HySim: A Hybrid Software/Hardware Simulation Framework for Early Architectural Exploration of Chip Multiprocessors

BY

Ayman Ali Mohammad Hroub

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY
In

COMPUTER SCIENCE AND ENGINEERING

December 2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN- 31261, SAUDI ARABIA
DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Ayman Ali Mohammad Hroub** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **PhD in Computer Science and Engineering.**

Dr. Muhammad Elrabaa
(Advisor)

عادل

Dr. Adel F. Ahmed

Department Chairman

Dr. Aiman El-Maleh
(Member)

Prof. Salam A. Zummo
Dean of Graduate Studies

Dr. Mahmood Niazi
(Member)

15/2/16

Date

Al-Glw

Prof. Mayez Al-Mouhamed
(Member)

M Alshyeb

Dr. Mohammad Alshayeb
(Member)

© Ayman Ali Mohammad Hroub

2015

To the memory of my mother

To the memory of my brother Abdul-Latif

To my wife

To my cute sons, Yamin and Abdul-Latif

TABLE OF CONTENTS

ACKNOWLEDGMENTS xviii

Abstract xix

ملخص الرسالة xxi

CHAPTER 1 INTRODUCTION 1

  1.1 Motivation 1

  1.2 Terminology and Nomenclature 3

  1.3 Dissertation Outline 7

  1.4 Contributions 8

CHAPTER 2 STATE OF THE ART OF MULTICORE ARCHITECTURES 9

  2.1 Intel Xeon Phi Coprocessor 9

    2.1.1 Coprocessor’s Core 10

    2.1.2 Interconnection Network 10

  2.2 Intel Xeon CPU E5-2680 12

  2.3 TILE-Gx8072 Processor: Telira Processor 13

  2.4 Intel’s 48-core SCC (Single-Chip Cloud Computer) Processor 15

  2.5 Multi-node Multicore Architectures for Irregular Applications 15

2.6 POWER7 16

2.7 ARM Architecture 17

2.8 AMD Processors 17

2.9 Axel 18

2.10 Summary and Discussion 18

CHAPTER 3 COMPUTER ARCHITECTURE SIMULATION TECHNIQUES 21

3.1 Simulator Design Trade-offs 21

3.2 Architectural Simulation Techniques 24

3.2.1 Execution-Driven Simulations 24

3.2.2 Trace-Driven Simulations 27

3.2.3 User-Level vs. Full-System Simulations 28

3.2.4 Abstract vs. Detailed Simulations 30

3.2.5 Software vs. Hardware Simulators 32

CHAPTER 4 REVIEW OF EXISTING MULTICORE SIMULATORS 36

4.1 Software Simulators 36

4.1.1 GEM5 36

4.1.2 Graphite 38

4.1.3 Sniper 40

4.1.4 PinPlay 42

4.1.5 McSimA+ 46

4.1.6 SiMany 48

4.1.7 HORNET..49

4.1.8 Manifold 50

4.1.9 Transformer 51

4.1.10 COTSon..52

4.1.11 Summary and Discussion 52

4.2 FPGA-based Simulators 54

4.2.1 RAMP Gold..54

4.2.2 HAsim..55

4.2.3 Arete 57

4.2.4 ScalableCore system 3.3...58

4.2.5 Summary and Discussion 58

4.3 Hybrid Software/Hardware Simulators 61

4.3.1 PROTOFLEX...61

4.3.2 FAST 62

4.3.3 Summary and Discussion 63

CHAPTER 5 OVERVIEW OF THE PROPOSED SIMULATION FRAMEWORK 65

5.1 Basic Strategy 65

5.2 Functional and Timing Models' Implementation Options 67
5.3 FPGA-based Simulation Framework Design Options 68
  5.3.1 Rigid FPGA-based Simulator 68
  5.3.2 Fully-flexible FPGA-based Simulator 69
  5.3.3 Quasi-flexibe FPGA-based Simulator 70
  5.3.4 Summary 74
5.4 HySim's Architecture and Workflow 76
5.5 HySim's Output 79
CHAPTER 6 COMPRESSED EXECUTION TRACE GENERATION 81
6.1 Introduction 81
6.2 The proposed Execution Trace Compression Technique 86
  6.2.1 Basic Strategy 88
  6.2.2 CET Encoding 92
  6.2.3 Loop Recognition 99
  6.2.4 CET Profiler 103
  6.2.5 CET Code Generation 106
  6.2.6 Emulation and CET Data Generation 111
  6.2.7 System Calls Latency 112
6.3 Experimental Results 115

6.3.1 Experimental Setup 115

6.4 Compression Ratio 118

6.5 Compression/Decompression Speed 133

CHAPTER 7 HYSIM TIMING MODEL 135

7.1 Bluespec SystemVerilog (BSV) 135

7.1.1 BSV Coding Productivity 136

7.1.2 BSV to Verilog Compilation 139

7.2 Timing Model Architecture 148

7.3 Timing Model’s Tile Architecture 151

7.3.1 HySim Core Model (CET Core) 153

7.3.2 Timing Model’s Cache Memory 164

7.3.3 Target Cache Hierarchy Model 167

7.3.4 Timing Model’s Router 170

7.4 NoC Model 172

7.5 Multi-threading Management 172

7.5.1 Thread Scheduling on the Same Core 173

7.6 Inter-Thread Interactions 178

7.7 FPGA Implementation Details 179

7.7.1 Host-FPGA Communication 181

7.7.2 CET Cache Filling Circuit 184

7.7.3 FPGA Resources Consumption 187

CHAPTER 8 HYSIM EXPERIMENTAL RESULTS 191

8.1 Experimental Setup 191

8.1.1 Target Machine specifications 191

8.1.2 Real Hardware Execution Time Measurement 197

8.1.3 Benchmarks 198

8.2 Simulation Monitor 209

8.3 Evaluation of Simulation Speed 214

8.4 Evaluation of Simulation Accuracy 225

8.4.1 Absolute Accuracy Relative to Real Hardware 225

8.4.2 Speedup Accuracy 234

8.4.3 Base CPI Effect on Speedup 239

8.5 Limitations 242

CHAPTER 9 CONCLUSION AND FUTURE WORK 243

References 245

APPENDIX A: SOURCE CODE 251

A.1 CET Tool 251

A.2 HySim Timing Model 252

Vitae 254

LIST OF FIGURES

Figure 1: Intel Xeon Phi Interconnection Network [1] 11

Figure 2: TILE-Gx8072 Architecture [15] 14

Figure 3:Simulation diamond illustrates the trade-offs in simulator accuracy, coverage, development [4] 23

Figure 4: High Level Block Diagram of PinPlay Framework Workflow 44

Figure 5: HySim Framework Structure 77

Figure 6: A Sample Simulation Results for One Thread... 80

Figure 7: CET Tool Work Flow. 91

Figure 8: Branch Results' Chain of a Loop Candidate X86 Conditional Branch Instruction.. 101

Figure 9: CET Profiler Flowchart.. 105

Figure 10: CET Code Generator Flowchart 108

Figure 11: Compression results for a simple C-code snippet... 110

Figure 12: System Time Histogram 114

Figure 13: Compression Ratio of Instruction Addresses Only Traces (IA) and the Full Trace... 120

Figure 14: Compression Ratio vs Problem Size for 3 single-threaded benchmarks. 122

Figure 15: Compression Ratio Comparison between the CET Tool and SBC Technique 127

Figure 16: Compression Time for CET and SBC Techniques 129

Figure 17: Decompression Time for CET and SBC Techniques 130

Figure 18: Compression Ratio vs Number of Threads 132

Figure 19: A Simple Adder BSV Code 141

Figure 20: The Auto Generated Verilog Code of the Simple Adder Interface 142

Figure 21: The Auto Generated Verilog Code of the Simple Adder Registers................143

Figure 22: The Auto Generated Verilog Code of the Simple Adder Rule Scheduling and Execution 144

Figure 23 : Simple Adder Manually Written Verilog Code 146

Figure 24: A Top Level Logical View of HySim's Timing Model 150

Figure 25: HySim's Timing Model's Tile Overview 152

Figure 26: CET Core Abstract Schematic 155

Figure 27: Loop Scheduling Unit Flowchart...163

Figure 28: CET Instruction and Data Caches..166

Figure 29: Unified L2 Cache Simplified Model..169

Figure 30: Router Block Diagram 171

Figure 31: Threads Management in Multithreaded Target Cores (Interleaved and Blocked) 175

Figure 32: Threads Management in Multithreaded Target Cores (Simultaneous)...........177

Figure 33: Virtex 6 XC6VLX550T FPGA Board 180

Figure 34: Data flow for the Ethernet Core. 183

Figure 35: High Level View of HySim Timing Model Interaction with the Main Memory 186

Figure 36: The Output of lscpu Linux Command 192

Figure 37: Cache Hierarchy Architectural Specifications of "ThinkStation" Workstation. 193
Figure 38: A Snapshot from a Sample Thread Profile 201
Figure 39: The Level of Parallelism for the Used Multi-threaded Benchmarks (I) 207
Figure 40: The Level of Parallelism for the Used Multi-threaded Benchmarks (II)....208
Figure 41: ChipScope Snapshot 210
Figure 42: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (I) 211
Figure 43: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (II)....212
Figure 44: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (III)....213
Figure 45: HySim Simulation Speed....216
Figure 46: HySim Absolute Accuracy Relative to Real Hardware (Application Level) 226
Figure 47: Simulated Time (HySim + Sys) Relative to the Min and Max Total Hardware Execution Time (I)....228
Figure 48: Simulated Time (HySim + Sys) Relative to the Min and Max Total Hardware Execution Time (II) 229
Figure 49: Simulated Speedup Accuracy Relative to Real HW Speedup (Application Level) (I) 235
Figure 50: Simulated Speedup Accuracy Relative to Real HW Speedup (Application Level) (II) 236

Figure 51: Simulated Speedup Accuracy Relative to Real HW Speedup (System Level) (I) 237

Figure 52: Simulated Speedup Accuracy Relative to Real HW Speedup (System Level) (II) 238

Figure 53: CPI Effect on Speedup (Application Level) (I) 240

Figure 54: CPI Effect on Speedup (Application Level) (II) 241

Figure 55: BSV Code Hierarchy 253

LIST OF TABLES

Table 1: Multicores' State of the Art Summary....................20

Table 2: Summary of the FPGA-based Simulators....................60

Table 3: Summary of the FPGA-based Hybrid Simulators....................64

Table 4: HySim's Parameters and Their Default Values....................72

Table 5: Pros and Cons of Different FPGA-based Simulation Framework Design Options....................75

Table 6: CET Code and Data format Summary....................94

Table 7: Benchmarks and Their Input Sets....................116

Table 8: Uncompressed and Compressed Traces Size and Compression Ratio....................124

Table 9: Compression/Decompression Speed (MIPS)....................134

Table 10: Comparison between the BSV Code Side and the Corresponding Auto-generated Verilog Code Size....................138

Table 11: FPGA Resources and Inferred Components are Identical for Both Manually Written and Auto Generated Verilog Codes....................147

Table 12: Actions Taken by CET Core for Different CET Instructions....................158

Table 13: The Amount of FPGA Resources Consumed by One and 16 CET Tiles....................188

Table 14: The Sizes of Different CET Caches for a Single CET Tile....................190

Table 15: Target Machine Architectural Specifications....................195

Table 16: Splash-2 Benchmarks and Their Input Sets....................199

Table 17: CET Static Code Size for Different Threads..203

Table 18: Number of dynamically Executed Instructions and Load/Store Percentages for Thread 0..205

Table 19: Simulation and Simulated Time and Clock Cycles for a Single Thread..........218

Table 20: Simulation and Simulated Time and Clock Cycles for 16 Threads.................219

Table 21: FPGA Cycles to Target Cycles Ratios for HAsim and HySim........................221

Table 22: HySim's Simulation Speed in MIPS Compared to HASim 223

Table 23: HySim Accuracy Relative to Interval and One-IPC Models 231

Table 24: Comparison between HySim and McSimA+ Accuracy...................................233

ACKNOWLEDGMENTS

I would like to thank the great university; **King Fahd University of Petroleum and Minerals** for supporting this research and for offering me the opportunity to pursue my graduate studies. Special thanks to the **National Science, Technology and Innovation Plan (NSTIP)** for their generous funding of this research under project **#12-INF3015-04**.

I wish to express my deep appreciation to my advisor, **Dr. Muhammed Elrabaa** for his guidance, support, help, cooperation and constructive feedback. Dr. Elrabaa dedicated a lot of his valuable time for me. He always provides me with brilliant ideas and smart solutions. I am also very grateful to my dissertation committee members: **Dr. Aiman El-Maleh**, **Dr. Mahmood Niazi**, **Prof. Mayez Al-Mouhamed**, and **Dr. Mohammad Alshayeb** for their help and cooperation.

I would like to thank the co-investigators in our NSTIP project: **Dr. Ahmad Khayyat** and **Dr. Muhamed Mudawar** for their help and participation in this research. Also I want to thank my colleagues: **Mr. Amran Al-Aghbari** and **Mr. Mohammed Al-Asli** for their participation in this project and helping us in FPGA design issues.

I want to thank **Bluespec** for providing us with a free license of their Bluespec SystemVerilog compiler over the past three years.

I would like to thank my family for their support, patience, and encouragement. I am so grateful to my father, my wife, my sons, my brothers, and my sisters.

ABSTRACT

Full Name: Ayman Ali Mohammad Hroub

Thesis Title: HySim: A Hybrid Software/Hardware Simulation Framework for Early Architectural Exploration of Chip Multiprocessors

Major Field: Computer Science and Engineering

Date of Degree: December, 2015

Simulation is the de facto tool for computer architecture performance evaluation. It implies modeling the events of interest in the intended architecture to be evaluated. Traditionally, software simulators have been used. Although such simulators are inexpensive and flexible, they lack the required speed, especially for cycle-accurate models. In the multicore era, processors became much more complex. They comprise large number of cores, complex memory hierarchies, and complex interconnection networks. Thus, the design space to be explored became much larger. Moreover, this kind of architecture has a voluminous number of concurrent events. Therefore, there is a crucial need for a very fast simulator even if it sacrifices degree of accuracy. In the early stages, the goal is to compare different architectures rather than to have accurate performance numbers. In this dissertation, we propose HySim, a hybrid software/hardware simulation framework for early architectural exploration of chip multiprocessors. It exploits the flexibility of software and the massive parallelism offered by the FPGAs. HySim is a two phase simulation framework. In the first phase, the application is natively executed under Intel pin tool. The output of this phase is the application's execution trace. In the second phase, this trace is fed into an FPGA-based timing model to perform timing simulation. As it is well known, the trace size is very large to store, especially on FPGAs because they have limited storage resources. Therefore, this trace is compressed on-the-fly into an executable format that can be executed by the timing model. Thus, the contribution in this dissertation is twofold: (1) an efficient trace compression technique with a compression ratio of up to 2987.9, (2) a very fast simulation framework. HySim has been validated against real hardware using a subset of SPLASH-2 and PARSEC benchmarks. The simulation results showed that HySim speed is up to 2204.257 MIPS with 14% average absolute error relative to real hardware execution time.

ملخص الرسالة

الاسم الكامل: أيمن علي محمد حروب.
عنوان الرسالة: هايسم: إطار محاكاة هجين من البرمجيات و العتاديات المادية للاستكشاف المبكر لبنية الرقائق متعددة المعالجات.
التخصص: علوم و هندسة الحاسب الآلي.
تارخ الدرجة العلمية: كانون الأول، 2015.

محاكاة بنية الحاسب الآلي هي الأداة الفعلية لتقييم أداء الحاسب. تتضمن المحاكاة بناء نموذج لمركبات الحاسب ذات التأثير على الأداء. لقد دأب الباحثون على استخدام المحاكيات البرمجية و ذلك بسبب مرونتها و سهولة بنائها و انخفاض تكلفتها. ولكن في المقابل تتصف هذه المحاكيات بالبطء الشديد و خصوصاً في حالة النماذج الدقيقة جداً. لقد ازدادت المعالجات تعقيداً و خاصة في حقبة المعالجات متعددة الأنوية، فلقد أصبح المعالج يتكون من عدد كبير من الأنوية التي تربطها شبكة معقدة و ازداد هرم الذاكرة تعقيداً. لذلك فإنه يتعين على الباحث أن يستكشف عدداً هائلاً من خيارات التصميم لاختيار التصميم الأفضل أداءً. علاوةً على ذلك، فإن المعالجات متعددة الأنوية تحتوي على عدد كبير من الأحداث المتوازية. لذلك أصبح ايجاد وسيلة محاكاة تتسم بسرعة كبيرة حاجة ملحة حتى لو كان ذلك على حساب شيء من دقة المحاكي من أجل تسريع المحاكاة. في مراحل التصميم الأولى يكون اهتمام المصمم بمقارنة خيارات التصميم المختلفة مع بعضها و استبعاد الخيارات غير المجدية أكثر من اهتمامه في الحصول على نتائج أداء متناهية الدقة. في هذه الأطروحة نقترح هايسم الذي هو عبارة عن إطار محاكاة هجين يتكون من البرمجيات و العتاديات المادية للاستكشاف المبكر لأداء الرقائق متعددة المعالجات. هايسم يستغل التوازي الناعم و الخشن الذي توفره أجهزة الـ (FPGAs) . المحاكاة في هايسم تتم على مرحلتين: المرحلة الأولى تتضمن تنفيذ التطبيق على المعالج الأم تحت أداة (pin). نتيجة هذه المرحلة هي تتبع لتنفيذ التطبيق. في المرحلة الثانية يتم تسليم تتبع التطبيق لنموذج التوقيت المبني على الـ (FPGA).

نظراً لأن حجم تتبع التنفيذ كبير جداً و يتعذر تخزينه خاصةً على الـ (FPGA) التي تمتلك مساحات تخزين محدودة قمنا بتطوير تقنية لضغط هذا التتبع و تقليص حجمه بحيث يتحول إلى شكل قابل للتنفيذ يمكن فهمه من قبل نموذج التوقيت. لذا فإن الابتكار في هذه الأطروحة ذو شقين: الشق الأول تقنية ضغط فعالة لضغط تتبعات التنفيذ حيث أصبح حجم التتبع المضغوط أصغر من حجمه الأصلي بـ 2987.9 مرة في أحسن الأحوال. الشق الثاني يتضمن إطار محاكاة سريع جداً. لقد تم التحقق من دقة هايسم من خلال مقارنته بعتاديات مادية حقيقية باستخدام مجموعة من تطبيقات (SPLASH-2 and PARSEC) . أظهرت النتائج أن سرعة هايسم قد تصل إلى 2204.257 مليون تعليمة في الثانية و أن معدل نسبة الخطأ المطلق حوالي 14% مقارنةً مع العتاديات المادية الحقيقية.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Chip multi-processors (CMPs) have lately gained considerable popularity and importance [1-3] . They have been identified as the only way to deliver high-performance computing as the chip manufacturing technology scales down to the NANO scale. A CMP consists of a large number of interconnected cores and a complex memory system on a single chip. When developing such system, the architects need to explore a large design space in the early stages to identify the type and number of cores, memory specifications (number of cache levels, size, associativity, replacement policies, etc.), cache coherence protocols, interconnection networks, etc.

Moreover, software application developers need to explore different machine's configurations for their different algorithms' implementations. Also they want to write and test their software before the real machine becomes available. Thus, having a flexible model of the machine, before it is fabricated, is of a great value to the applications' developers.

CMPs' large design space cannot be explored analytically due to the lack of accuracy of this approach. Analytical evaluation might be useful for high level decisions, e.g., to determine the area of interest in a huge design space [4]. Furthermore, hardware prototyping of the target machine will take too much time and effort. Moreover, in the early stages, the full machine’s specifications are not completely clear, which makes hardware prototyping not an optimum option. Thus, there should be a way to capture the key performance characteristics of the target machine, and provide the architect with a quick feedback. This can be done via simulations.

A simulator models the events of interest in the architecture being investigated (target architecture). Traditionally, single-threaded software simulators were used [5]. Although these simulators are flexible, easy to develop, and can be cycle-accurate, they lack the required speed, especially when they are used for CMPs. Actually the simulation slowdown grows at least linearly with the number of simulated cores when single-threaded software simulators are used to simulate CMPs [6]. Because CMPs have too many parallel events and hence processing these events sequentially means that a target cycle requires too many host machine’s cycles (the machine that hosts the simulator) to be simulated.

Researchers started exploiting the parallel structure of CMPs to develop parallel software simulators that run on multicore host machines [7]. Although parallelizing software simulators improved simulation speed, the communication among the different cores of the host machine still limits achieving more simulation speedup.

In the last few years, researchers exploited the fine/coarse grained parallelism in FPGAs (Field Programmable Gate Arrays) to accelerate computer architecture’s simulation [8-11]. This is possible for two reasons: (1) the structural nature of CMPs exhibits massive fine/coarse grained parallelism which makes them ideal candidates for

FPGA acceleration, (2) recent FPGAs [12, 13] have large number of logic cells and large size of on-chip memory which makes them large enough to host CMPs' simulators.

In this dissertation, I propose HySim, a user-level hybrid software/hardware simulation framework for CMPs. It combines the flexibility of software with the speed and accuracy of hardware (FPGAs). HySim's implements a two-phase simulation technique. In the first phase, the application is natively executed and instrumented under Intel pin tool [14]. The output of this phase is the execution trace of the benchmark in a compressed executable format that is architecture agnostic. In the second phase, the compressed executable trace is fed to the FPGA-based timing model for timing simulation of the target architecture.

1.2 Terminology and Nomenclature

- Application and workload are interchangeable in this dissertation.
- Application-level and user-level are interchangeable.
- **BSV:** Bluespec SystemVerilog. It is a high level fully synthesizable hardware description language.
- **CMP:** Chip Multiprocessor.
- **CPI:** Clocks per Instruction, it refers to the average number of clock cycles a processor needs to complete one instruction.
- **CPU:** Central Processing Unit of the computer. It is interchangeable with the term core.

- **Cycle-accurate:** all memory, register, pipeline contents are accounted for and updated per each target's clock cycle.
- **DMA:** Direct Memory Access.
- **Fidelity:** is not just simple accuracy, but refers to how much of the detailed simulation events are recorded.
- **FPGA:** Field Programmable Gate Arrays, which is a configurable device that hosts custom computing machines.
- **Host Machine:** refers to the machine that hosts the simulator, it can be an FPGA or a computer. Everything related to such machine can be prefixed by the term "host", e.g., host clock cycle, which refers to the clock cycle period of this machine. This term can be interchangeable with other terms, such as host processor, host core, host computer, etc.
- **Host Operating System,** is the operating system running on the host machine.
- **Host Thread:** is a simulation thread, a thread of a multithreaded simulator that is responsible for simulating a target core or any target architecture component.
- **Host Thread Synchronization:** refers to synchronizing the target clocks of the target cores being simulated on distributed host cores.
- **HySim:** **Hy**brid **Sim**ulator
- **"In-Core":** refers to the architectural features inside the processing core, e.g., functional units, issue logic, branch prediction, etc.
- **Intel Pin [14]:** is a user-level dynamic binary instrumentation framework for the IA-32 and X86-64 instruction-set architectures. The tool that is implemented

under Pin is called a Pintool. Pintools can be used for dynamic programs analysis in Linux, Android, and Windows environments. Pin allows a tool to insert C/C++ code in arbitrary places in the executable. The code is added dynamically while the executable is running. It can intercept the program's instructions one by one and it has an access to the program symbols.

- **Interval Simulation:** it is a simulation technique based on a mechanistic analytical model where the execution time is split into intervals by miss events, such as: branch miss-predictions, load misses, etc. The functional model executes the application's instructions and identifies theses miss events. The executed instructions and miss events are fed to the interval timing model. Then, the timing model derives the timing of these instructions and misses events based on an analytical model.
- **IO:** Input/Output.
- **IPC:** Instruction per Cycle, it refers to the average number of instructions a processor can complete per one clock cycle. It is the reciprocal of ***CPI*** and it represents the processor's throughput.
- **ISA:** Instruction Set Architecture.
- **KB, MB, GB:** Kilobyte, megabyte, gigabyte, respectively.
- **KIPS:** Kilo Instructions per Second. It is used to measure a processor performance in terms of its throughput, i.e., the average number of instructions that can be executed in a unit of time. It can be used also to measure a simulator performance; it shows the average number of instructions that can be simulated in a unit of time.

- **L1, L2, L3 cache:** Level1, level2, level3 cache, respectively.
- **LLC:** Last Level Cache. The cache memory closest to the main memory.
- **MIPS:** Million Instructions per Second. Same as **KIPS**.
- **Native Execution:** refers to executing the application on a real existing machine which has the same ISA of the target machine.
- **NoC:** Network on Chip.
- **OS:** Operating System.
- **QPI:** Quick Path Interface,
- **Simulated Time:** is interchangeable with target execution time. It is the expected execution time of the application on the machine being simulated.
- **Simulation Speed:** refers to how fast the simulator can evaluate the target machine performance for a given application.
- **Simulation Time:** refers to the amount of time the simulator takes to evlaute the target machine for a given application.
- **SMP:** Symmetric Multiprocessing.
- **SMPD:** Single Program Multiple Data.
- System-level and kernel-level are interchangeable.
- **Target Architecture:** refers to the architecture being investigated. The term target is interchangeable with other terms, such as intended, simulated, and investigated. Also the term architecture can be interchangeable with other terms, such as machine and processor. Everything related to this architecture can be prefixed by

the term “target”, e.g., target clock cycle, target clock, target frequency, target core, target thread, target cache hierarchy, target interconnection network, etc.

- **“Un-Core”:** refers to the architectural features outside the processing core, e.g., memory hierarchy and interconnection network.
- **User:** the term user in this dissertation refers to the person who eventually uses the simulator. This person is usually a computer architecture researcher, software developer, or a student.
- **Verilog:** a hardware description language.
- **VHDL:** VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuit.

1.3 Dissertation Outline

The rest of this dissertation is structured as follows:

Chapter two presents the state of the art of multicore architectures. This quick survey is important to identify the features of the recent CMPs in order to support them in our simulator. Chapter three introduces the reader to the computer architecture simulator design trade-offs and different simulation techniques. Chapter four reviews the existing multicore architectures’ simulators. Chapter five presents an overview of the proposed simulation framework. It discusses the different options that we evaluated until we reached the current version of HySim. Chapter six covers the proposed trace compression technique. It surveys the existing trace compression techniques and presents the experimental results of our proposed technique. HySim's FPGA-based timing model is detailed in chapter seven. Also chapter seven covers the FPGA implementation details of HySim. Chapter eight discusses the experimental results of HySim. It shows the absolute accuracy of HySim relative to real hardware and other simulators. Moreover, it shows the speed of HySim and compares it with the speed of other simulators. Finally, we concluded in chapter nine.

1.4 Contributions

This dissertation has two main contributions:

1. An efficient trace compression technique for multithreaded applications which achieved a compression ratio of up to **2987.9** with compression speed of up to **789.1 MIPS**.
2. HySim, which is a very fast trace-driven FPGA-accelerated simulation framework for CMPs. HySim achieved a simulation speed of up to **2204.257 MIPS** with **14%** average absolute error relative to real hardware execution time.

CHAPTER 2

STATE OF THE ART OF MULTICORE ARCHITECTURES

In order to determine the key features that should be supported by a new multicore simulation framework, a survey of the most recent multi-core architectures was conducted. This survey included both commercial and academic multi-core architectures. A brief description of the surveyed architectures is provided below.

2.1 Intel Xeon Phi Coprocessor

Intel Xeon Phi coprocessor [1] is based on the Intel MIC (Many Integrated Cores) architecture. Intel Xeon Phi *coprocessor* (consists of over 60 cores) is connected to the Intel Xeon Phi processor (it is also called the host processor) through a PCIe (PCI Express) bus. This configuration supports heterogeneous applications such that some parts of the application run on the host processor and other parts run on the coprocessor. The coprocessor's cores can communicate with each other through PCIe, peer to peer interconnect, or through a network card without any intervention from the host processor.

2.1.1 Coprocessor’s Core

Each core contains a 32 KB L1 private instruction cache, a 32 KB L1 private data cache, and a 512 KB private unified L2 cache that is kept coherent by a global-distributed tag directory. Each core has in-order short pipeline that is capable to support four threads in hardware. Moreover, each core in the coprocessor has a VPU (Vector Processing Unit) that features 512-bit SIMD (Single Instruction Multiple Data) instruction set. VPU supports FMA (Fused Multiply-Add) instructions, SP (single precision) floating point operations, DP (Double Precision) floating point operations, integer operations, gather and scatter instructions, and it supports EMU (Extended Math Unit) that executes operations, such as square root, reciprocal, log, etc. in a vector fashion.

2.1.2 Interconnection Network

Figure 1 shows that Xeon Phi coprocessor has a bidirectional ring interconnect. Each direction consists of three independent rings, namely, (1) a 64-byte data ring, (2) an address ring, which is much smaller than the data ring and it is used to transfer the read/write commands and memory addresses, and (3) an acknowledgement ring, which is the smallest one and it sends the flow control and coherence messages.

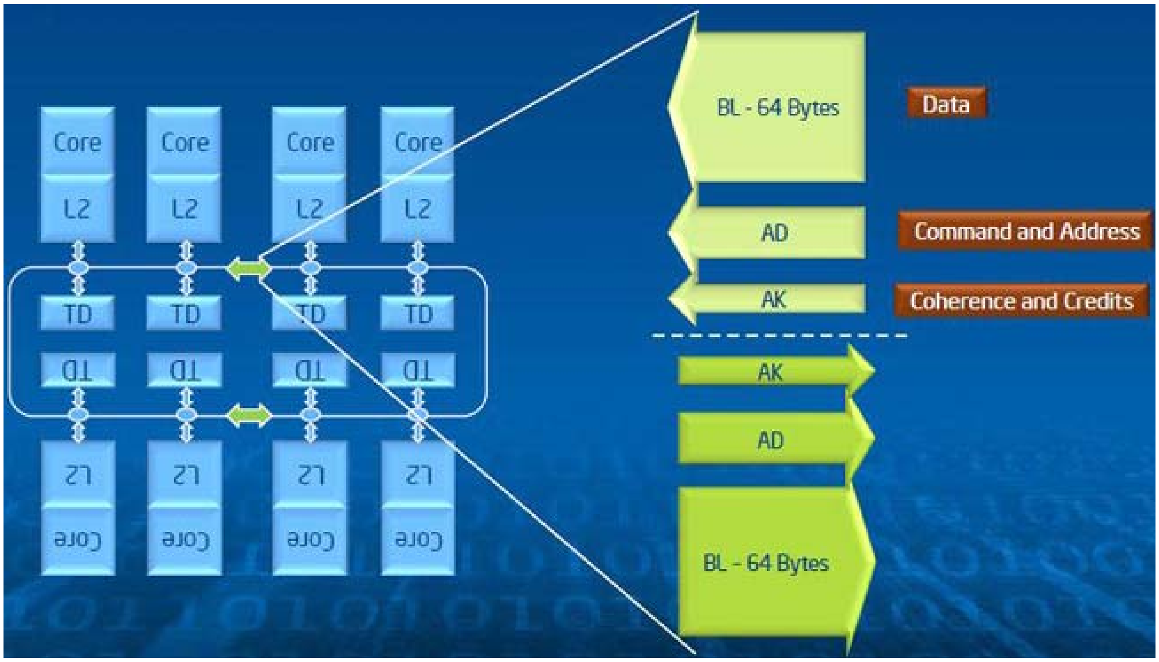


Figure 1: Intel Xeon Phi Interconnection Network [1]

Upon an L2 cache miss, the address is sent to the directories over the address ring. If the block is found in another core's L2 cache, the address is forwarded to that core's L2 cache. If the block is not found in any core's L2 cache, the core generates another address request and queries the data from the main memory.

The number of requests and acknowledgements is larger than the number of data blocks transferred over the network. Simulation results showed that the address and acknowledgement rings would become a performance bottleneck beyond 32 cores [1]. Because address and acknowledgement rings are much less expensive than the data ring, these two rings have been doubled to satisfy the bandwidth requirements of requests and acknowledgments.

2.2 Intel Xeon CPU E5-2680

Each socket of Intel Xeon CPU E5-2680 processor [2] contains eight cores interconnected via an un-buffered ring. Each core is 2-way multi-threaded. Different sockets are interconnected via QPI (Quick Path Interface).

Regarding cache hierarchy, each core has a private 32KB L1data cache, a private 32 KB L1 instruction cache, and a 256 KB private unified (instructions and data) L2 cache. Each socked has an L3 unified cache of 20 MB that is shared among the eight cores.

2.3 TILE-Gx8072 Processor: Telira Processor

TILE-Gx8072 [15] is a 72-core processor optimized for intelligent networking, multimedia, and cloud applications. Figure 2 depicts TILE-Gx8072's architecture. The 72 tiles are connected via a mesh NoC. Each tile comprises a processor core with three pipelines, a 32 KB L1 private data cache, a 32 KB L1 private instruction cache, a 256 KB L2 private and unified cache, and a non-blocking Terabit/sec switch to connect the tile into the mesh. Telira processor has an 18 MB L3 cache that is shared and dynamically distributed. This L3 cache is kept coherent via a directory-based cache coherence protocol.

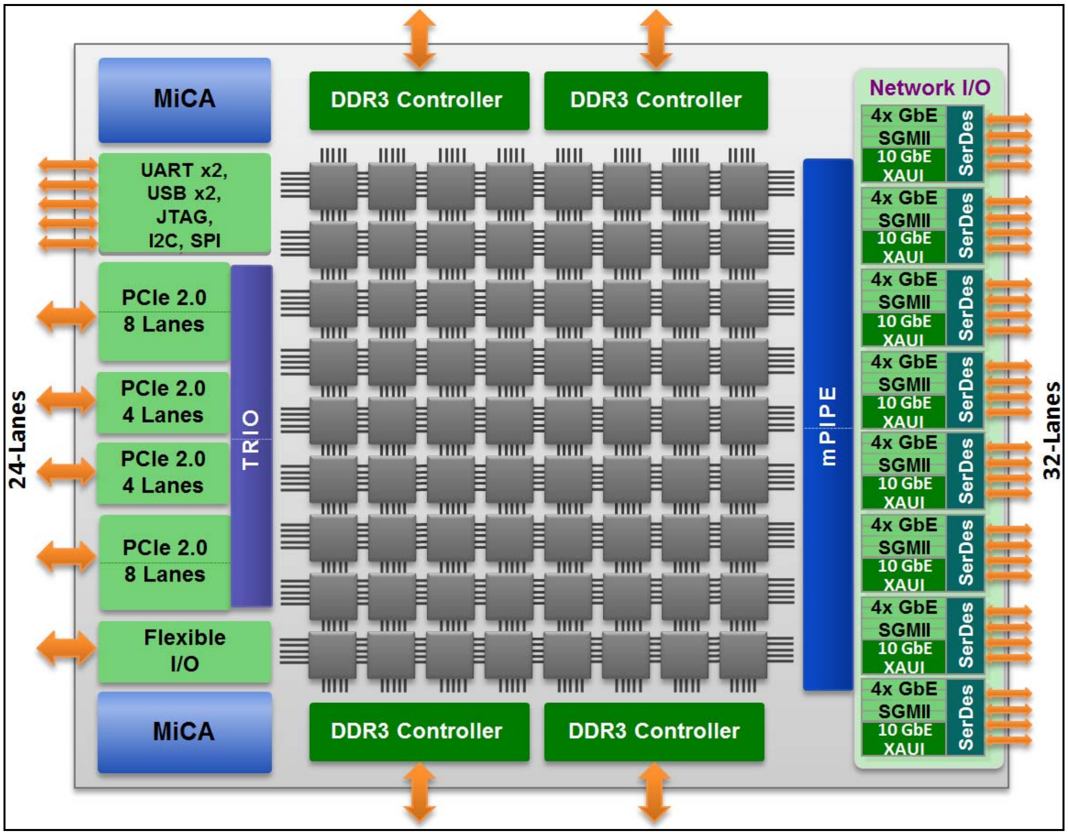


Figure 2: TILE-Gx8072 Architecture [15]

2.4 Intel’s 48-core SCC (Single-Chip Cloud Computer) Processor

SCC is a many-core processor produced by Intel [16]. It supports both message passing and shared memory communication. Cache coherence is the responsibility of the programmer. The memory architecture is composed of multiple distinct address spaces. Each core has a private region and a shared region of the address space.

The 48 cores are organized in 24 dual-core tiles connected by a mesh interconnection network. Each core has a 16 KB L1 instruction cache, a 16 KB L1 data cache, and a 256 KB L2 unified cache. The cores are second generation Pentium processors. They are simple and in-order execution cores. The two cores on a single tile are connected via FSB (Front Side Bus).

2.5 Multi-node Multicore Architectures for Irregular Applications

Secchi et al. [17] introduced a multi-node multicore multi-threaded architecture for irregular applications based on commodity processors. This architecture targets irregular applications, such as data mining, knowledge discovery, and social networks analysis. They were motivated by the fact that irregular applications do not scale well on the cache-based processors, because these applications have poor spatial and temporal locality due to the dynamic data structures, such as unbalanced trees and graphs. This architecture has transparent hardware support for PGAS (Partitioned Global Address Space) and hardware support for inter-thread synchronization.

This architecture has multiple nodes interconnected via an on-chip bus. Each node comprises the following components:

1. Processor core: it has an in order pipeline, I-Cache and scratchpad memory. All cores share a memory controller for the DDR3 RAM.
2. GMAS (Global Memory Access Scheduler): it provides the global address space across multiple nodes of the system. It intercepts load/store operations (local and remote) that the core issues. The requested memory address is decoded, if it is global, then it is forwarded to the remote node through the network interface.
3. GNI (Global Network Interface): this module is responsible for interfacing the node with the inter-node network.
4. GSYNC (Global Synchronization): this module is responsible for managing the lock and un-lock operations on the memory addresses of the node.

2.6 POWER7

POWER7 processor [18] consists of eight cores. Each core is a 4-way SMT (Simultaneous Multithreaded). The cache hierarchy consists of three levels, (1) 32 KB L1 instruction cache and 32 KB L1 data cache, (2) 256 KB L2 unified cache, and (3) 4 MB local region of a 32-MB shared L3 cache. The on-chip components are interconnected via a bus and the cache coherence is maintained through a snoop-based cache coherence protocol.

2.7 ARM Architecture

ARM is a main player in providing high performance and low power configurable IPs (Intellectual Properties) for SoCs (Systems on Chips) that are implemented in embedded systems. ARM Cortex-A15 MPCore [19] implements ARMv7-A architecture with some extensions, such as having advanced SIMD architecture for floating point and integer vector operations.

Cortex-A15 MPCore processor can be configured for up to four cores. Each core has a fixed 32 KB L1 instruction and data caches. The L2 cache is shared and it has configurable size of 512KB, 1MB, 2MB, or 4MB. The on-chip communication is achieved via a bus. To maintain coherency among L1 data caches and the L2 cache, a snoop-based hybrid MESI (Modified Exclusive Shared Invalid) and MOESI (Modified Owned Exclusive Shared Invalid) protocols are used.

2.8 AMD Processors

AMD produces a verity of servers that can be used as HPC (High Performance Computers) platforms, web servers, cloud servers, etc. AMD integrates from 4 to 16 processor cores on-chip with a cache hierarchy depth from two to three levels [9]. The on-chip components interact with a direct interconnects architecture.

The AMD Phenom II X6 processor is the most advanced AMD desktop processor [20]. It can be a quad-core and triple-core. These cores communicate on die rather than on package for better performance. Each core has a private L2 cache of 512 KB. Moreover, a shared cache of 6 MB or 4 MB is shared among all cores.

2.9 Axel

Axel [21] is a heterogeneous cluster produced at Imperial College in London. It is a NNUS (Non-uniform Node Uniform System) system, i.e., each node contains different PEs (Processing Elements), but all nodes are the same in the system.

Each node comprises a quad-core AMD Phenon processor, a 240-core Nvidia Tesla, and an FPGA Vertex 5 LX 330T. The GPU and FPGA accelerators are connected to the CPU through PCIe, whereas the inter-node communication is achieved through Gigabit Ethernet through the NIC (Network Interface Card) on each node. AMD quad-core [22] integrates four cores on-chip that are communicating directly. It has three levels of caches, where L3 is shred among the four cores.

2.10 Summary and Discussion

This short survey revealed that the number of cores in the recent multicore machines can be in tens. This number is expected to increase according to Moore’s law. Also it showed that these cores are interconnected in different NoC topologies (mesh, ring, bus). Moreover, this survey showed that most of the recent multicore machines have up to three cache levels. Besides that, as the number of cores increases, the last level cache (LLC) size increases. Table 1 summarizes the features of these machines.

Based on these findings, a new multicore simulator has to cover all of these features. It has to model as many cores as possible. Also it should model a three-level cache hierarchy in which the LLC size can reach tens of megabytes. Moreover, a new multicore simulator has to support different NoC topologies and the most popular cache coherence protocols.

Table 1: Multicores' State of the Art Summary

Processor	No. Cores	No. threads per core	No. Cache Levels	NoC Topology
Intel’s Xeon Phi	61	4	2	Ring
Intel Xeon CPU E5-2680	8 cores per socket	2	3	Ring per socket, QPI across sockets
TILE-Gx8072	72	-	3	Mesh
Intel’s SCC	48	1	2	Mesh
Secchi Architecture (Irregular Applications)	4-32	1-4	-	On-chip Bus
IBM POWER7	8	4	3	Bus
ARM Cortex-A15 MPCore	1-4	1	2	Bus
AMD Processors	4-16	-	2-3	Bus
Axel	16 x (Quad-core CPU, FPGA, GPU)	-	3 levels in the CPU	PCIe per node, Gigabit Ethernet across nodes

CHAPTER 3

COMPUTER ARCHITECTURE SIMULATION TECHNIQUES

This chapter discusses computer architecture simulation trade-offs. It presents and evaluates different simulation aspects that affect simulation speed, accuracy, and fidelity. The different choices that were made in developing our simulation framework were pointed out with a brief justification in appropriate places.

3.1 Simulator Design Trade-offs

An ideal simulator is a one that is very fast, cycle accurate, and easy to configure in order to cover all configurations of the intended architecture. Unfortunately, this ideal simulator simply does not exist, because its features are contradictory. For example, a cycle accurate simulator implies modeling every component of the target machine precisely, yet this precise modeling requires too much time to develop. Also it will be very slow since for each target clock cycle, voluminous amount of things need to be checked and updated.

Unfortunately, simplifying the simulator to reduce development and simulation times also implies sacrificing simulation fidelity. Figure 3 shows the simulation diamond which illustrates these trade-offs [4].

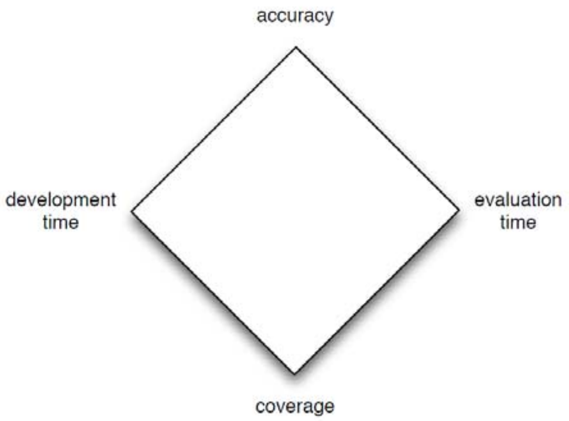


Figure 3:Simulation diamond illustrates the trade-offs in simulator accuracy, coverage, development [4]

Our key approach is to develop new techniques that circumvent the above tradeoffs and allow us to retain good accuracy while speeding up the simulations significantly.

3.2 Architectural Simulation Techniques

A computer architecture simulator is a bipartite consisting of a functional model and a timing model of the target architecture. The **functional model** is responsible for the correct execution of the application, i.e., it models the target ISA (Instruction Set Architecture). On the other hand, the **timing model** captures the timing characteristics of the target machine and it is responsible for performance evaluation of that machine.

This section presents the key different simulation techniques. Some of these techniques are presented in pairs because they are counterparts.

3.2.1 Execution-Driven Simulations

In **execution-driven simulators**, the functional and timing models are combined together. This combination achieves more accuracy because it guarantees that the time-dependent events, such as thread interleaving in multi-threaded applications, are modeled accurately. This combination ranges from integrating the functional and timing models together in one entity to decoupling them into two separated interacting entities. In most cases, configuring the target architecture implies changing the timing model only. Thus, in decoupled simulators, the timing model can be replaced by another one easily.

However, in the integrated ones, modifications are not straightforward and they are error prone.

Mauer et al. [23] classified execution-driven simulators into four categories based on the degree of coupling between the functional and timing models:

1. **Integrated,** where the functional and timing models are tightly integrated together as one entity. Although this kind of simulator can be very accurate, it is complex to develop and maintain. It lacks modifiability, extensibility, and flexibility. E.g., GEM5 simulator [24].
2. **Timing-directed,** where the timing model directs the functional model. In other words, the timing model asks the functional model to perform a specific task, e.g., executing an instruction, loading a datum into the cache, selecting a certain thread-interleaving, etc. in the correct time. Thus, the functional model keeps the architectural states such as registers and memory values and it waits for requests from the timing model. An example of such simulator is Asim [25].
3. **Functional-first,** where the functional model runs ahead of the timing model and feeds it with an instruction trace. This trace is fed on-the-fly, i.e., it does not need to be stored. It can be fed through a UNIX pipe. This kind of simulator is faster than timing-directed simulators because it allows the functional and timing models to run simultaneously. However, in the timing-directed, the timing model runs and when it needs any service from the functional model, it calls it.

For time-dependent events ordering, the functional model is able to roll back. For example, the functional model executes the correct instruction path and it is not aware if there is a branch miss-prediction, thus, when the timing model detects a branch miss-prediction, it orders the functional model to roll back to the state prior to the branch. An example of such simulator is COTSon [26].

4. **Timing-first,** it was defined by Mauer et al. in 2002 [23] as a new approach for decoupling functional and timing models. TFsim full-system simulator [23] was the first implementation of the timing-first simulation. In this approach, the functionality of those instructions that are required for performance evaluation is augmented into the timing model in conjunction with the main correct decoupled functional model. Some advantages of this approach include reducing the simulator development time and allowing for more detailed modeling of the microarchitecture because part of the functional features is integrated into the timing model. However, the functional part integrated into the timing model does not perfectly model speculative instructions along miss-predicted paths and inter-thread events. Therefore, the correct functional model is responsible for repairing the timing model when it takes the wrong path. In timing-first simulation, the timing model runs ahead of the functional model, i.e., the timing model executes each dynamic instruction ahead of the functional model. When the timing model commits an instruction, it invokes the correct functional model (the decoupled functional model) to verify if the timing model deviates from the correct execution path or not. If there is any deviation from the correct path, the functional model corrects the timing model by loading the correct architectural state into the timing model before it can proceed.

Simply, in timing-first simulators, the timing model can be considered as an integrated execution-driven simulator. However, its functional part is not perfectly reliable. Thus, the correct functional model acts as a reference for this simulator to repair it whenever it deviates from the correct execution path.

