# EXPERIMENTAL EVALUATION OF TECHNIQUES FOR CAPTURING AND COMPRESSING HARDWARE TRACES IN MULTICORES

by

AMRISH K. TEWAR

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical & Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2015

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____ _____

(student signature) (date)

THESIS APPROVAL FORM

Submitted by Amrish K. Tewar in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering.

_____ Committee Chair
(Date)

_____

_____

_____

_____ Department Chair

_____ College Dean

_____ Graduate Dean

# ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering College/Dept. Engineering/Electrical & Computer Engineering

Name of Candidate Amrish K. Tewar
Title Experimental Evaluation of Techniques for Capturing and Compressing Hardware Traces in Multicores

Modern embedded systems are indispensable in all aspects of modern life. The increasing complexity of hardware and software stacks and tightening time-to-market deadlines make software development and testing the most critical aspects of system development. To help developers find software bugs faster, modern embedded systems increasingly rely on on-chip resources for debugging and tracing. Unfortunately, capturing and streaming all hardware events of interest for program debugging is cost-prohibitive in multicores where tens of processor cores work concurrently at very high speeds. This thesis focuses on capturing control-flow and data traces in multicores. It introduces two new techniques: *mcfTRaptor* for capturing control-flow traces and *mlvCFiat* for capturing load data value traces. The effectiveness of the commercial state-of-the-art and the proposed techniques are experimentally evaluated by measuring the number of bits needed to be streamed off the chip for both functional and timed traces. The results show that the proposed techniques are very effective, while requiring modest hardware support.

Abstract Approval: Committee Chair _____

Department Chair _____

Graduate Dean _____

This thesis is dedicated to
my mother and wife
without whose love and support
I would not be where I am

# ACKNOWLEDGMENTS

I am grateful to the Almighty, who has always given me good mentors to achieve big steps throughout my life. This Thesis is one of those big steps which has immensely built me up. On completion of this step, I want to express my gratitude to those who have played an important role, either directly or indirectly.

Foremost, I would like to express my gratitude to my advisor, Dr. Aleksandar Milenković. He is an exceptional and patient teacher, and a great motivator. I have always admired his honest comments regarding my work on each and every problem. His work ethic and continuous counselling has always provided me with inspiration and support. For all his efforts on my behalf, I thank him and wished him continued success in his career.

I would like to thank Dr. Rafael Ubal (Multi2Sim) & Albert Myers (mTrace) for developing the tools which were vital to this research.

I would like to express thanks to Professors Rhonda Gaede and Earl Wells for serving in my thesis committee. I am thankful to all the ECE Department professors who taught me various subjects in Computer Engineering. All the staff members of ECE and COE have helped me in many ways during my time as a Master student.

I would also like to acknowledge several people, on a personal front, who have provided me support and the resources to excel. Foremost, I express my immense gratitude to Mr. S. N. Bhatt, and Mrs. M. S. Bhatt for leading me to the path of engineering. I would like to thank Dr. K. Bhatt, Mrs. J. Rosano, Mr. P. B. Pandya, Mrs. R. H. Bhatt, Mr. H. I. Bhatt, and Mr. S. J. Patel for their support. I would also like to thank Mr. A. M. Tewar, Mrs. V. R. Desai, Mr. M. G. Pandya, Dr. B. B. Patel, Mr. M. P. Gandhi, Mrs. H. C. Desai, Mr. M. Shah, and Mr. K. Thakkar for their guidance and help.

Finally, but most importantly, I would express my gratitude to my family. I want to thank my parents, Harshaben and KiritKumar Tewar, for their love and support. I thank my better half, Ruchi Bhatt, for giving me inspiration for furthering my studies, and for her continuous support.

# TABLE OF CONTENTS

# LIST OF FIGURES

Figure Page

# LIST OF TABLES

Table Page

# CHAPTER 1

# INTRODUCTION

Without efforts you cannot achieve – Do not desire anything free
--Rev. Pandurang Shastri Athavale

## 1.1 Background and Motivation

Embedded computer systems are indispensable in modern communications, transportation, manufacturing, medicine, entertainment, and national security. Embedded computer systems are often used as a part of larger physical systems they control or serve by providing computational services. Such systems are often referred to as cyber-physical systems. Faster, cheaper, smaller, more sophisticated, and more power-efficient embedded computer systems spur new applications that require very complex software stacks. The growing software and hardware complexity and tightening time-to-market deadlines make software development and debugging the most critical aspects of embedded system development.

A study by the National Institute of Standard and Technology (NIST, RTI, 2002) [1] found that software developers spend 50 to 75% of their development time debugging programs. Thus, with 800,000 software developers in the U.S. with annual gross salaries of $120,000, the annual cost of software debugging is $48 billion. In spite of these efforts, the U.S. still loses approximately $20-$60 billion a year due to

software bugs and glitches. The recent shift toward multicore architectures makes software development and debugging even more challenging.

Ideally, software developers would like to have perfect visibility into the system state during program execution. However, achieving complete visibility of all internal signals in real time is not feasible due to limited I/O bandwidth, high internal complexity, and high operating frequencies. To address these challenges, modern embedded processors increasingly include on-chip hardware modules solely devoted to debugging and tracing. These modules encompass logic for stop-control debugging and resources to capture, filter, buffer, and output control-flow and data traces. These traces, coupled with powerful software debuggers, enable a faithful program replay that allows developers to locate and correct software bugs faster.

Figure 1.1 shows a typical embedded system-on-a-chip (SoC) with 4 processor cores and its on-chip debugging resources that include run-control logic, logic for capturing program traces, and buffers that serve to temporarily store captured traces before they are streamed out through a trace port to an external trace probe. The external trace probe typically includes large trace buffers on the order of gigabytes and interfaces to the target platform's trace port and to the host workstation. The host workstation runs a software debugger that replays the program execution off-line by reading and processing the traces from the external probe and executing the program binary. This way, software developers can faithfully replay the program execution on the target platform and gain insights into behavior of the target system while it is running at full speed.

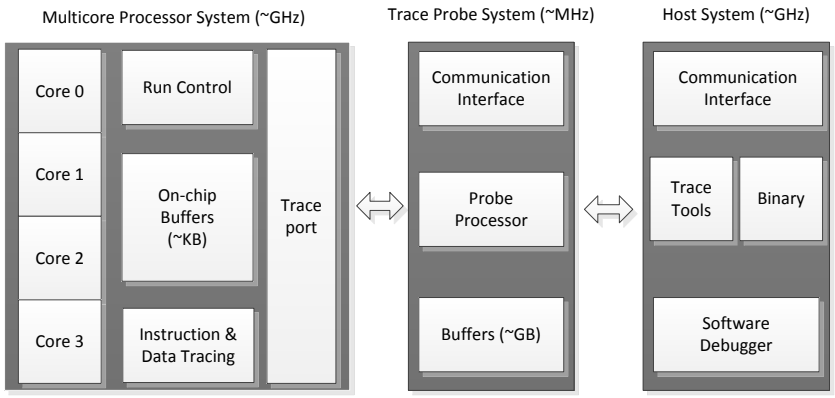


Figure 1.1 Debugging and tracing in embedded multicores: a system view

The IEEE's Nexus 5001 standard [2] defines functions and interfaces for debugging and tracing in embedded processors for four classes of debugging and tracing operations (Class 1 – Class 4). State-of-the-art trace modules employ filtering and encoding to reduce the number of bits necessary to recreate program execution. Yet, trace port bandwidths are still in the range of 1 to 4 bits per instruction executed per core for control-flow traces [3] and 16 bits per instruction executed per core for data-flow traces [3]. With these trace port bandwidth requirements, a 1 KB on-chip trace buffer per processor core may capture control-flow of program segments on the order of 2,000-8,000 instructions or data-flow of program segments of merely 400-800 instructions. Such short program segments are often insufficient for locating software errors in modern systems with more sophisticated software stacks where the distance between a bug's origin and its manifestation may span billions of executed instructions. Increasing the size of the buffers and the number of pins for trace ports is not an attractive alternative to chip manufacturers as it significantly increases the system complexity and cost. This problem is exacerbated in multicore processors where the number of I/O pins dedicated to trace ports cannot keep pace

with the exponential growth of the number of processor cores on a single chip. Yet, debugging and tracing support in multicores is critical because of their increased proliferation in embedded systems and their increased sophistication and complexity.

## 1.2 What is this thesis about?

Developing cost-effective hardware support for debugging and tracing in multicores is of great importance for future embedded systems. On-chip debug and trace infrastructure should be able to unobtrusively capture control-flow and data-flow traces from multiple processor cores at minimal cost (which translates into minimal on-chip trace buffers) and stream them out in real-time through narrow trace ports.

This thesis focuses on capturing and compressing control-flow and load data value hardware traces in multicores. These traces are sufficient to replay programs offline in the software debugger under certain conditions. We first analyze requirements for real-time tracing in multicores as a function of the number of cores by running a set of parallel benchmark programs. We analyze trace port bandwidth requirements for control-flow Nexus-like trace (*mcfNX_b*) and load data value traces (*mlvNX_b*). We introduce two new techniques for capturing and compressing hardware traces, namely *mcfTRaptor* for control-flow traces and *mlvCFiat* for load data value traces. *mcfTRaptor* is a multicore implementation of the previously proposed single-core technique called *TRaptor* [4]. *mlvCFiat* is a multicore implementation of the previously proposed *CFiat* [5]. We explore effectiveness of the proposed techniques as a function of the complexity of the proposed hardware predictors and encoding mechanism. Experimental evaluation involves both functional traces collect-

ed using *mTrace* tools [6] and timed traces collected using a cycle-accurate architectural simulator [7].

## 1.3 Main Results

The main results of our experimental evaluation are as follows. The total trace port bandwidth for Nexus-like control-flow hardware traces (*mcfNX_b*) for the Splash2x benchmark suite ranges from 0.93 bits per instruction executed when the number of cores N = 1 to 1.15 bpi when N = 8. The trace port bandwidth in bits per clock cycle is 1.21 when N = 1 and 4.68 when N = 8. The total trace port bandwidth for Nexus-like load data value traces (*mlvNX_b*) ranges from 18.25 bits per instruction when N = 1 to 19.08 when N = 8. The trace port bandwidth in bits per clock cycle is 23.7 when N = 1 and 78.56 when N = 8. These results indicate that capturing control-flow and especially load data value traces on the fly in multicores requires both large trace buffers and wide trace ports.

The proposed *mcfTRaptor* method dramatically reduces the trace port bandwidth. With *mcfTRaptor,* the trace port bandwidth ranges from 0.07 when N = 1 to 0.09 bpi when N = 8, a 12-fold improvement relative to the Nexus-like control-flow trace. With a large [Section 4.3.3] configuration the improvement is between 36.5 times when N = 1 and 30.3 times when N = 8. *tmcfTRaptor* involves streaming out timestamps with each trace message and is also very effective, providing improvements of 12 ~ 13 times for a small [Section 4.3.3] configurations and 18 ~ 20 times for a large configurations relative to the Nexus-like timed control-flow trace.

The proposed *mlvCFiat* technique offers significant improvements relative to the Nexus-like load data value traces. The trace port bandwidth is reduced 3.9

times when N = 1 and 4.6 when N = 8 for relatively small [Section 4.3.3] data caches. The trace port bandwidth is reduced 6.7 times when N = 1 and for 7.4 times when N = 8 for relatively large [Section 4.3.3] data cache sizes. *tmlvCFiat* achieves even better results mainly due to relatively smaller data sets used in the Splash benchmarks that serve as the workload for timed traces.

## 1.4 Contributions

This thesis makes the following contributions to the field of hardware support of on-chip tracing and debugging in multicore processors:

- Introduces *mcfTRaptor* and *mlvCFiat*, hardware/software techniques for capturing and compressing hardware control-flow traces (*mcfTRaptor*) and load data value traces (*mlvCFiat*) in multicores;
- Develops framework for experimental evaluation of tracing techniques for functional and timed control-flow and load data value traces;
- Evaluates effectiveness of techniques for functional (*mcfNX* and *mcfTRaptor*) and timed control-flow tracing (*tmcfNX* and *tmcfTRaptor*);
- Evaluates effectiveness of techniques for functional (*mlvNX* and *mlvCFiat*) and timed load data value tracing (*tmlvNX* and *tmlvCFiat*).

## 1.5 Outline

The outline of this thesis is as follows. Chapter 2 gives background, focusing on control-flow trace and memory data traces, tracing and debugging in embedded systems, and commercial and academic state-of-the-art. Chapter 3 introduces the *mcfTRaptor* and *mlvCFiat* techniques for filtering and compressing control-flow and

load data value traces in multicores. Chapter 4 describes our experimental evaluation for functional traces. Chapter 5 describes the results of the experimental evaluation for functional traces. Chapter 6 describes our experimental evaluation of timed traces and Chapter 7 describes the results of the evaluation. Finally, Chapter 8 gives concluding remarks.

# CHAPTER 2

# BACKGROUND AND MOTIVATION

Efforts are never in vain - Do not despair

--Rev. Pandurang Shastri Athavale

This Chapter focuses on types of program execution traces, namely control-flow (Section 2.1) and memory data read and write traces (Section 2.2), that are commonly used in program debugging. Section 2.3 gives a more detailed system view of trace-based debugging in embedded systems and surveys the commercial state-of-the-art. Section 2.4 gives a brief survey of the academic state-of-the-art in the field of capturing and compressing program execution traces.

## 2.1 Control Flow Traces

Control-flow traces are created by the recording memory addresses of all committed instructions in a program. However, such traces include a lot of redundant information that can be inferred by the software debugger with access to the program binary. To recreate the program's control-flow off-line, the debugger needs only information about changes in the program flow caused by control-flow instructions or exceptions. When a change in control-flow occurs, we could record the program counter (PC) and the branch target address (BTA) in case of a control-flow instruction or the exception target address (ETA) in case of an exception. However, such a sequence of (PC, BTA/ETA) pairs still contains redundant information. To

reduce the number of bits to encode lengthy (PC, BTA/ETA) pairs, we can replace PC with the number of sequentially executed instructions in an *instruction stream*, also known as stream length (SL). An instruction stream or dynamic basic block is a sequence of sequentially executed instructions starting at the target of a taken branch and ending with the first taken branch in the sequence [8],[9]. In addition, the target addresses of direct taken branches (BTA) do not need to be recorded as they can be inferred by the software debugger. Therefore, to recreate the program's control-flow in the software debugger, only the following changes in the control-flow need to be reported from the target platform.

- A *taken conditional direct branch* generates a trace message that contains only the number of sequentially executed instructions in the instruction stream, (*SL*, -); the target address can be inferred from the program binary.
- An *indirect unconditional branch* generates a trace message that includes the stream length and the address of the indirect branch, (*SL*, *BTA*); and
- An *exception event* generates a trace message that includes the message type (*eType*), the number of instructions executed since the last reported event *(iCnt*), and the exception target address (*ETA*), (*eType*, *iCnt*, *ETA*).

For multicores executing multithreaded programs, control-flow trace messages need to include information about the core on which a particular code segment has been executed. Note: Without loss of generality, we assume that each thread executes on a single core (*Ti* = *Ci*). Though threads can migrate between the cores, these migrations can be captured by system software rather than through hardware methods and can be merged with the hardware trace in the software debugger.

To illustrate capturing control-flow traces of a multithreaded program, consider the OpenMP C program shown in Figure 2.1 that sums up elements of an integer array. An assembly code snippet in the middle shows the instructions executed. We can identify the following instruction streams: the stream A with 15 instructions starting at the address 0x80488b3, the stream B with 14 instructions starting at the address 0x80488b6, the stream C with 15 instruction starting at the address 0x80488b6, and the stream D with 5 instructions starting at the address 0x80488e9. The same code snippet is executed in two threads ($Ti = 0$ and $Ti = 1$). Figure 2.1 shows a functional control-flow trace for both threads. Each thread sums up 4 elements of the original array and the sequence of reported instruction streams is as follows: A, B, B, C, D. The stream D ends with an indirect branch (*retq* instruction), so the last trace message will also include the target address (not known in compile time). The streams A, B, and C end with direct branches with inferable targets, and thus their target addresses are not included (traced out). On the bottom, timed trace messages are shown that include time stamps recording the clock cycle when a particular trace message is captured.

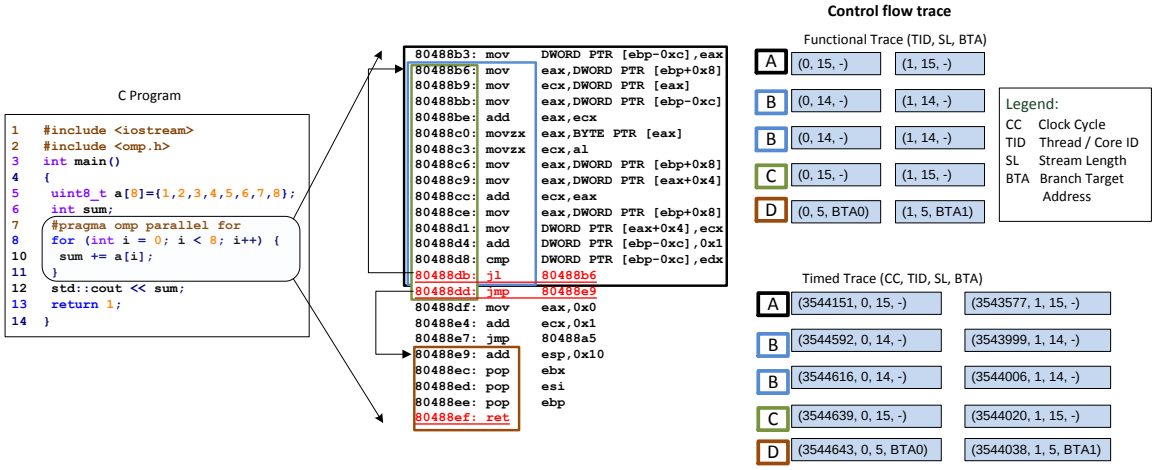


Figure 2.1. Control-flow trace: an example

## 2.2 Memory Data Traces

Memory data traces are created by recording relevant information for each memory read or write operation in a program execution. This information typically includes the following: the instruction address (PC), type of memory operation -- read or write (R/W), the operand address (OA), the operand size (OS), and the operand value (OV). In multicores, each trace record should include the thread or core index (Ti). Finally, in the case of timed traces, each record includes a time stamp indicating the clock cycle in which the event has occurred.

Figure 2.2 shows a memory read trace excerpt for the OpenMP C program shown in Figure 2.1. On the right hand side the flow trace records caused by the move instruction at address 0x80488c0 that reads a byte from the input byte array. Each trace record includes the thread index, the operand address, the operand size, and the operand value. Below are the trace records that in addition to the fields above include time stamps when particular memory reads have completed.

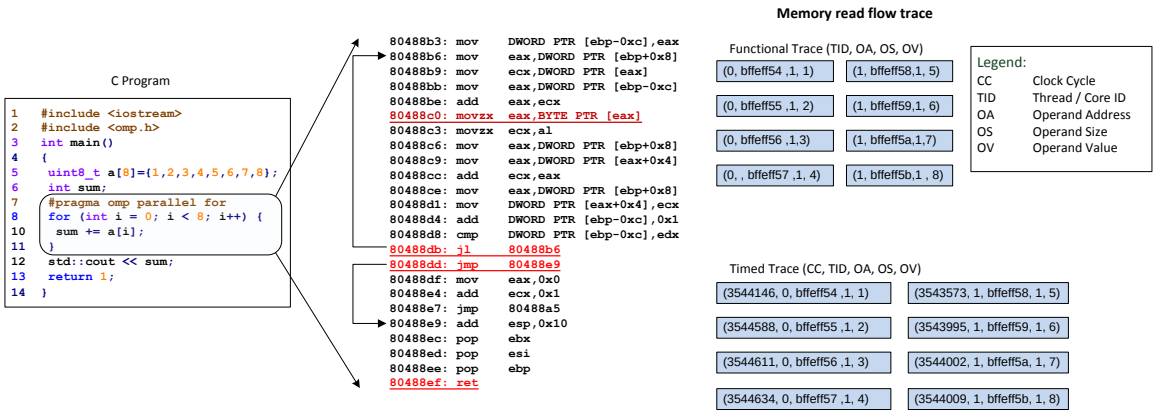


Figure 2.2. Memory read trace: an example

Control-flow traces alone are sufficient to reconstruct the program-flow. However, for certain classes of software bugs (e.g., data races), control-flow traces alone are insufficient and data traces are also required. Data traces are critical in multicore systems, as they offer valuable information about shared memory access patterns and possible data race conditions. Unfortunately, data traces tend to be very large: a 64-bit machine reading an 8-byte operand from memory generates a trace record with more than 24 bytes (8 byte instruction address, 8-byte operand address, 8-byte operand value). Streaming an entire memory data trace through a trace port is thus cost prohibitive. Fortunately, replaying the program offline requires only memory reads and not writes, as that information is infer from software debugger.

The software debugger needs only a portion of the data trace to replay the program. Exception traces and load (memory read) data value traces captured on the target platform and streamed out to a software debugger are necessary to deterministically replay programs offline. Exception traces are created by recording

exceptions that occur in program execution, and load data value traces record only values read from memory and I/O devices. In addition to the traces, the software debugger needs the following to faithfully replay the program offline: (i) an instruction set simulator (ISS) of the target platform, (ii) access to the program's binary, and (iii) the initial state of the general-purpose and special-purpose registers of individual cores. ISS is a simulation model tool to mimic the behavior of processors. In multicores, the exception and data traces need to be either streamed in the order of occurrence or with global timestamps.

## 2.3 Tracing in Embedded Multicores

Trace and debug modules encompass hardware that can support different classes of debugging and tracing operations. The IEEE Nexus 5001 standard [2] defines functions and interfaces for debugging and tracing in embedded processors for four classes of debugging and tracing operations (Class 1 – Class 4). Class 1 supports basic debug operations for run-control debugging such as single-stepping, setting breakpoints, and examining and modifying processor registers and memory locations when the processor is halted. It is traditionally supported through a JTAG interface [10]. The higher classes progressively add more sophisticated operations at the cost of additional on-chip resources (logic, buffers, and interconnects) solely devoted to tracing and debugging. Thus, Class 2 adds support for nearly unobtrusive capturing and streaming out control-flow traces in real-time. Class 3 adds support for capturing and streaming out data-flow trace (memory and I/O read and write data values and addresses). Finally, Class 4 adds resources to support emulated memory and I/O accesses through the trace port.

Class 1 operations are routinely deployed in modern platforms. However, Class 1 operations are lacking in several important aspects. First, they place a burden on software developers to perform time-consuming and demanding steps such as setting breakpoints, single-stepping through programs and examining visually the content of registers and memory locations. Moreover, setting breakpoints is not practical or feasible in cyber-physical and real-time systems. Finally, since the processor needs to be halted, the debugging operations are obtrusive and may perturb sequences of events on the target platform and thus cause original bugs to disappear during debug runs.

To address these challenges, many chip vendors have recently introduced trace modules with support for Class 2 and less frequently for Class 3 debug and trace operations. Figure 2.3 shows a typical embedded system-on-a-chip (SoC) with multiple processor cores and its debugging and tracing resources. The SoC is connected to a trace probe system through a trace port. The multicore SoC includes various components, such as multiple processor cores (Core0 – Core3), a DSP core, and a DMA core, all connected through a system interconnect. Each component includes its own trace and debug logic (trace modules) that captures program execution traces of interest. Individual trace modules are connected through a trace and debug interconnect to on-chip trace buffers. On-chip buffers store program execution traces temporarily before they are read out through a trace port to an external trace probe. The external trace probe typically includes large trace buffers on the order of gigabytes and interfaces to the target platform's trace port and to the host workstation. The host workstation runs a software debugger that replays the program execution offline by reading and processing the traces from the external probe and

executing the program binary. This way, software developers can faithfully replay the program execution on the target platform and gain insights into the behavior of the target system while it is running at full speed.

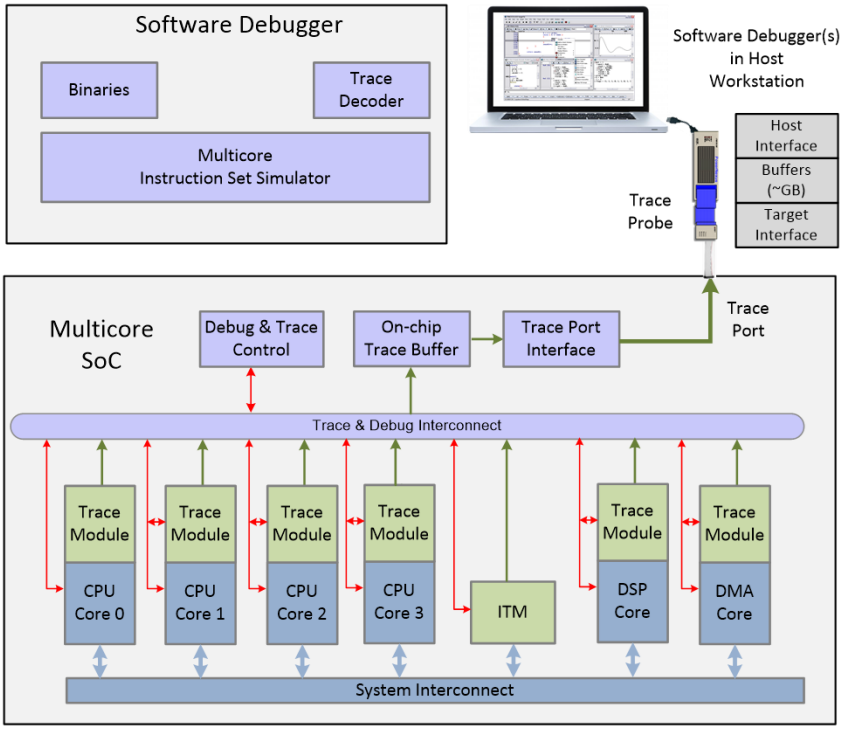


Figure 2.3 Debugging and tracing in multicores: a detailed view

Some examples of commercial trace modules include ARM's CoreSight [3], MIPS's PDTrace [11], Infineon's MCDS [12], and Freescale's MPC5500 [13]. State-of-the-art trace modules employ filtering and encoding to reduce the number of bits necessary to recreate program execution. Yet, trace port bandwidths are still in the range of 1 to 4 bits per instruction executed per core for control-flow traces [3] and 8 to 16 bits per instruction executed per core for data-flow traces [3]. With these trace port bandwidth requirements, a 1 KB on-chip buffer per processor core may capture

control-flow of program segments on the order of 2,000-8,000 instructions or data-flow of program segments of merely 400-800 instructions. Such short program segments are often insufficient for locating software errors in modern systems with more sophisticated software stacks where a distance between a bug's origin and its manifestation may span billions of executed instructions. Increasing the size of the buffers and the number of pins for trace ports is not an attractive alternative to chip manufacturers as it significantly increases the system complexity and cost. This problem is exacerbated in multicore processors where the number of I/O pins dedicated to trace ports cannot keep pace with the exponential growth of the number of processor cores on a single chip. Yet, debugging and tracing support in multicores is critical because of their increased proliferation in embedded systems and their increased sophistication and complexity.

## 2.4 Related Work

Commercially available trace modules typically implement only rudimentary forms of hardware filtering with a relatively small compression ratio. Irrgang and Spallek analyzed the Nexus and trace port configurations and their impact on achievable compression for instruction traces and found port width of 8bits with history messaging is effective [14]. Several recent research efforts in academia propose trace-specific compression techniques that achieve higher compression ratios. These techniques rely on hardware implementations of general-purpose compressors [15] [16]. For example, Kao et al. [17] introduce an LZ-based compressor specifically tailored to control-flow traces. The compressor encompasses three stages: filtering of branch and target addresses, difference-based encoding, and hardware-based LZ

compression. Novel approach, stream based compression algorithm[18] exploits inherent characteristics program execution traces for compression. A double-move-to-front compressor introduced by Uzelac and Milenkovic [15] encompasses two stages, each featuring a history table performing the move-to-front transformation. Although these techniques significantly reduce the size of the control-flow trace that needs to be streamed out, they have a relatively high complexity (50,000 gates and 24,600 gates, respectively).

A set of recently developed techniques relies on architectural on-chip structures such as stream caches[19], [20], [21] and branch predictors [4] [22] [23] with their software counterparts in software debuggers, as well as effective trace encoding to significantly reduce the size of traces that needs to be streamed out [4], [5],[19],[20]. Uzelac et al. [4] introduced *TRaptor* for control-flow traces that achieves 0.029 bits per instruction on the trace port (~34-fold improvement over the commercial state-of-the-art) at hardware cost of approximately 5,000 gates. For load value traces, Uzelac and Milenkovic [24] [5] introduced cache first-access tracking mechanism (c-fiat) that reduces the trace size between 5.8 to 56 times, depending on the cache size.

However, these techniques have been demonstrated on uniprocessors only. The problem of tracing requirements in multicores running parallel programs is not fully understood. What is the required trace port bandwidth? How does trace port bandwidth scale up with a multiple processor cores? How the existing techniques may be applied to multicores? These are some of the questions that needs to be fully addressed [25]. In this thesis, we want to explore requirements for real-time tracing

in multicores and introduce cost-effective solutions that scale well with a multiple processor cores.

# CHAPTER 3

# NEW TECHNIQUES FOR TRACING IN MULTICORES

> Continue to make efforts, pray for help, and help is assured – Do not lose faith
>
> --Rev. Pandurang Shastri Athavale

This chapter describes *mcfTRaptor* and *mlvCFiat* techniques for capturing and compressing hardware control-flow and load-value data traces in multicore embedded processors.

## 3.1 *mcfTRaptor*

In this section, we introduce a technique called multicore control flow tracing branch predictor alias *mcfTRaptor*. *mcfTRaptor* is an extension of the existing *TRaptor* technique for capturing and filtering control-flow traces in single-core processors [4], [24]. Figure 3.1 illustrates a multicore system-on-a-chip with tracing and debugging resources. The multicore connects to a software debugger running on a development workstation via a debug & trace interface. Each core has its own trace module that captures information about committed control-flow instructions. The trace module includes predictor structures in hardware, solely dedicated to capturing and filtering control flow traces. These structures are looked up and updated every time a non-inferable control-flow instruction (a conditional branch or an indirect branch) commits in the corresponding processor core. For a given control-flow instruction, the predictor structures either (a) correctly predict the outcome or target address; (b) incorrectly predict the outcome or target address, or (c) cannot make a

prediction (e.g., due to a predictor miss). In all cases, the predictor structures are updated based on their update policies, similarly to branch predictors in processor pipelines. The key insight that leads to a significant reduction in the number and size of trace messages is that trace messages need be generated only when rare mispredictions occur in the *mcfTRaptor* structures on the target platform. The messages are stored in a trace buffer, streamed out of the platform, and read by the software debugger. The software debugger has access to the program binary, instruction set simulator, and the trace messages captured on the target platform. The debugger maintains software copies of all *mcfTRaptor* structures. These structures are looked up and updated during program replay in the same way their hardware counterparts are looked up and updated on the target platform.

Figure 3.2 shows a block diagram of typical *mcfTRaptor* structures for the core with index *i*. The processor's trace module receives information about committed control flow instructions including, time, program counter (PC), direct/indirect branch type, outcome (taken/not taken), and branch target address (BTA) or exception target address (ETA). *mcfTRaptor* includes structures for predicting (a) target addresses of indirect branches, e.g., an indirect Branch Target Buffer (iBTB) and a Return Address Stack (RAS) [26]; and (b) outcomes of conditional branches, such as an outcome gshare predictor [7]. In addition, *mcfTRaptor* includes two counters: an instruction counter *Ti.iCnt* and a branch counter *Ti.bCnt*. *Ti.iCnt* is incremented upon retirement of each executed instruction, and *Ti.bCnt* is incremented only upon retirement of control-flow instruction that could generate trace messages (e.g., conditional direct and unconditional indirect branches).

