TECHNIQUES FOR CAPTURING AND FILTERING DATA VALUE TRACES IN MULTICORES

by

MOUNIKA PONUGOTI

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

2016

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 Mounika Ponugoti 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

(Dr. Aleksandar Milenkovic) (date)

(Dr. Rhonda Gaede)

(Dr. Earl Wells)

Department Chair

(Dr. Ravi Gorur) (date)

College Dean

(Dr. Shankar Mahalingam) (date)

Graduate Dean

(Dr. David Berkowitz) (date)

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 Mounika Ponugoti
Title Techniques for Capturing and Filtering Data Value Traces in Multicores

Software testing and debugging represent critical aspects of the design of modern multicore-based embedded computer systems due to growing hardware and software complexity, increased integration, and tightening time-to-market. The existing tracing and debugging techniques offer limited visibility of the system under test or rely on large on-chip buffers and wide trace ports that increase the system cost. This thesis introduces three hardware/software techniques for capturing and filtering load data value traces in multicores. They track memory read accesses in data caches on the target platform and simulate their behavior in the software debugger to significantly reduce the number of trace events that need to be streamed out of the target platform. Our experimental evaluation explores the effectiveness of the proposed techniques by measuring the trace port bandwidth as a function of system parameters. The results show that the proposed techniques significantly reduce the total trace port bandwidth. The improvements relative to the existing Nexus-like load data value tracing range from 10 to 60 times for a single core and from 19 to 74 times for an octa core.

Abstract Approval: Committee Chair _____

Department Chair _____

Graduate Dean _____

ACKNOWLEDGMENTS

The work presented in this thesis would have been incomplete without thanking people who helped me directly and indirectly. First, I would like to express my sincere gratitude to my advisor Dr. Aleksandar Milenkovic for his unlimited support at every stage of this work, and for supporting me as a graduate research assistant. He inspired me personally and professionally with his patience and his interest towards student learning. This work was supported in part by US National Science Foundation (NSF) grant CNS-1217470.

I would like to thank Mr. Amrish K. Tewar who developed TmTrace module that I used in my research. I would like to thank Mr. Tewar and Mr. Armen Dzhagaryan for helping me to get started in the laboratory.

I would like to thank Dr. Rhonda Gaede and Dr. Earl Wells for serving on my committee. I would also like to thank all the professors and staff members who helped me during my time at the University of Alabama in Huntsville.

Also, I would like to thank Srinivas R. Mynampally and his family for giving me great support since the first day of my arrival to the United States.

Finally, I would like to express my deepest gratitude to my parents, Bhagavanth Rao and Vimala, for their unconditional love and support. I would like to thank my life partner, Vamshi Krishna, for providing continuous support and encouragement for higher studies.

TABLE OF CONTENTS

Contents Page
LIST OF FIGURES....ix
LIST OF TABLES....xii

CHAPTER 1 1
1.1 Background and Motivation 1
1.2 Scope of this Thesis 2
1.3 Contributions 4
1.4 Outline 5

CHAPTER 2 6
2.1 Tracing in Embedded Multicores 6
2.2 Memory Data Traces 9
2.3 Related Work 11

CHAPTER 3 15
3.1 mlvCFiat 15
3.2 mc²RT 22
3.3 mc²RFiat 33

CHAPTER 4 45
4.1 Software Timed Trace Generator 46
4.2 *mlvCFiat* Simulator 47
4.2.1 Implementation Details 51
4.2.2 Verification Details 54
4.3 *mc²RT* Simulator 61
4.3.1 Implementation Details 63
4.3.2 Verification Details 65
4.4 mc²RFiat Simulator 73

4.4.1 Implementation Details 73
4.4.2 Verification Details 75
4.5 Software to Hardware Trace Translation 84
4.6 Experimental Environment 88
4.6.1 Experimental Setup 88
4.6.2 Benchmarks 90
4.6.3 Experiments 92
4.6.4 Granularity Study 93
4.6.5 Variable Encoding 95
CHAPTER 5 99
5.1 Trace Port Bandwidth for Load Data Value Traces 99
5.1.1 NX_b 99
5.1.2 mlvCFiat 104
5.1.3 mc²RT 111
5.1.4 *mc²RFiat* 117
5.2 Dynamic Trace Port Bandwidth Analysis for Load Data Value Traces 124
5.3 Putting It All Together 126
CHAPTER 6 130
REFERENCES 133

LIST OF FIGURES

Figure Page

Figure 2.1 Debugging and tracing in multicores: a detailed view 9

Figure 2.2. Memory read trace: an example a) C program b) equivalent x86 assembly c) memory read flow traces 11

Figure 3.1 A system view of *mlvCFiat* 16

Figure 3.2. *mlvCFiat* structures for core *i* 18

Figure 3.3 *mlvCFiat* operations on the target core *i* for a) memory reads, b) memory writes, and c) external invalidations 20

Figure 3.4 *mlvCFiat* operations in the software debugger on core *i* for a) memory reads, b) memory writes, and c) external invalidations 22

Figure 3.5 A system view of *mc²RT* 23

Figure 3.6 *mc²RT* structures for core *i* 25

Figure 3.7 *mc²RT* operation on the target core *i* for memory reads 27

Figure 3.8 Coherent Read Transaction in *mc²RT* on the target core *i* 28

Figure 3.9 *mc²RT* operation on the target core *i* for memory writes 30

Figure 3.10 Coherent Read and Invalidate in *mc²RT* on target core *i* 31

Figure 3.11 Coherent Invalidate in *mc²RT* on target core *i* 31

Figure 3.12 *mc²RT* operation in the software debugger on core *i* for a) memory reads b) memory writes 33

Figure 3.13 A system view of *mc²RFiat* 35

Figure 3.14 *mc²RFiat* structures for core *i* 36

Figure 3.15 *mc²RFiat* operation on the target core *i* for memory reads 38

Figure 3.16 Coherent Read Transaction in *mc²RFiat* on the target core *i* 39

Figure 3.17 *mc²RFiat* operation on the target core *i* for memory writes 41

Figure 3.18 Coherent Read and Invalidate in *mc²RFiat* on the target core *i* 42

Figure 3.19 Coherent Invalidate in *mc²RFiat* on the target core *i* 42

Figure 3.20 *mc²RFiat* operation in the software debugger on core *i* for a) memory reads, and b) memory writes 44

Figure 4.1 Experiment flow to create hardware traces 46

Figure 4.2 Trace messages generated for memory reads and writes 47

Figure 4.3 *mlvCFiat* trace descriptor format 49

Figure 4.4 *mlvCFiat* simulator statistics example 50

Figure 4.5 *mlvCFiat* simulator functional flow 53

Figure 4.6 Testing *mlvCFiat*: single cache block access 57

Figure 4.7 Testing *mlvCFiat*: multi cache block access 61

Figure 4.8 *mc²RT* trace descriptor format 62

Figure 4.9 *mc²RT* simulator statistics example 63

Figure 4.10 Testing *mc²RT*: single cache block access 69

Figure 4.11 Testing *mc²RT*: multi cache block access 72

Figure 4.12 Testing *mc²RFiat*: single cache block access 80

Figure 4.13 Testing *mc²RFiat*: multi-cache block access 84

Figure 4.14 Formats of trace messages for *NX_b*, *CF_b*, CF*_e, RT_b, RT_e, RF_b and RF_e* 87

Figure 4.15 Multicore model in Mult2Sim 89

Figure 4.16 Normalized trace port bandwidth as a function of first-access granularity for *mlvCFiat* 94

Figure 4.17 Normalized trace port bandwidth as a function of first-access granularity for *mc²RFiat* 95