3.2.2 Trace-Driven Simulations

Trace-driven simulators [27-29] completely separate the functional model from the timing model. Trace-driven simulation is performed in two phases. In the first phase, the application is functionally executed either natively or using a functional simulator (simple ISA simulator). The result of this first phase is an execution trace that comprises the executed instructions along with their corresponding memory references. In the second phase, the trace is fed to the timing model of the target architecture to perform timing simulation. This separation allows running the functional model only once and using it many times for different target architecture configurations, thus increasing the simulation speed and efficiency.

A trace-driven simulator can be a complete simulator for the whole computer system or specific for a certain component, such as a branch predictor or instruction cache. Trace fidelity refers to how many of the original execution events can be re-constructed from the trace.

Although trace-driven simulators are easy to use and develop, they cannot capture timing-dependent thread execution interleaving when they are used to simulate CMPs. Since the trace is fixed, the threads’ ordering included in the trace is fixed too, but the target architecture may have a different threads ordering. However, researchers and architects continued to use trace-driven simulation for CMPs [28, 30, 31], because there are ways around this drawback such as trying to manage parallelism dynamically during timing simulation, e.g., [30].

Another major challenge of trace-driven simulation is the large size of trace files. Although disk storage is currently inexpensive, the disk access time is still high. Moreover, the situation is not improved when FPGAs are used for trace-driven simulation due to their limited storage resources. This drawback has been greatly alleviated via trace compression techniques [32-35].

Our proposed simulation framework (dubbed HySim) uses trace-driven simulation methodology. However, HySim’s trace is greatly compressed in an executable code format that can be directly interpreted by the timing model. All multi-threading related events, such as, starting, pausing, waking, synchronizing, and terminating threads are encoded into HySim’s compressed trace. Thus, though HySim is a trace-driven technique since it separates the functional model from the timing model, it incorporates some execution-driven features such as maintaining the correct ordering of multi-threading events.

3.2.3 User-Level vs. Full-System Simulations

Simulators are classified based on whether they model an operating system (OS) or not into **full-system simulators**, e.g., GEM5 [24], SimOS [36], and QEMU Embra [37], or **user-level simulators**, e.g., Graphite [38] and Sniper [39].

A **user-level simulator** simulates only the user-level code of the workload, whereas a **full-system simulator** simulates both the user-level and system-level codes, i.e., a full computer system. Thus, this type of simulator should be able to boot an unmodified commercial operating system. It looks to the user as a system emulator or a virtual machine.

User-level simulators might be sufficient for workloads consisting of limited system-level code; however, a full-system simulator is far more accurate for workloads with significant system-level code, such as, database servers, web servers, email servers, etc. [4]. Moreover, missing the OS model from CMPs simulators may lead to inaccurate performance numbers because multi-threaded applications are affected by the OS scheduling and decisions [4]. However, developing a full-system simulator is far from trivial because it has to cover a complete system.

The OS model has to be incorporated into the functional model to simulate unmodified workloads and into the timing model to estimate the time spent in system calls. Thus, to have an accurate full-system simulator, the simulator has to be execution-driven in order to execute the system calls and evaluate their latencies directly. However, in trace-driven simulators, the OS model can be incorporated into the functional model and hence unmodified workloads can be functionally simulated. Regarding system calls, they can be incorporated into the trace. Then the timing model either ignores them or approximates their latencies based on a certain model, e.g., the user specifies the latencies of the system calls. This is the strategy we adopted for HySim. The OS is implicitly incorporated into HySim's functional model through instrumented native execution (e.g. using Intel's pin instrumentation tool [14] or Valgrind [40]) with system calls encoded into the generated trace. Though the timing model does not model a full OS, it accurately simulates all threading-related function/library calls, such as, start, pause, wake a thread, etc. Other non-critical OS calls are simply assigned constant latencies. All system calls' codes are preserved and appear in HySim's compressed trace. The current version of HySim allows the user to specify the different system calls' latencies.

3.2.4 Abstract vs. Detailed Simulations

The level of details in microarchitecture modeling is used to trade simulation accuracy for speed. Simulators are classified based on this into abstract and detailed simulators. As the term implies, a**bstract simulators** have an abstract model of the core's microarchitecture, when speed is valued over accuracy, e.g., Graphite [38] and Sniper [39]. In this approach, the focus of the simulator can remain on the "un-core" features, namely, the memory hierarchy and interconnection network.

Abstract simulators are good for early architectural exploration [41] because they provide the architect with a quick feedback on the performance trend of the target architecture. There are many ways to abstract processors' cores, such as, **(1) One-IPC model**, which implies that the processor can complete only one instruction per clock cycle, e.g., Graphite [38] and RAMP Gold [42], **(2) Interval model [43]**, which is a mechanistic analytical model where the execution time is split into intervals by miss events, such as: branch miss-predictions, load misses, etc. The functional model executes the application's instructions and identifies theses miss events. The executed instructions and miss events are fed to the interval timing model. Then, the timing model derives the timing of these instructions and miss events based on an analytical model. This model was implemented in Sniper simulator [39], **(3) k-CPI model,** which assumes that k clock cycles are required to execute one instruction, e.g., Manifold [44].

At first glance, it seems that abstract models do not affect the accuracy of evaluating the “un-core” features. In contrast, having unrealistic core’s model can affect the accuracy of the “un-core” features because it can generate the “un-core” related events in the wrong time and in the wrong rate. Examples of these events include cache misses and coherence transactions.

In contrast, **detailed simulators** have cycle-accurate models of the microarchitecture, e.g., Zestro simulator [45]. Although these simulators offer the maximum fidelity, they have longer development and simulation time because they cover the micro details of the target architecture. However, they are vital when there is a micro-architectural innovation. On the other hand, if the innovation is on the “un-core” level only and the target machine will be built from off-the-shelf cores, then abstract simulators can be sufficient.

Since HySim is intended for early architectural exploration of CMPs, it abstracts the core microarchitecture. The current version of HySim implements the ***basic-CPI*** model, which refers to how many clock cycles the processor needs to complete one instruction assuming an ideal cache hierarchy and NoC, i.e., no cache misses and no NoC latency. Thus, the basic-CPI abstracts the “in-core” time, such as, computation time, hazards’ penalties, branch miss-predictions’ penalties, etc. Regarding the “un-core” time, it is added later via the timing model. The user can specify the value of the basic-CPI based on some theory, previous experience, simulation results, published numbers, etc.

3.2.5 Software vs. Hardware Simulators

Simulators are classified based on their implementation technology into software, hardware (FPGA-based) and hybrid types.

Software simulators are implemented purely as software. This category includes sequential software simulators, e.g., GEM5 [24], and parallel software simulators, e.g., Graphite [38]. **Hardware simulators** are implemented on configurable hardware i.e., FPGA, e.g., Arete [9] and RAMP Gold [46]. In H**ybrid simulators**, some components are implemented in software and others in hardware, e.g., PROTOFLEX [10] and FAST [47].

A sequential software simulator includes a single simulation thread that simulates voluminous amount of parallel events of the target architecture sequentially. Thus, each target clock cycle is simulated in too many host clock cycles. This number of host clock cycles is proportional to the level of details included in the timing model and the size of the target architecture. Simply, the simulation thread comprises a loop that iterates over the target architecture model until the workload is completed. In each loop iteration, the simulation thread traverses the target architecture's model component by component sequentially (the component can be a model of a physical component such as a cache memory or a processor core, or it can be an algorithm such as cache replacement policy). For each component, the simulation thread checks the type of event generated by this component, calculates the penalty of this event, if any, and updates the model's state accordingly.

Based on the above, sequential simulators are not practical to simulate CMPs, because CMPs have larger number of components and therefore larger number of parallel events. Thus, adding more cores to the target architecture results in at least linear simulation slowdown when a single-threaded software simulator is used [6]. One attempt to improve the simulation speed of CMPs is to parallelize the sequential software simulation. This parallelization implies that the software simulator comprises multiple concurrent simulation threads. The components of the target architecture and hence the parallel events are partitioned and each partition is assigned to a simulation thread. The partition granularity is usually at the tile level, e.g., in Graphite [38], a simulation thread is created to simulate one target tile and the OS scheduler is responsible for scheduling these simulation threads. The tile typically comprises a processor core, a part of the memory subsystem, and a network interface.

This parallelization achieved some speedup, e.g., in Graphite, simulating 1024 tiles on ten host machines achieved a speedup of 3.85, parallel Transformer [48] achieved an average speed up of 35.3% over GEMS [49] sequential simulation. However, we should not be much optimistic about this approach because even when the simulation is parallelized, things are still sequential inside the single simulation thread. Moreover, if the number of target cores and hence the number of simulation threads is greater than the number of the available host cores, then the simulation threads have to be scheduled on the available cores and not all of them can run concurrently.

The most important thing that should be considered when parallelizing software simulators is that the target CMP should work as one unit to achieve higher accuracy. Thus, the simulation threads have to communicate in order to be aware of the state of each other; this is known as simulation thread synchronization. For cycle accurate simulation, this synchronization has to be done after each target clock cycle, which prevents achieving significant simulation speedup via software-based simulation parallelization.

Most of the existing parallel software simulators use loose synchronization, i.e., they scarify a degree of accuracy to gain more simulation speedup. In loose synchronization, the simulation threads are synchronized upon a certain event or periodically instead of synchronizing them after each target clock cycle, e.g., HORNET uses periodic synchronization [50], Graphite uses lax synchronization [38], Sniper uses barrier synchronization [39], and SiMany uses spatial synchronization [51]. These loose synchronization techniques will covered in more details in the next chapter.

Recently, FPGAs appeared as ideal accelerators for CMPs' simulators due to the massive fine and coarse grained parallelism they offer. Using FPGAs, the concurrent components of the target CMP can be mapped to concurrent models on the FPGA. Thus, the parallel events of the target CMP can be simulated in parallel and hence the simulation speed is greatly improved. Moreover, FPGAs are more realistic for CMP simulation because the concurrent structure of the target CMP's model resembles the target CMP structure itself and hence higher accuracy is achieved. Although FPAG-based simulators are faster than their software counterparts by orders of magnitude, there are three drawbacks related to this approach:

1. **Design Complexity:** developing a hardware model of a multicore machine is time-consuming and requires advanced skills in hardware design and verification. However, this issue has been greatly alleviated due to high level hardware description languages, such as SystemVerilog, Bluespec System Verilog, and SystemC.

2. **Lack of Flexibility:** FPGA-based simulators lack flexibility and usability because they require users to be able to implement designs on FPGAs. However, this issue can be alleviated by developing a friendly software frontend that interacts with the FPGA on the user's behalf.
3. **Limited FPFA Area:** Although recent FPGAs are large enough to host multicores' simulators, FPGA area is still limited and hence it can only host a model up to a certain limit. In order to host larger models, however, either multiple FPGAs are used [9] [52] or a smaller model is timely-multiplexed among a larger model's components [46, 53]. It is important to note that the former approach is costly, whereas the latter increases the simulation time and sacrifices a degree of accuracy.

CHAPTER 4

REVIEW OF EXISTING MULTICORE SIMULATORS

Computer architecture simulation is an old open research problem. In 2006, Yi and Lilja [54] surveyed the computer architecture simulation techniques at that time. This chapter focuses on the recent major efforts in multicore architectures simulation. The reviewed simulators in this chapter are classified based on their implementation technology to software, FPGA-based, and hybrid simulators.

4.1 Software Simulators

This section presents some of the key multicore simulators that were implemented as pure software.

4.1.1 GEM5

GEM5 [24, 55] is a full-system computer architecture simulation infrastructure that merges the best aspects of M5 [56] and GEMS [49] simulators. M5 provides configurable simulation framework, multiple ISAs, and multiple core models. GEMS complements these features by providing a detailed and flexible memory system, multiple cache coherence protocols, and different interconnect models. GEM5 was jointly developed by multiple academic and industrial institutions including AMD, ARM, HP, MIPS, Princeton, MIT, and the Universities of Michigan, Texas, and Wisconsin.

GEM5 offers the flexibility to the user to simulate the target architecture at different levels of details and hence control the accuracy-speed trade-offs. To achieve that, GEM5 provides different models of different levels of abstractions for the main components of the target architecture, e.g., different CPU models and different memory system models.

GEM5 supports different ISAs such as, ARM, ALPHA, MIPS, Power, SPARC, and x86. Moreover, it supports four different CPU models, **(1) AtomicSimple**, which is a simple un-pipelined one-IPC model that completes one instruction per clock cycle. **(2) TimingSimple,** it is the same as *AtomicSimple,* but it simulates the timing of memory references. **(3) InOrder**, it is an “execute-in-execute” accurate model of an in-order pipelined CPU. **(4) *O3***, it is an “execute-in-execute” accurate model of an out-of-order pipelined CPU. “execute-in-execute” refers to that instructions are executed only in the execution stage after all dependencies are resolved. . Thus, GEM5 is an example of integrated execution-driven simulators. Although the last two models emphasized accuracy, it was not claimed that they are cycle-accurate models.

GEM5 inherited two memory models, **(1) Classic mode**, which was inherited from M5 [56] simulator. This model is easily configurable and fast. **(2) Ruby model**, which was inherited from GEMS simulator [49]. It is a flexibly infrastructure that allows accurately simulating different cache-coherent memory systems.

Regarding NoC modeling, Ruby memory model can create any NoC topology since it is composed of point-to-point links. In a simple Python file, the connections among the components are determined and shortest-path algorithms are used to generate the routing tables. Ruby has two network models, **(1) simple model**, which models the router and link latency and the link bandwidth. However, it does not model contention and flow control. Thus, it sacrifices a degree of accuracy for the sake of faster simulation. **(2) Garnet model,** which includes detailed router microarchitecture and a timing model of contention and flow control. *Garnet* model is suitable for NoC studies.

GEM5 can operate in two modes, **(1) Full-System mode**, which models an operating system, and the computer devices such as IO peripherals. It simulates user-level and system-level codes. In this mode, GEM5 is capable to boot Linux operating system. **(2) System –call Emulation mode**, which does not include a complete OS model. Hoverer, it emulates most common system calls, such as reading from a file operation. When a system call is encountered, gem5 traps and emulates that call, often by passing it to the host operating system.

The current version of GEM5 is sequential. Thus, GEM5 suffers from long simulation time, especially when it is used for detailed architectural models of CMPs. Moreover, configuring the target architecture requires that the user has hands on experience in Python, which is not always guaranteed.

4.1.2 Graphite

Graphite [38] was developed at MIT as a user-level parallel software simulation infrastructure that targets multicore architectures. It is an open source distributed simulator that runs on commodity Linux machines. Graphite is a functional-first execution-driven simulator, in which the functional model runs ahead of the timing model. It is a flexible and configurable simulator, which makes it convenient for the user to explore many architectural alternatives. It has a modular architecture such that each component is modeled as a separated module with well-defined interfaces. Thus, a new target architecture instance can be configured via swapping the appropriate modules.

Graphite uses pin tool to functionally execute the workloads. The executed instructions along with their information, e.g., memory references, are consumed by the timing model. Graphite's core model is an abstract in-order model that is responsible for deriving the execution time of the workload on the target core via accumulating the latencies of the different events. Thus, Graphite is not a cycle-accurate simulator because it has a high level abstraction of the core model. When an "uncore" event occurs, the NoC model computes the round-trip latency of the network message generated by this event, e.g., a load miss event, then the memory model adds the memory access latency, and finally the core model accumulates these latencies on the target execution time

Simulation in Graphite includes running a multithreaded application on the target architecture defined in Graphite simulator. Each application thread is mapped onto a tile in the target architecture and each target tile is mapped onto a Graphite host thread (simulation thread). Graphite host threads are distributed on the cores of the distributed host machines, and the host operating system scheduler is responsible for scheduling these threads.

To achieve a higher simulation speed and scalability from the budget of simulation fidelity, Graphite uses different loose synchronization techniques for synchronizing the different target clocks. These techniques are based on what so called lax synchronization, which allows the clocks of the different target cores to differ from each other and the true synchronization occurs occasionally. One flavor of this lax synchronization is barrier synchronization in which the simulation threads wait on a barrier after a certain number of clock cycles specified by the user. This technique allows the user to trade accuracy for speed as desired. The higher the frequency of this synchronization barrier the higher the simulation accuracy and the lower the simulation speed.

Regarding Graphite speed, for SPLASH-2 benchmarks, the simulation time was longer than the native execution time by 1751 times when one host machine was used. However, this slowdown was reduced to 1213x when eight host machines were used.

4.1.3 Sniper

Sniper is a parallel software simulator proposed by Penry et al. [39] to simulate multicore and multiprocessor systems. It was derived from Graphite simulator [38] by adding the interval model [43] to Graphite. Sniper's level of abstraction (interval modeling) falls between the accurate-slow detailed microarchitecture models and the inaccurate-fast abstract models, such as the one-IPC model.

The interval model is a mechanistic analytical model, where the execution time is split into intervals by miss events, such as: branch miss-predictions, load misses, etc. Each interval has two subintervals, (1) the busy subinterval, in which the core is doing useful work, and (2) the non-busy subinterval, in which the core is idle.

Sniper models the timing for individual target cores. It maintains a window of instructions per target core. This window corresponds to the reorder buffer in the out-of-order cores and is used to detect the overlapping between the miss events and the long latency load misses. The functional simulator (Graphite in this case) is responsible for executing the instructions, detecting the miss events, and injecting them into the instruction window’s tail. Thus, in addition to the functional execution of the application, Sniper requires that the functional model has to model the functionality of different target architecture’s components, such as the cache hierarchy and NoCs in order to detect the miss events related to these components.

Sniper is considered a functional-first execution-driven simulator because the functional model runs ahead of the timing model. Regarding OS modeling, Sniper is considered a user-level simulator. However, it assigns constant latencies to some OS related events, such as spinlocks

Sniper’s timing model derives the simulated time of a certain target core based on the analytical interval model. It consumes and manipulates the executed instructions from the instructions window’s head. If a miss event is encountered by the timing model, the penalty of this miss event is added to the core’s simulated time. Otherwise, the instructions are dispatched at the effective dispatch rate, and the simulated time is incremented by the average instruction execution time that excludes miss events’ penalties.

Sniper has a unique feature, namely the CPI (Clock per Instruction) stack, which is a stacked bar. It breaks up the target execution time into its different components, such as computation time, synchronization time, cache misses’ penalties, etc. This feature is very useful, because it explains where the execution time has been spent. It helps the software developer to identify the performance bottleneck and therefore makes the suitable improvements. Sniper achieved a speed of up to 2 MIPS with absolute average error of 25%, when it was validated against real hardware for a variety of multi-threaded workloads.

4.1.4 PinPlay

PinPlay is a framework for deterministic regeneration of a program execution. It is based on Intel pin dynamic binary instrumentation framework, namely pin. Its main objective is to address the non-determinism in multithreaded program execution. Successive runs of the same multithreaded program have different threads interleaving and different shared memory access order. For debugging and computer architectural simulation purposes, it is desired to have one deterministic execution of the program.

In Pinplay, the program is executed once and some information is recorded in order to regenerate the same execution again and again. PinPlay comprises the following two pin tools:

1. **Logger:** is a Pin tool that takes the binary program alongside its input set as input. Then, the program is instrumented and natively executed under this Pin tool. The logger captures the initial architecture state (initial memory image and initial registers values) and non-deterministic events during a program execution in a set of files called ***pinballs***. Due to heavy instrumentation (every instruction is instrumented), the logger is slower than native execution by 100-200X.

2. **Re-player:** is another Pin tool that is run on the pinballs to deterministically reproduce the execution that was captured by the logger. It can be combined with an execution-driven architectural simulator to allow simulation based on pinballs instead of the original program binary and hence perform apples-to-apples comparison because the same execution is used in multiple simulations. Moreover, it can be combined with a debugger to debug a deterministic execution of a multithreaded program. The re-player is slower than native execution by less than 50X.

Figure 4 shows a high-level block diagram that depicts the workflow of Pinplay.

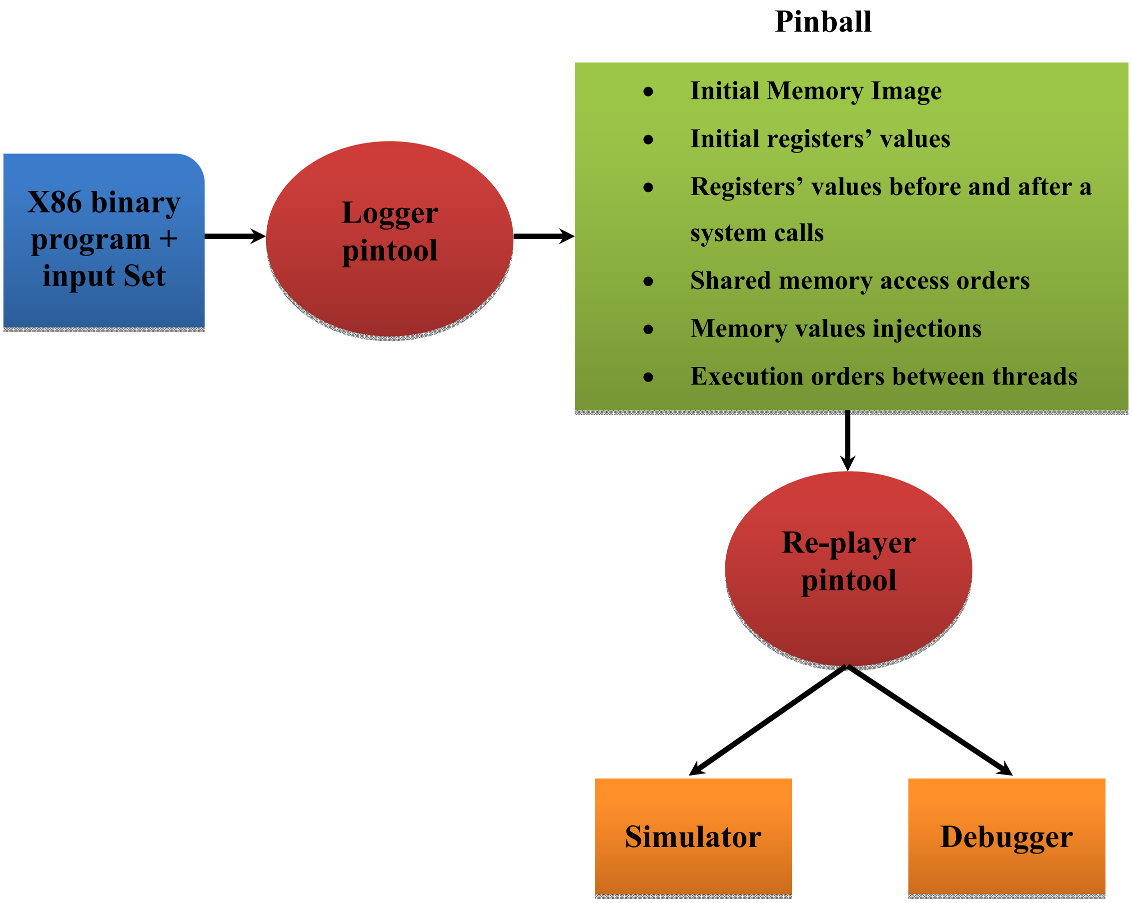

Pinball
Initial Memory Image
Initial registers' values
Registers' values before and after a system calls
Shared memory access orders
Memory values injections
Execution orders between threads
X86 binary program + input Set
Logger pintool
Re-player pintool
Simulator
Debugger

Figure 4: High Level Block Diagram of PinPlay Framework Workflow

4.1.4.1 Pinballs

A Pinball is a user-level format that is created and consumed under Intel’s Pin framework. It is a checkpoint produced by the logger. It can be loaded and replayed to repeat the captured program execution. Pinball is self-contained, i.e., the binary program and input data set are no longer required after logging. Pinball is not a trace and not a sequence of static records. The difference between Pinball execution (replay) and the original program execution is that in replay the system calls are skipped and only their side effects (on the registers) are injected. Moreover, in replay, shared memory accesses by multiple threads are forced to be in the captured order. Otherwise, replay is simply a normal execution of the original program.

Therefore, a Pinball keeps only the information that is required to repeat the captured execution. Pinball is organized into multiple text files. Some files are global (for all threads) and some of them are per thread (each thread has its own copy of the file). The following are the most important pinball files:

1. ***.text,** global, it contains the initial memory image.
2. ***.sel (system effects log),** per-thread, memory value injections tagged by instruction counts, i.e., the injection occurs when the number of instructions executed by the thread reaches the recorded value.
3. ***.reg,** per-thread, multiple register value records for initial register state, registers differing from before and after system calls, etc.
4. ***.race:** per-thread: records to enforce shared memory access order between threads. e.g., if the file is for thread i, the records are of the format ‘i ***counti***

j $icount_j$’ implying thread i must wait at instruction count $count_i$ till thread j reaches instruction count $icount_j$.

5. ***.sync text,** per-thread, records to enforce execution order between threads. Its records have the same format as the ***.race** file.

4.1.4.2 PinPlay and Architectural Simulation

PinPlay can be combined with Pin-based simulators directly, such as Sniper. In this case, PinPlay serves as a functional model of such simulator, i.e., it replays the program (executes the pinball) and feeds the timing model with the executed instructions. However, for non-Pin simulators, there should be a convertor between the PinPlay format and the simulator format.

Since replay implies real execution of the program, Pinballs cannot be consumed by trace-driven simulators because pinballs execution needs functional units that are missing in such simulators.

4.1.5 McSimA+

McSimA+ [41] is a cycle-level detailed microarchitecture simulator for multicore and emerging many-core processors. It was jointly developed by Seoul National University and HP Labs. McSimA+ is a functional-first execution-driven simulator. The functional model is based on native execution under Pin tool. The executed instructions along with their information are injected to the event-driven timing model to derive the execution time of the workload on the target processor.

McSimA+ is not a full-system simulator. However, it implements a thread management layer to manage the target threads. Thus, McSimA+ falls between the application-level and full-system simulators and hence it was called *application-level+* simulator. They designed a special Pthread [57] library to be a part of McSimA+. This library comprises two parts: (1) Pthread controller, which implements the Pthread functionality, such as, thread creation, thread termination, and thread's storage management. (2) Pthread scheduler, which is responsible for scheduling the target threads on the target cores, i.e., blocking and resuming the target threads.

The core microarchitecture is highly detailed in McSimA+; it has a variety of detailed core models including single-threaded, multi-threaded, in-order, and out-of-order cores. Moreover, the target memory hierarchy model is highly detailed. McSimA+ supports a flexible cache hierarchy model that allows the user to explore different alternatives. Furthermore, McSimA+ supports multiple cache coherence protocols.

Regarding NoC model, McSimA+ supports different NoCs, such as, buses, crossbars, and multi-hop NoCs of different topologies (ring and 2D mesh). Moreover, McSimA+ models hierarchical NoCs, where the cores are grouped into local clusters and these clusters are interconnected via a global NoC. McSimA+'s NoC model has links and routers. The hop latency is a tunable parameter.

McSimA+ speed was not reported. Regarding accuracy, McSimA+ was validated against a real hardware, namely Intel Xeon E5540 using SPLASH-2 benchmarks. They compared the IPC computed by McSimA+ with the IPC resulted from the real hardware and the average absolute error was 14.2%.

Our proposed framework has a thread management layer same to McSimA+, but it is implemented in hardware (FPGA).

4.1.6 SiMany

SiMany is a discrete-event many-core simulator proposed by O. Certner et al. [51]. It supports task-based programming models, such as CILK and TBB (Threading Building Blocks). Each target core is simulated via a different simulation thread. However, these simulation threads are scheduled on a single host core. Thus, SiMany cannot be considered as a parallel simulator.

SiMany tried to increase the simulation speed by scarifying a lot of fidelity via raising the level of abstraction of the core, cache hierarchy, and NoC models. Thus, SiMany is not a cycle-accurate simulator. Moreover, SiMany has no OS model and no ISA emulation. The program is natively executed on the host machine. Once an inter-thread interaction is encountered, the timing model intervenes. Therefore, SiMany focuses only on the concurrent interactions among the target cores. The regions among the concurrent interaction points (the sequential parts of the code) are just executed natively, i.e., they are ignored, which greatly reduces the simulation accuracy. SiMany can be considered as a functional-first execution-driven simulator because the application runs ahead of timing model interventions.

SiMany uses what so called Virtual Time (VT), which is the clock of the target core. If all cores are perfectly synchronized, their VTs will be the same. However, VTs are synchronized in a distributed fashion, called spatial synchronization. When a memory access or remote request is issued by a core, it is initially stamped by the current value of the VT of that core. The value of this time stamp is increased incrementally while this request navigates through the model components.

In this spatial synchronization mechanism, the cores synchronize their VTs with their neighbors only. When the request comes back to its initiator core, this core's VT is updated to the latest value of the request's time stamp. Then this core sends a VT update message to its immediate neighbors and this update propagates to the whole network.

If a core's VT is greater than the VT of its neighbor by T, this core stalls until its neighbor's VT increases to be equal to its VT. This feature lowers the time drift between cores under T, which is a parameter specified by the user. It represents speed/accuracy tradeoff, the higher the T the faster and the less accurate the simulator, and vice versa is true.

SiMany has been validated against UNISIM-based simulator [58]. It showed a geometric mean of errors equals to 8.8% for 16 cores, 18.8% for 32 cores, and 22.9% for 64 cores. They claimed that SiMany speed is two or more orders of magnitude over the existing approaches.

4.1.7 HORNET

HORNET is a cycle-level parallel software simulator for many-core architectures proposed by P. Ren et al. [50]. It is a highly configurable simulator, which provides the architect with the required flexibility to explore the architectural space.

HORNET supports three flavors of core models, (1) trace-driven packet injector, which is suitable for simulating NoCs only, with this flavor, HORNET is considered as a NoC trace-driven simulator. (2) Single-cycle in-order MIPS core models, and (3) Threads of an executable run under Intel pin framework (native execution). In the last two models, HORNET is considered a functional-first execution-driven simulator.

HORNET has a configurable memory system, in which the user can specify the number of cache levels, sizes, private/shared, etc. Moreover, it implements MSI cache coherence protocol.

Regarding NoC modeling, HORNET possesses a cycle-accurate NoC model. It supports different topologies, such as ring and multilayer mesh. Also it supports both static and adaptive routing. Furthermore, HORNET can operate in NoC mode only, where the a trace is used to inject traffic to the NoC model

Periodic synchronization is used by HORNET to trade simulation speed for accuracy. It includes synchronizing all simulation threads on a barrier periodically. Increasing the synchronization period enhances the simulation speed from the accuracy budget, and vice versa is true.

4.1.8 Manifold

Manifold is an open source parallel full-system software simulation framework for multicores. It was proposed by J. Wang [44]. Manifold has a parallel simulation kernel as well as a library of micro-architectural components, which offers the architect the capability of building up a customized simulator from these micro- architectural components. It supports a range of core models that includes cycle-accurate models, analytical models, and k-CPI models. It uses parallel multicore emulator frontend to execute binaries. Manifold supports cycle accurate NoC components and different synchronization algorithms. Manifold's mean simulation speed was 242.03 KIPS.Regarding speed and accuracy, Manifold is not just a simulator, it is a simulation framework, and hence it supports both abstract and detailed components. Thus, the constructed model speed and accuracy vary according to the level of abstraction selected.

4.1.9 Transformer

Transformer [48] is a cycle-accurate full system simulator for multicores based on GEMS simulator [23]. It is a functional-first execution-driven simulator, where the functional model runs ahead of the timing model in Transformer. The output of each instruction executed by the functional model is fed to the timing model to evaluate its timing.

Transformer provides an architecture-independent interface between the functional and timing models to leverage simulator extensibility. In the case of functional-timing divergence, for example, a miss-prediction occurs in the functional model and a correction step is required, Transformer has a lightweight scheme to detect and recover from such scenario.

Transformer has been compared against GEMS simulator. The sequential version of Transformer achieved an average speedup of 8.4% over GEMS simulator. However, the average speedup was 35.3% when the functional and timing models were parallelized in a pipelined manner.

4.1.10 COTSon

COTSon [26] was jointly developed by HP Labs and AMD. It is a parallel functional-first execution-driven full system simulation framework that targets cluster-level systems of many cores. It uses AMD’s SiMNow simulator [59] for the functional simulation of the benchmark on each node of the cluster. All events generated by the functional simulator are fed to their timing models. It uses sampling techniques to improve the simulation speed. COTSon can dynamically adjust speed and accuracy.

4.1.11 Summary and Discussion

In this section, nine sequential and parallel software simulators have been reviewed. The flexibility and ease of development of software simulators compared to the FPGA-based ones made them popular in computer architecture community. However, the slowness of such simulators, especially when they target CMPs, pushed researchers to investigate how to accelerate these simulators.

In this section, we noticed that researchers tried to alleviate the slowness of software simulators in two ways, (1) scarifying a degree of accuracy via raising the level of abstraction of the target architecture model. This includes using simple models of the processor cores, NoCs, and memory subsystems. (2) Parallelizing such simulators and running them on the existing parallel machines.

Unfortunately, the slowness drawback of software simulators still exists even after these two solutions. Abstract models eliminate a fraction of the details to be simulated, i.e., they reduce the number of parallel events occurring in a single target clock cycle; however, this number is still high. Moreover, parallelizing software simulators partitions these parallel events and assigns them to multiple parallel simulation threads. This approach is supposed to achieve simulation speedup that is proportional to the computation power of the host machine. However, this speedup is limited because the parallel events are still simulated sequentially in the same simulation thread.

Another limiting factor of parallel simulators speedup is the inter-core communication for synchronization. In parallel software simulators, each target core is mapped to a simulation thread, such as in Graphite [38] and these simulation threads are mapped to different host cores. For cycle-accuracy, the clocks of the different target cores have to be perfectly synchronized. This means they have to communicate after each target clock cycle, which collapses the simulator performance. To prevent this performance degradation, the existing parallel software simulators, such as Graphite [38] and HORNET [50] use loose synchronization techniques in which the different target clocks can be synchronized periodically by letting the simulation threads wait on a synchronization barrier every fixed number of clock cycles. Of course, rescuing the performance via loose synchronization is from the accuracy budget.

Furthermore, in these simulators, the user is able to specify the time period separating each two synchronization barriers to adjust the accuracy/speed trade-off. At first glance, this looks as a good feature, although it is not. Because nothing will tell the user how accurate the simulator became after tuning the synchronization period.

Based on the discussion above, we conclude that there is a need for a solution to dramatically accelerate CMPs simulations. This solution is the FPGAs. The upcoming couple of sections present some concrete examples of FPGA-accelerated simulators.

4.2 FPGA-based Simulators

This section presents the key FPGA-accelerated simulators in which both the functional and timing models were hosted on FPGA.

4.2.1 RAMP Gold

RAMP Gold [8, 42] is a high-throughput and cycle-accurate FPGA-based simulator for many-core architectures that was developed at UC Berkeley. It is a timing-directed execution-driven simulator. Moreover, RAMP Gold is a full-system simulator that is capable of booting Linux operating system. RAMP Gold uses host-multithreading, it simulates 64 target cores on a single physical timing model using fine-grained time multiplexing.

RAMP Gold decouples the functional model from the timing model. The former executes the target ISA and maintains the architectural state, while the latter determines the time required by the target machine to execute an instruction and schedules the threads to be executed by the functional model.

It was claimed that RAMP Gold is a cycle-accurate simulator; although the NoC and cache coherence models are missing from this simulator. Moreover, RAMP Gold core’s model is just a simple one-IPC in-order single-issue core model that completes one instruction per cycle except in the case of a data or instruction cache miss. On the other hand, RAMP Gold has a detailed timing model of the cache hierarchy.

RAMP Gold achieved two orders of magnitude speedup over software simulators. It simulated a target machine of 64 cores at almost 50 MIPS. In terms of FPGA resource usage, RAMP Gold consumes 90% of the BRAM blocks, 25% of LUTs (LookUp Tables), and 34% of the registers in a Virtex 5 LX110T FPGA. The functional model consumes the significant part of the FPGA resources. These resources were consumed to implement the core’s components, such as fetch unit, decode unit, register file, ALU, and floating point units. Moreover, a significant amount of block RAMs were used to cache the input data set of the application. Therefore, moving the functional model to software will release these resources to build a more detailed and larger timing model.

4.2.2 HAsim

HAsim [53] was jointly developed by MIT and Intel. It is the FPGA-based implementation of Asim software simulator [25]. HAsim is a highly-detailed simulator that targets shared-memory multicore processors. It has a single highly detailed physical core, a single cache, and a single router on a single FPGA. The cores’ internal states (the program counters and the register files) are duplicated for each target core. HAsim is a timing-directed execution-driven full system simulator. It currently supports the Alpha ISA only.

HAsim simulates multiple target cores sequentially using fine-grained time division multiplexing, i.e., the single physical core is multiplexed among multiple target cores in a round robin manner. Each pipeline stage in the physical core can simulate a different target core, i.e., the number of target cores that can be simulated simultaneously is limited by the number of pipeline stages. This scheme is called host-multithreading, which means that the simulator has multiple threads and each thread is responsible for one target core. In FPGA-based simulators context, host-multithreading means the same hardware component, such as core model or router model is timely multiplexed among different target cores and the simulator keeps track of the architectural state of all of these cores.

HAsim simulates the on-chip network of any topology through permutations using a single physical router. For the message port in the ring network, the output from router0 is the input for router1, the output of router 1 is the input of router2, and so on. The output from router N-1 is the input for router0. For the credit port, 0 goes to N-1, 1 to 0, 2 to 1, and so on. This cross-router communication pattern is represented as a small permutation that can be stored in a queue and a side buffer.

HAsim’s accuracy was not reported. Concerning simulation speed, for a single-thread target architecture, HAsim used 11 FPGA cycles on average to simulate one target cycle and the simulation rate was 4.54 MHz, i.e., it can simulate 4.54 million target cycles on average per second. However, for sixteen threads, HAsim used 80 FPGA cycles on average to simulate one target cycle and the simulation rate was 625 KHz. Regarding FPGA resources; HAsim consumes 57% of the FPGA registers, 79% of LUTs, and 27% of the BRAMs when 16 target cores are simulated on a Virtex 5 LX330T FPGA.

4.2.3 Arete

Arete [9] is an FPGA-based cycle-accurate simulator for multicore PowerPC architecture. It is a full-system simulator that is capable to boot an off-the-shelf SMP (Symmetric Multiprocessing) Linux to run unmodified applications, such as PARSEC benchmark suite. Arete is an execution-driven simulator that tightly integrates the functional and timing models together. Furthermore, Arete does not implement host-multithreading, i.e., all target cores run concurrently which makes it more accurate.

Arete's target architecture is tile-based. Each tile contains multiple PowerPC cores. Each core has 10-stage in-order pipeline. Moreover, each tile has two cache levels, where L2 is shared among all tile's cores. Arete implements a bidirectional NoC, which supports point-to-point topology. Also it implements a directory-based MSI (Modified, Shared, and Invalid) cache coherence protocol.

For the efficient use of FPGA resources, the LI-BDN (Latency Insensitive Bounded Data Networks) technique [60] was used. LI-BDN aims at reducing the FPGA resources usage by using multiple FPGA clock cycles to simulate one target clock cycle.

Arete's average speed was 55 MIPS for 8 cores on 4 FPGAs, and up to 11 MIPS for one core on a single FPGA. In terms of FPGA resources consumption, Arete is expensive because it covers all components of the target architecture in details. One Virtex 5 FPGA can fit for up to two realistic PowerPC cores.

4.2.4 ScalableCore system 3.3

ScalableCore system 3.3 [52] is a cycle accurate FPGA-based full-system simulator for mesh NoC-based tile architectures. The main goal was to achieve scalability, i.e., the simulator allows adding more cores. They had two contributions:

1. **Local Barrier Synchronization:** to satisfy the cycle-accuracy, the newest simulation state is transferred to the neighbor units in the next clock cycle. Each node will be updated about its four neighbors only. This local barrier synchronization strategy allows adding more cores without a need to increase the synchronization overhead.
2. **Virtual Cycle:** they used multiple FPGA cycles to implement one target cycle.

ScalableCore's target architecture is the M-Core architecture, which is mesh NoC-based tiled architecture. It consists of many homogenous cores. The communication among the cores and the off-chip memory occurs through DMA (Direct Memory Access). Each core is connected to its four neighbors.

In 100 nodes simulation, ScalableCore was 129 times faster than SimMc (software counterpart simulator for M-Core running on Core i7 processor). Although this simulator is scalable and cycle-accurate, it is very expensive because each target core needs to be hosted in a separated FPGA device to avoid time division multiplexing.

4.2.5 Summary and Discussion

This section summarizes the findings of surveying the existing FPGA-based simulators for multicores. Table 2 summarizes the main features of these simulators. All of the existing simulators decouple the target cycle from the FPGA cycle (host cycle), which allows the target cycle to be simulated in multiple FPGA cycles and hence less FPGA resources.

From Table 2, it is clear that simulators without time multiplexing can simulate only very few number of cores. This is because the functional model occupies a significant area on the FPGA and the design components are not reused through time multiplexing. Thus, it would be more efficient to implement the functional model as software and move it to the PC. In this case, the CPU functional units are utilized to functionally execute the application using the host's native instructions and hence more FPGA area is freed to host a larger timing model.

Although time multiplexing increases FPGA resources' utilization, it sacrifices a degree of fidelity. Because the state of only some core (s) can be visible at a single host clock cycle and the states of other cores and the messages on the NoC are hidden, i.e., no complete snapshot of the target architecture can be taken in the same host clock cycle.

None of the surveyed simulators modeled L3 cache, although the majority of recent CMPs have this level, and it is in tens of megabytes. Adding L3 cache to these simulators will dramatically reduce the number of cores that can be simulated, because L3 model will occupy a significant fraction of the FPGA BRAMs. This reemphasizes the conclusion that the functional model has to be moved to the PC.

Table 2: Summary of the FPGA-based Simulators

Simulator	Time Multiplexed	Core Details	NoC	Cycle-Accurate	Full system	No. Target Cores/FPGA
RAMP Gold	Yes	No	No	No	Yes	64
HAsim	Yes	Yes	Yes	Yes	No	16
Arete	No	Yes	Yes	Yes	Yes	2
ScalableCore	It can be	Yes	Yes	Yes	No	1

4.3 Hybrid Software/Hardware Simulators

This section presents two hybrid FPGA-accelerated simulators, namely, PROTOFLEX and FAST.

4.3.1 PROTOFLEX

PROTOFLEX [10] is an FPGA-accelerated hybrid functional simulation/emulation platform that was designed at Carnegie Mellon University. It does not include a timing [61] model; however, it was intended to utilize FPGAs to accelerate the functional simulation only. It provides the same functionality as Simics simulator [61].

The frequent behaviors (common operations), such as arithmetic operations are emulated in hardware, and complex and infrequent behaviors, such as the I/O devices are simulated as software. Hardware emulated and software simulated components of the target system run concurrently on their respective hosts. PROTOFLEX is a full-system simulator that is capable of booting Solaris 8 and running commercial workloads. Moreover, it employs host-multithreading via time-multiplexing to simulate multiple SPARC V9 cores.

Coupling PROTOFLEX with a software timing model will not accelerate simulation because timing simulation is the most critical part and it supposed to be targeted by simulation acceleration. In contrast, this coupling might slowdown the simulation because the timing model will wait for responses from the FPGA to proceed.

In other words, in such hybrid approach, there will be a performance bottleneck on the FPGA/PC boundary.

On the other hand, coupling PROTOFLEX with a hardware timing model on the same FPGA makes the design larger and hence smaller target architecture can be simulated without time division multiplexing.

Based on the discussion above, it is better to offload the functional part to native execution to utilize the host machine resources to functionally execute the application and saves the FPGA area to host larger timing models.

4.3.2 FAST

FAST is a hybrid software/hardware simulation methodology developed at The University of Texas at Austin. It produces fast, complete and cycle accurate simulators. In their first implementation [47], FAST supported single core simulation. It achieved an average simulation speed of 1.2 MIPS. FAST consists of two parts, (1) simulator-level speculative functional model implemented using a modified full- system software simulator [62], and (2) timing model implemented on an FPGA. The functional model is responsible for the ISA level simulation, whereas the timing model captures the micro-architectural timing features of the target architecture.

In FAST, both functional and timing models run in parallel. The functional model executes instructions independently from the timing model. Then, it passes the executed instruction trace to the timing model, which simulates the timing of the executed instructions according to the micro-architectural model. Thus, FAST is a functional-first execution-driven simulator. FAST's timing model affects the order of instruction execution, when it detects a miss-speculation caused by the functional model; it corrects the functional model by commanding it to roll-back and returns to the correct path.

4.3.3 Summary and Discussion

Table 3 summarizes the main features of the two reviewed hybrid simulators, namely, FAST and PROTOFLEX.

As stated before, in hybrid simulators, either the functional or the timing model is hosted on an FPGA and the other on a PC. Having these two models running concurrently will reduce the simulator scalability and speed due to the intensive communication on the FPGA/PC boundary. The situation gets worse when rolling back is required to correct miss-speculations.

Again, it would be more efficient for PROTOFLEX to implement the functional model as software and the timing model on the FPGA. In such implementation, the functional units of the host machine are utilized to perform the complex arithmetic operations and hence the whole FPGA can be utilized to simulate larger timing models.

Table 3: Summary of the FPGA-based Hybrid Simulators

Simulator	Functional Model	Timing Model	Time Multiplexed	Core Details	NoC	Cycle-Accurate	Full system
FAST	Software	FPGA	No	Yes	No	Yes	Yes
PROTOFLEX	FPGA	Software	Yes	Yes	No	No	Yes

CHAPTER 5

OVERVIEW OF THE PROPOSED SIMULATION FRAMEWORK

This chapter presents the proposed simulation framework. It also summarizes all the design decisions and trade-offs that have been evaluated to reach the current version of the framework.

5.1 Basic Strategy

The basic strategy of the proposed simulation framework can be summarized as follows:

1. Exploiting FPGAs to accelerate CMPs simulation while keeping FPGA design issues transparent to the end user. This transparency is achieved via a software layer between the user and the FPGA, i.e., users (such as computer architect and application developers) won’t need to write HDL code (such as Verilog or VHDL).
2. Modeling the largest possible target architecture on a single FPGA without time division multiplexing. Therefore, the functional model has been implemented as software, namely, using Intel pin instrumentation tool to free more FPGA

resources for a larger timing model. Thus, the proposed simulation framework is hybrid and hence it is called HySim (**Hy**brid **Sim**ulator).

3. The functional and timing models are completely separated and hence there is no communication bottleneck at the FPGA/PC boundary. The functional execution is done first and then the execution trace is fed to the timing model later. Thus, HySim is a trace-driven simulation framework.
4. To avoid storing large traces on the FPGA, the execution trace is compressed in an executable code format called CET code (Compressed Executable Trace code). The fraction of the trace that cannot be embedded into the CET code is kept besides the CET code and called the CET data.
5. HySim's timing model is able to interpret the CET code and data, and hence regenerate the original execution events on-the-fly.
6. HySim implements a threads management layer. Therefore, the multi-threading events such as, thread creation, termination, locking, and unlocking are encoded into the CET code. Moreover, the timing model is capable of executing these events and hence preserves the timing-dependent threads interleaving that is lost in the traditional trace-driven simulators. Thus, HySim combines the convenience of trace-driven and the accuracy of execution-driven simulators.