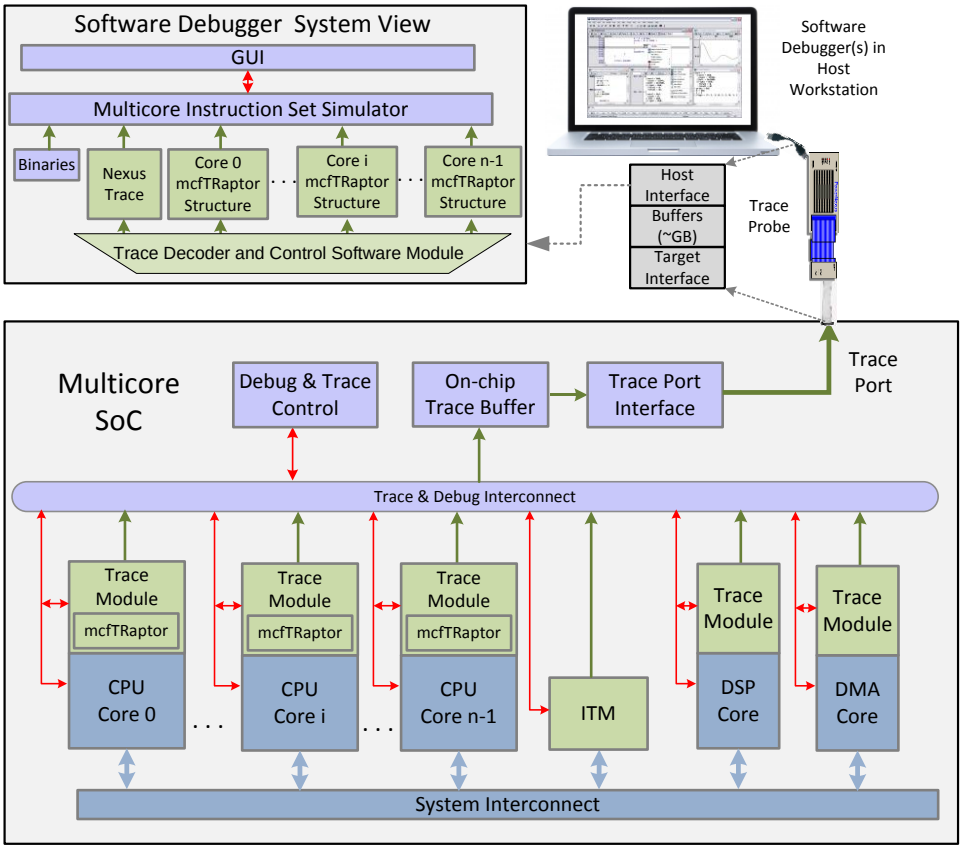


Figure 3.1 A system view of *mcfTRaptor*

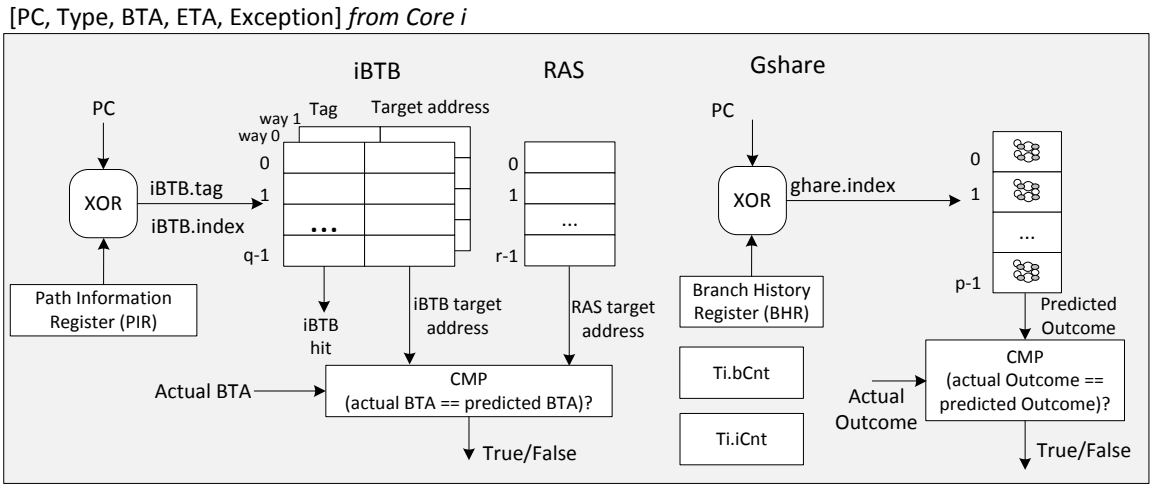


Figure 3.2. *mcfTRaptor* structures for core *i*

Figure 3.3 describes the operation of a trace module attached to core *i* when capturing control-flow tracing using *mcfTRaptor*. The instruction counter is incremented for each committed instruction. The branch counter is incremented for each control-flow instruction capable of generating a trace message. For indirect unconditional branches, the trace module generates a trace message only if the predicted target address does not match the actual target address. For direct conditional branches, the trace module generates a trace message only if the predicted outcome does not match the actual outcome. When a trace message is generated and placed in a trace buffer for streaming out, the counters *Ti.iCnt* and *Ti.bCnt* are cleared. The predictor structures are updated according to an update respective structures. In case of exceptions, a trace message is generated with *Ti.bCnt* = 0 to indicate a special case, followed by the instruction count (*Ti.iCnt*) and the exception address (*Ti.ETA*).

```
1. // For each committed instruction in Thread with index i on core i
2. Ti.iCnt++; // increment iCnt
3. if ((Ti.iType == IndBr) || (Ti.iType == DirCB)) {
4.     Ti.bCnt++; // increment bCnt
5.     // target address misprediction
6.     if ((Ti.iType == IndBr) && (Ti.BTA != p.BTA)) {
7.         Encode&Emit trace message <Ti, Ti.bCnt, Ti.BTA>;
8.         Place trace message into the Trace Buffer;
9.         Ti.iCnt = 0;
10.        Ti.bCnt = 0;
11.     }
12.     // outcome misprediction
13.    else if ((Ti.iType==DirCB) && (Ti.Outcome != p.Outcome)) {
14.        Encode&Emit trace message <Ti, Ti.bCnt>;
15.        Place trace message into the Trace Buffer;
16.        Ti.iCnt = 0;
17.        Ti.bCnt = 0;
18.    }
19.    Update predictor structures;
20. }
21. if (Exception event) {
22.    Encode&Emit trace message <Ti, 0, iCnt, ETA>;
23.    Place record into the Trace Buffer;
24.    Ti.iCnt = 0;
25.    Ti.bCnt = 0;
26. }
```

Figure 3.3 *mcfTRaptor* operation on core $i$

The software debugger replays the instructions as shown in Figure 3.4. The replay starts by reading trace messages for each thread and initializing the counters. If a non-exception trace message is processed, the software copy of $Ti.bCnt$ is decremented every time a control-flow instruction is executed. For indirect unconditional branches, if the counter reaches zero, the actual target address is retrieved from the current trace message; otherwise if ($Ti.bCnt > 0$), the target address is retrieved from the *mcfTRaptor* structures maintained by the software debugger. For direct conditional branches, if the counter reaches zero, the actual outcome is opposite to the one provided by the *mcfTRaptor* structures maintained by the software debugger; otherwise if ($Ti.bCnt > 0$), the actual outcome matches the predicted one. When $Ti.bCnt$ reaches zero, the next trace message for that thread is fetched. Handling of exceptions events is described in lines 3-8 in Figure 3.4.

```
1.  // For each instruction on core i
2.  Replay the current instruction (if not trace event generating);
3.  if (Exception message is processed) {
4.     Ti.iCnt--;
5.     if (Ti.iCnt == 0) {
6.         Go to exception handler at Ti.ETA;
7.         Get the next trace message;
8.     }
9.  }
10. if ((Ti.iType == IndBr) || (Ti.iType == DirCB)) {
11.    Ti.bCnt--; // decrement Ti.bCnt
12.    if ((Ti.iType == IndBr) && (Ti.bCnt > 0))
13.       Actual BTA = predicted BTA in software;
14.    else if (Ti.iType == DirCB) && (Ti.bCnt > 0))
15.       Actual outcome = predicted outcome in software;
16.    else if ((Ti.iType == IndBr) && (Ti.bCnt == 0))
17.       Actual BTA = BTA read from the trace message;
18.    else if (Ti.iType == DirCB) && (Ti.bCnt == 0))
19.       Outcome is opposite to predicted outcome;
20.    Update software predictor structures;
21.    if (Ti.bCnt == 0) Get the next trace message;
22. }
```

Figure 3.4 Program replay in software debugger for *mcfTRaptor*

## *3.2 mlvCFiat*

*mlvCFiat* or *multicore load value cache first access tacking* is a hardware-based mechanism that reduces load data value traces by collecting a minimal set of trace messages through the use of a cache first access mechanism. *mlvCFiat* is an extension of the existing *CFiat* mechanism for capturing and filtering load data value traces in single-core processors [5], [24].

Figure 3.5 shows a block diagram of a multicore SoC with infrastructure for debugging and tracing; green boxes represent additional *mlvCFiat* modules. Each processor core is coupled to its trace module through an interface that carries information about committed memory read and memory write instructions. The trace module includes structures in hardware solely dedicated to capturing and filtering memory read traces. The *mlvCFiat* structure is looked up when instructions that

read from or write to memory commit in the corresponding processor core. The key insight that leads to a significant reduction in the number and size of trace message is that trace messages need to be generated only when misprediction occur in the *mlvCFiat* structures on the target platform. The messages are stored in a trace buffer, streamed out of the platform, and read by the software debugger. The software debugger has access to the program binary, instruction set simulator, and the trace messages captured on the target platform. It maintains software copies of all *mlvCFiat* structures. These structures are updated during program replay in the same way their hardware counterparts are updated on the target platform.

Figure 3.6, shows the *mlvCFiat* structures for core *i*. Each data cache block in each processor core on the target platform is augmented with first access tracking flags. The first access tracking flags keep track of sub-blocks that need to be reported to the software debugger. Let us assume a data cache with 64-byte cache blocks. If a first-access tracking flag protects a 4-byte sub-block, each cache block needs to be augmented with a 16-bit first access flag vector. The previously reported sub-blocks do not have to be reported again as they can be inferred by the software debugger. This way we exploit the temporal and spatial locality of data access to significantly reduce the number of trace events that need to be reported. In addition to the first-access tracking bits, each trace module includes a local first-access counter (*Ti.fahCnt*) that counts the number of consecutive first access hits.

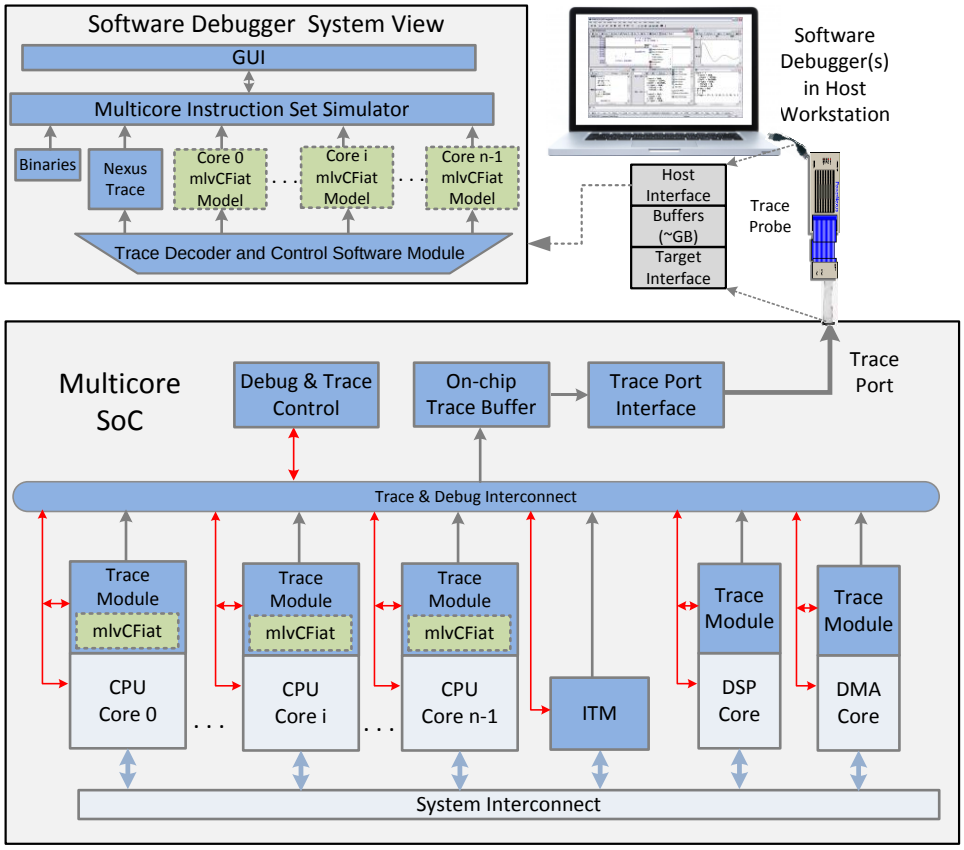


Figure 3.5 A system view of *mlvCFiat*

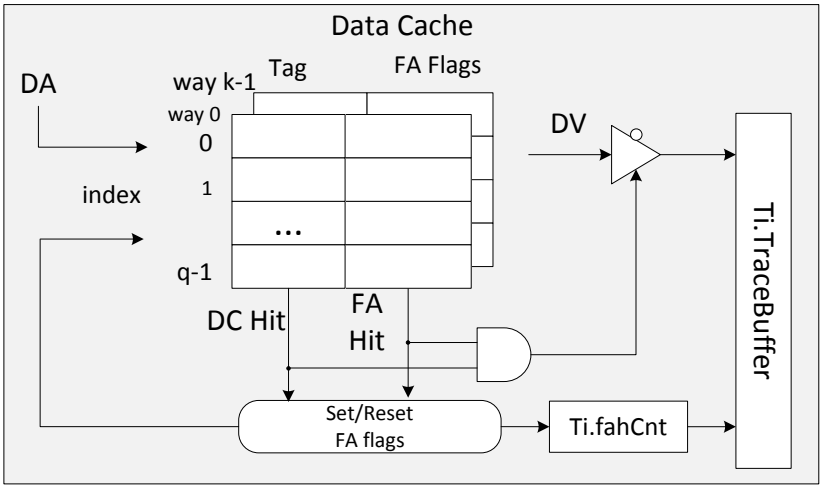


Figure 3.6 *mlvCFiat* structures for core $i$

Figure 3.7 describes the operation of the *mlvCFiat* mechanism on core *i*. Each memory read causes a data cache lookup; if the requested data item is found in the data cache (a cache hit event) and the corresponding first-access flags are set (an FA hit event), the data value does not need to be reported to the software debugger and *Ti.fahCnt* is incremented (line 3). In case of an FA miss event, a trace message is streamed out of the chip; it includes the core id (*Ti*), the current value of the *Ti.fahCnt*, and the load data value that is being reported for the first time (line 5). In addition, the corresponding FA flags are set and the counter *Ti.fahCnt* is cleared (line 7 and 8). In case of a data cache miss event, the newly fetched block's FA tracking flags are cleared and then steps 5-8 are carried out. Similarly, external cache block invalidation or update requests clear the corresponding FA flags (line 17). Finally, memory write operations set the corresponding FA tracking flag(s) (line 15).

```
1. // For each instruction that reads n bytes on core i
2. if (CacheHit) {
3.     if (corresponding FA flags are set) Ti.fahCnt++;
4.     else {
5.         Encode&Emit trace message (Ti, Ti.fahCnt, loadValue);
6          Place trace message into the Trace buffer
7.         Set corresponding FA flags;
8.         Ti.fahCnt = 0;
9.     }
10.  } else { // cache miss event
11.   Clear all FA bits for newly fetched cache block;
12.   Perform steps 5-7;
13. }

14. // For each retired store that writes n bytes
15. Set corresponding FA bits;

16. // For external invalidation/update request
17. Clear FA bits for entire cache block
```

Figure 3.7 *mlvCFiat* operation on core *i*

The software debugger carries out steps that mirror actions on the target (Figure 3.8). It maintains software copies of the data caches and the *Ti.fahCnt* counters; these are updated during program replay using the same policies employed on the target platform. The program replay starts by reading the trace messages received from the target for each core separately. The debugger replays the instructions for each core using ISS. For memory read instructions the debugger performs steps described in lines 1-11. The *Ti.fahCnt* counter is decremented; if *Ti.fahCnt>0*, the debugger retrieves the load data value from the software data cache and moves to the next instruction. If *Ti.fahCnt = 0* we have a first read miss event; the load value is retrieved from the trace message, the software data cache is updated, and a new trace message for a given core is read from the target.

```
1. // For each load on Core i that reads n bytes
2. Ti.fahCnt --;
3. if (Ti.fahCnt > 0) {
4.     Perform lookup in the SW data cache;
5.     Retrieve data value from SW cache;
6. }
7. else { // FA miss event
8.     Read n bytes from trace record;
9.     Update SW cache;
10.    Get the next trace message (Ti, Ti.fahCnt, LoadValue);
11. }

12. // For each store that writes N bytes
13. Update SW cache;
14. Set corresponding n SW cache FA bits;
```

Figure 3.8 *mlvCFiat* operation in software debugger for core *i*

# CHAPTER 4

# EXPERIMENTAL EVALUATION OF FUNCTIONAL TRACES

Fearlessness is result of faith in one self and faith in God
--Rev. Pandurang Shastri Athavale

Functional program execution traces capture behavioral aspects of running programs. They capture control and data flow information from multithreaded programs, preserving intra-threaded ordering of events, but may not provide accurate inter-thread ordering. We use functional traces (i) to investigate requirements of hardware tracing in multicore platforms and (ii) to evaluate the effectiveness of our techniques and their sensitivity to system parameters. This chapter focuses on our experimental flow based on functional traces.

As a measure of effectiveness, we use trace port bandwidth expressed in the number of bits streamed out of the chip per instruction executed (bpi). Figure 4.1 shows the experiment flow for determining trace port bandwidth in case of functional (non-timed) traces. The flow encompasses three steps: (i) software trace generation using the *mTrace* tool suite [6], (ii) software to hardware trace translation, and (iii) trace port bandwidth analysis. Section 4.1 describes the trace generation step. *mTrace* is designed to support a range of trace applications, such as Instruction Set Architecture (ISA) profiling, trace-driven simulation, and software trace compression. Thus, *mTrace* does not include support for analyzing hardware tracing and trace descriptor encoding at the trace port level. In addition, software traces gener-

ated by *mTrace* often include information that may be inferred by a software debugger. To support the evaluation of trace port bandwidth in the context of hardware tracing on multicore platforms, we develop custom tools that read software traces and produce hardware traces with no redundant information. Section 4.2 describes software to hardware trace translation. Section 4.3 describes the experimental set-up, benchmarks used, and the experimental methodology.

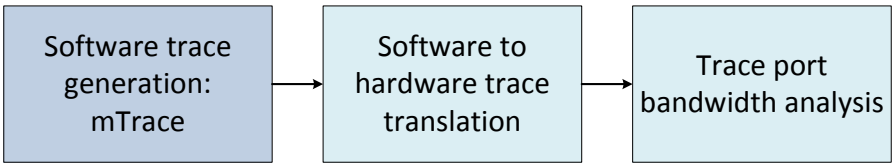


Figure 4.1 Experiment flow for determining trace port bandwidth requirements using functional traces

## 4.1 Software Trace Generation

To generate control-flow and data functional traces for multi-threaded software we use the *mTrace* tool suite [6]. Figure 4.2 illustrates a trace generation flow. *mTrace* relies on the Intel's binary instrumentation framework called Pin [27] that works like a just-in-time-compiler and enables custom binary instrumentation through a well-defined application programming interface. The *mTrace* tools are developed as Pin tools and capture functional traces. They take application input parameters, the number of threads, parameters controlling the tracing process, and

configuration parameters for new compression methods as input parameters, and generate raw trace files, optionally, compressed trace files.

The *mTrace* tool suite consists of four different tools:

- *mcfTrace*: a tool for capturing and compressing control-flow traces;
- *mlsTrace*: a tool for capturing and compressing data traces;
- *mcfTRaptor*: a tool for capturing and compressing control-flow traces using the *TRaptor* mechanism for multi-threaded programs; and
- *mlvCFiat*: a tool for capturing and compressing data traces using the *CFiat* mechanism for multi-threaded programs.

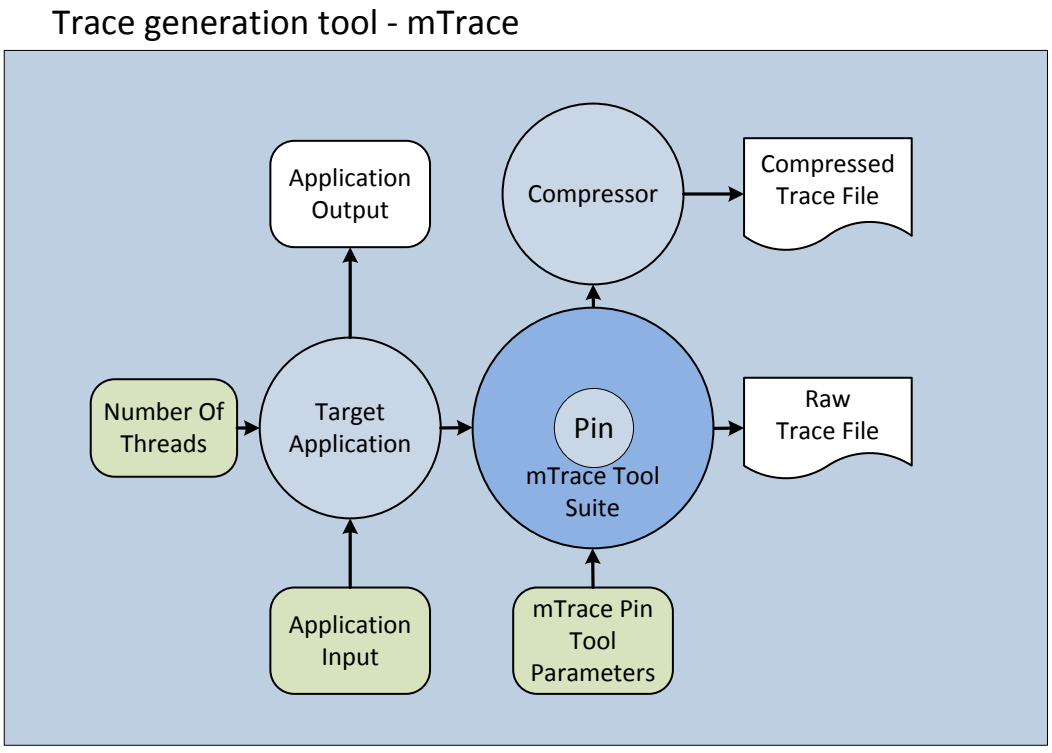


Figure 4.2 Functional trace generation using *mTrace* tool suite

## 4.2 Software to Hardware Trace Translation

To enable evaluation of trace port bandwidth in the context of hardware tracing on multicore platforms, we develop tools that perform software to hardware trace translation. These tools read raw traces generated by the *mTrace* tools, filter out redundant trace descriptors and redundant trace fields that can be inferred by the software debugger, and perform analysis to determine effective encoding of trace descriptors. The output of the trace analyzer tools is the overall trace port bandwidth measured in bits per instruction executed.

Figure 4.3 shows the flow from input software traces to output hardware traces. Raw control-flow traces generated by the *mcfTrace* tool are filtered out and encoded to generate Nexus-like hardware control flow traces called *mcfNX_b*. The mc*TRaptor* control-flow traces generated by the *mcfTRaptor* tool are also filtered out and then encoded using either a fixed encoding mechanism to generate *mcfTR_b* compressed control-flow hardware traces or using a variable encoding mechanism to generate *mcfTR_e* compressed control-flow hardware traces. This way, we can separately evaluate the effectiveness of the *mcfTRaptor* filtering mechanism and the effectiveness of encoding mechanisms. Similarly, a memory load data value trace generated by the *mlsTrace* tool is filtered and encoded to generate a Nexus like hardware load data value trace (*mlvNX_b*). The *mlvCFiat* trace is filtered and encoded using a fixed or a variable encoding mechanism to generate compressed memory load data value traces *mlvCF_b* and *mlvCF_e*, respectively. The following subsections shed more light on each of these trace transformations.

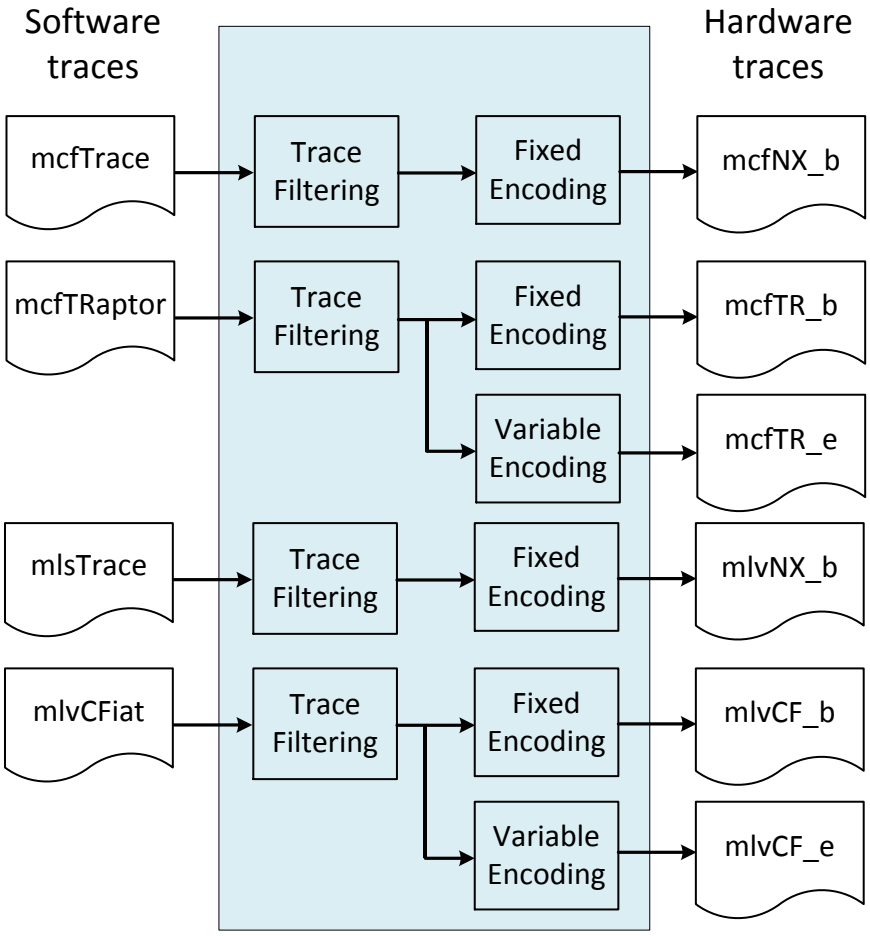


Figure 4.3 Software to hardware trace translation

### *4.2.1 mcfNX_b*

The *mcfTrace* tool captures all control-flow instructions and exceptions. Figure 4.4 (a) shows the format of a trace descriptor generated by *mcfTrace*. A descriptor includes the thread ID, control-flow instruction address (program counter), target address, branch type (direct vs. indirect, conditional vs. unconditional), branch outcome (taken or not taken), and the number of instructions in the basic

block that ends with the control-flow instruction. A descriptor in ASCII format may require up to 58 bytes.

To generate hardware control-flow traces, we filter out information that is not required to replay the program's control-flow offline. To recreate the control-flow, the debugger only needs information about changes in the program flow due to taken conditional branches or exceptions. For each change, we can send the program counter (PC address) and the branch target address (BTA) for branch instructions or exception target address (ETA) for exceptions. Yet, the trace with all (PC, BTA/ETA) pairs still contains redundant information that can be inferred by the debugger providing it has access to the program binary. Instead of sending the program counter we can send the number of instructions executed sequentially from the program starting address or from the target of the last taken branch. The target addresses of direct branches (BTA) can be inferred from the program binary and thus do not need to be streamed out. The target addresses of indirect branches do need to be reported though. However, instead of sending the entire target address, we can send only the absolute difference (|DiffTA|) between the previous indirect branch target address and the current branch target address with sign bit.

Thus, hardware control-flow trace requires tracing descriptors to be emitted as follows:

- for *taken direct conditional branches,* the trace descriptor should include (Ti, Ti.SL, -), where Ti is the thread ID, Ti.SL is the number of instructions executed in a given thread since the last reported trace descriptor;
- for *indirect unconditional branches,* the trace descriptor should include (Ti, Ti.SL, Ti.DiffTA), where Ti.DiffTA is the difference target address; and

- for *exceptions* the trace descriptor should include (Ti, Ti.SL, Ti.ETA).

The resulting format of a trace descriptor for Nexus-like control flow trace is shown in Figure 4.4 (b). To encode the thread index we use 0 bits when N = 1, 1 bit for N = 2, 2 bits for N = 4, and 3 bits for N = 8. The number of bits needed to encode the field Ti.SL varies with benchmarks and phases within a benchmark. Instead of using a long field that would encode any possible value of SL, we use Nexus-like encoding with each field divided into multiple chunks. Each chunk is followed by a connect bit that indicates whether it is the terminating chunk (C = 0) for the given field or more chunks follow (C = 1). For the Nexus-like control-flow trace, we adopt chunk size of 8 bits for the Ti.SL field. Similarly, we encode the absolute value of |diffTA| into two 32-bit chunks.

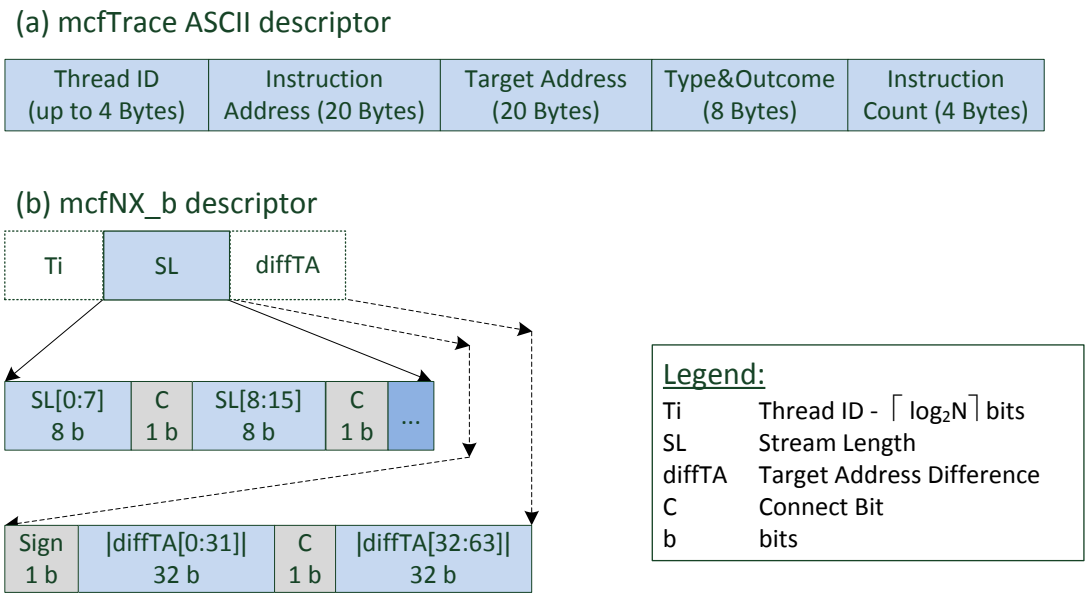


Figure 4.4 *mcfTrace* and *mcfNX_b* trace descriptors

### 4.2.2 *mcfTR_b* and *mcfTR_e*

The *mcfTRaptor* tool implementing the *mcfTRapor* trace compression generates *mcfTRaptor* trace descriptors as described in Table 4.1. It emits trace descriptors upon (i) outcome misprediction of conditional branches, (ii) target address misprediction of indirect unconditional branches, and (iii) exceptions. In case of outcome misprediction, a descriptor is emitted that includes information about the thread index, Ti, and the number of branch instructions encountered in a given thread since the last reported event, Ti.bCnt. For indirect unconditional branches, in addition to Ti and Ti.bCnt, the indirect target address, Ti.BTA, is included as well. Finally, in the case of exceptions, the Ti.bCnt is set to a zero, followed by a field with the number of instructions executed since the last reported event (Ti.iCnt), and the exception target address (Ti.ETA). Figure 4.5(a) illustrates the *mcfTRaptor* descriptors in ASCII format that can require from 18 to 46 bytes.

Table 4.1 *mcfTRaptor* events and trace descriptor fields

| ***mcfTRaptor* Events** | **Trace Descriptors** |
|---|---|
| Outcome misprediction for direct conditional branch | <Ti, Ti.bCnt> |
| Target address misprediction for indirect unconditional branch | <Ti, Ti.bCnt, Ti.BTA> |
| Exception event | <Ti, 0, Ti.iCnt, Ti.ETA> |

The *mcfTRaptor* traces are filtered out to replace the target addresses, Ti.BTA with Ti.diffTA values. The Ti.bCnt and Ti.diffTA fields can take a number of values. Similar to *mcfNX_b* trace encoding, the branch counter field is divided into

multiple 8-bit chunks. If an 8-bit field is sufficient to encode the counter value, the following connect bit C = 0, thus indicating the terminating chunk for Ti.bCnt. Otherwise, C = 1, and the following chunk carries the next 8 bits of the branch counter value. The trace descriptors for target address misprediction events carry information about the correct target address. An alternative to reporting the entire address (64-bit in our case) is to encode the difference between subsequent target addresses and thus exploit locality in programs to minimize the size of trace messages. The trace module maintains the previous target address, that is, the target address of the last mispredicted indirect branch (PTA). When a new target misprediction is detected, the trace module calculates the difference target address, *diffTA*, *diffTA* = *TA* – *PTA* and *PTA* gets the value of current address *TA*, *PTA* = *TA*. The absolute value of *diffTA* is divided into 32-bit chunks, and the connect bit indicates whether one or two 32-bit fields are needed to encode the message. Figure 4.5 (b) shows the encoding of a generic trace descriptor for *mcfTR_b*.

By analyzing profiles of reported counter values (*Ti.bCnt* and *Ti.iCnt*) as well as *diffTA* values, we find that the number of required bits for encoding trace messages can be further minimized by allowing for variable encoding. Instead of using fixed-length chunks for *Ti.bCnt*, we allow for chunks of variable size, *i0*, *i1*, *i2*, as shown in Figure 4.5 (c). Similarly, we can use variable chunk sizes of lengths, *j0*, *j1*, *j2*, for encoding *diffTA*. This encoding approach is called *mcfTR_e*. The length of individual chunks is a design parameter and can be determined empirically. In determining the length of individual chunks, we need to balance the overhead caused by the connect bits and the number of bits wasted in individual chunks. A detailed analysis to find good chunk sizes is performed and selected parameters are used for

all benchmarks. It should be noted that the variable encoding offers an additional level of flexibility to adjust encoding lengths for individual benchmarks or even inside different phases of a single benchmark. However, dynamic adaptation of the field lengths is left for future work.

