operates in one of the DBT simulation modes, with the exception of the rotate benchmark which runs slightly slower in SCC DBT simulation mode than it does interpretively.

Overall, the DBT simulation modes exhibit a mean speed-up of at least 1.45 over interpretive cycle-accurate simulation as summarized in table 8.1. Page DBT based simulation performs the best with a mean speed-up of 1.56, followed by SCC DBT simulation, then CFG DBT simulation and lastly BB DBT based simulation. Page

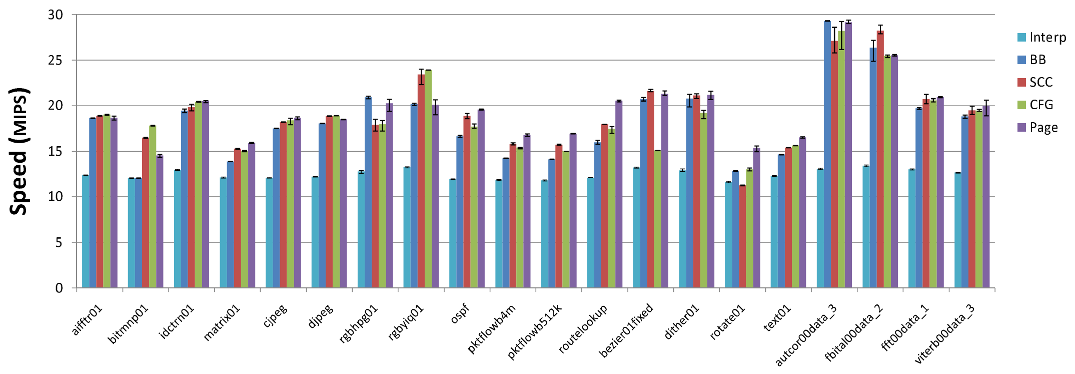


Figure 8.3 Cycle Accurate Simulation Speed. This figure shows the simulation speed for each benchmark using interpretive and DBT based simulation modes. The simulation speeds presented are for the main simulation loop. The speeds shown are the average of 10 simulation runs in which all target instructions had previously been translated.

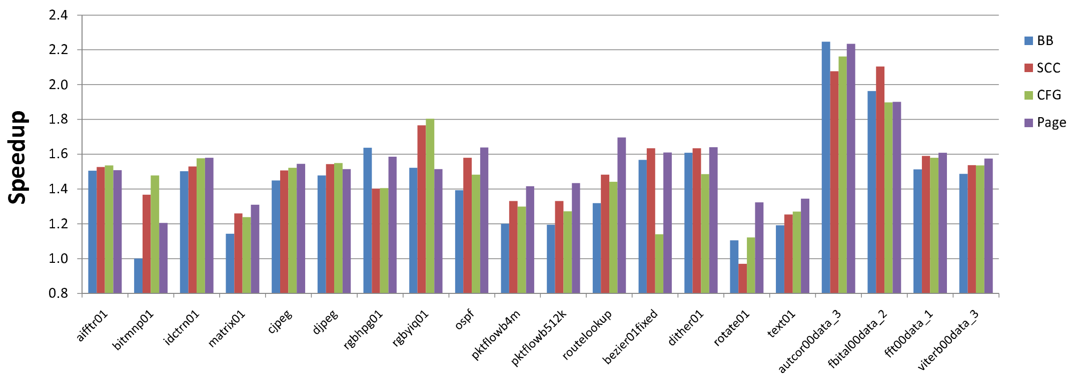


Figure 8.4 Cycle Accurate Simulation Speedup. This figure shows the simulation speed-ups for each benchmark for each DBT mode relative to interpretive simulation.

DBT based simulation is 1.08 times faster than BB DBT based simulation. Page DBT based simulation simulated 12 of the benchmarks fastest, followed by CFG DBT based simulation which simulated 4 of the benchmarks the fastest.

	Interpretive	BB	SCC	CFG	Page
SPEED (MIPS)					
Slowest	11.6	12.1	11.3	13.0	14.5
Fastest	13.4	29.3	28.2	28.2	29.1
Median	12.3	18.4	18.9	18.2	19.8
Average	**12.5**	**18.3**	**19.1**	**18.7**	**19.6**
SPEED-UP					
Geo. Mean	**1**	**1.45**	**1.52**	**1.49**	**1.56**
Goe S.D.	0	0.29	0.26	0.25	0.22

Table 8.1 Cycle Accurate Performance Summary. The geometric mean speed-up for each DBT mode is relative to interpretive simulation.

Figure 8.5 shows to what degree cycle-accurate modelling of the different system components contribute to the overall simulation time. In all simulation modes the amount of time spent modelling the memory sub-system is less than that required to model the pipeline. The proportion of simulation time spent modelling the pipeline is 30% for interpretive simulation and approximately 51% for all of the DBT simulation modes. The proportion of time required to model the memory hierarchy is 28% for interpretive simulation, 42% for BB DBT based simulation and approximately 44% for the LTU DBT simulation modes. In all simulation modes, approximately 14% of the time required to model the memory hierarchy is spent modelling data accesses, the remaining 86% is spent modelling instruction fetches.