Figure 4.18 CDF of the minimum length for *fahCnt* for *mlvCFiat* 96

Figure 4.19 CDF of the minimum length for *dCC* for *mlvCFiat* 96

Figure 4.20 Average *fahCnt* and *dCC* fields as a function of chunk sizes for *mlvCFiat* 97

Figure 5.1 Breakdown of trace port bandwidth for *NX_b* for Splash2 benchmarks 102

Figure 5.2 First Access Miss Rate for *mlvCFiat* for Splash2 benchmarks ... 106

Figure 5.3 Total average trace port bandwidth in bpi for *CF_b* and *CF_e* .. 107

Figure 5.4 Total average trace port bandwidth in bpc for *CF_b* and *CF_e* . 111

Figure 5.5 Trace Miss Rate for *mc²RT* for Splash2 benchmarks 112

Figure 5.6 Total average trace port bandwidth in bpi for *RT_b* and *RT_e* . 113

Figure 5.7 Total average trace port bandwidth in bpc for *RT_b* and *RT_e* . 117

Figure 5.8 First Access Miss Rate for *mc²RT* for Splash2 benchmarks 119

Figure 5.9 Total average trace port bandwidth in bpi for *RF_b* and *RF*_e . 120

Figure 5.10 Total average trace port bandwidth in bpc for *RF_b* and *RF_e* 124

Figure 5.11 Dynamic trace port bandwidth in bpc during execution of *raytrace* for *N*=8 125

Figure 5.12 Dynamic trace port bandwidth in bpc during execution of *water-ns* for *N*=8 126

Figure 5.13 Trace port bandwidth in bpi for *CS64* configuration 127

Figure 5.14 Trace port bandwidth in bpc for *CS64* configuration 128

LIST OF TABLES

Table Page

Table 4.1 *mlvCFiat* flags 49

Table 4.2 Splash2 benchmark suite characterization 91

Table 4.3 Characterization of memory reads in Splash2 92

Table 4.4 Experiments conducted 93

Table 4.5 Summary of variable encoding parameters for different fields 98

Table 5.1 Trace port bandwidth for *NX_b* for Splash2 benchmarks 101

Table 5.2 Compression ratios achieved by gzip 104

Table 5.3 Trace port bandwidth bpi for *CF_b* 108

Table 5.4 Trace port bandwidth bpi for *CF_e* 109

Table 5.5 Compression ratio of *CF_e* relative to *NX_b* 110

Table 5.6 Trace port bandwidth bpi for *RT_b* 114

Table 5.7 Trace port bandwidth bpi for *RT_e* 115

Table 5.8 Compression ratio of *RT_e* relative to *NX_b* 116

Table 5.9 Trace port bandwidth bpi for *RF_b* 121

Table 5.10 Trace port bandwidth bpi for *RF_e* 122

Table 5.11 Compression ratio of *RF_e* relative to *NX_b* 123

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Growing complexity and sophistication of modern embedded systems and the shift toward multicores make software testing and debugging one of the most critical aspects of system development. Faster and cheaper processors with an increased level of integration have enabled new applications that were impossible just a decade ago. Users' expectations and their reliance on embedded systems have also gone up. As a result, the complexity of the software stack in embedded systems keeps growing and the time-to-market is decreasing. A recent report from the International Technology Roadmap for Semiconductors found that the software engineering and tool costs account for 80% or more of the total development cost of modern high-end embedded systems [1].

It is important to give software developers tools to quickly locate and correct all software bugs with minimum effort. When debugging, software developers often need perfect visibility of the system under test. However, achieving this visibility is not feasible due to high system complexity, limited available bandwidth for debugging data, and high operating frequencies. Traditional debugging techniques rely on single stepping, setting breakpoints, and examining the content of registers and memory locations while the processor is halted. This approach is effort- and time-consuming for software developers. In addition, it perturbs the sequence of events on

target platforms and thus is not practical in real-time cyber-physical systems. Finally, it does not scale well to multicores.

To address these challenges, modern embedded processors include on-chip resources dedicated for tracing and debugging. The trace module on the target platform collects the traces for the program of interest. With certain conditions, collected traces can be used to replay the program offline. State-of-the-art trace modules require 1 to 4 bits per executed instruction per core for control-flow traces and 8 to16 bits per executed instruction per core for data flow traces. Thus, a 1 KB on-chip trace buffer per processor core may capture control flow traces for program segments on the order of 2000 – 8000 instructions and load data value traces for program segments on the order of 500 – 1000 instructions. It should be noted that limited program traces are not sufficient in locating software bugs in modern processors because there could be millions of instructions between the origin of the bug and its manifestation. To capture traces, especially Nexus-like load data value traces for the entire program, we require deep trace buffers and wide trace ports. However, hardware vendors have little incentive to spend a lot of on-chip resources, thus increasing the system cost, just to help software developers find more bugs faster.

1.2 Scope of this Thesis

Capturing load data value traces on-the-fly is important in debugging multicore embedded systems. Load data value traces are created by recording values read from memory and I/O devices. These traces captured on the target platform and streamed out to a software debugger are necessary under certain conditions to deterministically replay programs offline.

In this thesis, trace port bandwidth requirements for Nexus-like load data value traces are analyzed to illustrate challenges in capturing data traces in multicores when running a set of parallel programs and using commercial state-of-the-art tracing techniques. The trace port bandwidth is measured in bits per instruction executed (bpi) and bits per processor clock cycle (bpc). The total average trace port bandwidth (bpi) for the entire Splash2 benchmark suite ranges from 12.34 when N=1 to 13.17 when N=8 where N is the number of cores. For a given multicore model the trace port bandwidth (bpc) ranges from 4.92 when N=1 to 25.64 when N=8. Several benchmarks require an average trace port bandwidth of over 40 bits per clock cycle. These results illustrate the challenges of capturing and streaming out load data value trace. Having trace ports with over 40 pins that work at processor clock speed or megabytes of on-chip trace buffers is not practical.

To address data tracing challenges, we introduce three new techniques for unobtrusive capturing and filtering of load data value traces in real-time in multicore platforms. These techniques are named (i) *mlvCFiat* – multicore load value cache first access tracking, (ii) *mc²RT* – multicore cache-coherent read tracking, and (iii) *mc²RFiat* – multicore cache-coherent read with first access tracking. They rely on on-chip tracking of memory reads in data caches on the target platform and equivalent changes in the software debugger. The software debugger simulates the behavior of data caches and trace modules on the target platform during program replay, thus reducing the number of trace messages that need to be emitted by the target platform. This way we reduce the trace port bandwidth requirements and requirements for on-chip trace buffers. The three techniques differ in the number of tracking bits per data cache block and the level of support for cache coherence proto-

col. Using an execution-driven cycle-accurate simulator and the Splash2 parallel programs, we evaluate the effectiveness of the proposed techniques as a function of system parameters, such as data cache size, the number of cores (*N*), and encoding mechanism.

The total average trace port bandwidth using the *mlvCFiat* technique with *CS64* configuration ranges from 0.21 bpi when *N*=1 to 0.53 bpi when *N*=8. *mlvCFiat* reduces the total average trace port bandwidth relative to Nexus-like load data value traces by 16.1 to 59.6 times when *N*=1 and 15.0 to 24.8 times when *N*=8. The total average trace port bandwidth using the *mc²RT* technique with *CS64* configuration ranges from 0.26 bpi when *N*=1 to 0.21 bpi when *N*=8. *mc²RT* reduces the total average trace port bandwidth relative to Nexus-like load data value traces from 10.1 to 47.3 times when *N*=1 and from 18.9 to 62.5 times when *N*=8. The total average trace port bandwidth using the *mc²RFiat* technique with *CS64* configuration ranges from 0.21 bpi when *N*=1 to 0.18 bpi when *N*=8. *mc²RFiat* reduces the total average trace port bandwidth relative to Nexus-like load data value traces by 16.1 to 59.6 times when *N*=1 and 27.6 to 73.8 times when *N*=8.