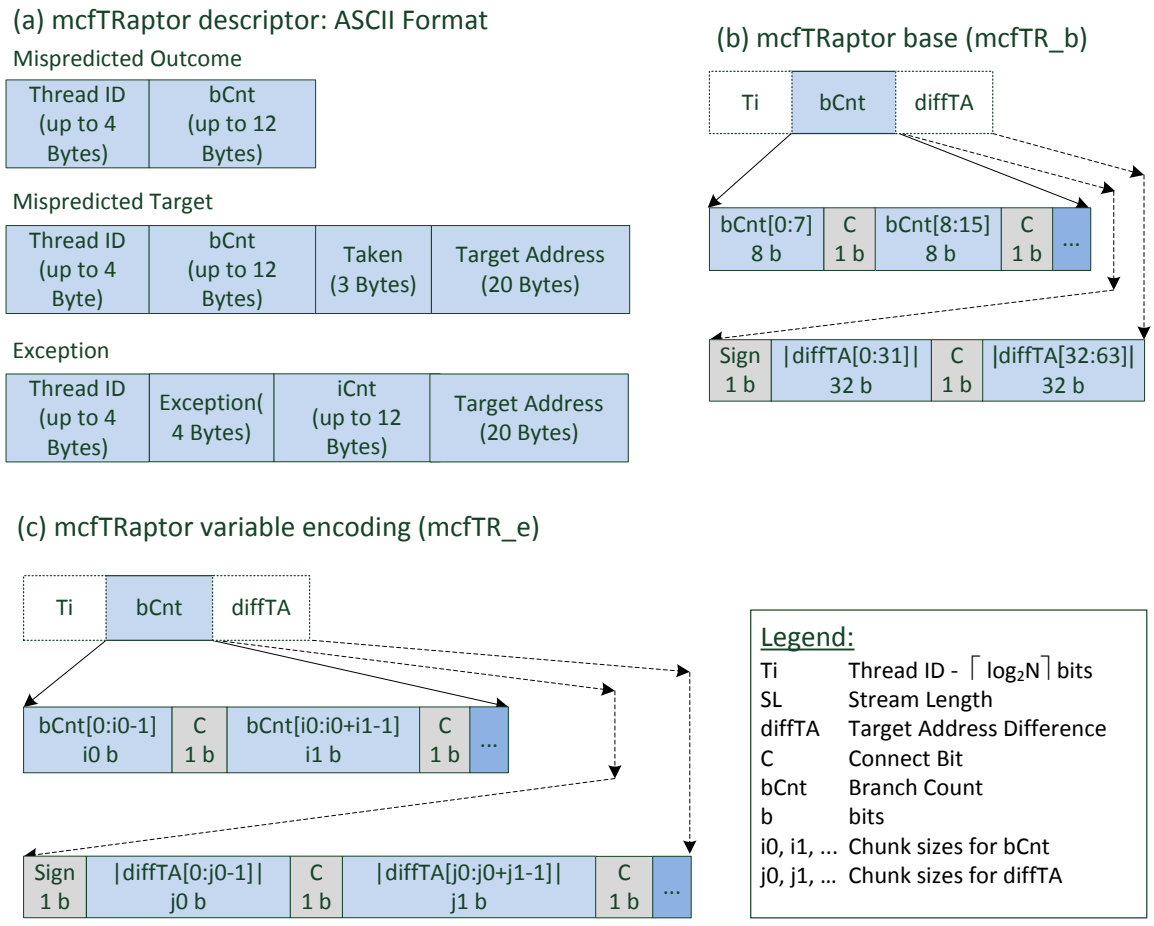


Figure 4.5 *mcfTRaptor*, *mcfTR_b*, and *mcfTR_e* trace descriptors

### *4.2.3 mlvNX_b*

The *mlsTrace* tool generates the load or store trace descriptors as shown in Figure 4.6(a). A descriptor contains information about the thread index, the instruc-

tion address (PC), the operand address, and the operand value. The software debugger needs the following to faithfully replay the program offline: (i) an instruction set simulator (ISS) of the target platform, (ii) access to the program binary, (iii) the initial stage of the general purpose and special purpose registers of the individual cores, and (iv) the load data values read from memory or input/output devices. Thus, the trace generated by the *mlsTrace* tool can be filtered out to remove the load instruction addresses (PC) and the operand address that can be inferred in the ISS. Figure 4.6 (b) shows the trace descriptor for Nexus-like load data value trace. The descriptor contains only the thread index and the load data value. The length of the load value field depends on the size of the operand specified by the instruction, and for the Intel 64 ISA varies between 1 byte and 120 bytes.

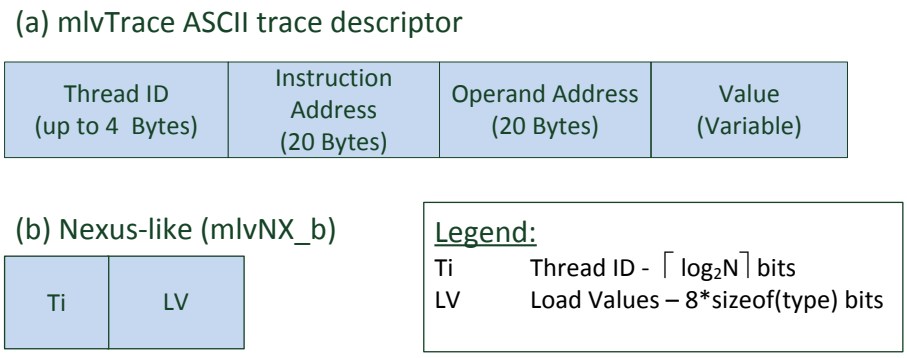


Figure 4.6 *mlvTrace* and *mlvNX_b* trace descriptors

### 4.2.4 *mlvCF_b* and *mlvCF_e*

The *mlvCFiat* tool generates trace descriptors as shown in Figure 4.7(a). A descriptor contains information about the thread index, the first access hit counter (fahCnt – number of consecutive first access hit events), the operand size, and the operand value. The operand size can be inferred by the software debugger with an

instruction set simulator and program binary. Figure 4.7 (b) shows trace descriptors for Nexus-like load data value trace (*mlvCF_b*). The descriptor includes the thread index (Ti), the first access hit counter, Ti.fahCnt, and the load value (Ti.LV). The number of bits needed to encode Ti.fahCnt varies as a function of the first-access miss rate. With *mlvCF_b* we use at least 8 bits to encode the Ti.fahCnt. The connect bit (C) determines whether more 8-bit chunks are needed to fully encode Ti.fahCnt value (C = 1) or not (C = 0).

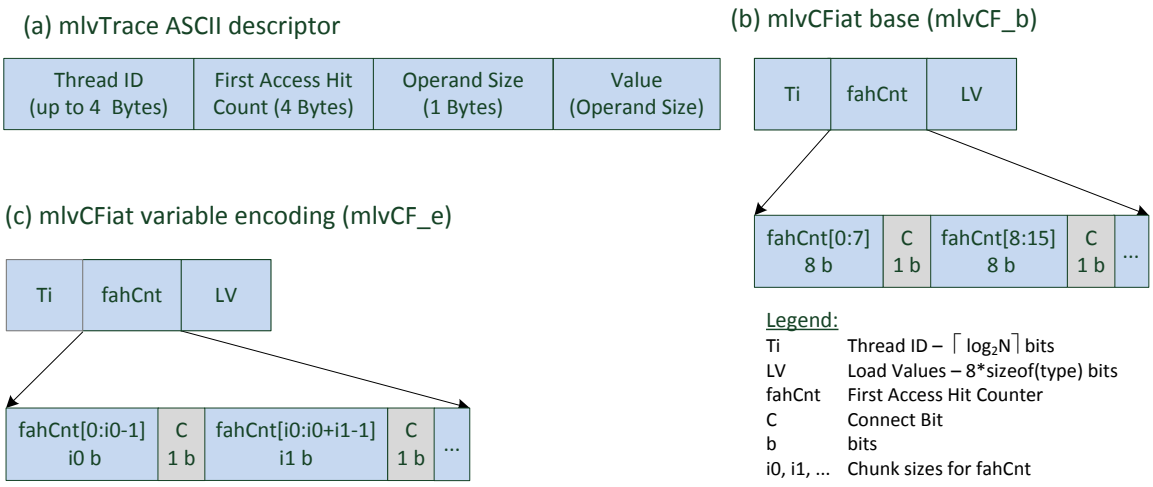


Figure 4.7 *mlvTrace*, *mlvCF_b*, and *mlvCF_e* trace descriptors

With *mlvCF_e*, we allow the length of individual chunks, *i0* and *i1* in Figure 4.7(c), to be between 1 and 8 bits. We evaluate different encoding arrangements to select a pair of good values that minimizes the number of bits needed to encode the counter value.

## 4.3 Experimental Environment

The goal of experimental evaluation is to determine the effectiveness of the newly proposed trace reduction techniques, *mcfTRaptor* and *mlvCFiat*, relative to the baseline Nexus-like control-flow, *mcfNX_b*, and load data value traces, *mlvNX_b*. To explore the effectiveness of *mcfTRaptor* and *mlvCFiat,* their structure sizes and configurations are varied. As a measure of effectiveness, we use the average number of bits emitted on the trace port per instruction executed. As the workload, we use control flow and load data value traces of 14 benchmarks from the Splash2x benchmark suite [28] collected on a machine executing the Intel 64 ISA. Machine setup is described in Section 4.3.1. The benchmarks are discussed in Section 4.3.2. Section 4.3.3 describes experiments conducted and Section 4.3.4 describes selection of encoding parameters for variable encoding mechanism.

### 4.3.1 Experimental Setup

The setup included a Dell PowerEdge T620 server with two octa-core Intel Xeon CPU E5-2650 v2 processors with total of 64 GB physical memory (Figure 4.8). The server runs the Ubuntu 14.04 operating system with 3.13.0-39-generic Linux kernel. The *mTrace* tools used Pin version 2.13.

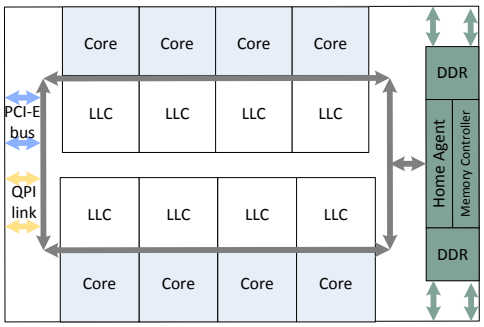


Figure 4.8 Block diagram of Intel Xeon E5-2650 v2 processor socket

### 4.3.2 Benchmarks

As a workload we use a full set of SPLASH2x [28] benchmarks. SPLASH2x is a collection of applications and kernels in the area of high performance and parallel computing. Each benchmark was executed with N = 1, 2, 4 and 8 processor cores. Each benchmark has six different input sets as follows: Test and Simdev are used for testing and development; Simsmall, Simmedium and Simlarge are used for simulations; finally, Native is used for the native program execution on actual hardware platforms. We use the *Simsmall* input set.

The trace port bandwidth for control flow traces depends on benchmark characteristics, specifically the frequency and type of control-flow instructions. Similarly the trace port bandwidth for load data value traces depends on the frequency and type of memory reads and data value sizes. Table 4.2 shows the control flow characterization of all 14 benchmarks in the SPLASH2X suite. The suite was compiled for the Intel 64 instruction set architecture with varying number of thread varies (N = 1, 2, 4, and 8). We show the number of executed instructions (instruction count) and the number of instructions executed per clock cycle (IPC). The last four columns show the frequency of control flow instructions for single threaded programs (N = 1), as well as the frequency of conditional direct branches (C, D), unconditional direct branches (U, D), and unconditional indirect branches (U, I). The number of instructions slightly increases with an increase in the number of threads due to overhead and data partitioning. The number of instructions varies between 0.367 (lu_ncb) ~ 3.2 (water-spatial) billion. The SPLASH2x benchmarks exhibit diverse behavior with respect to control flow instructions; their frequency ranges from as high as ~15% (*raytrace, radiosity*) to as low as 1.06% (*radix*). The total frequency of control

flow instructions is relatively low (9.57%). Conditional direct branches are the most frequent type of branches.

Table 4.2 Splash2x benchmark suite control flow characterization

| *Benchmark* | *Instruction Count [x10$^9$]* | | | | *Instruction per Cycle* | | | | *% branches for Thread=1* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Thread* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* | *branch* | *C,D* | *U,D* | *U,I* |
| *cholesky* | 0.889 | 0.916 | 0.976 | 1.047 | 2.28 | 2.93 | 3.75 | 4.03 | 6.11 | 5.43 | 0.45 | 0.22 |
| *fft* | 0.764 | 0.764 | 0.764 | 0.765 | 1.28 | 1.63 | 1.96 | 2.10 | 8.38 | 5.65 | 1.36 | 1.37 |
| *lu_cb* | 0.381 | 0.381 | 0.382 | 0.383 | 2.93 | 4.89 | 3.67 | 4.91 | 14.30 | 13.74 | 0.28 | 0.28 |
| *lu_ncb* | 0.367 | 0.367 | 0.368 | 0.369 | 2.02 | 2.82 | 3.54 | 2.84 | 14.86 | 14.35 | 0.22 | 0.29 |
| *radix* | 0.703 | 0.704 | 0.704 | 0.707 | 0.33 | 0.59 | 1.18 | 2.26 | 1.06 | 1.06 | 0.00 | 0.00 |
| *barnes* | 1.606 | 1.606 | 1.608 | 1.608 | 1.58 | 2.81 | 4.76 | 6.87 | 13.31 | 7.02 | 4.05 | 2.25 |
| *fmm* | 2.265 | 2.268 | 2.270 | 2.272 | 2.23 | 3.96 | 7.27 | 9.71 | 7.34 | 5.55 | 1.65 | 0.15 |
| *ocean_cp* | 1.316 | 1.317 | 1.334 | 1.337 | 1.58 | 2.41 | 3.02 | 3.21 | 2.91 | 2.47 | 0.40 | 0.04 |
| *ocean_ncp* | 1.293 | 1.294 | 1.308 | 1.311 | 0.51 | 1.35 | 1.80 | 2.52 | 2.59 | 2.57 | 0.02 | 0.001 |
| *radiosity* | 1.402 | 1.433 | 1.434 | 1.434 | 1.74 | 2.63 | 3.45 | 3.94 | 14.59 | 9.01 | 3.37 | 2.21 |
| *raytrace* | 1.646 | 1.646 | 1.649 | 1.651 | 1.47 | 2.34 | 3.34 | 3.17 | 15.27 | 11.28 | 2.14 | 1.85 |
| *volrend* | 0.741 | 0.758 | 0.783 | 0.807 | 1.50 | 2.65 | 3.34 | 3.10 | 6.30 | 5.48 | 0.66 | 0.17 |
| *water_nsq* | 0.431 | 0.432 | 0.433 | 0.435 | 1.84 | 2.77 | 4.16 | 5.58 | 11.74 | 10.32 | 0.71 | 0.70 |
| *water_spa* | 3.221 | 3.221 | 3.221 | 3.221 | 2.03 | 3.10 | 4.96 | 7.29 | 11.38 | 9.38 | 0.97 | 1.02 |
| *Total* | 17.024 | 17.107 | 17.232 | 17.346 | 1.30 | 2.26 | 3.33 | 4.12 | 9.57 | 7.16 | 1.48 | 0.93 |

Table 4.3 shows the memory read flow characterization of Splash2x benchmarks while varying the number of threads (N = 1, 2, 4, and 8). We show the number of instructions executed, the number of instructions executed per clock cycle, and the frequency of memory read instructions with respect to the total number of instructions. The frequency of memory read instructions increases with an increase in the number of threads. The percentage of memory read instructions varies between 10.61% (*radix*) and 35.73% (*barnes*). The overall frequency of read instructions is 27.46% for N = 1 and 28.8% for N = 8. The overall IPC as a function of the number of cores indicates how well performance of individual benchmarks scales. For exam-

ple, *radix* scales well, reaching a speedup for 8 cores, S(8) = IPC(N= 8)/IPC(N=1) = 6.6, but *lu_ncb* does not scale well because its 8-core speedup is only S(8) = 1.4.

Table 4.3 Splash2x benchmark suite memory read characterization

| *Benchmark* | *Instruction Count [x10$^9$]* | | | | *Instructions per Cycle* | | | | *%Load* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Thread* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* |
| *cholesky* | 0.89 | 0.92 | 0.98 | 1.05 | 2.28 | 2.93 | 3.75 | 4.03 | 29.02 | 29.59 | 32.24 | 37.18 |
| *fft* | 0.76 | 0.76 | 0.76 | 0.76 | 1.28 | 1.63 | 1.96 | 2.10 | 21.61 | 21.61 | 21.62 | 21.62 |
| *lu-cb* | 0.38 | 0.38 | 0.38 | 0.38 | 2.93 | 4.89 | 3.67 | 4.91 | 26.03 | 26.04 | 26.07 | 26.11 |
| *lu-ncb* | 0.37 | 0.37 | 0.37 | 0.37 | 2.02 | 2.82 | 3.54 | 2.84 | 26.70 | 26.71 | 26.73 | 26.78 |
| *radix* | 0.70 | 0.70 | 0.70 | 0.71 | 0.33 | 0.59 | 1.18 | 2.26 | 10.61 | 10.64 | 10.69 | 10.81 |
| *barnes* | 1.61 | 1.61 | 1.61 | 1.61 | 1.58 | 2.81 | 4.76 | 6.87 | 35.72 | 35.72 | 35.72 | 35.73 |
| *fmm* | 2.27 | 2.27 | 2.27 | 2.27 | 2.23 | 3.96 | 7.27 | 9.71 | 15.61 | 15.63 | 15.65 | 15.71 |
| *ocean-cp* | 1.32 | 1.32 | 1.33 | 1.34 | 1.58 | 2.41 | 3.02 | 3.21 | 34.92 | 34.96 | 35.31 | 35.42 |
| *ocean-ncp* | 1.29 | 1.29 | 1.31 | 1.31 | 0.51 | 1.35 | 1.80 | 2.52 | 29.59 | 29.60 | 29.85 | 29.89 |
| *radiosity* | 1.40 | 1.43 | 1.43 | 1.43 | 1.74 | 2.63 | 3.45 | 3.94 | 30.42 | 31.03 | 30.90 | 31.02 |
| *raytrace* | 1.65 | 1.65 | 1.65 | 1.65 | 1.47 | 2.34 | 3.34 | 3.17 | 31.02 | 31.02 | 31.06 | 31.07 |
| *volrend* | 0.74 | 0.76 | 0.78 | 0.81 | 1.50 | 2.65 | 3.34 | 3.10 | 24.88 | 25.97 | 27.63 | 29.22 |
| *water-nsq* | 0.43 | 0.43 | 0.43 | 0.44 | 1.84 | 2.77 | 4.16 | 5.58 | 29.22 | 29.26 | 29.32 | 29.44 |
| *water-spa* | 3.22 | 3.22 | 3.22 | 3.22 | 2.03 | 3.10 | 4.96 | 7.29 | 29.92 | 29.92 | 29.92 | 29.92 |
| *Total* | 17.02 | 17.11 | 17.23 | 17.35 | 1.30 | 2.26 | 3.33 | 4.12 | 27.46 | 27.60 | 27.86 | 28.22 |

The memory trace port bandwidth depends not only on the frequency of read operations but also on operand sizes. Table 4.4 shows the frequency of memory reads for different operand sizes: byte (8 bit), words (16 bit), doubleword (32 bit), quadword (64 bit), extended precision (80 bit) operands, octaword (128 bit) operands, hexaword (256 bit) operands, and others. The results indicate that quad-word operands dominate in all benchmarks except for *radiosity* which has mostly double word operands.

Table 4.4 Benchmark characterization of memory reads

| Benchmark | Total Memory Reads | Byte Operands | Word operands | Doubleword operands | Quadword operands | Extended Precision operands | Octaword operands | Hexaword operands | OtherSize operands |
|---|---|---|---|---|---|---|---|---|---|
| *cholesky* | 257829613 | 1.41 | 0.01 | 0.37 | 93.55 | 0.00 | 0.50 | 0.00 | 4.16 |
| *fft* | 165033526 | 0.02 | 10.17 | 0.01 | 85.33 | 0.00 | 0.64 | 0.00 | 3.83 |
| *lu_cb* | 99130206 | 0.02 | 1.85 | 0.98 | 97.15 | 0.00 | 0.00 | 0.00 | 0.00 |
| *lu_ncb* | 97978154 | 0.02 | 1.88 | 0.36 | 97.75 | 0.00 | 0.00 | 0.00 | 0.00 |
| *radix* | 74608179 | 0.02 | 0.00 | 0.01 | 95.74 | 0.00 | 0.00 | 0.00 | 4.22 |
| *barnes* | 573750613 | 0.00 | 0.00 | 0.10 | 99.86 | 0.00 | 0.03 | 0.00 | 0.01 |
| *fmm* | 353635215 | 0.00 | 0.00 | 0.48 | 96.64 | 0.00 | 0.25 | 0.00 | 2.62 |
| *ocean_cp* | 459344111 | 0.00 | 0.00 | 0.01 | 94.42 | 0.00 | 0.00 | 0.00 | 5.57 |
| *ocean_ncp* | 382545985 | 0.00 | 0.00 | 0.01 | 98.37 | 0.00 | 0.62 | 0.00 | 1.00 |
| *radiosity* | 426368828 | 0.01 | 0.00 | 65.27 | 34.60 | 0.00 | 0.00 | 0.00 | 0.12 |
| *raytrace* | 510473124 | 0.96 | 0.00 | 1.28 | 97.74 | 0.00 | 0.02 | 0.00 | 0.00 |
| *volrend* | 184409952 | 15.50 | 10.90 | 38.24 | 35.34 | 0.00 | 0.01 | 0.00 | 0.00 |
| *water_nsq* | 125938251 | 0.46 | 0.00 | 0.27 | 99.00 | 0.00 | 0.27 | 0.00 | 0.00 |
| *water_spa* | 963693545 | 0.39 | 0.00 | 0.19 | 98.78 | 0.00 | 0.63 | 0.00 | 0.00 |
| *Total* | 4674739302 | 0.89 | 0.87 | 7.75 | 88.96 | 0.00 | 0.26 | 0.00 | 1.27 |

### 4.3.3 Experiments

Table 4.5 lists the pairs (technique, configuration) considered in the experimental evaluation. For control-flow traces we compare the trace port bandwidth of *mcfNX_b* versus *mcfTR_b* and *mcfTR_e*, while varying the number of threads (N=1, 2, 4, and 8). To assess the impact of organization and size of predictor structures in *mcfTRaptor* on its effectiveness, we consider the following configurations:

- *Small*: it includes a p=512-entry *gshare* outcome predictor and an 8-entry RAS;
- *Medium*: it includes an p=1024-entry *gshare* outcome predictor, a 16-entry RAS, and a 16-entry iBTB (2x8); and
- *Large*: it includes a p=4096-entry *gshare* outcome predictor, a 32-entry RAS, and a 64-entry iBTB (2x32).

The index function for the *gshare* outcome predictor is *gshare.index* = *BHR[log2(p):0] xor PC[4+ log2(p):4]*, where the BHR register holds the outcome history of the last *log2(p)* conditional branches. The iBTB holds target addresses that are tagged. Both the iBTB tag and iBTB index are calculated based on the information maintained in the path information register [29], [30].

Table 4.5 Functional trace experiments

| *control flow (vary N = 1,2,4 & 8)* | | | | *load flow (vary N = 1,2,4 & 8)* | | | |
|---|---|---|---|---|---|---|---|
| *Method* | *Small* | *Medium* | *Large* | *Method* | *Small* | *Medium* | *Large* |
| *mcf_NX_b* | √ | | | *mlv_NX_b* | √ | | |
| *mcf_TR_b* | √ | √ | √ | *mlv_CF_b* | √ | √ | √ |
| *mcf_TR_e* | √ | √ | √ | *mlv_CF_e* | √ | √ | √ |

For load data value traces, we compare the trace port bandwidth of *mlvNX_b* versus the *mlvCF_b* and *mlvCF_e* while varying the number of threads (N=1, 2, 4, and 8). To assess the impact of organization and size data caches on *mlvCFiat* effectiveness, we consider the following cache configurations. All cache structures are 4-way set associative, use round robin replacement policy, feature a block size of 64 bytes, and the first-access flag granularity is set to 4 bytes. We consider three configurations as follows:

- *Small*: 16 KB data cache;
- *Medium*: 32 KB data cache; and
- *Large*: 64 KB data cache.

### 4.3.4 Variable Encoding

To evaluate the impact of encoding mechanisms, we analyze trace port bandwidth for both encoding approaches *mcfTR_b* and *mcfTR_e*. To select good encoding parameters (i0, i1, j0, j1, …), we profiled the Splash2x benchmarks to determine the minimum required bit length of Ti.bCnt and Ti.|diffTA| fields. Figure 4.9 shows the cumulative distribution function (CDF) for the minimum number of bits needed to encode Ti.bCnt (left) and Ti.|diffTA| (right) for the *raytrace*, *radiosity*, and *fft* benchmarks with N = 1 and the *Large mcfTRaptor* configuration. These benchmarks are selected because they have a relatively high frequency of control-flow instructions. The number of bits needed to encode the value of Ti.bCnt counters depends on benchmark characteristics as well as on misprediction rates of the *mcfTRaptor* predictors, which makes the selection of good parameters a challenging task. However, we can see that ~60% of possible Ti.bCnt values encountered during tracing *raytrace* and *radiosity* can be encoded with 3 bits, and very few trace descriptors require more than 8 bits. Similarly, over 60% of Ti.|diffTA| values encountered in the trace require fewer than 16 bits to encode for *raytrace* and *radiosity*, and just 3 bits are sufficient in case of *fft* that has a relatively high frequency of indirect branches.

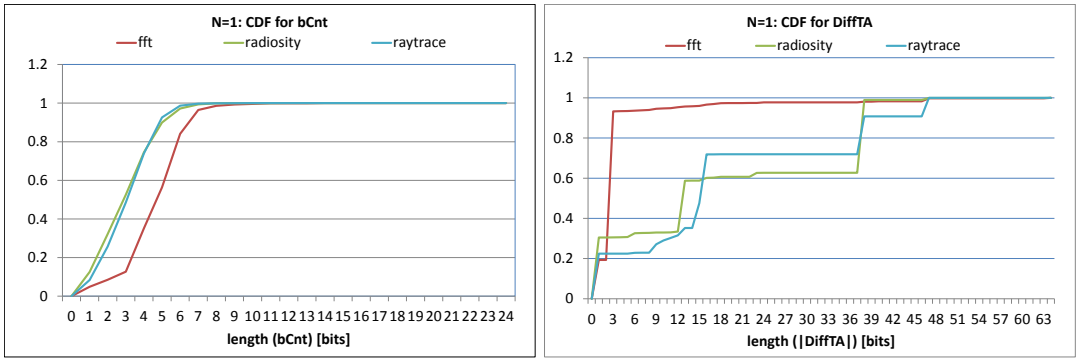


Figure 4.9 CDF of the minimum length for bCnt and |diffTA| fields

Whereas each (benchmark, *mcfTRaptor* configuration) pair may yield an optimal combination of the encoding parameters, we search for a combination that performs well across all benchmarks. We limit the design space by requiring that *i1*=*i2*=… *ik*, and *j1*=*j2*=… =*jl*, where {*i0*, *i1*}={1-6} and {*j0*, *j1*}={1-12}. Figure 4.10 shows the total average size of the bCnt and |diffTA| fields for a selected set of chunk sizes for N = 1, 2, 4, and 8. We find that *i0* = 4, *i1* = 2 results in the smallest average bCnt field size, regardless of the number of threads. Similarly, we find that *j0* = 3 and *j1* = 5 are chunk sizes that result in the smallest average |diffTA| field size.

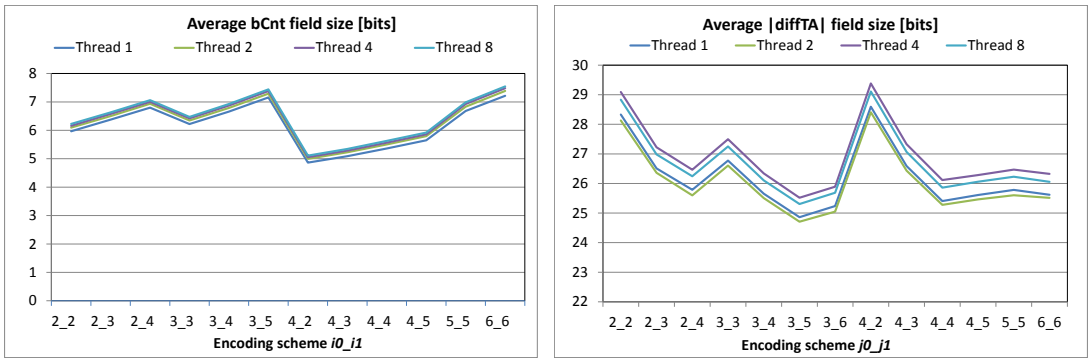


Figure 4.10 Total average bCnt and |diffTA| field sizes as a function of encoding

To evaluate the impact of encoding mechanisms for load data value traces, we analyze trace port bandwidth for both encoding approaches *mlvCF_b* and *mlvCF_e*. To select good encoding parameters (i0, i1), we profiled the Splash2x benchmarks to determine the minimum required bit length of the Ti.fahCnt field. Figure 4.11, left shows the cumulative distribution function of the Ti.fahCnt field length for selected

single threaded Splash2x benchmarks. The number of bits needed to encode the Ti.fahCnt varies as a function of benchmark characteristics (frequency load values, locality), cache configuration, and first-access granularity. We can see that more than 60% of trace descriptors require fewer than 2 bits for encoding Ti.fahCnt, and more than 90% of descriptors require fewer than 6 bits. Figure 4.11, right shows the average Ti.fahCnt field size as a function of chunk sizes (i0, i1) = {(2,2), ... (6,6)}. The results indicate that $i0 = 2$ and $i1 = 2$ chunk sizes give the shortest field sizes regardless of the number of threads.

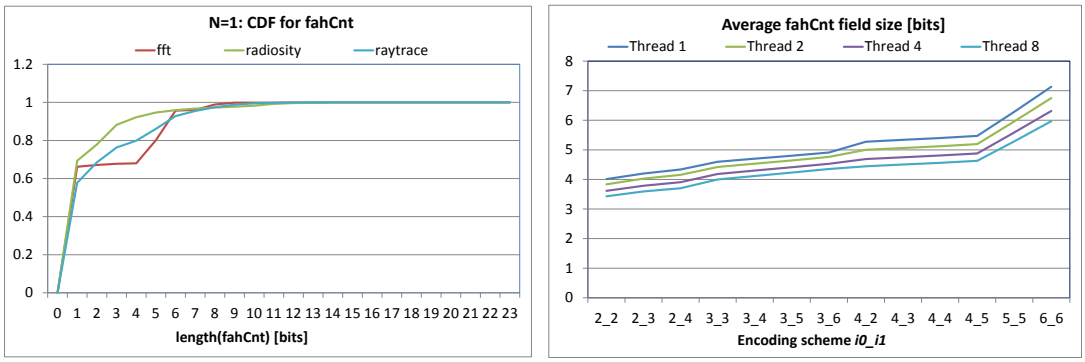


Figure 4.11 CDF of the minimum length for Ti.fahCnt and the average Ti.fahCnt for variable encoding

# CHAPTER 5

## TRACE PORT BANDWIDTH ANALYSIS FOR FUNCTIONAL TRACES

Those who see invincible, can do impossible

--Rev. Pandurang Shastri Athavale

This chapter shows the main results of the experimental evaluation for functional traces. We measure trace port bandwidths for control-flow and load data value traces as a function of the number of processor cores, encoding mechanism, as well as configuration parameters of the trace filtering structures. Trace port bandwidth is measured in bits per instruction executed [*bpi*], calculated as the number of bits needed to be streamed out through the trace port of a system-on-a-chip divided by the number of instructions executed. In addition, we consider bits per clock cycles [*bpc*], calculated as the total number of bits streamed out through the trace port of a system-on-the-chip divided by the number of clock cycles needed to complete a benchmark of interest. Section 5.1 discusses the results for control-flow functional traces, specifically the trace port bandwidth requirements for the Nexus-like control-flow trace, *mcfNX_b*, as well as the trace port bandwidth for the *mcfTRaptor* technique with the fixed encoding, *mcfTR_b*, and the variable encoding, *mcfTR_e*. Section 5.2 discusses the results for memory load data value functional traces, specifically the trace port bandwidth requirements for the Nexus-like traces, *mlvNX_b*, and the *mlvCFiat* technique with the fixed, *mlvCF_b*, and the variable encoding, *mlvCF_e*.

## 5.1 Trace Port Bandwidth for Control-Flow Traces

### *5.1.1 mcfNX_b*

Table 5.1 shows the trace port bandwidth (TPB) in bpi and bpc for the Nexus-like control flow traces, *mcfNX_b*, for all the benchmarks as a function of the number of threads/cores (N = 1, 2, 4, and 8). The last row shows the total trace port bandwidth when all benchmarks are considered together. The total bandwidth in bpi is calculated as the sum of trace sizes for all benchmarks divided by the sum of the number of instructions executed for all benchmarks. Similarly, the total bandwidth in bpc is calculated as the sum of trace sizes for all benchmarks divided by the sum of the execution times in clock cycles for all benchmarks. For single-threaded benchmarks (N = 1) the TPB ranges between 0.09 bpi for *radix* and 1.67 bpi for *radiosity*. The required bandwidth varies across benchmarks and is highly correlated with the frequency of control-flow instructions. Thus, *radiosity*, *raytrace*, *water-spatial* and *barnes* have relatively high TPB in bpi requirements due to the relatively high frequency of branch instructions and especially indirect branches (see Table 4.2), conversely, *radix,* has very low TPB in bpi requirements due to the extremely small frequency of control flow instructions. The required trace port bandwidth in bits per instruction increases as we increase the number of cores, due to additional information such as Ti that needs to be streamed out. Thus, when N = 8, the TPB ranges between 0.13 bpi for *radix* and 1.98 bpi for *radiosity*. The total bandwidth for the entire benchmark suite ranges between 0.93 bpi when N = 1 and 1.14 bpi when N = 8.

Table 5.1 Trace port bandwidth for *mcfNX_b* for Splash2x benchmark

| Benchmark | Trace Port Bandwidth [bpi] | | | | Trace Port Bandwidth [bpc] | | | |
|---|---|---|---|---|---|---|---|---|
| N (cores) | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 |
| cholesky | 0.40 | 0.46 | 0.51 | 0.60 | 0.92 | 1.35 | 1.93 | 2.40 |
| fft | 1.18 | 1.23 | 1.28 | 1.32 | 1.51 | 2.00 | 2.50 | 2.78 |
| lu_cb | 1.25 | 1.38 | 1.50 | 1.62 | 3.67 | 6.72 | 5.50 | 7.95 |
| lu_ncb | 1.31 | 1.43 | 1.56 | 1.69 | 2.63 | 4.05 | 5.53 | 4.79 |
| radix | 0.09 | 0.11 | 0.12 | 0.13 | 0.03 | 0.06 | 0.14 | 0.30 |
| barnes | 1.65 | 1.75 | 1.85 | 1.94 | 2.62 | 4.92 | 8.79 | 13.36 |
| fmm | 0.41 | 0.45 | 0.48 | 0.52 | 0.91 | 1.77 | 3.53 | 5.09 |
| ocean_cp | 0.20 | 0.22 | 0.24 | 0.27 | 0.32 | 0.53 | 0.73 | 0.86 |
| ocean_ncp | 0.20 | 0.22 | 0.25 | 0.27 | 0.10 | 0.30 | 0.45 | 0.69 |
| radiosity | 1.67 | 1.78 | 1.88 | 1.98 | 2.90 | 4.66 | 6.48 | 7.80 |
| raytrace | 1.45 | 1.54 | 1.63 | 1.71 | 2.14 | 3.61 | 5.43 | 5.44 |
| volrend | 0.63 | 0.74 | 0.88 | 1.04 | 0.95 | 1.95 | 2.96 | 3.23 |
| water_nsq | 0.98 | 1.06 | 1.14 | 1.22 | 1.81 | 2.94 | 4.75 | 6.82 |
| water_spa | 1.20 | 1.28 | 1.36 | 1.45 | 2.44 | 3.97 | 6.76 | 10.53 |
| **Total** | **0.93** | **1.00** | **1.07** | **1.14** | **1.21** | **2.26** | **3.55** | **4.68** |

Whereas the bandwidth in bpi increases with the number of cores, it does not fully capture the pressure multiple processor cores place on the trace port, a shared resource. The trace port bandwidth in bits per clock cycle better illustrates this pressure. Thus, control-flow trace of *barnes* with 8 threads executing on 8 cores requires 13.36 bpc on average. Generally, the trace port bandwidth in bpc is a function of benchmark characteristics as well as the scalability of individual benchmarks. The total TPB in bpc ranges between 1.21 bpc when N = 1 and 4.68 when N = 8. These results indicate that capturing control-flow trace on the fly in multi-cores requires significantly large trace buffers and wide trace ports. As shown in the next section, one alternative is to develop hardware techniques that significantly reduce the volume and size of trace messages that are streamed out.