The proportion of total simulation time devoted to cycle-accurate modelling is 58% for interpretive simulation, 94% for BB DBT based simulation and 96% for all of the LTU DBT simulation modes. As the time spent modelling the cycle-accurate operation of the system accounts for the vast majority of the simulation time in DBT based simulators, the best opportunities for increasing the simulation speed lie in optimizing the cycle-accurate models. Also, cycle-accurate modelling of the memory sub-system takes up almost as much time as modelling the pipeline, therefore reducing the time required to model both will result in significant increases in performance. This may be

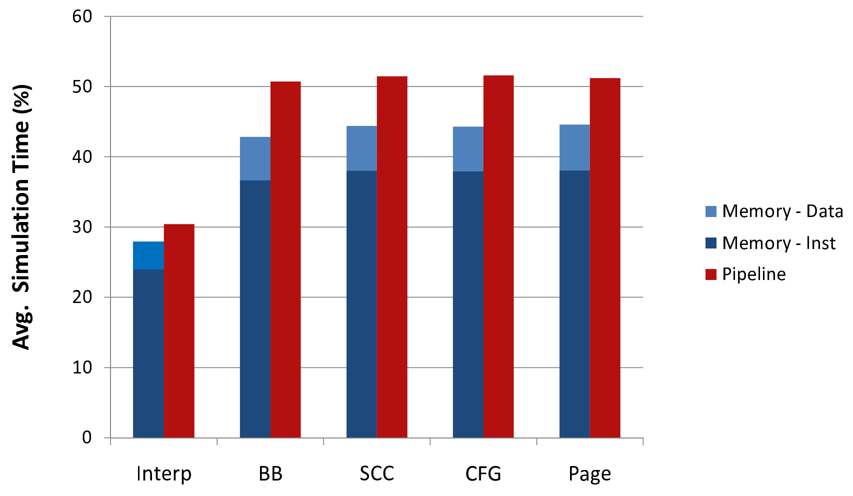


Figure 8.5 Cycle Accurate Pipeline and Memory Models. This graph shows the proportion of the total simulation time spent performing cycle-accurate modelling of the memory sub-system and processor pipeline. The results shown are averages for simulation of the EEMBC benchmark suite on the EHS simulator.

achieved by simplifying the target models for the pipeline and memory sub-system.

8.3 Cycle Approximate Simulation Analysis

This section introduces three new cycle-approximate simulation models designed to provide high-speed simulation and to generate statistics with minimal error. The first uses a simplified model for the processor pipeline, the second uses a simplified model for the memory sub-system and the third combines both of these cycle-approximate models.

8.3.1 The Pipeline Model

The cycle-approximate model for the ARC based processor pipeline is detailed in figure 8.6. The simplified pipeline models the inter-dependencies between the stages in which stalls may be initiated, namely the fetch, execute and memory stages. However, it does not model stalls caused by source register dependencies resulting from instruction execution, as the latencies involved are typically very small compared to those of instruction fetches and data loads from memory. As with the cycle-accurate pipeline model, the pipeline and processor states are updated after each target instruction is emulated.

```
// Fetch Stage
pipeline[FETCH] += inst_fetch_cycles;
if (pipeline[FETCH] < pipeline[EXECUTE] - 3)
    pipeline[FETCH] = pipeline[EXECUTE] - 3;

// Execute Stage
pipeline[EXECUTE] = pipeline[FETCH] + 3 + inst_exe_cycles;
if (pipeline[EXECUTE] < pipeline[MEMORY])
    pipeline[EXECUTE] = pipeline[MEMORY];

// Memory Stage
pipeline[MEMORY] = pipeline[EXECUTE] + memory_load_cycles;
```

Figure 8.6 Cycle Approximate Pipeline Model. This figure shows the cycle-approximate pipeline model. It is a simplified version of the pipeline which models the Fetch, Execute and Memory pipeline stages and ignores instruction register availability. The processor state is updated after each instruction has been emulated.

The simulation results for each DBT simulation mode are shown in figure 8.7 and summarized in table 8.2.

	Interpretive	BB	SCC	CFG	Page
SPEED (MIPS)					
Slowest	11.6	14.2	12.3	15.6	19.4
Fastest	13.4	49.0	53.8	54.3	54.5
Median	12.3	34.1	36.9	39.6	39.9
Average	**12.5**	**34.1**	**36.8**	**37.2**	**38.3**
SPEED-UP					
Geo. Mean	**1**	**2.57**	**2.76**	**2.81**	**2.94**
Geo. S.D.	0	0.79	0.90	0.87	0.75

Table 8.2 Cycle Approximate Pipeline Performance Summary. The geometric mean speed-up for each DBT mode is relative to the interpretive simulation speed.

Page DBT based simulation performs the best overall with a mean speed-up of 2.94 across all benchmarks (compared to cycle-accurate interpretive simulation) followed by CFG DBT based simulation then SCC DBT based simulation and lastly BB DBT based simulation. Page DBT based simulation simulated 14 benchmarks the fastest followed by SCC DBT based simulation which simulated 4 benchmarks the fastest. The relative mean absolute error (RMAE) in the cycle count for all of the benchmarks was 0.019 with standard deviation (RMAE SD) of 0.028.

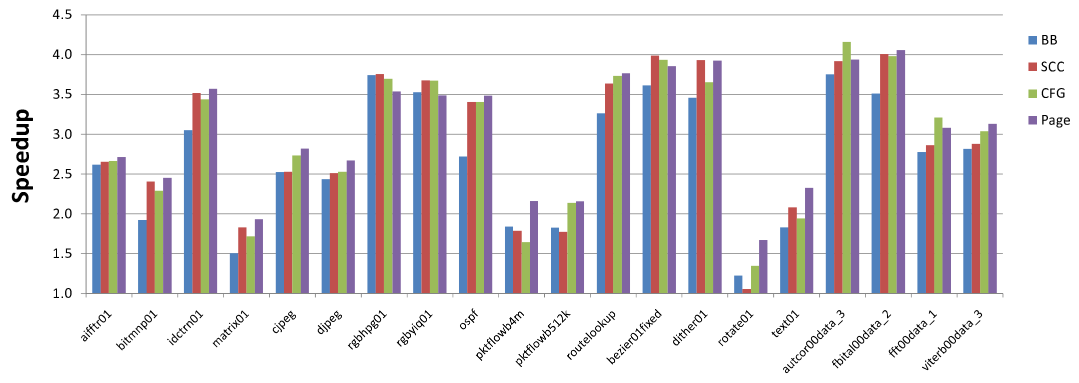


Figure 8.7 Cycle Approximate Pipeline Speedup. This figure shows the simulation speed-ups for each benchmark for each DBT mode relative to interpretive simulation.