Since HySim is intended for early architectural exploration, there was no need for a detailed microarchitecture model at this stage. Thus, an abstract base-CPI core model is used. The base CPI includes the "incore' time and excludes the "uncore" events. The "incore" time includes computation time, miss-prediction penalties, hazards' penalties, etc. whereas the "uncore" one includes cache miss penalties, NoC latency, etc. The latter is added during timing simulation. The base-CPI is a tunable architectural parameter. Therefore, the base-CPI is added to the target core execution time for every instruction. The timing of instructions that do not result in “uncore” miss events (e.g., ALU and control instructions) is solely covered by the base-CPI. However, in the case of instructions that cause “uncore” miss events, the penalty of these events are added to the target execution time. E.g., in the case of a cache read miss, the access time of all memories accessed to serve this event and the NoC latency, if any, is added to the target core execution time in addition to the base-CPI.

5.2 Functional and Timing Models’ Implementation Options

In hybrid FPGA-accelerated simulators, there are two options for implementing the decoupled functional and timing models:

1. Implementing the functional model on FPPGA and keeping the timing model in software, e.g., PROTOFLEX [10]. In this option, timing simulation is not accelerated and it remains sequential, although simulation acceleration is supposed to be intended for the timing model. The only thing that can be accelerated in this option is some complex arithmetic operations. Thus, this option was excluded from our strategy.
2. Implementing the functional model in software and the timing model on FPGA, e.g., FAST [47]. This option makes more sense because timing simulation will be greatly accelerated. Moreover, the already existing functional units in the host

machine are utilized for functional execution (through native execution). Therefore, we adopted this option.

After deciding how to implement the functional and timing models, we need to determine how they interact. Running them simultaneously and letting them invoke each other requires a high-bandwidth communication link between the FPGA and the PC. Moreover, this link can be a performance bottleneck, especially when simulating a large number of cores. Therefore, we decided to separate them completely and make HySim a trace-driven simulator that can preserve the correct threads ordering during timing simulation.

5.3 FPGA-based Simulation Framework Design Options

When FPGAs are used for computer architecture simulation acceleration, there is a critical trade-off between the flexibility and usability of the simulator and its complexity. The FPGA-accelerated simulator design options can be classified based on their flexibility into three options, (1) rigid simulator, (2) fully-flexible simulator, (3) quasi-flexibly simulator option which falls in between. The rest of this section delves into the details of these three design options.

5.3.1 Rigid FPGA-based Simulator

A rigid FPGA-based simulator is a one that models a specific target architecture (whether detailed or abstract). As such, it has no flexibility and new HDL code has to be generated for every architectural change, synthesized and downloaded to the FPGA. This means that an experiment would take about a full working day to implement. Besides the significant time and effort required for customizing this simulator to another instance of the design space, it requires that the user possesses advanced skills in circuit design and hardware description languages, which is not guaranteed. Moreover, the user should be familiar with FPGA platforms and synthesis tools. Furthermore, it requires resynthesizing the design and reconfiguring the FPGA even for a slight change in the target architecture. Thus, this approach has been excluded from our strategy.

5.3.2 Fully-flexible FPGA-based Simulator

In such a simulator, the base FPGA model is fixed and only run-time parameters are used to change the model run-time behavior. Hence, running architectural experiments involves only changing input parameters to the model. This requires the model to be highly configurable and very inclusive of all possible variations, something very difficult and costly with hardware models.

Initially, we targeted this ambitious approach, which offers full flexibility to the user. In this approach, the FPGA design issues are completely transparent to the user, i.e., the framework is used as if no FPGA exists in the picture. Thus, the user does not need to have any background in hardware design and verification. It allows the user to prepare a new experiment with only several mouse clicks. Moreover, the design is synthesized only once and also the FPGA is configured only once. To reach this level of flexibility, the FPGA-based simulator should be very generic and a new instance of the design space can be configured by changing the architectural parameters at runtime. This saves a lot of time because design synthesis and FPGA configuration requires significant time, usually more than the simulation time for some benchmarks or applications.

Although this simulator is a dream for the end user, developing such simulator is too complex. Because building a very generic simulator that covers all instances of the design space is not a trivial task. It will take a long time (usually in years) and require a large team of skillful hardware engineers and computer architects. Moreover, such generic simulator usually is very large and hence requires multiple expensive FPGAs to host it.

5.3.3 Quasi-flexibe FPGA-based Simulator

After realizing the complexity of the fully-flexible simulator, we decided to make HySim less ambitious at this point, but much more flexible than the rigid one. In HySim, the FPGA design issues are still transparent to the end user. It contains a Verilog template of a shared-memory multicore architecture timing model. This template is used as a mold to generate new timing models for different shared-memory multicore configurations. The Verilog template contains a default timing model instance. A new timing model instance can be generated by changing some parameters, e.g., cache size, number of cores, cache associativity, etc. or by replacing the default modules by ready-made modules, e.g., changing the cache hierarchy from inclusive to exclusive or changing the last level cache from private to shared, etc. according to the user’s input. This cuts down experimentation set up time from days to few hours (most of the time is spent in the synthesis phase). The user won’t have to write any HDL code.

This Verilog template was auto generated from a BSV code (Bluespec System Verilog) [63] in which HySim's timing model was developed. BSV is a very high level and fully synthesizable hardware description language. We adopted BSV to reduce the time and effort required to develop the timing model Verilog template.

Regarding design re-synthesis and FPGA reconfiguration, HySim has three types of design parameters;

(1) Runtime parameters, which can be modified at runtime by passing their values to the timing model through the FPGA's ports. Thus, changing these parameters does not require design re-synthesis and FPGA reconfiguration (exploring different design points takes minutes).

(2) Reconfiguration parameters, changing these parameters require resynthesizing the design and reconfiguring the FPGA.

(3) Post-simulation parameters, changing such parameters does not even require re-simulation, such as, measuring the effect of changing the base-CPI, this parameter affects only the "incore" time, which can be computed by multiplying the number of executed instructions by the base-CPI. Thus the post-simulation parameters effect is captured through calculations and not through re-simulation. Table 4 lists all of these parameters with some description and default values. These default values are the values assigned to the parameters in the Verilog template.

Table 4: HySim's Parameters and Their Default Values

Parameters Category	Parameter Name	Default	Notes
Runtime Parameters	Number of sockets	2	
	Cores per socket	8	
	Threads per core	4	Number of threads that are scheduled on one core.
	L1 instruction cache latency	3 cycles for data access, 1 cycle for tag access	
	L1 data cache latency	3 cycles for data access, 1 cycle for tag access	
	L2 cache latency	13 cycles for data access, 3 cycle for tag access	
	L3 cache latency	38 cycles for data access, 12 cycle for tag access	
	Main memory latency	175 cycles	
	Reorder buffer size	96	
	NoC topology	Ring	Ring or mesh.
	Cache coherence protocol	MSI	It can be MSI, MESI, or MOESI, where M: Modified, S: Shared, I: Invalid, E: Exclusive, O: Owned.

Re-configuration Parameters	Cache hierarchy	inclusive	It can inclusive, exclusive, or not inclusive.
	Cache line size	64 Bytes	
	L1 I–cache size	32 KB per core	
	L1 D–cache size	32 KB per core	
	L2 cache size	256 KB per core	
	L3 cache size	20 MB per socket	
	L1 instruction cache associativity	8	
	L1 data cache associativity	8	
	L2 cache associativity	8	
	L3 cache associativity	20	
Post-Simulation Parameters	Base-CPI	0.5 clocks per instruction	
	Hop Latency	2 cycles	The latency of passing through one node on the NoC.
	CPU frequency	1.2GHz	It is used to convert from clock cycles to seconds.

5.3.4 Summary

Table 5 summarizes the pros and cons of the three simulation framework design options discussed above.

Table 5: Pros and Cons of Different FPGA-based Simulation Framework Design Options

Framework	Pros	Cons
Rigid Simulator	• Quick to develop	• Manual HDL code customization • Design re-synthesis and FPGA reconfiguration • Not transparent to the FPGA design issues
Quasi-Flexible Simulator (HySim)	• Transparent to the FPGA design issues • Automatic HDL code customization • Moderate design size • Moderate development time	• Occasional design re-synthesis and FPGA reconfiguration
Fully-Flexible Simulator	• Transparent to the FPGA design issues • No HDL code customization. • No design re-synthesis and FPGA reconfiguration	• Very large design size • Very long development time

5.4 HySim's Architecture and Workflow

Figure 5 shows a high level view of HySim's architecture. It comprises two main components, namely, the software frontend and hardware backend. The main purpose of the software frontend is to provide a friendly software layer between the user and the FPGA. Moreover, it contains the functional model of HySim (currently Intel pin tool) and the trace compression tool (CET tool). On the other hand, the hardware backend is the FPGA-based configurable timing model. It captures the timing characteristics of the target architecture and derives the execution time of the user application on that architecture.

As shown in Figure 5, the software frontend comprises a tool suite that comprises graphical user interface, Pin dynamic binary instrumentation tool [14], Xilinx ISE design suite, CET tool, and the control panel.

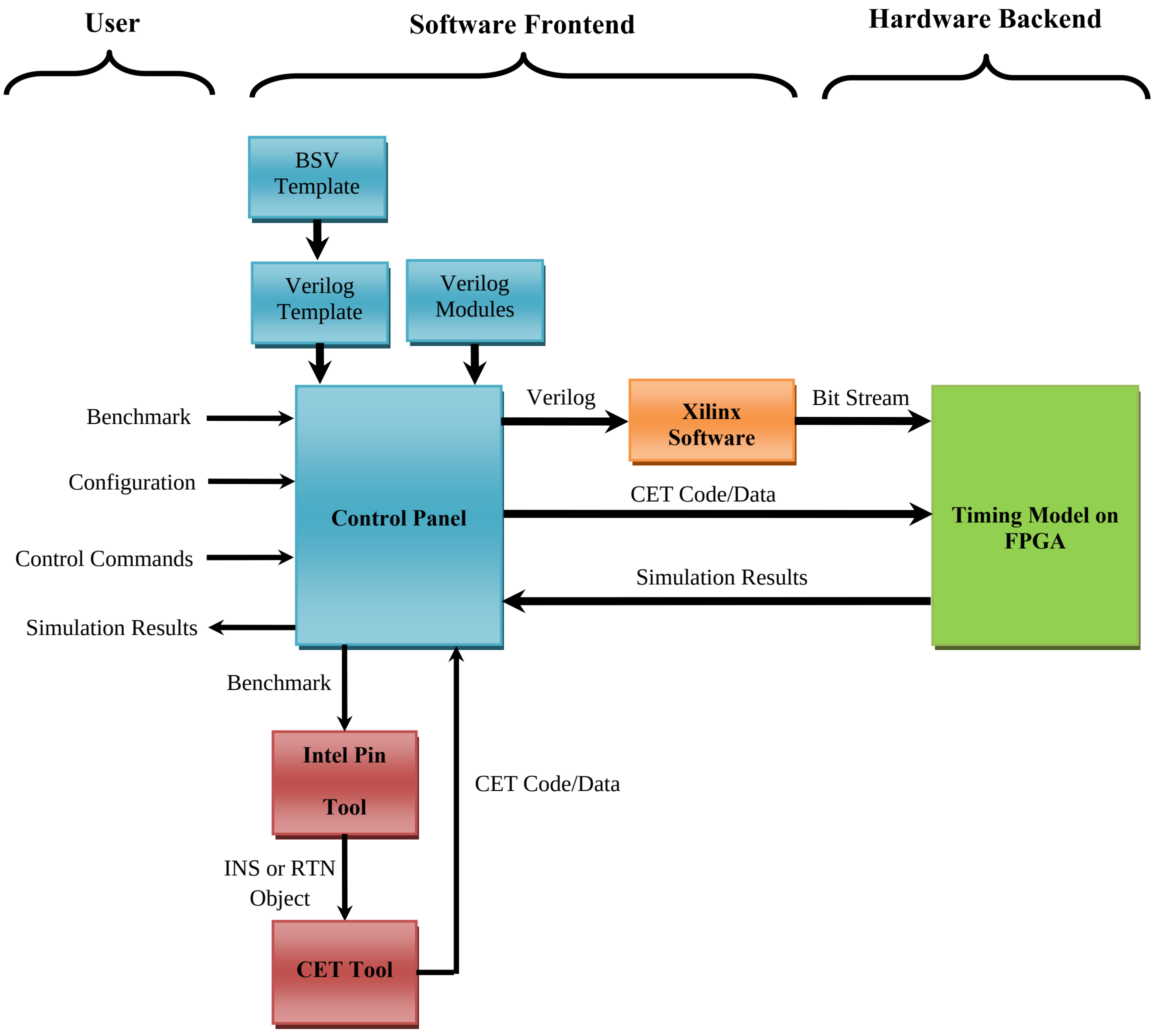


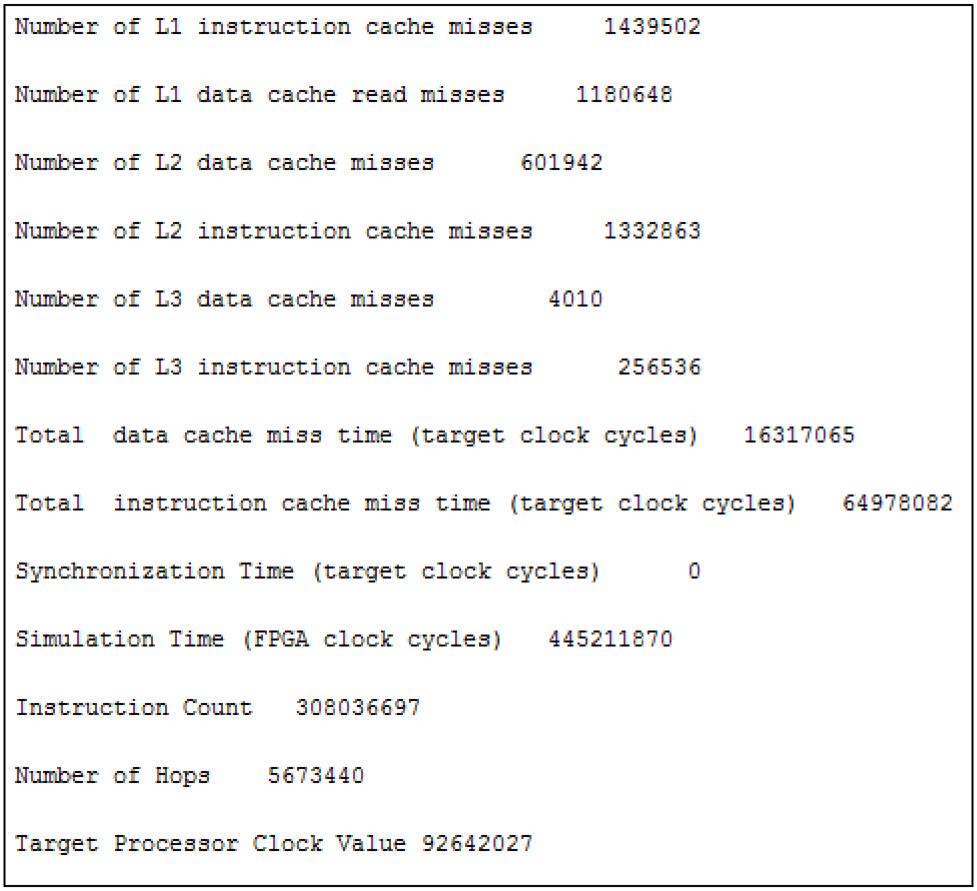
Figure 5: HySim Framework Structure

The following procedure summarizes HySim's complete cycle for performing one simulation experiment from scratch:

1. The user selects the benchmark/application and specifies the target architecture's parameters.
2. The control panel modifies the timing model's Verilog code's template to generate a timing model instance for the specified target architecture.
3. The Verilog code is fed to Xilinx software and the bit stream of the timing model instance is generated.
4. The application is natively executed and dynamically instrumented via Intel pin tool.
5. Intel pin intercepts each instruction and routine and sends its information, such as thread Id, instruction address, data memory address in the case of load/store, target address in the case of control instruction, the conditional branch instruction result (taken or not taken), etc. to the CET tool to generate the CET code and data of the application on-the-fly, i.e., without waiting for the whole trace to be generated.
6. The bit stream and the CET code and data are downloaded onto the FPGA.
7. The timing model executes the CET code with the help of CET data to evaluate the target architecture.
8. When simulation finishes, the control panel reads the simulation results from the FPGA and displays them to the user.

5.5 HySim's Output

HySim's output includes the simulation results, namely, the excepted execution time of the benchmark on the target machine (the simulated time). It also shows the different components of this execution time, such as computation time, synchronization time, data cache miss time, etc. Besides that, the simulation results include the number of cache misses at each cache level. Figure 6 shows a snapshot of the simulation results for one thread.



```
Number of L1 instruction cache misses     1439502

Number of L1 data cache read misses     1180648

Number of L2 data cache misses      601942

Number of L2 instruction cache misses     1332863

Number of L3 data cache misses        4010

Number of L3 instruction cache misses      256536

Total  data cache miss time (target clock cycles)   16317065

Total  instruction cache miss time (target clock cycles)   64978082

Synchronization Time (target clock cycles)      0

Simulation Time (FPGA clock cycles)   445211870

Instruction Count   308036697

Number of Hops    5673440

Target Processor Clock Value 92642027
```

Figure 6: A Sample Simulation Results for One Thread

CHAPTER 6

COMPRESSED EXECUTION TRACE GENERATION

As explained in chapter 5, our proposed hybrid simulation platform is composed of two parts; a SW frontend and a HW backend. The SW frontend generates a compressed trace of the input application (using instrumentation). In this chapter, the proposed trace generation and compression technique is described. After a brief description of the trace compression problem and the major existing techniques, details of the different phases of the proposed trace compression technique are provided. Experimental results for the compression ratio and speed achieved by our technique compared to other published techniques are presented at the end.

6.1 Introduction

Trace-driven simulation of computer systems has been widely used among computer architects and application developers [29]. This is due to its convenience and ease of implementation. A trace is generated once and can be used to carry out many architectural explorations via simulations. Trace-driven simulation can reveal considerable useful information, such as an application’s average clocks per instruction (CPI), cache performance, locality of references, efficiency of branch prediction and pre-fetching. A typical trace comprises the executed instructions along with their corresponding memory references. A trace-driven simulator can be a complete simulator for the whole computer system or specific for a certain component, such as a branch predictor or instruction cache. Trace fidelity refers to how many of the original execution events can be re-constructed from the trace.

In the multi-core era, researchers paid more attention to execution-driven simulation than trace-driven simulation because trace-driven simulators do not capture timing-dependent thread execution interleaving. However, researchers and architects continued to use trace-driven simulation to simulate multi-threaded applications on multi-core machines [28, 30, 31].

Another major challenge of trace-driven simulation is the large size of trace files. Although disk storage is currently inexpensive, the disk access time is still high. Moreover, the situation is not improved when FPGAs are used for trace-driven simulation due to their limited storage resources.

Although existing trace compression techniques succeeded in achieving excellent compression rates, these techniques still suffer from two drawbacks. First, all of these techniques take the full original trace as input. Because the primary objective of trace compression is to avoid having such large trace files, it would be more efficient to avoid having them from the beginning. In other words, it would be more efficient to start compression on-the-fly, i.e., during the original trace generation. The second drawback is that some of these techniques, [32, 64], require a decompression stage to reproduce the original trace. Decompression requires additional time and space and regenerates the huge original trace.

In [33], the authors proposed a lossless trace compression technique that exploits spatial and temporal locality. It performs an on-the-fly decompression. This technique is limited to instructions and their addresses, i.e., data addresses are not covered. The instructions’ addresses have been classified into two categories: (1) sequential addresses, in which the difference between any two consecutive addresses is constant, and (2) non-sequential addresses, in which the difference between them is variable.

The input trace consists of pairs of numbers. The first number is the instruction address, and the second is the instruction itself. The output comprises three components: (1) the static program instructions, any instruction is required is fetched from the static program using the instruction address; (2) sequent address file, it consists of a very long bit vector. Each bit corresponds to a trace element. If this bit is ‘0’, the corresponding address is sequential. If it is ‘1’, the corresponding address is non-sequential. (3) A file that contains the differences among the non-sequential addresses and can be compressed further based on locality.

In [34], the authors proposed an address trace compression technique based on loop detection. They used control flow analysis to detect loops in the address trace. They only handled constant and varying-by-constant addresses. They detected them by scanning the trace and finding the repeated patterns. The decompression stage implies running these detected loops. This technique does not handle complex situations in which loops have function calls and complex structures.

S. Budanur et al. [65] proposed a memory trace compression technique for SPMDs (single program multiple data). Their technique is based on PRSD (power regular section descriptors) [66, 67] abstractions but it is finer grained. They called it EPRSD (extended

PRSD). A pin based instrumentation tool (memtrace) takes an application as input and generates the memory trace of it. The generated trace is compressed using EPRSD. The memtrace tool runs as a set of MPI processes. Each process instruments an SPMD program and outputs the trace into a pipe. The trace compressor consumes the trace from the pipe. The compressor performs intra-thread compression utilizing the repetitive patterns. After instrumentation terminates, it performs inter-thread compression by factoring out the common parts among threads and finally performs inter-process merging among all processes of the SMPD application. This technique requires a decompression phase. It reduced the trace size by half for the AMG benchmark.

A. Janapsatya et al. [68] proposed a trace compression technique for instructions' addresses alongside an instruction cache analysis method. Their main objective was not to maximize the compression ratio but to accelerate trace processing. This technique is limited to instructions' addresses only. Their technique achieved a simulation speed up of 9.67 over the existing techniques, but the trace compression ratio was 2 to 10 times worse than Gzip.

In [32], four VPC (value prediction-based compression) algorithms were introduced, namely VPC1, VPC2, VPC3 and VPC4. In these algorithms, the input trace consists of pairs of numbers. The first number is a 32-bit PC, and the second one is a 64-bit extended data (ED). VPC algorithms use predictors to predict the next value based on the previously observed values. If the next value is predicted correctly, the index of the predictor that predicts it is output. The unpredicted values are output to a different stream. If more than one predictor predicts a certain value, there are heuristics to select the best one. For example, VPC1 uses Huffman encoding. If more than one predictor is correct, then the shortest Huffman code is selected. Because the number of predictors is small, the number of bits to encode the predictor's index is smaller than the corresponding trace element. Therefore, the trace is compressed. The same algorithm is applied in the reverse manner to decompress the compressed trace.

A. Ketterlin et al. [69] proposed a lossless trace compression algorithm. The input trace is a sequence of numbers. They scanned these numbers to detect loop nests using the linear progressions of these numbers. The output of this algorithm is a sequence of loop nests. This algorithm can handle simple loops only and is limited to data addresses. The decompression implies running the obtained loop nests.

Martin Burtscher proposed TCgen [70], which is a tool that auto-generates a value prediction-based trace compressor based on user specifications. The user describes the trace format in text for TCgen that generates the optimized C code of the specified trace compressor.

Kenneth C. Barr and Krste Asanovi´c [71] presented a technique to compress branch trace information to be used in snapshot-based microarchitecture simulation. The compressed trace can be used to warm up any arbitrary branch predictor's state before timing simulation of the snapshot. However, this technique is specific for branch information.

Kenneth C. Barr et al. [72] proposed a technique for directory and cache state reconstruction to accelerate sampled multiprocessor simulation. This reconstruction is like warming up. They used a software structure called MTR (Memory Timestamp Record) that can be updated during fast forwarding (functional simulator that updates the architectural state in between sampling points). For each memory block (cache block), there is an MTR record that registers the ID of the last processor that modified this block, the time stamp of the last write operation, and an array of time stamps of the read operations on the block (each timestamp per processor). During fast-forwarding, a read/write operation will update the MTR record.

The directory and cache state reconstruction occurs right before each sampling point. This is done in two steps: (1) determining the subset of blocks that are still cached. (2) Check cross-processor interactions to determine which of these blocks should be valid or dirty according to the cache coherence protocol. This technique works for sampled execution-driven simulators and it does not work for trace-driven simulators.

Other techniques, such as [73], [74], and PinPlay [75] concern about deterministic replay of the program by recording a fixed execution path for the non-deterministic events, e.g., threads interleaving and memory operations order. This deterministic replay is useful for software debugging and computer architecture simulation. However, since replay implies real execution of the program, then these techniques do not work for trace-driven simulators because real execution requires functional units that are missing in such simulators.

6.2 The proposed Execution Trace Compression Technique

This dissertation presents a novel methodology for efficiently compressing execution traces of multi-threaded applications running on multi-core architectures. A special compressed execution trace (CET) format has been developed. It retains all the low-level execution events (maximum fidelity), including threading events, with minimum size and can be processed directly without decompression. Hence, HySim's timing model can reconstruct all the execution events in the correct order from a CET trace including threading-related events (starting, sleeping, waking, synchronization, and termination). Also, a complete tool suit that generates the CET trace has been implemented and used to evaluate the proposed methodology.

The proposed trace compression method in this work translates a multi-threaded input application's or benchmark's executable into another binary format called CET code. The latter encodes the original application static code and the data required for timing simulation in a compressed format. The data that cannot be compressed, i.e. embedded into the CET code, is kept aside and is called CET data. So each thread of the application is translated into five files, namely the CET code, branch results, jump displacements, loop counters (in the case of inner loop whose counters do not follow a certain pattern), and data addresses (for non-uniform data referencing). The resulting CET code size is less than double the application's executable size. The CET data file size varies depending on the application. CET code and data are generated only once, for a specific input program, and can be used to simulate many architectural configurations. The only case in which the CET tool needs to be rerun for the same input program is when the number of threads changes. However, if the number of cores of the target machine changes and the number of threads is kept unchanged, then these threads are rescheduled on the new target machine configuration.

The compressed trace is intended for simulation only, not for debugging. The multi-threading synchronization events are captured in the compressed trace. The CET format defines primitives to create, pause, resume, and terminate threads. These primitives are used to implement barriers and locks/unlocks. Therefore, synchronization barriers, access to critical sections, and atomic read-modify-write operations are captured by the compressed trace

6.2.1 Basic Strategy

The basic strategy in the proposed compression technique is to remove all possible redundancy, both in instructions and data memory references, from the input execution trace while preserving fidelity. Our methodology implies constructing an executable static code (CET code alongside its CET data) from the input trace with the following features:

1) CET code preserves the execution order (control flow) of the original program without keeping any instructions' addresses except the initial thread address.
2) Contiguous data addresses, where consecutive addresses differ by a constant value, are captured in the CET code.
3) CET data includes:
    a. Non-contiguous data addresses. Only the difference from the previous address is encoded in the CET data, not the complete address. This reduces the size of these references by at least 50%. The user specifies the size of this field (default is 16 bits).
    b. The results of conditional branch instructions (taken or not taken) when the conditional branch is executed multiple times and it does not represent a loop instruction. The size of this field is 1 bit.

c. Dynamic target addresses of unconditional jump, call and return instructions. Only the displacement (in number of CET codes) between the current instruction and target instruction is stored. The user specifies the size of this field (default is 16 bits).

d. Loop counters (number of iterations) of the inner loops when the inner loop has a different number of iterations per outer loop iteration and these counters do not follow a certain pattern. The user specifies the size of this field (default is 32 bits).

Thus, each thread of the application is translated into five files, namely the CET code, branch results, jump displacements, loop counters, and non-contiguous data addresses differences. The resulting CET code size is less than double the application's executable size. The CET data file size varies depending on the application. CET code and data are generated only once, for a specific input program, and can be used to simulate many architectural configurations. The only case in which the CET tool needs to be rerun for the same input program is when the number of threads changes. However, if the number of cores of the target machine changes and the number of threads is kept unchanged, then these threads are rescheduled on the new machine configuration.

A specific tool has been developed to verify the effectiveness of the proposed CET code generation methodology. It can be integrated with the trace generator, i.e., the functional simulator or the instrumentation tool. This facilitates the start of compression on-the-fly, i.e., while the program is being executed, or emulated, and the trace is being generated, making our method extremely efficient in terms of time and memory requirements.

Figure 7 shows the work flow of the proposed CET generation methodology. The input is an executable file of the multithreaded program alongside its input data. This input goes through a chain of phases, namely profiler, code generator and the emulator and CET data generator. These three phases are repeated for all threads of the input program. The final output of the tool comprises the CET code and data for each thread separately. Producing separate CET codes and data for threads allows parallel processing of these threads (e.g., via the timing model). Moreover, the CET tool generates a log file of useful information for the user. It also generates the starting address of each thread. The current version of CET tool supports X86 architecture only. The Intel Pin framework [14] has been used for instrumentation. Other ISAs can be supported using other instrumentation tools such as Valgrind [40].

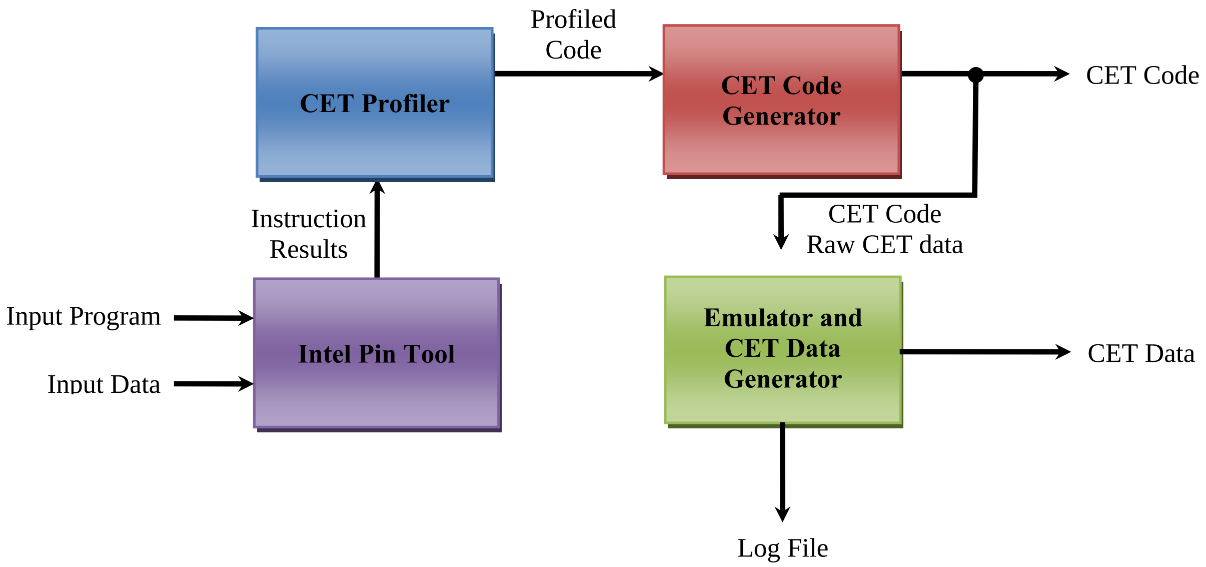


Figure 7: CET Tool Work Flow.

The rest of this sub-section delves into the different phases of the CET tool

6.2.2 CET Encoding

Instructions and function calls in the original execution trace are classified into one of **18** unique categories that belong to **six** different classes. These 18 categories are agnostic to any specific general purpose architecture. Each category is assigned a unique CET code and has special arguments. Table 6 summarizes the different instruction classes, categories, their CET code format, and their corresponding CET data (if any). In addition to the CET codes' formats shown in Table 6**,** the CET code contains the register numbers of the corresponding original instruction. This is important to capture hazards in the CET code. For example, the load instruction format can be: ***Load address, Rd, Ra;*** where Rd and Ra are the destination and the source registers, respectively. The 6 instructions and function calls classes are:

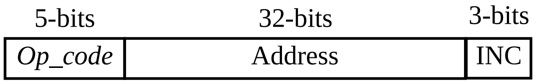
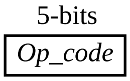
1. **Unconditional Branch Instructions:** includes the unconditional jump instructions, as well as the procedures' calls and return instructions.
2. **Conditional Branch Instructions:** includes all conditional branches. These instructions are used to encode loops.
3. **Memory Instructions:** includes all load and store operations.
4. **Synchronization Function Calls:** includes all system/library calls related to multi-threading, such as: thread creation, thread termination synchronization barrier, spinlock etc.

5. **ALU Instructions:** includes all ALU instructions of the original trace. They are classified according to their latency, of course, their functional unit, such as: integer ALU instructions, floating-point ALU instructions etc.
6. **System calls:** This class includes all other system calls not related to synchronization. The unique system call identifier/number is encoded in the CET code using 10-bits. This is more than sufficient for all existing operating systems where the number of system calls does not exceed 500. For example, Linux system call identifiers are available in many sites, e.g., [13].

Table 6: CET Code and Data format Summary

CET Code	CET Code Format	CET Data	Description
Unconditional Branch Instructions			
JUMP	5-bits 16-bits *Op_code* Displacement	None	jump/call/return instructions that always jump to the same target address
JUMP-M	5-bits *Op_code*	jump's displacement	jump/call/return instructions that jump to different targets
Conditional Branch Instructions			
BRANCH	5-bits 16-bits 1-bit *Op_code* Displacement BR	branch result (Taken/Not taken)	Normal conditional branch instruction. The BR bit records whether the branch was taken

LOOP	5-bits 16-bits 20-bits *Op_code* Displacement Counter	None	Loop instruction that always has the same counter (Number of iterations)
LOOP-C	5-bits 16-bits 20-bits 3-bits *Op_code* Displacement Counter INC	None	Inner loop instruction whose counter differs by constant (INC) each outer loop iteration
LOOP-R	5-bits *Op_code*	loop’s counters	Inner loop instruction whose counter differs by a random value each outer loop iteration
Memory Instructions			
LOAD/STORE	5-bits 32-bits *Op_code* Address	None	Load/store instruction that accesses the

			same memory location every time it is encountered
LOAD-C, STORE-C		None	Load/store instruction that accesses a contiguous block of data in memory e.g., vector. INC is the size of the data element
LOAD-NC, STORE-NC		Data addresses	Load/store instruction that accesses a non-contiguous (scattered) block of data in memory e.g., dynamic data structure

Synchronization Function Calls			
START		None	Start a new thread
PAUSE	5-bits 10-bits *Op_code* Thread Id	None	Pause a thread (corresponds to lock, wait, and sleep)
WAKE	5-bits 10-bits *Op_code* Thread Id	None	Wake a sleeping or waiting thread
TERMINATE	5-bits 10-bits *Op_code* Thread Id	None	Terminate a thread
ALU Instructions			
INT-ALU	5-bits *Op_code*	None	Integer ALU instruction
FP-ALU	5-bits *Op_code*	None	Floating-Point ALU instruction
MULTIPLY	5-bits *Op_code*	None	
DIVIDE	5-bits *Op_code*	None	

SYS_CALL	5-bits 10-bits *Op_code* Sys Call Number	None	System calls other than thread-related calls.

6.2.3 Loop Recognition

X86 architecture has multiple explicit loop instructions, namely, LOOP, LOOPE, LOOPNE, LOOPZ and LOOPNZ. These instructions are easily detected by the CET profiler and turned into CET loops. However, compilers often use the conditional branch instructions to translate loops. Therefore, there is a need to distinguish between the conditional branch instructions that implement loops and other conditional branches.

Loops represent the main venue for an execution trace compression. Moreover, detecting the X86 conditional branches that implement loops and translating them into CET loops will minimize the size of CET data significantly. For example, if all X86 conditional branches are left as they are, then a loop of one million iterations will require a storage of one million bits to store its branch's results (taken or not taken). However, with loop detection, this branch instruction is translated into a one CET loop instruction whose number of iterations is embedded into its body.

Conditional branches implementing loops are distinguished from other conditional branches using a two-phase algorithm. The first phase checks the loop candidacy, i.e., checks if a conditional branch can be a loop or not. The second phase occurs during the CET code emulation stage. In this phase, the loop candidates are filtered. If a loop candidate does not pass, it is switched back to a normal conditional branch. Thus, the second phase is a correction step.

As noted above, the CET profiler stores the branch's results of the conditional branch instruction in a list. However, this list is compressed such that similar consecutive results are stored in one node with a counter. The loop candidate will have a branch's results chain, as shown in Figure 8. Thus, an X86 conditional branch instruction is considered as a loop candidate if it has the following behavior:

1. All not-taken nodes have a counter of one.
2. The last node must be a not-taken node.
3. The first node can be either taken or not-taken depending on if the loop is outer or inner. Thus, the loop instruction has a flag bit to indicate if the first node is taken or not.

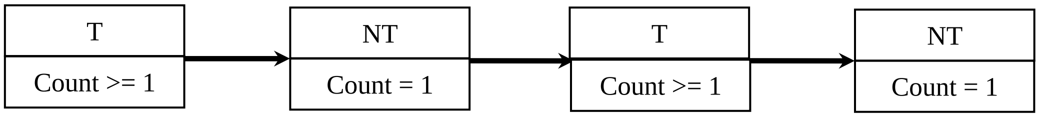


Figure 8: Branch Results' Chain of a Loop Candidate X86 Conditional Branch Instruction

This algorithm may consider some conditional branches as loops that were not intended to be loops, e.g., if a conditional branch is executed twice, being taken the first time and not taken the second time, then this algorithm considers it as a loop with one iteration. This behavior, however, is still correct.

In the emulation phase, the generated CET code is functionally executed by the CET emulator. The loop candidates are filtered in this phase. If a loop candidate does not pass, it is switched back to a conditional branch instruction. A stack is used to schedule the loops execution and to filter the loop candidates as follows:

1. Let ***S*** be a special stack of loop entries. In addition to its push and pop functions, ***S*** can be scanned and an element can be removed from the middle.
2. The loop entry is a structure with two fields: instruction address and counter.
3. When a loop or loop candidate instruction ***I*** is encountered, do the following:
    a. If ***I*** does not exist on ***S*,** push it.
    b. Else, if ***I*** is the top element of ***S*** and its counter is not zero, decrement the counter and branch.
    c. Else, if ***I*** is the top element of ***S*** and its counter is zero, don’t branch and pop ***S*** off.
    d. Else, if ***I*** exists on S and it is not the top element, ***I*** is not a loop; it is removed from ***S*** and switched back to a normal conditional branch.

6.2.4 CET Profiler

In this phase, the application’s trace is generated using functional simulations or native execution on the target machine itself. Using instrumentation, execution information regarding the instructions is collected, e.g., memory references accessed by the instruction in the case of load/store, branch results in the case of conditional branch etc. The profiling output is an intermediate representation of the input program, in which each instruction is represented as an object. This object contains all execution information regarding the instruction. The input program is profiled dynamically by instrumenting each instruction; when an instruction is encountered for the first time, a new object for this instruction is created and mapped to a unique location in the profiled image. If the same instruction address is encountered again later, its corresponding object is updated if required.

Figure 9 shows a flowchart of the CET profiler. It comprises the following steps:

1. While the program is not finished, do the following:
2. Let ***I*** = next instruction or routine.
3. Execute ***I***.
4. The analysis function corresponding to ***I*** is invoked.
5. If ***I*** does not exist in the profiled image, create a new object of ***I*** and add it to the image.
6. Check the opcode of ***I***:
    a. If it is a memory instruction, add the memory address to the list of addresses of ***I***. If the number of addresses added thus far is fifty (this number can be a

parameter), check if the instruction is load/store, load/store-c or load/store-nc and change its opcode accordingly. This early test accelerates compression and reduces space. This is because the profiler does not wait to store the whole addresses' list and then checks the memory instruction type.

b. If it is a conditional branch instruction, add the branch result to the branch results' list (Taken or not taken).

c. If it is an unconditional jump, call or return instruction, add its target address to the addresses' list.

d. If it is an explicit loop instruction, increment its counter.

e. If it is an ALU instruction, add the corresponding opcode, such as: INT-ALU, FP-ALU etc.

f. If it is a system, add SYS_CALL instruction.

g. If it is a synchronization function call, add the corresponding opcode, such as: START, PAUSE, WAKE etc.

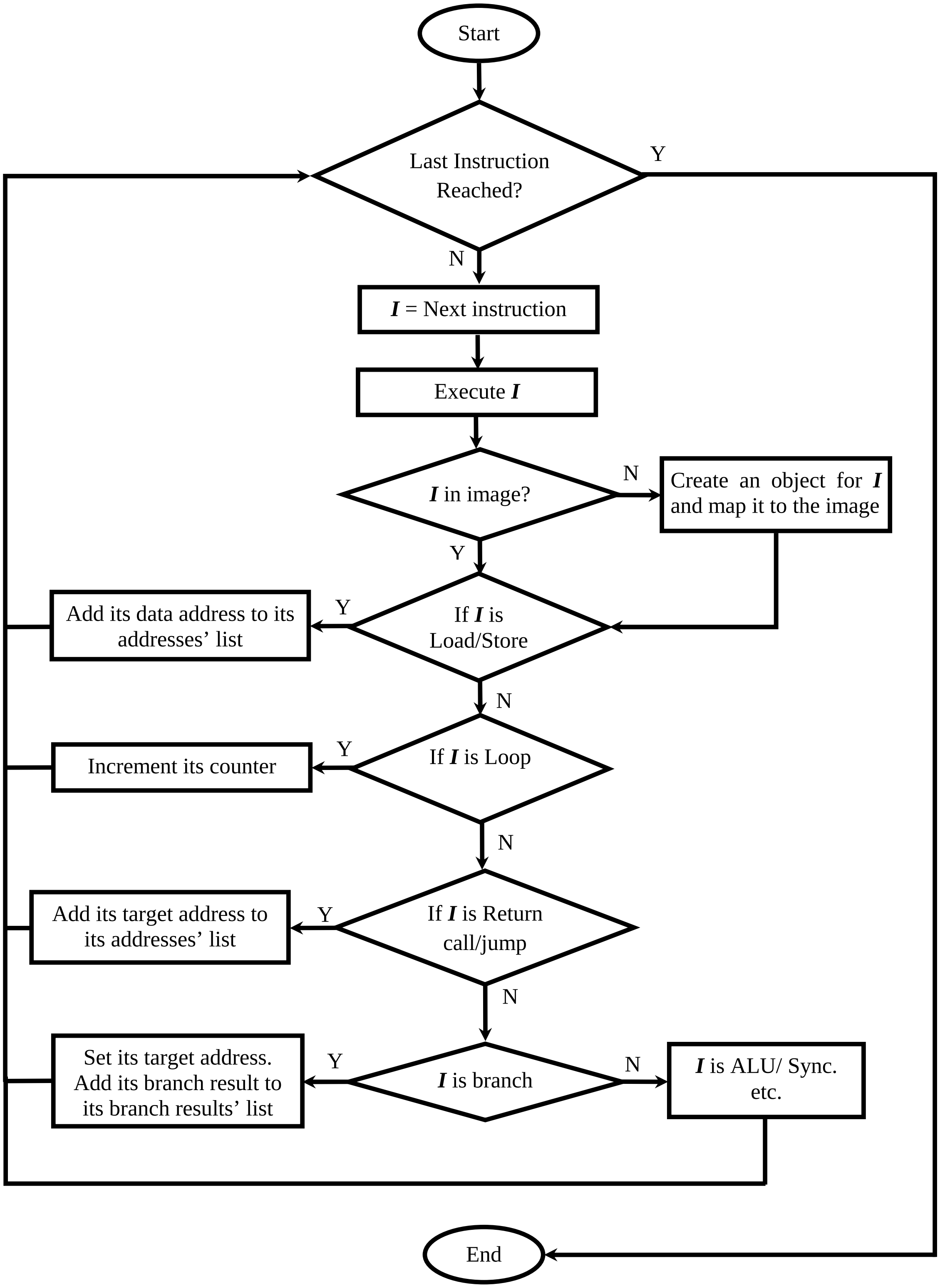

Start
Last Instruction Reached?
Y
N
I = Next instruction
Execute *I*
I in image?
N
Create an object for *I* and map it to the image
Y
If *I* is Load/Store
Y
Add its data address to its addresses' list
N
If *I* is Loop
Y
Increment its counter
N
If *I* is Return call/jump
Y
Add its target address to its addresses' list
N
I is branch
Y
Set its target address. Add its branch result to its branch results' list
N
I is ALU/ Sync. etc.
End

Figure 9: CET Profiler Flowchart

6.2.5 CET Code Generation

In this phase, the profiled image is refined and its instructions are replaced by the corresponding CET codes. Figure 10 shows the flowchart of the CET code generator with the following steps:

1. Let ***CetCode*** be a list of CET instructions.
2. For each instruction ***I*** in the profiled image, do the following:
3. If ***I*** is a loop:
    a. If ***I*** has a constant counter:

       `CetCode.add(LOOP displacement, counter)`

    b. Else, if ***I*** has multiple different counters that follow a certain pattern i.e., it is an inner loop whose number of iterations increases/decreases by a fixed value for each new outer loop iteration:

       `CetCode.add(LOOP-C displacement, counter, increment)`

    c. Else, if ***I*** has multiple different counters that do not follow a certain pattern:

       `CetCode.add(LOOP-R displacement)`

4. If ***I*** is load/store (this step is done earlier in the profiler when the number of addresses is 50 or above):
    a. If it has only one memory address or multiple similar addresses:

       `CetCode.add(LOAD/STORE address)`

    b. If it has multiple memory addresses and the difference between these addresses is constant:

       `CetCode.add(LOAD-C/STORE-C address, increment)`

   c. If it has multiple memory addresses and the difference between these addresses is not constant:

      `CetCode.add(LOAD-NC/STORE-NC)`

5. If ***I*** is an unconditional jump, return or call instruction:
   a. If it has only one target address:

      `CetCode.add(JUMP displacement)`

   b. If it has multiple target addresses:

      `CetCode.add(JUMP-M)`

6. If ***I*** is a conditional branch instruction:
   a. If it is always taken, `CetCode.add(JUMP displacement)`
   b. If it is always not taken, `CetCode.add(ALU-INT)`
   c. Else, `CetCode.add(BRANCH displacement)`
7. Otherwise, add ***I*** into *CetCode* as it is.
8. Dump *CetCode* into a text file in binary format.

The symbols in Figure 10 represent the following, ***I***: Instruction, ***Ai***: Address i, ***Ci***: Counter i, ***K***: Constant.