### *5.1.2 mcfTRaptor*

The effectiveness of *mcfTRaptor* in reducing the trace port bandwidth depends on prediction rates as the trace messages are generated only on rare misprediction events. Table 5.2 shows the total misprediction rates collected on the entire benchmark suite for the *Small*, *Medium*, and *Large* predictor configurations, when the number of cores is varied between N = 1 and N = 8. Figure 5.1 illustrates the total outcome misprediction rates and Figure 5.2 shows the total target address misprediction rates as a function of the number of threads and predictor configuration. The outcome misprediction rates decrease as we increase the size of the *gshare* predictor. They also slightly decrease with an increase in the number of processor cores as fewer branches compete for the same resource. Relatively high misprediction rates indicate that even better trace compression could be achieved if more sophisticated outcome predictors are used. However, this is out of scope of this thesis. The target address misprediction rates are very low for the Medium and Large configurations. The *Small* configuration does not include the iBTB predictor resulting in higher target address misprediction rates. These results demonstrate a strong potential of *mcfTRaptor* to reduce the trace port bandwidth requirements.

Table 5.2 Outcome and target address misprediction rates

| | Outcome Misprediction Rate [%] | | | | Target Address Misprediction Rate [%] | | | |
|---|---|---|---|---|---|---|---|---|
| Configuration | N = 1 | N = 2 | N = 4 | N = 8 | N = 1 | N = 2 | N = 4 | N = 8 |
| *Small* | 8.20 | 7.95 | 7.95 | 7.54 | 21.78 | 21.84 | 21.86 | 21.88 |
| *Medium* | 6.85 | 6.88 | 6.60 | 6.51 | 2.69 | 2.70 | 2.72 | 2.74 |
| *Large* | 5.16 | 5.10 | 5.00 | 4.84 | 0.77 | 0.78 | 0.79 | 0.78 |

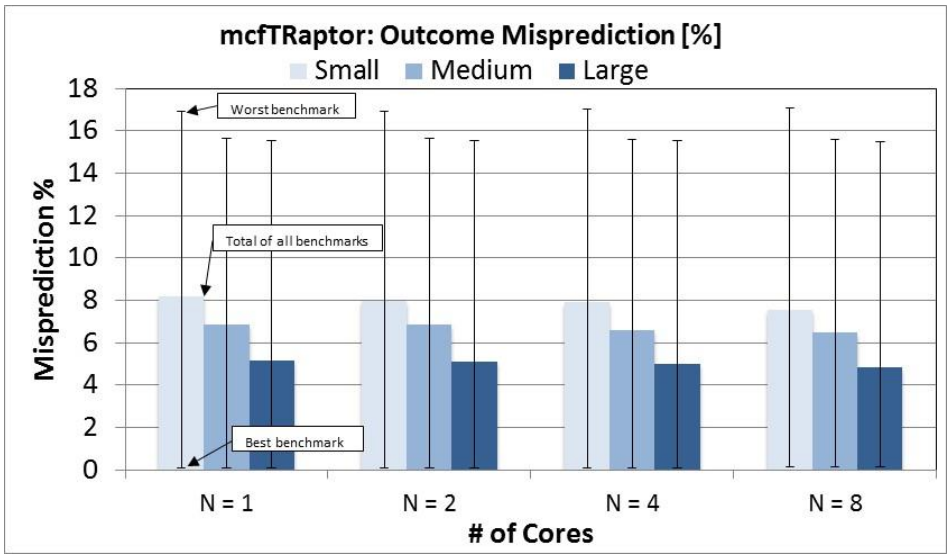


Figure 5.1 Outcome misprediction rates for Splash2x bechmark

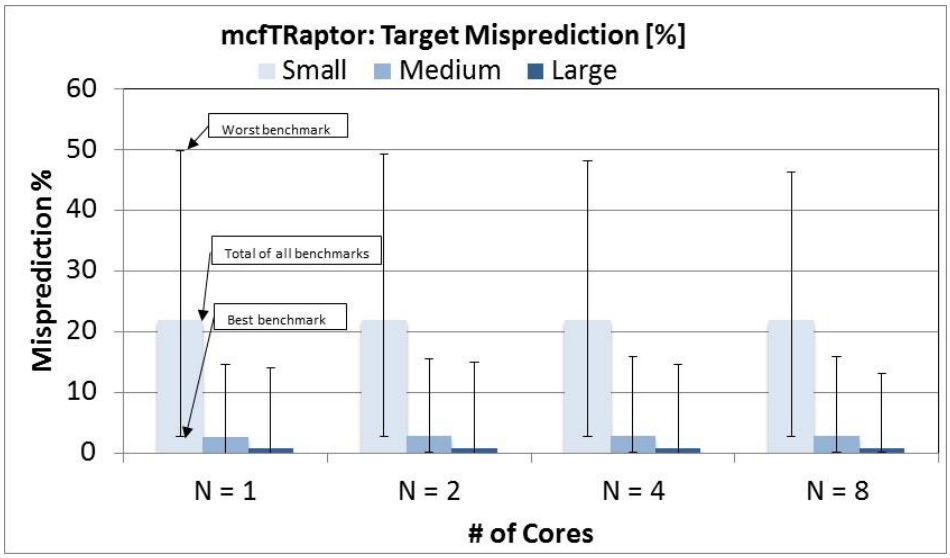


Figure 5.2 Target address misprediction rates for Splash2x benchmark

To quantify the effectiveness of *mcfTRaptor*, we analyze the total trace port bandwidth in bpi for the entire benchmark suite as a function of the number of threads (N = 1, 2, 4, and 8), the encoding mechanism (*mcfTR_b* and *mcfTR_e*), and the *mcfTRaptor* organization (*Small*, *Medium*, and *Large*)[Section 4.3.3]. Figure 5.3 shows the total average trace port bandwidth.

*mcfTR_b* dramatically reduces the total trace port bandwidth compared to *mcfNX_b*. Specifically, we observe the following findings.

- *Small* configuration: 0.14 bpi (N = 1) and 0.16 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 6.65 times for N = 1 and 6.98 times for N = 8.
- *Medium* configuration: 0.05 bpi (N = 1) and 0.07 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 16.56 times for N = 1 and 15.40 for N = 8.
- *Large* configuration: 0.03 bpi (N = 1) and 0.04 (N = 8). This equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 24.88 times for N = 1 and 22.92 for N = 8.

*mcfTR_e* further reduces the average trace port bandwidth as follow:

- *Small* configuration: 0.07 bpi (N = 1) and 0.09 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 12.45 times for N = 1 and 11.58 for N = 8.
- *Medium* configuration: 0.03 bpi (N = 1) and 0.05 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 26.56 times for N = 1 and 21.6 for N = 8.

- *Large* configuration: 0.02 bpi (N = 1) and 0.03 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *mcfNX_b* 36.5 times for N = 1 and 30.3 for N = 8.

Table 5.3 shows the compression ratios for *mcfTR_b* relative to *mcfNX_b*, as a function of the predictor configuration (*Small*, *Medium*, *Large*) and the number of threads for each benchmark. The compression ratio is calculated as follows: TPB(*mcfNX_b*)/TPB(*mcfTR_b*). For N = 1, the compression ratio ranges from 3.5 (*radiosity*) to 699.6 (*radix*) for the *Small* configuration and from 6.5 (*lu_ncb*) to 948.8 (*radix*) for the Large configuration. For N = 8, the compression ratio ranges from 3.5 (*radiosity*) to 399.7 (*radix*) for the *Small* configuration and from 6.3 (*lu_ncb*) to 565.2 (*radix*) for the *Large* configuration. The gains in compression ratio achieved when increasing the number of cores (threads) are relatively more significant when using smaller predictor structures.

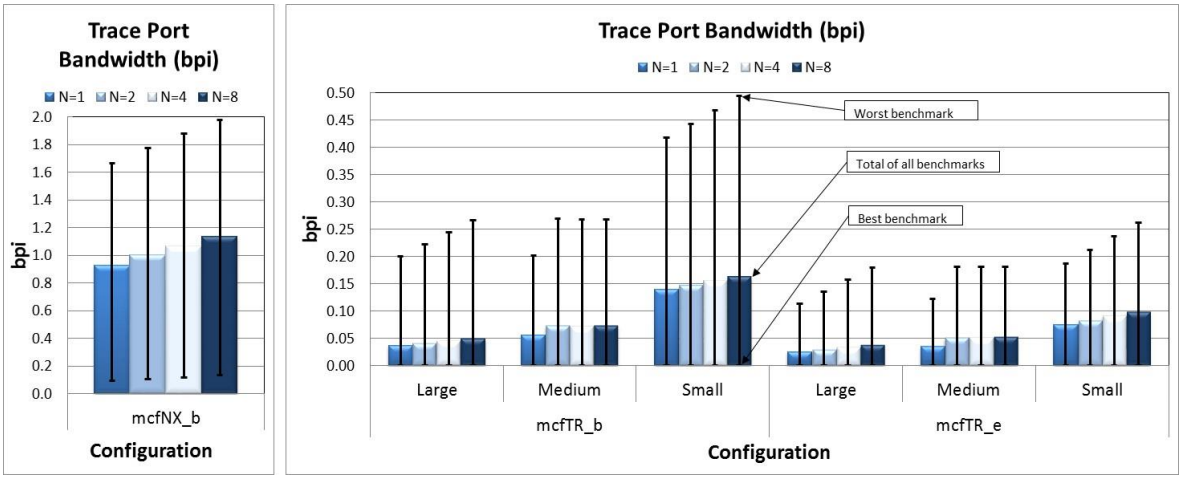


Figure 5.3 Total trace port bandwidth in bpi for control flow traces

Table 5.3 Compression ratio for *mcfTR_b* relative to *mcfNX_b*

| *Cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *cholesky* | 7.8 | 13.9 | 19.9 | 8.1 | 11.7 | 19.6 | 8.5 | 13.0 | 19.7 | 9.2 | 15.0 | 20.7 |
| *fft* | 6.1 | 83.0 | 167.0 | 6.2 | 65.2 | 156.8 | 6.3 | 66.9 | 146.9 | 6.3 | 68.7 | 137.5 |
| *lu_cb* | 11.8 | 16.4 | 16.5 | 11.9 | 13.5 | 16.3 | 12.0 | 14.8 | 16.2 | 12.1 | 15.9 | 16.1 |
| *lu_ncb* | 5.6 | 6.5 | 6.5 | 5.6 | 5.3 | 6.4 | 5.6 | 5.8 | 6.4 | 5.6 | 6.3 | 6.3 |
| *radix* | 699.6 | 869.0 | 948.8 | 598.3 | 644.1 | 811.1 | 512.3 | 602.5 | 704.2 | 399.7 | 528.7 | 565.2 |
| *barnes* | 20.1 | 31.6 | 38.5 | 19.6 | 25.2 | 36.6 | 19.1 | 26.6 | 35.0 | 18.7 | 27.9 | 33.7 |
| *fmm* | 7.5 | 14.4 | 19.0 | 7.7 | 12.6 | 19.1 | 7.9 | 13.7 | 18.8 | 8.1 | 14.4 | 18.7 |
| *ocean_cp* | 15.5 | 53.9 | 56.9 | 15.7 | 42.1 | 54.6 | 14.0 | 32.6 | 39.0 | 13.1 | 31.2 | 36.3 |
| *ocean_ncp* | 72.9 | 82.4 | 86.8 | 59.1 | 56.7 | 73.6 | 36.4 | 38.5 | 46.4 | 28.0 | 32.8 | 38.2 |
| *radiosity* | 6.5 | 11.1 | 22.9 | 6.4 | 9.5 | 21.6 | 6.4 | 10.0 | 21.2 | 6.4 | 10.6 | 20.6 |
| *raytrace* | 3.5 | 7.2 | 12.2 | 3.5 | 6.1 | 11.9 | 3.5 | 6.5 | 11.5 | 3.5 | 6.8 | 11.3 |
| *volrend* | 5.3 | 9.8 | 12.5 | 5.7 | 8.9 | 13.7 | 6.5 | 10.6 | 15.3 | 7.3 | 12.6 | 16.4 |
| *water_ns* | 8.1 | 18.7 | 22.3 | 8.2 | 15.1 | 21.6 | 8.3 | 16.2 | 21.0 | 8.3 | 17.5 | 20.7 |
| *water_sp* | 6.4 | 78.1 | 117.1 | 6.8 | 48.1 | 114.4 | 6.8 | 67.6 | 109.0 | 7.3 | 53.9 | 107.1 |
| *Total* | 6.7 | 16.6 | 24.9 | 6.8 | 13.7 | 24.1 | 6.8 | 14.8 | 23.4 | 7.0 | 15.4 | 22.9 |

Table 5.4 shows the compression ratios for *mcfTR_e* relative to *mcfNX_b*. *mcfTR_e* achieves higher compression ratios than *mcfTR_b*, especially when using the Small predictor structures that exhibit a relatively high number of mispredictions and thus report Ti.bCnt values that can benefit from variable encoding. For N = 1, the compression ratio ranges from 7.8 (*radiosity*) to 1224.9 (*radix*) for the Small configuration and from 11.5 (*lu_ncb*) to 975.9 (*radix*) for the Large configuration. For N = 8, the compression ratio ranges from 6.5 (*radiosity*) to 1035.2 (*radix*) for the Small configuration and from 9.4 (*lu_ncb*) to 805.1 (*radix*) for the Large configuration.

Table 5.4 Compression ratio for *mcfTR_e* relative to *mcfNX_b*

| *Cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *cholesky* | 15.0 | 20.5 | 27.5 | 14.4 | 15.3 | 26.0 | 14.1 | 17.0 | 25.4 | 14.6 | 19.7 | 26.2 |
| *fft* | 23.1 | 123.1 | 209.3 | 21.6 | 86.3 | 191.6 | 20.4 | 88.8 | 177.1 | 19.3 | 90.9 | 162.8 |
| *lu_cb* | 17.7 | 19.6 | 19.7 | 17.2 | 15.4 | 19.1 | 16.8 | 16.8 | 18.6 | 16.4 | 18.1 | 18.3 |
| *lu_ncb* | 10.7 | 11.4 | 11.5 | 9.9 | 7.9 | 10.5 | 9.3 | 8.6 | 9.9 | 8.8 | 9.3 | 9.4 |
| *radix* | 1224.9 | 1417.3 | 1549.8 | 975.9 | 924.0 | 1246.6 | 796.5 | 862.4 | 1035.2 | 599.4 | 754.1 | 805.1 |
| *barnes* | 33.8 | 46.2 | 52.2 | 31.4 | 33.1 | 48.0 | 29.4 | 34.9 | 44.6 | 27.8 | 36.6 | 42.0 |
| *fmm* | 11.3 | 19.5 | 24.8 | 11.2 | 15.9 | 24.2 | 11.2 | 17.2 | 23.4 | 11.3 | 18.3 | 22.9 |
| *ocean_cp* | 37.4 | 63.9 | 65.9 | 35.1 | 48.7 | 63.1 | 26.8 | 38.0 | 44.9 | 23.5 | 37.4 | 42.0 |
| *ocean_ncp* | 78.5 | 83.7 | 86.2 | 65.0 | 58.6 | 73.7 | 41.5 | 41.5 | 48.4 | 33.4 | 37.4 | 41.5 |
| *radiosity* | 13.6 | 21.2 | 35.9 | 12.5 | 15.2 | 32.3 | 11.9 | 16.1 | 30.3 | 11.3 | 17.0 | 28.5 |
| *raytrace* | 7.8 | 11.9 | 18.9 | 7.3 | 9.0 | 17.6 | 6.9 | 9.5 | 16.5 | 6.5 | 10.0 | 15.6 |
| *volrend* | 8.4 | 15.5 | 18.9 | 8.7 | 12.2 | 19.5 | 9.6 | 14.6 | 21.1 | 10.5 | 17.4 | 21.9 |
| *water_ns* | 12.3 | 26.9 | 30.7 | 12.0 | 19.6 | 28.8 | 11.9 | 21.1 | 27.2 | 11.7 | 22.6 | 26.1 |
| *water_sp* | 9.7 | 103.8 | 148.0 | 10.2 | 57.8 | 140.6 | 10.0 | 83.6 | 132.5 | 10.5 | 64.9 | 128.4 |
| *Total* | 12.5 | 26.6 | 36.5 | 12.1 | 19.3 | 33.9 | 11.7 | 20.9 | 31.8 | 11.6 | 21.7 | 30.3 |

Figure 5.4 shows the total trace port bandwidth in bits per clock cycle for *mcfNX_b* (left), *mcfTR_b* and *mcfTR_e* (right). *mcfTR_e* offers superior performance. Thus, the *mcfTR_e* for the Large configuration when N = 8 requires merely 0.154 bpc on average (ranging from ~0 to 0.51 bpc), whereas NX_b requires 4.68 bpc (ranging between 0.30 to 13.36 bpc). These results further underscore the effectiveness of the proposed *mcfTRaptor* predictor structures for a range of diverse benchmarks. The results indicate that with *mcfTR_e,* even a single-bit data trace port would be sufficient to stream out the control-flow trace from an 8-core system-on-a-chip, thus dramatically reducing the cost of on-chip debugging infrastructure.

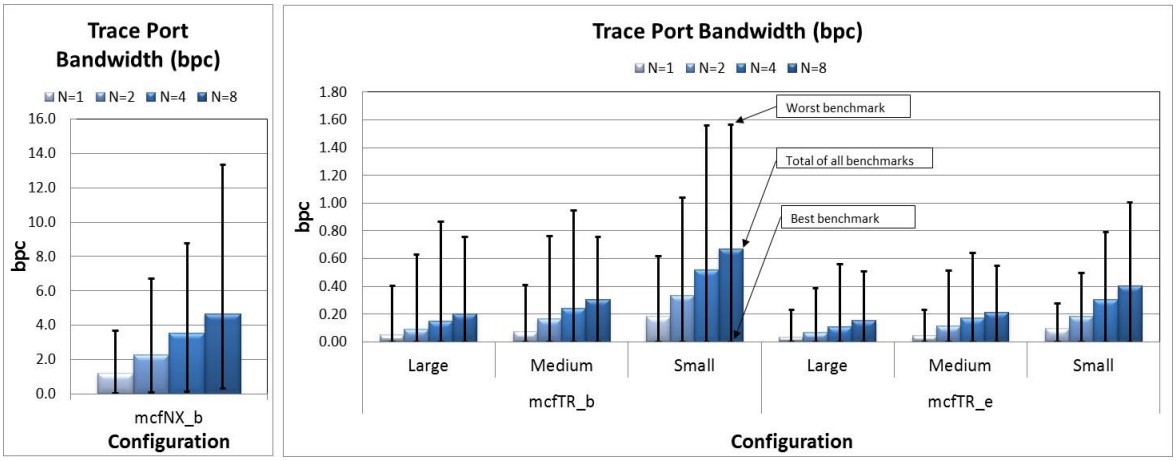


Figure 5.4 Trace port bandwidth in bpc for control-flow traces

## 5.2 Trace Port Bandwidth for Memory Load Data Value Traces

### *5.2.1 mlvNX_b*

Table 5.5 shows the trace port bandwidth in bpi and in bpc for the Nexus-like load data value traces, *mlvNX_b*, for all benchmarks as a function of the number of threads/cores (N=1, 2, 4, and 8). The last row shows the total trace port bandwidth when all benchmarks are considered together. The total bandwidth in bpi is calculated as the sum of trace sizes for all benchmarks divided by the sum of the number of instructions executed for all benchmarks. Similarly, the total bandwidth in bpc is calculated as the sum of trace sizes for all benchmarks divided by the sum of the execution times in clock cycles for all benchmarks.

For single-threaded benchmarks (N = 1), the TPB ranges between 8.80 bpi for *radix* and 31.06 bpi for *ocean_cp*. The required bandwidth varies across benchmarks and is highly correlated with the frequency and type of memory reads. Thus, *barnes*,

*ocean_cp* have relatively high TPB requirements due to the relatively high frequency of load instructions and especially those with large operand sizes (see Table 4.3 and Table 4.4), unlike *radix,* which has very low TPB requirements due to extremely small frequency of memory read instructions. The trace port bandwidth increases slightly with an increase in the number of cores for two reasons: (a) an increase in the number of bits needed to report thread index, and (b) an increase in the frequency of load instructions caused by synchronization primitives. Thus, when N = 8, the TPB ranges between 9.23 bpi for *radix* and 31.99 bpi for *ocean_cp*. The total bandwidth for the entire benchmark suite ranges between 18.25 bpi when N = 1 and 19.08 bpi when N = 8.

Whereas the bandwidth in bpi increases with the number of cores, it does not fully capture the pressure multiple processor cores place on the trace port, a shared resource. The trace port bandwidth in bpc better illustrates this pressure. Thus, load data value trace for *ocean_cp* reaches 49.11 bpc when N = 1 and 102.83 bpc when N = 8; *barnes* requires 36.23 when N = 1 and 164.38 bpc when N = 8. The total trace port bandwidth in bpc ranges from 23.76 bpc when N = 1 to 78.56 when N = 8. The trace port bandwidth in bpc is heavily influenced not only by the frequency and type of memory reads but also by the scalability of individual benchmarks. For example, *barnes*, *water_spa*, and *fmm* exhibit high scalability (see IPC in Table 4.3), which contributes to a significant increase in the trace port bandwidth requirements for N = 4 and N = 8. These results indicate that capturing load data value trace on the fly in multicores requires large trace buffers and wide trace ports. It also shows that capturing load data value trace is a much more challenging proposition than capturing control-flow trace. As shown in the next section, one alternative is to de-

velop hardware techniques that significantly reduce the volume and size of trace data that are streamed out.

Table 5.5 Trace port bandwidth for *mlvNX_b* for Splash2x benchmark

| *Benchmark* | *Trace Port Bandwidth* | | | | *Trace Port Bandwidth [bpc]* | | | |
|---|---|---|---|---|---|---|---|---|
| *Thread* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* |
| *cholesky* | 23.80 | 21.98 | 21.98 | 23.11 | 54.23 | 64.50 | 82.53 | 93.03 |
| *fft* | 16.57 | 16.78 | 17.00 | 17.21 | 21.16 | 27.39 | 33.30 | 36.14 |
| *lu_cb* | 16.34 | 16.59 | 16.84 | 17.06 | 47.88 | 81.09 | 61.81 | 83.77 |
| *lu_ncb* | 16.82 | 17.07 | 17.32 | 17.55 | 33.90 | 48.23 | 61.26 | 49.81 |
| *radix* | 8.80 | 8.92 | 9.05 | 9.23 | 2.87 | 5.24 | 10.67 | 20.90 |
| *barnes* | 22.87 | 23.22 | 23.56 | 23.91 | 36.23 | 65.23 | 112.09 | 164.38 |
| *fmm* | 11.82 | 11.98 | 12.15 | 12.33 | 26.42 | 47.51 | 88.35 | 119.74 |
| *ocean_cp* | 31.06 | 31.42 | 31.62 | 31.99 | 49.11 | 75.79 | 95.41 | 102.83 |
| *ocean_ncp* | 20.38 | 20.65 | 20.91 | 21.17 | 10.45 | 27.79 | 37.57 | 53.39 |
| *radiosity* | 13.27 | 13.58 | 13.84 | 14.21 | 23.08 | 35.66 | 47.69 | 55.96 |
| *raytrace* | 19.56 | 19.87 | 20.17 | 20.47 | 28.79 | 46.58 | 67.33 | 64.96 |
| *volrend* | 9.42 | 10.14 | 11.11 | 12.00 | 14.14 | 26.88 | 37.16 | 37.26 |
| *water_nsq* | 18.65 | 18.94 | 19.22 | 19.48 | 34.35 | 52.40 | 79.97 | 108.64 |
| *water_spa* | 19.19 | 19.49 | 19.78 | 20.08 | 38.96 | 60.35 | 98.04 | 146.35 |
| ***Total*** | **18.25** | **18.43** | **18.71** | **19.08** | **23.76** | **41.66** | **62.32** | **78.56** |

### *5.2.2 mlvCFiat*

The effectiveness of *mlvCFiat* directly depends on the first access flag miss rate – the lower it is, the fewer trace messages need to be streamed out through the trace port. Figure 5.5 shows the total first-access miss rate as a function of the number of cores for three data cache configurations (*Small*, *Medium*, *Large*). The total first-access miss rate is calculated as the total number of first-access misses when all benchmarks are considered together divided by the total number of data reads. The first-access miss rate decreases with an increase in the number of cores,

e.g., from 6.4% when N = 1 to 5.6% when N = 8 for the *Medium* configuration. As expectedly, larger data caches result in a smaller number of miss events and thus a smaller number of first-access miss events. e.g., the first-access miss rate ranges from 7.9% for the *Small* configuration to 4.2% for the *Large* configuration when the number of cores is set to four (N = 4). Figure 5.5 also indicates the minimum and the maximum first-access miss rates. Thus, the first-access miss rate reaches as high as ~30% for *ocean-ncp* and as low as 0.3% - 1% for *water-spa*, depending on the number of cores and data cache size. These results confirm that *mlvCFiat* indeed can reduce the number of trace messages.

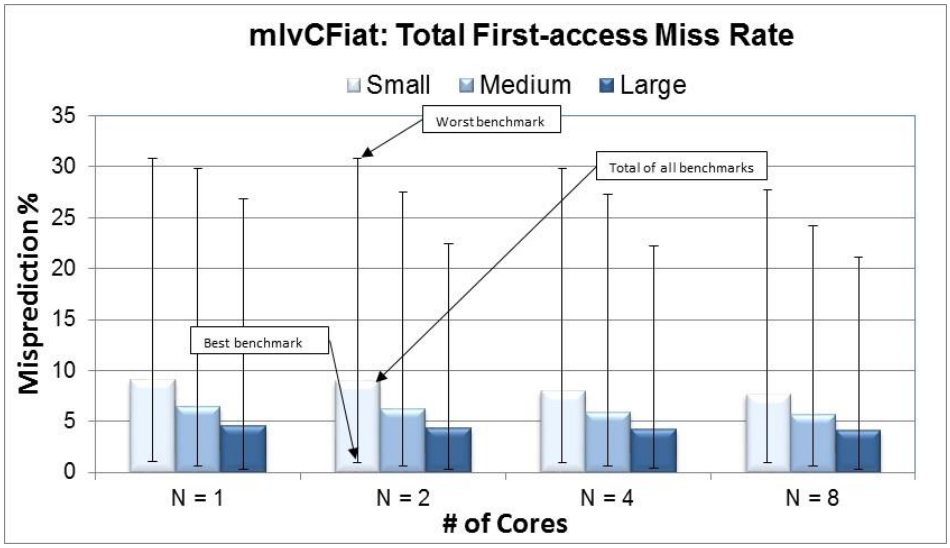


Figure 5.5 First access miss rate for Splash2x benchmark

Figure 5.6 shows the total average trace port bandwidth for Nexus–like memory read flow traces (*mlvNX_b*), and *mlvCFiat* (*mlvCF_b*, *mlvCF_e*) as a function of the number of threads (N = 1, 2, 4 and 8) and the *mlvCFiat* configuration (Small, Medi-

um and Large). Table 5.6 shows the trace port bandwidth for *Large* configuration. For N = 1, *mlvNX_b* requires on average 18.25 bpi when N = 1 and ranges between 31.06 bpi (*ocean_cp*) and 8.80 bpi (*radix*); for N = 8, *mlvNX_b* requires on average 19.08 bpi (N = 8) ranges between 31.99 (*ocean_cp*) and 9.23 (*radix*).

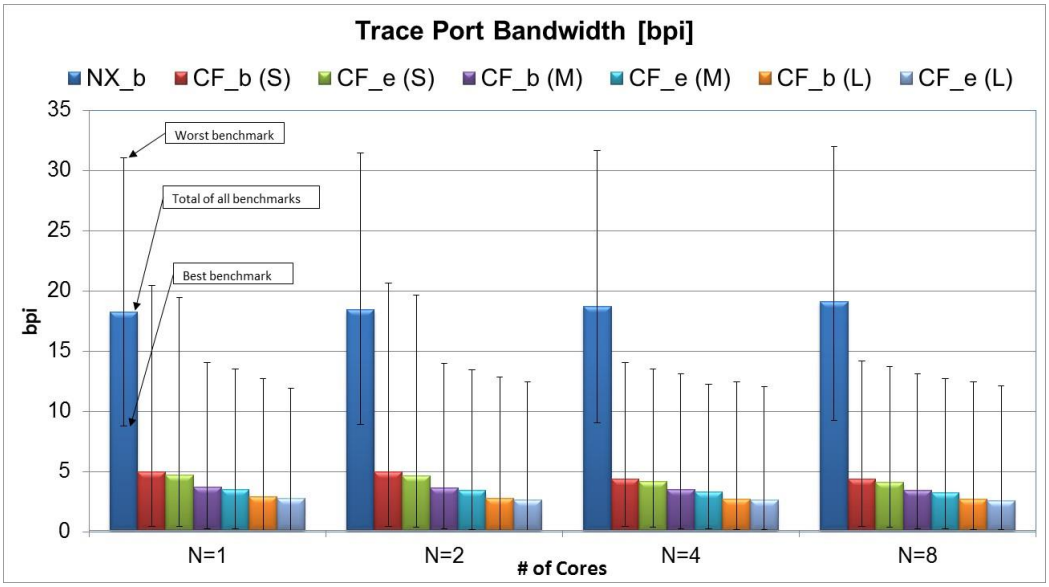


Figure 5.6 Trace port bandwidth bpi for load data value trace

*mlvCF_b* dramatically reduces the average trace port bandwidth as follows:

- Small configuration: 4.98 bpi (N = 1) and 4.34 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 3.66 times for N = 1 and 4.38 times for N = 8.
- Medium configuration: 3.70 bpi (N = 1) and 3.42 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 4.93 times for N = 1 and 5.56 times for N = 8.

- Large configuration: 2.88 bpi (N = 1) and 2.70 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 6.33 times for N = 1 and 7.04 times for N = 8.

Table 5.6 TPB for *mlvNX_b*, *mlvCF_b*, and *mlvCF_e for large configuration*

| *Thread* | *1* | | | *2* | | | *4* | | | *8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Mechanism ->* | *mlvNX_b* | *mlvCF_b* | *mlvCF_e* | *mlvNX_b* | *mlvCF_b* | *mlvCF_e* | *mlvNX_b* | *mlvCF_b* | *mlvCF_e* | *mlvNX_b* | *mlvCF_b* | *mlvCF_e* |
| *Benchmark* | | | | | | | | | | | | |
| *cholesky* | 23.802 | 4.816 | 4.651 | 21.979 | 3.700 | 3.562 | 21.979 | 3.125 | 2.995 | 23.108 | 2.568 | 2.446 |
| *fft* | 16.570 | 2.889 | 2.791 | 16.785 | 2.904 | 2.806 | 16.997 | 2.922 | 2.824 | 17.206 | 2.937 | 2.839 |
| *lu_cb* | 16.342 | 1.116 | 1.060 | 16.594 | 0.947 | 0.890 | 16.837 | 0.939 | 0.883 | 17.062 | 0.875 | 0.816 |
| *lu_ncb* | 16.816 | 6.340 | 5.838 | 17.074 | 6.410 | 5.910 | 17.322 | 6.488 | 5.989 | 17.551 | 6.557 | 6.060 |
| *radix* | 8.797 | 3.538 | 3.430 | 8.917 | 3.566 | 3.457 | 9.054 | 3.608 | 3.498 | 9.230 | 3.678 | 3.565 |
| *barnes* | 22.867 | 0.831 | 0.786 | 23.224 | 0.810 | 0.767 | 23.558 | 0.822 | 0.779 | 23.913 | 1.028 | 0.976 |
| *fmm* | 11.824 | 0.385 | 0.364 | 11.982 | 0.399 | 0.378 | 12.146 | 0.407 | 0.386 | 12.335 | 0.418 | 0.396 |
| *ocean_cp* | 31.056 | 12.317 | 11.937 | 31.420 | 12.833 | 12.413 | 31.619 | 12.459 | 12.069 | 31.986 | 12.464 | 12.074 |
| *ocean_ncp* | 20.376 | 12.702 | 11.791 | 20.654 | 11.447 | 10.691 | 20.915 | 11.460 | 10.715 | 21.173 | 11.149 | 10.440 |
| *radiosity* | 13.274 | 0.353 | 0.335 | 13.581 | 0.362 | 0.344 | 13.836 | 0.360 | 0.343 | 14.206 | 0.379 | 0.360 |
| *raytrace* | 19.562 | 0.789 | 0.742 | 19.866 | 0.821 | 0.772 | 20.174 | 0.840 | 0.791 | 20.466 | 0.835 | 0.786 |
| *volrend* | 9.421 | 0.109 | 0.098 | 10.141 | 0.130 | 0.118 | 11.110 | 0.174 | 0.157 | 12.001 | 0.204 | 0.184 |
| *water_nsq* | 18.653 | 0.308 | 0.290 | 18.940 | 0.315 | 0.297 | 19.218 | 0.298 | 0.283 | 19.480 | 0.308 | 0.291 |
| *water_sp* | 19.186 | 0.156 | 0.147 | 19.485 | 0.172 | 0.162 | 19.784 | 0.175 | 0.166 | 20.083 | 0.176 | 0.167 |
| ***Average*** | **18.252** | **2.882** | **2.738** | **18.425** | **2.771** | **2.638** | **18.710** | **2.736** | **2.605** | **19.076** | **2.708** | **2.579** |

*mlvCF_e* further reduces the average trace port bandwidth as follow:

- Small configuration: 4.69 bpi (N = 1) and 4.10 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 3.89 times for N = 1 and 4.64 times for N = 8.
- Medium configuration: 3.50 bpi (N = 1) and 3.25 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 5.21 times for N = 1 and 5.86 times for N = 8.
- Large configuration: 2.74 bpi (N = 1) and 2.58 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *mlvNX_b* 6.66 times for N = 1 and 7.39 times for N = 8.