8.3.2 The Memory Model

The cycle-approximate model for the memory sub-system uses a fixed number of cycles for each instruction fetch and data read from memory. The model performs a form of sampling, gathering cycle-accurate data for those instructions interpreted within each simulation epoch. The simulator maintains running totals for the number of clock cycles consumed by instruction fetches and data loads for each instruction. At the end of each simulation epoch, the average number of cycles (rounded integer) used to fetch and to load data are calculated for each instruction and then inserted into the cycle-approximate model of the pipeline prior to translation. As with the cycle-accurate pipeline model, the pipeline and processor states are updated after each instruction is emulated. Note that if the memory configuration for the target system is changed, all of the translations must be regenerated as the memory latencies are hard-coded into the TFs.

The simulation results for each DBT based simulation mode are shown in figure 8.8 and summarized in table 8.3. CFG DBT based simulation performs the best overall with a mean speed-up of 2.06 across all benchmarks (compared to cycle-accurate interpretive simulation) followed by SCC DBT based simulation then Page DBT based simulation and lastly BB DBT based simulation. Page DBT and CFG DBT based simulation both simulated 7 benchmarks the fastest followed by SCC DBT which simulated 4 benchmarks the fastest.

	Interpretive	BB	SCC	CFG	Page
SPEED (MIPS)					
Slowest	11.6	13.4	11.5	11.9	16.5
Fastest	13.4	41.2	43.7	44.2	42.6
Median	12.3	25.5	26.2	27.1	25.8
Average	**12.5**	**25.4**	**26.8**	**26.9**	**26.0**
SPEED-UP					
Geo. Mean	**1**	**1.95**	**2.04**	**2.06**	**2.03**
Geo. S.D.	0	0.52	0.60	0.61	0.46

Table 8.3 Cycle Approximate Memory Performance Summary. The geometric mean speed-up for each DBT mode is relative to the interpretive simulation speed.

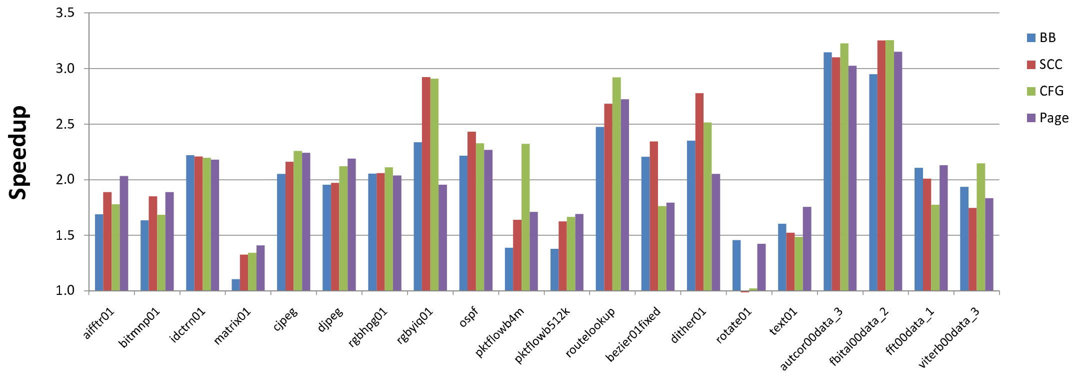


Figure 8.8 Cycle Approximate Memory Speedup. This figure shows the simulation speedups for each benchmark for each DBT mode relative to interpretive simulation.

Figure 8.9 shows the relative mean absolute error in the cycle count for different simulation epochs. Not surprisingly, the average error in the cycle count decreases as the size of the simulation epoch is increased. The RMAE is shown to jump from 0.142 with a simulation epoch of 1000 blocks down to 0.012 at 100,000 blocks (RMAE SD = 0.015).

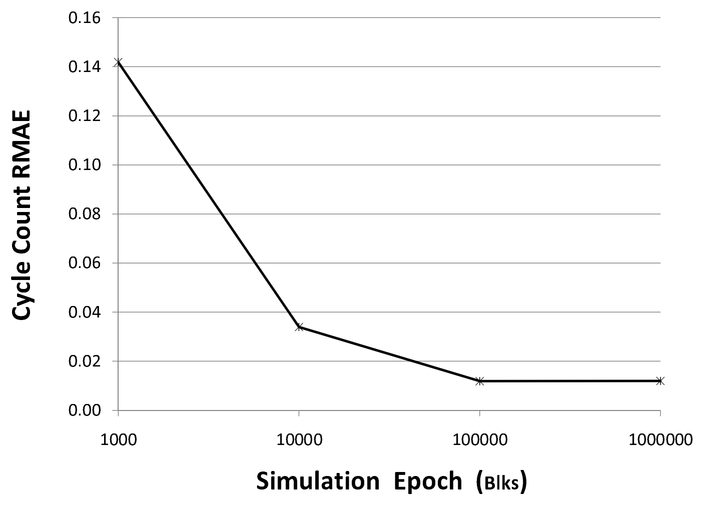


Figure 8.9 Cycle Approximate Memory Cycle Count Error. This figure shows the relative mean absolute error (RMAE) in the cycle count for the benchmarks for varying simulation epochs.

8.3.3 The System Model

The system cycle-approximate model combines the cycle-approximate models for the pipeline and memory sub-system described in the previous sections. The speeds for the different modes of simulation are shown in figure 8.10 and the speed-ups shown in figure 8.11. Table 8.4 provides a summary of the performance results.

	Interpretive	BB	SCC	CFG	Page
SPEED (MIPS)					
Slowest	11.6	29.3	55.1	56.4	57.1
Fastest	13.4	125.7	137.5	136.1	133.8
Median	12.3	85.9	109.0	112.6	115.8
Average	**12.5**	**83.7**	**107.4**	**107.8**	**109.3**
SPEED-UP					
Geo. Mean	**1**	**6.33**	**8.37**	**8.40**	**8.51**
Geo. S.D.	0	1.88	1.66	1.68	1.78

Table 8.4 Cycle Approximate System Performance Summary. The geometric mean speed-up for each DBT mode is relative to the interpretive simulation speed. Results are from simulation using a 100,000 block simulation epoch.