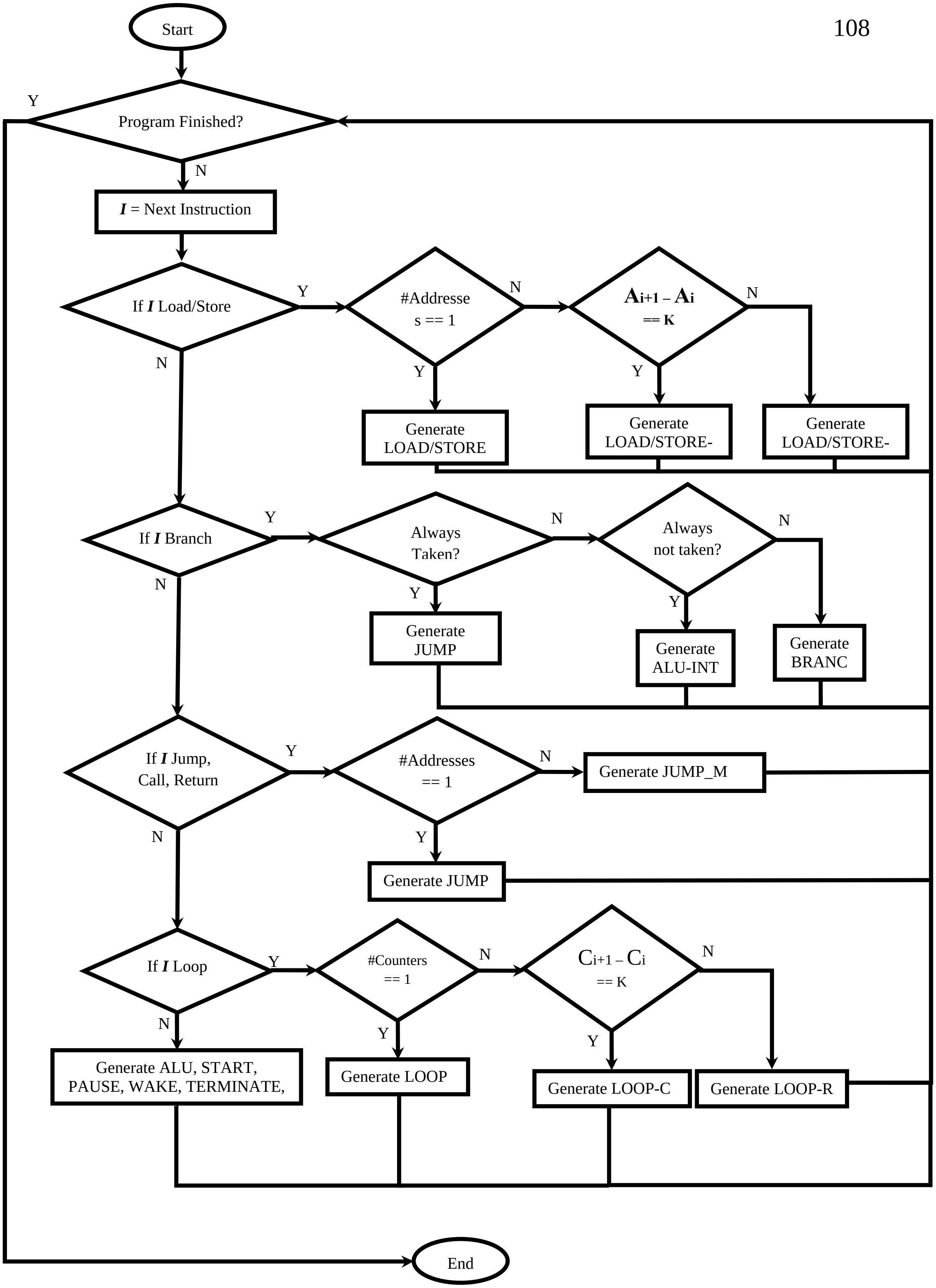
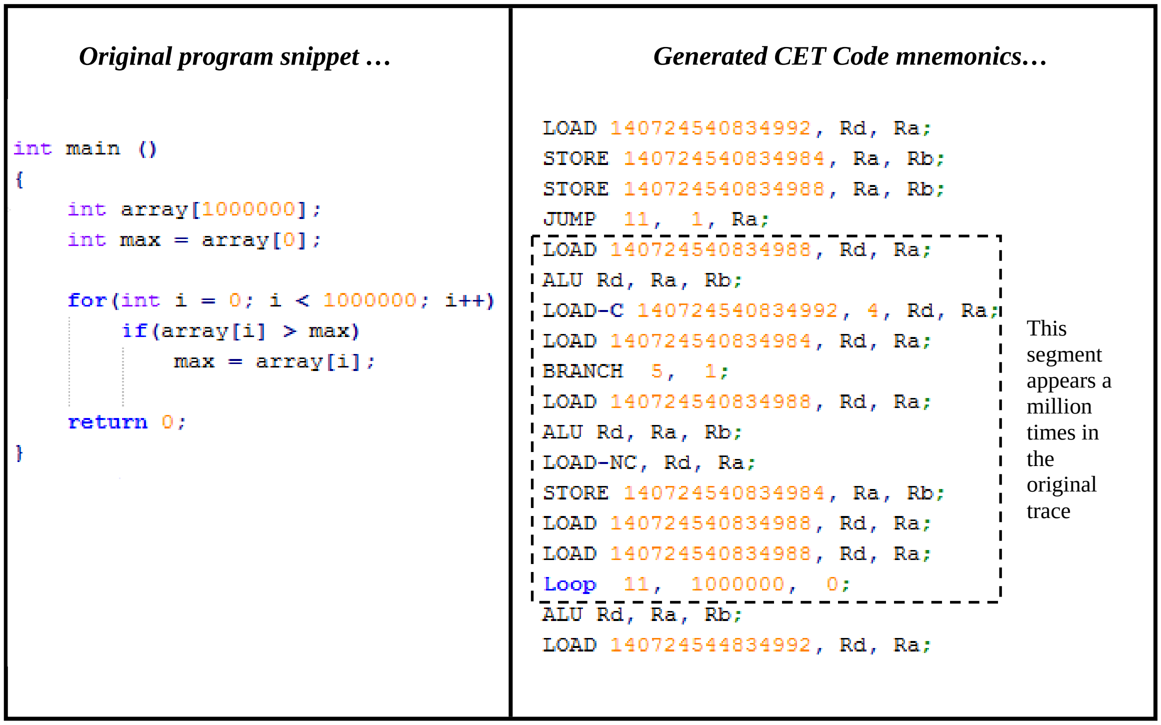


Figure 10: CET Code Generator Flowchart

Figure 11 below shows the generated compressed execution trace for a small C-code snippet (a loop to find the maximum of a 1 million integers array) to illustrate the power of the proposed trace compression methodology. For this simple example, the compression ratio is approximately 1 millionth (i.e., 0.000001).



Original program snippet …

```
int main ()
{
    int array[1000000];
    int max = array[0];

    for(int i = 0; i < 1000000; i++)
        if(array[i] > max)
            max = array[i];

    return 0;
}
```

Generated CET Code mnemonics…

```
LOAD 140724540834992, Rd, Ra;
STORE 140724540834984, Ra, Rb;
STORE 140724540834988, Ra, Rb;
JUMP  11,  1, Ra;
LOAD 140724540834988, Rd, Ra;
ALU Rd, Ra, Rb;
LOAD-C 140724540834992, 4, Rd, Ra;
LOAD 140724540834984, Rd, Ra;
BRANCH  5,  1;
LOAD 140724540834988, Rd, Ra;
ALU Rd, Ra, Rb;
LOAD-NC, Rd, Ra;
STORE 140724540834984, Ra, Rb;
LOAD 140724540834988, Rd, Ra;
LOAD 140724540834988, Rd, Ra;
Loop  11,  1000000,  0;
ALU Rd, Ra, Rb;
LOAD 140724544834992, Rd, Ra;
```

This segment appears a million times in the original trace

Figure 11: Compression results for a simple C-code snippet.

6.2.6 Emulation and CET Data Generation

The generated CET code is emulated in this phase. The main purpose of this phase is to generate the CET data in a file with proper sequential order (similar to a FIFO). In other words, when processing the CET code (e.g., via a timing simulator), data required by any CET instruction can be consumed from the CET data file sequentially in the proper order in which they are needed. The other purpose of this step is to test the correctness of the CET code and report any bugs if necessary.

The following is a brief description of the emulation and CET data generation phase:

1. Let ***CetFifo*** be the corresponding CET data FIFO (e.g., Addresses FIFO and branch results FIFO etc.).
2. Let pc = the initial address of the thread.
3. Let ***CetCode*** is the CET code memory.
4. While ***CetCode*** is not finished, do the following
5. Let ***I*** = ***CetCode***(pc)
6. If ***I*** is a loop candidate that did not pass, switch it to a conditional branch.
7. If ***I*** is any branch instruction (JUMP, JUMP_M, BRANCH, LOOP etc), pc = target address.
8. Else, pc = pc + 1
9. If ***I*** is LOAD-NC/STORE-NC:
    a. ***CetFifo***.enqueue(***I***.addresses.front).
    b. ***I***.addresses.dequeue.

```
10. Else, If I is BRANCH
        a. CetFifo.enqueue(I.BranchResults.front).
        b. I. BranchResults.dequeue.
11. Else, If I is JUMP_M
        a. CetFifo.enqueue(I. addresses.front).
        b. I. addresses.dequeue.
12. Else, If I is LOOP-R, and this is a new outer iteration:
        a. CetFifo.enqueue(I. counters.front).
        b. I. counters.dequeue.
13. Convert CETFifo to binary and output it.
```

6.2.7 System Calls Latency

As stated before, HySim timing model is a user-level model and hence it does not simulate the system-level code, except for threading management, although CET code encodes the system calls. However, we tried to quantify the approximate time consumed by different system calls through reading Linux system time right before and after the system call and taking the difference. We performed this experiment with the help of Intel Pin instrumentation tool. This experiment aimed at grouping the different system calls according to their latency and making this latency a tunable parameter. Unfortunately, we observed that the same system call can have a different latency within the same benchmark and across different benchmarks. This latency might be significant, i.e., it can be in orders of magnitude.

We used the Linux ***time*** command to quantify the amount of execution time of the application that is consumed by the system calls. A sample output of this command is as follows:

```
0:02.00 real, 0.00 user, 0.00 sys
```

This command outputs three values, (1) real, which is the time elapsed between the invocation and termination of the application, (2) user, the application time (user space time), and (3) sys, which is the time consumed by the system calls. We noticed that the system time of the application increases significantly by increasing the number of threads. This is a natural observation because more threads require more work (management and scheduling) from the operation system.

Figure 12 shows plots of the histogram of the system time for several numbers of threads, namely, 1,2,4,8, and 16. The X-axis splits the system time into intervals and the Y-axis shows the number (frequency) of benchmarks whose system time falls within this interval. Then, we calculated the average system time for each number of threads to be used by the timing model to compensate for the system time component.

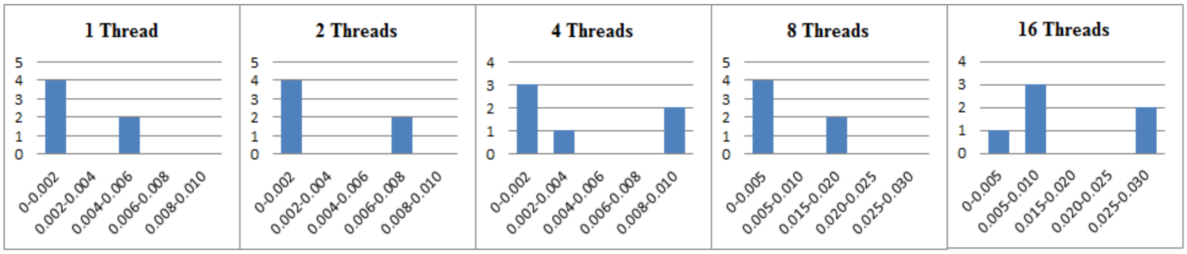


Figure 12: System Time Histogram

6.3 Experimental Results

6.3.1 Experimental Setup

We evaluated the CET tool using a wide range of benchmarks that includes a subset of Splash-2 [76], PARSEC [77], MediaBench I [78], and SECP CPU 2000 [79]. Table 7 lists the used benchmarks with their input sets. These experiments were run only once on an Intel Xeon CPU E5-2680 machine. The CET tool has been evaluated in two modes. (1) Instruction Addresses (IA) mode in which the baseline trace entry comprises the instruction along with its address, i.e., (32-bit instruction address, 32-bit instruction). (2) Full mode, in which the whole trace, instructions, instructions' addresses and data addresses (if any) are compressed, i.e., (32-bit instruction address, 32-bit instruction, [32-bit data address]).

Table 7: Benchmarks and Their Input Sets

Benchmark	Input Set
swaptions (small)	16 swaptions, 5,000 simulations
swaptions (medium)	32 swaptions, 10,000 simulations
swaptions (large)	64 swaptions, 20,000 simulations
Blackscholes (small)	4,096 options
Blackscholes (medium)	16,384 options
Blackscholes (large)	65,536 options
bodytrack (small)	4 cameras, 1 frame, 1,000 particles, 5 annealing layers
bodytrack (medium)	4 cameras, 2 frames, 2,000 particles, 5 annealing layers
bodytrack (large)	4 cameras, 4 frames, 4,000 particles, 5 annealing layers
LU	512×512 matrix
FFT	256K points
Ocean	258×258 ocean
Radix	256K integers
Water-sp	512 molecules
Water-nsq	512 molecules
cjpeg	input_base_4CIF.ppm

g721decoder	clinton.g721
g721encoder	clinton.g721.pcm
pegwit_d/e	Default
164.gzip, 179.art, 176.gcc, 181.mcf, 186.crafty, 300.twolf, 183.equake, 175.vpr, and 256.bzip2	The first two billion instructions of the reference input set.

We used two metrics to evaluate the CET tool. First, the compression ratio, this is the most important metric for evaluating a compression tool. It shows how many times the compressed trace is smaller than the uncompressed one. So it is calculated by dividing the size of the uncompressed trace over the size of the compressed one. The latter is the summation of the sizes of the CET code and CET data. In IA mode, each trace element (executed instruction) in the uncompressed trace is 64-bit (32 bits for the instruction address and 32 bits for the instruction itself) whereas it is 96-bit in the full mode; extra 32 bits are added to represent the data address, if any. The second metric is the compression and decompression speed, which is expressed in MIPS, i.e., how many millions of instructions of the execution trace can be compressed or uncompressed in one second. Although compression speed is required, this metric is less important than the compression ratio and decompression speed. Because the execution trace of a specific application is compressed only once and used many times.

6.4 Compression Ratio

Figure 13 shows the compression ratio for the two modes. In general, IA mode has a higher compression ratio than full mode, because IA mode ignores data memory references. Thus, the compressed trace in IA mode does not include data addresses, which are often the largest component of the compressed trace. However, full mode can achieve higher compression ratio when the application has few non-contiguous load/store addresses, such as ocean and blackscholes benchmarks. This is because the compressed trace is nearly the same for the two modes, but the uncompressed trace is larger in the full mode.

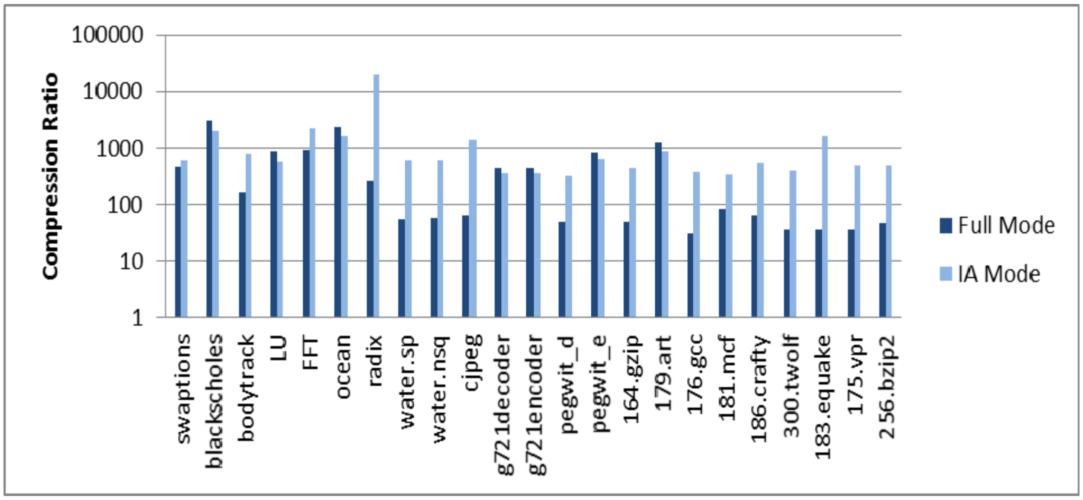


Figure 13: Compression Ratio of Instruction Addresses Only Traces (IA) and the Full Trace

Figure 14 shows the compression ratio versus different problem sizes (small, medium and large) of three different single-threaded benchmarks. From this figure, it is obvious that for the swaptions and blackscholes benchmarks, the compression ratio is nearly constant for the three aforementioned problem sizes. However, it decreases when the problem size is increased for the bodytrack benchmark.

Increasing the problem size increases the uncompressed trace size. However, the effect of increasing the problem size on the compressed trace size depends on the application structure, i.e., the distribution of the non-contiguous addresses or dynamic unconditional jumps across the application. Thus, if the compressed trace size increases in the same rate as the uncompressed one, the compression ratio is sustained. Otherwise, the compression ratio might increase or decrease due to increasing the problem size.

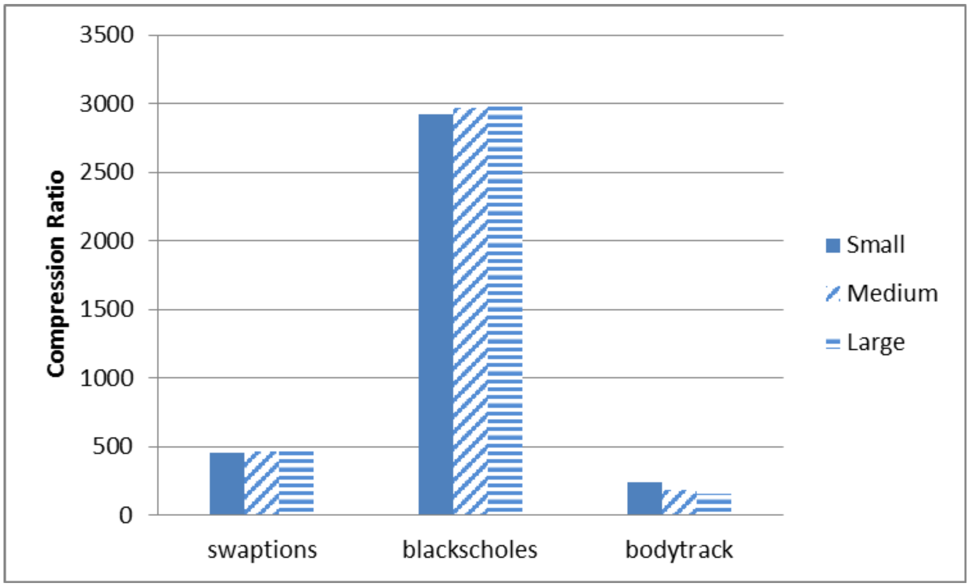


Figure 14: Compression Ratio vs Problem Size for 3 single-threaded benchmarks.

The compression ratio achieved by the CET tool varies according to the application’s structure, because it controls the content of the CET data. For example, large number of non-contiguous memory addresses, dynamic function calls, dynamic unconditional jumps, large number of conditional branches inside loop bodies etc. results in a larger CET data and therefore lower compression ratio, and vice versa is true.

Table 8 lists the compression ratio archived by the CET tool in the two modes for 23 single-threaded benchmarks. Moreover, it shows the compressed and uncompressed trace sizes. Our CET tool outperforms Ching-Wen Chen’s technique [33], which achieved a compression ratio between 16.67 and 50. Chen’s technique has the same baseline trace as our IA mode. This table shows that CET tool in IA mode achieved a better compression ratio than Chen’s technique by at least one order of magnitude. Moreover, in the full mode, the CET tool is still better by at least one order of magnitude for most of the benchmarks. CET tool does not have any case worse than Chen’s technique.

Our CET tool in IA mode outperforms Ching-Wen Chen’s technique because it handles the instruction addresses in a different manner. Their compressed trace contains a very long bit vector, one bit per instruction, to indicate whether the current instruction’s address is sequential or not. Furthermore, it included the differences among the non-sequential instruction addresses. On the other hand, our compressed trace captures the program flow control and hence when the CET code is executed the instruction addresses are regenerated on-the-fly.

Table 8: Uncompressed and Compressed Traces Size and Compression Ratio

Benchmark	**Uncompressed Trace Size (MB)**	**Compressed Trace Size (MB) (Full Mode)**	**Compressed Trace Size (MB) (IA Mode)**	**CET Compression Ratio (Full Mode)**	**CET Compression Ratio (IA Mode)**
swaptions	121298.2	262.96	131.40	461.2	615.4
Blackscholes	18721.4	6.27	6.27	2987.9	1992.0
bodytrack	153523.0	959.76	132.06	160.0	775.0
LU	5099.9	6.00	5.99	849.7	568.0
FFT	2486.6	2.64	0.76	940.1	2186.9
Ocean	5934.9	2.57	2.45	2304.8	1612.5
Radix	1066.1	4.05	0.04	263.5	20283.7
water.sp	3113.0	56.14	3.45	55.5	600.8
water.nsq	3525.4	61.70	3.98	57.1	589.9
cjpeg	580.0	8.82	0.28	65.6	1374.7
g721decoder	1649.2	3.78	3.10	436.1	354.2
g721encoder	5279.4	12.03	9.87	438.8	356.7
pegwit_d	106.5	3.78	3.10	49.8	22.9
pegwit_e	37.7	0.05	0.04	811.7	635.4
164.gzip	22888.2	463.32	33.97	49.4	449.3
179.art	22888.2	17.77	17.63	1288.1	865.3
176.gcc	22888.2	748.74	40.98	30.6	372.3
181.mcf	22888.2	272.70	44.42	83.9	343.5
186.crafty	22888.2	365.34	28.32	62.6	538.9

300.twolf	22888.2	649.48	37.40	35.2	408.0
183.equake	22888.2	636.22	9.18	36.0	1661.8
175.vpr	22888.2	634.85	30.91	36.1	493.7
256.bzip2	22888.2	499.72	31.82	45.8	479.5
Min	37.7	0.05	0.04	30.6	22.9
Max	153523.0	959.76	132.06	2987.9	20283.7
Average	22974.6	246.90	25.11	502.1	1633.9

Figure 15 compares the compression ratios achieved by the CET tool and the SBC (Stream-Based Compression) technique [80] for a subset of SPEC CPU2000 benchmarks. SBC uses a baseline trace whose entry is 38-bit whereas CET's baseline trace entry is 96-bit. This figure shows that in most cases both techniques achieved compression ratios within the same order of magnitude. For some cases SBC is better and for other cases CET is better. SBC compresses the trace in a different manner. It compresses both instruction and data addresses by associating them with an instruction stream and stores the stream identifiers, the data addresses strides, and their number of receptions in the compressed trace. The stream identifier includes the starting address of the stream and the stream length.

SBC tends to have a better compression ratio than the CET technique because it has a variable stride length that ranges from zero to eight bytes. This variable stride length saves storage significantly because the compressed trace will be very tight. However, this variable stride length does not work for FPGAs because in FPGA the data have to be aligned in order ensure quick access.

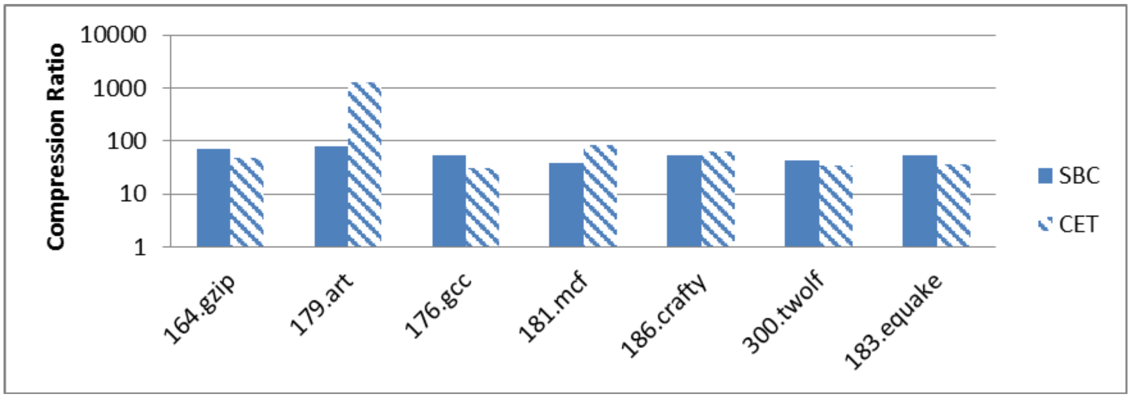


Figure 15: Compression Ratio Comparison between the CET Tool and SBC Technique

Moreover, SBC targets a specific simple trace type, namely, memory reference only, whereas the CET tool target a more detailed trace. On the other hand, the CET tool is much faster the SBC technique, i.e., it has a lower compression and decompression time. Figure 16 and Figure 17 shows the compression and decompression time for the CET and SBC techniques. From these figures, we notice that CET is faster than SBC by orders of magnitude. This is because CET compresses the trace on-the-fly, i.e., it profiles the application and retrieves the required CET data. One of the most time consuming-actions in CET compression is to check whether the addresses of a certain load/store are contiguous. However, this step has been accelerated by checking a small fraction of these addresses which is enough. Regarding decompression, the simple compressed trace structure generated by the CET tool made the decompression stage very efficient. It just implies executing the CET code and once a CET datum is required, it will be ready on the front of the corresponding FIFO, i.e., decompression does not imply complex decoding steps.

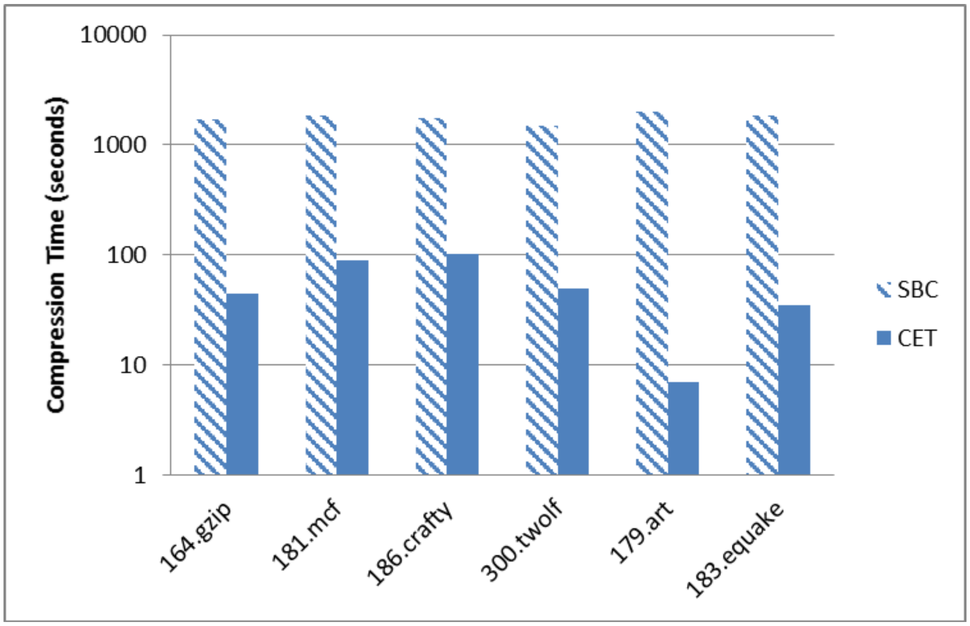


Figure 16: Compression Time for CET and SBC Techniques

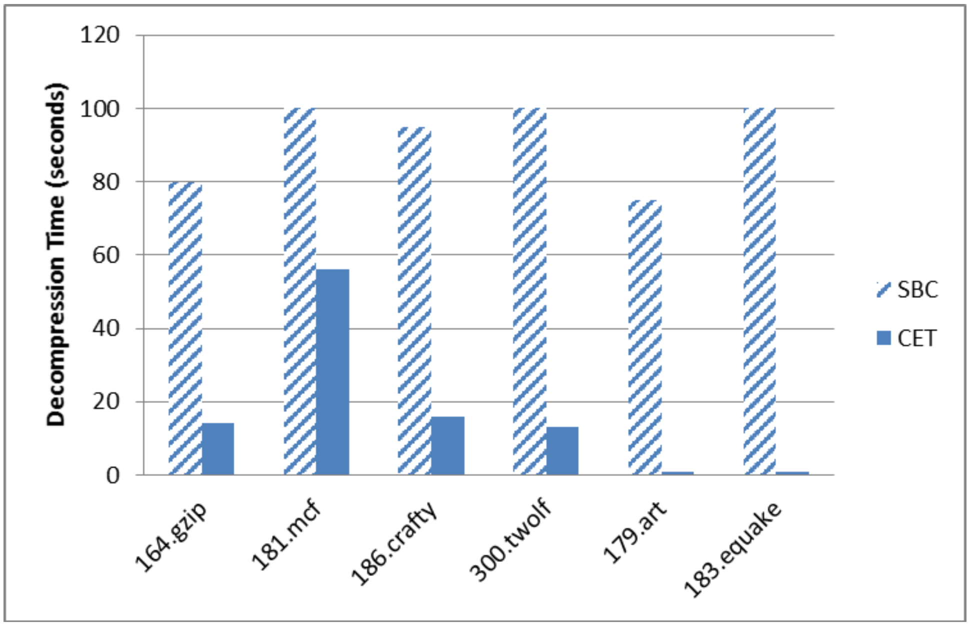


Figure 17: Decompression Time for CET and SBC Techniques

Figure 18 shows the full mode compression ratio of nine benchmarks for different number of threads, namely, 1, 2, 4, 8 and 16 threads. In this experiment, the total uncompressed and compressed traces’ sizes are the summations of the uncompressed and compressed traces’ sizes of all threads, respectively. In most cases, the compression ratio remains nearly constant as the number of threads increases. Because the application is distributed on the available threads, the total uncompressed and compressed traces’ sizes do not change markedly. However, the compression ratio decreases for the ocean benchmark. This variation is due to the variation of the CET data size, especially the number of non-contiguous addresses, when the number of threads changes.

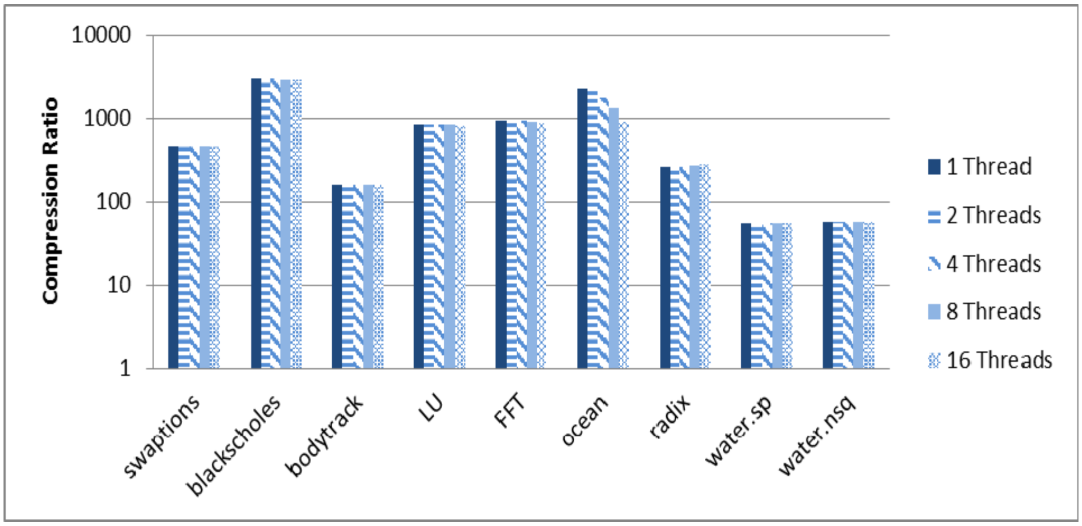


Figure 18: Compression Ratio vs Number of Threads

6.5 Compression/Decompression Speed

Table 9 shows the trace compression speed achieved by the CET tool. The maximum compression speed is **789.1 MIPS** in the case of **Bodytrack** benchmark, whereas the average speed is **186.4MIPS**. Also this table shows that decompression much faster than compression. This is natural because decompression just implies executing the CET code. The compression speed depends on the benchmark’s structure, for example, the longer the loop’s chains and addresses’ lists the slower the compression. This is because CET tool will take more time to process such data.

Table 9: Compression/Decompression Speed (MIPS)

Benchmark	Compression Speed	Decompression Speed
Swaptions	623.5	10599.2
blackscholes	62.9	65.4
Bodytrack	789.1	1219.5
164.gzip	44.4	142.9
179.art	285.7	2000.0
181.mcf	22.2	35.7
186.crafty	19.4	125.0
300.twolf	40.8	153.8
183.equake	57.1	2000.0
175.vpr	46.5	333.3
256.bzip2	58.8	117.6
Min	19.4	35.7
Max	789.1	10599.2
Average	186.4	1526.6

CHAPTER 7

HYSIM TIMING MODEL

This chapter delves into the architecture and implementation issues of HySim timing model. First, it presents the implementation technology we adopted to develop this model, namely, Bluespec SystemVerilog (BSV) technology. Then, it explains the timing model architecture and how the target machine performance is evaluated via this model.

7.1 Bluespec SystemVerilog (BSV)

We adopted BSV [63, 81, 82] to implement HySim FPGA-based timing model. It is a modern, fully synthesizable language developed at MIT. BSV is a high level hardware description language used in the design of electronic systems (ASICs, FPGAs and systems). In BSV, the design behavior is expressed with Guarded Atomic Actions (rewrite rules). BSV code is translated to Verilog via the BSC compiler. BSV allows the hardware designer to focus on the overall architecture and leave the details to the compiler which is designed and maintained by the RTL designers. Thus, BSV code is more on the architecture level rather than on the RTL level. BSV was adopted to implement many of the major FPGA-accelerated simulators, such as PROTOFLEX [10], HAsim [53], FAST [47], and Arete [9].

BSV has a modular nature that allows designing the architecture as a set of modules that are eventually turned into actual hardware. Each module can instantiate other modules forming a module hierarchy, which simplifies the large and complex systems. All BSV code should be organized into packages which are like namespaces. The BSV compiler assumes that there is one package per file and the file name should be <package name>.bsv. Each BSV module consists of zero or more sub-modules, rules to operate on the sub-modules, and an interface to the surrounding hierarchy. The interface comprises a set of methods to drive the signals and buses in and out the module.

A BSV rule basically consists of the rule condition and the rule body. The rule condition is pure combinational logic. It evaluates to a single Boolean value. The rule can fire only if this entry condition is true. The rule body consists of a set of actions that operate on the state elements and it is also pure combinational logic.

7.1.1 BSV Coding Productivity

The level of abstraction in BSV makes the size of BSV code smaller than its Verilog counterpart. Therefore, coding in BSV is more productive than coding in Verilog because a shorter code will be written and hence fewer bugs appear. Table 10 lists the BSV static code size and its corresponding auto-generated static Verilog code size measured in the number of lines of code for all HySim’s timing model modules. The number of code lines includes spaces and comments. This table shows that the BSV code is smaller than the corresponding Verilog code for all modules. The BSV code is ***3.34*** times smaller than the Verilog code for the overall design.

Table 10: Comparison between the BSV Code Side and the Corresponding Auto-generated Verilog Code Size

Module Name	**BSV Code Size (lines of code)**	**Auto-generated Verilog Code Size (lines of code)**	**Verilog to BSV Code Size Ratio**
Multi-core top module	268	1752	6.54
Tile top module	381	1602	4.20
Core	1456	5873	4.03
CET I-cache	145	436	3.01
CET D-cache	118	430	3.64
L1 D-cache model	406	953	2.35
L1 I-cache model	245	494	2.02
L2 cache model	613	1327	2.16
L3 cache model	749	1515	2.02
Router	314	1285	4.09
Total	4695	15667	3.34

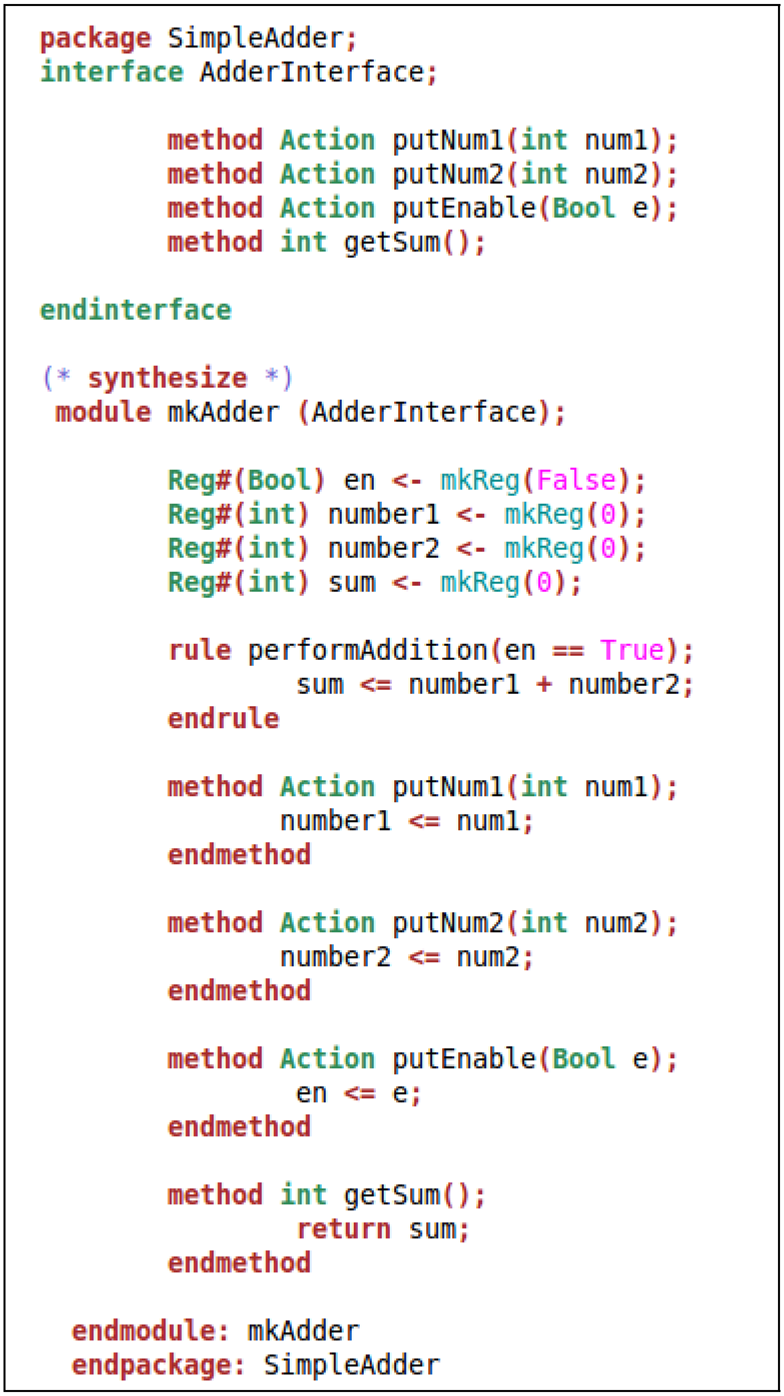
7.1.2 BSV to Verilog Compilation

The BSC compiler translates the BSV code to Verilog as follows:

1. Interface methods are mapped to port lists in the generated Verilog code in a straightforward manner.
2. CLK and RST_N input signals are added to the generated Verilog code's port list.
3. For each input port, enable and ready signals are added to the generated Verilog code's port list.
4. For each output port, ready signal is added to the generated Verilog code's port list.
5. State elements are mapped to the generated Verilog code exactly as they are in the BSV source. There is no state elements inference during BSV compilation.
6. Each module in the generated Verilog code has a corresponding module in the BSV source. Module hierarchy is directly recognizable from the BSV code.
7. Each rule has a control path comprises **CAN_FIRE** and **WILL_FIRE** signals in the generated Verilog. CAN_FIRE signal is the output of the rule condition and it indicates whether the rule can fire at this clock cycle. On the other hand, WILL_FIRE signal is the scheduled version of the signal, i.e., when WILL_FIRE is true, then the rule will certainly fire at that clock cycle.
8. The combinational logic in the rule condition and the rule body appears in the generated Verilog code as it is in the BSV source except some logic optimizations.
9. The BSC compiler adds scheduler logic and data path (multiplexers) when more than one rule is competing for the same sub-module/state element.

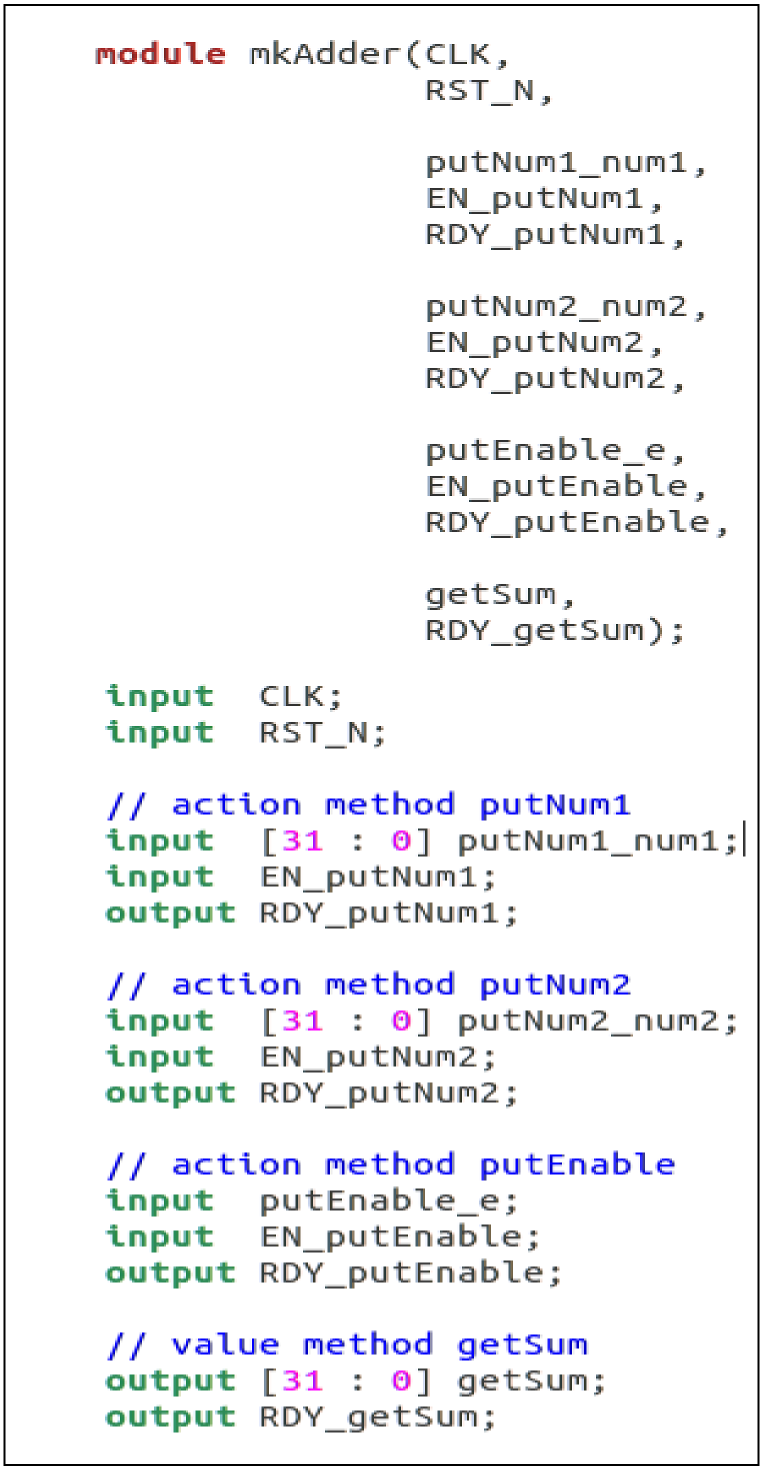
Figure 19 shows a simple BSV example to illustrate the BSV code structure and how it is translated to Verilog. The example is a simple adder circuit; it receives two integers, namely, num1 and num2; stores them into registers, adds them, and finally outputs the result. This example shows how the I/O ports are implemented via methods, and how the internal registers are instantiated. Moreover, it shows the rule ***performAddition***. To fire this rule, both the implicit and explicit conditions should be satisfied. This rule has one explicit condition, namely, the enable signal. Furthermore, it has some implicit conditions related to the readiness of the registers' values.