Table 5.7 shows the compression ratio of *mlvCF_e* relative to *mlvNX_b*, as a function of the number of threads (N = 1, 2, 4 and 8) and configuration (S, M, L) calculated as TPB(*mlvNX_b*)/TPB(*mlvCF_e*). For the Small configuration average compression ratio is 3.9 (N = 1) and 4.6 (N = 8). For the Medium configuration the average compression ratio is 5.2 (N = 1) and 5.9 (N = 8). For the Large configuration the total compression ratio is 6.7 (N = 1) and 7.4 (N = 8). The best performing is *water-spatial* (N = 1) and the worst performing is *ocean_ncp* (N = 8).

Table 5.7 Compression ratio of *mlvCF_e* relative to *mlvNX_b*

| *Cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config.* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *cholesky* | 3.0 | 4.1 | 5.1 | 3.7 | 5.0 | 6.2 | 4.4 | 5.9 | 7.3 | 5.5 | 7.6 | 9.4 |
| *fft* | 4.1 | 5.3 | 5.9 | 4.2 | 5.4 | 6.0 | 4.2 | 5.4 | 6.0 | 4.2 | 5.4 | 6.1 |
| *lu-cb* | 13.4 | 14.2 | 15.4 | 13.3 | 14.7 | 18.6 | 13.4 | 14.8 | 19.1 | 13.6 | 17.5 | 20.9 |
| *lu-ncb* | 1.5 | 1.6 | 2.9 | 1.5 | 1.6 | 2.9 | 1.5 | 1.7 | 2.9 | 1.5 | 1.7 | 2.9 |
| *radix* | 2.2 | 2.3 | 2.6 | 2.3 | 2.4 | 2.6 | 2.3 | 2.4 | 2.6 | 2.3 | 2.4 | 2.6 |
| *barnes* | 4.9 | 8.6 | 29.1 | 4.9 | 8.3 | 30.3 | 5.0 | 8.4 | 30.2 | 5.0 | 8.0 | 24.5 |
| *fmm* | 11.1 | 20.9 | 32.5 | 9.4 | 20.7 | 31.7 | 10.3 | 20.5 | 31.5 | 10.2 | 20.3 | 31.1 |
| *ocean-cp* | 1.6 | 2.3 | 2.6 | 1.6 | 2.3 | 2.5 | 2.3 | 2.6 | 2.6 | 2.3 | 2.5 | 2.6 |
| *ocean-ncp* | 1.6 | 1.6 | 1.7 | 1.6 | 1.7 | 1.9 | 1.6 | 1.7 | 2.0 | 1.7 | 1.9 | 2.0 |
| *radiosity* | 14.2 | 29.4 | 39.6 | 12.8 | 28.5 | 39.5 | 12.7 | 28.5 | 40.4 | 12.8 | 29.6 | 39.5 |
| *raytrace* | 5.2 | 9.6 | 26.4 | 5.1 | 9.5 | 25.7 | 5.1 | 9.6 | 25.5 | 5.1 | 9.0 | 26.0 |
| *volrend* | 22.9 | 44.9 | 96.5 | 27.8 | 52.7 | 86.2 | 30.2 | 53.4 | 70.9 | 31.8 | 52.8 | 65.1 |
| *water-nsq* | 10.9 | 11.3 | 64.2 | 10.9 | 11.3 | 63.7 | 10.9 | 11.4 | 67.9 | 10.9 | 11.6 | 66.9 |
| *water-spa.* | 39.3 | 71.8 | 130.4 | 41.3 | 70.3 | 120.3 | 41.1 | 70.6 | 119.3 | 41.2 | 71.4 | 120.3 |
| *Total Avg.* | 3.9 | 5.2 | 6.7 | 4.0 | 5.4 | 7.0 | 4.5 | 5.6 | 7.2 | 4.6 | 5.9 | 7.4 |

Figure 5.7 shows the total trace port bandwidth in bits per clock cycle. CF_e and CF_b are highly effective in reducing the trace port bandwidth. When N = 8, the total required bandwidth for *mlvCF_e* is just 10.62 bpc compared to 78.56 for *mlvNX_b*. Our variable encoding scheme in *mlvCF_e* offers improvement in range of ~5% when compared to fixed encoding *mlvCF_b* for the Large configuration.

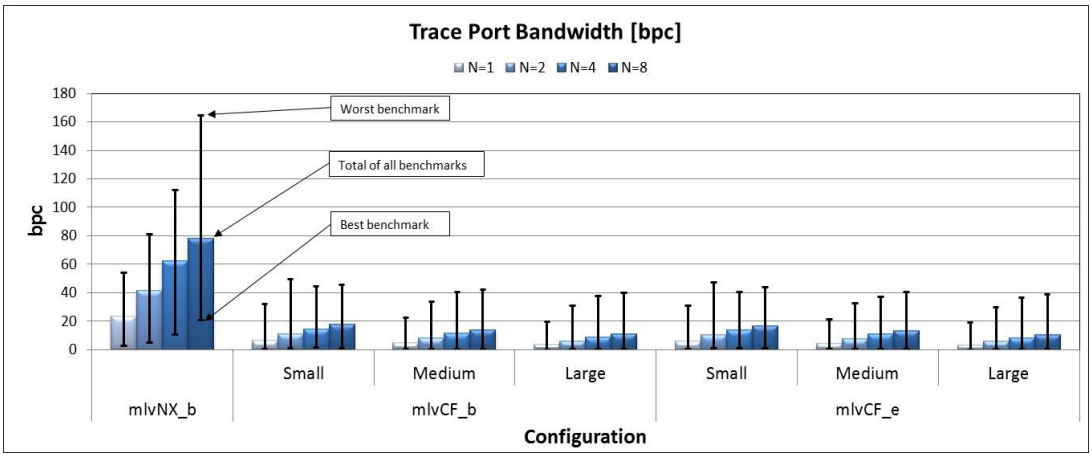


Figure 5.7 Trace port bandwidth in bpc for load data value trace

# CHAPTER 6

## EXPERIMENTAL EVALUATION FOR TIMED TRACES

Progress of mankind is progress of mind and intellect

--Rev. Pandurang Shastri Athavale

Timed program execution traces capture complete information about program execution including a correct intra-thread and inter-thread ordering of traced events. Unlike functional traces, timed traces include time stamped trace descriptors. We use timed traces to explore the challenges and opportunities of hardware tracing in multicore platforms and to evaluate the effectiveness of different techniques and their sensitivity to system parameters.

This chapter focuses on experimental flow based on timed traces. Figure 6.1 shows the experiment flow for determining trace port bandwidth requirements for timed traces in the number of bits per instruction executed (bpi) and the number of bits per clock cycle (bpc). The flow includes three major components, as follows: (a) trace generation using *TmTrace* tool suite, (b) software to hardware trace translation using custom tools that model trace compressors (*tmcfTRaptor* and *tmlvCFiat*), and (c) trace port bandwidth analysis using a custom tool suite.

The *TmTrace* tool suite that generates timed control-flow and load data value traces is described in Section 6.1. Whereas *TmTrace* tools are designed to support a range of trace applications, such as ISA profiling and trace-drive simulation, they donot include support for analyzing the hardware tracing and trace message encod-

ing at the trace port level, and software trace compression. We develop custom simulators (*tmcfTRaptor* and *tmlvCFiat*) and translators that generate timed versions of control-flow traces (*tmcfNX_b*, *tmcfTR_b* and *tmcfTR_e*) and load data value traces (*tmlvNX_b*, *tmlvCF_b*, and *tmlv_CF_e*). Section 6.2 describes the *tmcfTRaptor* simulator. Section 6.3 describes the *tmlvCFiat* simulator. The Software to hardware trace conversion described in Section 6.4 invokes similar steps to those used for functional traces. Section 6.4 describes the experimental environment. The timed traces are generated for the Splash2 benchmark suite [31], a predecessor of the Splash2x benchmark suite used to generate functional traces. These benchmarks are compiled for the IA32 ISA.

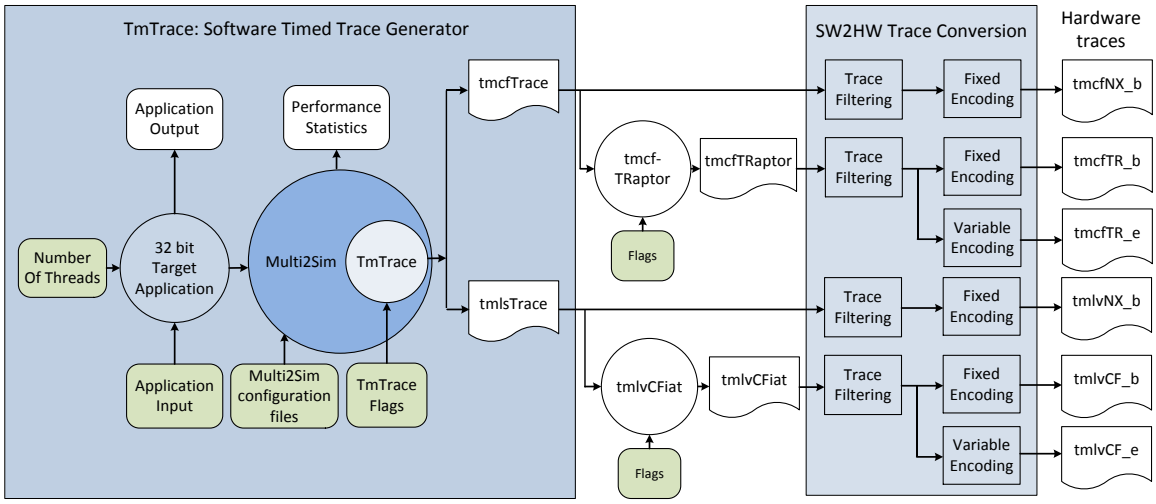


Figure 6.1 Experiment flow for timed traces

## 6.1 Software Timed Trace Generation

*TmTrace* (Timed Multithreaded Traces) is a software tool developed to capture timed program execution traces. It is designed as an extension of a heterogeneous cycle-accurate system simulator called Multi2Sim [7]. *TmTrace* can be directed to capture a timed control flow trace (*tmcfTrace*) or a timed memory read or write trace (*tmlsTrace*). Section 6.1.1 gives a functional description of *TmTrace*, Section 6.1.2 describes format of *TmTrace*-generated traces, Section 6.1.3 describes *TmTrace* implementation, and Section 6.1.4 describes *TmTrace* verification.

### 6.1.1 Functional Description

The *TmTrace* tool can generate the following types of traces: control flow, memory read, memory write, and a trace that includes all committed instructions. Figure 6.1 shows a system view of generating software timed traces the using Multi2Sim simulator. The Multi2sim simulator takes as inputs the following: (a) the number of threads, (b) a target application executable (x86 32-bit ISA), and (c) the application input parameters. In addition, it takes a system configuration that includes processor model, memory hierarchy, and the system interconnect. Multi2Sim generates general statistics related to program execution including instruction count, IPC, branch accuracy, and simulation time. The *TmTrace* component takes custom flags that control trace generation and consequently generates compressed ASCII trace files.

Table 6.1 shows flags that control the captures of software timed traces. The *mTrace* flag enables the tracing feature in the Multi2Sim simulator. Additional flags specify the type of the trace to be generated (*mcf* for control flow, *mld* for

memory reads, and *mst* for memory writes). To study a segment of a program, one can use flags to specify how many instructions should be skipped before the tracing is turned on (*mTraceSkip*) and how many instructions should be traced (*mTrace-Length*). The Intel ISA instructions are implemented as a sequence of micro instructions in the Multi2Sim simulator. The *mTraceMax* flag enables capturing micro instruction for traced instructions. The *mTraceSysPrg* flag enables capturing the instructions executed by the simulator in system calls.

Table 6.1 *TmTrace* custom flags

| Parameter | Description |
|---|---|
| --*mTrace* <file.gz> | Captures program execution trace for x86 in ASCII format. The program execution trace includes relevant information for committed instructions only. Note: if –*mTrace*SysPrg is used, both system and user program traces are captured; otherwise only user program traces are captured. |
| --mcf | Captures control flow program execution trace. Note: requires –*mTrace* flag. |
| --mld | Captures memory reads. Note: requires –*mTrace* flag. |
| --mst | Captures memory writes. Note: requires–*mTrace* flag. |
| --*mTrace*Skip <#Skipped INS> | Specifies the number of instructions skipped before the tracing is turned ON. Note: requires -*mTrace* and -*mTrace*Length flags. |
| --*mTrace*Length <Length in #INS> | Specifies the number of instructions to be traced. Note: requires -*mTrace* and -*mTrace*Skip flags. |
| --*mTrace*SysPrg | Enables tracing of system code. Note: requires -*mTrace* flag. |
| --*mTrace*Max | Captures assembly instructions and micro instructions for control flow instructions and load values for memory read operations. |

### 6.1.2 Format of Timed Trace Descriptors

Figure 6.2 shows the format of a trace descriptor generated when a trace with all committed instructions is generated. The trace descriptor includes the following fields:

- CC: clock cycle in which the instruction is committed;
- Ti: thread index;

- PC: instruction address; and
- ASM and UASM: assembly language instruction and micro instruction.

An example descriptor shown in Figure 6.2 is interpreted as follows: at clock cycle 135 from the start of the program simulation, a thread with index 0 commits an instruction at the address 0x8048d0a; the assembly instruction is xor ebp, ebp.

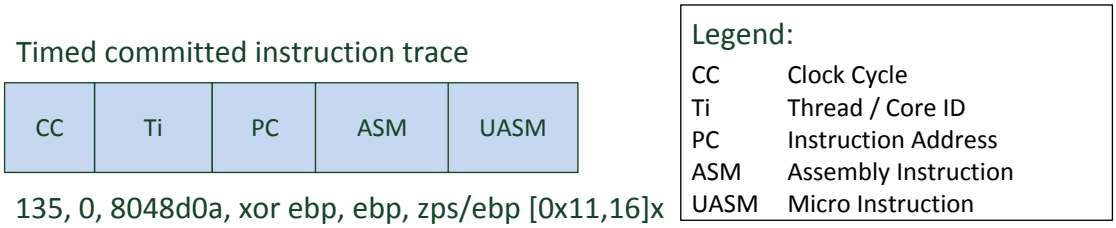


Figure 6.2 Trace descriptor when all committed instructions are traced

Figure 6.3 shows a timed control flow trace descriptor captured by *TmTrace* when the *mTrace* and *mcf* flags are set. The descriptor is generated for each control-flow instruction and includes the following fields:

- CC: clock cycle in which the instruction is committed;
- Ti: thread index;
- PC: branch instruction address;
- TA: target address of the branch;
- InstSize: size of the instruction;
- D/I: type of the branch, direct or indirect;
- C/U: type of the branch, conditional or unconditional;
- T/N: branch outcome, taken or not taken;

- BrType: type encoded as follows: call – 1, ret – 2, syscall – 3, ibranch – 4, jump – 5, branch – 6;
- BBL: the number of instructions in a basic block ending with the branch.

An example descriptor shown in Figure 6.3 is interpreted as follows: at clock cycle 463 in thread 0 a control-flow instruction at the address of 0x804814c is committed; its target address is 0x8058090 and the instruction size is 5 bytes. The instruction is a direct unconditional taken branch of type "call". The number of instructions executed in the program since the previous descriptor is 13.

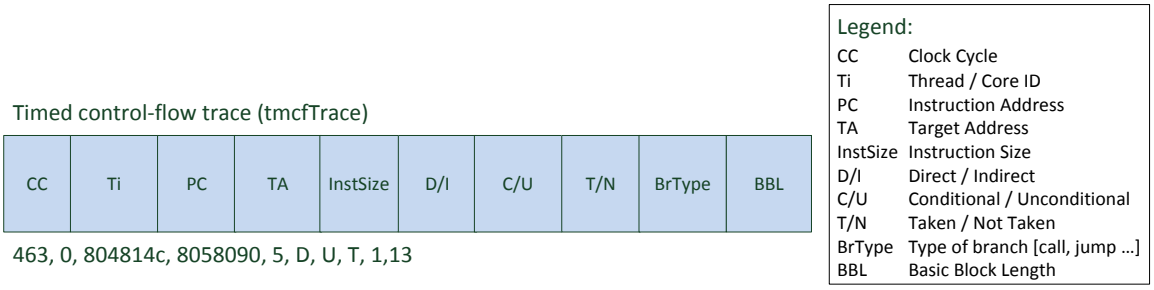


Figure 6.3 Trace descriptor for timed control-flow trace

Figure 6.4 shows several types of trace descriptors generated for memory read (with *–mld* switch) and memory write operations (*–mst* switch). Figure 6.4 (a) shows a trace descriptor generated for both memory reads and writes. It includes the following fields:

- CC: time in clock cycles in which the memory instruction is committed;
- Ti: thread index;
- L/S: read (L=0) or write (S=1) operation;
- PC: instruction address;

- Oaddr: operand address;
- Osize: operand size.

An example descriptor shown in Figure 6.4 (a) is interpreted as follows: at clock cycle time 456 in thread 0 a load instruction residing at the address 0x8048132 reads a 4-byte operand from the address 0xbfff0000. The following descriptor describes a store operation committed at clock cycle 457 in thread 0; the address of the instruction is 0x8048138 and it writes a 4-byte operand at the address 0xbffefffc in memory. Figure 6.4(b) shows the trace descriptor format when only memory reads or memory writes are generated. Figure 6.4(c) shows the trace descriptor format when memory reads with the operand values are generated.

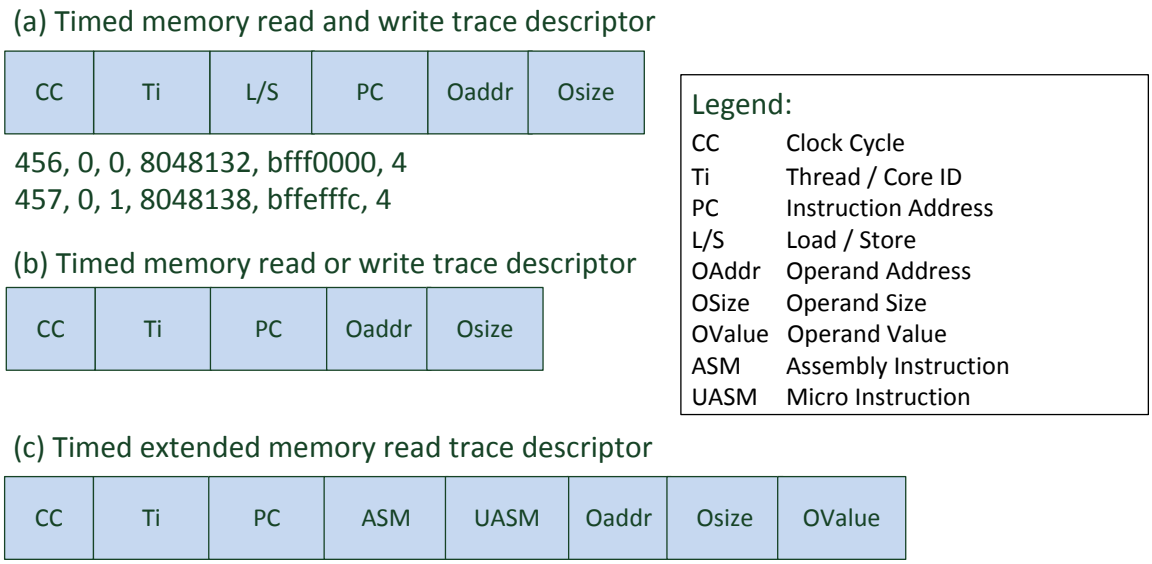


Figure 6.4 Trace descriptors generated for memory reads and writes

### 6.1.3 *TmTrace* Implementation Details

*TmTrace* is implemented as a component in Multi2Sim, a cycle accurate computer system simulator. Whereas Multi2Sim supports several instruction set architectures including Intel x86 (IA32), MIPS-32, ARM, AMD GPU, and NVIDIA GPU, *TmTrace* is implemented for Intel's IA32 ISA. The simulator offers a range of input parameters that can be configured to model processor, memory hierarchy, and interconnection network. More details about Mult2Sim can be found at its web site [32]. Mult2Sim's modular software design with well-defined interfaces allow researchers to add new functionality.

Figure 6.5 illustrates capturing timed control-flow traces enabled by the *mTrace* and *mcf* flags. The timing simulator models all pipeline stages from the fetch to the commit stage. *TmTrace* augments the fetch stage to capture relevant information about control-flow instructions needed to create a trace descriptor. However, we do not emit a trace descriptor in the fetch stage. Some control-flow instructions may be discarded in the pipeline. If the control-flow instruction commits, then a corresponding trace descriptor is generated and written into a file, including the current clock cycle time.

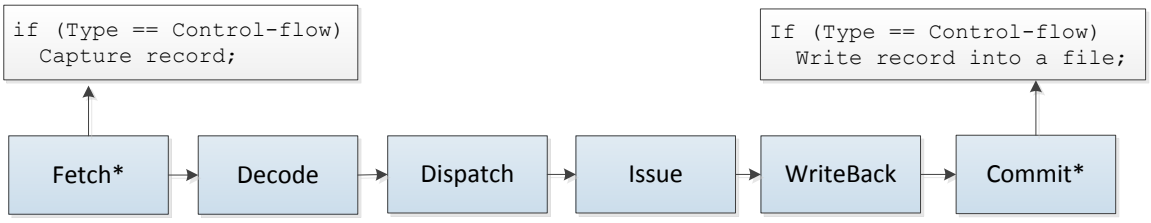


Figure 6.5 Capturing timed control flow traces

Figure 6.6 illustrates capturing timed memory read and/or write traces enabled by the *mTrace*, *mld*, and/or *mst* flags. If an instruction is identified as a memory referencing instruction, in the fetch stage we capture relevant information for a trace descriptor and store it into a data structure (address, operand address, size, load value). If a memory referencing instruction is committed, its corresponding trace descriptor will be written into a trace file.

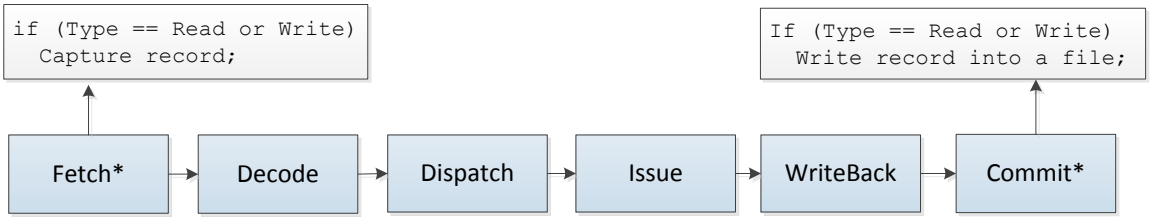


Figure 6.6 Capturing timed memory read and write traces

### 6.1.4 Verification Details

To verify that *TmTrace* captures complete control-flow or memory read and write traces, we develop a number of assembly language test programs. These test programs are designed to include a wide variety of characteristic instructions that result in trace events. For each test program, an expected trace is derived and then compared to the one generated by *TmTrace*.

Figure 6.7 (a) shows a selected portion of the *testControlEnumeration.s* test program. The selection includes a number of conditional direct branches and all of them are not taken. Figure 6.7(b) shows trace descriptors for the selection from Fig-

ure 6.7(a). All trace descriptors match the expected ones and thus we can conclude that we captured a correct *tmcfTrace*.

Figure 6.8 (a) shows a program section with unconditional and indirect branch instructions. Both *jmp* and *call* instructions are unconditional direct branch instructions, while the return is an indirect branch. The captured descriptors shown in Figure 6.8 (b) correspond to the expected ones.

(a) Code Sample

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#section 1
#unsigned conditional direct branches
#all branches will not be taken
#branch if strictly above
#Taken when CF and ZF are both zero
mov %eax, 1
mov %eax, 2
jnbe exit1

#branch if strictly below
#Taken when CF is 1
mov %eax, 2
mov %eax, 1
jb exit1

#branch if  not equal/ not zero
#Taken when ZF is 0
mov %eax, 1
cmp %eax, 1
jnz exit1

#branch if not parity/parity odd
#Taken when PF is 0
jnp exit1

#branch if CX is zero
mov %eax, 1
jecxz exit1
#End of unsigned conditional direct branches
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

(b) t*mcfTrace* (timed control-flow)

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
. . .
637711, 0, 80483e1, 804844c, 2, D, C, N, 5, 6
637712, 0, 80483ed, 804844c, 2, D, C, N, 3, 6
637713, 0, 80483f7, 804844c, 2, D, C, N, 3, 6
637713, 0, 80483f9, 804844c, 2, D, C, N, 1, 6
637713, 0, 8048400, 804844c, 2, D, C, N, 2, 6
. . .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.7 Conditional branches in *testControlEnumeration.s*

(a) Code Sample

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        #Unconditional Direct jump
        jmp Label1
        #Not executed
        test %eax, %eax
Label1:
        #send the string to console
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        #unconditional direct branch - call
        call    puts
        mov     eax, 1
exit1:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

(b) t*mcfTrace* (timed control flow)

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
. . .
637718, 0, 8048437, 804843b, 7, D, U, T, 5, 2
637768, 0, 8048442, 80482f0, 5, D, U, T, 1, 2
637893, 0, 80482f0, 80482f6, 7, I, U, T, 5, 1
637894, 0, 80482fb, 80482e0, 7, D, U, T, 5, 2
637900, 0, 80482e6, 144f0, 7, I, U, T, 5, 2
663378, 0, 804844d, b7dfca83, 1, I, U, T, 2, 3
. . .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.8 Unconditional branches in *testControlEnumeration.s*

Figure 6.9 demonstrates how to run the Mult2Sim with tracing functions enabled. It shows a command line that simulates a benchmark execution (./one). The timed control-flow trace is captured for a selected code segment of a test program (./one) that starts after the first 85,717 instructions are committed and includes 20 instructions.

```
    tmcfTrace (timed control flow)
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --x86-config x86-config --mTrace six-
teen.gz --mcf --mTraceSysPrg --mTraceSkip 85717 --mTraceLength 20 ./one
    amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ more sixteen
    mTrace: x86.init version="1.671" num_cores=8 num_threads=1
    mTrace: Only Committed instructions for User Code
    tMcf: MultiThread control flow traces
    SIZE: 8, 31
    tMcf: clock cycle,CoreID, PC, TA, instSize, Direct-D/ Indirect-I , Cond-
C/UnCond-U, Outcome [T / NT], brcategory, BBL
    tMcf: brcategory [call 1 / ret 2 / syscall 3 / ibranch 4 / jump 5 / branch
6]
    657710, 0, b7e539eb, b7f0a5eb, 5, D, U, T, 1, 3
    657711, 0, b7f0a5ee, b7e539f0, 1, I, U, T, 2, 2
    657713, 0, b7e539ff, b7e53b58, 2, D, C, N, 6, 5
    657714, 0, b7e53a08, b7e53a58, 2, D, C, N, 6, 2
    657715, 0, b7e53a0f, b7e53b80, 2, D, C, N, 6, 3
    657716, 0, b7e53a1b, b7e53b28, 2, D, C, N, 6, 3
    657717, 0, b7e53a24, b7e53ae8, 2, D, C, N, 6, 2
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.9 Tracing enabled for a specific code segment

Software generation of timed read and write traces is tested with a program that executes memory read operations with varying operand sizes. We have verified memory reads of 8-, 16-, 32- and 64-bit signed and unsigned integers, floats, doubles, extended precision, and SIMD (single instruction multiple data). In this section, we look at the load data value output for 8 bit signed / unsigned integers, extended precision and SIMD type of data types.

Figure 6.10 (a) shows a program that creates an unsigned and a signed 8 bit array. The code in lines 8-10 prints the addresses of the arrays in memory, and the code in 11-14 touches the elements of the arrays. Figure 6.10 (b) shows the addresses of both arrays. Figure 6.10 (c) shows the trace snippet for the lines 12, 14. It shows that the array addresses and values are correct (Figure 6.10(b) & (c))

```
    (a) Code sample
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    1. int i;
    2. volatile uint8_t uint8[17];
    3. volatile int8_t sint8[17];
    4. for(i=0;i<17;i++){
    5.   uint8[i] = i;
    6.   sint8[i] = -i;
    7. }
    8. printf("uint8 address: %p\n",uint8);
    9. printf("sint8 address: %p\n",sint8);
    10.printf("Begin uint8/int8 test\n");
    11.for(i=0;i<17;i++)
    12.   uint8[i];
    13.for(i=0;i<17;i++)
    14.   sint8[i]
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
    (b) Run time
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --x86-config x86-config --mTrace
mlvTest_ld_trace.gz --mld  ./mlvTest_akt
    ; Multi2Sim 4.2 - A changed Simulation Framework for CPU-GPU Heterogeneous
Computing
    . . .
    uint8 address: 0xbffeff3a
    sint8 address: 0xbffeff4b
    Begin uint8/int8 test
    . . .
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
    (c) tmlsTrace (timed Memory read trace)
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    . . .
    740363, 0, 8048604,------------,load edx/ea [0xbffeff3a,1],bffeff3a,1,0
    740364, 0, 8048604,------------,load edx/ea [0xbffeff3b,1],bffeff3b,1,1
    740365, 0, 8048604,------------,load edx/ea [0xbffeff3c,1],bffeff3c,1,2
    740366, 0, 8048604,------------,load edx/ea [0xbffeff3d,1],bffeff3d,1,3
    . . .
    740425, 0, 8048616,------------,load edx/ea [0xbffeff4b,1],bffeff4b,1,0
    740426, 0, 8048616,------------,load edx/ea [0xbffeff4c,1],bffeff4c,1,ff
    740427, 0, 8048616,------------,load edx/ea [0xbffeff4d,1],bffeff4d,1,fe
    740432, 0, 8048616,------------,load edx/ea [0xbffeff4e,1],bffeff4e,1,fd
    . . .
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.10 Testing *TmTrace* load data value traces: an example

Figure 6.11(a) shows a program accessing an extended precision array. Lines 2 initialize the extended precision array, and lines 3-4 print the address and value respectively. Figure 6.11(b) shows the address of the array at runtime. Figure 6.11

(b) & (c) shows that the address `0xbffeff60` and value `1.2` (line 2) match the address and value `3FFF999999999999980` (in hex format) printed in the program.

```
   (a) Code sample
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   1. volatile long double ex[1];
   2. ex[0] = 1.2;
   3. printf("double address: %p\n",ex);
   4. ex[0];
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
   (b) Run time
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --x86-config x86-config --mTrace
mlvTest_akt_trace.gz --mld --x86-debug-isa isa.txt  ./mlvTest_akt_extended

   ; Multi2Sim 4.2 - A changed Simulation Framework for CPU-GPU Heterogeneous
Computing
   . . .
   double address: 0xbffeff60
   . . .
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
   (c) tmlsTrace (timed Memory read trace)
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   . . .
   11646, 0, 8048e7f, ------------,load data/ea [0xbffeff60,10],bffeff60,10,
3FFF999999999999980
   . . .
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.11 Testing *TmTrace* for an extended data type

Figure 6.12(a) shows a test program which uses an SIMD (single instruction, multiple data) vector instruction. Lines 1-2 show __m128i directives for initializing two 128-bit variables. Lines 3 – 7 show assembly instructions that calculate an xor function for two input variables. Line 8 prints the result. Figure 6.12(b) shows the result in hex string while running the program. Figure 6.12 (c) shows a trace segment for the SIMD variables. The first four trace messages show loading of the values in to the registers. The last trace descriptor at clock cycle 82560 shows the correct hex result matching the running program.

```
    (a) Code sample
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1.    __m128i a = _mm_setr_epi32(0x00ffff00, 0x00ffff00, 0x00ffff00,
      0x10ffff00);
2.    __m128i b = _mm_setr_epi32(0x0000ffff, 0x0000ffff, 0x0000ffff,
      0x0000ffff), x;
3.    asm(
4.    "pxor %2, %0;"
5.    :"=x"(a)
6.    :"x"(a), "x"(b)
7.    );
8.    print128(a);
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
    (b) Run time
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --x86-config x86-config --mTrace
mlvTest_akt_trace.gz --mld ./MMX

    ; Multi2Sim 4.2 - A changed Simulation Framework for CPU-GPU Heterogeneous
Computing
    . . .
    10ff00ff00ff00ff 00ff00ff00ff00ff
    . . .
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
    (c) tmlsTrace (timed memory read trace)
    <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    . . .
    82266, 0, 8048f29,------------,load xmm_data/ea [0xbffeff50,16],bffeff50,16,
10FFFF0000FFFF0000FFFF0000FFFF00
    82554, 0, 8048fca,------------,load xmm_data/ea [0xbffeff60,16],bffeff60,16,
0000FFFF0000FFFF0000FFFF0000FFFF
    82556, 0, 8048fd9,------------,load xmm_data/ea [0xbffeff30,16],bffeff30,16,
10FFFF0000FFFF0000FFFF0000FFFF00
    82557, 0, 8048fdf,------------,load xmm_data/ea [0xbffeff40,16],bffeff40,16,
0000FFFF0000FFFF0000FFFF0000FFFF
    82560, 0, 8048fef,------------,load xmm_data/ea [0xbffeff30,16],bffeff30,16,
10FF00FF00FF00FF00FF00FF00FF00FF
    . . .
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.12 Testing *TmTrace* for SIMD data types

## 6.2 *tmcfTRaptor* Simulator

The *tmcfTRaptor* simulator takes a timed control-flow trace as an input, models multicore trace predictor structures, and generates a timed *tmcfTRaptor* compressed control-flow trace. The *tmcfTRaptor* maintains *TRaptor* predictor struc-

tures for each thread separately and carries out steps that correspond to hardware trace compression in the trace module. Mispredicted trace descriptors are written into an output trace file. Section 6.2.1 gives functional description for the *tmcfTRaptor*, Section 6.2.2 describes its implementation, and Section 6.2.3 describes the verification of the *tmcfTRaptor* simulator.