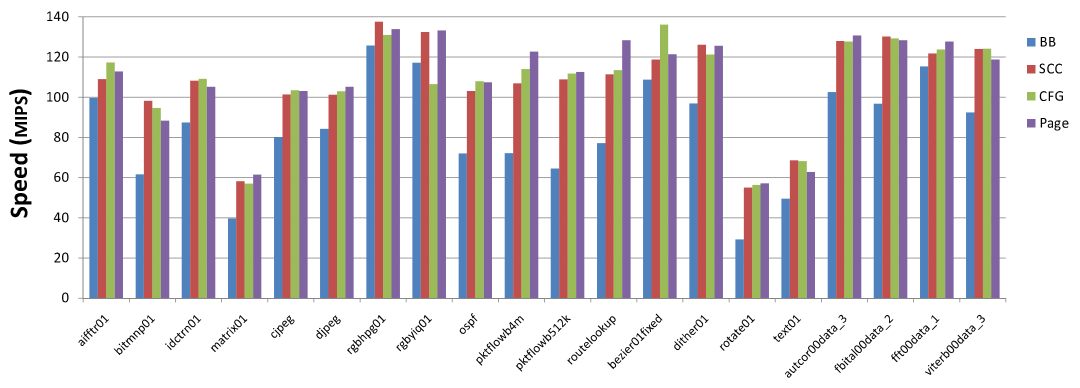


Figure 8.10 Cycle Approximate System Simulation Speed. This figure shows the simulation speed for each benchmark for all DBT based simulation modes. The simulation speeds presented are for the main simulation loop. The speeds shown are the average of 10 simulation runs in which all target instructions had previously been translated. Results are from simulation using a 100,000 block simulation epoch.

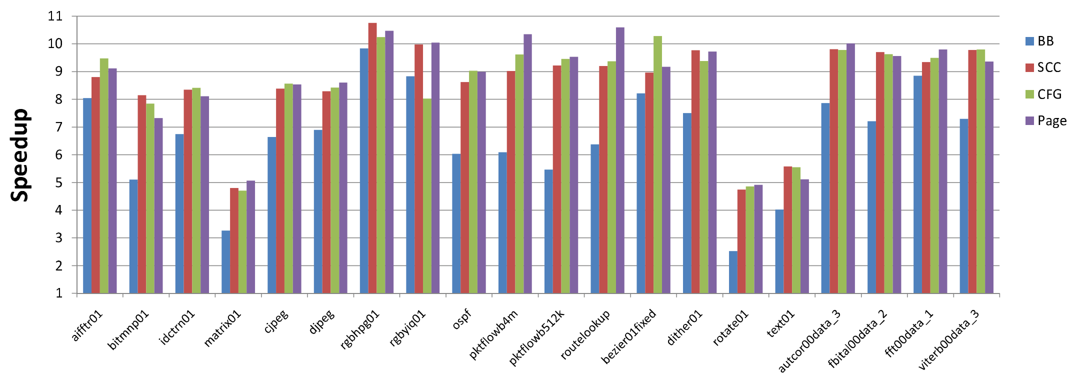


Figure 8.11 Cycle Approximate System Speedup. This figure shows the simulation speedups for each benchmark for each DBT mode relative to interpretive simulation. Results are from simulation using a 100,000 block simulation epoch.

The simulations were carried out using a simulation epoch of 100,000 blocks and a translation threshold of 1. The size of the simulation epoch was chosen to minimize the cycle count error introduced by the cycle-approximate memory model.

The LTU DBT based cycle-approximate simulation speeds ranged from 55.1 MIPS for the rotate benchmark in SCC DBT simulation mode up to 137.5 MIPS for the rgbhpg benchmark also in SCC DBT simulation mode. Page DBT based simulation performs the best overall with a mean speed-up of 8.51 and standard deviation of 1.78, followed by CFG DBT based simulation then SCC DBT based simulation and lastly BB DBT based simulation. Page DBT based simulation simulated 9 benchmarks the fastest followed by CFG DBT based simulation which simulated 6 benchmarks the fastest. Page DBT based simulation is on average 1.34 times faster than BB DBT based simulation and 8.51 times faster than interpretive simulation.

The errors in the cycle count for each benchmark are shown figure 8.12. The cycle count errors range from 8.76% for the rgbyiq benchmark to -8.08% for the ospf benchmark. The RMAE in the cycle count for all benchmarks is 0.024 with standard deviation of 0.026. It is impossible to work out what proportion of the cycle count error is attributable to which of the cycle-approximate models. This is because inter-dependencies still exist between the simplified pipeline and memory models.

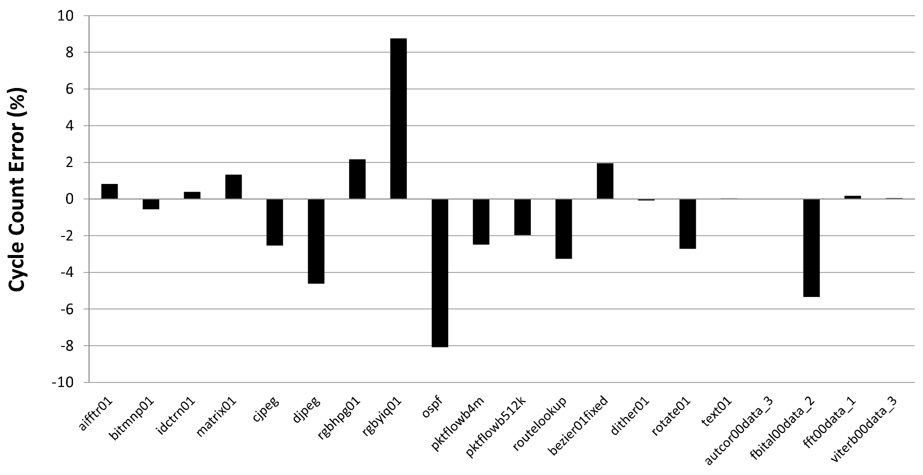


Figure 8.12 Cycle Approximate System Cycle Count Errors. This graph shows the cycle count errors for each benchmark. Results are from simulation using a 100,000 block simulation epoch.