As mentioned before, the BSV interface methods are translated into the Verilog module's port list. Besides that, CLK, RST_N, enable, and ready signals are added to this port list. Figure 20 shows the auto-generated Verilog code for this simple adder circuit interface.



```
package SimpleAdder;
interface AdderInterface;

        method Action putNum1(int num1);
        method Action putNum2(int num2);
        method Action putEnable(Bool e);
        method int getSum();

endinterface

(* synthesize *)
 module mkAdder (AdderInterface);

        Reg#(Bool) en <- mkReg(False);
        Reg#(int) number1 <- mkReg(0);
        Reg#(int) number2 <- mkReg(0);
        Reg#(int) sum <- mkReg(0);

        rule performAddition(en == True);
                sum <= number1 + number2;
        endrule

        method Action putNum1(int num1);
               number1 <= num1;
        endmethod

        method Action putNum2(int num2);
               number2 <= num2;
        endmethod

        method Action putEnable(Bool e);
                en <= e;
        endmethod

        method int getSum();
                return sum;
        endmethod

  endmodule: mkAdder
  endpackage: SimpleAdder
```

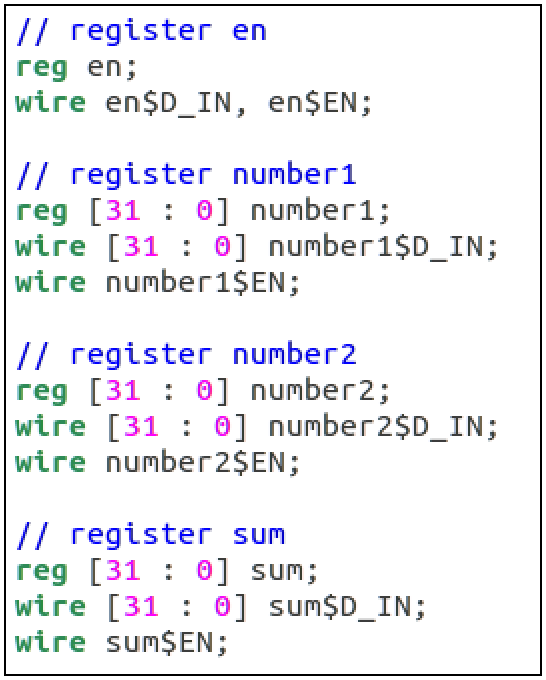
Figure 19: A Simple Adder BSV Code



```
module mkAdder(CLK,
               RST_N,

               putNum1_num1,
               EN_putNum1,
               RDY_putNum1,

               putNum2_num2,
               EN_putNum2,
               RDY_putNum2,

               putEnable_e,
               EN_putEnable,
               RDY_putEnable,

               getSum,
               RDY_getSum);

input  CLK;
input  RST_N;

// action method putNum1
input  [31 : 0] putNum1_num1;
input  EN_putNum1;
output RDY_putNum1;

// action method putNum2
input  [31 : 0] putNum2_num2;
input  EN_putNum2;
output RDY_putNum2;

// action method putEnable
input  putEnable_e;
input  EN_putEnable;
output RDY_putEnable;

// value method getSum
output [31 : 0] getSum;
output RDY_getSum;
```

Figure 20: The Auto Generated Verilog Code of the Simple Adder Interface

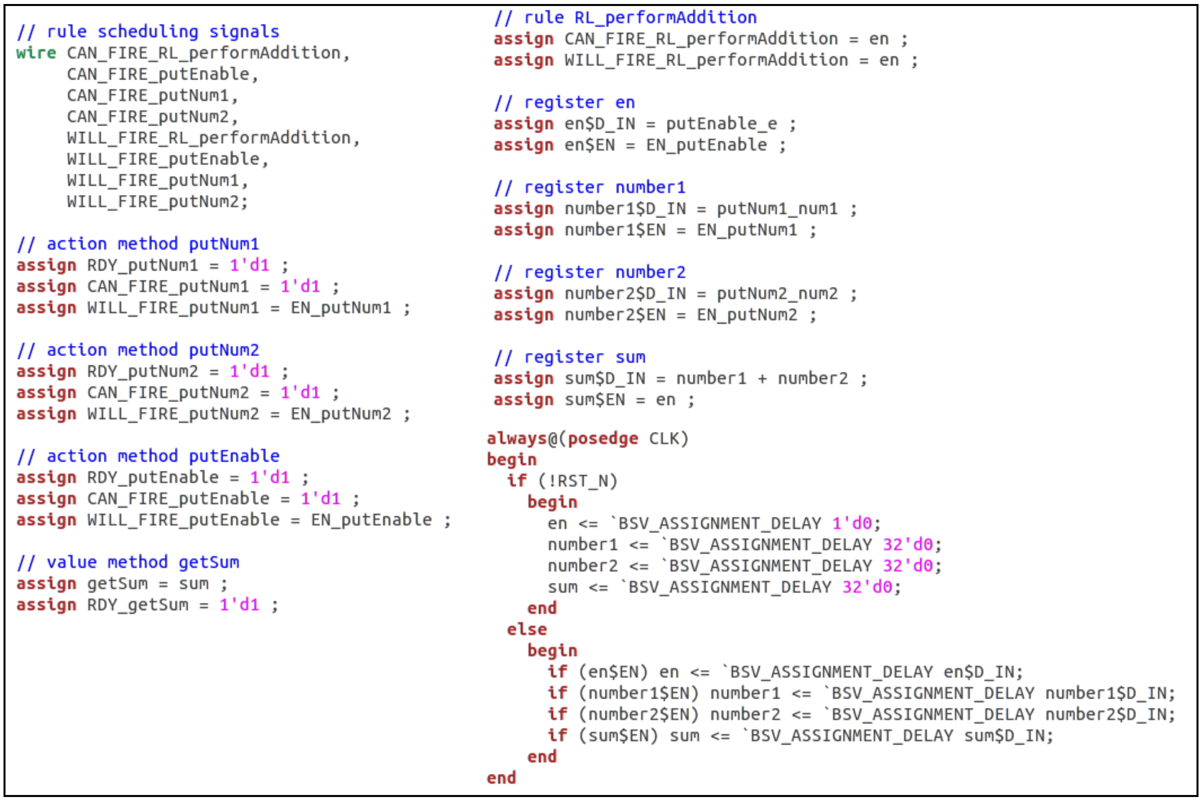
Figure 21 shows how the state elements (registers) are translated to Verilog directly.



```
// register en
reg en;
wire en$D_IN, en$EN;

// register number1
reg [31 : 0] number1;
wire [31 : 0] number1$D_IN;
wire number1$EN;

// register number2
reg [31 : 0] number2;
wire [31 : 0] number2$D_IN;
wire number2$EN;

// register sum
reg [31 : 0] sum;
wire [31 : 0] sum$D_IN;
wire sum$EN;
```

Figure 21: The Auto Generated Verilog Code of the Simple Adder Registers

Figure 22 shows how the rule condition and body’s combinational logic is translated to Verilog.



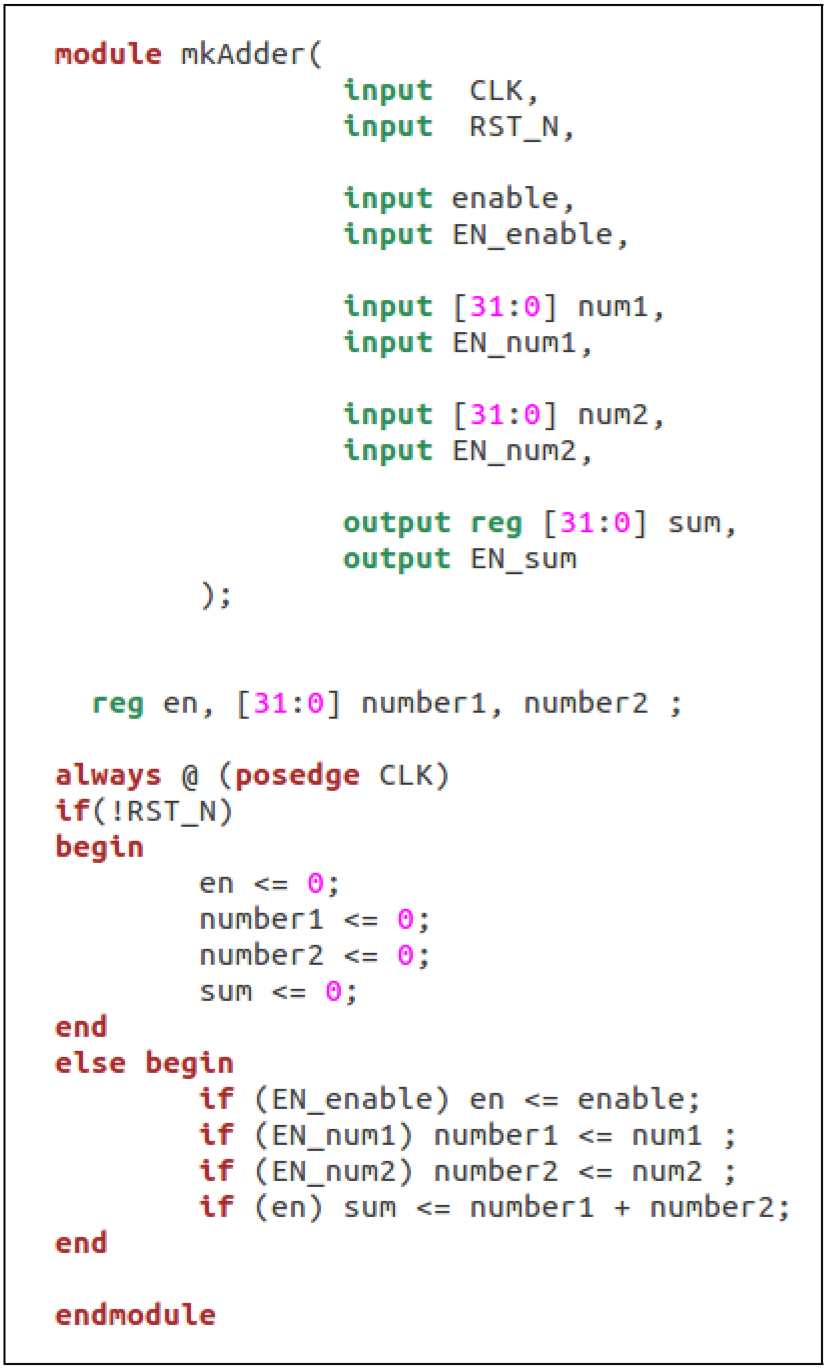
```
// rule scheduling signals
wire CAN_FIRE_RL_performAddition,
     CAN_FIRE_putEnable,
     CAN_FIRE_putNum1,
     CAN_FIRE_putNum2,
     WILL_FIRE_RL_performAddition,
     WILL_FIRE_putEnable,
     WILL_FIRE_putNum1,
     WILL_FIRE_putNum2;

// action method putNum1
assign RDY_putNum1 = 1'd1 ;
assign CAN_FIRE_putNum1 = 1'd1 ;
assign WILL_FIRE_putNum1 = EN_putNum1 ;

// action method putNum2
assign RDY_putNum2 = 1'd1 ;
assign CAN_FIRE_putNum2 = 1'd1 ;
assign WILL_FIRE_putNum2 = EN_putNum2 ;

// action method putEnable
assign RDY_putEnable = 1'd1 ;
assign CAN_FIRE_putEnable = 1'd1 ;
assign WILL_FIRE_putEnable = EN_putEnable ;

// value method getSum
assign getSum = sum ;
assign RDY_getSum = 1'd1 ;
```

```
// rule RL_performAddition
assign CAN_FIRE_RL_performAddition = en ;
assign WILL_FIRE_RL_performAddition = en ;

// register en
assign en$D_IN = putEnable_e ;
assign en$EN = EN_putEnable ;

// register number1
assign number1$D_IN = putNum1_num1 ;
assign number1$EN = EN_putNum1 ;

// register number2
assign number2$D_IN = putNum2_num2 ;
assign number2$EN = EN_putNum2 ;

// register sum
assign sum$D_IN = number1 + number2 ;
assign sum$EN = en ;

always@(posedge CLK)
begin
  if (!RST_N)
    begin
      en <= `BSV_ASSIGNMENT_DELAY 1'd0;
      number1 <= `BSV_ASSIGNMENT_DELAY 32'd0;
      number2 <= `BSV_ASSIGNMENT_DELAY 32'd0;
      sum <= `BSV_ASSIGNMENT_DELAY 32'd0;
    end
  else
    begin
      if (en$EN) en <= `BSV_ASSIGNMENT_DELAY en$D_IN;
      if (number1$EN) number1 <= `BSV_ASSIGNMENT_DELAY number1$D_IN;
      if (number2$EN) number2 <= `BSV_ASSIGNMENT_DELAY number2$D_IN;
      if (sum$EN) sum <= `BSV_ASSIGNMENT_DELAY sum$D_IN;
    end
end
```

Figure 22: The Auto Generated Verilog Code of the Simple Adder Rule Scheduling and Execution

The above figures show the price of BSV coding simplicity, namely the huge size of Verilog output even for such a small and simple example. Most of the output Verilog code is simply wire assignment to other wires or constants. The FPGA logic synthesis tool however, takes care of that. The code is further optimized to produce minimum HW on the FPGA (through common sub-expression extraction, constants propagation, wire renaming, and so on).

To demonstrate that the long auto-generated Verilog code eventually consumes the same hardware resources and generates the same hardware modules as the manually written counterpart, we manually wrote the Verilog code for this circuit that is shown in in Figure 23. Then, we synthesized both codes (the manually written and the auto generated) via Xilinx synthesis tool (XST) tool after setting Xilinx to optimize the design area. Table 11 shows the amount of FPGA resources consumed, and generated hardware modules for the two Verilog codes. As this table shows, XST inferred exactly the same hardware from these two codes and hence consumed the same FPGA resources.



```
module mkAdder(
                input  CLK,
                input  RST_N,

                input enable,
                input EN_enable,

                input [31:0] num1,
                input EN_num1,

                input [31:0] num2,
                input EN_num2,

                output reg [31:0] sum,
                output EN_sum
        );


  reg en, [31:0] number1, number2 ;

always @ (posedge CLK)
if(!RST_N)
begin
        en <= 0;
        number1 <= 0;
        number2 <= 0;
        sum <= 0;
end
else begin
        if (EN_enable) en <= enable;
        if (EN_num1) number1 <= num1 ;
        if (EN_num2) number2 <= num2 ;
        if (en) sum <= number1 + number2;
end

endmodule
```

Figure 23 : Simple Adder Manually Written Verilog Code

Table 11: FPGA Resources and Inferred Components are Identical for Both Manually Written and Auto Generated Verilog Codes

Component	**Count, auto generated Verilog**	**Count, manually written Verilog**
Number of inferred adders	1 32-bit adder	1 32-bit adder
Number inferred of flip-flops	97	97
Number of slice LUTs	33	33

7.2 Timing Model Architecture

HySim's timing model is an FPGA-based processor-like model. It receives the benchmark or application, in the compressed trace format (CET code + CET data), from the software frontend and stores it in an external SDRAM memory on the FPGA board. Then, it interprets and executes the CET codes to perform timing simulation. HySim's timing model can be configured to capture the timing characteristics of shared-memory multicore target architecture. Since the functional part has already been offloaded to a standard PC (the benchmark or application is natively executed), the timing model does not have functional units, such as, ALUs and floating-point units. Moreover, it does not need to store the input set of the benchmark. This significantly alleviates the hardware resources required to implement such model.

HySim's timing model decouples the target's clock (the clock of the multicore system being simulated) from the host clock (FPGA clock). Hence, a number of target cycles can be simulated in a different number of host cycles (that could be more or less). This decoupling helps in minimizing both the simulation time and the hardware area of the timing model. For example, an operation may take one target cycle can be simulated in multiple host cycles, but with less hardware resources. On the other hand, an operation may take several target cycles can be simulated in only one host cycle, which reduces the simulation time.

The timing model has a tiled architecture and can be comprised of any number of tiles as long as they can be hosted by the available FPGA resources. These tiles are interconnected via a ring interconnection network. Ring topology was selected for HySim timing model because it is simple to implement, consumes minimal resources and more tiles can be easily added by simply inserting them in the ring. Each tile models a target machine's processing core, a fraction of the memory subsystem, and a NoC router. Moreover, each tile contains special caches to cache CET code and data.

Figure 24 shows a top level logical view of HySim's timing model. Tile 0 contains the master core which executes the master thread that contains the sequential and parallel regions of the benchmark or application. The remaining tiles contain the worker cores which are responsible for executing the worker threads, i.e., the parallel regions. The timing model is able to simulate a target multicore machine with a number of cores less than or equals to those in the timing model itself without a need for time multiplexing. Figure 24 shows an abstract view of the HySim's timing model.

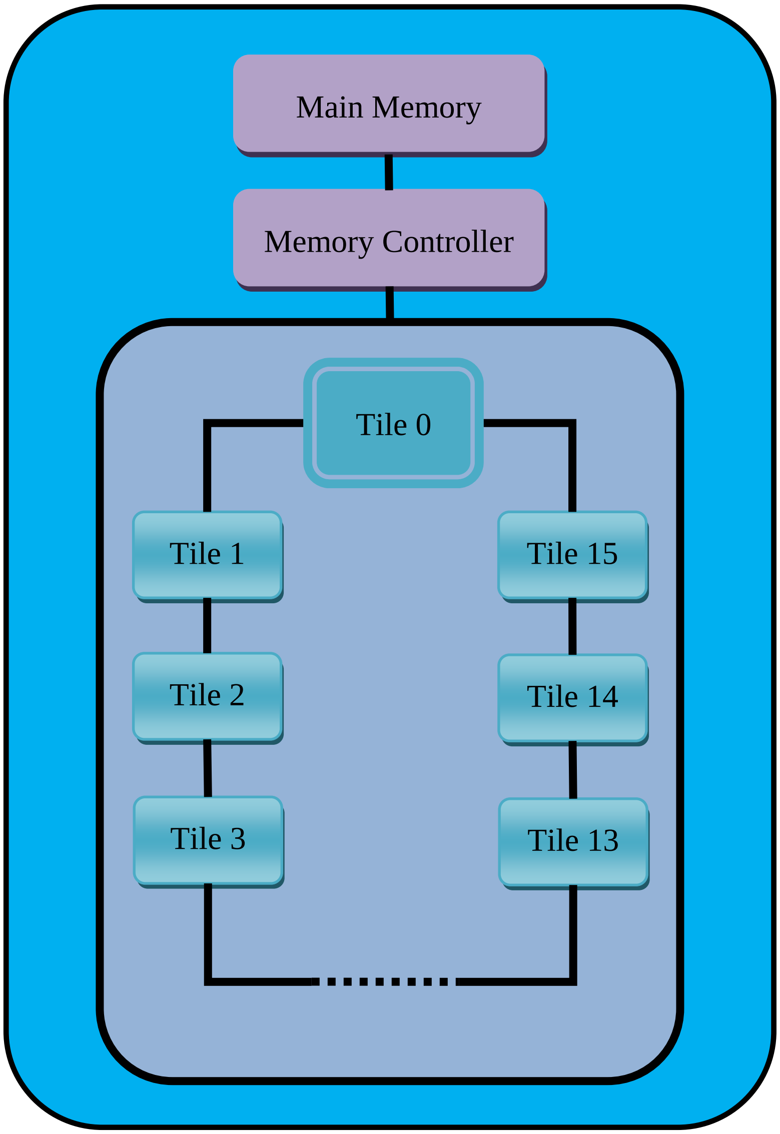


Figure 24: A Top Level Logical View of HySim's Timing Model

7.3 Timing Model's Tile Architecture

As shown in Figure 25, HySim's tile comprises a core model, CET code and data caches, target architecture's instruction and data cache models, and the NoC router. This section details these components.

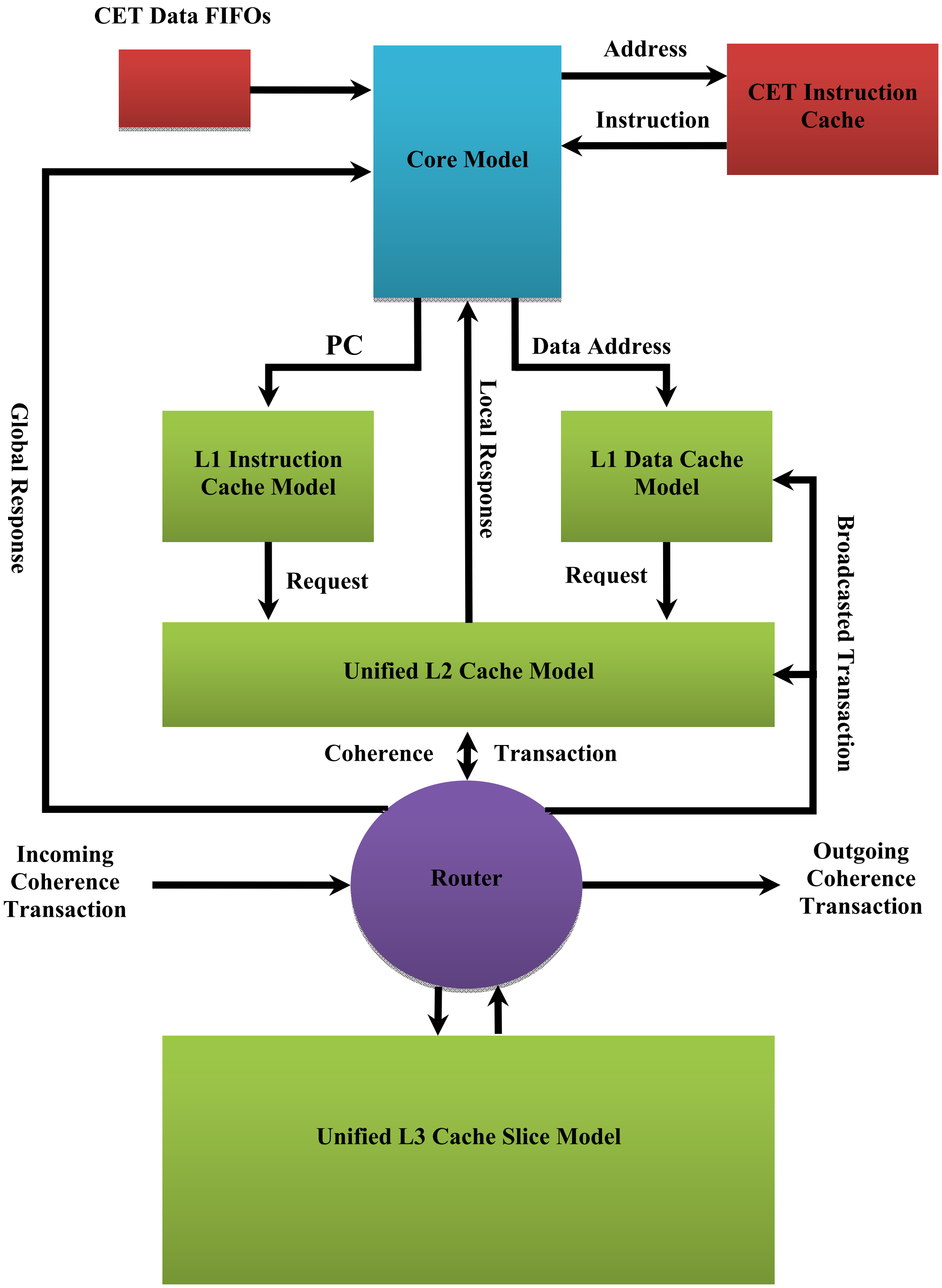


Figure 25: HySim's Timing Model's Tile Overview

7.3.1 HySim Core Model (CET Core)

As stated before, HySim currently focuses on the “uncore” features of the multicore architectures. Thus, CET core is an abstract core model of the target core. The target “in-core” timing is abstracted via the base CPI. CET core executes the CET code in order to evaluate the performance of the target machine. It has an execution pipeline of three stages**: fetch, decode and execute**. The fetch stage retrieves the next instruction from the CET code cache into the core’s instruction queue. If the CET instruction is not found in the CET instruction cache, the whole timing model stalls until this miss is resolved.

Execution in the CET core context is different from the normal known execution. In CET core, execution means an on-the-fly decompression of the compressed execution trace of the application, and taking the appropriate actions for each dynamic instruction to predict the execution time of this application on the target machine. Therefore, CET core has to be equipped with the necessary logic for fetching and decoding CET instructions. Moreover, it should contain the architectural parameters registers (to store the values of the target architecture parameters, e.g., base CPI and cache access latencies), the different performance counters (registers), and the necessary logic required to interact with these registers.

Figure 26 shows an abstract schematic view of the CET core. It contains the required control and data paths to execute the CET code and evaluate the expected execution time. Also, it contains a unit to schedule the execution of the loop nests of the CET code. This loop scheduling unit will be detailed in the next subsection.

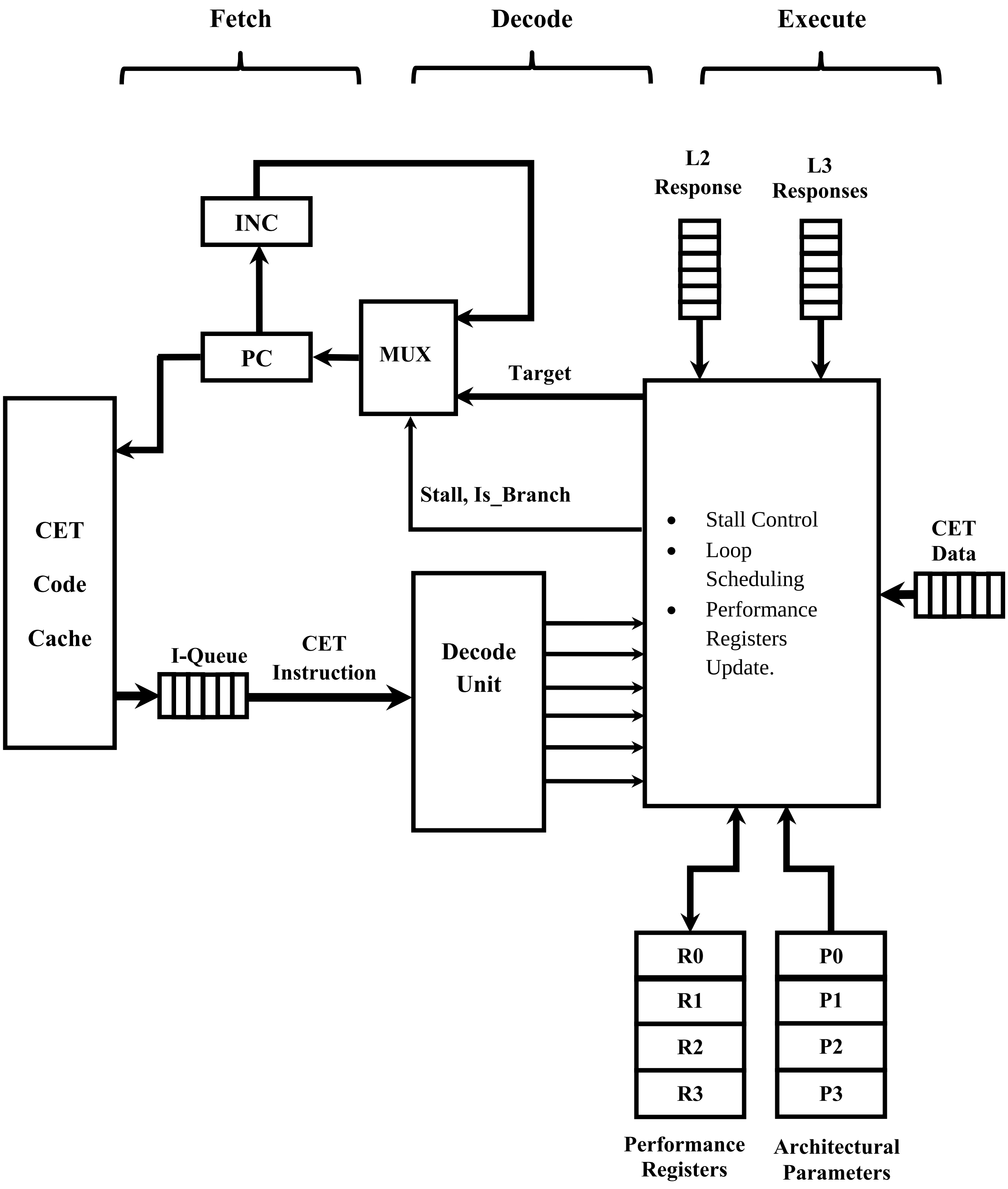


Figure 26: CET Core Abstract Schematic

In order to execute the CET code and derive the target execution time, CET core takes the appropriate actions for each dynamic instruction. These actions include updating the performance counters, sending memory requests in the case of load/store to the target cache hierarchy model, updating the program counter (PC) in the case of control instructions, etc. Table 12 summarizes the different actions taken by the CET core for different instructions. Loop instructions scheduling is explained in the next subsection.

Regarding target execution time derivation, each target processor core has a clock. The application starts with the master thread, whose initial target clock is **zero**. Each time a **`START`** instruction is encountered by the master thread, the next inactive CET core is activated to simulate the new thread. The initial clock of this target core is set to the current target clock of the master thread. For each barrier, (N-1) threads have the instruction **`WAIT`** to wait on this barrier, where N is the total number of threads. However, only one thread, namely the slowest one, has the instruction **`WAKE`** for that barrier.

When the **`wake`** instruction is encountered by a thread, it sends a wake signal to the other threads. This wake signal is a packet that contains the current target clock value of the CET core running this thread. When this packet is received by a thread, it computes the difference between the local target clock value and the value in the packet. This difference is the waiting time of this thread on the synchronization barrier, and it is accumulated to the local target clock. Then this thread is resumed. Thus, the tiles’ local target clocks are synchronized on the synchronization barriers.

For each instruction, the CET core adds the base CPI to the target clock. This is enough if the instruction did not result in miss events, e.g., D-cache or I-cache miss. However, the CET core sends the address of each instruction to L1 I-cache to check whether there is an instruction cache miss. Moreover, when a load or store instruction is encountered, a cache coherence transaction is packed and sent to the L1 D-cache to check whether there is a data cache miss. This transaction packet contains the initiator thread ID, time stamp, the memory reference, the operation type (read or write), and a field to store the number of navigated hops.

After this transaction navigates through the memory hierarchy, it comes back to the initiator core. Then the initiator core updates the target clock and performance registers based on the transaction result and the target architecture parameters after taking into account the overlapping between the timing of independent miss events. More details on memory hierarchy navigation are provided later in this chapter.

Table 12: Actions Taken by CET Core for Different CET Instructions

CET Opcode	Action(s)
ALU, Sys-Call	Just increment the PC (Program Counter).
JUMP	Add the displacement value (embedded into the CET instruction body) to the current PC.
JUMP_M	1. Read the displacement value, which is the front of displacements CET data FIFO. 2. Dequeue the displacements FIFO. 3. Add this displacement value to the current PC.
BRANCH	1. Read the branch result, which is the front of branches CET data FIFO. 2. Dequeue the branches FIFO. 3. If the branch result is taken, add the displacement value (embedded into the CET instruction body) to the current PC. 4. If the branch result is not taken, just increment the PC.
LOOP	1. If this loop instruction is not on the top of the loop stack, push an entry of this instruction on the loop stack. The entry comprises the loop instruction address (i.e., the instruction ID) and the loop counter -1 (the loop counter is embedded into the CET instruction body).

	2. If this loop instruction is the top element of the loop stack and its current counter is not zero, add the displacement value (embedded into the CET instruction body) to the current PC and decrement the loop counter on the top of the loop stack. 3. If this loop instruction is the top element of the loop stack and its current counter is zero (the loop is done), pop off the loop stack and increment the PC.
LOOP-C	Same as LOOP except when the loop instruction is added to the loop stack, its counter is taken from the instruction body and then the increment (stride) value is added to this counter and the instruction is updated.
LOOP-R	Same as LOOP except when the loop instruction is added to the loop stack, its new counter value is fetched from the LOOP-R CET FIFO, and then this FIFO is dequeued.
LOAD/STORE	1. Send a request to the L1 data cache. The request includes the memory reference (embedded into the CET instruction body) and the operation type (read or write). 2. When the response on this request arrives to the CET core, it updates the target clock and performance registers accordingly.

LOAD-C, STORE-C	1. Send a request to the L1 data cache. The request includes the memory reference (embedded into the CET instruction body) and the operation type. 2. Add the increment value to the address field of the instruction body. 3. Write back the updated instruction to the CET instruction cache. 4. When the response on this request arrives to the CET core, it updates the target clock and performance registers accordingly.
LOAD-NC, STORE-NC	1. Reads the address difference, which is the front of the addresses CET data FIFO. 2. Dequeue the addresses FIFO. 3. Add this difference to the current address (embedded into the CET instruction body). 4. Send a request to the L1 data cache. 5. Write back the updated instruction to the CET instruction cache. 6. When the response on this request arrives to the CET core, it updates the target clock and performance

	registers accordingly.
START	1. This instruction appears in the master thread only. 2. It activates the next idle CET core to run the new thread. 3. The initial target clock value of the new thread is the current target clock value of the master thread.
PAUSE	When it is encountered by a thread, it stalls until it receives a wake signal.
WAKE	1. When it is encountered by a thread, it sends broadcasts a wake signal. 2. When a wake signal is received by a thread, it resumes execution.
TERMINATE	1. The CET core becomes idle. 2. The value of the target clock is sent to the master thread.

7.3.1.1 Loop Scheduling Unit

Although the loop counter value might be embedded into the CET code itself, it cannot be decremented in the code body at runtime. Because the initial value of the loop counter might be reused later in the case it is an inner loop. Also in the case of a loop-R (as stated before, LOOP_R is an inner loop whose number of iterations across different outer loop iterations does not follow a certain pattern, i.e., at each outer loop iteration, this inner loop has a random number of iterations), the counter value is not embedded in the code body because there is no single counter value. Thus, a stack is used to implement CET loops. This stack is called the loop stack.

When a loop instruction is encountered and it is not on the top of the loop stack, an entry of its counter and address is pushed on the loop stack. However, if the loop is on the top of the stack, its counter is decremented and the PC value is set to the loop target address, i.e., the address of the first instruction in the loop block. This is repeated until the loop counter becomes 1. After that, the loop stack is popped off and the PC is updated to the address of the instruction right after the loop instruction. The flowchart shown in Figure 27 depicts how the loop scheduling unit works.

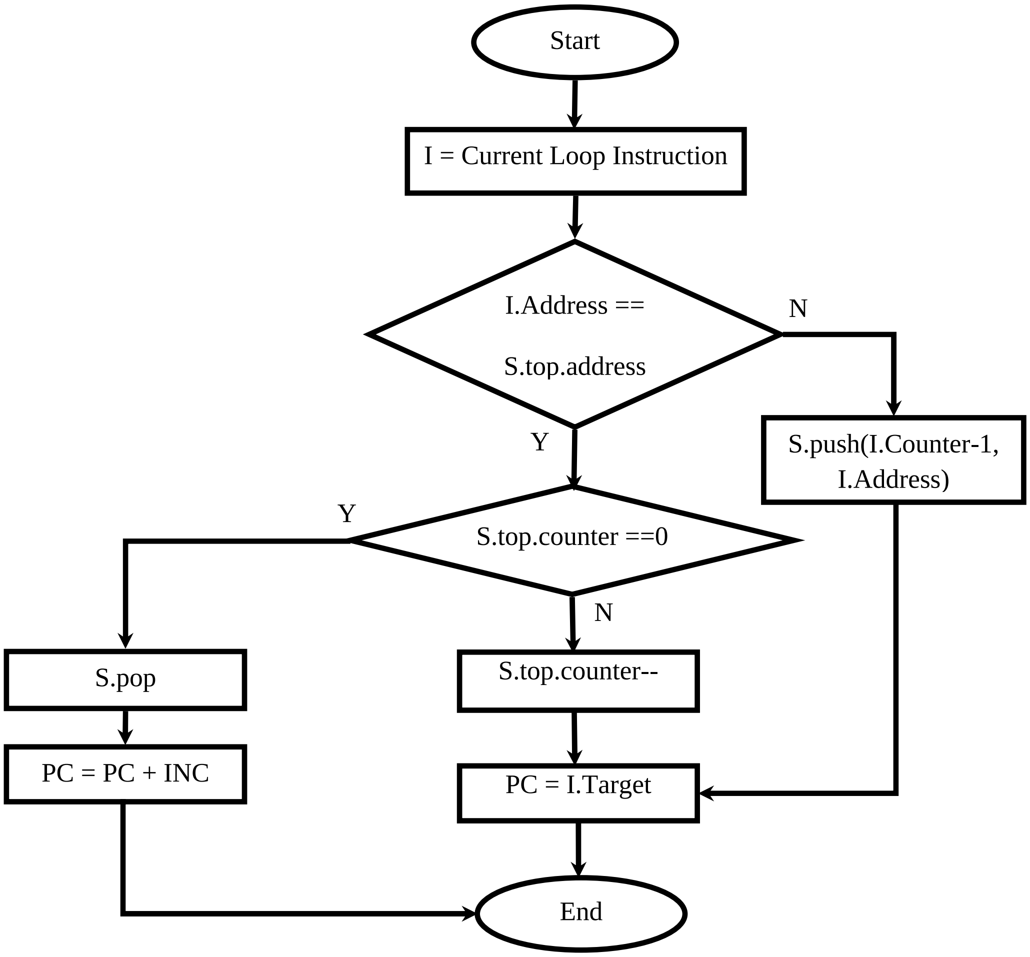


Figure 27: Loop Scheduling Unit Flowchart

7.3.2 Timing Model's Cache Memory

HySim's timing model has an off-chip main memory on the FPGA board to store the CET code and data of the application. The off-chip SDRAM is filled from the host workstation through the Ethernet port. Moreover, each tile has on-chip caches to cache a fraction of the CET code and data for quicker access. If the CET code and data of an application do not fit in the CET main memory, although this is rare, then the whole CET code and data are stored on the host PC disk and the FPGA board main memory works as a second level cache. In this case, CET main memory is organized into pages and the PC disk works as the virtual memory.

The CET code cache is implemented as a normal processor cache because the instructions of the CET code are fetched in the same order as the original program. Thus, the CET code cache is organized into cache blocks that are grouped into sets. On the other hand, the CET data cache is implemented as FIFOs because CET data are consumed sequentially. Since different CET data have different widths, they are implemented in separate FIFOs, e.g., address difference FIFO and branch results FIFO. Figure 28 shows a high level view of the timing model memory hierarchy.

The memory controller works as an interface between the CET caches and the main memory. It receives instruction requests from the different tiles (in the case of CET instruction miss) and schedules them to the main memory. Then it receives the responses to these requests (instruction blocks) and passes them to the requesting tiles. Moreover, it senses the ***IsEmpty*** and ***IsFull*** signals of all FIFOs of the different tiles. It brings the CET data from the main memory and feeds them to the empty FIFOs. However, if none of the

FIFOs is empty, it pre-fetches CET data from the main memory and feeds it to the non-full FIFOs.

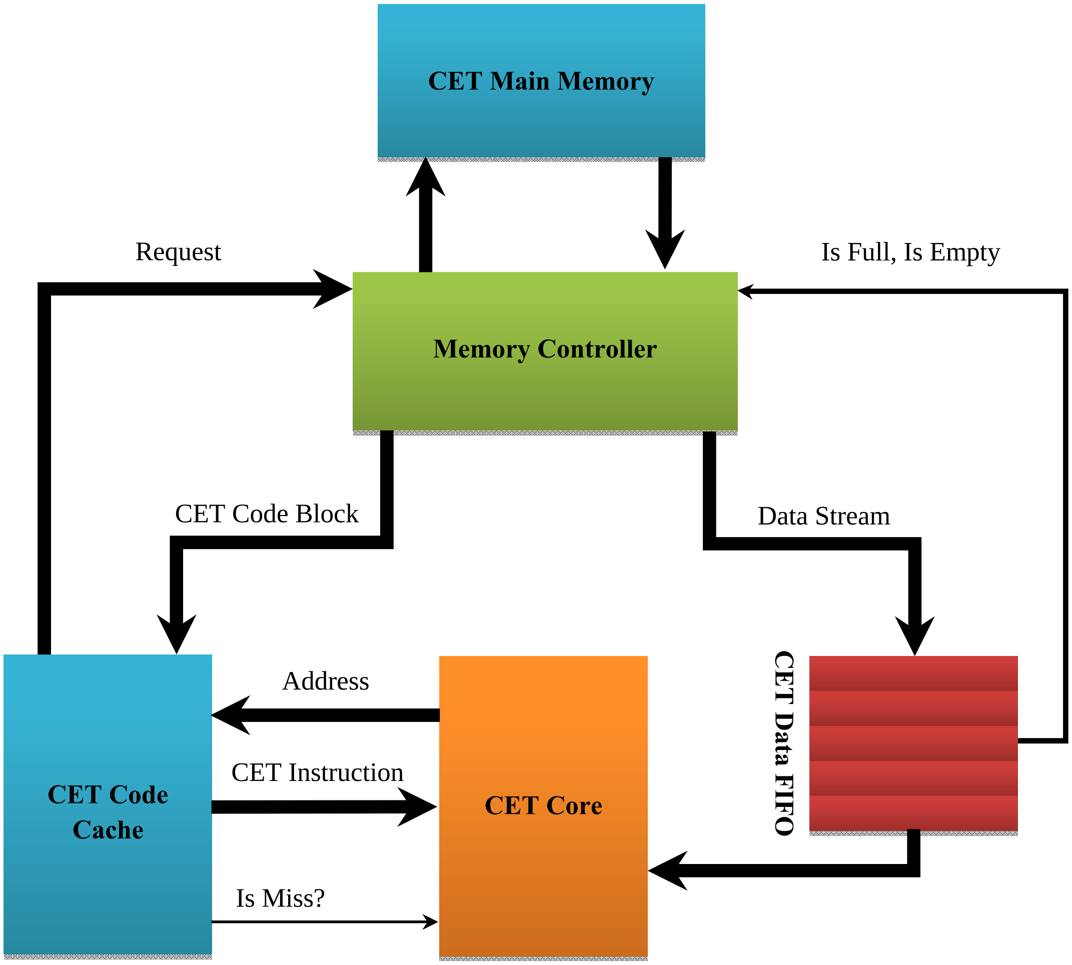


Figure 28: CET Instruction and Data Caches

7.3.3 Target Cache Hierarchy Model

HySim supports a target cache hierarchy model of up to three cache levels. The cache model only stores the data required for performance evaluation, such as: coherence states and tags and the data required for cache replacement policy. The CET core issues a request to the L1 data cache model when a load or store instruction is encountered. Besides that, it issues a request to the L1 instruction cache model each time an instruction is fetched.

If there is an L1 cache miss, then the request, either data or instruction, is forwarded to the L2 cache level. The requested address is looked up into the L2 cache model. If there is an L2 cache hit, the CET core adds the L1 cache's tag access time and the L2 cache's data access time to the simulated time. On the other hand, if there is an L2 cache miss, a cache coherence transaction is formed and delivered to the router. The router routes this transaction to the owner of the requested cache block across the ring, and finally to the initiator CET core to update the simulated time.

If there is an L3 cache hit, then L1 and L2 caches' tag access time and L3 cache data access time is added to the simulated time. However, if there is an L3 cache miss, the tag access time of L1, L2 and L3 caches is added to the simulated time in addition to the main memory access time. Moreover, the NoC latency which is calculated based on the number of hops traversed by the transaction (according to the target processor NoC topology) is added to the simulated time as well. These penalties are added after considering the overlapping among the independent events.

In the case of a cache miss, the cache model controller writes the tag and the correct state of the missed cache block in any available cache line in the cache set to mimic brining this data or instruction block from the lower level cache, i.e., the CET core does not really need to wait for the cache access time which speeds up the simulation. If there are no available cache lines in that set, then the replacement policy is applied to evict a cache block.

When the coherence transaction reaches its target L3 slice, the cache coherence protocol is applied. If this L3 slice has the requested block in a shared state, then it is the owner of the requested block. Otherwise, it broadcasts the request on the ring to search for the owner of the requested block, i.e., the tile which has the requested block in the modified state. If invalidation is required, an invalidation request is broadcasted over the ring and all copies of the block in L1 and L2 caches are invalidated.

Figure 29 shows a sample target cache hierarchy model for a 2-way set associative L2 cache. It has three input queues: one to queue the instruction requests, another one to queue the data requests and a third one to queue the coherence transaction requests. The coherence transactions come to L2 to update coherence states, e.g., to invalidate or to change from modified to shared state. The CET core stalls if any of these three queues becomes full. However, these queues are large enough to reduce this stall time.

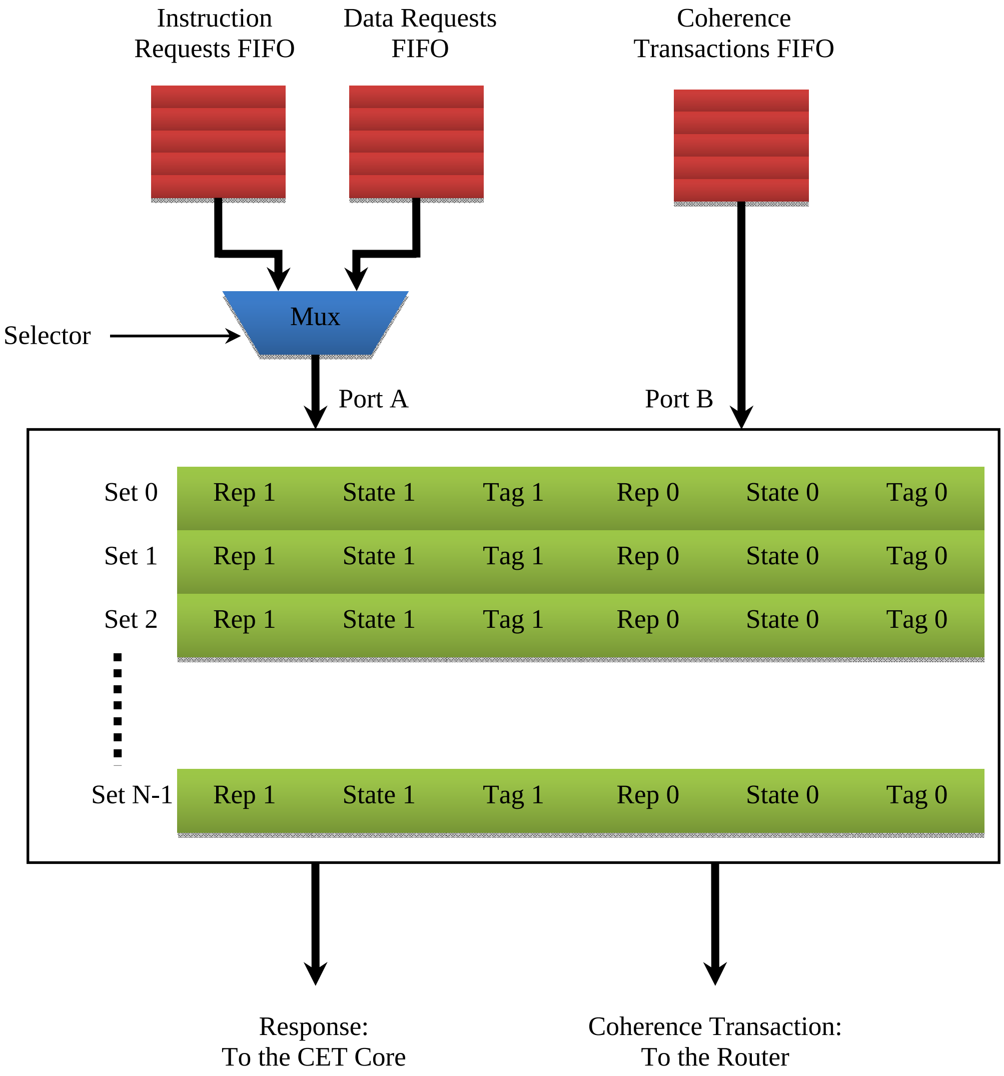


Figure 29: Unified L2 Cache Simplified Model

7.3.4 Timing Model's Router

The router module is the network interface of the tile. It is responsible for routing the messages within the same tile and among different tiles. Figure 30 shows an illustrative block diagram of the router. From this figure, this router has three input ports and four output ports. Each input port has an input queue to store the incoming messages. The messages from the different input queues are selected via a multiplexer on the router in a round robin manner, i.e., every time a message from another queue is selected. The input ports come from the following components:

1. **L2 Cache:** upon an L2 cache miss, a coherence transaction is packed and delivered to the router to forward it.
2. **L3 Cache:** to forward any request issued by the home directory.
3. **External port:** receives messages from the previous tile.

The output ports are connected to the following components:

1. **L1-Data and L2 Caches:** to deliver the invalidation messages or coherence transactions those change the cache block state from M to S.
2. **L3:** to deliver the coherence transactions to the home directory; which is embedded into L3 cache.
3. **CET Core:** after the coherence transaction is served, it is delivered to its initiator core. The initiator core adds any penalty incurred by this miss event.
4. External port which delivers messages to the next tile.