1.3 Contributions

The main contributions of this work are as follows.

- Characterization of trace port bandwidth requirements in multicores for Nexus-like time stamped load data value traces as a function of the number of cores. Both bits per instruction and bits per clock cycle as measures of the required trace port bandwidth are considered.

- Introduction of hardware/software techniques *mlvCFiat*, *mc²RT*, and *mc²RFiat* that capture and compress load data value traces in multicores.
- A detailed experimental evaluation of the trace port bandwidth required by the proposed techniques, while varying the number of cores, cache sizes, and encoding approaches.
- An analysis of dynamic trace port bandwidth during benchmarks' execution.

1.4 Outline

The rest of the thesis is organized as follows. CHAPTER 2 discusses the background, focusing on tracing and debugging in state-of-the-art embedded systems and especially on load data value traces. CHAPTER 3 describes the proposed techniques for capturing and compression of load data value traces in multicores. We introduce three techniques namely, *mlvCFiat*, *mc²RT*, and *mc²RFiat* that differ in their effectiveness and the level of hardware support. CHAPTER 4 describes the experimental evaluation of the proposed techniques and experimental environment used to create Nexus-like load data value traces and filtered traces. CHAPTER 5 discusses the results of the experimental evaluation. Finally, CHAPTER 6 gives concluding remarks.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter gives a more detailed view of debugging with tracing in embedded systems (Section 2.1). It also describes four classes of operations as defined in the Nexus 5001 standard. Section 2.2 describes load data value traces in multicores. Section 2.3 gives an overview of related work for this research.

2.1 Tracing in Embedded Multicores

Embedded processor and systems-on-a-chip often include on-chip trace modules that include resources to support tracing and debugging operations. The IEEE Nexus 5001 standard [2] defines functions and a general-purpose interface for software development and debugging of embedded processors. Nexus 5001 specifies 4 classes of debugging operations (Class 1 – Class 4). Higher classes support more complex debug operations but require more on-chip resources and wider trace ports, thus increasing the system cost.

Class 1 supports basic run control debugging, including setting breakpoints, single stepping, and changing the content of the memory or registers while the processor is halted. It is typically supported through the JTAG interface [3]. Class 2 adds support for capturing control-flow traces and streaming them out in near real time. Class 3 adds support for capturing and streaming out data flow traces (memory reads, memory writes, I/O reads, and I/O writes) in near real time. Finally, Class 4 adds support for emulating memory and I/O through the trace port.

Class 1 operations are routinely used to debug programs and widely supported in modern embedded platforms but are unsatisfying in many respects. First, it is more time-consuming and puts the burden on the software developer. Second, setting a breakpoint in real-time embedded systems and cyber-physical systems is not practical. Third, when the processor is halted for debugging, the actual order of events executed by the processor may change relative to the normal operation, which in turn may result in disappearing bugs. Finally, Class 1 operations do not scale well with multicores where setting breakpoints in one core may have an adverse impact on other processor cores.

To address these challenges, modern embedded processors rely on on-chip resources dedicated for tracing and debugging. Figure 2.1 shows a multicore system-on-a-chip (SoC) with trace and debug infrastructure. The multicore has *N* processor cores, a DSP core and a DMA core, all connected through a system interconnect. Each core has its own trace module which is responsible for capturing traces of interest. All the trace modules are connected to a debug and trace control unit through a trace and debug interconnect. Traces collected by the trace modules are temporarily stored in on-chip trace buffers before they are streamed out through a dedicated trace port to an external trace probe. The external trace probe connected to the host workstation may include Gigabytes of memory to store collected traces. These traces are then read by the software debugger running on a development workstation during a program replay, thus enabling a complete reconstruction of events that occurred on the target platform.

Control-flow traces enable the reconstruction of the program's flow only; for certain classes of hardware and software bugs control flow traces are sufficient, but

for some classes of bugs such as data race conditions, control-flow traces are not sufficient. Data-flow traces enable a complete replay of the executed program under certain conditions. To replay the program offline, the software debugger on the host workstation (Figure 2.1) relies on an instruction set simulator of the target platform, the program binary, initial conditions of the target platform and exception traces in addition to the data-flow traces. Replaying a program offline gives valuable information about shared memory access patterns, possible data race conditions, and better insights into the behavior of the target system.

Many chip vendors are including trace modules, examples include ARM's CoreSight [4], MIPS's PDTrace [5], Infineon's MCDS [6], and Freescale's MPC5500 [7]. State-of-the-art trace modules require trace port bandwidth in the range of 1 to 4 bits per executed instruction per core for control flow traces [4] and 8 to 12 bits per executed instruction per core for data flow traces [4]. Thus, a 1 KB on-chip trace buffer per processor core may capture control flow traces for program segments on the order of 2000 – 8000 instructions and data flow traces for program segments on the order of 500 – 1000 instructions. These limited traces are not enough to find the bugs because the origin of the bug and its manifestation are often millions of instructions apart. To capture traces, especially data-flow traces, for the entire program, deep trace buffers and wide trace ports are needed, significantly increasing the system complexity and cost. Hence, hardware vendors rarely support the higher classes of the Nexus 5001 standard. This problem is even worse in multicores where the number of I/O pins dedicated to the trace port cannot keep pace with the exponential growth in the number of cores on a single chip.

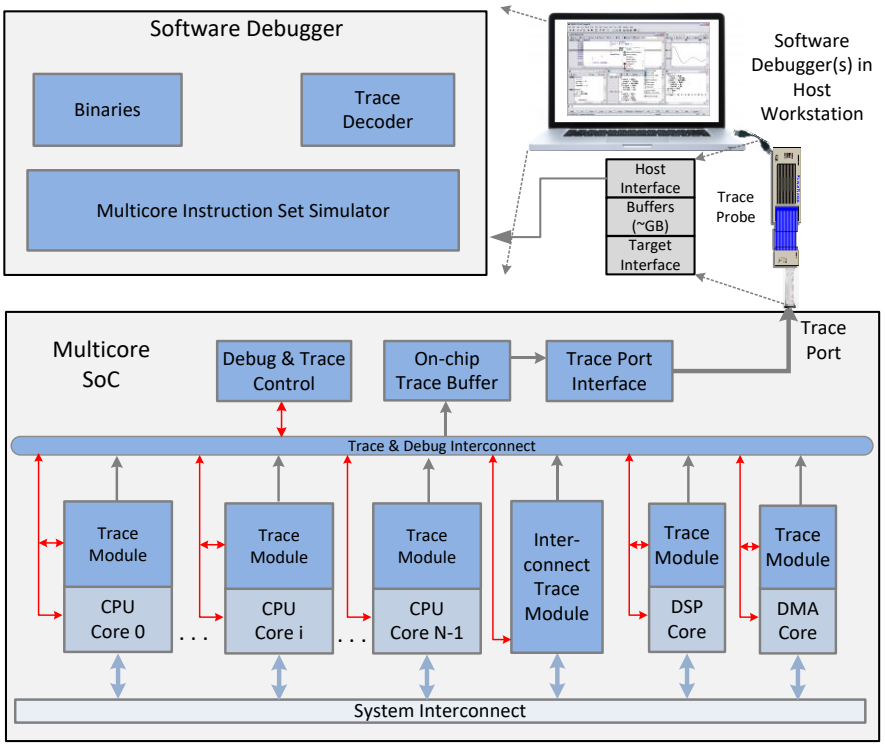

Figure 2.1 Debugging and tracing in multicores: a detailed view

2.2 Memory Data Traces

Data traces contain information about executed memory read and memory write instructions in the order in which they occurred during program execution. A typical data trace message may have relevant information, such as instruction address, type of memory operation (read or write), operand address, operand size, and operand value. In multicores, in addition to these fields, a trace message holds information about core/thread id and a global time stamp if the trace messages are not emitted in the order they occurred. The format of the trace messages may change depending on the context in which they are used and sometimes traces are needed only memory read or memory write.