8.3.4 Comparison with Sampling

One of the requirements for a research simulator is that it supports hardware/software co-design and verification of microprocessor systems. Whilst sampling based simulators are very good for performing low level DSE, because they can perform cycle-approximate simulation at speeds many times faster than DBT based simulators, they do not provide state observability. Dynamic binary translation based simulators on the other hand do support observability which is a pre-requisite for hardware/software co-design and verification.

8.4 Summary

The results for cycle-accurate simulation show that the DBT based simulation modes (1000 block simulation epoch) are on average at least 1.45 times faster than interpretive cycle-accurate simulation. Page DBT based simulation is the fastest and is 1.56 time faster than interpretive simulation and 1.08 times faster than BB DBT based simulation. The sole exception is the rotate benchmark which runs slightly slower in SCC DBT simulation mode than it does when simulated interpretively.

The results for cycle-approximate simulation, using simplified models of the processor pipeline and memory sub-system, show that the LTU DBT based simulation modes (100,000 block simulation epoch) are on average at least 8.37 times faster than interpretive cycle-accurate simulation. Page DBT based simulation is the fastest and is 8.51 times faster than interpretive simulation and 1.34 times faster than BB DBT based simulation. The average error in the cycle count was shown to be 2.4%.

Chapter 9

Conclusion

This thesis presents new techniques to speed up instruction level and cycle-approximate DBT based simulation of microprocessor systems. The research hypothesis states that faster simulation speeds can be realized by identifying and translating larger sections of the target program at runtime. This is accomplished by discovering LTUs within control-flow graphs generated for the target program during simulation. This research shows that the new LTU DBT simulation techniques provide significant increases in simulation speed over that attainable using basic block DBT based simulation.

The application of DBT techniques to cycle-accurate simulation is shown to provide only moderate increases in the simulation speed. However, this thesis describes how simplified timing models can be deployed to achieve significant speed-ups in cycle-approximate DBT based simulation. Simplified models of the target processor pipeline and memory sub-system are shown to result in high speed simulation across all DBT simulation modes whilst maintaining a high degree of accuracy.

This chapter summarizes the main contributions to research, provides a critical analysis of the work and outlines future research.

9.1 Contributions to Research

The contributions to the field of high speed DBT based simulation are outlined in the following sub-sections.

9.1.1 High Speed Simulation Techniques

This thesis proposed LTUs as a means of increasing the simulation speed of DBT based research simulators. Large translation units, which are described in chapter 6, are based on standard computer software objects and consist of one or more basic blocks. The techniques used to profile the target program and to identify and translate LTUs at runtime are unique to the Edinburgh High Speed simulator.

The EHS simulator was developed as a research simulator for the purpose of performing high speed design-space exploration and hardware/software co-design of novel processor architectures. The processor state is updated after each instruction is emulated and is observable at every translation unit boundary. The simulator can perform both process and system level simulation, and incorporates advanced management of cached translations to support the emulation of self-modifying code.

Three different types of LTU were investigated. The LTUs consisted of either strongly connected components, control-flow graphs or physical pages. Target programs are initially interpreted during which time a profile of the program's execution path is built-up. At the end of each simulation epoch the program's path profile is analyzed in order to extract the translation units prior to translation. Large translation units are identified and stored on a per physical page basis to facilitate the simulation of complex software including operating systems.

Increasing the size of the translation unit provides the translator with greater scope to optimize the binary code generated for speed. It also means that more target instructions are emulated per translated function call. As a result the simulator returns to the main simulation loop on fewer occasions, spending less time searching for the next translated function to call. Both of these factors contribute towards increasing the overall simulation speed.

9.1.2 Analysis of Simulation Techniques

The simulation characteristics of the different LTU DBT modes are analyzed in order to provide an insight into their effectiveness and future potential as high speed simulation techniques. Many aspects of LTU DBT based simulation are investigated in detail in chapters 7 and 8, including analysis of the number of instructions emulated by a translated function on the first and second simulation runs; the time spent performing the different simulation tasks; the size and number of translation units generated; the size and frequency of the translated functions called; the factors which cause translation unit fragmentation and the effects of varying the size of the simulation epoch.

During instruction level and cycle-accurate simulation of the EEMBC benchmark suite, less than 1% of all instructions emulated are interpreted on the first run for all benchmarks. However on the second run, all of the instructions are emulated by translated functions which highlights the efficiency of DBT based simulation. For the majority of benchmarks, 70% or more of total simulation time is spent performing translation on the first run for all DBT simulation modes. This demonstrates that LTU DBT based simulation is almost as efficient as basic block DBT based simulation in terms of the translation overhead incurred. The proportion of the overall simulation time spent performing translation is primarily dependent upon DBT mode, benchmark behaviour and simulation duration.

Whilst the average static translated function size is shown to increase for all LTU DBT simulation modes as the size of the simulation epoch is increased, the average dynamic translated function size only increases significantly for Page based DBT. The distribution of dynamic translated function sizes indicates that a high degree of translation unit fragmentation is occurring across all the DBT simulation modes. However, the simulation speed is shown to increase as the dynamic, and static, translated function size increases. It is likely that faster simulation speeds can be attained - particularly if using a larger simulation epoch - by employing a more capable host machine. If continuous profiling were implemented this would prevent translation unit fragmentation from occurring. The larger static and dynamic translated functions generated as a result should produce a corresponding increase in the simulation speed.

9.1.3 Instruction Level Performance

The instruction level simulation performance of the different LTU DBT modes are compared in chapter 7. The results, for simulation of a subset of the EEMBC benchmark suite, show that all of the LTU DBT simulation modes are at least 1.63 times faster on average than basic block DBT based simulation. Page DBT based simulation performs the best overall with an average speed-up of 1.67, followed by CFG DBT with a speed-up of 1.64, then SCC DBT with a speed-up of 1.63. Page DBT based simulation is also shown to be 14.8 times faster on average than interpretive simulation.