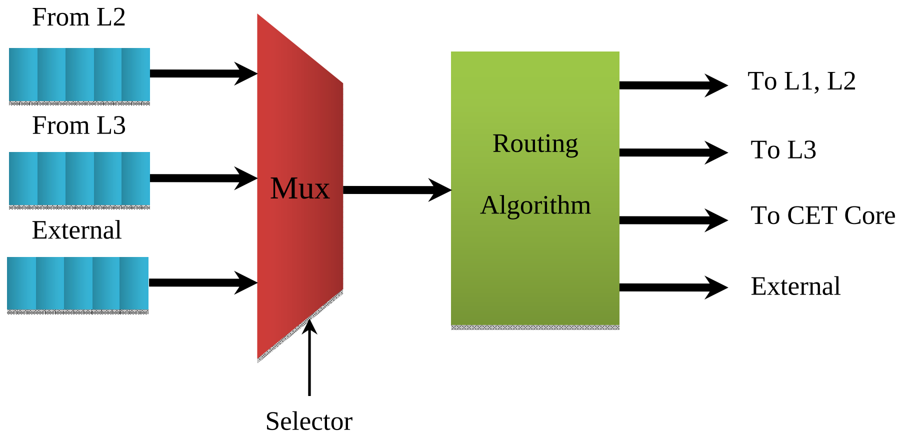


Figure 30: Router Block Diagram

7.4 NoC Model

The current version of HySim implements a simple NoC model. Although the timing model's tiles are connected via a ring, it can model a target processor with a different topology. For each coherence transaction, the initiator CET core of this transaction tracks how many hops are navigated by this transaction according to the target processor's NoC topology. Finally, the total latency of the transaction is calculated according to an analytical model that depends on the number of hops. Currently, the latency is calculated by multiplying the number of hops by the hop latency. The latter is a tunable parameter.

7.5 Multi-threading Management

As stated before, the CET tool generates separate CET code and data files per thread even if multiple threads are assigned to the same core. In HySim, these threads are assigned to the available target cores' models statically, e.g., threads 0 and 1 are assigned to core 0, and so on. The number of threads per core is an architectural tunable parameter; it refers to the maximum number of threads that can be processed by a single processor core concurrently. Nevertheless, the total number of threads should equal the number of target cores times the number of threads per core.

7.5.1 Thread Scheduling on the Same Core

Multiple threads can be scheduled on the same processor core in three ways [83], **(1) interleaved**, an instruction of another thread is fetched and fed into the execution pipeline at each clock cycle, **(2) blocked**, the instructions of a thread are executed successively until a long latency event occurs which results in a context switch, **(3) simultaneous**, the instructions are simultaneously issued from multiple threads to the execution units of a superscalar.

Because instructions can be fetched from only one thread at a time in *interleaved* and *blocked* multithreading techniques, HySim simulates these techniques by partitioning the CET code and data caches of the CET core among the simulated threads assigned to this core. In other words, the CET I-cache and CET data FIFOs are divided into equal-sized partitions such that each partition belongs to a different thread. However, CET core keeps a distinct context per simulated thread, this context includes a distinct loop stack and registers to store the CET instruction and data memory addresses. On the other hand, the target architecture memory model remains as is regardless the number of threads assigned to the core.

Figure 31 depicts how multiple threads (in interleaved and blocked techniques) can be scheduled on a single CET core. Multiplexers are used to determine the address of which thread should be selected to access the CET code and data caches. The control signal of these multiplexers is the active thread ID. Thread IDs are stored in a circular queue, called thread queue. The active thread is the one whose ID is the first element of this queue. When a certain thread is done, its ID entry is pooped off permanently from the thread queue and hence it will not be scheduled anymore.

In interleaved multithreading technique, the context switch occurs after fetching each instruction. Thus, this technique is a fine-grained multithreading technique. In contrast, in blocked technique, the context switch occurs when the active thread generates a long latency miss event. At each context switch, the thread queue is popped off and the popped off element (current thread ID) is queued at the tail of this queue.

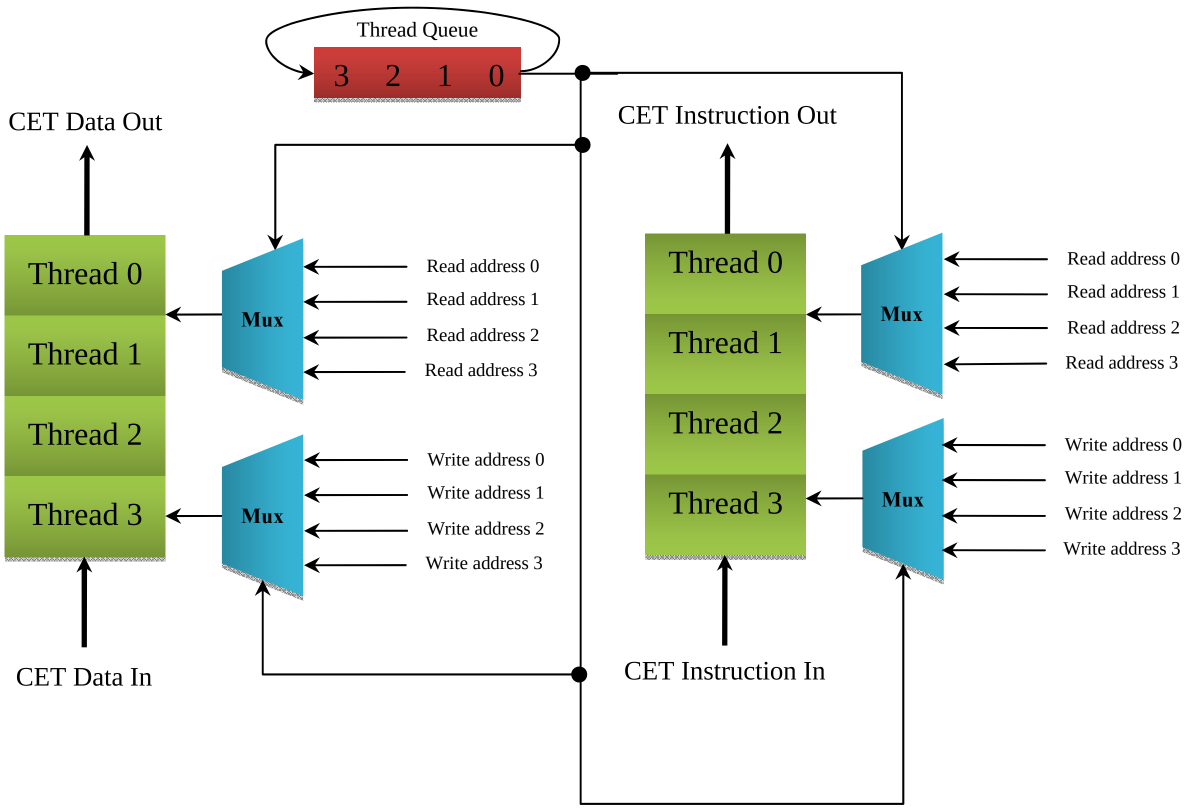


Figure 31: Threads Management in Multithreaded Target Cores (Interleaved and Blocked)

In contrast, simulating a simultaneous multithreaded core (i.e., hardware threads) requires replicating the fetch unit; decode logic, loop stack, latency computing logic, and CET code and data caches. Replicating the CET cache is necessary because each FPGA block RAM has only two ports. Thus, replicating the CET cache results in more efficient access rather than queuing the requests and serving them serially. Figure 32 shows a simple schematic that depicts how a simultaneous multithreaded core can be simulated in HySim.

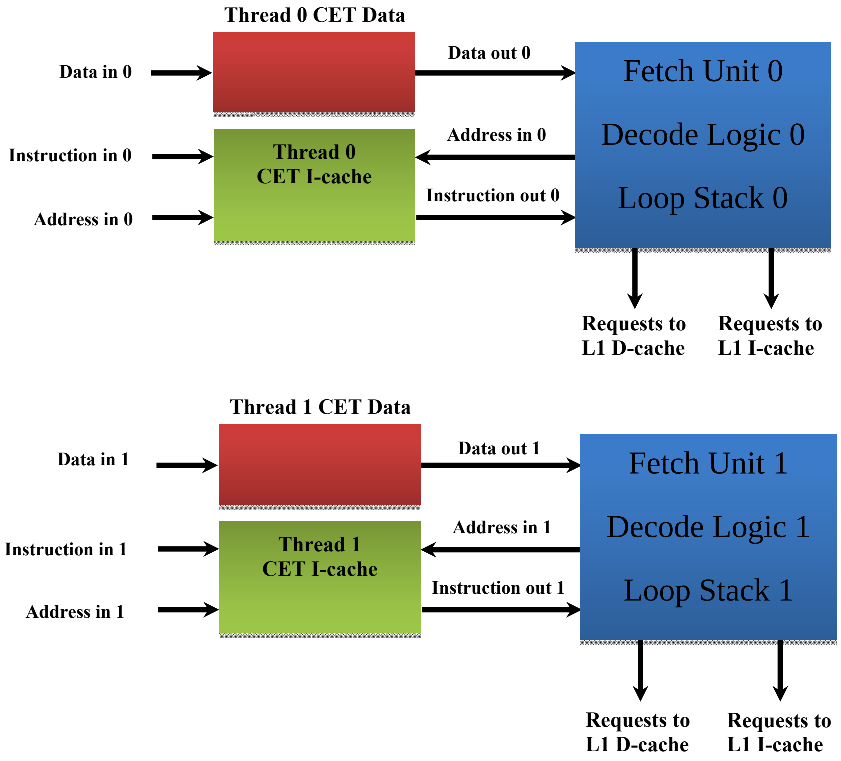


Figure 32: Threads Management in Multithreaded Target Cores (Simultaneous)

7.6 Inter-Thread Interactions

As stated before, HySim targets coherent shared memory multicores. Therefore, it models a cache coherence protocol to capture the effect of coherency on the derived parallel execution time. There is a local target clock per core model. Thus, each core model derives the execution time on the corresponding target core individually. This execution time includes the different latencies a target core might experience, such as, computation time, cache miss penalties, NoC latency, and synchronization barrier waiting time.

For example, assume **core 0** wants to read a data block **'A'** that is not cached in any of the caches. **Core 0** will look it up in its private L1 data cache, then L2 cache, and finally in the L3 cache and a read cache miss occurs at each level. After that, this block is brought from the main memory to the private caches of **core 0** (L1 and L2) and the shared L3 cache in the 'shared' state assuming that the cache hierarchy is inclusive. Then, **core 0** will account for the cache misses penalties across the whole memory hierarchy. After block **'A'** is cached, assume that **core 1** wants to write to this block, it will experience a write miss. However, this block has already been cached. Therefore, **core 1** will get it from **core 0** rather than from the main memory, i.e., a core-to-core communication happens. According to the cache coherence protocol, an invalidation message is broadcasted to the other cores to invalidate block **'A'** that is eventually brought to **core 1**'s private caches in the 'modified' state. The state of block **'A'** becomes 'invalid' in **core 0** private caches and 'modified' in the shared L3 cache. However, if **core 1** wants to read block 'A' instead of writing to it, it will experience a read miss in its L1 and L2 private caches and a read hit in **L3** cache because block **‘A’** has already been brought from the main memory by **core 0**. In this case, **core 1** accounts for L1 and L2 misses and L3 hit penalties only, i.e., it does not account for the main memory latency because it did not access the main memory in this case.

Regarding synchronization barrier, HySim models a counter-based barrier. The barrier is encoded in the CET code by the wait instruction. When a thread **Ti** reaches a barrier, it increments the barrier counter and registers the time stamp **TSi** (the current value of the core’s local target clock). When the last thread reaches the barrier, it resets the barrier counter and broadcasts the time stamp **T** to other threads. After a thread **Ti** receives the time stamp **T**, it calculates the difference between its **TSi** and **T**. This difference between the two time stamps is the barrier waiting time for thread **Ti**. Moreover, this leads to synchronizing the local target clocks of all cores.

7.7 FPGA Implementation Details

HySim timing model has been implemented on a Xilinx Virtex 6 XC6VLX550T FPGA board, Figure 33. The maximum FPGA frequency achieved was ~170 MHz.

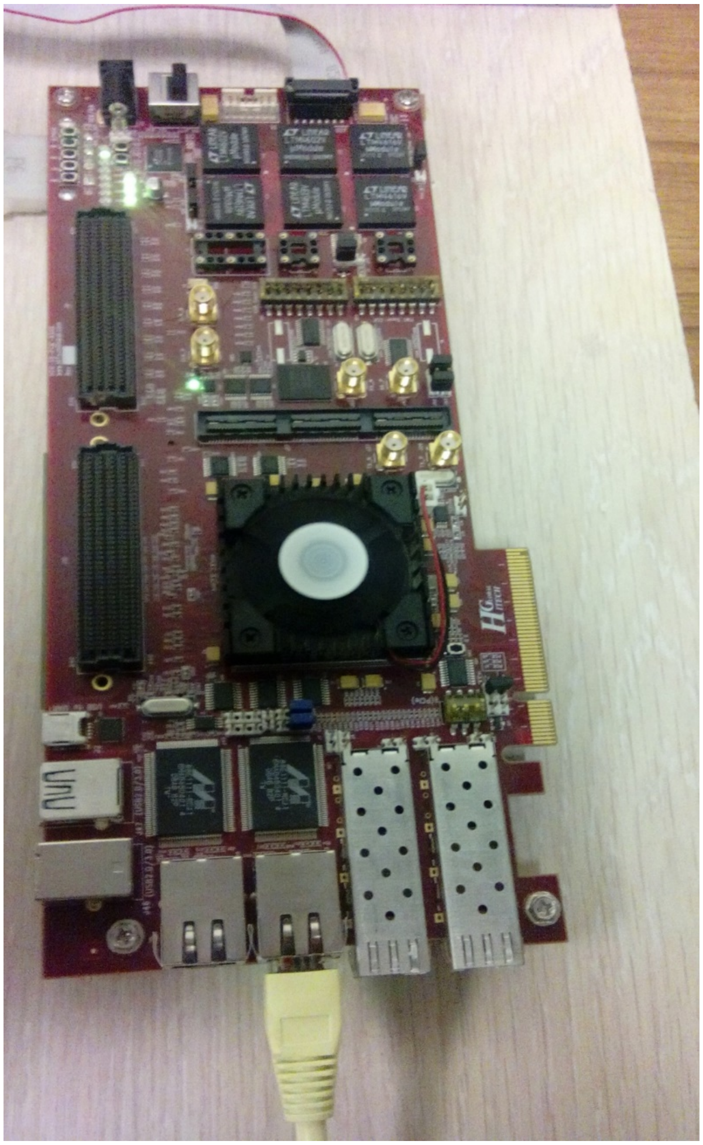


Figure 33: Virtex 6 XC6VLX550T FPGA Board

Since the DDR3 controller consumed a significant part of the XC6VLX550T FPGA resources and lowered the frequency on which the FPGA operates, we decided to keep the CET main memory (the whole CET code and data of the simulated application) on the hosting workstation disk while caching a significant fraction of these data on the FPGA (CET caches). Of course, the larger the CET caches the faster the simulation. The size of CET caches is tunable. It can be adjusted to fit the application subject to the constraint that the total size of these CET caches is less than or equal the size of the available BRAMs (BRAMs are blocks of SRAMs embedded in the FPGA).

Fortunately, we were able to cache all branches' results on the FPGA. Furthermore, for the majority of applications we profiled, the entire CET code size can be cached too. Thus, the remaining CET data components that require caching are the non-contiguous addresses, JUMP_M displacements, and random loop counters. Moreover, since *thread 0* (the master thread) contains both the sequential and parallel regions of the application, it has a larger CET code and data size than the other threads. Hence, the CET caches of *CET core 0* (the master core which simulates thread 0) was made larger than the CET caches of other cores.

7.7.1 Host-FPGA Communication

The Ethernet interface was used for the communications between the host workstation containing the whole CET code and data and the FPGA running the timing model. UDP (User Datagram Protocol) protocol has been adopted as the communication protocol since it has a small header and can be routed safely through the network devices.

The communication circuit on the FPGA is explained in Figure 34. The Ethernet core buffers the received packet at 125MHz speed, reads it from the buffer at the user design speed, namely, 170 MHz, , buffers it again in the transmitting buffer (with modification if needed) at the user design speed, and then sends a reply to the workstation at 125MHz. The packet is thrown once received if it has a wrong CRC (Cyclic Redundancy Check) or wrong MAC address. Thus, each user packet has an immediate replay by a packet of the same size and same architecture.

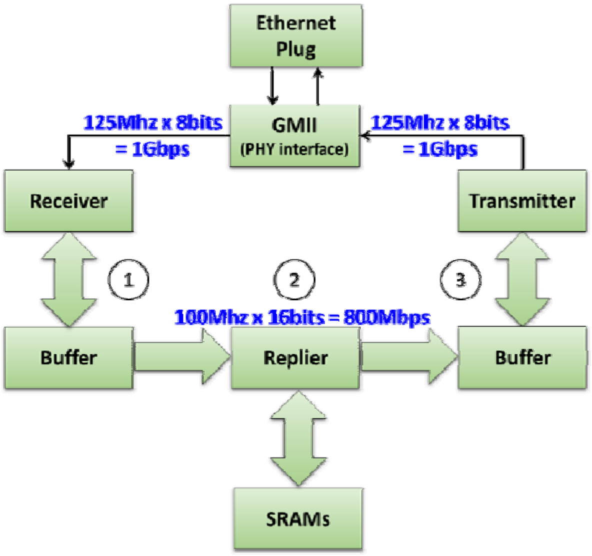


Figure 34: Data flow for the Ethernet Core.

Each packet carries a list of read/write commands. It can target a block of memory, one word in memory, or a register. Once the packet is received by the communication circuit on the FPGA, all of the commands inside it are executed via the Ethernet core. For the write commands, it writes the data without modifying the packet. However, if it is a read command, then it replaces the data that follows the command by the read one.

The modification on the packet header includes interchanging the source/destination MAC addresses, source/destination IP addresses, the source/destination port addresses, and put the payload checksum to zero to indicate that it is not calculated. Each packet is originally an Ethernet packet that contains a UDP packet which in turn contains a serial of command packets. The total Length of the packet should be more than 50 bytes and could reach around one thousand bytes.

7.7.2 CET Cache Filling Circuit

We implemented a circuit on the FPGA to monitor and fill the CET caches. This circuit has been called the CET cache filling circuit. As stated above, the communication between the workstation and this circuit is done through the Ethernet. Figure 35 shows how this filling circuit interacts with the workstation and the HySim timing model. In addition to these signals shown in this figure, there are handshaking signals for each data and address bus. The cache filling circuit interacts with HySim tiles as follows:

1. Initially, the CET code and data caches are filled at FPGA configuration time.

2. The CET cache filling circuit maintains the current addresses of all CET memories of each tile. i.e., for addresses data, conditional branch results, etc.
3. It keeps sensing the ***IsEmpty*** and ***IsFull*** signals of all tiles. Actually each CET data FIFO on the tile has this pair of signals.
4. If the ***IsEmpty*** signal of a certain FIFO is set, then this FIFO is empty and the filling circuit fetches the required data from the hosting workstation and updates the data address associated with this FIFO.
5. If none of the FIFOs is empty, then the filling circuit pre-fetches CET data from the hosting workstation for the FIFOS whose ***IsFull*** signals are zero, in a round robin manner**.**
6. If the CET code size is not large and can be stored entirely on the FPGA BRAMs which is a common situation, then the filling circuit has nothing to do with the CET code cache because there will be no CET code misses. In this case, the CET cache is just a normal memory that stores the CET instructions only, i.e., there are no tags stored like in the normal caches.
7. However, if the CET code size is large and hence cannot be stored entirely on the FPGA, which is a rare situation, then the filling circuit keeps listening for CET instructions' requests. Once it receives an instruction request, it fetches an entire CET code block from the hosting workstation and delivers it to the requesting tile.

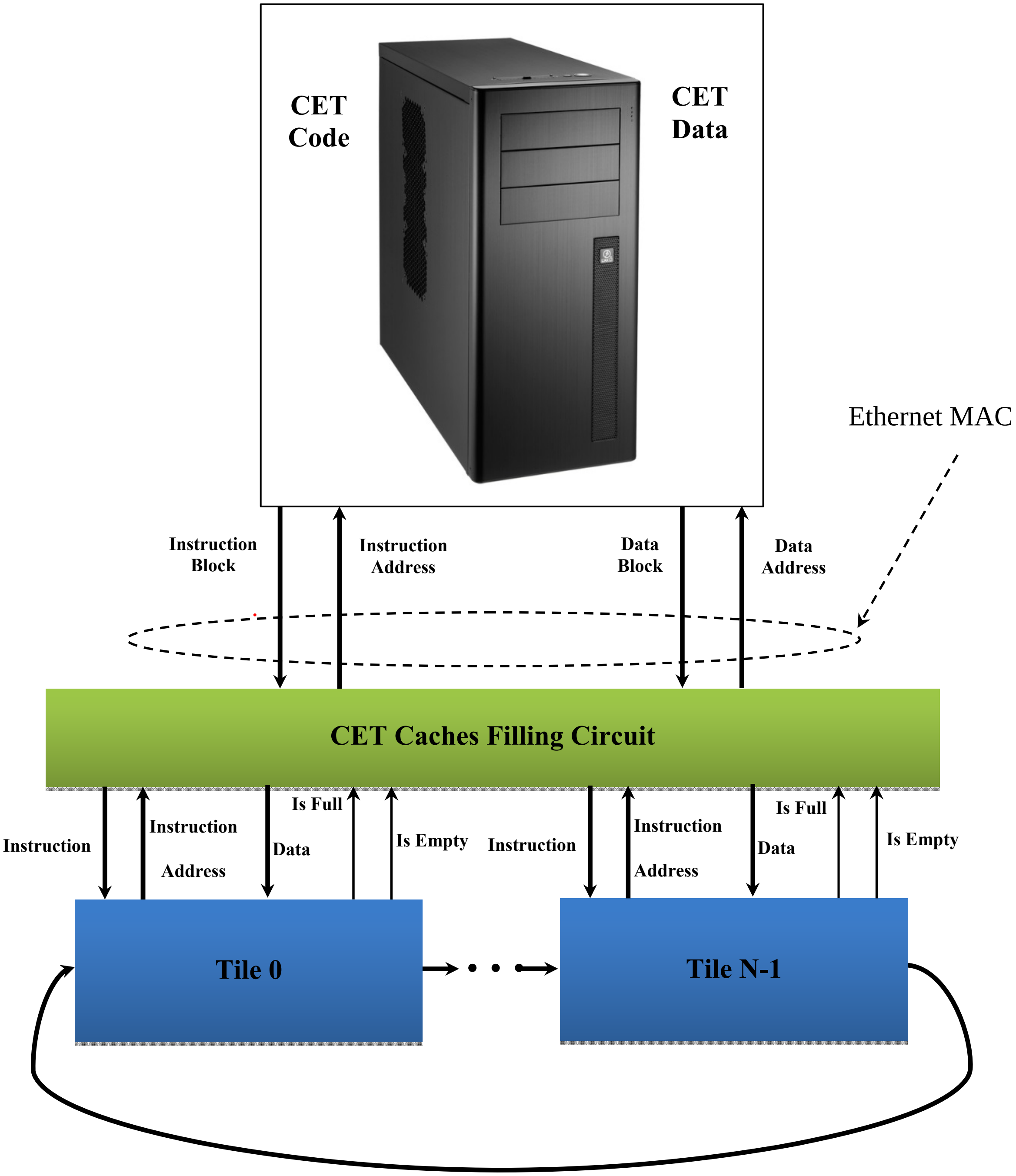


Figure 35: High Level View of HySim Timing Model Interaction with the Main Memory

7.7.3 FPGA Resources Consumption

Table 13 shows the amount of FPGA resources consumed by sixteen tiles with and without the CET cache filling circuit. These sixteen tiles model Xeon E5-2680 processor [2]. This processor comprises two sockets. Each socket has eight cores. 32 KB L1 D-cache, 32 KB L1 I-cache, and 256 KB unified L2 cache are private for each core. A 20 MB L3 unified cache per socket shared and distributed among the eight cores of each socket. This table shows that the CET caches filling circuit consumes few FPGA resources only.

Table 13: The Amount of FPGA Resources Consumed by One and 16 CET Tiles

FPGA Resource	Without CET Cache Filling Circuit		With CET Cache Filling Circuit	
	Used	Utilization	Used	Utilization
One CET Tile				
Number of Slice Registers	2171	0%	2,306	1%
Number of Slice LUTs	9964	2%	2,306	1%
Number of Fully Used LUT-FF Pairs	1704	16%	2,306	23%
Number of Block RAM/FIFO	39	6%	43	6%
Sixteen CET Tiles				
Number of Slice Registers	30342	4%	45,709	6%
Number of Slice LUTs	142594	41%	139,694	40%
Number of Fully Used LUT-FF Pairs	23894	16%	34,576	23%
Number of Block RAM/FIFO	517	81%	544	86%

We notice that the FPGA block RAMs is the most critical FPGA resource. It is recommended to utilize all of the available block RAMs to maximize the CET caches size and hence maximize the simulation speed. Moreover, Table 13 shows that most of the registers and LUTs are still free; these free resources can be utilized to build larger CET caches or more complex NoC and core models.

Table 14 shows the sizes of the different CET caches of a single CET tile. These caches are large enough to minimize the CET caches miss rate. Moreover, the CET code cache is large enough to accommodate the whole CET code for the majority of applications which means no CET code misses. Fortunately, CET data are not required per instruction, which means less pressure on the CET data caches. Another notice from Table 14 is the size of LOOP_R counters cache is too small because they are consumed very slowly, e.g., one loop counter might be sufficient for one million loop iterations.

Table 14: The Sizes of Different CET Caches for a Single CET Tile

CET Cache	**Size** (**Number of words x word width (*bits*)**)
CET Code Cache	13000 x 55
JUMP_M displacements	4096 x 17
Non-Contiguous Data Addresses Differences	8192 x 16
LOOP_R Counters	50 x 32

CHAPTER 8

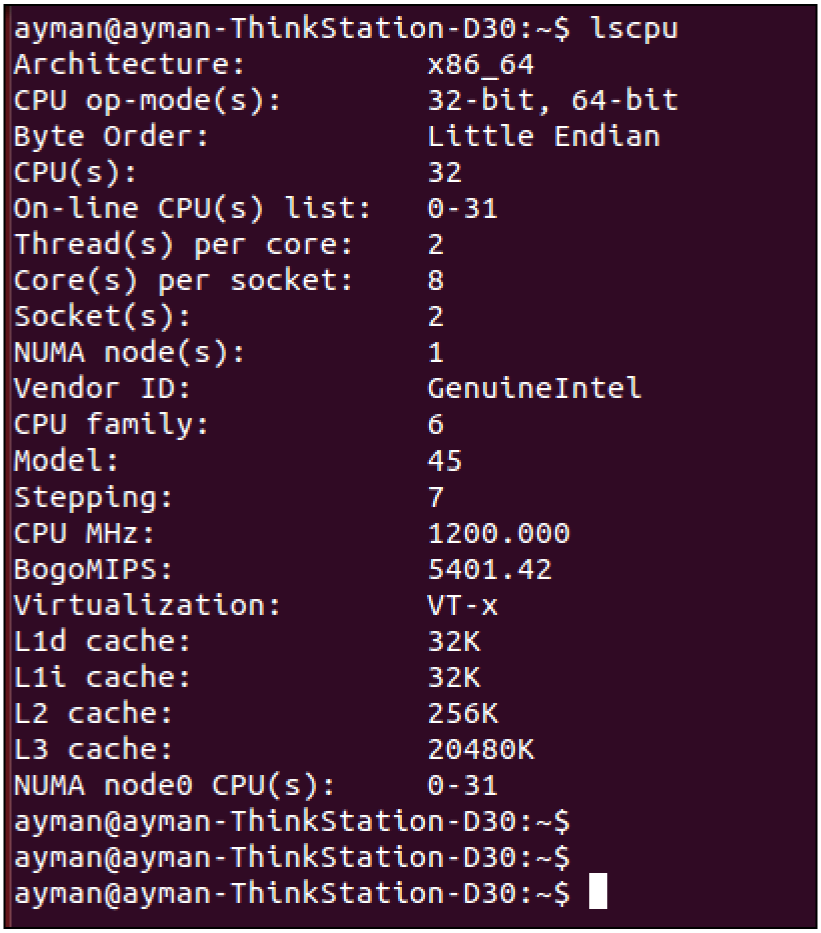
HYSIM EXPERIMENTAL RESULTS

HySim has been evaluated using several benchmarks. This chapter delves into the results of experiments we performed to evaluate HySim's speed and accuracy. It presents the simulation speed in MIPS and shows the ratio between the simulation and simulated time. Moreover, it presents the absolute simulation accuracy relative to real existing hardware execution and shows the ability of HySim to capture the performance trend of the target architecture.

8.1 Experimental Setup

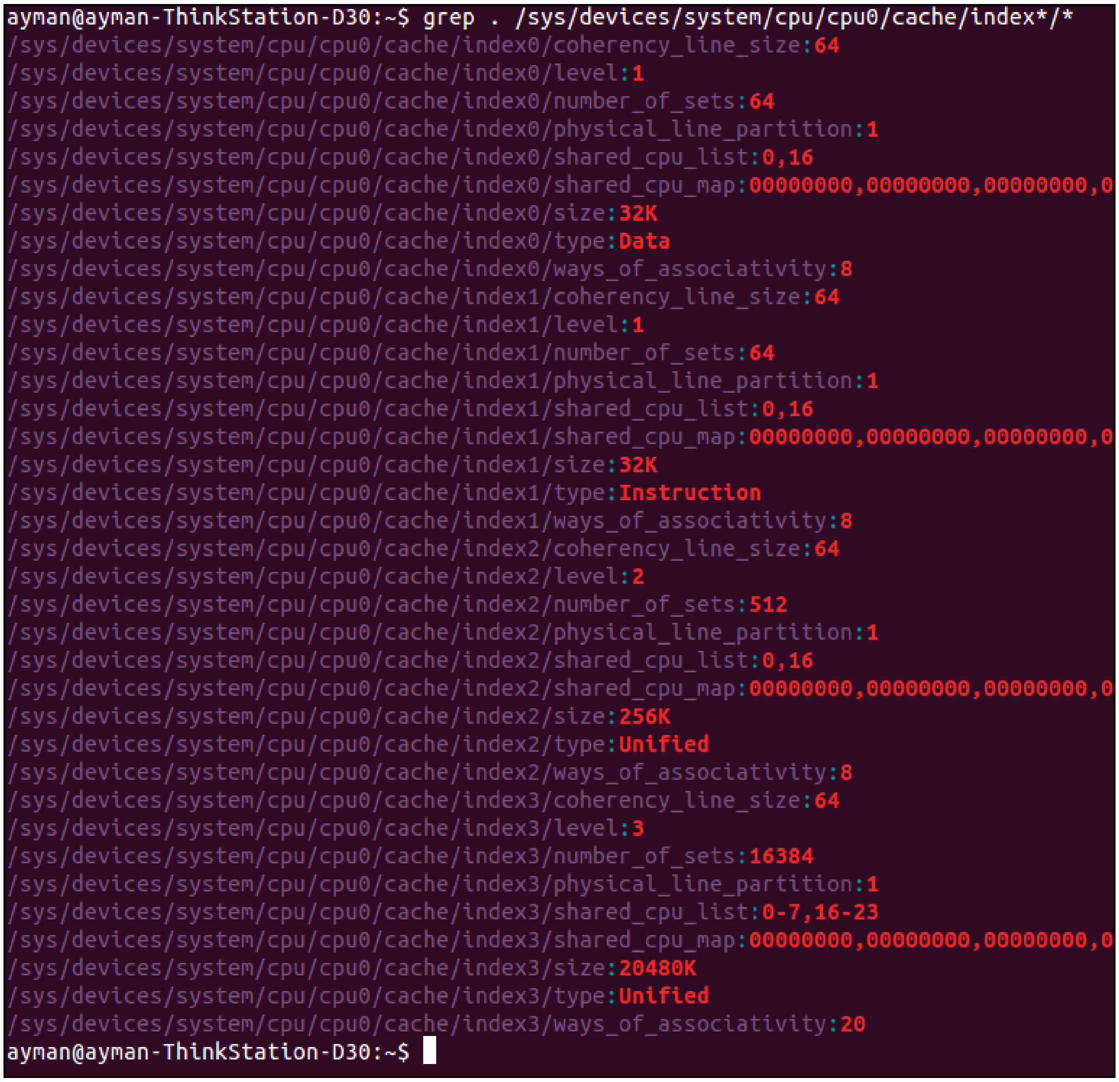
8.1.1 Target Machine specifications

HySim has been validated against a real hardware processor, namely Intel Xeon CPU E5-2680 [2] on a "*ThinkStation*" workstation. Many of the architectural specifications of this machine have been gathered from the machine itself using Linux commands and from some on-line documentations, such as [84]. Figure 36 and Figure 37 show sample snapshots of the output of some Linux commands which have been used to retrieve the machine's specifications. Then, HySim timing model has been configured to capture the machine's specifications listed in Table 15.



```
ayman@ayman-ThinkStation-D30:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Stepping:              7
CPU MHz:               1200.000
BogoMIPS:              5401.42
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-31
ayman@ayman-ThinkStation-D30:~$
ayman@ayman-ThinkStation-D30:~$
ayman@ayman-ThinkStation-D30:~$ 
```

Figure 36: The Output of *lscpu* Linux Command



```
ayman@ayman-ThinkStation-D30:~$ grep . /sys/devices/system/cpu/cpu0/cache/index*/*
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets:64
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:0,16
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map:00000000,00000000,00000000,0
/sys/devices/system/cpu/cpu0/cache/index0/size:32K
/sys/devices/system/cpu/cpu0/cache/index0/type:Data
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity:8
/sys/devices/system/cpu/cpu0/cache/index1/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index1/level:1
/sys/devices/system/cpu/cpu0/cache/index1/number_of_sets:64
/sys/devices/system/cpu/cpu0/cache/index1/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index1/shared_cpu_list:0,16
/sys/devices/system/cpu/cpu0/cache/index1/shared_cpu_map:00000000,00000000,00000000,0
/sys/devices/system/cpu/cpu0/cache/index1/size:32K
/sys/devices/system/cpu/cpu0/cache/index1/type:Instruction
/sys/devices/system/cpu/cpu0/cache/index1/ways_of_associativity:8
/sys/devices/system/cpu/cpu0/cache/index2/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index2/level:2
/sys/devices/system/cpu/cpu0/cache/index2/number_of_sets:512
/sys/devices/system/cpu/cpu0/cache/index2/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list:0,16
/sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_map:00000000,00000000,00000000,0
/sys/devices/system/cpu/cpu0/cache/index2/size:256K
/sys/devices/system/cpu/cpu0/cache/index2/type:Unified
/sys/devices/system/cpu/cpu0/cache/index2/ways_of_associativity:8
/sys/devices/system/cpu/cpu0/cache/index3/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index3/level:3
/sys/devices/system/cpu/cpu0/cache/index3/number_of_sets:16384
/sys/devices/system/cpu/cpu0/cache/index3/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list:0-7,16-23
/sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_map:00000000,00000000,00000000,0
/sys/devices/system/cpu/cpu0/cache/index3/size:20480K
/sys/devices/system/cpu/cpu0/cache/index3/type:Unified
/sys/devices/system/cpu/cpu0/cache/index3/ways_of_associativity:20
ayman@ayman-ThinkStation-D30:~$
```

Figure 37: Cache Hierarchy Architectural Specifications of "*ThinkStation*" Workstation.

Table 15 shows that the Xeon E5-2680 machine has two sockets interconnected via a QPI (Quick Path Interface). Each socket has eight 2-way multithreaded cores. Each core has a 32 KB L1 D-Cache, a 32 KB L1 I-Cache and a 256 KB unified cache. The eight cores of each socket share a 20 MB L3 unified cache. This L3 cache is split into 10 slices. A ring interconnects the eight cores, the L3 cache’s slices, and it has stops for the QPI and the memory agent of the socket.

Table 15: Target Machine Architectural Specifications

Parameter	Value
Number of sockets	2
Cores per socket	8
CPU minimum frequency	1200 MHz
CPU maximum frequency	2.7 GHz
Threads per core	1
Architecture	X86_64
Cache line size	64 Byte
L1 I–cache size	32 KB per core
L1 D–cache size	32 KB per core
L2 cache size	256 KB per core
L3 cache size	20 MB per socket
L1 instruction cache associativity	8
L1 data cache associativity	8
L2 cache associativity	8
L3 cache associativity	20
L1 instruction cache latency	3 cycles data access (in the case of a hit),

	1 cycle tag access(in the case of a miss)
L1 data cache latency	3 cycles data, 1 cycle tag access
L2 cache latency	12 cycles data, 3 cycles tag access
L3 cache latency	38 cycles data, 12 cycles tag access
Main memory latency	~175-350 Cycles [84]
Cache coherence protocol	MSI
NoC model (per socket)	Un-buffered ring
NoC across sockets	QPI
Hop Latency	2 cycles
Reorder Buffer size	96

8.1.2 Real Hardware Execution Time Measurement

The execution time of the benchmarks on the existing real machine was measured as follows:

1. The benchmarks were executed under ***Ubuntu 14.3*** operating system.
2. The CPU frequency was fixed to 1200 MHz (the minimum frequency of the machine) because the machine frequency can vary between 1200 MHz and 2700 MHz on demand (for power and performance tradeoffs). We did that via ***`cpufrequtils`*** software.
3. All measurements were taken in Linux ***Console Mode*** to alleviate the system overhead on the measured execution time.
4. The benchmarks were run for 100 times successively and the running average was considered.
5. The execution time has been measured using the Linux ***`time`*** command which shows the amount of time spent in the application level code and system level code.
6. Hyper threading was disabled from the system ***setup*** to ensure that only one thread is assigned to each core at a time.
7. Since disabling cache pre-fetching is not visible to the user in modern Intel processors, we tried to approximately mimic the real machine by implementing a simple cache pre-fetcher in HySim in which the next cache block is pre-fetched upon any cache miss.

8.1.3 Benchmarks

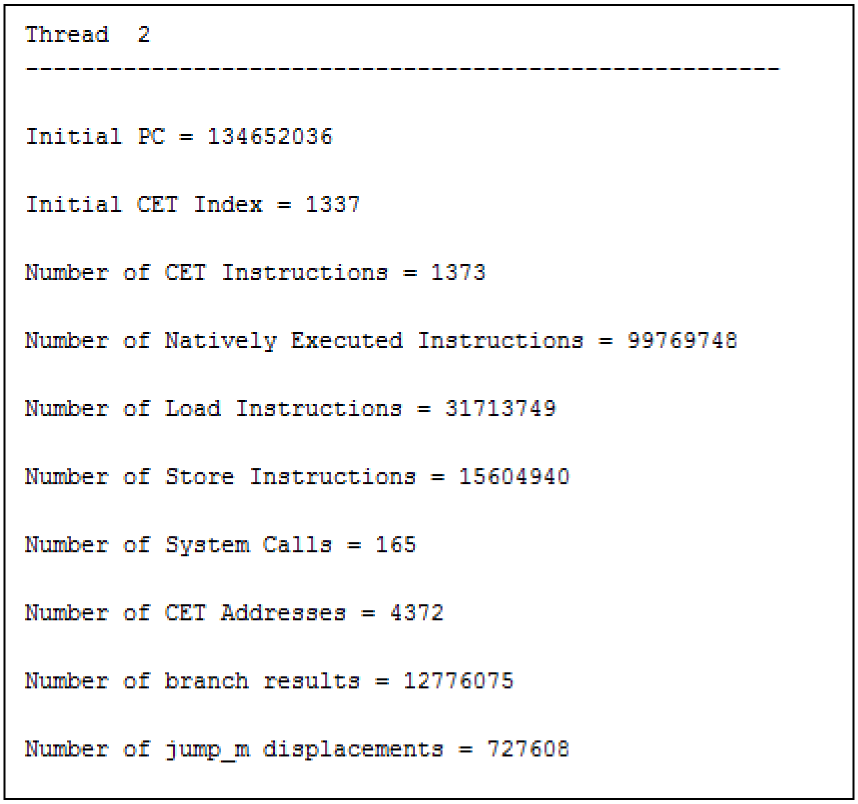
HySim has been evaluated using a mix of Splash-2 workloads [76] and PARSEC benchmarks suite [77]. Table 16 lists these benchmarks with their input set sizes used in HySim's evaluation.

Table 16: Splash-2 Benchmarks and Their Input Sets

Benchmark	Input Set Size
Swaptions	16 swaptions, 5,000 simulations
Blackscholes	4,096 options
LU-cont	512×512 matrix
FFT	256K points
Ocean-cont	258×258 ocean
Radix	256K integers
Water-sp	512 molecules
Water-nsq	512 molecules

8.1.3.1 Benchmarks Profiling

These benchmarks have been executed natively under our Pin-based CET tool to generate CET code and data for them. Besides that, the CET tool generates a profile for each thread. This profile includes the thread ID, the starting address of the CET code, the starting address of the thread's original code in memory to simulate the I-cache, the number of CET instructions and CET data, etc. Figure 38 shows a sample snapshot of such profile.



```
Thread  2
------------------------------------------------------

Initial PC = 134652036

Initial CET Index = 1337

Number of CET Instructions = 1373

Number of Natively Executed Instructions = 99769748

Number of Load Instructions = 31713749

Number of Store Instructions = 15604940

Number of System Calls = 165

Number of CET Addresses = 4372

Number of branch results = 12776075

Number of jump_m displacements = 727608
```

Figure 38: A Snapshot from a Sample Thread Profile

Table 17 shows the static CET code size in number of CET instructions. It is noticeable that **thread 0** (the master thread) has the largest CET code size and also it usually has the largest CET data size. This is normal because the master thread contains both the sequential and parallel regions of the application. This information helps the user to customize the CET cache sizes in order to minimize or even eliminate CET cache misses and hence accelerate the simulation. However, customizing such caches requires re-synthesizing the design and hence reconfiguring the FPGA.

Table 17: CET Static Code Size for Different Threads

Benchmark	Thread 0	Thread 3	Thread 6	Thread 9	Thread 12	Thread 15
Swaptions	2992	2616	2612	2616	2616	2614
Blackscholes	1706	577	571	571	571	571
LU	8130	1379	1464	1328	1393	1388
FFT	7743	1556	1563	1808	1564	1530
Ocean	17159	7089	6606	6950	7083	7234
Radix	7375	1256	1439	1398	1251	1291
Water-sp	13349	2657	2641	2627	2642	2659
Water-nsq	12895	2453	2466	2538	2503	2499

The CET tool can tell the user the number of dynamically executed instructions per thread and the percentages of different instructions, e.g., the percentage of loads and stores. This information helps the user in analyzing the simulation results, e.g., correlating the simulated time fraction in data memory with the percentage of loads in the application, and so on. Table 18 shows the number of natively executed instructions and load/store percentages for thread 0 for different benchmarks.

Table 18: Number of dynamically Executed Instructions and Load/Store Percentages for Thread 0

Benchmark	No. Dynamic Instructions	No. Loads	% Loads	No. Stores	% Stores
Swaptions	663549677	204985515	31	40073815	6
Blackscholes	102274425	21565922	21	10824091	11
LU	445635023	143393009	32	69568782	16
FFT	217280607	45676848	21	29356648	14
Ocean	518596741	214901848	41	42604059	8
Radix	93158278	37128167	40	16824188	18
Water-sp	272018359	57730032	21	27385908	10
Water-nsq	308050190	63417694	21	29879723	10

Furthermore, the number of instructions natively executed per each thread reveals the scalability of the application and whether the workload has been well balanced.

Figure 39 and Figure 40 show how instructions are distributed among the different threads. The horizontal bars are divided into rectangles such that each rectangle represents the number of dynamic instructions per thread. These rectangles from left to right represent threads 0, 1, 2, 4, etc. The workload is well balanced for some benchmarks, such as, ***radix*** and ***ocean.*** However, the load is not well balanced for other benchmarks, such as, ***FFT*** and ***LU***, where **thread 0** executed much more instructions than the other threads.

Load unbalancing can be because that the application itself has significant inherently sequential parts, or due to bad programming, i.e., the programmer could not identify all parallelism in the application. Load unbalancing limits the application execution speedup because significant part of the program has to be executed sequentially regardless the computation power of the machine. The same thing applies to HySim, load unbalancing increases the simulation time of multi-threaded applications because significant part of the application is simulated sequentially.