### 6.2.1 Functional Description

Table 6.2 shows flags for controlling behavior of *tmcfTRaptor*. These flags are used for specifying: (i) output file size, (ii) *TRaptor* branch predictor structures, and (iii) output file name. *tmcfTRaptor* contains a *gshare* branch outcome predictor, a return address stack (RAS), and an indirect branch target buffer (iBTB). We can specify the number of entries for the *gshare* outcome predictor (0, 256, 512, 1024, 2048, and 4096), the RAS (0, 8, 16, and 32), and iBTB (0, 16, 32, and 64).

Figure 6.13 shows the format of trace descriptors generated by *tmcfTRaptor*. There are three types of distinct trace descriptors for (i) outcome mispredictions, (ii) target mispredictions, and (iii) exceptions.

Table 6.2 *tmcfTRaptor* flags

| Parameter | Description |
|---|---|
| --help | Generates help messages |
| --f [size] | Output file size in MB. If file exceeds the specified size, tracing stops. Default 50000 MB |
| --gshare [entries] | gshare – outcome predictor with entries = {0, 256, 512, 1024, 2048, 4096}. Note: entries = 0 means no gshare. Default is 256. |
| --ras [entries] | RAS – return address stack with entries = {0, 8, 16, 32}. Note: entries = 0 means no RAS. Default is 8. |
| --ibtb [entries] | iBTB – 2 way set associativity indirect branch target buffer with {0, 16, 32, 64} entries. Note: entries = 0 means no iBTB. Default is 64. |
| --o [filename] | Specifies output trace file filename. Default – tMcfTRaptor_out_yr_mon_day_hr_min_sec. Note: *.txt = descriptors, *.Statistics = Statistics |

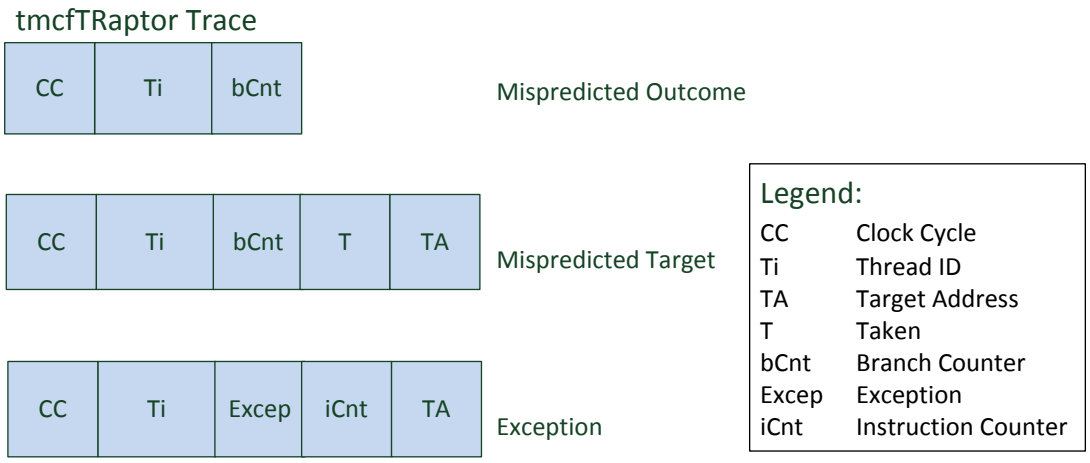


Figure 6.13 *tmcfTRaptor* trace descriptor formats

Figure 6.14(a) shows a sample output trace file generated by *tmcfTRaptor*. Figure 6.14 (b) shows a statistics report generated when running the *barnes* benchmark. The statistics file includes information about the type and frequency of control flow instructions, the predictor structure statistics, and binary sizes for individual descriptor fields that can be used in evaluating the effectiveness of the *tmcfTRaptor*.

```
 (a) tmcfTRaptor trace
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
akt0001@eb245-drina:/mnt/drive02/ttraces/benRun_Detail_v1/barnes/1$ head
LargeTRaptorSerialBarnes.txt
1413, 0, 1
1426, 0, 10
2371, 0, 4, T, 0x08073380
2600, 0, 1
2831, 0, 2, T, 0x080731d0
2847, 0, 1
2861, 0, 2, T, 0x08073368
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
(b) tmcfTRaptor Statistics
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
akt0001@eb245-drina:/mnt/drive02/ttraces/benRun_Detail_v1/barnes/1$ more
LargeTRaptorSerialBarnes.Statistics
; tmcfTRaptor: Instrumentation Time 617331 ms
; timed Control Flow Stats
Recorded 218360061 control transfer instructions.
        70954154 ( %32.49 ) Conditional Direct Taken
        59093985 ( %27.06 ) Conditional Direct Not Taken
        51871988 ( %23.76 ) Unconditional Direct
        36439934 ( %16.69 ) Unconditional Indirect
; timed Control Flow TRaptor Stats
tmcfTRaptor: Recorded 218360061 direct conditional branches, indirect uncon-
ditional branches, and exceptions
181919908 conditional direct branches
        172325660 ( %94.73 ) outcomes predicted
        9594248 ( %5.27 ) outcomes mispredicted
36440153 unconditional indirect branches
        36439991 ( %100.00 ) targets predicted
        162 ( %0.00 ) targets mispredicted
0 exceptions
; TRaptor structure stats
        correct ,Total(%hitRate) structure type
        172325660       ,181919908 ( %94.73 ) gshare
        36302064        ,36302067 ( %100.00 ) ras
        137927  ,138086 ( %99.88 ) ibtb
; branch stats
        Total #
        call instruction:       36302072
        ret instruction:        36302067
        syscall instruction:    0
        ibranch instruction:    219
        jump instruction:       15707783
        branch instruction:     130047920
; File size in Binaries
; Type, TotalSizeofTime, TotalSizeofLine, TotalSize
Input, 1746880488, 6769161891, 8516042379
OutputDirect, 76753984, 47971240, 124725224
OutputIndirect, 1296, 1620, 2916
OutputException, 0, 0, 0
Output, 76755280, 47972860, 124728140
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.14 *tmcfTRaptor* output files

### 6.2.2 Implementation Details

Figure 6.15 shows the organization of the *tmcfTRaptor* simulator. It takes two inputs: (i) a timed control flow trace generated by the *tmcfTrace* tool, and (ii) *tmcfTRaptor* configuration parameters (Table 6.2). It gives two output files: (i) trace descriptors for mispredicted events in the *mcfTRaptor* structures, and (ii) statistics of input and output traces. The simulator reads the input trace descriptors, analyzes them (*doAnalysisOnBranch()*), and writes emitted trace descriptors and statistics into the output files. The *doAnalysisOnBranch* procedure checks the type of branch and invokes corresponding procedures as follows: *Private_DirectConditional* for direct branch and call instructions, *Private_Ret* for return instructions, *Private_IndirectCall* for indirect call instructions, and *Private_OtherUnconditionalIndirect* for all other unconditional indirect branch instructions.

*Private_DirectConditional* checks whether its input is a conditional branch or a call. For conditional branches it performs a lookup in the *gshare* branch outcome predictor structure. A *gshare* index is generated and the predicted outcome is retrieved and compared to the actual outcome. If the prediction is incorrect, an ASCII descriptor associated with instruction is created and stored in a dequeue ADT (abstract data type) structure. The *gshare* is then updated accordingly. If the branch is a call instruction, the return address is pushed on to the RAS structure.

*Private_Ret* accesses the RAS structure and retrieves the target address from the top. If the predicted address does not match the actual one, an ASCII descriptor associated with the instruction is created and stored in the dequeue ADT structure.

*Private_IndirectCall* and *Private_OtherUnconditionalIndirect* access the iBTB structure in *mcfTRaptor*. An iBTB index is generated and a lookup is performed to retrieve a predicted target address. If the prediction is incorrect, an ASCII descriptor associated with the instruction is created and stored in the dequeue ADT structure. The iBTB structure is updated. If the indirect branch is a call instruction, its target address is pushed onto the RAS structure.

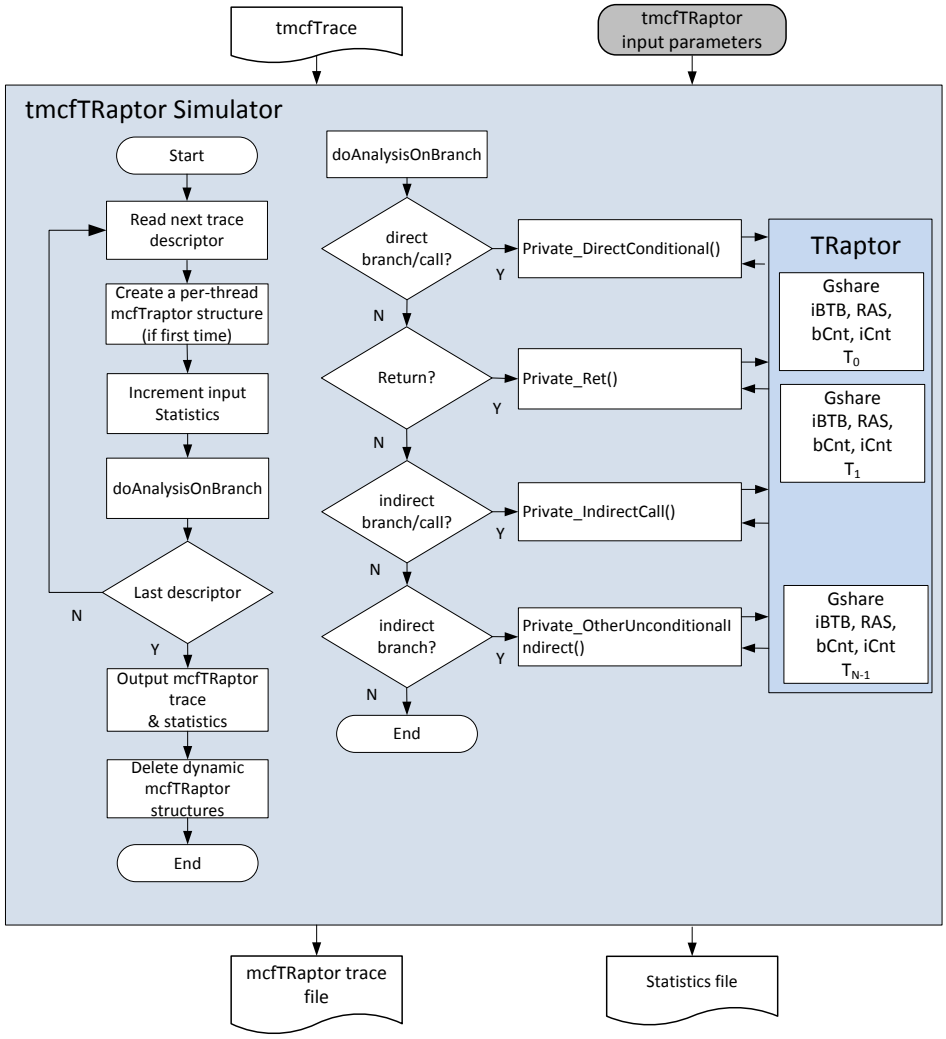


Figure 6.15 *tmcfTRaptor* simulator organization

### 6.2.3 Verification Details

*tmcfTRaptor* has been tested using custom test programs tailored to verify the behavior of the predictor structures, namely the gshare outcome predictor, the return address stack, and the indirect branch target buffer. This section describes test programs for each structure. Figure 6.16(a) shows a program used to verify the gshare structure. The program includes a *for* loop which terminates after iterating 20 times (0 to 19). The last instruction of the *for* loop is a direct conditional branch, which is taken for 20 times and not taken the last time. This loop branch will be incorrectly predicted 5 times until the branch history register (BHR) and the gshare entries are warmed up (the gshare entries are initialized in the weak not taken state). Figure 6.16 (b) illustrates the trace events for the loop branch. We track the state of the bCnt counter and relevant entries in the gshare predictor through warming up (lines 1-30). The first six instances of the loop branch result in trace messages as shown. Once the predictor structures are warmed up, all lookups occur in the gshare entry with index 187. The last trace message is emitted on the loop exit: a trace message includes bCnt = 15 (14 correctly predicted outcomes with the last one incorrectly predicted because the predictor is trained to be in the strong taken state and the actual branch is not taken).

```
(a) Code Sample
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
volatile int x;
for( x = 0; x < 20; x++ );
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
(b) tmcfTRaptor: Gshare entries
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1. bCnt: 1
2. prediction is correct 0
3. Actual Result: T
4. Next GSHARE[35]: 2 (WT)
5. Mis-predicted outcome descriptor 0, 1

6. bCnt: 1
7. prediction is correct 0
8. Actual Result: T
9. Next GSHARE[139]: 2 (WT)
10. Mis-predicted outcome descriptor 0, 1

11. bCnt: 1
12. prediction is correct 0
13. Actual Result: T
14. Next GSHARE[219]: 2 (WT)
15. Mis-predicted outcome descriptor 0, 1

16. bCnt: 1
17. prediction is correct 0
18. Actual Result: T
19. Next GSHARE[123]: 2 (WT)
20. Mis-predicted outcome descriptor 0, 1

21. bCnt: 1
22. prediction is correct 0
23. Actual Result: T
24. Next GSHARE[59]: 2 (WT)
25. Mis-predicted outcome descriptor 0, 1

26. bCnt: 1
27. prediction is correct 0
28. Actual Result: T
29. Next GSHARE[187]: 2 (WT)
30. Mis-predicted outcome descriptor 0, 1

31. bCnt: 1
32. prediction is correct 1
33. Actual Result: T
34. Next GSHARE[187]: 3 (ST)
. . .
86. bCnt: 15
87. prediction is correct 0
88. Actual Result: N
89. Next GSHARE[187]: 2 (WT)
90. Mis-predicted outcome descriptor 0, 15
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.16 GShare verification example and results

Figure 6.17 (a) shows a test program for verifying the return address stack. The test include multiple nested calls, `main` calls the functions `f0` and `f4`, whereas `f0` calls `f1`, `f1` calls `f2`, `f2` calls `f3`, `f3` calls `f4`, and `f4` does not call other functions. Figure 6.17 (b) shows the results with the RAS updates and the status after each step. All returns are correctly predicted except the exit from `main`.

```
(a) Code Sample
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void f4(int x){ x; }
void f3(int x){ f4(x); }
void f2(int x){ f3(x); }
void f1(int x){ f2(x); }
void f0(int x){ f1(x); }
int main() {
   volatile int x;
   f0(x);
   f4(x);
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
(b) tmcfTRaptor: RAS updates and hit
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Instruction call f0(x)
RAS[2] = 804848f
Instruction call f1(x)
RAS[3] = 804846c
Instruction call f2(x)
RAS[4] = 8048459
Instruction call f3(x)
RAS[5] = 8048446
Instruction call f4(x)
RAS[6] = 8048433
Instruction ret from f4(x)
isRASHit = 1
RAS[5] = 8048446
Instruction ret from f3(x)
isRASHit = 1
RAS[4] = 8048459
Instruction ret from f2(x)
isRASHit = 1
RAS[3] = 804846c
. . .
Instruction ret from main
isRASHit = 0
RAS[0] = b7669470
Emitted descriptor 0, 7, T, 0xb7dfca83
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.17 Return address stack example

Figure 6.18 shows a program for testing the indirect branch target buffer. It includes a loop that calls a function through a pointer 20 times. Table 6.3 has iBTB structure parameters PC, TA, index mask, path information register (PIR), iBTB index, prediction hit or miss, and the number of iterations. These results can be independently verified to match the expected events. Once the iBTB is fully warmed up, the iBTB provides correct target address for the last 13 iterations of the loop.

```
1.   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2.   int loop_inc (int loop_count) {
3.        return ++loop_count;
4.   }
5.   int main(void) {
6.        int loop;
7.        int (*pf)(int);
8.        pf = loop_inc;
9.        for (loop = 0; loop < 20; loop++)
10.            pf (1);
11.       return 1;
12.  }
13.  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
14.  Assembly code
15.  <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
16.  08048e50 <main>:
17.   8048e50:   55                        push   ebp
18.   8048e51:   89 e5                     mov    ebp,esp
19.   8048e53:   83 e4 f0                  and    esp,0xf0
20.   8048e56:   83 ec 20                  sub    esp,0x20
21.   8048e59:   c7 44 24 1c 44 8e 04      mov    DWORD PTR
     [esp+0x1c],0x8048e44
22.   8048e60:   08
23.   8048e61:   c7 44 24 18 00 00 00      mov    DWORD PTR [esp+0x18],0x0
24.   8048e68:   00
25.   8048e69:   eb 12                     jmp    8048e7d
26.   8048e6b:   c7 04 24 01 00 00 00      mov    DWORD PTR [esp],0x1
27.   8048e72:   8b 44 24 1c               mov    eax,DWORD PTR [esp+0x1c]
28.   8048e76:   ff d0                     call   eax
29.   8048e78:   83 44 24 18 01            add    DWORD PTR [esp+0x18],0x1
30.   8048e7d:   83 7c 24 18 13            cmp    DWORD PTR [esp+0x18],0x13
31.   8048e82:   7e e7                     jle    8048e6b
32.   8048e84:   b8 01 00 00 00            mov    eax,0x1
33.   8048e89:   c9                        leave
34.   8048e8a:   c3                        ret
35.   8048e8b:   66 90                     xchg   ax,ax
36.   8048e8d:   66 90                     xchg   ax,ax
37.   8048e8f:   90                        nop
38.  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.18 iBTB test example

Table 6.3 iBTB status and updates for the test example

| PC | TA | index Mask | PIR | index | Miss/ Hit | New PIR | Lastway Used | Prediction correct | iteration |
|---|---|---|---|---|---|---|---|---|---|
| 8048e76 | 8048e44 | 1f | 13c9 | 20 | Miss | 4fc3 | 0 | 0 | |
| 8048e76 | 8048e44 | 1f | 4fc3 | 8 | Miss | 3feb | 1 | 0 | |
| 8048e76 | 8048e44 | 1f | 3feb | 24 | Miss | ff4b | 0 | 0 | 3 |
| 8048e76 | 8048e44 | 1f | ff4b | 24 | Hit | fdcb | 0 | 1 | 1 |
| 8048e76 | 8048e44 | 1f | fdcb | 26 | Miss | f7cb | 1 | 0 | |
| 8048e76 | 8048e44 | 1f | f7cb | 16 | Miss | dfcb | 0 | 0 | 2 |
| 8048e76 | 8048e44 | 1f | dfcb | 24 | Hit | 7fcb | 0 | 1 | 1 |
| 8048e76 | 8048e44 | 1f | 7fcb | 24 | Hit | ffcb | 0 | 1 | 1 |
| 8048e76 | 8048e44 | 1f | ffcb | 24 | Hit | ffcb | 0 | 1 | 12 |

## 6.3 *tmlvCFiat* Simulator

The *tmlvCFiat* simulator takes a timed memory read and write trace as an input, implements the *mlvCFiat* compression, and generates an output timed load data value trace. The *tmlvCFiat* maintains private *CFiat* structures per processor core. Section 6.3.1 gives functional description for the *tmlvCFiat*, Section 6.3.2 describes its implementation, and Section 6.3.3 describes verification efforts.

### 6.3.1 Functional Description

Table 6.4 shows the flags for controlling the behavior of *tmlvCFiat*. These flags are used to control the following: (i) output file size, (ii) *CFiat* structures size and configuration, and (iii) output file name. *tmlvCFiat* contains a data cache model with first access mechanism. We can set different parameters for the data cache, including cache size, cache line size, and cache associativity. In addition, we can set first-access flag granularity, i.e., the number of bytes in a data cache block guarded

by a single first-access flag. Figure 6.19 shows the format of trace descriptors generated by the *tmlvCFiat* simulator.

Table 6.4 *tmlvCFiat* flags

| Parameter | Description |
|---|---|
| --help | Generates help messages |
| --f [size] | Output file size in MB. If file exceeds the specified size the tracing stops. Default 50,000 MB. |
| --cs [kilobytes] | Cache size in kilobytes. Default is 32 KB. |
| --cls [line size] | Cache line size in bytes. Default is 32 B. |
| --ca [associativity] | Sets the associativity of the cache. Default is 4. |
| --cfg [granularity] | First access flag granularity, with each flag protecting an operand of size granularity in a cache line. Default is 4 words (8 bytes). |
| --o [filename] | Specifies output trace file name. Default is *tmlvCFiat*_out_yr_mon_day_hr_min_sec<br>Note: *.txt = descriptors, *.Statistics = Statistics of *tmlvCFiat* |

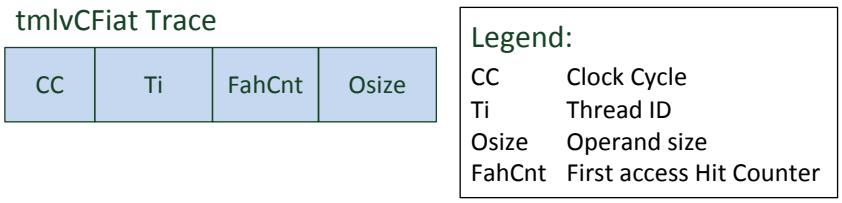


Figure 6.19 *tmlvCFiat* trace descriptor format

### 6.3.2 Implementation Details

Figure 6.20 shows the organization of the *tmlvCFiat* simulator. The tool takes two inputs: (i) a timed memory read and write trace generated by the t*ml-sTrace* tool, and (ii) the *tmlvCFiat* flags shown in Table 6.4. The tool gives two outputs: (i) emitted *tmlvCFiat* compressed load value trace, and (ii) statistics of input and output memory read value traces. The simulator reads the input trace descriptors, analyzes them (*doAnalysisMemoryInst()*), and writes emitted trace descriptor and statistics into the output files. The *doAnalysisMemoryInst* procedure checks the type and size of operand value and invokes corresponding procedures as follows: *Load_SingleCacheLine* for reads that touch a single cache line, *Store_SingleCacheLine* for writes that touch a single cache line, *Load_MultiCacheLine* for reads that span multiple cache lines, and *Store_MultiCacheLine* for stores that span multiple cache lines.

Here we take a closer look at the *Load_MultiCacheLine* procedure. It handles memory read operations that may extend over multiple cache lines. *tmlvCFiat* has a private *CFiat* structure for each thread, so the respective private structure is retrieved using thread id information for that memory write instruction. The operand is looked up in the cache. If it is a cache miss, the descriptor associated with that operand is pushed to a deque ADT and the Ti.fahCnt is reset and the requested cache lines are loaded into the cache. If it is a cache hit, the corresponding first-access flags are checked. A first-access hit occurs when all flags associated with the requested operand are set. If at least one first-access flag is not set, we have a first-access miss event.

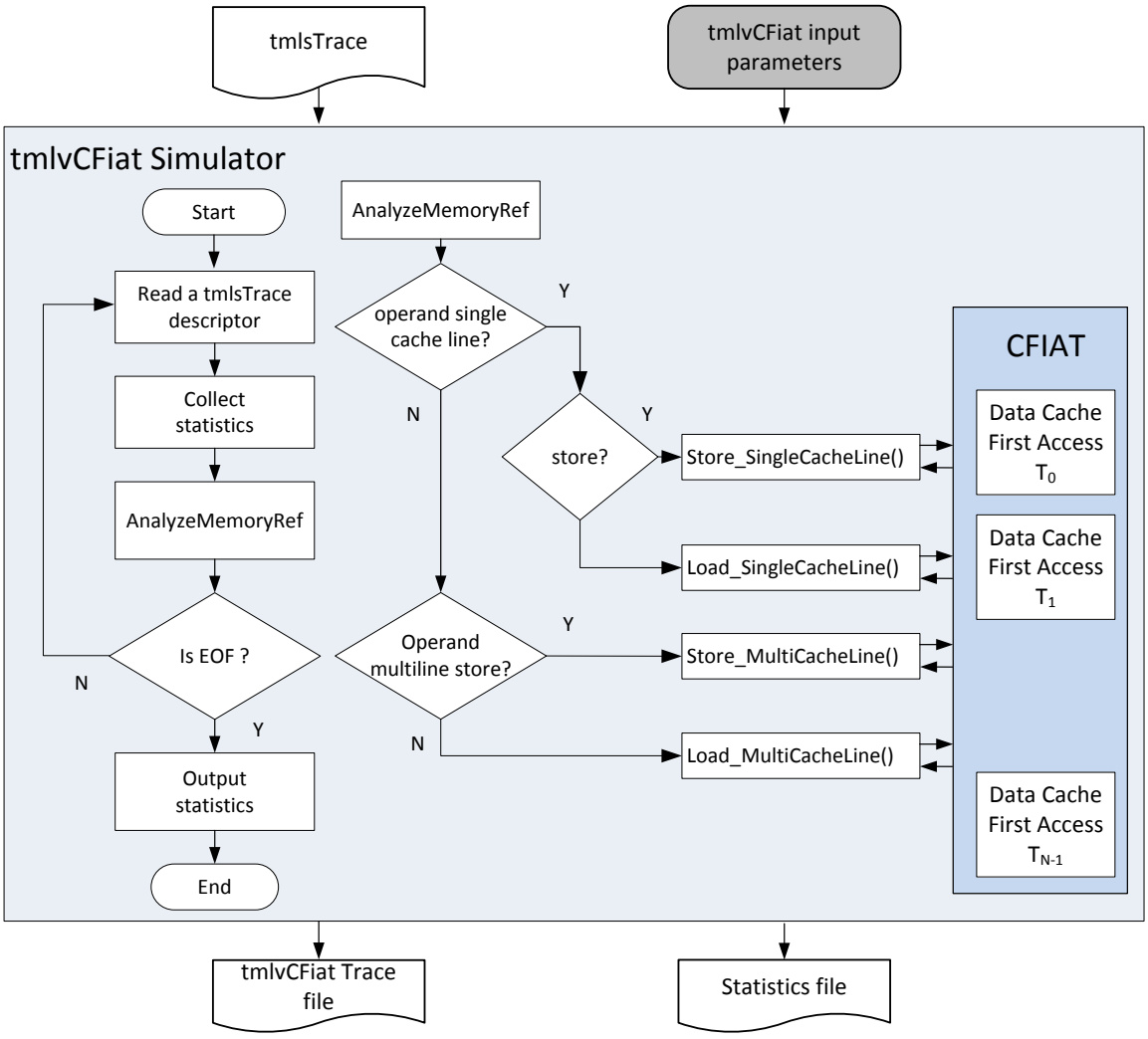


Figure 6.20 *tmlvCFiat* simulator organization

### 6.3.3 Verification Details

*tmlvCFiat* was tested with test programs to verify the correct behavior of single cache line operands and multi cache line operands. Figure 6.21(a) shows a test program for variables that fit into a single cache line. The program traverses a 4-byte integer array with 9 elements. The data cache parameters are as follows: 32-

byte cache size, 16-byte line size, 2-way set associativity, and first-access flag granularity of 16 bytes. The program should cause three misses.

Figure 6.21(b) captures the behavior of the `for` loop in line number 5. Each entry contains information of memory read instruction and *tmlvCFiat* structure parameters tag, set index, line index, new way index, is cache hit, and is descriptor emitted. We see total three cache misses (line 1-4, 5-8 and 21-24). Last two cache misses (5-8, and 21-24) is followed by three iteration of hits (9-12, 13-16, 17-20, and 25-28, 29-32, 33-36).

```
(a) Code sample
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1. volatile uint32_t uint32[9];
2. for (int i = 0; i< 9; i++){
3.        uint32[i] = 256 + i;}
4. printf("uint32 address: %p\n", uint16);
5. for (int i = 0; i< 9; i++){
6.        uint32[i];}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
(B) Run time
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --mTrace mlvCFiat_NINE_v1.gz --mld --mst
./mlvCFiat_NINE

; Multi2Sim 4.2 - A changed Simulation Framework for CPU-GPU Heterogeneous
Computing
. . .
uint32 address: 0xbffeff3c
. . .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
(C) tmlvCFiat parameter
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1. 6191139, 0,  804860e, bffeff3c, 4
2. tag = bffeff3 setIndex = 0 lineIndex = c
3. cacheHit = 0 New wayIndex = 0
4. emit ? 1

5. 6191185, 0,  804860e, bffeff40, 4
6. tag = bffeff4 setIndex = 0 lineIndex = 0
7. cacheHit = 0 New wayIndex = 1
8. emit ? 1

9. 6191186, 0,  804860e, bffeff44, 4
10. tag = bffeff4 setIndex = 0 lineIndex = 4
11. cacheHit = 1 Found in  wayIndex = 1
12. emit ? 0
```

```
13. 6191187, 0,  804860e, bffeff48, 4
14. tag = bffeff4 setIndex = 0 lineIndex = 8
15. cacheHit = 1 Found in  wayIndex = 1
16. emit ? 0

17. 6191188, 0,  804860e, bffeff4c, 4
18. tag = bffeff4 setIndex = 0 lineIndex = c
19. cacheHit = 1 Found in  wayIndex = 1
20. emit ? 0

21. 6191190, 0,  804860e, bffeff50, 4
22. tag = bffeff5 setIndex = 0 lineIndex = 0
23. cacheHit = 0 New wayIndex = 0
24. emit ? 1

25. 6191191, 0,  804860e, bffeff54, 4
26. tag = bffeff5 setIndex = 0 lineIndex = 4
27. cacheHit = 1 Found in  wayIndex = 0
28. emit ? 0

29. 6191192, 0,  804860e, bffeff58, 4
30. tag = bffeff5 setIndex = 0 lineIndex = 8
31. cacheHit = 1 Found in  wayIndex = 0
32. emit ? 0

33. 6191193, 0,  804860e, bffeff5c, 4
34. tag = bffeff5 setIndex = 0 lineIndex = c
35. cacheHit = 1 Found in  wayIndex = 0
36. emit ? 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.21 Testing *tmlvCFiat*: single cache line access

Figure 6.22(a) shows a test program for operands spanning multiple cache lines. The program stores and loads an 8-byte double precision operand. The cache parameters are set as follows: 8-byte cache size, 4-byte line size, associativity of 2, and first-access flag granularity of 4 bytes. As the operand size exceeds the cache line size, it takes two cache lines to store a single operand value. A write operation is followed by a read operation, so we expect to get a read hit. Figure 6.22(b) shows the results of program memory read and write traces and run time operand addresses. Figure 6.22(c) shows the behavior of *tmlvCFiat* structures. Each entry contains memory read / write trace and *tmlvCFiat* structure parameters address, tag, set index, line index, new way index, is local hit, and next cache line address. The simula-

tor dump shows a hit in multi-line read access (lines 9-13, and 14-17), thus matching our expectations.

```
   (a) Code sample
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   volatile double db[1];
   db[0] = 1.2;
   db[0];
   printf("double address: %p\n",db);
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
   (b) Run time
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   amrishktewar@eb136i-lacasa-gx280:~/Desktop/multi2sim/multi2sim-
4.2/samples/x86$ m --x86-sim detailed --mTrace mlvCFiat_MLine_v1.gz --mld --mst
./mlvCFiat_akt_double

   ; Multi2Sim 4.2 - A changed Simulation Framework for CPU-GPU Heterogeneous
Computing
   . . .
   double address: 0xbffeff58
   . . .
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
   (c) tmlsTrace (timed Memory read and write trace)
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   . . .
   663730, 0, 1,  804847c, bffeff58, 8
   663731, 0, 0,  8048480, bffeff58, 8
   . . .
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
   (d) tmlvCFiat parameter
   <<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   1.  663730, 0, 1,  804847c, bffeff58, 8
   2.  storeMulti: highAddr = bffeff60
   3.  storeMulti:  tag = 2fffbfd6 setIndex = 0 lineIndex = 0
   4.  localHit = 0 cacheAllHit = 0 new wayIndex is = 0

   5.  next cache line address = bffeff5c
   6.  storeMulti:  tag = 2fffbfd7 setIndex = 0 lineIndex = 0
   7.  localHit = 0 cacheAllHit = 0 new wayIndex is = 1
   8.  next cache line address = bffeff60

   9.  663731, 0, 0, 8048480, bffeff58, 8
   10. loadMulti: highAddr = bffeff60
   11. loadMulti:  tag = 2fffbfd6 setIndex = 0 lineIndex = 0
   12. localCacheHit = 1 cacheAllHit = 1 localFlagHit = 1
   13. Found in wayIndex = 0 next cache line address = bffeff5c

   14. loadMulti:  tag = 2fffbfd7 setIndex = 0 lineIndex = 0
   15. localCacheHit = 1 cacheAllHit = 1 localFlagHit = 1
   16. Found in wayIndex = 1 next cache line address = bffeff60
   17. Mulitipleload: emit ? 0
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Figure 6.22 Testing *tmlvCFiat*: multi-line cache access

## 6.4 Software to Hardware Trace Translation

Similarly to functional control-flow and load data value traces, we translate software traces generated by our simulators into hardware traces by eliminating redundant fields and applying our encoding schemes (see Figure 6.1).

Figure 6.23 shows the format of trace messages for control-flow traces, *tmcfNX_b*, *tmcfTR_b*, and *tmcfTR_e*. Figure 6.24 shows the format of trace messages for load data value traces, *tmlvNX_b*, *tmlvCF_b*, and *tmlvCF_e*. These descriptors correspond to the corresponding descriptors used for functional traces described in Section 4.2. The only difference is that all trace messages include a time stamp. Instead of encoding a full time stamp that may require a large number of bits to encode, we apply a differential encoding. The time stamp is reported relatively to the last reported time stamp and contains the number of clock cycles that expired from the last reported trace event. The differential time stamp is divided into chunks of 8 bits, with the connect bit indicating whether this is a terminating chunk (C = 0) or not (C = 1). In case of *tmcfTR_e* and *tmlvCF_e* we allow variable encoding of the differential time stamp with chunk sizes *h0* and *h1* that are determined experimentally.