Figure 2.2 shows an example of trace messages for memory reads of an OpenMP C program. The program scales each element of the input array and writes the result to the same location. The input array holds 20 single byte elements. All the operations executed by each thread are independent, thus each thread processes 5 elements. Figure 2.2b shows the assembly code of the parallel loop executed by each thread. Figure 2.2c shows trace messages capturing memory reads during program execution. Each message includes a global time stamp, thread id, operand address, operand size, and operand value.

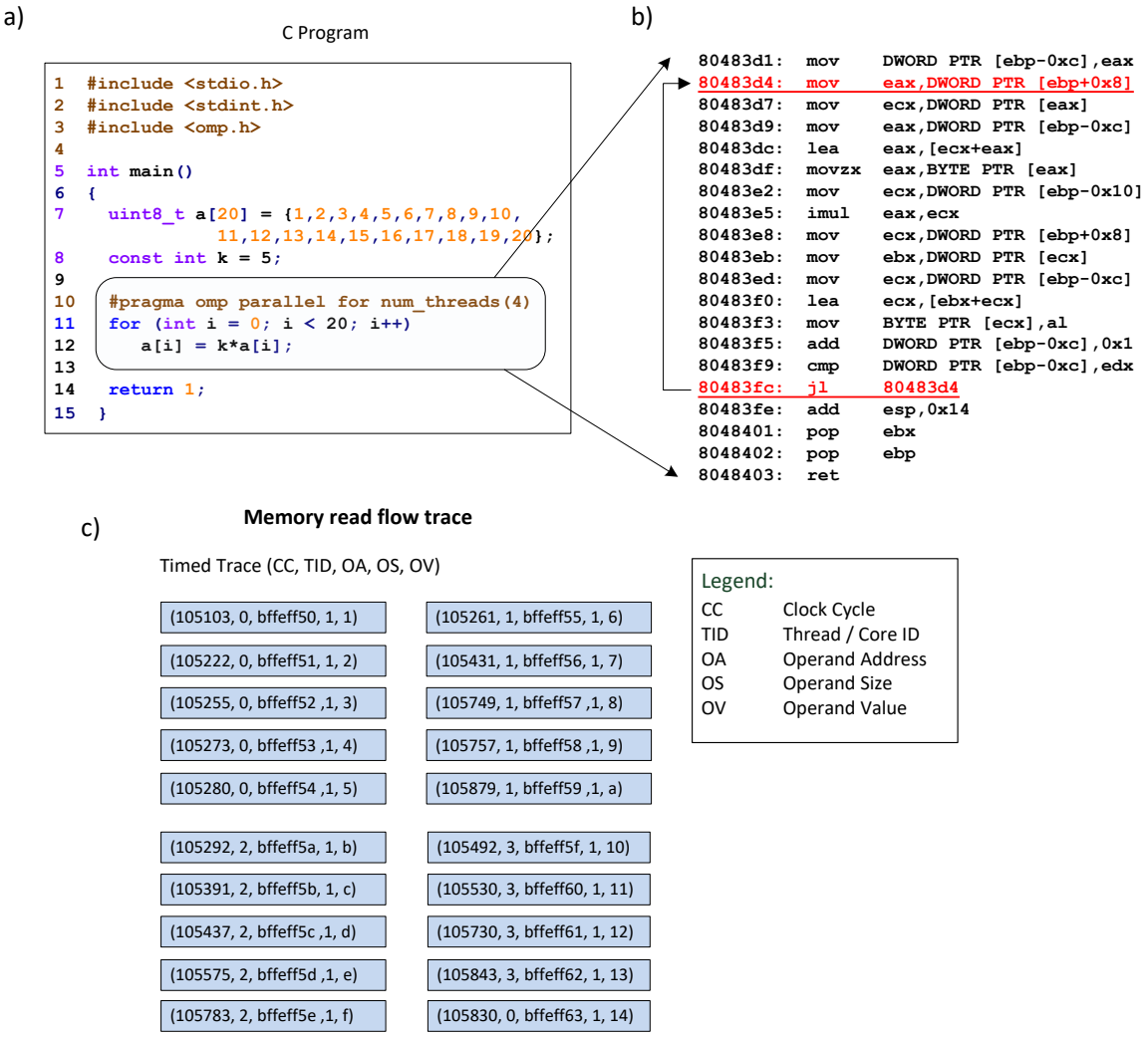

Figure 2.2. Memory read trace: an example a) C program b) equivalent x86 assembly c) memory read flow traces

2.3 Related Work

Modern embedded processors increasingly include on on-chip trace and debug infrastructure [4]–[9]. However, commercially available trace modules typically implement only rudimentary forms of hardware filtering with a relatively small com-

pression ratio. Irrgang and Spallek analyzed the Nexus and trace port configurations and their impact on achievable compression for instruction traces and found that 8-bit trace ports are sufficient [10].

Several recent research efforts in academia propose trace-specific compression techniques that achieve higher compression ratios. A class of these techniques is applicable only to software traces as they combine trace-specific and general-purpose compression algorithms and, in general, are not applicable to hardware tracing [11]–[19].

Another group of proposed techniques is applicable to hardware tracing. Several techniques rely on hardware implementations of general-purpose compressors [20], [21]. For example, Kao et al. [22] 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. A novel approach, the stream based compression algorithm [15], exploits inherent characteristics of program execution traces for compression. A double-move-to-front compressor introduced by Uzelac and Milenkovic [20] 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 [23]–[25] and branch predictors [14], [26], [27] 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. Uzelac et

al. [26] introduced *TRaptor* for control-flow traces that requires only 0.029 bits per instruction on the trace port (~34-fold improvement over the commercial state-of-the-art) per processor core at the hardware cost of approximately 5,000 gates. Tewar et. al. introduced *mcfTRaptor* that extends *TRaptor* to multicore platforms [28].

Whereas a number of studies in academia focus on capturing and compressing control-flow traces, relatively few studies look at on-the-fly data tracing. One interesting solution for debugging multicore SoCs called hidICE was proposed by Hochberger and Weiss [29]. It relies on a hardware emulator that replicates all master cores and memories from the target platform. The target platform reports only exceptions and data reads from peripherals that cannot be inferred by the emulator. However, hidICE is cost-prohibitive because it requires not only changes on the target platform to include a synchronization core and a new trace port, but also requires a sophisticated hardware emulator that replicates all the master modules and the RAM memory from the target platform. In addition, there has been no quantitative evaluation of hidICE.

This research relies on the prior work of Uzelac and Milenkovic. For load value traces, Uzelac and Milenkovic [30], [31] 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, this technique has been demonstrated on uniprocessors only. In addition, prior studies were based on functional simulation and did not address the challenges of producing ordered or time-stamped trace messages coming from multiple cores. To the best of our knowledge, there have been no academic studies focusing on the quantitative evaluation of data tracing requirements and development of cost-effective trace filtering techniques scalable to multicores.