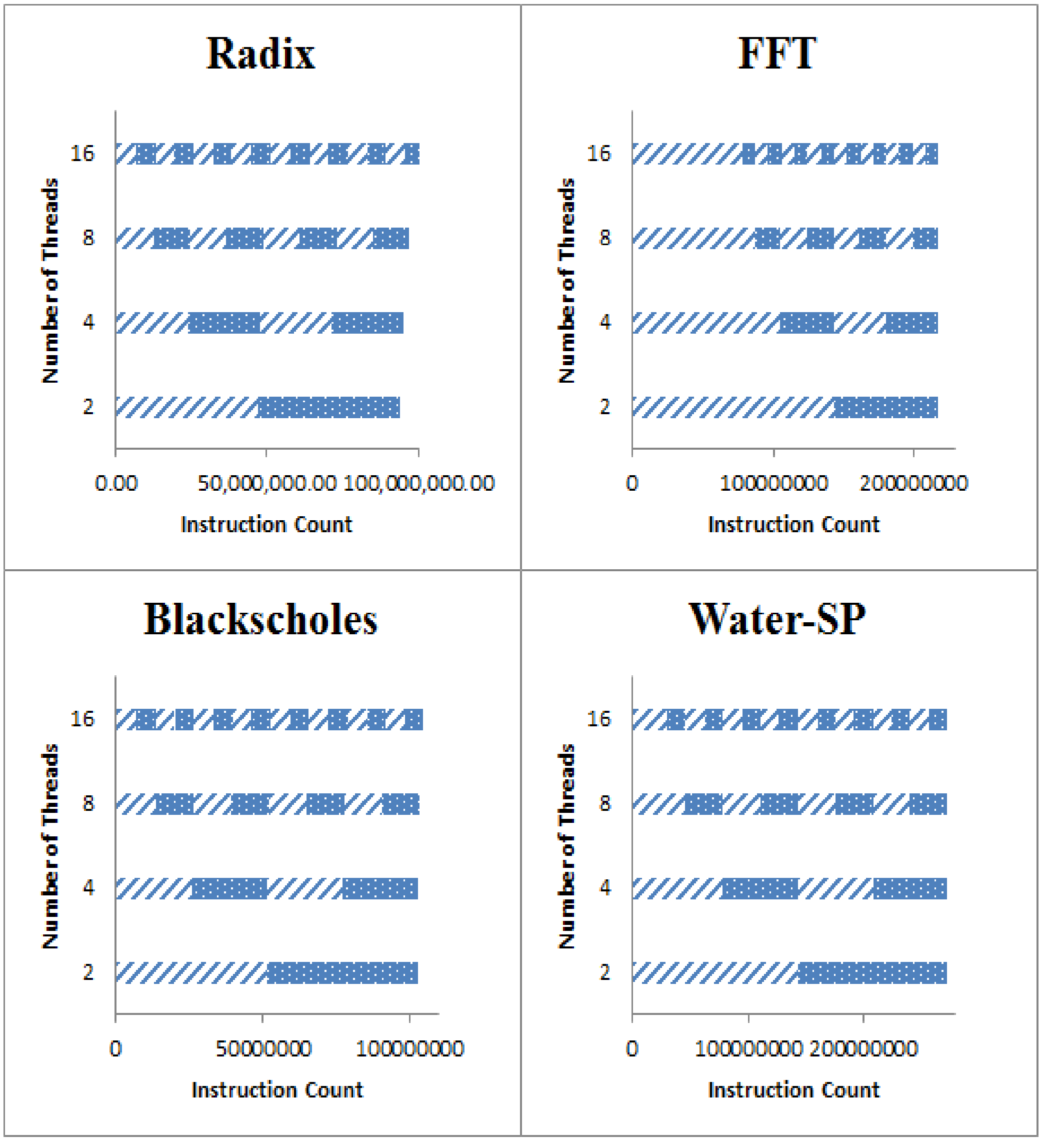
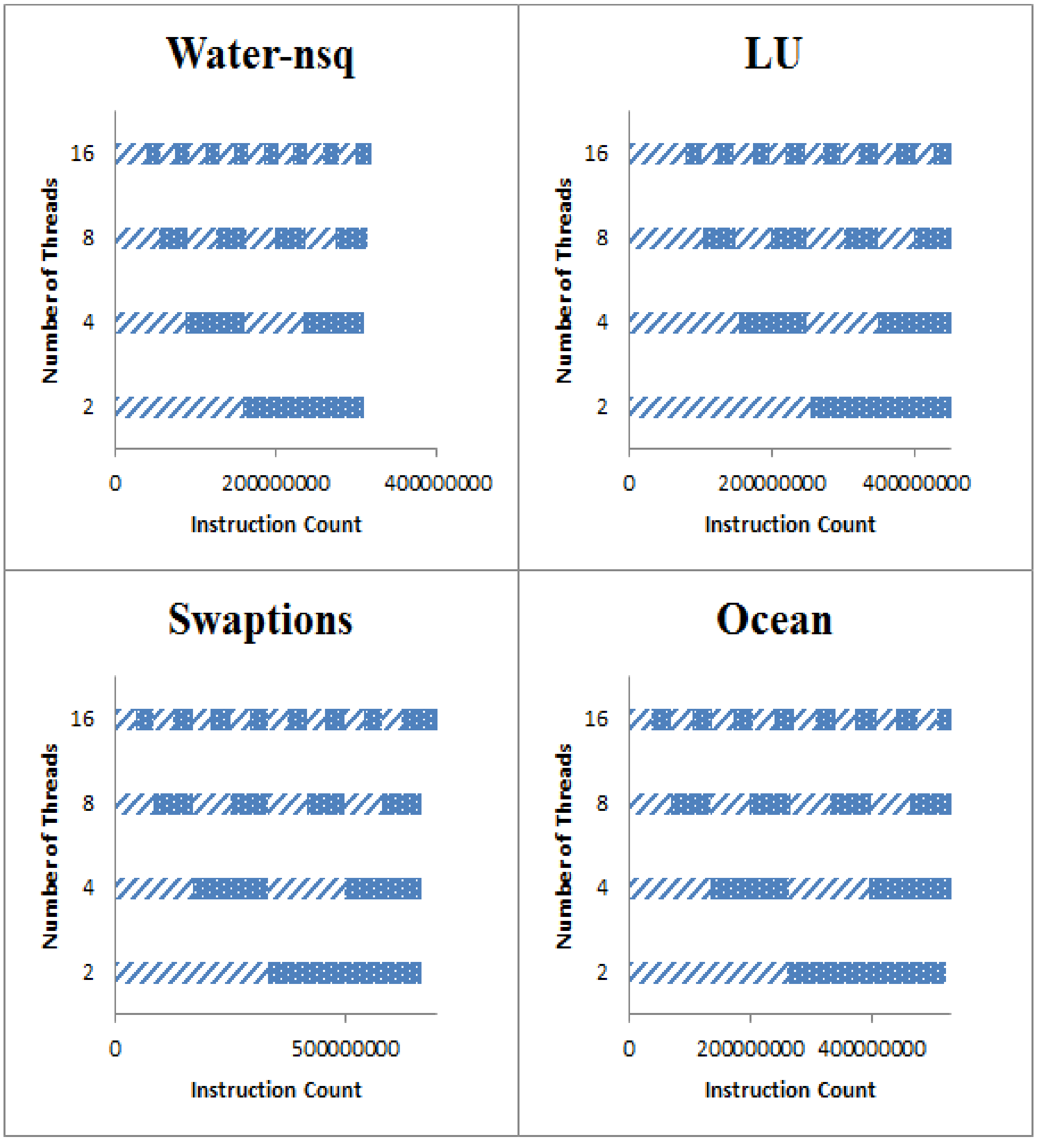


Figure 39: The Level of Parallelism for the Used Multi-threaded Benchmarks (I)



Water-nsq

LU

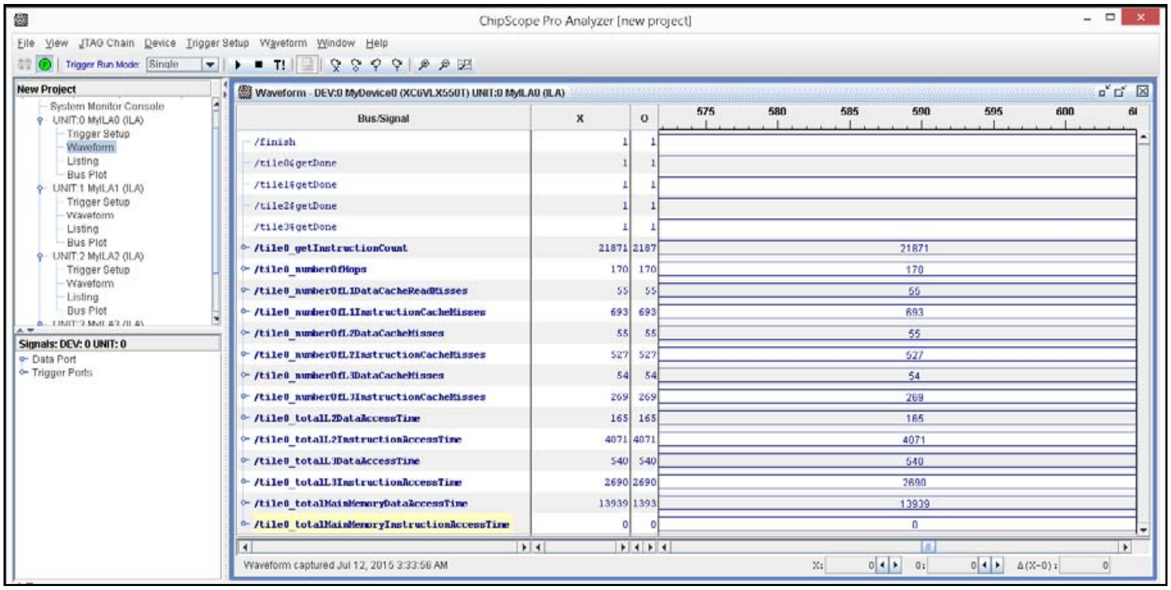
Swaptions

Ocean

Figure 40: The Level of Parallelism for the Used Multi-threaded Benchmarks (II)

8.2 Simulation Monitor

As mentioned in chapter 7, HySim FPGA-based timing model has been downloaded onto *Xilinx Virtex 6 XC6VLX550T* FPGA. Xilinx made a set of tools implemented on the FPGA called ***ChipScope*** in order to probe he internal signals of the design on the FPGA. Unfortunately, ChipScope occupies a significant area on the FPGA and lowers the design frequency. Thus, we developed a software monitor as a part of HySim's software frontend to monitor the internal signals and registers of the design that is being run on the FPGA. This software monitor interacts with the FPGA through its IO ports and displays the values of the signals and registers dynamically. ChipScope was used in the beginning to verify the software monitor, i.e., to make sure that this software monitor displays the same values as ChipScope. Figure 41 shows a snapshot of ChipScope. Figure 42, Figure 43, and Figure 44 show snapshots of the FPGA software monitor.

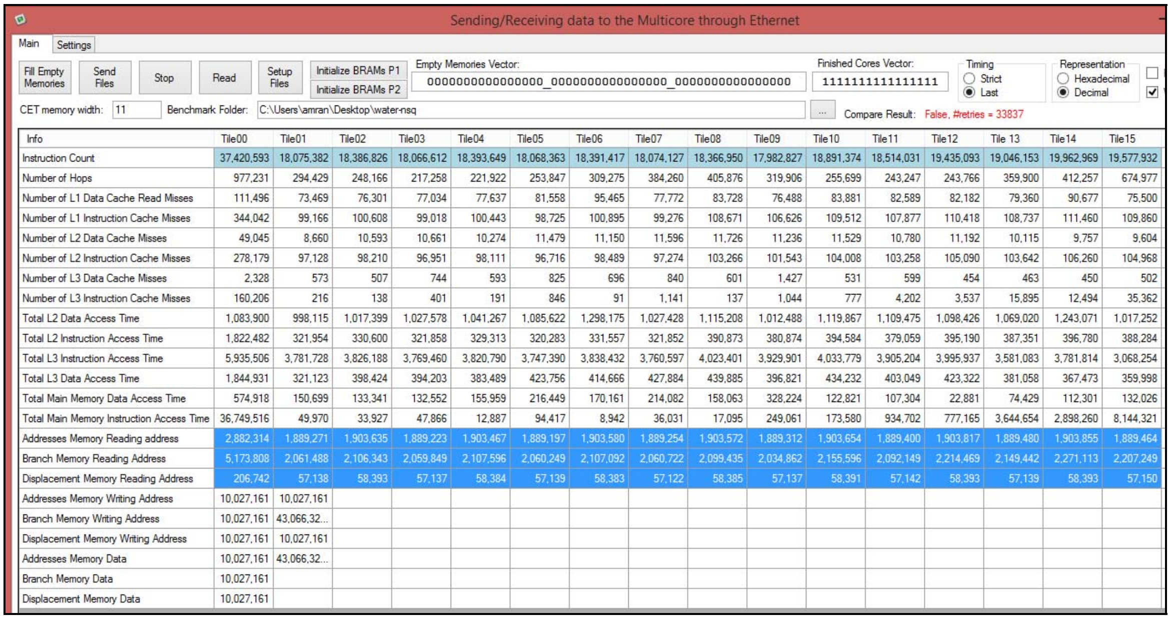


Bus/Signal	X	O	
/finish	1	1	
/tile0$getDone	1	1	
/tile1$getDone	1	1	
/tile2$getDone	1	1	
/tile3$getDone	1	1	
/tile0_getInstructionCount	21871	2187	21871
/tile0_numberOfHops	170	170	170
/tile0_numberOfL1DataCacheReadMisses	55	55	55
/tile0_numberOfL1InstructionCacheMisses	693	693	693
/tile0_numberOfL2DataCacheMisses	55	55	55
/tile0_numberOfL2InstructionCacheMisses	527	527	527
/tile0_numberOfL3DataCacheMisses	54	54	54
/tile0_numberOfL3InstructionCacheMisses	269	269	269
/tile0_totalL2DataAccessTime	165	165	165
/tile0_totalL2InstructionAccessTime	4071	4071	4071
/tile0_totalL3DataAccessTime	540	540	540
/tile0_totalL3InstructionAccessTime	2690	2690	2690
/tile0_totalMainMemoryDataAccessTime	13939	1393	13939
/tile0_totalMainMemoryInstructionAccessTime	0	0	0

Waveform captured Jul 12, 2015 3:33:56 AM

X: 0 O: 0 Δ(X-O): 0

Figure 41: ChipScope Snapshot



Empty Memories Vector: 0000000000000000_0000000000000000_0000000000000000

Finished Cores Vector: 1111111111111111

Timing: ○ Strict ● Last

Representation: ○ Hexadecimal ● Decimal

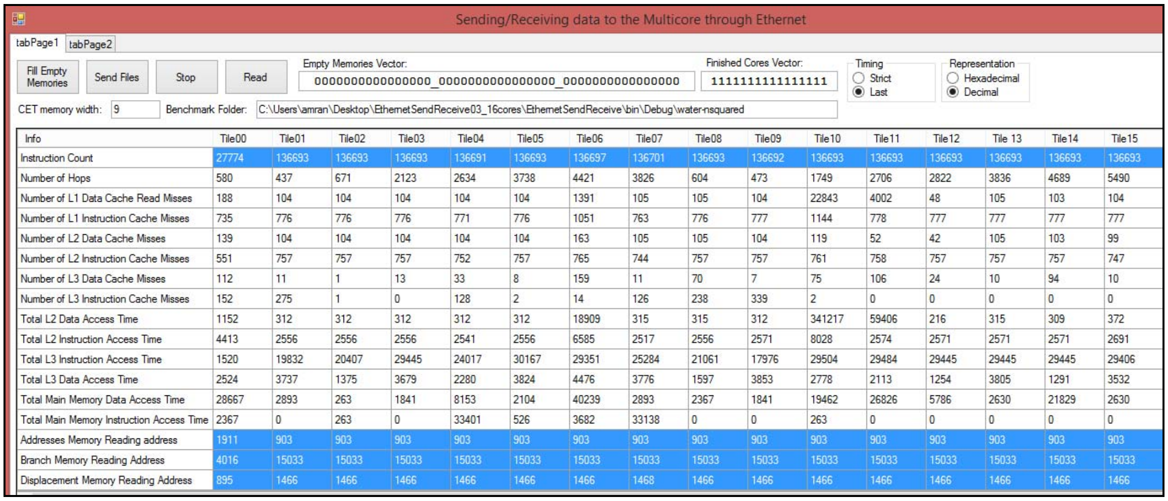
CET memory width: 11

Benchmark Folder: C:\Users\amran\Desktop\water-nsq

Compare Result: False, #retries = 33837

Info	Tile00	Tile01	Tile02	Tile03	Tile04	Tile05	Tile06	Tile07	Tile08	Tile09	Tile10	Tile11	Tile12	Tile 13	Tile14	Tile15
Instruction Count	37,420,593	18,075,382	18,386,826	18,066,612	18,393,649	18,068,363	18,391,417	18,074,127	18,366,950	17,982,827	18,891,374	18,514,031	19,435,093	19,046,153	19,962,969	19,577,932
Number of Hops	977,231	294,429	248,166	217,258	221,922	253,847	309,275	384,260	405,876	319,906	255,699	243,247	243,766	359,900	412,257	674,977
Number of L1 Data Cache Read Misses	111,496	73,469	76,301	77,034	77,637	81,558	95,465	77,772	83,728	76,488	83,881	82,589	82,182	79,360	90,677	75,500
Number of L1 Instruction Cache Misses	344,042	99,166	100,608	99,018	100,443	98,725	100,895	99,276	108,671	106,626	109,512	107,877	110,418	108,737	111,460	109,860
Number of L2 Data Cache Misses	49,045	8,660	10,593	10,661	10,274	11,479	11,150	11,596	11,726	11,236	11,529	10,780	11,192	10,115	9,757	9,604
Number of L2 Instruction Cache Misses	278,179	97,128	98,210	96,951	98,111	96,716	98,489	97,274	103,266	101,543	104,008	103,258	105,090	103,642	106,260	104,968
Number of L3 Data Cache Misses	2,328	573	507	744	593	825	696	840	601	1,427	531	599	454	463	450	502
Number of L3 Instruction Cache Misses	160,206	216	138	401	191	846	91	1,141	137	1,044	777	4,202	3,537	15,895	12,494	35,362
Total L2 Data Access Time	1,083,900	998,115	1,017,399	1,027,578	1,041,267	1,085,622	1,298,175	1,027,428	1,115,208	1,012,488	1,119,867	1,109,475	1,098,426	1,069,020	1,243,071	1,017,252
Total L2 Instruction Access Time	1,822,482	321,954	330,600	321,858	329,313	320,283	331,557	321,852	390,873	380,874	394,584	379,059	395,190	387,351	396,780	388,284
Total L3 Instruction Access Time	5,935,506	3,781,728	3,826,188	3,769,460	3,820,790	3,747,390	3,838,432	3,760,597	4,023,401	3,929,901	4,033,779	3,905,204	3,995,937	3,581,083	3,781,814	3,068,254
Total L3 Data Access Time	1,844,931	321,123	398,424	394,203	383,489	423,756	414,666	427,884	439,885	396,821	434,232	403,049	423,322	381,058	367,473	359,998
Total Main Memory Data Access Time	574,918	150,699	133,341	132,552	155,959	216,449	170,161	214,082	158,063	328,224	122,821	107,304	22,881	74,429	112,301	132,026
Total Main Memory Instruction Access Time	36,749,516	49,970	33,927	47,866	12,887	94,417	8,942	36,031	17,095	249,061	173,580	934,702	777,165	3,644,654	2,898,260	8,144,321
Addresses Memory Reading address	2,882,314	1,889,271	1,903,635	1,889,223	1,903,467	1,889,197	1,903,580	1,889,254	1,903,572	1,889,312	1,903,654	1,889,400	1,903,817	1,889,480	1,903,855	1,889,464
Branch Memory Reading Address	5,173,808	2,061,488	2,106,343	2,059,849	2,107,596	2,060,249	2,107,092	2,060,722	2,099,435	2,034,862	2,155,596	2,092,149	2,214,469	2,149,442	2,271,113	2,207,249
Displacement Memory Reading Address	206,742	57,138	58,393	57,137	58,384	57,139	58,383	57,122	58,385	57,137	58,391	57,142	58,393	57,139	58,393	57,150
Addresses Memory Writing Address	10,027,161	10,027,161														
Branch Memory Writing Address	10,027,161	43,066,32...														
Displacement Memory Writing Address	10,027,161	10,027,161														
Addresses Memory Data	10,027,161	43,066,32...														
Branch Memory Data	10,027,161															
Displacement Memory Data	10,027,161															

Figure 42: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (I)



Empty Memories Vector: 0000000000000000_0000000000000000_0000000000000000

Finished Cores Vector: 1111111111111111

Timing: ○ Strict ● Last

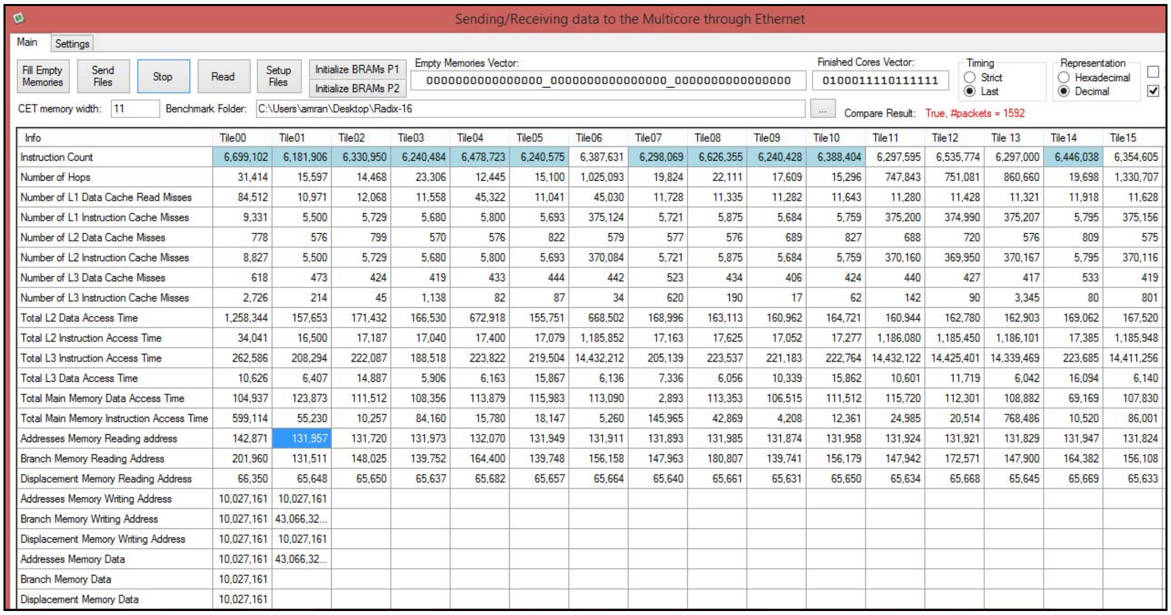
Representation: ○ Hexadecimal ● Decimal

CET memory width: 9

Benchmark Folder: C:\Users\amran\Desktop\EthernetSendReceive03_16cores\EthernetSendReceive\bin\Debug\water-nsquared

Info	Tile00	Tile01	Tile02	Tile03	Tile04	Tile05	Tile06	Tile07	Tile08	Tile09	Tile10	Tile11	Tile12	Tile 13	Tile14	Tile15
Instruction Count	27774	136693	136693	136693	136691	136693	136697	136701	136693	136692	136693	136693	136693	136693	136693	136693
Number of Hops	580	437	671	2123	2634	3738	4421	3826	604	473	1749	2706	2822	3836	4689	5490
Number of L1 Data Cache Read Misses	188	104	104	104	104	104	1391	105	105	104	22843	4002	48	105	103	104
Number of L1 Instruction Cache Misses	735	776	776	776	771	776	1051	763	776	777	1144	778	777	777	777	777
Number of L2 Data Cache Misses	139	104	104	104	104	104	163	105	105	104	119	52	42	105	103	99
Number of L2 Instruction Cache Misses	551	757	757	757	752	757	765	744	757	757	761	758	757	757	757	747
Number of L3 Data Cache Misses	112	11	1	13	33	8	159	11	70	7	75	106	24	10	94	10
Number of L3 Instruction Cache Misses	152	275	1	0	128	2	14	126	238	339	2	0	0	0	0	0
Total L2 Data Access Time	1152	312	312	312	312	312	18909	315	315	312	341217	59406	216	315	309	372
Total L2 Instruction Access Time	4413	2556	2556	2556	2541	2556	6585	2517	2556	2571	8028	2574	2571	2571	2571	2691
Total L3 Instruction Access Time	1520	19832	20407	29445	24017	30167	29351	25284	21061	17976	29504	29484	29445	29445	29445	29406
Total L3 Data Access Time	2524	3737	1375	3679	2280	3824	4476	3776	1597	3853	2778	2113	1254	3805	1291	3532
Total Main Memory Data Access Time	28667	2893	263	1841	8153	2104	40239	2893	2367	1841	19462	26826	5786	2630	21829	2630
Total Main Memory Instruction Access Time	2367	0	263	0	33401	526	3682	33138	0	0	263	0	0	0	0	0
Addresses Memory Reading address	1911	903	903	903	903	903	903	903	903	903	903	903	903	903	903	903
Branch Memory Reading Address	4016	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033	15033
Displacement Memory Reading Address	895	1466	1466	1466	1466	1466	1466	1468	1466	1466	1466	1466	1466	1466	1466	1466

Figure 43: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (II)



Empty Memories Vector: 0000000000000000_0000000000000000_0000000000000000

Finished Cores Vector: 0100011110111111

Timing: ○ Strict ● Last

Representation: ○ Hexadecimal ● Decimal

CET memory width: 11

Benchmark Folder: C:\Users\amran\Desktop\Radix-16

Compare Result: True, #packets = 1592

Info	Tile00	Tile01	Tile02	Tile03	Tile04	Tile05	Tile06	Tile07	Tile08	Tile09	Tile10	Tile11	Tile12	Tile 13	Tile14	Tile15
Instruction Count	6,699,102	6,181,906	6,330,950	6,240,484	6,478,723	6,240,575	6,387,631	6,298,069	6,626,355	6,240,428	6,388,404	6,297,595	6,535,774	6,297,000	6,446,038	6,354,605
Number of Hops	31,414	15,597	14,468	23,306	12,445	15,100	1,025,093	19,824	22,111	17,609	15,296	747,843	751,081	860,660	19,698	1,330,707
Number of L1 Data Cache Read Misses	84,512	10,971	12,068	11,558	45,322	11,041	45,030	11,728	11,335	11,282	11,643	11,280	11,428	11,321	11,918	11,628
Number of L1 Instruction Cache Misses	9,331	5,500	5,729	5,680	5,800	5,693	375,124	5,721	5,875	5,684	5,759	375,200	374,990	375,207	5,795	375,156
Number of L2 Data Cache Misses	778	576	799	570	576	822	579	577	576	689	827	688	720	576	809	575
Number of L2 Instruction Cache Misses	8,827	5,500	5,729	5,680	5,800	5,693	370,084	5,721	5,875	5,684	5,759	370,160	369,950	370,167	5,795	370,116
Number of L3 Data Cache Misses	618	473	424	419	433	444	442	523	434	406	424	440	427	417	533	419
Number of L3 Instruction Cache Misses	2,726	214	45	1,138	82	87	34	620	190	17	62	142	90	3,345	80	801
Total L2 Data Access Time	1,258,344	157,653	171,432	166,530	672,918	155,751	668,502	168,996	163,113	160,962	164,721	160,944	162,780	162,903	169,062	167,520
Total L2 Instruction Access Time	34,041	16,500	17,187	17,040	17,400	17,079	1,185,852	17,163	17,625	17,052	17,277	1,186,080	1,185,450	1,186,101	17,385	1,185,948
Total L3 Instruction Access Time	262,586	208,294	222,087	188,518	223,822	219,504	14,432,212	205,139	223,537	221,183	222,764	14,432,122	14,425,401	14,339,469	223,685	14,411,256
Total L3 Data Access Time	10,626	6,407	14,887	5,906	6,163	15,867	6,136	7,336	6,056	10,339	15,862	10,601	11,719	6,042	16,094	6,140
Total Main Memory Data Access Time	104,937	123,873	111,512	108,356	113,879	115,983	113,090	2,893	113,353	106,515	111,512	115,720	112,301	108,882	69,169	107,830
Total Main Memory Instruction Access Time	599,114	55,230	10,257	84,160	15,780	18,147	5,260	145,965	42,869	4,208	12,361	24,985	20,514	768,486	10,520	86,001
Addresses Memory Reading address	142,871	131,957	131,720	131,973	132,070	131,949	131,911	131,893	131,985	131,874	131,958	131,924	131,921	131,829	131,947	131,824
Branch Memory Reading Address	201,960	131,511	148,025	139,752	164,400	139,748	156,158	147,963	180,807	139,741	156,179	147,942	172,571	147,900	164,382	156,108
Displacement Memory Reading Address	66,350	65,648	65,650	65,637	65,682	65,657	65,664	65,640	65,661	65,631	65,650	65,634	65,668	65,645	65,669	65,633
Addresses Memory Writing Address	10,027,161	10,027,161														
Branch Memory Writing Address	10,027,161	43,066,32...														
Displacement Memory Writing Address	10,027,161	10,027,161														
Addresses Memory Data	10,027,161	43,066,32...														
Branch Memory Data	10,027,161															
Displacement Memory Data	10,027,161															

Figure 44: A Snapshot of HySim Software Frontend Displaying Performance Registers from the FPGA (III)

8.3 Evaluation of Simulation Speed

HySim simulation speed has been expressed in MIPS, which refers to the simulator throughput, i.e., the average number of instructions that can be simulated per second. Equation 1 shows how simulation speed in MIPS in calculated, and equation 2 shows how to calculate the simulation time.

Figure 45 shows HySim's simulation speed in MIPS for different number of threads, namely 1, 2, 4, 8, and 16 threads. For 16 threads, the minimum speed was **380.370 MIPS** for FFT benchmark, the maximum speed is **2204.257 MIPS** for ocean benchmark, and the average speed is **1445.35 MIPS**. The standard deviation of the simulation speed for 16 threads is **732.43 MIPS**. On the other hand, the maximum speed achieved by the software simulator counterpart, namely Sniper [39] is **2 MIPS**.

$$\mathit{Simulation\ Speed\ in\ MIPS} = \frac{\mathit{Total\ Instruction\ Count}}{10^6 \times \mathit{Simulation\ Time}} \quad \text{..................} \quad \textbf{(1)}$$

$$\mathit{Simulation\ Time} = \frac{\mathit{Number\ of\ FPGA\ Cycles}}{\mathit{FPGA\ Frequnecy}} \quad \text{..................} \quad \textbf{(2)}$$

The low MIPS of the multithreaded version of FFT benchmark is interpreted by the lack of load balancing. In 16-threaded FFT version, the number of instructions executed by threads 0 is larger than the number of instructions executed by the worker threads by at least eight times. Figure 45 shows that the simulation speed is doubled by doubling the number of threads for the well balanced benchmarks, such as, ***radix,***

blackscholes, ***oceans***, etc. However, the simulation speed increases slightly by doubling the number of thread for the poorly balanced threads, such as, ***LU*** and ***FFT***. Moreover, HySim simulation speed depends on the size of CET data of the application. Applications with larger CET data are expected to have longer simulation time because more time will be wasted on fetching these data.

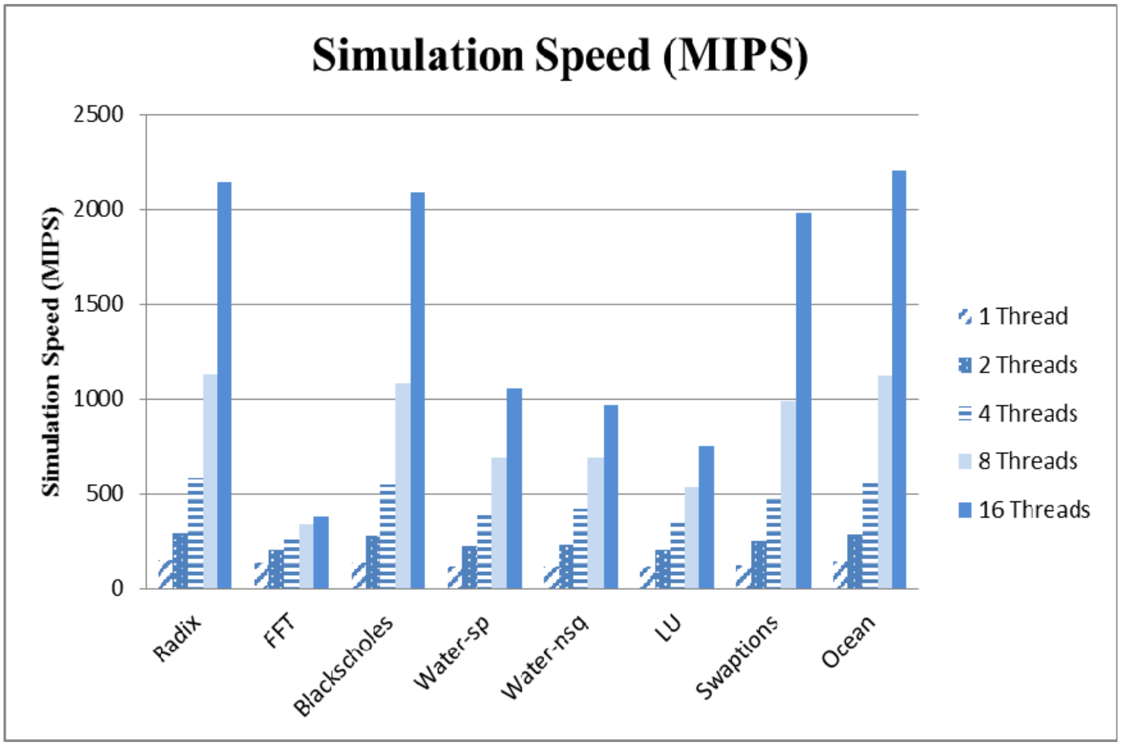


Figure 45: HySim Simulation Speed

In addition to the MIPS metric, HySim simulation speed has been measured as a ratio of the simulation time over the simulated time. Equation 3 shows how the simulated time is calculated. The lower this ratio the faster the simulator because this means that the simulation time is closer to the execution time on the real machine. Table 19 and Table 20 list the simulation and simulated time in seconds and in number of clock cycles for one and sixteen threads, respectively. The average simulation to simulated time ratio for a single-threaded application is **26.27** while it is **7.48** for sixteen threads. This is normal because HySim timing model is parallel, and hence in the multi-threaded version of an application, the workload is divided among the available simulation threads and therefore takes less time to simulate.

$$\mathbf{Simulated\ Time} = \frac{\mathit{Number\ of\ Target Cycles}}{\mathit{Target\ Machine Frequnecy}} \qquad \dots\dots\dots\dots\dots\dots \ \mathbf{(3)}$$

Table 19: Simulation and Simulated Time and Clock Cycles for a Single Thread

Benchmark	No. FPGA Cycles	No. Target Cycles	Ratio	Simulation Time (seconds)	Simulated Time (seconds)	Ratio
Swaptions	911358394	201794465	4.52	5.3609	0.1682	31.88
Blackscholes	125468628	95855246	1.31	0.7381	0.0799	9.24
LU	662625225	84166002	7.87	3.8978	0.0614	63.44
FFT	275808652	86269686	3.20	1.6224	0.0719	22.57
Ocean	621097304	387463764	1.60	3.6535	0.3229	11.32
Radix	108032053	75697089	1.43	0.6355	0.0631	10.07
Water-sp	395483683	91819472	4.31	2.3264	0.0765	30.40
Water-nsq	445211870	100646875	4.42	2.6189	0.0839	31.22
Min			**1.31**			**9.24**
Max			**7.87**			**63.44**
Average			**3.58**			**26.27**

Table 20: Simulation and Simulated Time and Clock Cycles for 16 Threads

Benchmark	No. FPGA Cycles	No. Target Cycles	Ratio	Simulation Time	Simulated Time	Ratio
Swaptions	56968656	27593286	2.06	0.3351	0.0230	14.57
Blackscholes	8539428	63287898	0.13	0.0502	0.0527	0.95
LU	102902102	77895629	1.32	0.6053	0.0649	9.32
FFT	97313098	46269866	2.10	0.5724	0.5724	1.00
Ocean	41751597	41746044	1.00	0.2456	0.0348	7.06
Radix	8099282	15450406	0.52	0.0476	0.0129	3.70
Water-sp	43840261	27638288	1.59	0.2579	0.0230	11.20
Water-nsq	55641136	32615519	1.71	0.3273	0.0272	12.04
Min			**0.13**			**0.95**
Max			**2.10**			**14.57**
Average			**1.31**			**7.48**

8.3.1.1 HySim's Speed Compared to Other Simulators

The FPGA-based simulator, namely HAsim [53] used a simulation speed metric called FMR (**F**PGA-cycle-to-**M**odel-cycle **R**atio) which means the ratio between simulation and simulated time expressed in number of clock cycles. FMR is calculated according to equation 4. This metric tells us the average number of FPGA cycles that is needed to simulate one target cycle (model cycle). Thus, it can be used to measure the simulation speed although the FPGA and the target machine work on two different frequencies, in this metric, the lower the FMR the faster the simulator. HAsim reported the minimum, maximum, and average FMR for a single-core and 16-cores target architectures for a range of SPEC benchmarks. Although we used different benchmarks, we compared our minimum, maximum, and average FMR with HAsim as shown in Table 21.

$$FMR = \frac{Number\ of\ FPGA\ Cycles}{Number\ of\ Model\ Cycles} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad \mathbf{(4)}$$

This table shows that HySim is on average **3.07** times faster than HAsim for a single thread and **61.07** times for sixteen threads. For a single thread, HySim is faster because HAsim is an execution-driven simulator and it has a detailed core model and hence significant part of the simulation time is spent on the core micro-architectural details and on functional execution (computation, data read misses, etc.). Moreover, in 16 threads, HySim outperforms HAsim much more than in the single-thread version. This is because HySim does not use time division multiplexing and hence the sixteen threads will be simulated simultaneously. In contrast, in HAsim, only a number of threads equals to the number of pipeline stages can be active simultaneously.

Table 21: FPGA Cycles to Target Cycles Ratios for HAsim and HySim

	Single Thread			Sixteen Threads		
	Min	**Max**	**Average**	**Min**	**Max**	**Average**
HAsim	5	27	11	16	218	80
HySim	1.31	7.87	3.58	0.13	2.1	1.31
HAsim/HySim Ratio	3.82	3.43	3.07	123.08	103.81	61.07

HAsim’s performance was also reported as the number of target cycles that can be simulated in one second. They referred to it as the simulation rate and it is measured in **hertz.** They reported the minimum, maximum, and average values. This simulation rate has been converted to MIPS assuming that that target architecture completes an average of one instruction per one clock cycle. Since HAsim used Virtex 5 LX330T FPGA, we synthesized our design on this FPGA in addition to the Virtex 6 one. The frequency of HySim on Virtex 5 FPGA was ~137 MHz. Table 22 shows that HySim is much faster than HAsim, especially for the 16 threads version. Again, HySim outperforms HAsim because HySim does not use time division multiplexing, it does not have a detailed core mode, and the functional part is executed prior to timing simulation.

Table 22: HySim's Simulation Speed in MIPS Compared to HASim

	Single Thread			Sixteen Threads		
	Min	**Max**	**Average**	**Min**	**Max**	**Average**
HySim (Virtex 6 FPGA)	114.33	146.60	129.25	380.37	2204.26	1445.35
HAsim	1.84	9.5	4.54	0.16	3.2	0.625
HySim/HAsim Ratio	**62.14**	**15.43**	**28.47**	**2,377.31**	**688.83**	**2,312.56**
HySim (Virtex 5 FPGA)	92.14	118.14	104.16	306.53	1776.37	1164.78
HAsim	1.84	9.5	4.54	0.16	3.2	0.625
HySim/HAsim Ratio	**50.08**	**12.44**	**22.94**	**1,915.81**	**555.16**	**1,863.65**

In HAsim [53], the authors did not report accuracy results and they didn't claim cycle- accuracy. However, Arete [9] FPGA-based simulator was claimed as a cycle accurate simulator. Although Arete is more accurate than HySim, it is much slower. Arete speed was up to 11 MIPS for a single thread and an average of 55 MIPS for eight threads. On the other hand, HySim has a maximum speed of 118.14 MIPS for a single thread and an average speed of 663.23 MIPS for eight threads when it was synthesized on a Virtex 5 FPGA. Moreover, Arete is much more expensive than HySim in terms of FPGA resources. In Arete, two PowerPC core models require an entire Virtex 5 FPGA. This is because Arete is an execution-driven full system simulator and hence it requires a plenty of FPGA resources to have a realistic model of the target architecture.

RAM Gold [46] is another FPGA-based simulator. It simulated a target machine of 64 cores at almost 50 MIPS. On the other hand, HySim's average speed was 1445.35 MIPS when it simulated 16 cores. As we noticed before, HySim's speed in MIPS increases by increasing the number of target cores. Thus, it is expected to increase by at most four times when the target architecture is extended to 64 cores. Although RAM Gold is much slower than HySim, it sacrifices a degree of accuracy. The NoC model and the cache coherence are missing from RAMP Gold.

8.4 Evaluation of Simulation Accuracy

8.4.1 Absolute Accuracy Relative to Real Hardware

Figure 46 shows HySim’s absolute accuracy relative to the average real hardware execution time. The black bold horizontal line in this figure represents the average application-level hardware execution time. This figure shows that the execution time predicted by HySim is in agreement with the average real hardware execution time. The average absolute error for one and 16 threads is **14%** with standard deviation **7.5%** and **8%**, respectively.

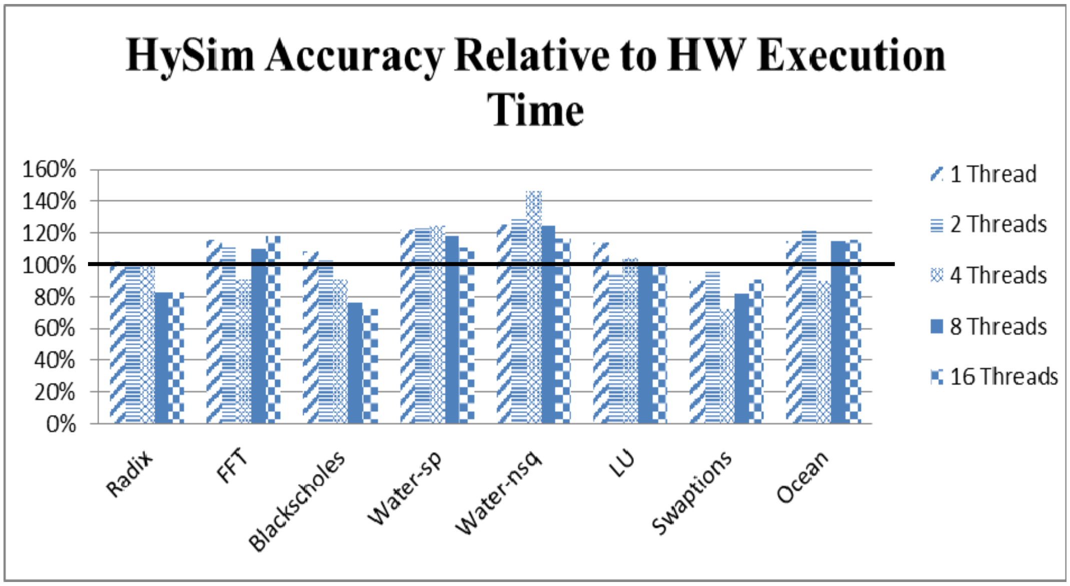


Figure 46: HySim Absolute Accuracy Relative to Real Hardware (Application Level)

Figure 47 and Figure 48 show HySim's simulated time relative to the minimum and maximum real hardware execution time. This figure shows the amount of variation in the measured hardware execution time for the 100 successive runs. For some cases, the maximum execution time is almost twice the minimum one. However, in almost all cases, HySim simulated time falls within the range of the measured hardware execution time.

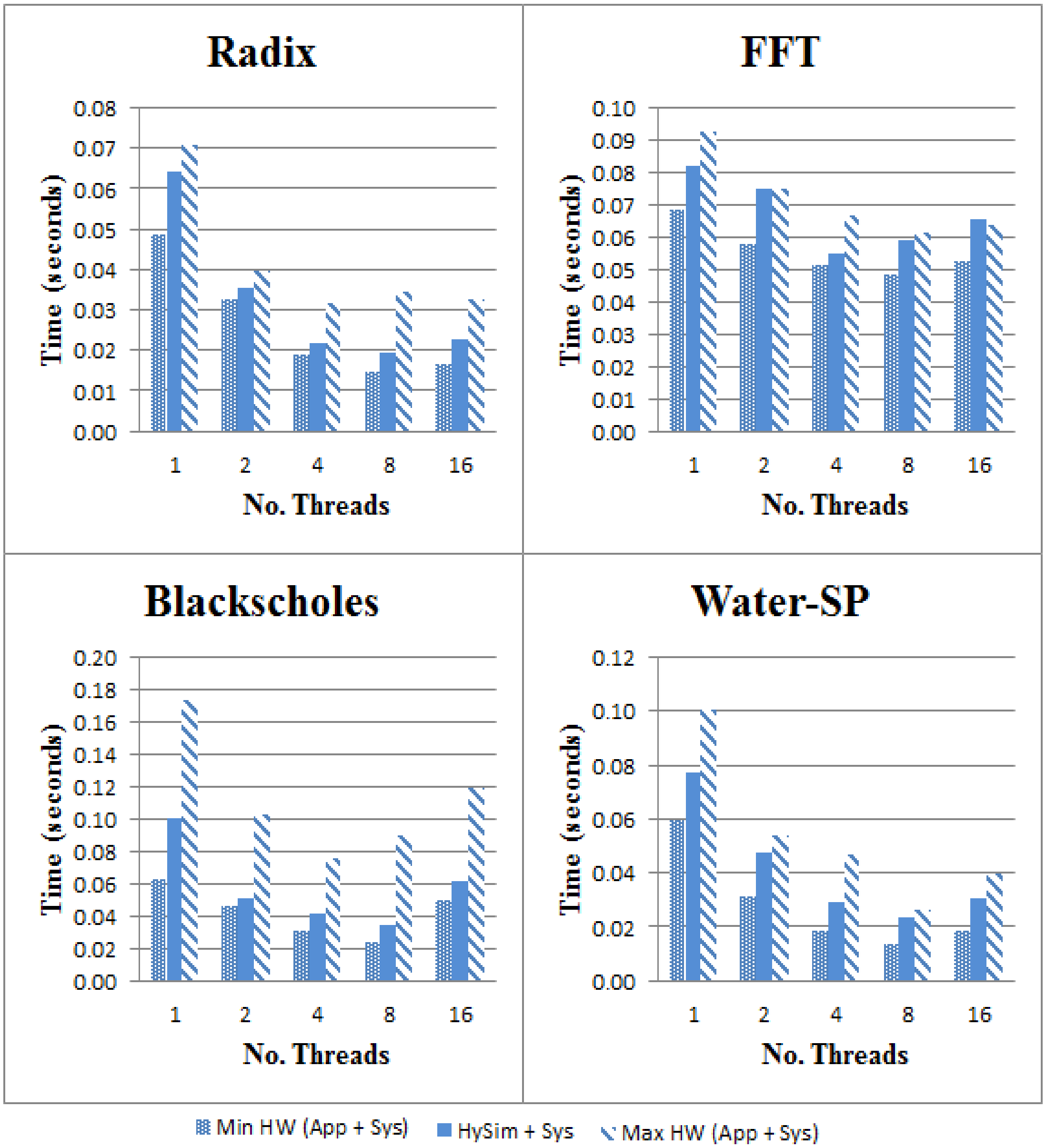


Figure 47: Simulated Time (HySim + Sys) Relative to the Min and Max Total Hardware Execution Time (I)

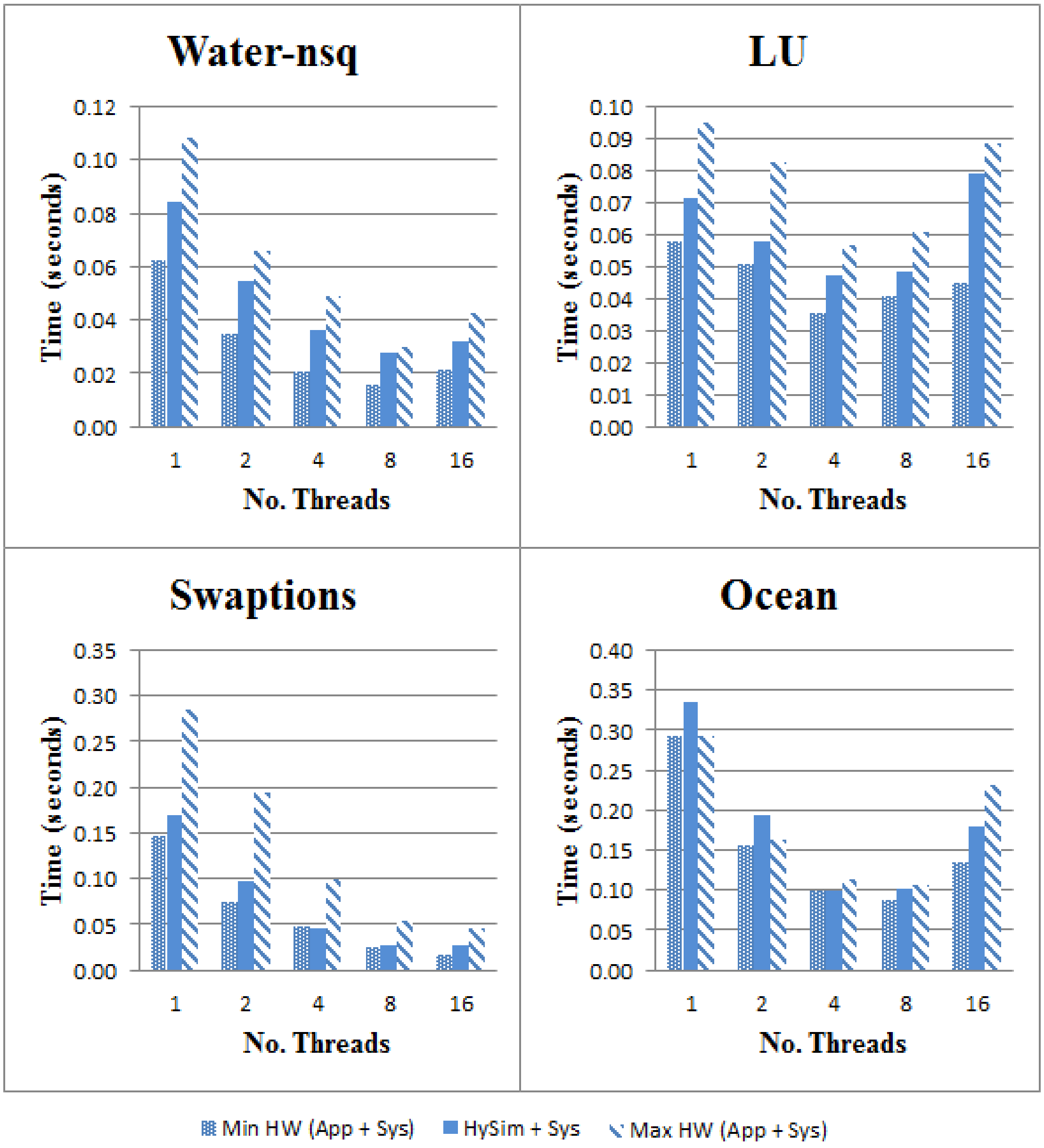


Figure 48: Simulated Time (HySim + Sys) Relative to the Min and Max Total Hardware Execution Time (II)

Table 23 lists the absolute error values for one and sixteen threads of HySim, Interval (Sniper), and one-IPC models Although HySim and Sniper simulated different target architectures. It shows that HySim has the smallest average absolute error.