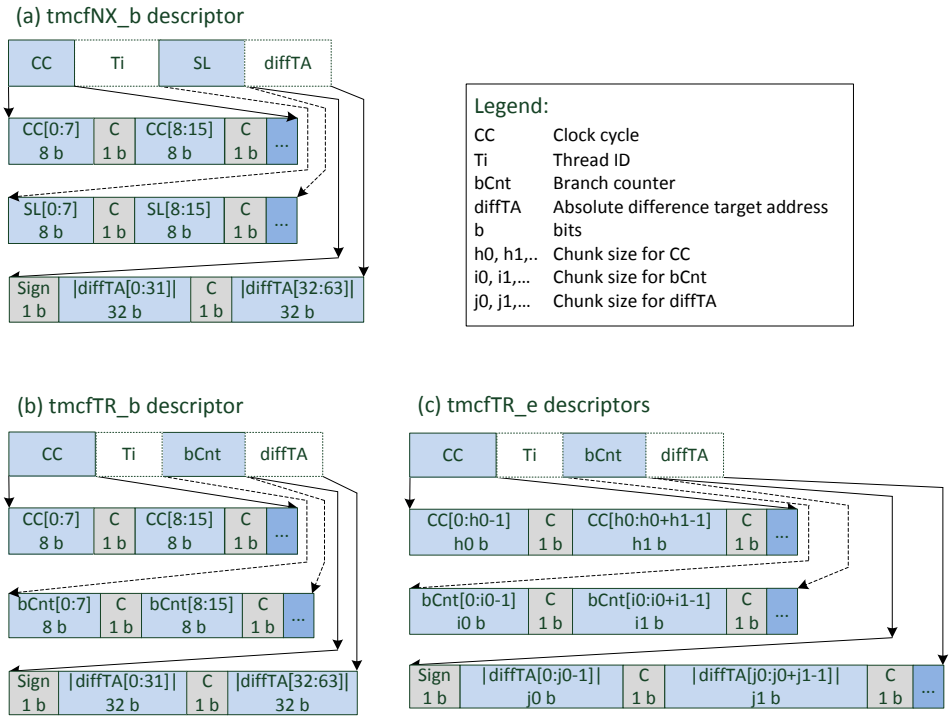


Figure 6.23 Trace descriptors for *tmcfNX_b*, *tmcfTR_b*, and *tmcfTR_e*

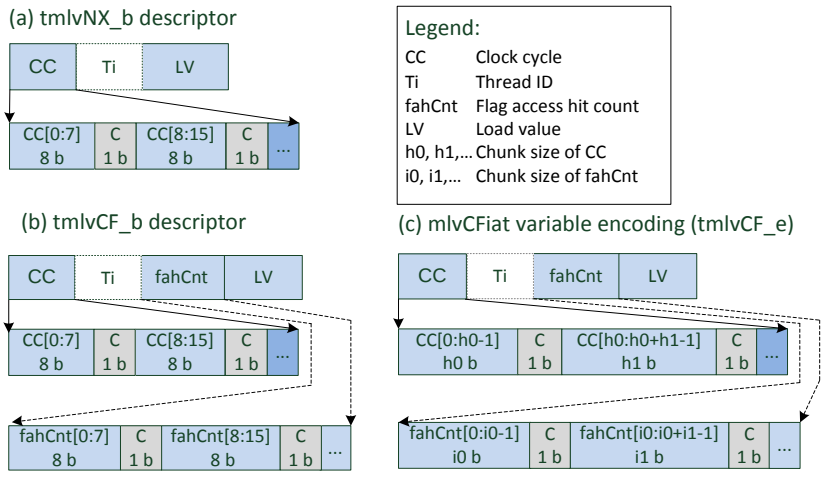


Figure 6.24 Trace descriptors for *tmlvNX_b*, *tmlvCF_b*, and *tmlvCF_e*

## 6.5 Experimental Environment

The goal of experimental evaluation is to determine the effectiveness of the newly proposed trace reduction techniques, *mcfTRaptor* and *mlvCFiat*, relative to the baseline Nexus-like control-flow and load data value traces for timed traces. As a measure of effectiveness, we use the average number of bits emitted on the trace port per instruction executed and the average number of bits emitted per clock cycle. As the workload, we use control flow and load value traces of 10 benchmarks from the Splash2 benchmark suite [31]. Machine setup is described in Section 4.3.1. The benchmarks are discussed in Section 6.5.2.

### 6.5.1 Experimental Setup

The Multi2sim simulator supports building a cycle-accurate model for a multicore processor including processor and memory hierarchy. Figure 6.25 shows a block diagram of a multicore used to generate timed traces. We use a multicore with 8 single-threaded x86 processor cores. Each core has its private L1 instruction and data caches. L1 data cache size is set to 8KB and L1 instruction cache size is set to 8KB with a latency of 2 clock cycles per core. The multicore has a shared L2 cache with a hit latency of 4 clock cycles. L2 cache size is 64KB x Number of cores e.g for 1 core L2 cache size is 64KB, and similar way for 8 core L2 cache size is 512KB. It has main memory with a block size of 256 KB and a latency of 200 clock cycle. The networks between L1 ~ L2 cache and L2 ~ Main Memory are identical in buffer size and bandwidth. The experiments are conducted on a Dell PowerEdge T620 server described in Section 5.3.1.

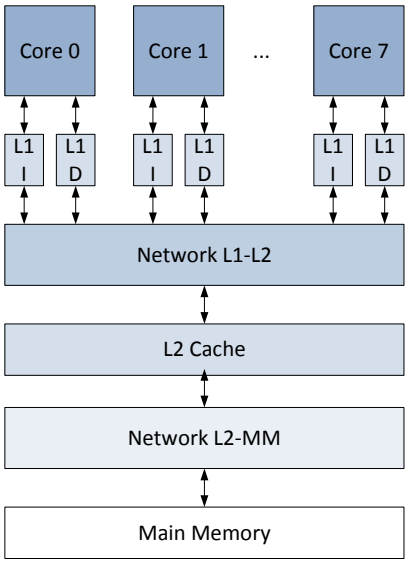


Figure 6.25 Block diagram of a modeled multicore in Mult2Sim

### 6.5.2 Benchmarks

The workload used is the Splash2 suite of [28] benchmarks. Splash2 benchmarks are predecessor benchmarks to Splash2x. The benchmarks are precompiled for Intel's IA32 ISA by Multi2Sim developers [31] . Each benchmark was executed with N = 1, 2, 4 and 8 processor cores using the Simsmall input set.

The trace port bandwidth for control-flow traces depends on benchmark characteristics, specifically the frequency and type of control-flow instructions. Similarly, the trace port bandwidth for load data value traces depends on the frequency and type of memory reads and data value sizes.

Table 6.5 shows the control flow characterization of Splash2 benchmarks. We show the number of executed instructions (instruction count) and the number of instructions executed per a clock cycle (IPC). The last four columns show the frequency of control flow instructions for single threaded programs (N = 1), as well as the frequency of conditional direct branches (C, D), unconditional direct branches (U,

D), and unconditional indirect branches (U, I). The number of instructions slightly increases with an increase in the number of threads due to overhead and data partitioning. The number of instructions varies between 0.45 (*lu*) and 5.03 billion (*water-spatial*). The Splash2 benchmarks exhibit diverse behavior with respect to control flow instructions; their frequency ranges from as high as ~14% (*water-nsquared, lu, raytrace*) to as low as 3.96% (*radix*). The total frequency for the entire benchmark suite is relatively low (10.46%). The conditional direct branches are the most frequent type of branches.

Table 6.5 Splash2 benchmark suite control flow characterization

| *Benchmark* | *Instruction Count [x10$^9$]* | | | | *Instructions per cycle* | | | | *% branch for Thread = 1* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Thread* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* | *branch* | *C,D* | *U,D* | *U,I* |
| *barnes* | 2.13 | 2.13 | 2.13 | 2.14 | 0.405 | 0.827 | 1.515 | 2.760 | 10.25 | 6.10 | 2.44 | 1.71 |
| *cholesky* | 1.27 | 1.37 | 1.85 | 2.77 | 0.307 | 0.610 | 1.372 | 3.034 | 7.11 | 6.50 | 0.42 | 0.18 |
| *fft* | 0.92 | 0.92 | 0.92 | 0.92 | 0.280 | 0.563 | 1.000 | 1.584 | 8.35 | 6.06 | 1.14 | 1.14 |
| *fmm* | 2.79 | 2.79 | 2.84 | 2.88 | 0.403 | 0.800 | 1.570 | 2.942 | 7.05 | 6.57 | 0.36 | 0.12 |
| *lu* | 0.45 | 0.45 | 0.45 | 0.45 | 0.577 | 1.088 | 1.896 | 2.995 | 13.63 | 11.98 | 0.83 | 0.82 |
| *radiosity* | 2.23 | 2.32 | 2.31 | 2.31 | 0.636 | 1.242 | 2.354 | 4.419 | 11.70 | 6.48 | 3.40 | 1.81 |
| *radix* | 1.59 | 1.59 | 1.59 | 1.60 | 0.219 | 0.373 | 0.605 | 0.752 | 3.96 | 1.85 | 1.06 | 1.06 |
| *raytrace* | 2.47 | 2.47 | 2.47 | 2.47 | 0.501 | 1.166 | 2.164 | 3.601 | 12.12 | 7.74 | 2.65 | 1.73 |
| *water-nsq* | 0.74 | 0.74 | 0.74 | 0.75 | 0.701 | 1.458 | 3.038 | 5.293 | 14.12 | 11.56 | 2.16 | 0.41 |
| *water-sp* | 5.03 | 5.03 | 5.03 | 5.03 | 0.820 | 1.346 | 2.204 | 3.510 | 13.53 | 11.49 | 1.51 | 0.53 |
| *Total* | 19.61 | 19.81 | 20.33 | 21.31 | 0.453 | 0.867 | 1.564 | 2.567 | 10.46 | 7.82 | 1.69 | 0.95 |

Table 6.6 shows the memory read flow characterization of Splash2 benchmarks while varying the number of threads (N = 1, 2, 4, and 8). We show the number of instructions executed, the number of instructions executed per clock cycle, and the frequency of memory read instructions with respect to the total number of in-

structions. The frequency of memory read instructions increases with an increase in the number of threads. The percentage of memory read instructions varies between 13.02 (*fmm*) and 35.09 (*radix*). The overall frequency of read instructions is 22.77% for N = 1 and 23.57% for N = 8. The overall IPC as a function of the number of cores indicates how well the performance of individual benchmarks scale. For example, *cholesky* scales well, reaching speedup S(8) = IPC(N= 8)/IPC(N=1) = 9.8 for 8 cores, but *radix* does not scale well because its 8-core speedup is only S(8) = 3.4

Table 6.6 Splash2 benchmark suite memory read characterization

| *Benchmark* | *Instruction Count [x10$^9$]* | | | | *Instructions per cycle* | | | | *% Load for Thread = 1* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Thread* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* |
| *barnes* | 2.13 | 2.13 | 2.13 | 2.14 | 0.405 | 0.827 | 1.515 | 2.760 | 28.78 | 28.78 | 28.78 | 28.79 |
| *cholesky* | 1.27 | 1.37 | 1.85 | 2.77 | 0.307 | 0.610 | 1.372 | 3.034 | 27.78 | 29.37 | 30.17 | 31.08 |
| *fft* | 0.92 | 0.92 | 0.92 | 0.92 | 0.280 | 0.563 | 1.000 | 1.584 | 19.20 | 19.20 | 19.20 | 19.21 |
| *fmm* | 2.79 | 2.79 | 2.84 | 2.88 | 0.403 | 0.800 | 1.570 | 2.942 | 13.02 | 13.06 | 13.36 | 13.68 |
| *lu* | 0.45 | 0.45 | 0.45 | 0.45 | 0.577 | 1.088 | 1.896 | 2.995 | 20.20 | 20.22 | 20.25 | 20.31 |
| *radiosity* | 2.23 | 2.32 | 2.31 | 2.31 | 0.636 | 1.242 | 2.354 | 4.419 | 27.51 | 27.49 | 27.42 | 27.02 |
| *radix* | 1.59 | 1.59 | 1.59 | 1.60 | 0.219 | 0.373 | 0.605 | 0.752 | 35.09 | 35.09 | 35.09 | 35.09 |
| *raytrace* | 2.47 | 2.47 | 2.47 | 2.47 | 0.501 | 1.166 | 2.164 | 3.601 | 28.49 | 28.47 | 28.51 | 28.48 |
| *water-nsq* | 0.74 | 0.74 | 0.74 | 0.75 | 0.701 | 1.458 | 3.038 | 5.293 | 16.31 | 16.33 | 16.36 | 16.42 |
| *water-sp* | 5.03 | 5.03 | 5.03 | 5.03 | 0.820 | 1.346 | 2.204 | 3.510 | 17.38 | 17.38 | 17.38 | 17.38 |
| *Total* | 19.61 | 19.81 | 20.33 | 21.31 | 0.453 | 0.867 | 1.564 | 2.567 | 22.77 | 22.92 | 23.17 | 23.57 |

The memory trace port bandwidth depends not only on the frequency of read operations but also on operand size. Table 6.7 shows the frequency of memory reads for different operand sizes: 8-bit bytes, 16-bit words, 32-bit double words, 64-bit quad words, 80-bit extended precision operands, 128-bit octa-word operands, 256-bit hexa-word operands, and others. The results indicate that double-word and quad-

word operands dominate in all benchmarks except of *radix* which has a majority of word operands.

Table 6.7 Characterization of memory reads in Splash2

| *Benchmark* | *Total Memory reads* | *Byte operand* | *Word operand* | *Doubleword operand* | *Quadword operand* | *Extended precesion operand* | *Octaword operands* | *Others* |
|---|---|---|---|---|---|---|---|---|
| *barnes* | 613094350 | 0 | 3.26 | 60.1 | 36.65 | 0 | 0 | 0 |
| *cholesky* | 352542968 | 1.33 | 0 | 54.09 | 44.59 | 0 | 0 | 0 |
| *fft* | 176253017 | 0.01 | 9.52 | 41.16 | 49.31 | 0 | 0 | 0 |
| *fmm* | 362805364 | 0 | 0.15 | 16.3 | 83.55 | 0 | 0 | 0 |
| *lu* | 90032624 | 0 | 2.04 | 41.77 | 56.19 | 0 | 0 | 0 |
| *radiosity* | 613310395 | 0 | 0 | 90.61 | 9.39 | 0 | 0 | 0 |
| *radix* | 558024069 | 4.51 | 29.31 | 57.16 | 9.02 | 0 | 0 | 0 |
| *raytrace* | 702413242 | 0.8 | 0.96 | 58.93 | 39.3 | 0 | 0 | 0 |
| *water-nsq* | 120913483 | 0.6 | 0.01 | 23.2 | 76.19 | 0 | 0 | 0 |
| *water-sp* | 874383921 | 0.55 | 0.02 | 22.63 | 76.8 | 0 | 0 | 0 |
| *Total* | 4463773433 | 0.92 | 4.70 | 50.25 | 44.14 | 0.00 | 0.00 | 0.00 |

### 6.5.3 Experiments

Table 6.8 lists (technique, configuration) pairs considered in the experimental evaluation. For control-flow traces we compare the trace port bandwidth of *tmcfNX_b* versus *tmcfTR_b* and *tmcfTR_e*, while varying the number of threads (N=1, 2, 4, and 8). To assess the impact of organization and size of predictor structures in *tmcfTRaptor* on its effectiveness, we consider the following configurations:

- *Small*: 512-entry gshare outcome predictor and an 8-entry RAS;
- *Medium*: 1024-entry gshare outcome predictor, a 16-entry RAS, and a 16-entry iBTB (2x8); and
- *Large*: 4096-entry gshare outcome predictor, a 32-entry RAS, and a 64-entry iBTB (2x32).

The index function for the gshare outcome predictor is *gshare.index* = *BHR[log2(p):0] xor PC[4+ log2(p):4]*, where the Branch History Register (BHR) register holds the outcome history of the last *log2(p)* conditional branches. The iBTB holds target addresses that are tagged. Both the iBTB tag and the iBTB index are calculated based on the information maintained in the path information register [29], [30].

Table 6.8 Timed trace experiments

| *control flow (vary N = 1,2,4 & 8)* | | | | *load flow (vary N = 1,2,4 & 8)* | | | |
|---|---|---|---|---|---|---|---|
| *Timed Method* | *Small* | *Medium* | *Large* | *Timed Method* | *Small* | *Medium* | *Large* |
| *tmcf_NX_b* | √ | | | *tmlv_NX_b* | √ | | |
| *tmcf_TR_b* | √ | √ | √ | *tmlv_CF_b* | √ | √ | √ |
| *tmcf_TR_e* | √ | √ | √ | *tmlv_CF_e* | √ | √ | √ |

For load data value traces, we compare the trace port bandwidth of *tmlvNX_b* versus *tmlvCF_b* and *tmlvCF_e* while varying the number of threads (N=1, 2, 4, and 8). To assess the impact of organization and size data caches on *tmlvCFiat* effectiveness, we consider the following cache configurations. All cache structures are 4-way set associative, use round robin replacement policy, feature line size of 64 bytes, and first-access flag granularity is set to 4 bytes. We consider three configurations as follows:

- *Small*: data cache of 16 KB;
- *Medium*: data cache of 32 KB; and
- *Large*: data cache of 64 KB.

### 6.5.4 Variable Encoding

Similarly to Section 4.3.4, we analyze the generated traces to find optimal variable length chunk sizes for the descriptor fields such as Ti.CC (time), Ti.bCnt, and Ti.fahCnt in both timed control-flow (*tmcfTR_e*) and timed load data value (*tmlvCF_e*) traces. Table 6.9 summarizes chunk sizes that work well for all benchmarks. To encode differential time stamps, our results indicate the first chunk should have 4 bits, followed by a connect bit, and every other chunk should have 2 bits. Interestingly, this combination (h0, h1) = (4, 2) works well regardless of the size of predictor structures or caches and is used in both *tmcfTR_e* and *tmlvCF_e*.

Table 6.9 Summary variable encoding parameter for different fields

| *Variable Encoding paramter i_j* | | | | |
|---|---|---|---|---|
| *Mechanism* | *fields* | *small* | *medium* | *large* |
| *tmcfTRaptor* | *Time* | 4_2 | 4_2 | 4_2 |
| | *bCnt* | 3_2 | 3_2 | 3_2 |
| | *DiffTA* | 3_4 | 3_4 | 3_4 |
| *tmlvCFiat* | *Time* | 4_2 | 4_2 | 4_2 |
| | *fahCnt* | 2_2 | 2_2 | 2_2 |

# CHAPTER 7

## TRACE PORT BANDWIDTH ANALYSIS FOR TIMED TRACES

God does not work for you, he works with you

--Rev. Pandurang Shastri Athavale

This chapter shows the main results of the experimental evaluation for timed traces. We measure the trace port bandwidth for control-flow and load data value traces as a function of the number of processor cores, encoding mechanism, as well as configuration parameters of the trace filtering structures. Trace port bandwidth is measured in bits per instruction executed [bpi], calculated as the number of bits needed to be streamed divided by the number of instructions executed. In addition, we consider bits per clock cycles [bpc], calculated as the total number of bits streamed divided by the number of clock cycles needed to complete a benchmark of interest. Section 7.1 discusses the results for timed control-flow functional traces, specifically the trace port bandwidth requirements for the Nexus-like timed control-flow trace, *tmcfNX_b*, as well as the trace port bandwidth for the *tmcfTRaptor* technique with the fixed encoding, *tmcfTR_b*, and with the variable encoding, *tmcfTR_e*. Section 7.2 discusses the results for timed memory load data value traces, specifically the trace port bandwidth requirements for the Nexus-like traces, *tmlvNX_b*, and the *tmlvCFiat* technique with the fixed, *tmlvCF_b*, and with the variable encoding, *tmlvCF_e*.

## 7.1 Trace Port Bandwidth for Timed Control-Flow Traces

### 7.1.1 *tmcfNX_b*

Table 7.1 shows the trace port bandwidth (TPB) in bpi and bpc for the Nexus-like timed control flow traces, *tmcfNX_b*, for all benchmarks as a function of the number of threads/cores (N=1, 2, 4, and 8). The last row shows the total trace port bandwidth when all benchmarks are considered together. The total bandwidth in bits per instruction is calculated as the sum of trace sizes for all benchmarks divided by the sum of the number of instructions executed for all benchmarks. Similarly, the total bandwidth in bits per cycle is calculated as the sum of trace sizes for all benchmarks divided by the sum of the execution times in clock cycles for all benchmarks. For single-threaded benchmarks (N = 1), the TPB ranges between 0.83 bpi for *fmm* and 2.45 bpi for *lu*. The required bandwidth varies across benchmarks and is highly correlated with the frequency of control-flow instructions. Thus, *lu*, *radiosity, raytrace* and *barnes* have relatively high TPB requirements due to the relatively high frequency of branch instructions and especially indirect branches (see Table 6.5), unlike *fmm,* which has very low TPB requirements due to the extremely small frequency of control flow instructions. The required trace port bandwidth in bits per instruction increases as we increase the number of cores, due to additional information such as Ti that needs to be streamed out. Thus, when N = 8, the TPB ranges between 1.06 bpi for *fmm* and 2.81 bpi for *lu*. The total bandwidth for the entire benchmark suite ranges between 1.57 bpi when N = 1 and 1.90 bpi when N = 8.

Table 7.1 Trace port bandwidth for *tmcfNX_b* for Splash2 benchmark

| *Benchmark* | *Trace port Bandwidth [bpi]* | | | | *Trace Port Bandwidth [bpc]* | | | |
|---|---|---|---|---|---|---|---|---|
| *Cores* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* |
| *barnes* | 1.95 | 2.16 | 2.17 | 2.18 | 0.79 | 1.79 | 3.29 | 6.03 |
| *cholesky* | 1.00 | 1.14 | 1.72 | 2.11 | 0.31 | 0.71 | 2.36 | 6.91 |
| *fft* | 1.62 | 1.81 | 1.81 | 1.81 | 0.45 | 1.02 | 1.81 | 2.86 |
| *fmm* | 0.83 | 0.97 | 1.06 | 1.15 | 0.33 | 0.78 | 1.67 | 3.41 |
| *lu* | 2.45 | 2.81 | 2.81 | 2.81 | 1.41 | 3.06 | 5.33 | 8.41 |
| *radiosity* | 2.10 | 2.35 | 2.38 | 2.38 | 1.33 | 2.92 | 5.56 | 10.74 |
| *radix* | 1.11 | 1.23 | 1.31 | 1.36 | 0.24 | 0.46 | 0.79 | 1.02 |
| *raytrace* | 2.08 | 2.32 | 2.32 | 2.33 | 1.04 | 2.70 | 5.02 | 8.37 |
| *water-ns* | 1.51 | 1.73 | 1.73 | 1.74 | 1.06 | 2.53 | 5.26 | 9.19 |
| *water-sp* | 1.55 | 1.78 | 1.78 | 1.78 | 1.27 | 2.40 | 3.92 | 6.25 |
| *Total* | 1.57 | 1.78 | 1.83 | 1.90 | 0.71 | 1.54 | 2.87 | 4.94 |

Whereas the bandwidth in bits per instruction increases with the number of cores, it does not fully capture the pressure on multiple processor cores place on the trace port, a shared resource. The trace port bandwidth in bits per clock cycle better illustrates this pressure. Thus, the control-flow trace for *radiosity* with 8 threads executing on 8 cores requires 10.74 bits per clock cycle on average. Generally, the trace port bandwidth in bits per clock cycle is a function of benchmark characteristics as well as the scalability of individual benchmarks. The total TPB in bpc ranges between 0.71 bpc when N = 1 and 4.94 bpc when N = 8. These results indicate that capturing control-flow trace on the fly in multicores requires significantly large trace buffers and wide trace ports. As shown in the next section, one alternative is to develop hardware techniques that significantly reduce the volume and size of trace messages that are streamed out.

### 7.1.2 *tmcfTRaptor*

The effectiveness of *tmcfTRaptor* in reducing the trace port bandwidth depends on prediction rates as the trace messages are generated only on rare misprediction events. Table 7.2 shows the total misprediction rates collected on the entire Splash2 benchmark suite for the *Small*, *Medium*, and *Large* predictor configurations, when the number of cores is varied between N = 1 and N = 8. Figure 7.1 illustrates the total outcome misprediction rates and Figure 7.2 shows the total target address misprediction rates as a function of the number of threads and predictor configuration. The outcome misprediction rates decrease as we increase the size of the *gshare* predictor. They also slightly decrease with an increase in the number of processor cores as fewer branches compete for the same resource. Relatively high misprediction rates indicate that even better trace compression could be achieved if more sophisticated outcome predictors are used. However, this is out of scope of this work. The target address misprediction rates are very low for the Medium and *Large* configurations. The *Small* configuration does not include the iBTB predictor resulting in higher target address misprediction rates. These results demonstrate a strong potential of *tmcfTRaptor* to reduce the trace port bandwidth requirements.

Table 7.2 Total outcome and target address misprediction rates for Splash

| | *Outcome Misprediction Rate [%]* | | | | *Target Address Misprediction Rate [%]* | | | |
|---|---|---|---|---|---|---|---|---|
| *Configuration* | *N=1* | *N=2* | *N=4* | *N=8* | *N=1* | *N=2* | *N=4* | *N=8* |
| *Small* | 7.86 | 7.83 | 7.20 | 6.23 | 8.47 | 8.59 | 8.58 | 8.55 |
| *Medium* | 6.69 | 6.67 | 6.14 | 5.31 | 2.05 | 2.12 | 2.08 | 2.05 |
| *Large* | 5.27 | 5.26 | 4.84 | 4.20 | 0.57 | 0.58 | 0.57 | 0.57 |

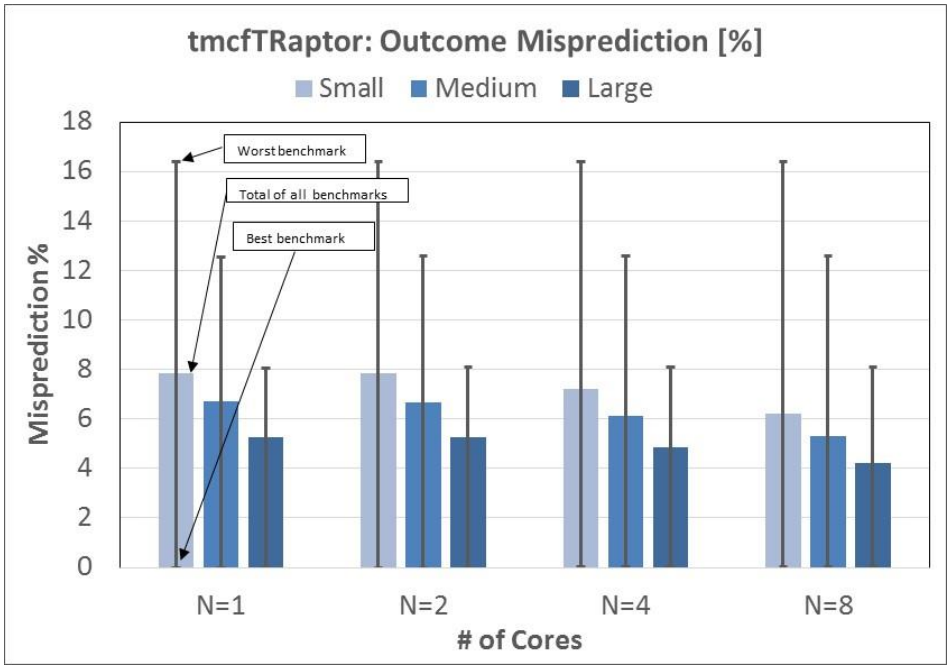


Figure 7.1 Outcome misprediction rates for Splash2 benchmark

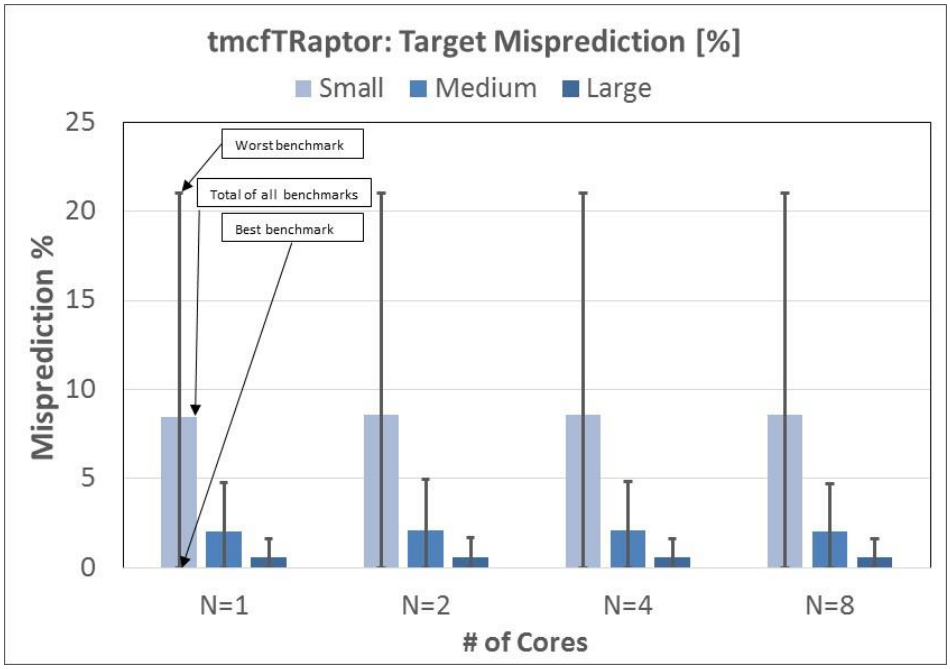


Figure 7.2 Target address misprediction rates for Splash2 benchmark

To quantify the effectiveness of *tmcfTRaptor*, we analyze the total trace port bandwidth in bits per instruction for the entire benchmark suite as a function of the number of threads (N = 1, 2, 4, and 8), the encoding mechanism (*tmcfTR_b* and *tmcfTR_e*), and the *tmcfTRaptor* organization (Small, Medium, and Large). Figure 7.3 shows the total average trace port bandwidth.

TR_b dramatically reduces the total trace port bandwidth as follows:

- Small configuration: 0.19 bpi (N = 1) and 0.20 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* 8.41 times for N=1 and 9.71 times for N=8.
- Medium configuration: 0.136 bpi (N = 1) and 0.144 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* 11.57 times for N = 1 and 13.19 times for N = 8.
- Large configuration: 0.10 bpi (N = 1) and 0.11 (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* 14.98 times for N = 1 and 17.02 for N = 8.

TR_e further reduce the average trace port bandwidth as follow:

- Small configuration: 0.13 bpi (N = 1) and 0.14 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* is 12.11 times for N = 1 and 13.26 for N = 8.
- Medium configuration: 0.11 bpi (N = 1) and 0.12 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* is 14.89 times for N = 1 and 16.35 for N = 8.

- Large configuration: 0.087 bpi (N = 1) and 0.095 bpi (N = 8). This is equivalent to reducing the trace port bandwidth relative to *tmcfNX_b* is 18.15 times for N = 1 and 20.07 for N = 8.

Table 7.3 shows the compression ratio for t*mcfTR_b* relative to *tmcfNX_b*, as a function of the predictor configuration (Small, Medium, Large) and the number of threads for each benchmark. The compression ratio is calculated as follows: TPB(*tmcfNX_b*)/TPB(*tmcfTR_b*). For N = 1, the compression ratio ranges from 4.07 (*raytrace*) to 30,296 (*radix*) for the Small configuration and from 10.91 (*water-sq*) to 42,393 (*radix*) for the Large configuration. For N = 8, the compression ratio ranges from 4.04 (*raytrace*) to 13,570 (*radix*) for the Small configuration and from 10.76 (*water-nsquared*) to 13,570 (*radix*) for the Large configuration. The gains in compression ratio achieved when increasing the number of cores (threads) are relatively more pronounced when we are using smaller predictor structures.

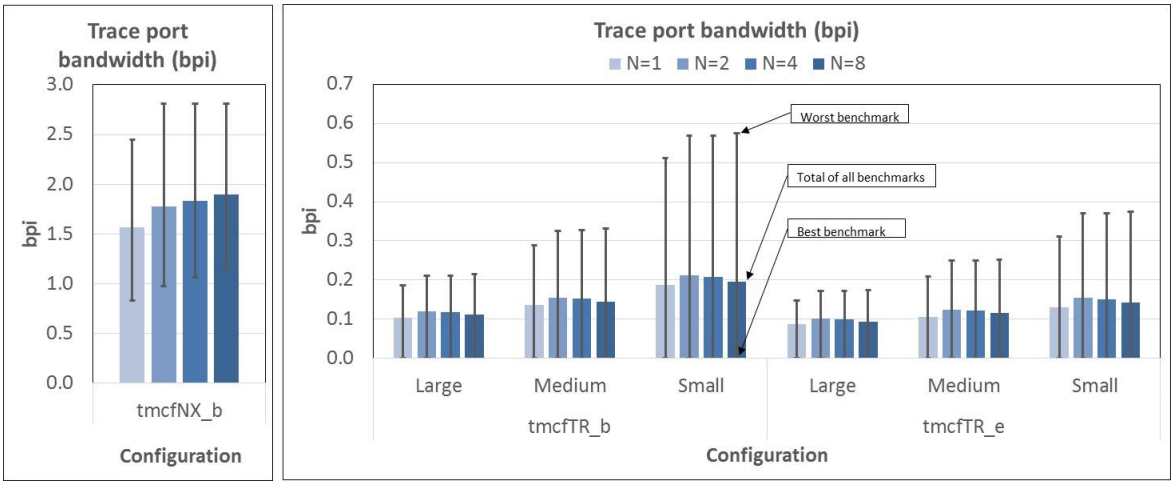


Figure 7.3 Total trace port bandwidth in bpi for timed control flow traces

Table 7.3 Compression ratio for *tmcfTR_b* relative to *tmcfNX_b*

| *cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *barnes* | 15.66 | 19.14 | 19.61 | 15.36 | 18.60 | 19.04 | 15.31 | 18.51 | 18.96 | 15.32 | 18.50 | 18.98 |
| *cholesky* | 18.49 | 29.52 | 34.03 | 18.97 | 28.71 | 32.85 | 33.99 | 49.95 | 57.07 | 60.78 | 87.51 | 99.95 |
| *fft* | 90.07 | 91.59 | 92.64 | 88.09 | 89.84 | 90.74 | 86.70 | 87.96 | 89.26 | 85.26 | 86.90 | 87.74 |
| *fmm* | 11.79 | 13.30 | 14.49 | 12.40 | 13.91 | 15.19 | 13.73 | 15.41 | 16.77 | 15.08 | 16.92 | 18.42 |
| *lu* | 19.34 | 19.41 | 19.51 | 19.09 | 19.17 | 19.27 | 19.08 | 19.17 | 19.28 | 18.98 | 19.10 | 19.24 |
| *radiosity* | 5.75 | 8.53 | 13.74 | 5.64 | 8.28 | 13.24 | 5.68 | 8.34 | 13.27 | 5.94 | 8.74 | 13.87 |
| *radix* | 30296 | 41510 | 42393 | 12335 | 30124 | 30610 | 13085 | 13085 | 13085 | 13570 | 13570 | 13570 |
| *raytrace* | 4.07 | 7.22 | 11.13 | 4.08 | 7.12 | 10.98 | 4.07 | 7.09 | 10.95 | 4.04 | 7.01 | 10.80 |
| *water-ns* | 7.80 | 9.48 | 10.91 | 7.77 | 9.49 | 10.93 | 7.76 | 9.49 | 10.92 | 7.69 | 9.39 | 10.76 |
| *water-sp* | 9.25 | 10.01 | 11.62 | 9.18 | 9.93 | 11.53 | 9.19 | 9.93 | 11.53 | 9.18 | 9.92 | 11.51 |
| *Total* | 8.41 | 11.57 | 14.98 | 8.39 | 11.42 | 14.80 | 8.85 | 12.04 | 15.57 | 9.71 | 13.19 | 17.02 |

Table 7.4 shows the compression ratio for *tmcfTR_e* relative to *tmcfNX_b*. *tmcfTR_e* achieves higher compression ratios than *tmcfTR_b*, especially when using the Small predictor structures that have a relatively high number of mispredictions and thus reported bCnt values will be shorter. For N = 1, the compression ratio ranges from 6.70 (*raytrace*) to 56,102 (*radix*) for the Small configuration and from 12.77 (*water-nsquared*) to 69,711 (*radix*) for the Large configuration. For N = 8, the compression ratio ranges from 6.20 (*raytrace*) to 13,570 (radix) for the Small configuration and from 12.45 (*water-nsquared*) to 13,570 (*radix*) for the Large configuration.