The problem of tracing requirements in multicores running parallel programs requires additional study and answering the following questions. What is the required trace port bandwidth? How does trace port bandwidth scale up with multiple processor cores? How the existing techniques may be applied to multicores? These are some of the questions that are fully addressed in this thesis [32]–[34]. In this thesis, we want to explore requirements for real-time load data value tracing in multicores and introduce cost-effective solutions that scale well with a number of processor cores.

CHAPTER 3

NEW TECHNIQUES FOR DATA TRACING IN MULTICORES

This chapter discusses novel techniques for capturing and compressing load data value traces in multicores; *mlvCFiat*, which stands for ***m****ulticore* ***l****oad* ***v****alue* ***c****ache* ***fi****rst-****a****ccess* ***t****racking* (Section 3.1), *mc²RT*, which stands for ***m****ulticore* ***c****ache* ***c****oherent* ***r****ead* ***t****racking* (Section 3.2) and *mc²RFiat*, which stands for ***m****ulticore* ***c****ache* ***c****oherent* ***r****ead with* ***fi****rst-****a****ccess* ***t****racking* (Section 3.3). All three techniques are designed to reduce the pressure on the trace port and require relatively modest hardware support on the target platform and the software debugger capable of mirroring hardware events during program replay.

3.1 mlvCFiat

mlvCFiat is a hardware-based mechanism for a multicore processor that reduces the number of load data value traces by collecting a minimal set of trace messages with the help of a cache first access tracking mechanism [33]. *mlvCFiat* is an extension of the existing *CFiat* mechanism for capturing and filtering load data value traces in single-core processors [35].

Figure 3.1 shows the block diagram of a multicore system-on-chip (SoC) with the trace and debug infrastructure; light blue boxes represent additional *mlvCFiat* hardware and software modules. *mlvCFiat* requires hardware changes to the L1 data caches to capture and compress load data value traces. It also requires the software debugger to maintain exact models of the data caches with the same organiza-

tion and updating policies as in the target platform. The data cache models in the software debugger are updated while replaying the program offline in the same way the data cache are updated on the target platform. *mlvCFiat* ensures that the hardware platform only emits trace messages with load data values that cannot be inferred by the software debugger from the software of the data cache. This approach significantly reduces the number of trace messages that need to be emitted by the target platform, thus reducing the pressure on the trace port, which in turn reduces the on-chip resources and system cost.

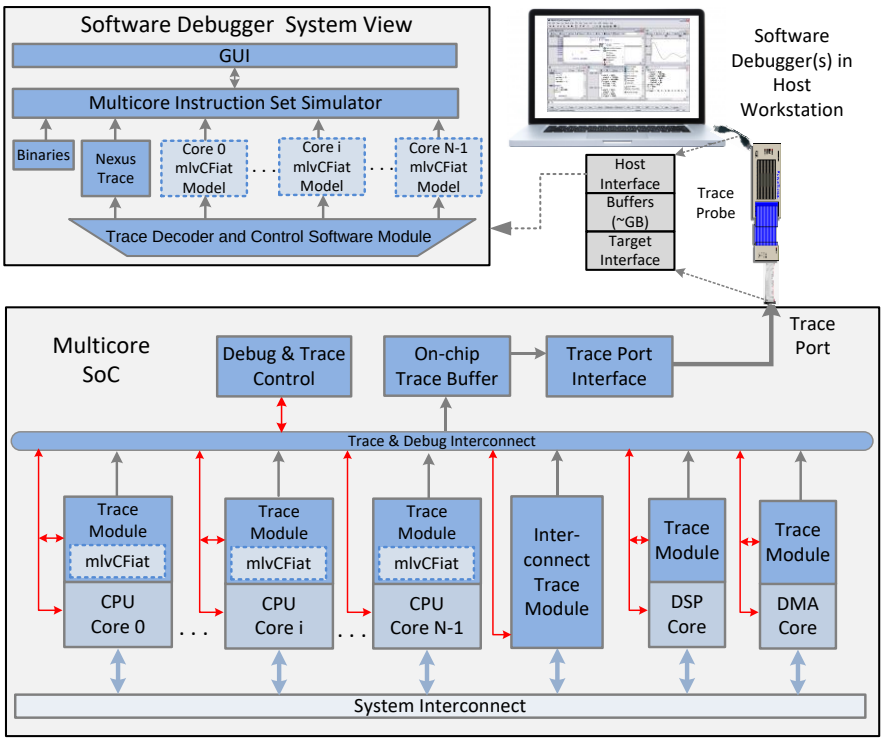

Figure 3.1 A system view of *mlvCFiat*

Figure 3.2 gives a detailed description of the hardware changes for a single core on the target platform (core *i*) required to support *mlvCFiat*. Each cache block is augmented with first-access (*FA*) tracking bits. These bits keep track of sub-blocks that need to be reported to the software debugger. For example, if a single first-access tracking bit can protect a sub-block of 4 bytes, then a 32-byte cache block would requires 8 first-access tracking bits. However, the size of sub-block protected by a first-access bit is a design parameter. When a sub-block is read for the first time, the sub-block is traced out and the corresponding *FA* flag is set. 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 temporal and spatial locality of data accesses to significantly reduce the number of trace messages that need to be reported. Each trace module includes a local first access hit counter (*Pi.fahCnt*) that counts the number of successive first-access hits on processor core *i.* The value of this counter is reported in a trace message on a first-access miss. The *Pi.PCC* register records the time stamp of a previously reported first-access miss. This register is used to a determine differentially encoded time stamp for the current trace miss event that occurs at clock cycle *Pi.CC* (*Pi.dCC=Pi.CC – Pi.PCC*).

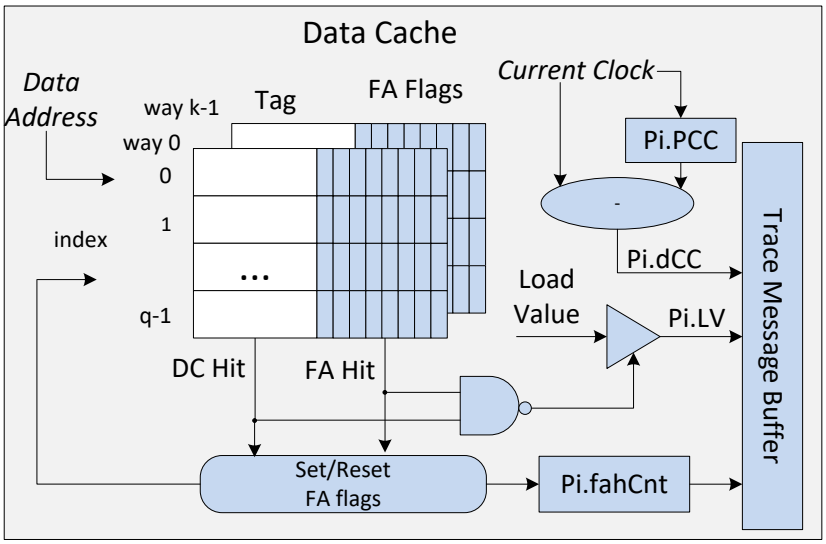

Figure 3.2. *mlvCFiat* structures for core i

Figure 3.3 describes *mlvCFiat* operation on the target platform carried out for memory reads, memory writes, and external invalidate requests. Each memory read results in an L1 data cache lookup; if the requested data item is found in the L1 data cache (*cache hit event*) and the corresponding *FA* bit(s) is set, we call this an *FA hit event*. In this case, we do not need to emit a trace message because the software debugger can retrieve the data from its software copy of the data cache. To synchronize *mlvCFiat* on the target platform with the software debugger, the first-access hit counter (*Pi.fahCnt*) is incremented (step 6 in Figure 3.3a). If the corresponding *FA* bit(s) is not set (*FA miss event*) then a trace message is emitted. The trace message includes a differentially encoded time stamp for that core (*Pi.dCC*), core id (*Pi*), the current value of the first-access hit counter (*Pi.fahCnt*), and the corresponding data cache sub-block that includes the load value (*Pi.LV*). Once the trace message is emitted, the corresponding *FA* bit(s) is set, and the *Pi.fahCnt* is cleared (step 4 in Figure