9.1.4 Cycle Approximate Performance and Accuracy

The cycle-approximate simulation performance of the different LTU DBT modes are compared in chapter 8. The results, for simulation of benchmarks running on a simplified model of the target system (pipeline and memory sub-system), show that all of the LTU DBT simulation modes are at least 1.32 times faster on average than basic block DBT based simulation. Page DBT based simulation performs the best overall with a speed-up of 1.34, followed by CFG DBT with a speed-up of 1.33, then SCC DBT with a speed-up of 1.32. Page DBT based simulation is also shown to be 8.51 times faster than interpretive cycle-accurate simulation. It is worth noting that much faster simulation speeds could be achieved by updating the cycle-approximate models on exiting the translated functions, rather than after each instruction.

The simplified models of the target pipeline and memory sub-system introduce small errors into the cycle count. The average cycle count error introduced by the simplified pipeline model is shown to be 1.9%. The cycle count error introduced by the simplified memory model is shown to decrease dramatically as the simulation epoch increases, from an average of 14% at 1000 blocks down to 1.2% at 1,000,000 blocks. The average cycle count error introduced by both simplified models is 2.4% when using a 100,000 block simulation epoch.

9.1.5 Comparison with State-of-the-Art Simulators

In addition to being a flexible research simulator, the EHS simulator is capable of performing instruction level simulation at speeds comparable with other state-of-the-art simulators that have been designed purely for speed (see chapter 7). The real-times taken to simulate a set of benchmarks were compared with Simit-ARM and QEMU, two functional simulators which model the ARM processor. The results show that the EHS simulator completed simulation of the benchmarks on average 1.07 times quicker than Simit-ARM and 1.26 times slower than QEMU. The QEMU simulator completed two-thirds of the benchmarks the quickest with the EHS simulator completing the remaining benchmarks the quickest.

9.2 Critical Analysis

All of the simulations were performed on a host machine running the Linux operating system. Whilst the host machine was running minimal system services during simulation, the non-deterministic effects of the operating system and the computer hardware introduce small errors into the simulation times recorded for each benchmark. In order to minimize such timing variations every simulation was repeated 10 times and the average simulation time calculated.

All of the research results were calculated from data generated by the Edinburgh High Speed simulator. Any errors in the cycle-accurate models of the target processor or memory sub-system are insignificant as they affect all of the DBT simulation modes equally. The simulator was used primarily to compare the relative performance of the different LTU DBT simulation modes.

9.3 Future Research

A number of questions arise from this research which may be followed up with further work. The most interesting areas for investigation are detailed in the following subsections.

9.3.1 Runtime Profiling

The ability to continuously monitor the target program's execution path has a number of advantages. Continuous profiling prevents LTU fragmentation from occurring as it supports profiling within translated functions. This results in the generation of larger translation units which in turn increases the average dynamic translated function block size. Larger LTUs provide greater scope for optimization during translation and enable larger sections of the target program to be emulated within a single translated function call. If the increase in speed outweighs the additional overhead of performing profiling within translated functions then faster simulation speeds will be realized. Even if this is not the case two versions of each translated function could be generated, a fast version which performs no profiling and a slower profiling version. Once the simulator recognizes that no more instructions are being interpreted it can switch from calling profiling translated functions to calling fast translated functions. If the simulator finds itself interpreting instructions again, indicating a change in the program execution path, it can switch back to calling the profiling translated functions so that the new paths can be traced.

Continuous profiling would enable the simulator to respond to changes in the target program's execution pattern. The simulator could be alerted to program phase changes and decide to identify the new hot paths by calling the profiling versions of the translated functions. Once the the new hot paths had been discovered the simulator could then discard the existing translated functions and generate new ones containing the hot paths.

9.3.2 Cycle Approximate Simulation

The results presented for cycle-approximate simulation were obtained using simplified models of the processor pipeline and memory sub-system which are called after each instruction is emulated. Calling the cycle-approximate models frequently involves a lot of processing which slows down simulation. With LTU DBT based simulation the opportunity arises to update the cycle-approximate models at the end of each basic block or on exiting translated functions, possibly after having emulated many thousands of target instructions. Reducing the number of times that the cycle-approximate models are called would increase the simulation speed significantly. The main challenge is to separate the pipeline and memory models whilst maintaining a high degree of accuracy.

Although sampling based simulators do not support hardware/software co-design they can perform cycle-approximate simulation many times faster than DBT based simulators. This is why sampling based simulators are more suited to performing low level DSE in which the design-space to be explored is very large.

9.3.3 Simulation Characterization

In order to obtain a deeper understanding as to why individual benchmarks run faster when the simulator is operating in a particular DBT simulation mode it would be instructive to quantify the interaction between the host machine, program behaviour and the DBT mode. The configuration of the host machine's L1 cache can for example have a significant impact on the simulation speed, being dependent on how much of the translation cache fits into it. The LTU DBT simulation modes are also sensitive to program phase changes and to changes in workload which can adversely affect simulation speeds.

Processor cycles need to be attributed to processes and hardware events during simulation in order to ascertain where and why cycles are being consumed. Armed with this knowledge a simulator could intelligently switch between the different DBT modes at runtime in response to changes in program behaviour so as to maintain the fastest possible simulation speed.

Appendix A

Glossary

Basic Block	A basic block is a maximal sequence of instructions such that none except the first is a branch target, and none except the last is a branch.
Cycle Accurate	Cycle-accurate simulation emulates a target program's behaviour in the same manner as an instruction level simulator. In addition, it accurately models the state and timing of the target system's micro-architecture. This feature is required in order to support low level DSE.
Cycle Approximate	Cycle-approximate simulation is very similar to cycle-accurate simulation except that it uses simplified models for components of the target system. Whilst this increases the simulation speed, the simulation statistics generated are typically less accurate.
Design Space	The design space is the set of all micro-architecture designs, compiler optimizations and benchmarks to be explored.
DSE	Design space exploration.
Functional	See instruction level simulation.
Host Machine	The hardware platform on which the simulator is run.