This table shows that HySim has a better average absolute error than Sniper although both simulators nearly have the same level of abstraction. On the other hand, Sniper has better absolute accuracy than HySim for some benchmarks as shown in this table. Moreover, Sniper has 100% accuracy for some cases although it has a high level of abstraction, which looks weird at first glance. These observations can be interpreted by the fact that these reported error values are relative to the average measured execution time. We had 100 runs and Sniper had 30 runs. As we have seen before, the measured execution time can vary for the successive runs. Therefore, we reported HySim's time, the minimum and maximum measured hardware execution time.

In addition to that, for absolute error computation, we subtracted the system time from the measured hardware execution time to ensure apples-to-apples comparison because HySim is an application-level simulator. However, in Sniper, they didn't mention if their measured time includes the system time or not.

Table 23: HySim Accuracy Relative to Interval and One-IPC Models

Benchmark	**1 Thread Absolute Error Relative to Hardware (%)**			**16 Threads Absolute Error Relative to Hardware (%)**		
	HySiM	**Interval (Sniper) [39]**	**One-IPC [39]**	**HySiM**	**Interval (Sniper) [39]**	**One-IPC [39]**
LU	14	30	290	19	15	140
FFT	16	0	310	18	0	280
Ocean	15	0	290	15	20	190
Radix	2	10	25	17	50	60
Water-sp	17	3	90	11	15	70
Water-nsq	26	90	110	17	50	130
Average	**15.00**	**22.17**	**185.83**	**16.17**	**25**	**145.00**

McSimA+ is a many-core software simulator with detailed microarchitecture modeling. Table 24 compares HySim accuracy with McSimA+ for five Splash-2 benchmarks. For this set of benchmarks, McSimA+ looks a little bit more accurate due to the detailed microarchitecture model, although they reported an average absolute error of 14.2% for a larger set of benchmarks. However, they did not report McSimA+ speed which is expected to be much lower than HySim’s speed because McSimA+ is a pure software simulator and it has a detailed microarchitecture model.

Table 24: Comparison between HySim and McSimA+ Accuracy

Benchmark	**1 Thread Absolute Error Relative to Hardware (%)**	
	HySiM	**McSimA+**
LU	14	5
FFT	16	0
Ocean	15	8
Radix	2	7
Water-sp	17	20
Average	**12.8**	**8**

8.4.2 Speedup Accuracy

Speedup accuracy is another metric; it shows the capability of the simulator to capture the performance trend of a certain application on a certain machine. Speedup in this context is defined as the execution time of the single-threaded version of the application divided on the execution time of the multi-threaded version. In other words, it is sequential time divided by the parallel time. Running the application using different number of threads and then calculating speedup is an important experiment to the computer architects and software designers. It tells how scalable the application on a specific machine is. To see how much HySim is accurate in detecting the performance trend of an application on a certain machine, we computed the speedup using the measured real hardware execution time and the execution time derived by HySim.

Figure 49 and Figure 50compares between the speedup on the real hardware and the speedup measured by HySim for application-level code. Moreover, Figure 51 and Figure 52 show the same thing but for system-level code. We noticed that HySim nearly detected the speedup curve in most cases.

The low speedup for some benchmarks such as FFT and LU can be interpreted by the load unbalance. Moreover, in the majority of the benchmarks, the speedup drops when the number of threads is increased to sixteen. This is normal because the sixteen threads will be distributed across the two sockets of the processor and hence the coherence transactions will incur longer delays, furthermore, the larger the number of threads the longer the waiting time on the barriers.

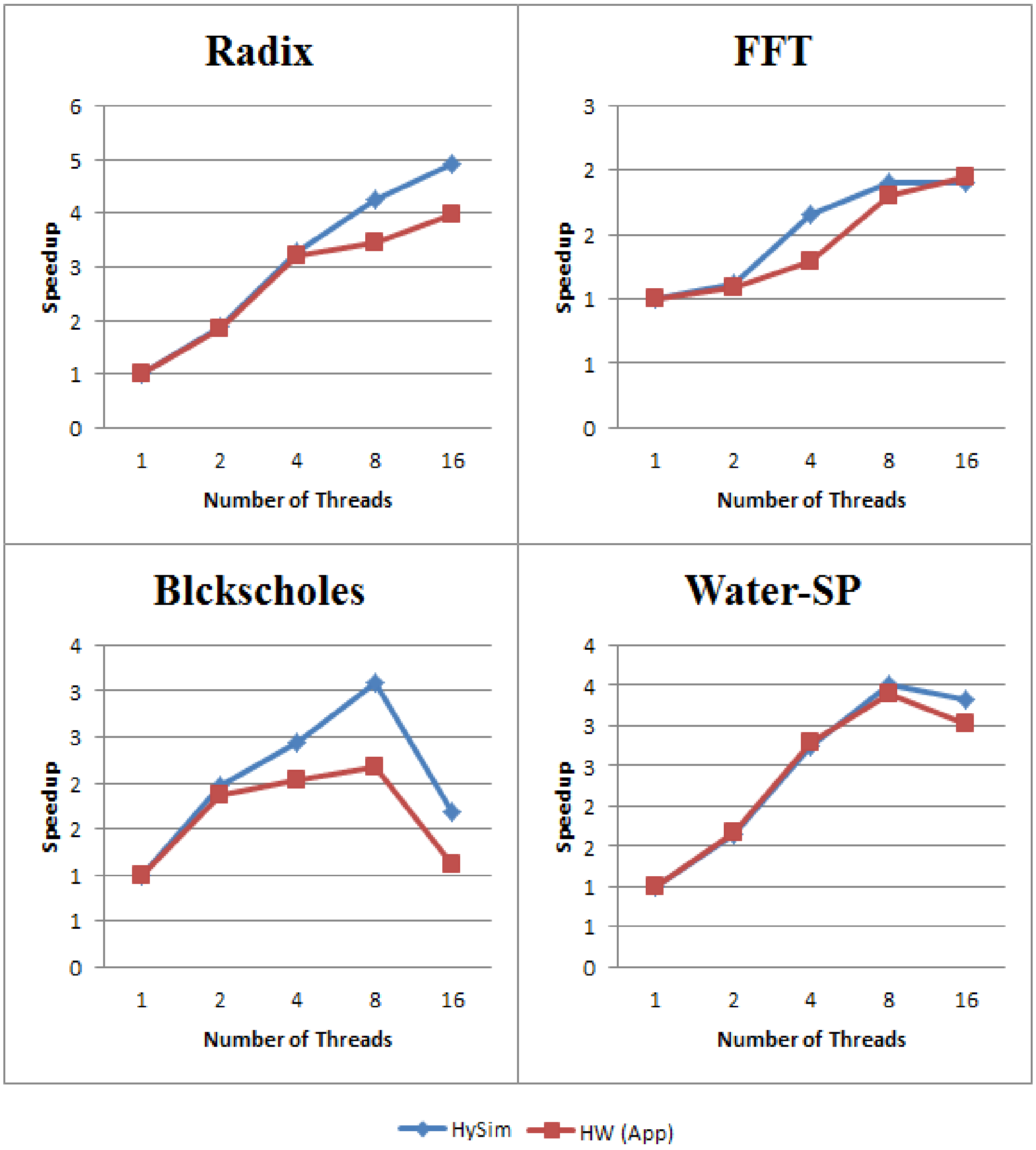


Figure 49: Simulated Speedup Accuracy Relative to Real HW Speedup (Application Level) (I)

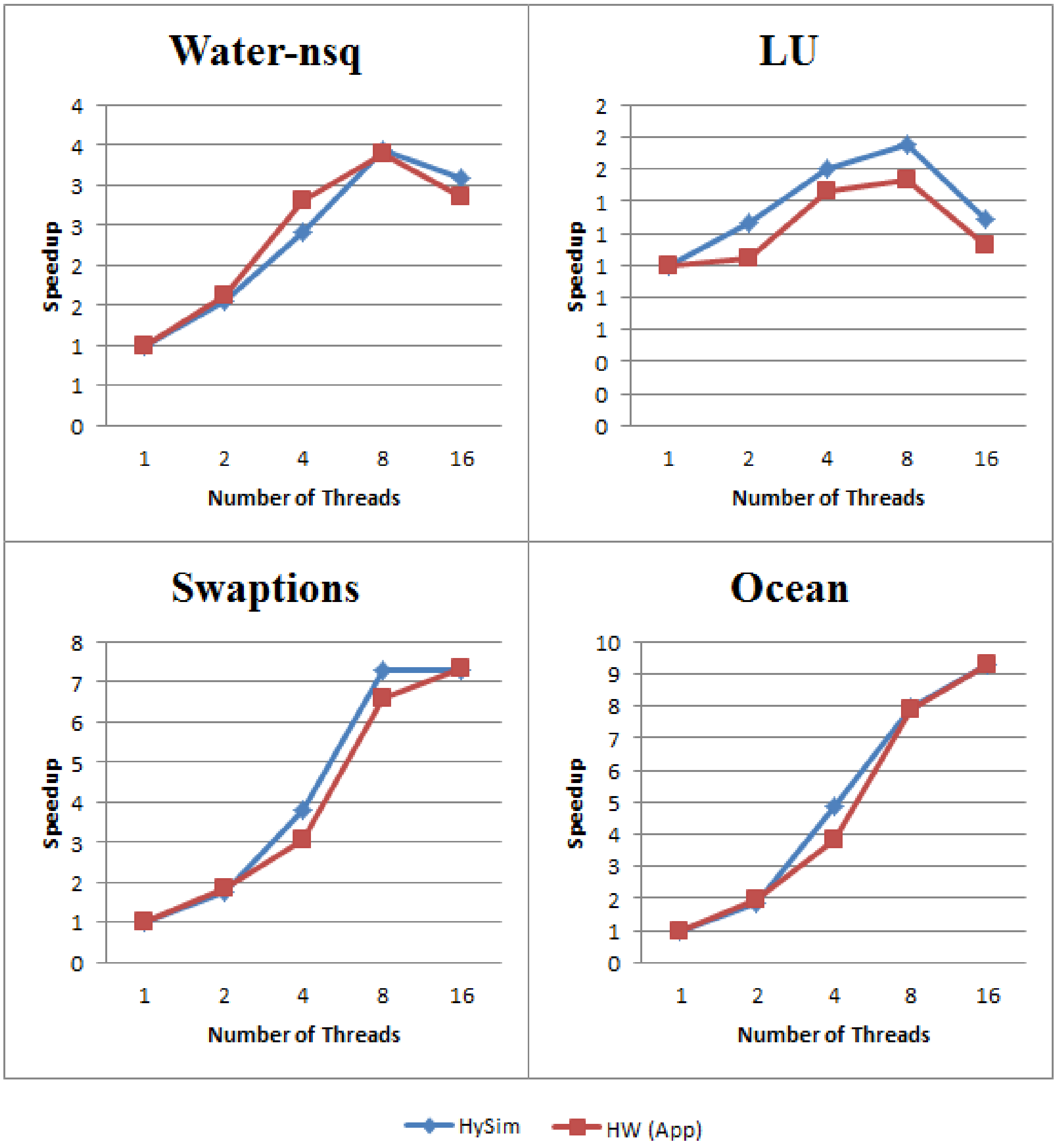


Figure 50: Simulated Speedup Accuracy Relative to Real HW Speedup (Application Level) (II)

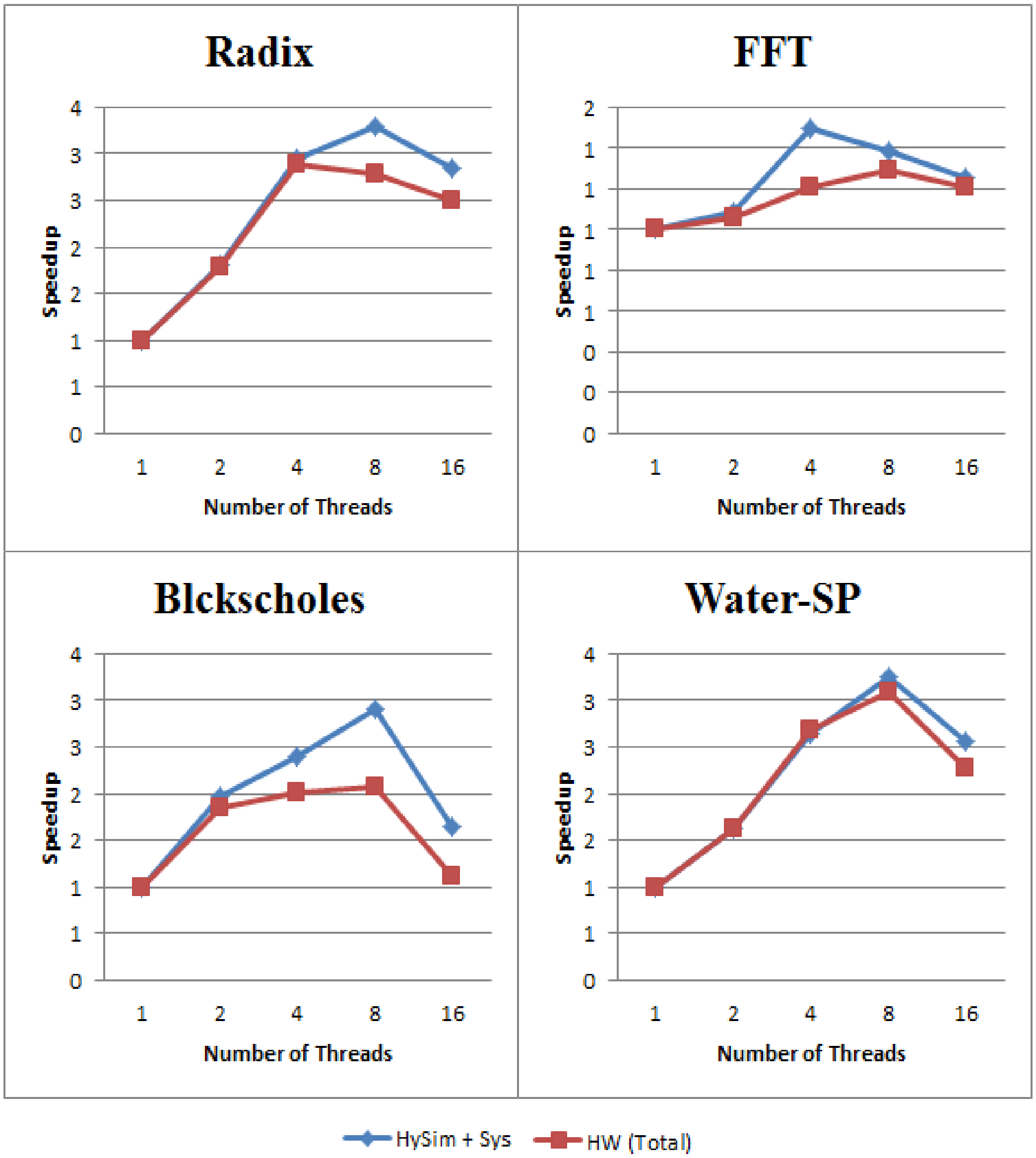


Figure 51: Simulated Speedup Accuracy Relative to Real HW Speedup (System Level) (I)

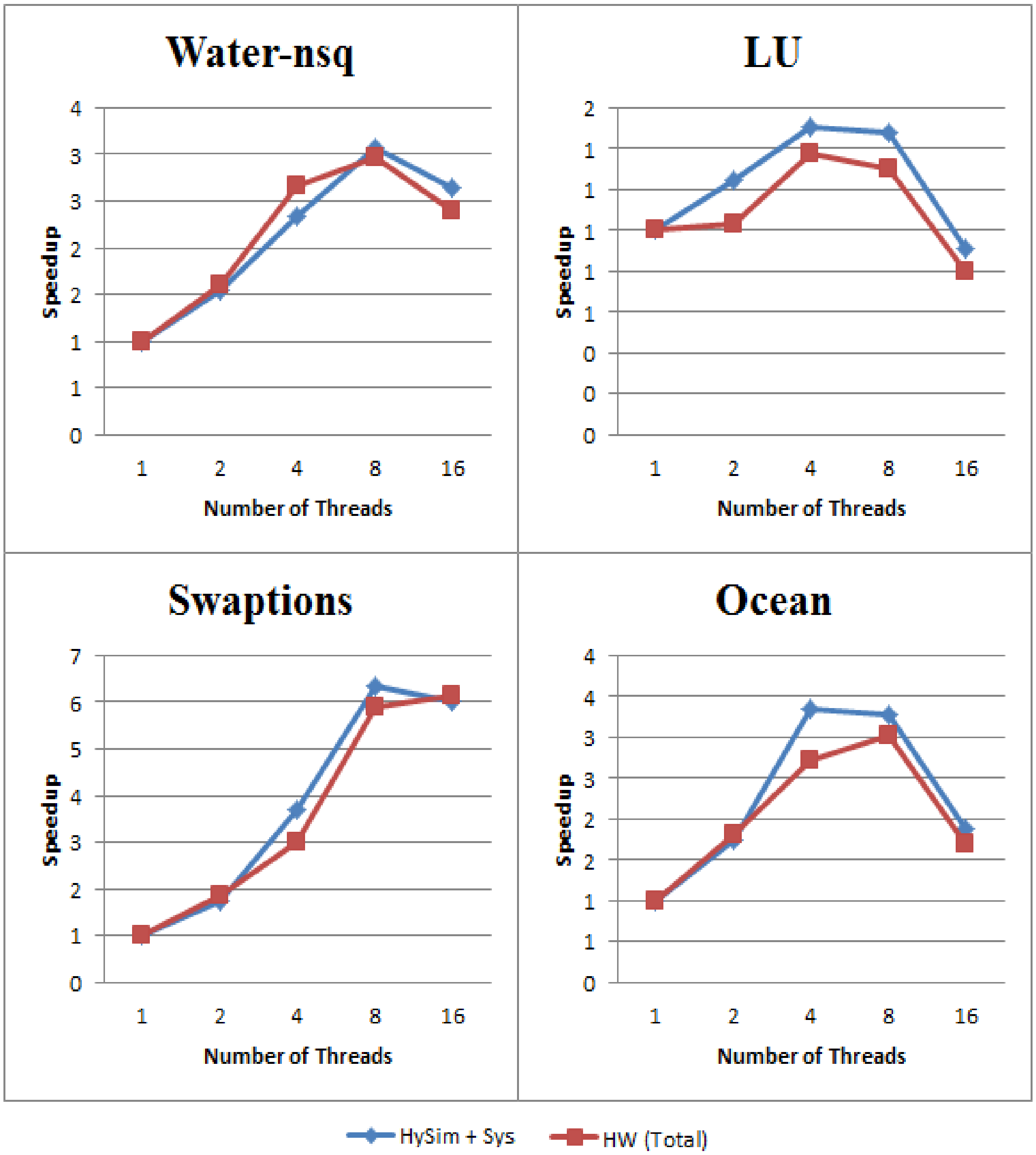


Figure 52: Simulated Speedup Accuracy Relative to Real HW Speedup (System Level) (II)

8.4.3 Base CPI Effect on Speedup

Since the base CPI is a tunable parameter, improper values of this parameter are expected to affect the absolute simulation accuracy. We measured the effect of base CPI variations on the speedup. We made a ±25% variation in the base CPI values. Then, we calculated the speedup for the three values of CPI (original CPI, CPI+, and CPI-). Figure 53 and Figure 54 show that the speedup is nearly constant when the base CPI is changed because it is changed by a constant value for each number of threads.

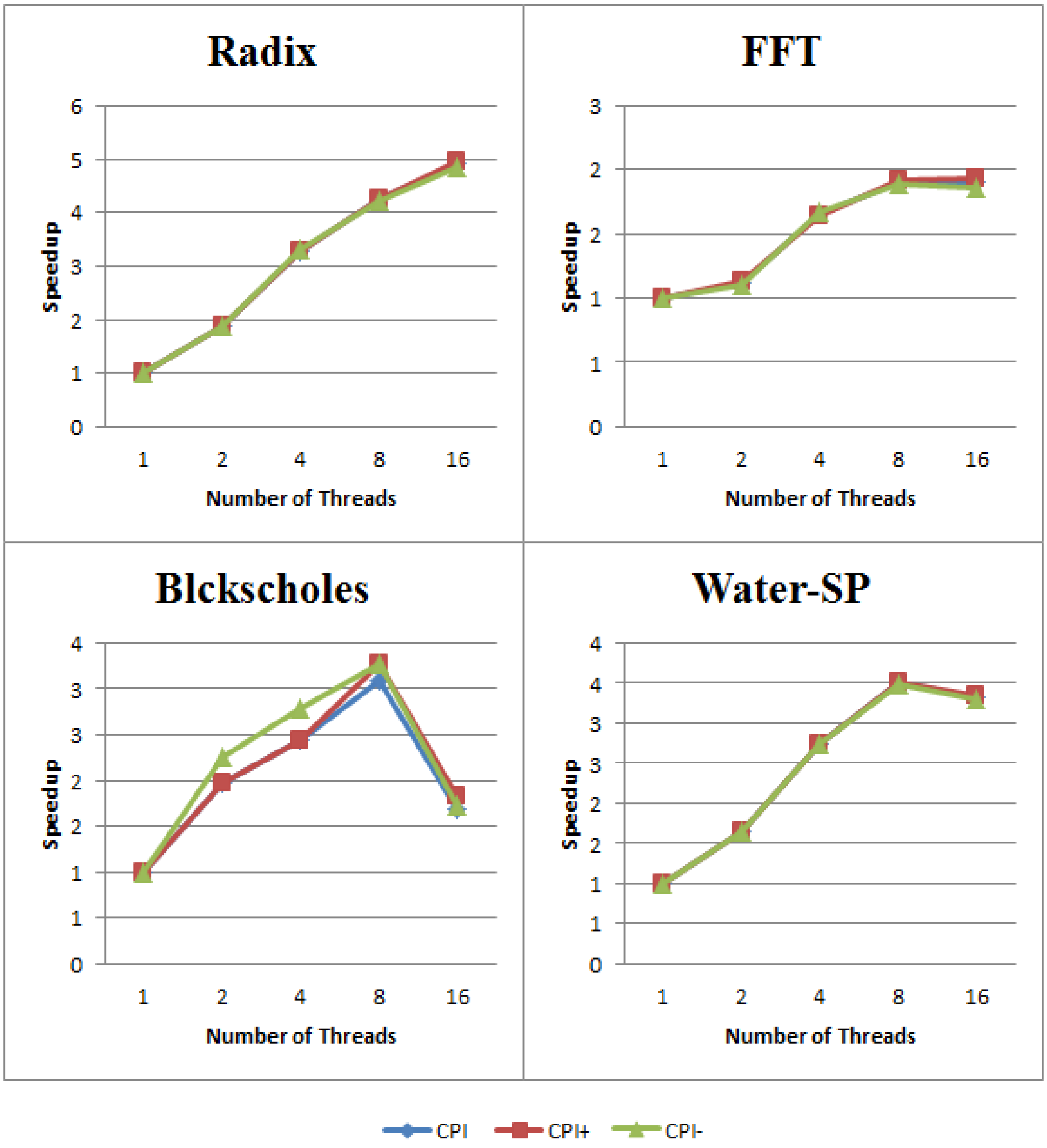


Figure 53: CPI Effect on Speedup (Application Level) (I)

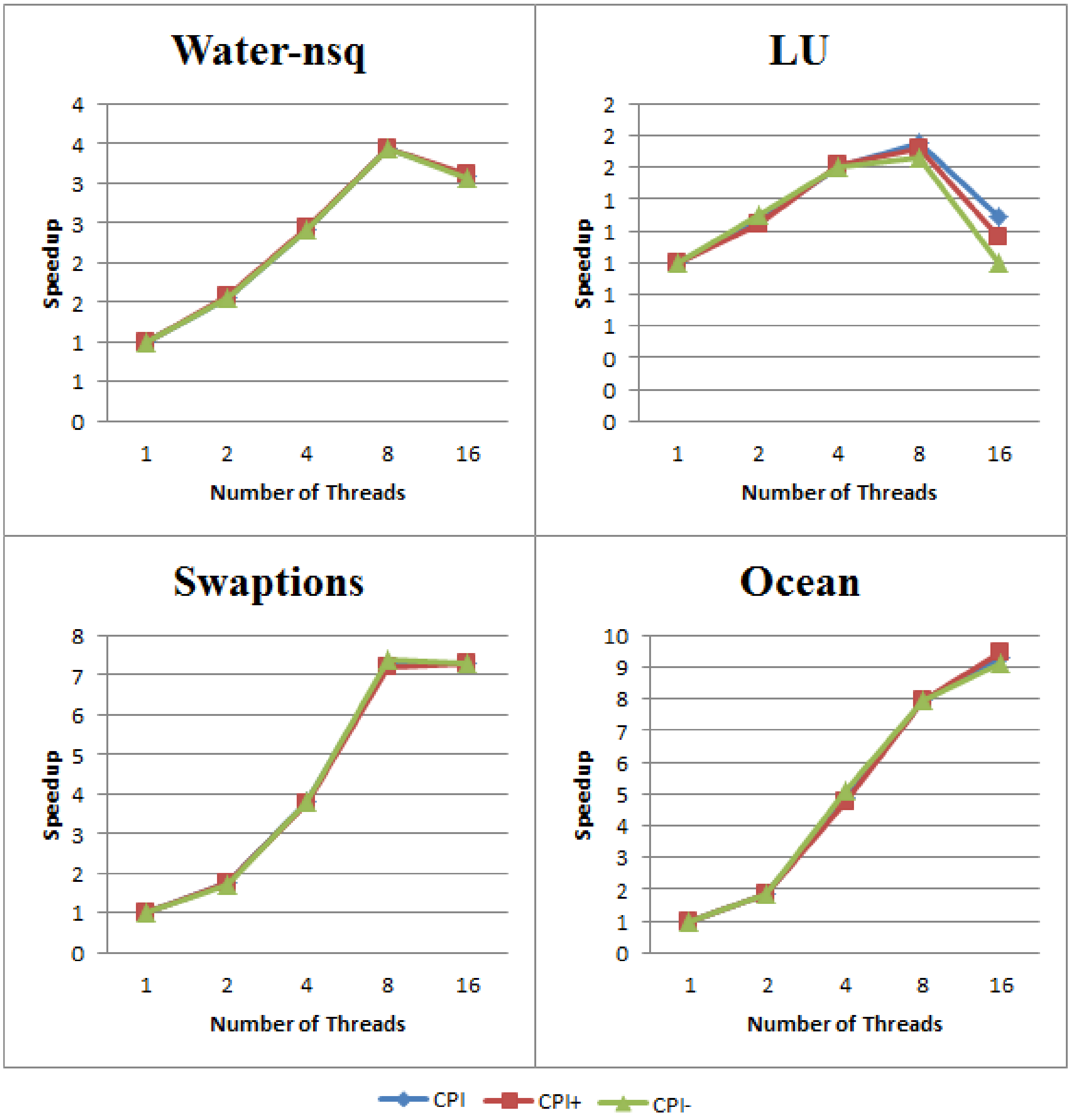


Figure 54: CPI Effect on Speedup (Application Level) (II)

8.5 Limitations

Although HySim is a very fast simulator with acceptable accuracy, it has the following limitations:

1. HySim is a user-level simulator. Therefore, it is not reliable for workloads of significant system-level code.
2. The base-CPI is a tunable parameter and hence it is a major source of error.
3. The NoC model is very simple. It only counts the number of hops traversed by the network message and then calculates the message latency by multiplying the number of hops by the hop latency, which is a tunable parameter. This model will work fine for simple un-buffered NoCs.
4. The reported accuracy is relative to real hardware execution time which has some uncertainty.

CHAPTER 9

CONCLUSION AND FUTURE WORK

In this dissertation, we studied the existing computer architecture simulation techniques and the major recent computer architecture simulators. Based on this, we proposed HySim, a hybrid software/hardware simulation framework for CMPs. We exploited Intel pin tool to natively execute and instrument the application to be run on the target machine model in order to generate a compressed executable trace of this application. The proposed trace compression technique achieved a compression ratio of up to **2987.9**. The trace compression is done on-the-fly, i.e., the trace is compressed while it is being generated and hence the original large trace is never stored as is.

Moreover, we exploited the fine and coarse grained parallelism offered by FPGAs to accelerate computer architecture timing simulation. In other words, the voluminous number of fine-grained parallel components of a CMP model has been simulated in parallel. Thus, HySim is the fastest existing simulator with a speed of up to **2204.257 MIPS** for 16-core target architecture. Although HySim is not a cycle-accurate simulator, its accuracy is in agreement with the majority of the existing simulators. When HySim accuracy has been validated against real hardware, the average absolute error was **~14%.**

This work can be extended in many ways. It opened the doors for many contributions. One of these extensions is to make this framework closer to the fully-usable one. This implies exploiting the reaming free resources on the FPGA to have a more generic model. In addition to that, the design can be extended and downloaded on multiple FPGAs to simulate larger target architectures. Other extensions include having an open source pool of architecture components that are used for building different target architectures. Moreover, HySim accuracy can be improved by preserving the precedence of memory operations.

REFERENCES

[1] I. C. George Chrysos. (2012). *Intel® Xeon Phi™ Coprocessor - the Architecture*. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[2] *Intel® Xeon® Processor E5-2680 (20M Cache, 2.70 GHz, 8.00 GT/s Intel® QPI)*. Available: http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI

[3] "Telira."

[4] U. o. MarkD. Hill, Ed., *Computer Architecture Performance Evaluation Methods* (SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE. Morgan & Claypool, 2010, p.^pp. Pages.

[5] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer,* vol. 35, pp. 59-67, 2002.

[6] D. C. Hari Angepat, Eric S. Chung, James C. Hoe, *FPGA-Accelerated Simulation of Computer Systems*, 2014.

[7] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, 2006, pp. 29-40.

[8] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovi, and D. Patterson, "A case for FAME: FPGA architecture model execution," *SIGARCH Comput. Archit. News,* vol. 38, pp. 290-301, 2010.

[9] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind, "Fast and cycle-accurate modeling of a multicore processor," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, 2012, pp. 178-187.

[10] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and M. Ken, "PROToFLEX: FPGA-accelerated Hybrid Functional Simulator," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1-6.

[11] D. Chiou, S. Dam, K. Joonsoo, N. Patil, W. H. Reinhart, D. E. Johnson, and X. Zheng, "The FAST methodology for high-speed SoC/computer simulation," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, 2007, pp. 295-302.

[12] *Xilinx*. Available: http://www.xilinx.com/

[13] *Altera*. Available: https://www.altera.com/

[14] S. B. (Intel). (2012). *Pin - A Dynamic Binary Instrumentation Tool*. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[15] (2013). *Telira*. Available: http://www.tilera.com/

[16] Borkar; G. Ruhl; S. Dighe, "The 48-core SCC processor: the programmer's view," 2010.

[17] S. Secchi, M. Ceriani, A. Tumeo, O. Villa, G. Palermo, and L. Raffo, "Exploring hardware support for scaling irregular applications on multi-node multi-core architectures," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, 2013, pp. 309-313.

[18] R. K. B. S. W. J. S. M. Floyd, "POWER7: IBM'S NEXT-GENERATION SERVER PROCESSOR," *IEEE Computer Society*, pp. 7-15, 2010.

[19] ARM. (2011-2012). *Cortex-A15 MPCore Technical Reference Manual*.

[20] "AMD Phenom™ II Processors."

[21] K. H. T. W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," *FPGA'10*, February 21–23 2010.

[22] AMD. (2009). *Key Architectural Features of AMD Phenom™ X4 Quad-Core Processors* Available: http://www.amd.com/us/products/desktop/processors/phenom/Pages/AMD-phenom-processor-X4-features.aspx

[23] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, pp. 108-116, 2002.

[24] *The gem5 Simulator*. Available: http://gem5.org/Main_Page

[25] J. Emer, P. Ahuja, E. Borch, A. Klauser, L. Chi-Keung, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: a performance model framework," *Computer*, vol. 35, pp. 68-76, 2002.

[26] E. Argollo, A. Falc, #243, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52-61, 2009.

[27] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, and S. Cho, "Two-phase trace-driven simulation (TPTS): a fast multicore processor architecture simulation approach," *Softw. Pract. Exper.*, vol. 40, pp. 239-258, 2010.

[28] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, "Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015, pp. 278-287.

[29] C. A. Prete, G. Prina, and L. Ricciardi, "A trace-driven simulator for performance evaluation of cache-based multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6, pp. 915-929, 1995.

[30] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, 2011, pp. 87-96.

[31] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones, "A trace-driven approach for fast and accurate simulation of manycore architectures," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, 2015, pp. 707-712.

[32] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam, "The VPC trace-compression algorithms," *Computers, IEEE Transactions on,* vol. 54, pp. 1329-1344, 2005.

[33] C.-J. K. A. T.-J. L. CHING-WEN CHEN, "Efficient Trace File Compression Design with Locality and Address Difference," *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING,* pp. 1055-1070, 2013.

[34] E. N. Elnozahy, "Address trace compression through loop detection and reduction," presented at the Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Atlanta, Georgia, USA, 1999.

[35] E. E. Johnson, H. Jiheng, and M. Baqar Zaidi, "Lossless trace compression," *Computers, IEEE Transactions on,* vol. 50, pp. 158-173, 2001.

[36] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.,* vol. 7, pp. 78-103, 1997.

[37] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," *SIGMETRICS Perform. Eval. Rev.,* vol. 24, pp. 68-79, 1996.

[38] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1-12.

[39] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1-12.

[40] "Valgrind."

[41] A. Jung Ho, L. Sheng, O. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013, pp. 74-85.

[42] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, K. Asanovi\, and \#263, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," presented at the Proceedings of the 47th Design Automation Conference, Anaheim, California, 2010.

[43] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1-12.

[44] W. Jun, J. Beu, R. Bheda, T. Conte, D. Zhenjiang, C. Kersey, M. Rasquinha, G. Riley, W. Song, X. He, X. Peng, and S. Yalamanchili, "Manifold: A parallel simulation framework for multicore systems," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014, pp. 106-115.

[45] G. H. Loh, S. Subramaniam, and X. Yuejian, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Performance Analysis of*

Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009, pp. 53-64.

[46] T. Zhangxi, A. Waterman, R. Avizienis, L. Yunsup, H. Cook, D. Patterson, and K. Asanovic, "RAMP gold: An FPGA-based architecture simulator for multiprocessors," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 463-468.

[47] D. Chiou, S. Dam, K. Joonsoo, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, 2007, pp. 249-261.

[48] F. Zhenman, M. Qinghao, Z. Keyong, L. Yi, H. Yibin, Z. Weihua, C. Haibo, L. Jian, and Z. Binyu, "Transformer: A functional-driven cycle-accurate multicore simulator," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012, pp. 106-114.

[49] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News,* vol. 33, pp. 92-99, 2005.

[50] R. Pengju, M. Lis, C. Myong Hyon, S. Keun Sup, C. W. Fletcher, O. Khan, Z. Nanning, and S. Devadas, "HORNET: A Cycle-Level Multicore Simulator," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 31, pp. 890-903, 2012.

[51] O. Certner, L. Zheng, A. Raman, and O. Temam, "A Very Fast Simulator for Exploring the Many-Core Future," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 443-454.

[52] S. Takamaeda-Yamazaki, S. Sano, Y. Sakaguchi, N. Fujieda, and K. Kise, "ScalableCore System: A Scalable Many-Core Simulator by Employing over 100 FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*. vol. 7199, O. S. Choy, R. C. Cheung, P. Athanas, and K. Sano, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 138-150.

[53] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 406-417.

[54] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations," *Computers, IEEE Transactions on,* vol. 55, pp. 268-280, 2006.

[55] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News,* vol. 39, pp. 1-7, 2011.

[56] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *Micro, IEEE,* vol. 26, pp. 52-60, 2006.

[57] D. R. Butenhof, *Programming with POSIX threads*: Addison-Wesley Longman Publishing Co., Inc., 1997.

[58] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development," *Computer Architecture Letters,* vol. 6, pp. 45-48, 2007.

[59] R. Bedicheck, "SimNow: Fast platform simulation purely in software," in *Hot Chips*.

[60] M. Vijayaraghavan and Arvind, "Bounded Dataflow Networks and Latency-Insensitive circuits," in *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, 2009, pp. 171-180.

[61] 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, pp. 50-58, 2002.

[62] S. Dam, K. Joonsoo, and D. Chiou, "QUICK: A flexible full-system functional model," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 249-258.

[63] *bluespec*. Available: http://www.bluespec.com/

[64] S. Kanev and R. Cohn, "Portable trace compression through instruction interpretation," presented at the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2011.

[65] S. Budanur, F. Mueller, and T. Gamblin, "Memory Trace Compression and Replay for SPMD Systems Using Extended PRSDs," *Comput. J.,* vol. 55, pp. 206-217, 2012.

[66] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo, "METRIC: tracking down inefficiencies in the memory hierarchy via binary rewriting," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, 2003, pp. 289-300.

[67] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. d. Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.,* vol. 69, pp. 696-710, 2009.

[68] A. Janapsatya, A. Ignjatovic, and J. Henkel, "Instruction Trace Compression for Rapid Instruction Cache Simulation," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, 2007, pp. 1-6.

[69] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," presented at the Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, Boston, MA, USA, 2008.

[70] *TCgen 2.0: A Tool to Automatically Generate Lossless Trace Compressors* 2006.

[71] K. C. Barr and K. Asanovic, "Branch trace compression for snapshot-based simulation," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, 2006, pp. 25-36.

[72] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic, "Accelerating Multiprocessor Simulation with a Memory Timestamp Record," in *Performance Analysis of*

Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on, 2005, pp. 66-77.

[73] M. Xu, M. D. Hill, and R. Bodik, "A regulated transitive reduction (RTR) for longer memory race recording," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 49-60, 2006.

[74] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," *SIGARCH Comput. Archit. News*, vol. 31, pp. 122-135, 2003.

[75] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," presented at the Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, Toronto, Ontario, Canada, 2010.

[76] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, 1995, pp. 24-36.

[77] *PARSEC*. Available: http://parsec.cs.princeton.edu/

[78] *MediaBench*. Available: http://euler.slu.edu/~fritts/mediabench/

[79] *Standard Performance Evaluation Corporation*. Available: https://www.spec.org/cpu2000/

[80] A. Milenkovi, and M. Milenkovi, "An efficient single-pass trace compression technique utilizing instruction streams," *ACM Trans. Model. Comput. Simul.*, vol. 17, p. 2, 2007.

[81] R. S. N. a. K. Czeck. (2010). *BSV by Example, The next-generation language for electronic system design*.

[82] Arvind and R. Nikhil, "Hands-on Introduction to Bluespec System Verilog (BSV)," in *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, 2008, pp. 205-206.

[83] B. R. T. Ungerer, and J. Šilc, "Multithreaded Processors," *The Computer Journal*, pp. 320-348, 2002.

[84] (2015). *Stampede Virtual Workshop, Multi-Core Optimization* Available: http://www.cac.cornell.edu/Stampede/CodeOptimization/multicore.aspx

APPENDIX A: SOURCE CODE

This appendix presents an overview of the source code that has been written in this dissertation. It includes the CET tool C++ code and the hardware description code of HySim's timing model in BSV and the corresponding auto-generated Verilog code. Click here for the full source code.

A.1 CET Tool

The CET tool source code comprises nearly 2800 lines of C++ code. In addition to the instrumentation and analysis functions, CET tool has different functions for application profiling, CET code generation, CET data generation, and application log generation. CET tool has two main objects, namely, the thread object and the instruction object.

The thread object stores all information regarding each thread. This includes thread ID, CET code starting address, original code starting address (initial PC value), CET code, CET data, and some statistics, such as, the number of instructions of each type and the sizes of different CET data components.

The instruction object stores all information regarding each instruction. Thus, the CET code is a list of instruction objects. The most important fields of the instruction object are:

1. opcode.
2. Instruction address.
3. Branch results counters to count the taken/not taken in the case of a conditional branch instruction.
4. List of addresses to store the data references of load/store instructions and target addresses in the case of branch instructions.
5. List of counters to store the number of iterations for inner loop instructions.

A.2 HySim Timing Model

HySim timing model has a hierarchal modular design. The top module (MultiCore.bsv) is the module where the CET tiles are instantiated and interconnected. Each tile comprises a core model, L1 D-cache model, L1 I-cache model, L2 cache model, L3 cache model, NoC router model, and cache memories to cache CET code and CET data. Figure 55shows the hierarchal view of the BSV code.

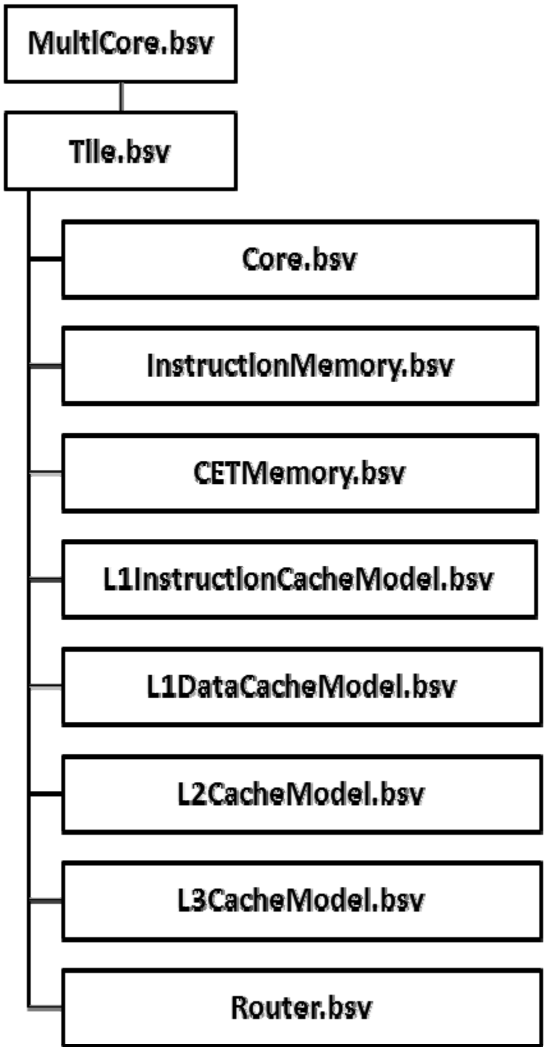


Figure 55: BSV Code Hierarchy

VITAE

Name: Ayman Ali Mohammad Hroub

Nationality: Palestinian

Date of Birth: September 19th, 1984

Email : ahroub@gmail.com

Address: Hebron, Palestine

Academic Background: Ayman received his B.Sc. degree in Computer Systems Engineering from Birzeit University in 2008. Then Ayman worked as a software application developer for more than one year. In September 2009, Ayman joined King Fahd University of Petroleum and Minerals (KFUPM) as a research assistant to pursue his M.Sc. degree in Computer Engineering. In June 2011, Ayman earned his M.Sc. in Computer Engineering from KFUPM. Then Ayman started his career as a lecturer-B at KFUPM to pursue his PhD in Computer Science and Engineering. Ayman Completed his in December 2015.

Ayman research interests include developing novel multicore architectures and efficient models for evaluating the performance of such architectures. Ayman co-authored the following six papers:

1. **Ayman Hroub**, Muhammad E. S. Elrabaa, Muhamed F. Mudawar, Ahmad Khayyat. Efficient Execution Trace Compression Technique for Multi-Core Architectural

Simulation. Submitted to ACM transactions on Modeling and Computer Simulation, 2016.

2. Muhammad E. S. Elrabaa, **Ayman Hroub**, Muhamed F. Mudawar, Ahmad Khayyat. A very fast trace-based simulation platform for chip-multiprocessors architectural explorations. submitted to IEEE Transactions on Parallel and Distributed Systems, 2015
3. M. Alshayeb, M. E. S. Elrabaa, **Ayman Hroub,** A. Al-Aghbari, A. H. El-Maleh, A. Bouhraoua. Towards a Test Definition Language for Integrated Circuits. Journal of Circuits, Systems and Computers (JSCS), 2014.
4. M. Niazi, S.Mahmood, M. Alshayeb**, Ayman Hroub**. An Empirical Investigation of Challenges of the Existing Tools Used in Global Software Development Projects. IET Software, 2014.
5. M. Niazi, S.Mahmood, M. Alshayeb, **Ayman Hroub**. Challenges of the Existing Tools Used in Global Software Development Projects. The Seventh International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services (CENTRIC) Oct. 12 - 16, 2014 - Nice, France.
6. M. Mudawar and **Ayman Hroub**, Clustering Cores for Parallel Thread Execution, 2nd International Conference on Advanced Computing and Communications (ACC-2012), June 27-29, 2012, Los Angeles, California, USA.