3.3a). For a *cache read miss event*, a new cache block is fetched from the memory and loaded into the data cache, the corresponding *FA* are cleared, and then the same steps as in *FA miss event* are carried out (steps 3 and 4 in Figure 3.3a). The *FA* bits are also updated for memory writes and external invalidations. For each memory write operation, if we have a cache hit and if the data is shared then the current processor acquires the ownership by invalidating the cache block in the other processor caches. If the current write operation writes an entire sub-block protected by an *FA* bit, then the corresponding *FA* bit is set (step 4 in Figure 3.3b). In case of a write miss, the corresponding *FA* bits are cleared for a newly fetched cache bock, and the same steps are carried out as in the case of a cache hit (steps 3 and 4 in Figure 3.3b). For external invalidations, the *FA* bits for the invalidated cache block are cleared. By capturing trace events at the L1 data cache level, cache coherence protocols are transparent to *mlvCFiat*. Thus, a write request to a shared block is treated as a miss in *mlvCFiat*.

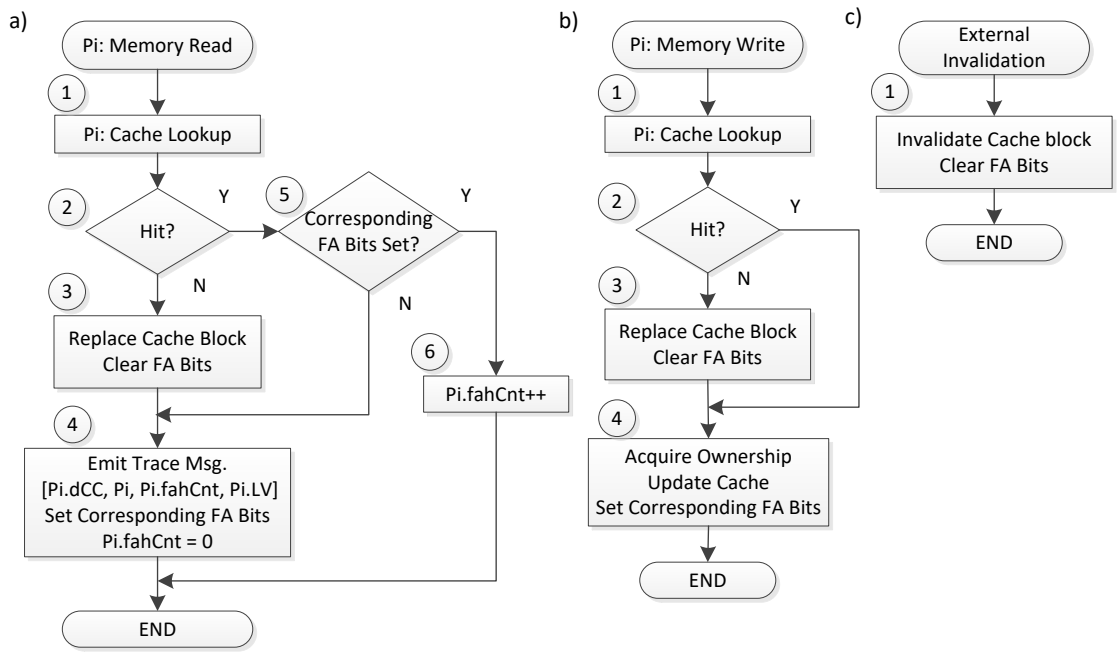

Figure 3.3 *mlvCFiat* operations on the target core *i* for a) memory reads, b) memory writes, and c) external invalidations

Figure 3.4 describes steps carried out by the software debugger in response to memory reads, memory writes, and external invalidations in a steady state. To replay the program offline, the software debugger relies on an instruction set simulator (ISS) of the target platform that uses the software models of data caches, first-access hit counters (*Pi.fahCnt),* the program binary, the exception traces, and the *mlvCFiat* trace messages received from the target platform. The software debugger reads and decodes trace messages while replaying the program. The formats of trace messages and lengths of the fields are known to the software debugger. The software copies of data caches and *Pi.fahCnt* are updated during the program replay using the same updating policies employed on the target platform. The debugger replays the instructions for each core using the corresponding ISS. For each memory read

operation, *Pi.fahCnt* is decremented by 1. If *Pi.fahCnt*>0 and the corresponding *FA* bits are set, the debugger retrieves the data values from the software copy of the data cache and moves to replay the next instruction. If the corresponding *FA* bits are not set it implies that there is an error in tracing (step 7 in Figure 3.4a). If *Pi.fahCnt*=0, we have a first read miss event: the load data value is retrieved from the current trace message, the software copy of the data cache is updated, and the corresponding *FA* bits are set. Then, a new trace message from the trace buffer is read for that core and *Pi.fahCnt* is updated with the new value extracted from the trace message (step 3 in Figure 3.4a). For each memory write operation, if the data is found in the software data cache and the cache block is shared, the current processor acquires ownership by invalidating copies of the cache block in other caches. Once the software copy of the cache block is updated, if the current write operation writes the entire sub-block protected by an *FA,* bit then the corresponding *FA* bit is set (step 4 in Figure 3.4b). In the case of external invalidations, the *FA* bits for the invalidated cache block are cleared.

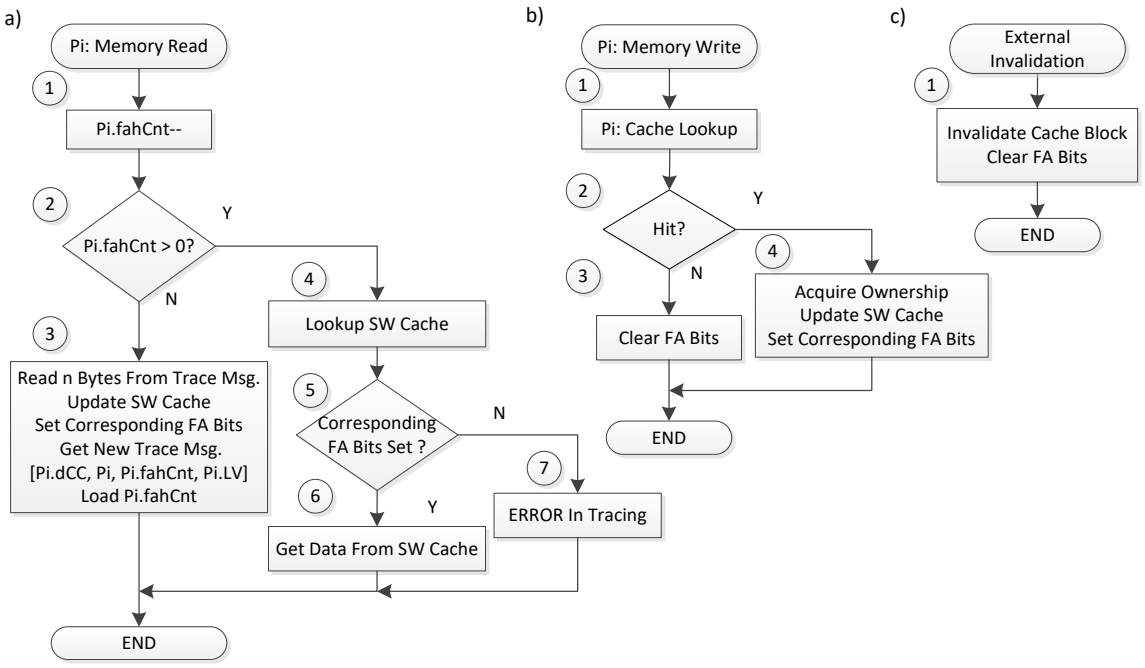

Figure 3.4 *mlvCFiat* operations in the software debugger on core *i* for a) memory reads, b) memory writes, and c) external invalidations

3.2 mc²RT

mc²RT is a hardware-based mechanism that reduces load data value traces by collecting a minimal set of trace messages by exploiting the *MOESI* [36] cache coherence protocol with a single tracking bit per data cache block [34].