Instruction Level	Instruction level simulation emulates a target program by carrying out the instruction operations within the simulation environment. The simulator need only model the target processor in enough detail to ensure that each instruction is emulated correctly. This feature is required in order to support high level DSE.
LTU	A large translation unit consists of one or more basic blocks and is based on a standard software object.
Simulator	The executable which simulates the running of the target program on a model of the target system.
State Observability	State observability enables the state of the target system to be ascertained at precise moments in simulated time. This feature is required in order to support hardware/software co-design and verification.
Statically Discoverable	A statically discoverable program is one in which all possible execution paths can be identified through static analysis of the target binary. Programs which are self-modifying or which use shared libraries are not statically discoverable.
Target Binary	The executable to be emulated by the simulator.
Target System	The hardware system modelled by the simulator on which the target program is emulated.
Translated Function	A host code function which when called emulates the instructions in the corresponding target code section.
Translation Unit	The target program code objects which are identified at runtime for translation.
Translator	The simulator component which translates target code sections into translated functions.

Bibliography

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Altm 00] E. R. Altman, D. Kaeli, and Y. Sheffer. “Guest Editors’ Introduction: Welcome to the Opportunities of Binary Translation”. *Computer*, Vol. 33, No. 3, pp. 40–45, 2000.

[Altm 01] E. R. Altman, K. Ebcioglu, K. E. glu, M. Gschwind, S. Member, and S. Sathaye. “Advances and Future Challenges in Binary Translation And Optimization”. In: *Proc. of the IEEE*, pp. 1710–1722, 2001.

[Andr 92] K. Andrews and D. Sand. “Migrating a CISC computer family onto RISC via object code translation”. In: *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pp. 213–222, ACM Press, New York, NY, USA, 1992.

[ARCo] *ARCompact*$^{\text{TM}}$ *Instruction Set Architecture*. ARC International, 2025 Gateway Place, Suite 140, San Jose, CA 95110, USA.

[ARMv] *ARMv5*$^{\text{TM}}$ *Architecture Reference Manual*. ARM Ltd, 110 Fulbourn Road, Cambridge, England.

[Aust 02] T. Austin, E. Larson, and D. Ernst. “SimpleScalar: An Infrastructure for Computer System Modeling”. *Computer*, Vol. 35, No. 2, pp. 59–67, 2002.

[Bart 06] D. Bartholomew. “QEMU: a multihost, multitarget emulator”. *Linux J.*, Vol. 2006, No. 145, p. 3, 2006.

[Bell 05] F. Bellard. “QEMU, a fast and portable dynamic translator”. In: *ATEC ’05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 41–41, USENIX Association, Berkeley, CA, USA, 2005.

[Berg 87] A. B. Bergh, K. Keilman, D. J. Magenheimer, and J. A. Miller. “HP 3000 Emulation on HP Precision Architecture Computers”. *j-HEWLETT-PACKARD-J*, Vol. 38, No. 11, pp. 87–89, Dec. 1987.

[Brau 01] G. Braun, A. Hoffmann, A. Nohl, and H. Meyr. "Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description". In: *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pp. 57–62, ACM Press, New York, NY, USA, 2001.

[Brau 04] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. "A universal technique for fast and flexible instruction-set architecture simulation". *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 23, No. 12, pp. 1625–1639, 2004.

[Brun 91] R. A. Brunner, Ed. *VAX architecture reference manual (2nd ed.)*. Digital Press, Newton, MA, USA, 1991.

[Burg 96] D. Burger, T. M. Austin, and S. Bennett. "Evaluating Future Microprocessors: The SimpleScalar Tool Set". Tech. Rep. CS-TR-1996-1308, 1996.

[Burg 97] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0". *SIGARCH Comput. Archit. News*, Vol. 25, No. 3, pp. 13–25, 1997.

[Cher 98] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. "FX!32: A Profile-Directed Binary Translator". *IEEE Micro*, Vol. 18, No. 2, pp. 56–64, 1998.

[Cifu 96] C. Cifuentes and V. M. Malhotra. "Binary Translation: Static, Dynamic, Retargetable?". In: *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pp. 340–349, IEEE Computer Society, Washington, DC, USA, 1996.

[Cmel 94] B. Cmelik and D. Keppel. "Shade: a fast instruction-set simulator for execution profiling". In: *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pp. 128–137, ACM Press, New York, NY, USA, 1994.

[DErr 06] J. D'Errico and W. Qin. "Constructing portable compiled instruction-set simulators: an ADL-driven approach". In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pp. 112–117, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006.

[Dixi 92] K. M. Dixit. "New CPU benchmark suites from SPEC". In: *COMPCON '92: Proceedings of the thirty-seventh international conference on COMPCON*, pp. 305–310, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992.

[EEMB] *EEMBC Benchmark Suite*. The Embedded Microprocessor Benchmark Consortium.

[Free 95] *FreePort Express*. Digital Equipment Corporation, 1995.

[Gold 73] R. P. Goldberg. “Architecture of virtual machines”. In: *Proceedings of the workshop on virtual computer systems*, pp. 74–112, ACM, New York, NY, USA, 1973.

[Half 94] T. R. Halfhill. “Emulation: RISC’s Secret Weapon”. *BYTE*, Vol. 19, No. 4, pp. 119–130, 1994.

[Hand 98] J. Handy. *The cache memory book (2nd ed.): the authoritative reference on cache design*. Academic Press, Inc., Orlando, FL, USA, 1998.

[Hech 77] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[Henn 02] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Henn 96] J. L. Hennessy and D. A. Patterson. *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Hook 97a] R. Hookway. “DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT”. *Computer Conference, IEEE International*, Vol. 0, p. 37, 1997.

[Hook 97b] R. J. Hookway and M. A. Herdeg. “DIGITAL FX!32: Combining Emulation and Binary Translation”. *Digital Technical Journal*, Vol. 9, No. 1, 1997.

[Hsu 89] P. Hsu. *Introduction to Shadow*. Sun Microsystems Inc., 1989.