Table 7.4 Compression ratio for *tmcfTR_e* relative to *tmcfNX_b*

| *cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *barnes* | 21.17 | 24.07 | 24.59 | 19.98 | 22.73 | 23.24 | 19.94 | 22.67 | 23.18 | 20.00 | 22.71 | 23.24 |
| *cholesky* | 28.60 | 34.14 | 38.02 | 27.27 | 32.66 | 36.42 | 48.27 | 57.64 | 64.07 | 85.73 | 101.88 | 113.39 |
| *fft* | 95.36 | 95.93 | 97.08 | 92.59 | 93.54 | 94.52 | 91.06 | 91.98 | 92.92 | 88.17 | 89.48 | 90.38 |
| *fmm* | 14.09 | 15.17 | 16.15 | 14.49 | 15.60 | 16.70 | 16.03 | 17.26 | 18.45 | 17.63 | 18.97 | 20.31 |
| *lu* | 20.36 | 20.41 | 20.52 | 20.06 | 20.12 | 20.22 | 20.08 | 20.15 | 20.26 | 20.03 | 20.11 | 20.26 |
| *radiosity* | 9.38 | 12.69 | 18.06 | 8.60 | 11.65 | 16.75 | 8.64 | 11.71 | 16.80 | 9.02 | 12.23 | 17.54 |
| *radix* | 56102 | 68676 | 69711 | 37431 | 44227 | 45303 | 26580 | 31397 | 32878 | 13570 | 13570 | 13570 |
| *raytrace* | 6.70 | 9.90 | 14.15 | 6.26 | 9.29 | 13.48 | 6.24 | 9.27 | 13.47 | 6.20 | 9.21 | 13.36 |
| *water-ns* | 9.90 | 11.43 | 12.77 | 9.52 | 11.11 | 12.48 | 9.51 | 11.12 | 12.50 | 9.47 | 11.09 | 12.45 |
| *water-sp* | 11.27 | 11.93 | 13.57 | 10.83 | 11.49 | 13.11 | 10.80 | 11.45 | 13.06 | 10.77 | 11.42 | 13.03 |
| *Total* | 12.11 | 14.89 | 18.15 | 11.48 | 14.17 | 17.43 | 12.10 | 14.93 | 18.34 | 13.26 | 16.35 | 20.07 |

Figure 7.4 shows the total trace port bandwidth in bits per clock cycle for *tmcfNX_b* (left), *tmcfTR_b* and *tmcfTR_e* (right). *tmcfTR_e* offers superior performance, the t*mcfTR_e* for Large configuration when N = 8 requires merely 0.246 bpc on average (ranging from ~0 to 0.739 bpc), whereas *tmcfNX_b* requires 0.290 bpc (ranging between ~0 to 0.854 bpc). These results further underscore the effectiveness of the proposed *tmcfTRaptor* predictor structures for a range of diverse benchmarks. The results indicate that with *tmcfTR_e* even a single-bit data trace port is sufficient to stream out the control-flow trace from an 8-core system-on-a-chip, thus dramatically reducing the cost of on-chip debugging infrastructure.

The improvements of *tmcfTR_b* and *tmcfTR_e* over *tmcfNX_b* are slightly smaller than those observed for of functional traces due to the overhead required for reporting time stamps.

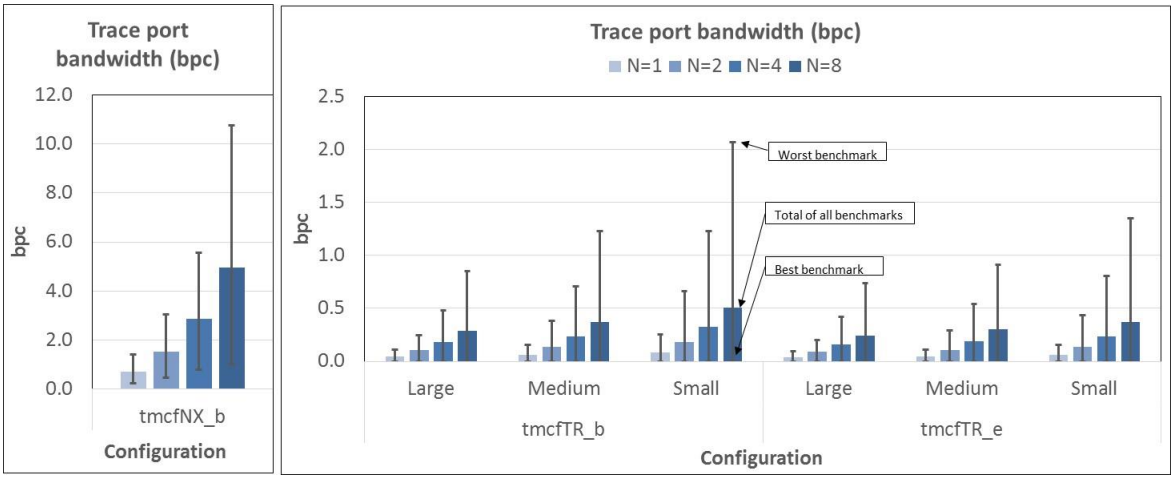


Figure 7.4 Trace port bandwidth in bpc for control flow traces

## 7.2 Trace Port Bandwidth for Timed Memory Load Data Value Traces

### 7.2.1 *tmlvNX_b*

Table 7.5 shows the trace port bandwidth in bpi and in bpc for the Nexus-like timed load data value traces, *tmlvNX_b*, for all benchmarks as a function of the number of threads/cores (N=1, 2, 4, and 8). The last row shows the total trace port bandwidth when all benchmarks are considered together. The total bandwidth in bits per instruction is calculated as the sum of trace sizes for all benchmarks divided by the sum of the number of instructions executed for all benchmarks. Similarly, the total bandwidth in bits per cycle is calculated as the sum of trace sizes for all benchmarks divided by the sum of the execution times in clock cycles for all benchmarks.

For single-threaded benchmarks (N = 1), the TPB ranges between 8.82 bpi for *fmm* and 15.29 bpi for *cholesky*. The required bandwidth varies across benchmarks

and is highly correlated with the frequency and type of memory reads. Thus, *barnes* and *cholesky* have relatively high TPB requirements due to the relatively high frequency of load instructions, unlike *fmm* which has very low TPB requirements due to the extremely low frequency of memory read instructions. The trace port bandwidth increases slightly with an increase in the number of cores for two reasons: (a) an increase in the number of bits needed to report thread index, and (b) an increase in the frequency of load instructions (caused by synchronization primitives). Thus, when N = 8, the TPB ranges between 9.30 bpi for *fmm* and 16.01 bpi for *raytrace*. The total bandwidth for the entire benchmark suite ranges between 12.34 bpi when N = 1 and 12.98 bpi when N = 8.

Whereas the bandwidth in bits per instruction increases with the number of cores, it does not fully capture the pressure multiple processor cores place on the trace port, a shared resource. The trace port bandwidth in bits per clock cycle better illustrates this pressure, load data value trace for *cholesky* reaches 4.69 bpc when N = 1 and 47.61 bpc when N = 8; *fmm* requires 3.55 when N = 1 and 27.63 bpc when N = 8. The total trace port bandwidth in bpc ranges from 5.59 when N = 1 to 33.79 when N = 8. The trace port bandwidth in bpc is heavily influenced not only by the frequency and type of memory reads but also by the scalability of individual benchmarks. For example, *barnes*, *water_spa*, and *fmm* exhibit high scalability (see IPC in Table 6.6) which contributes to a significant increase in the trace port bandwidth requirements for N = 4 and N = 8. These results indicate that capturing load data value traces on the fly in multicores requires large trace buffers and wide trace ports. As shown in the next section, one alternative is to develop hardware tech-

niques that significantly reduce the volume and size of trace data that are streamed out.

Table 7.5 Trace port bandwidth for *tmlvNX_b* for Splash2 benchmark

| *Benchmark* | *Trace port Bandwidth [bpi]* | | | | *Trace Port Bandwidth [bpc]* | | | |
|---|---|---|---|---|---|---|---|---|
| *Cores* | *1* | *2* | *4* | *8* | *1* | *2* | *4* | *8* |
| *barnes* | 15.04 | 15.89 | 15.87 | 15.81 | 6.09 | 13.15 | 24.07 | 43.81 |
| *cholesky* | 15.29 | 16.53 | 16.29 | 14.54 | 4.69 | 10.28 | 22.28 | 47.61 |
| *fft* | 10.65 | 11.21 | 11.20 | 11.20 | 2.98 | 6.31 | 11.20 | 17.74 |
| *fmm* | 8.82 | 9.22 | 9.29 | 9.30 | 3.55 | 7.38 | 14.59 | 27.63 |
| *lu* | 11.86 | 12.46 | 12.47 | 12.47 | 6.84 | 13.56 | 23.64 | 37.36 |
| *radiosity* | 12.11 | 12.90 | 13.00 | 12.45 | 7.71 | 16.04 | 30.32 | 55.95 |
| *radix* | 13.42 | 14.47 | 14.45 | 14.52 | 2.94 | 5.39 | 8.75 | 10.92 |
| *raytrace* | 15.18 | 16.02 | 16.01 | 16.01 | 7.60 | 18.67 | 34.51 | 57.56 |
| *water-ns* | 10.65 | 11.14 | 11.14 | 11.15 | 7.46 | 16.24 | 33.84 | 59.02 |
| *water-sp* | 11.38 | 11.90 | 11.90 | 11.90 | 9.33 | 16.02 | 26.22 | 41.77 |
| *Total* | 12.34 | 13.06 | 13.13 | 12.98 | 5.59 | 11.34 | 20.50 | 33.79 |

### 7.2.2 *tmlvCFiat*

The effectiveness of *tmlvCFiat* directly depends on the first-access flag miss rate – lower values result in fewer trace messages needing to be streamed out through the trace port. Figure 7.5 shows the total first-access miss rate as a function of the number of cores for three data cache configurations (*Small*, *Medium*, and *Large*). The total first-access miss rate is calculated as the total number of first-access misses when all benchmarks are considered together divided by the total number of data reads. The first-access miss rate decreases with an increase in the number of cores, e.g., from more than 3.09% when N = 1 to 2.81% when N = 8 for the *Medium* configuration. As expected, larger data caches result in a smaller number of miss events and thus a smaller number of first-access miss events. e.g., the first-

access miss rate ranges from 4.74 % for the Small configuration to 1.73 % for the *Large* configuration when the number of cores is set to four (N = 8). Figure 7.5 also indicates the minimum and the maximum first-access miss rates. Thus, the first-access miss rate reaches as high as ~14.26% for *fft* and as low as 0.48% for *water-spa*, depending on the number of cores and the data cache size. These results confirm that *tmlvCFiat* indeed can reduce the number of trace messages.

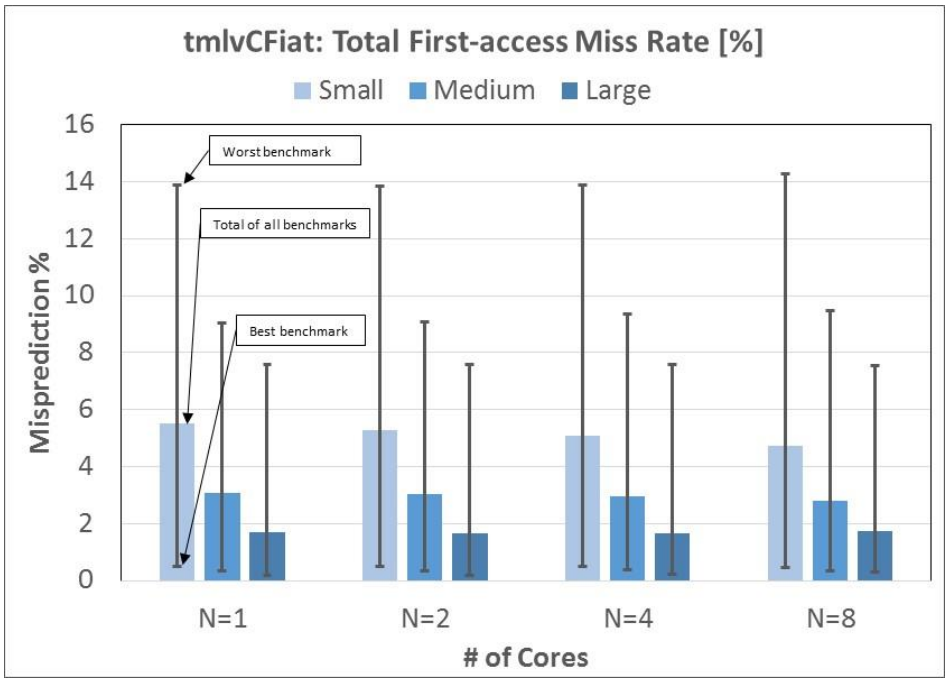


Figure 7.5 First Access Miss Rate for Splash2 benchmark

Figure 7.6 shows the total average trace port bandwidth for Nexus–like memory read flow traces (*tmlvNX_b*), *tmlvCFiat* (*tmlvCF_b*, *tmlvCF_e*) as function of the number of threads (N = 1, 2, 4, and 8 ) and the *tmlvCFiat* configuration (Small, *Medium* and *Large*). Table 7.6 shows the trace port bandwidth for *Large* configuration. For N = 1, *tmlvNX_b* requires on average 12.34 bpi when N = 1 and ranges be-

tween 8.8 bpi (*fmm*) and 15.29 bpi (*cholesky*); for N = 8, *tmlvNX_b* requires 12.98 bpi ranges between 9.29 (*fmm*) and 16.01 (*cholesky*).

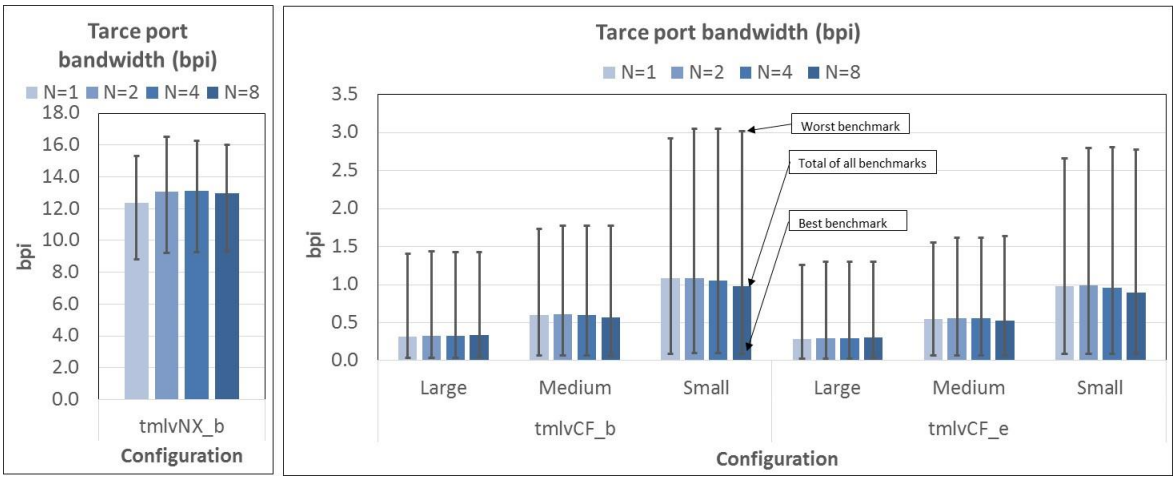


Figure 7.6 Trace port bandwidth bpi for timed load data value trace

*tmlvCF_b* dramatically reduces the average trace port bandwidth as followa:

- *Small* configuration: 1.08 bpi (N = 1) and 0.98 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to NX_b 11.41 times for N = 1 and 13.24 times for N = 8.
- *Medium* configuration: 0.59 bpi (N = 1) and 0.57 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to NX_b 20.6 times for N = 1 and 22.64 times for N = 8.
- *Large* configuration: 0.317 bpi (N = 1) and 0.339 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to NX_b 38.9 times for N = 1 and 38.25 times for N = 8.

Table 7.6 Trace port bandwidth for *tmlvNX_b* and *tmlvCFiat*

| *Thread* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Mechanism* | *tmlvNX_b* | *tmlvCF_b* | *tmlvCF_e* | *tmlvNX_b* | *tmlvCF_b* | *tmlvCF_e* | *tmlvNX_b* | *tmlvCF_b* | *tmlvCF_e* | *tmlvNX_b* | *tmlvCF_b* | *tmlvCF_e* |
| *barnes* | 15.041 | 0.414 | 0.382 | 15.888 | 0.414 | 0.383 | 15.867 | 0.462 | 0.429 | 15.810 | 0.837 | 0.775 |
| *cholesky* | 15.292 | 0.951 | 0.861 | 16.534 | 0.897 | 0.813 | 16.294 | 0.691 | 0.626 | 14.537 | 0.430 | 0.388 |
| *fft* | 10.651 | 1.401 | 1.260 | 11.209 | 1.433 | 1.303 | 11.201 | 1.426 | 1.300 | 11.196 | 1.423 | 1.296 |
| *fmm* | 8.821 | 0.200 | 0.183 | 9.219 | 0.209 | 0.192 | 9.285 | 0.207 | 0.192 | 9.299 | 0.205 | 0.190 |
| *lu* | 11.858 | 0.488 | 0.492 | 12.464 | 0.323 | 0.319 | 12.466 | 0.329 | 0.326 | 12.474 | 0.253 | 0.244 |
| *radiosity* | 12.108 | 0.062 | 0.053 | 12.902 | 0.070 | 0.062 | 12.999 | 0.076 | 0.066 | 12.450 | 0.072 | 0.064 |
| *radix* | 13.424 | 0.723 | 0.636 | 14.467 | 0.759 | 0.675 | 14.454 | 0.758 | 0.676 | 14.519 | 0.767 | 0.683 |
| *raytrace* | 15.176 | 0.223 | 0.199 | 16.025 | 0.234 | 0.211 | 16.010 | 0.234 | 0.211 | 16.012 | 0.234 | 0.210 |
| *water-ns* | 10.646 | 0.033 | 0.031 | 11.137 | 0.033 | 0.031 | 11.139 | 0.033 | 0.031 | 11.150 | 0.044 | 0.041 |
| *water-sp* | 11.379 | 0.040 | 0.039 | 11.900 | 0.046 | 0.044 | 11.899 | 0.047 | 0.045 | 11.899 | 0.046 | 0.044 |
| *Total* | *12.342* | *0.317* | *0.287* | *13.065* | *0.321* | *0.293* | *13.127* | *0.321* | *0.293* | *12.982* | *0.339* | *0.310* |

*tmlvCF_e* further reduces the average trace port bandwidth as follows:

- *Small* configuration: 0.977 bpi (N = 1) and 0.893 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *tmlvNX_b* 1.10 times for N = 1 and 1.09 times for N = 8.
- *Medium* configuration: 0.54 bpi (N = 1) and 0.52 bpi when N = 8. This is equivalent to reducing the trace port bandwidth relative to *tmlvNX_b* 1.09 times for N = 1 and 1.09 times for N = 8.
- *Large* configuration: 0.287 bpi (N = 1) and 0.310 bpi when N = 4. This is equivalent to reducing the trace port bandwidth relative to *tmlvNX_b* 1.10 times for N = 1 and 1.09 times for N = 8.

Table 7.7 shows the compression ratio or speedup of *tmlvCF_e* relative to *tmlvNX_b*, calculated as TPB(*tmlvNX_b*)/TPB(*tmlvCF_e*) as a function of the number of threads (N = 1, 2, 4 and 8) and configuration (*Small*, *Medium*, *Large*). For the *Small* configuration, the average compression ratio is 12.63 for N = 1 and 14.54 for N = 8. For the *Medium* configuration, the average compression ratio is 22.65 for N = 1 and 24.75 for N = 8. For the *Large* configuration, the average compression ratio

42.98 for N = 1 and 41.95 for N = 8. The best performing is *water-spatial* (N = 1) and the worst performing is *fft* (N = 8).

Table 7.7 Compression ratio of *tmlvCF_e* ralative to *tmlvNX_b*

| *cores* | *N=1* | | | *N=2* | | | *N=4* | | | *N=8* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Config* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* | *S* | *M* | *L* |
| *barnes* | 5.65 | 10.68 | 39.40 | 5.68 | 10.55 | 41.45 | 5.65 | 10.20 | 37.01 | 5.70 | 9.64 | 20.41 |
| *cholesky* | 5.75 | 11.67 | 17.77 | 7.92 | 14.61 | 20.33 | 11.07 | 19.37 | 26.05 | 17.17 | 28.93 | 37.46 |
| *fft* | 4.63 | 6.84 | 8.46 | 4.72 | 6.95 | 8.60 | 4.70 | 6.95 | 8.61 | 4.71 | 6.97 | 8.64 |
| *fmm* | 16.95 | 28.95 | 48.12 | 17.16 | 29.08 | 47.99 | 17.60 | 29.57 | 48.46 | 18.04 | 30.15 | 48.99 |
| *lu* | 18.42 | 20.62 | 24.12 | 18.57 | 22.37 | 39.07 | 18.60 | 22.37 | 38.24 | 19.17 | 32.26 | 51.06 |
| *radiosity* | 35.50 | 100.15 | 227.17 | 35.12 | 86.42 | 209.79 | 34.71 | 99.53 | 195.76 | 34.58 | 105.87 | 196.06 |
| *radix* | 16.36 | 18.44 | 21.10 | 16.51 | 18.68 | 21.44 | 16.40 | 18.57 | 21.38 | 16.17 | 18.35 | 21.26 |
| *raytrace* | 11.02 | 27.11 | 76.15 | 10.93 | 26.84 | 76.09 | 10.82 | 26.31 | 76.02 | 10.68 | 26.16 | 76.21 |
| *water-ns* | 21.83 | 22.48 | 345.65 | 21.79 | 22.66 | 359.25 | 21.65 | 23.07 | 364.00 | 21.49 | 22.65 | 270.64 |
| *water-sp* | 126.85 | 175.60 | 295.55 | 130.05 | 173.47 | 271.69 | 128.78 | 171.71 | 265.02 | 132.21 | 174.22 | 269.82 |
| *Total* | 12.63 | 22.65 | 42.98 | 13.25 | 23.21 | 44.65 | 13.70 | 23.71 | 44.80 | 14.54 | 24.75 | 41.95 |

Figure 7.7 shows the total trace port bandwidth in bits per clock cycle. *tmlvCF_e* and *tmlvCF_b* are highly effective in reducing the trace port bandwidth. When N = 8, the total required bandwidth for *tmlvCF_e* is just 2.32 bpc compared to 33.78 for *tmlvNX_b* for the *Small* configuration. Our variable encoding scheme in *tmlvCF_e* offers improvement in the range of 9 % when compared to fixed encoding *tmlvCF_b* for the *Large* configuration.

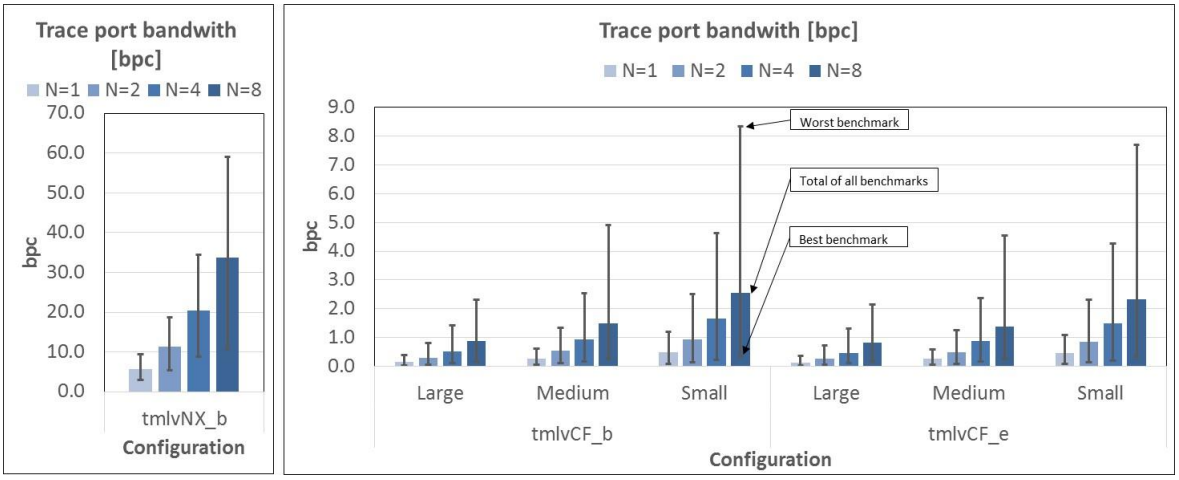


Figure 7.7 Trace port bandwidth in bpc for timed load data value trace

# CHAPTER 8

# CONCLUSIONS

Knowledge, Action and Devotion are complementary to each other
--Rev. Pandurang Shastri Athavale

Modern embedded and cyber-physical computer systems are shaped by their increasing sophistication and complexity, diversification, proliferation, as well as ever-tightening time-to-market. A growing number of such systems relies on multi-core systems-on-a-chip. More complex software stacks running on more sophisticated and complex hardware platforms place additional burdens on software developers who spend a significant portion of development time on software debugging and testing. Developing hardware/software techniques to help developers locate and correct software bugs faster is critical in meeting time-to-market deadlines, reducing system cost, and improving system reliability by providing well-tested bug-free software. Capturing all events of interest for debugging in hardware and streaming them out of the chip is cost prohibitive because a huge amount of debug and trace data generated on systems with multiple processor cores running at clock frequencies in GHz.

This research focuses on developing and evaluating hardware techniques for unobtrusive capturing and filtering of control-flow and load data value traces in real-time for multicore systems. These traces coupled with sophisticated software debuggers enable a faithful reconstruction of events from the target platform in the software debugger, thus helping software developers locate and correct bugs faster.

The thesis introduced *mcfTRaptor* and *mlvCFiat* hardware structures to capture and compress control-flow and load data value traces, respectively. To dramatically reduce the number of trace messages that needs to be streamed out, these structures are modelled in a software debugger as well and work in sync with corresponding hardware structures. This way, trace messages are generated only in the case of rare miss events on predictor or cache structures on the target platforms – events that cannot be properly inferred by the software debugger.

Our experimental evaluation demonstrated the effectiveness of the proposed techniques by evaluating trace port bandwidth requirements on a selected set of benchmark programs. The experimental evaluation is based on functional traces that assume that events are ordered on the target platform and timed traces that include explicit time stamps. The evaluation is performed while varying the number of processor cores, size and organization of tracing structures, and encoding schemes. We evaluate several encoding schemes and find those that minimize the number of bits that needs to be streamed out of the chip.

The results of our experimental evaluation show that *mcfTRaptor* dramatically reduces the trace port bandwidth when compared to the current state-of-the-art. The total improvements are between ~12 times for the *Small* configuration and over ~30 times for the *Large* configuration. The proposed method is robust, reducing the trace port bandwidths regardless of the number of processors on a chip. Similarly, *mlvCFiat* proves to be very effective in reducing the trace port bandwidth for load data value traces when compared to the state-of-the-art. It reduces the bandwidth in the range of 3.9 – 4.6 times for relatively the *Small* configuration and 6.7 to 7.4 times for the *Large* configuration.

The research tools designed in this thesis can be used to support future research in the area of on-the-fly tracing in multicores. The proposed techniques *mcfTRaptor* and *mlvCFiat* may utilize shared resources (predictors and caches). Another promising research area is to expand *mlvCFiat* technique to utilize cache coherence protocols and thus further reduce the number of trace messages that need to be reported.

# REFERENCES


[1] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," 2002. [Online]. Available: http://www.rti.org/pubs/software_testing.pdf.

[2] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface V 3.01," *Nexus 5001 Forum*, 2012. [Online]. Available: http://www.nexus5001.org/standard. [Accessed: 27-Sep-2014].

[3] W. Orme, "Debug and Trace for Multicore SoCs." White paper, ARM, Sep-2008.

[4] V. Uzelac, A. Milenković, M. Milenković, and M. Burtscher, "Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 1008–1020, Apr. 2014.

[5] V. Uzelac and A. Milenković, "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 1–18, May 2013.

[6] A. Myers, "A Binary Instrumentation Tool Suite for Capturing and Compressing Traces for Multithreaded Software," Thesis, University of Alabama in Huntsville, Huntsville, AL, USA, 2014.

[7] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, USA, 2012, pp. 335–344.

[8] A. Milenkovic and M. Milenkovic, "Exploiting Streams in Instruction and Data Address Trace Compression," in *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, 2003, pp. 99–107.

[9] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *IEEE Computer Architecture Letter*, vol. 2, no. 1, pp. 9–12, 2003.

[10] IEEE, "IEEE Standard 1149.1-2013 for Test Access Port and Boundary-Scan Architecture," *IEEE standards Association*, May-2013. [Online]. Available: http://standards.ieee.org/findstds/standard/1149.1-2013.html. [Accessed: 27-Sep-2014].

[11] MIPS, "MIPS PDtrace Specification Rev 7.50." MIPS Technologies Inc., CA, Dec-2012.

[12] A. Mayer, H. Siebert, and C. Lipsky, "MCDS - Multi-Core Debug Solution." White paper, IPextreme, May-2007.

[13] N. Stollon and R. Collins, "Nexus Based Multi-Core Debug," in *Proceedings of the Design Conference International Engineering Consortium*, Santa Clara, CA, USA, 2006, vol. 1, pp. 805–822.

[14] K.-U. Irrgang and R. G. Spallek, "Comparison of Trace-Port-Designs for On-Chip-Instruction-Trace," in *IEEE Germany Student Conference, University of Passau*, 2012.

[15] V. Uzelac and A. Milenkovic, "A Real-Time Program Trace Compressor Utilizing Double Move-To-Front Method," in *Proceedings of the 46th Annual Design Automation Conference (DAC'09), July 26-31*, San Francisco, CA, USA, 2009, pp. 738–743.

[16] B. Mihajlović and Ž. Žilić, "Real-time address trace compression for emulated and real system-on-chip processor core debugging," in *Proceedings of the 21st Edition on Great lakes symposium on VLSI*, Lausanne, Switzerland, 2011, pp. 331–336.

[17] C.-F. Kao, S.-M. Huang, and I.-J. Huang, "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Transactions on Circuits and Systems*, vol. 54, pp. 530–543, 2007.

[18] A. Milenkovic and M. Milenkovic, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 17, no. 1, pp. 1–27, 2007.

[19] M. Milenkovic, A. Milenkovic, and M. Burtscher, "Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces," in *Proceedings of the 2007 Data Compression Conference (DCC'07), Mar 27-29*, Snowbird, UT, 2007, pp. 55–65.

[20] V. Uzelac, A. Milenkovic, M. Milenkovic, and M. Burtscher, "Real-time, unobtrusive, and efficient program execution tracing with stream caches and last stream predictors," in *Proceedings of IEEE International Conference on Computer Design, 2009. ICCD 2009*, Lake Tahoe, CA, 2009, pp. 173–178.

[21] A. Milenkovic, V. Uzelac, M. Milenkovic, and M. Burtscher, "Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 992–1005, Jul. 2011.

[22] V. Uzelac, A. Milenković, M. Burtscher, and M. Milenković, "Real-time Unobtrusive Program Execution Trace Compression Using Branch Predictor Events," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES'10)*, Scottsdale, AZ, 2010, pp. 97–106.

[23] A. Milenkovic, M. Milenkovic, and J. Kulick, "N-Tuple Compression: A Novel Method for Compression of Branch Instruction Traces," in *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, Reno, NV, 2003, pp. 49–55.

[24] V. Uzelac and A. Milenković, "Hardware-based data value and address trace filtering techniques," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded System (CASES'10)*, Scottsdale, AZ, USA, 2010, pp. 117–126.

[25] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 174–184, Feb. 2006.

[26] K. Driesen and U. Hölze, "Accurate indirect branch prediction," in *SIGARCH Computer Architecture News*, 1998, vol. 26, pp. 167–178.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005, pp. 190 – 200.

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

[29] V. Uzelac, "Microbenchmarks and mechanisms for reverse engineering of modern branch predictor units," Thesis, University of Alabama in Huntsville, Huntsville, AL, USA, 2008.

[30] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, April 2009*, Boston, MA, USA, 2009, pp. 207–217.

[31] "SPLASH-2, 32 bit binary archive," *Multi2Sim benchmarks*. [Online]. Available: https://www.multi2sim.org/benchmarks/splash2.php. [Accessed: 07-Mar-2015].

[32] "Multi2Sim," *Multi2Sim, A Heterogeneous System Simulator*. [Online]. Available: https://www.multi2sim.org/. [Accessed: 07-Mar-2015].