Figure 3.5 shows the block diagram of a multicore system-on-chip (SoC) with the trace and debug infrastructure; light blue boxes represent additional *mc²RT* hardware and software modules. *mc²RT* requires hardware changes on the *L1* data caches to capture and compress load data value traces. It also requires a software debugger to maintain an exact model of the data caches with the same organization and updating policies as the target platform. The data cache models in the software

debugger are updated while replaying the program offline in the same way the data caches are updated on the target platform. *mc²RT* ensures that the hardware platform only emits trace messages with load data values that cannot be inferred by the software debugger from the software copy of the data caches. This approach significantly reduces the number of trace messages that need to be emitted by the target platform, thus reducing the pressure on the trace port, which in turn reduces the on-chip resources and the system cost.

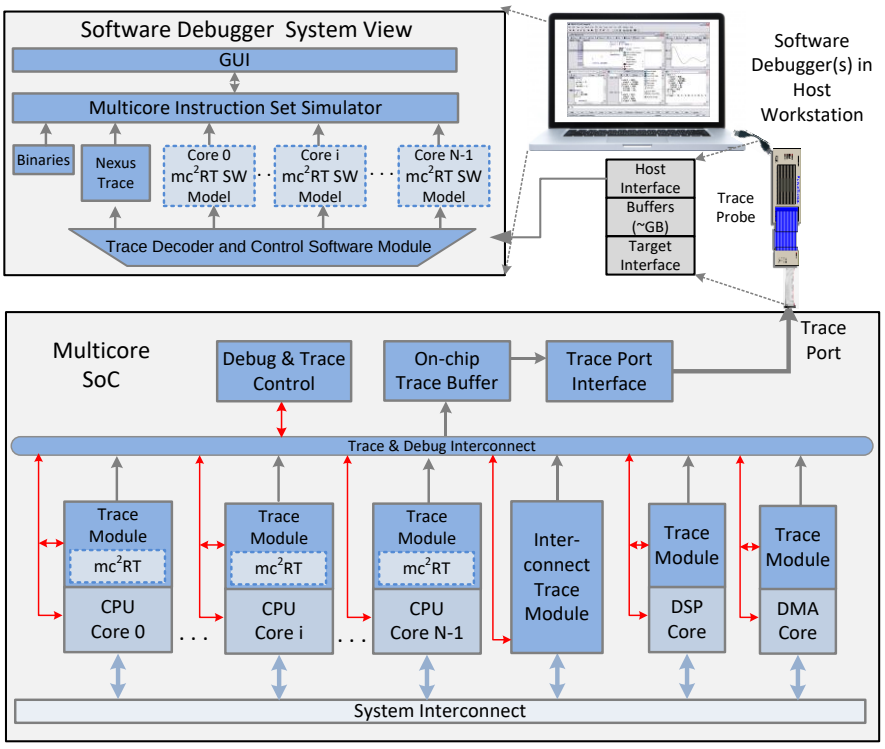

Figure 3.5 A system view of *mc²RT*

Figure 3.6 gives a detailed description of hardware changes for a single core on the target platform (core *i*) required to support *mc²RT*. Each cache block is aug-

mented with a trace tracking bit (*TR*). The trace bit keeps track of whether the associated cache block is reported or not to the software debugger. The cache block fetched from the memory for the first time by a processor having a read miss will be emitted through the trace port. Once the cache block is emitted, the trace bit is set. Previously reported cache 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 accesses to significantly reduce the number of trace messages that need to be reported. Each trace module includes a local trace hit counter (*Pi.THCnt*) that counts the number of successive trace hits on processor core *i*. The current value of this counter is reported together in a trace message on a trace miss. The *Pi.PCC* register records the time stamp of a previously reported trace message. The register is used to determine a differentially encoded time stamp for the current trace miss event that occurs at a clock cycle *Pi.CC* (*Pi.dCC* = *Pi.CC* – *Pi.PCC*).

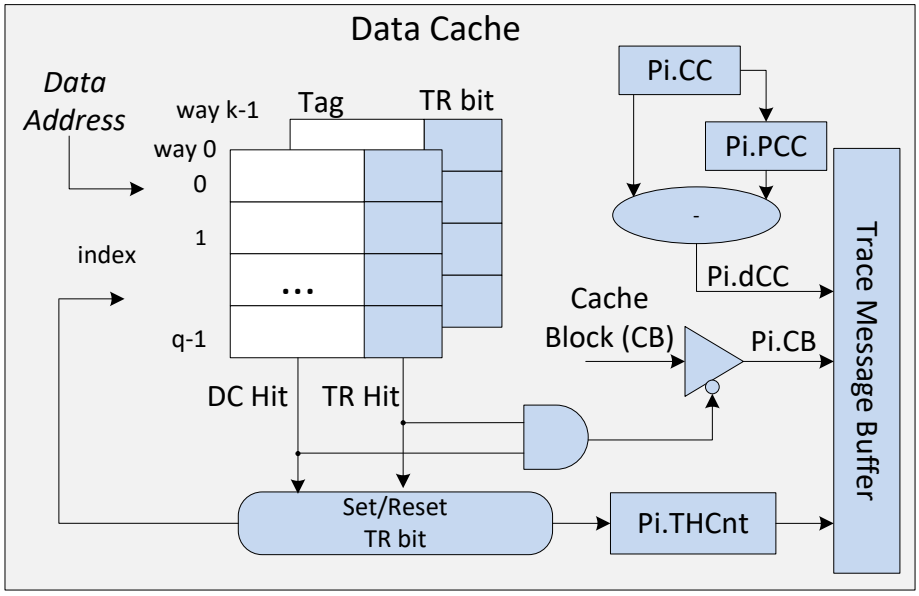

Figure 3.6 *mc²RT* structures for core *i*

Figure 3.7 and Figure 3.8 describe *mc²RT* operation on the target platform for memory reads by core *i*. Each memory read results in an L1 data cache lookup. If the requested data item is found in the L1 data cache (*cache hit event*) and the trace bit is set, we have a *trace hit event*. In this case, we do not need to emit a trace message because the software debugger can retrieve the data from its software copy of the data cache. To synchronize trace hit events in *mc²RT* on the target platform with the software debugger, the trace hit counter (*Pi.THCnt*) is incremented (step 7). If the corresponding trace bit is not set (*trace miss event),* a trace message is emitted. The trace message includes a differentially encoded time stamp for that core (*Pi.dCC*), core id (*Pi*), the current value of the trace hit counter (*Pi.THCnt*) and the corresponding cache-block that includes the load data value (*Pi.CBj*) (step 4). Once the trace message is emitted the corresponding trace bit is set, and *Pi.THCnt* is

cleared (step 5). For a *cache read miss event,* a *Coherent Read Transaction* is issued (step 6). The requested cache block is supplied to the processor core *i* (*Pi*) by another processor cache (*Px*) or by main memory. In a *Coherent Read Transaction*, a snoop lookup is performed by all the caches, as follows and shown in Figure 3.8.