[Hunt 89] C. Hunter and J. Banning. “DOS at RISC”. *BYTE*, Vol. 14, No. 12, pp. 361–368, 1989.

[Jone 09] D. Jones and N. Topham. “High Speed CPU Simulation using LTU Dynamic Binary Translation”. In: *HiPEAC ’09: Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, Paphos, Cyprus, 2009.

[Kahl 05] J. Kahle. “The Cell Processor Architecture”. In: *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, p. 3, IEEE Computer Society, Washington, DC, USA, 2005.

[Kane 88] G. Kane. *MIPS RISC Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Kong 05] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: A 32-Way Multithreaded Sparc Processor”. *IEEE Micro*, Vol. 25, No. 2, pp. 21–29, 2005.

[Kron 93] N. Kronenberg, T. R. Benson, W. M. Cardoza, R. Jagannathan, and B. J. Thomas. “Porting OpenVMS from VAX to Alpha AXP”. *Commun. ACM*, Vol. 36, No. 2, pp. 45–53, 1993.

[Lawt 96] K. P. Lawton. “Bochs: A Portable PC Emulator for Unix/X”. *Linux J.*, p. 7, 1996.

[Magn 02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. “Simics: A Full System Simulation Platform”. *Computer*, Vol. 35, No. 2, pp. 50–58, 2002.

[Magn 98] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. “SimICS/sun4m: A Virtual Workstation”. pp. 119–130, 1998.

[May 87] C. May. “Mimic: a fast system/370 simulator”. In: *SIGPLAN ’87: Papers of the Symposium on Interpreters and interpretive techniques*, pp. 1–13, ACM Press, New York, NY, USA, 1987.

[Mill 91] C. Mills, S. C. Ahalt, and J. Fowler. “Compiled Instruction Set Simulation”. *Software, Practice and Experience*, Vol. 21, No. 8, pp. 877–889, 1991.

[Much 97] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Nohl 02] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. “A universal technique for fast and flexible instruction-set architecture simulation”. In: *DAC ’02: Proceedings of the 39th conference on Design automation*, pp. 22–27, ACM Press, New York, NY, USA, 2002.

[Oluk 96] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. “The case for a single-chip multiprocessor”. In: *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 2–11, ACM Press, New York, NY, USA, 1996.

[Patt 85] D. A. Patterson. “Reduced instruction set computers”. *Commun. ACM*, Vol. 28, No. 1, pp. 8–21, 1985.

[Pope 74] G. J. Popek and R. P. Goldberg. “Formal requirements for virtualizable third generation architectures”. *Commun. ACM*, Vol. 17, No. 7, pp. 412–421, 1974.

[Qin 06] W. Qin, J. D'Errico, and X. Zhu. "A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation". In: *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 193–198, ACM, New York, NY, USA, 2006.

[Resh 03] M. Reshadi, P. Mishra, and N. Dutt. "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation". In: *DAC '03: Proceedings of the 40th conference on Design automation*, pp. 758–763, ACM Press, New York, NY, USA, 2003.

[Resh 09] M. Reshadi, P. Mishra, and N. Dutt. "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation". *ACM Trans. Embed. Comput. Syst.*, Vol. 8, No. 3, pp. 1–27, 2009.

[Rose 95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. "Complete Computer System Simulation: The SimOS Approach". *IEEE Parallel Distrib. Technol.*, Vol. 3, No. 4, pp. 34–43, 1995.

[Sher 02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically characterizing large scale program behavior". In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 45–57, ACM Press, New York, NY, USA, 2002.

[Site 93a] R. L. Sites. "Alpha AXP architecture". *Commun. ACM*, Vol. 36, No. 2, pp. 33–44, 1993.

[Site 93b] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. "Binary translation". *Commun. ACM*, Vol. 36, No. 2, pp. 69–81, 1993.

[Site 95] R. L. Sites and R. T. Witek. *Alpha AXP architecture reference manual (2nd ed.)*. Digital Press, Newton, MA, USA, 1995.

[Smit 05a] J. E. Smith and R. Nair. "The Architecture of Virtual Machines". *Computer*, Vol. 38, No. 5, pp. 32–38, 2005.

[Smit 05b] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Stal 01] R. Stallman. "Using and Porting the GNU Compiler Collection". *M.I.T. Artificial Intelligence Laboratory*, 2001.

[Stal 90] W. Stallings. *Reduced Instruction Set Computers*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[Tarj 72] R. Tarjan. "Depth-First Search and Linear Graph Algorithms". *SIAM Journal on Computing*, Vol. 1, No. 2, pp. 146–160, 1972.

[Thom 96] T. Thompson. "An Alpha in PC Clothing". *BYTE*, Vol. 19, No. 2, pp. 195–196, 1996.

[Toph 07] N. Topham and D. Jones. "High Speed CPU Simulation using JIT Binary Translation". In: *MoBS '07: Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, San Diego, CA, USA, 2007.

[Torc 07] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Wall 05] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten. "Vasa: A Simulator Infrastructure with Adjustable Fidelity". In: *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005)*, Phoenix, Arizona, USA, Nov. 2005.

[Wirb 88] L. Wirbel. "DOS-to-Unix Compiler". *Electronic Engineering Times*, Vol. 14, p. 83, 1988.

[Witc 96] E. Witchel and M. Rosenblum. "Embra: fast and flexible machine simulation". In: *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 68–79, ACM Press, New York, NY, USA, 1996.

[Woo 95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 programs: characterization and methodological considerations". In: *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 24–36, ACM, New York, NY, USA, 1995.

[Wund 03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling". In: *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pp. 84–97, ACM Press, New York, NY, USA, 2003.

[Zhu 02] J. Zhu and D. D. Gajski. "An ultra-fast instruction set simulator". *IEEE Trans. Very Large Scale Integr. Syst.*, Vol. 10, No. 3, pp. 363–373, 2002.

[Zhu 99] J. Zhu and D. D. Gajski. "A retargetable, ultra-fast instruction set simulator". In: *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, p. 62, ACM Press, New York, NY, USA, 1999.