- If the snoop lookup finds the requested cache block in the *Modified (M)* state, it is transferred to *Pi* along with its trace bit. The state of the cache block in *Px* transitions to *Owned (O)* and the new state of the *Pi* cache block is *Shared (S)* (step 13). If this cache block is not previously reported (*Px.CBj.TR*=0) by processor *Px*, then it is reported by processor *Pi*. Since the cache block is going to be reported by the processor *Pi*, the trace bit for *Px* is set to 1 (step 15).
- If the snoop lookup finds the requested cache block in the *Exclusive (E)* state, it is transferred to *Pi* along with its trace bit. Since this cache block is already reported first time when it is read from memory, trace bit for *Pi* and *Px* is 1. The states of the cache block in *Pi* and *Px* are updated to *Shared* state (step 17).
- If the snoop lookup finds the requested cache block in the O*wned* state, the requested cache block is transferred to *Pi* along with its trace bit and the new state of *Pi* is updated to *Shared*. The trace bit for *Pi* and *Px* is 1 because the only way to transition to *Owned* state is from the *Modified* state by having coherent read request from another processor to that cache block (step 19).
- If the snoop lookup finds the requested cache block in the *Shared* state, it is transferred to *Pi* along with its trace bit only if the current

processor is responsible to transfer. By design a cache block can be in the *Shared* state only after *Coherent Read Transaction* therefore, the cache block is already reported by another processor and the trace bit for *Pi* and *Px* is 1. The state of the cache block in *Pi* is updated to *Shared* state (step 21).

- If the requested cache block is not found in any processor cache, then it is retrieved from main memory and corresponding trace bit is cleared. The new state of the cache block is set to *Exclusive*.

After *Coherent Read Transaction*, the same steps as in the *CPU READ* operation are carried out.

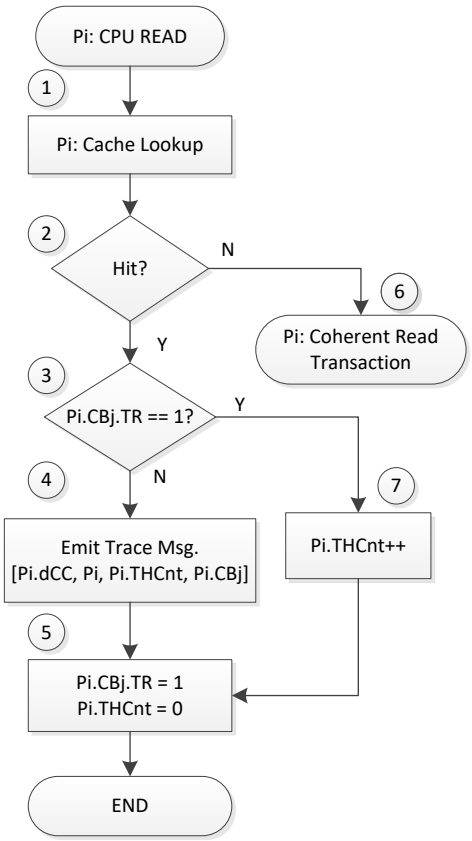

Figure 3.7 *mc²RT* operation on the target core *i* for memory reads

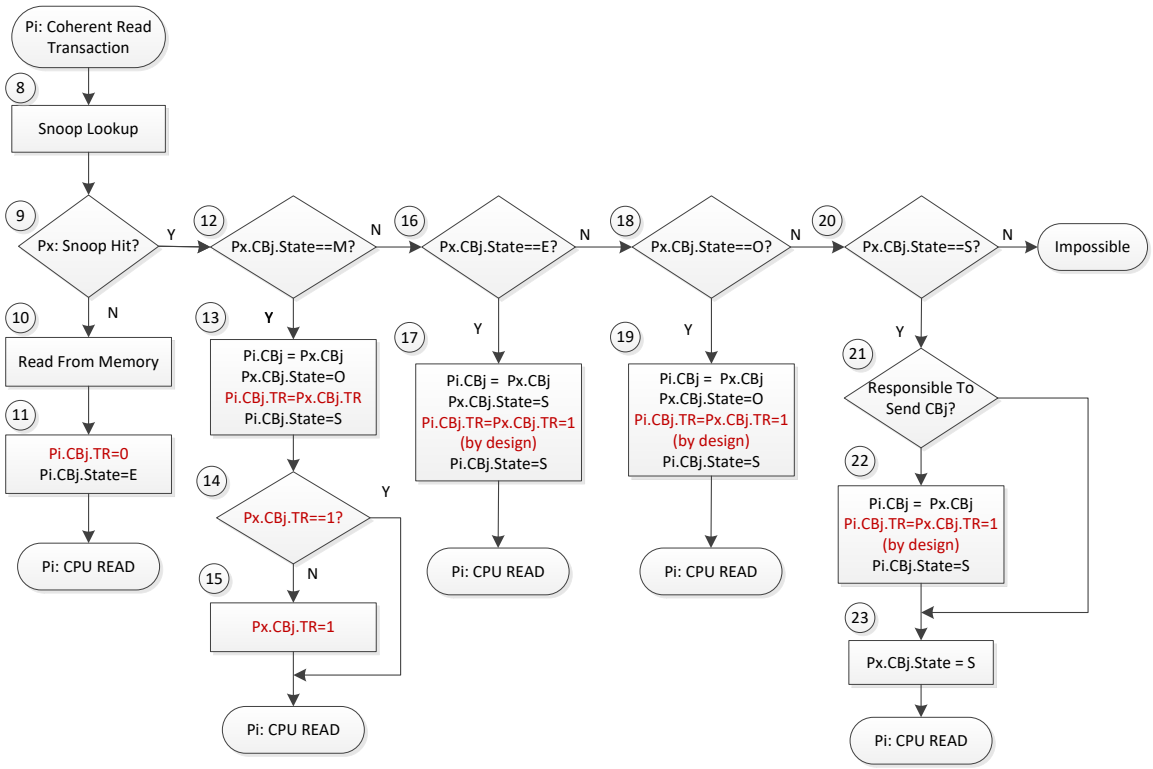

Figure 3.8 Coherent Read Transaction in *mc²RT* on the target core *i*

Figure 3.9, Figure 3.10, and Figure 3.11 describes the operation of *mc²RT* on the target platform for memory writes by core *i*. The data cache is looked up for the requested cache block. In the case of a cache hit in the *Exclusive* state, the state of the cache block is upgraded to the *Modified* state (step 6). If the requested cache block is in the *Owned* or *Shared* state, a *Coherent Invalidate* transaction is initiated. A snoop lookup is performed by all other caches. If the snoop lookup finds the cache block in the *Shared* state or *Owned* state, then the cache block is invalidated and the trace bit is cleared. The state of the cache block in *Pi* is updated to *Modified* (step 28 and step 30). If the requested cache block is not found in the processor *Pi*, then a

Coherent Read and Invalidate transaction is initiated. The sequence of events performed by other caches in response to a *Coherent Read and Invalidate* transaction is described as follows:

- If the snoop lookup finds the requested cache block in the *Modified* state, it is transferred to *Pi* along with its trace bit. The new state of the *Pi* cache block is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The trace bit for the invalidated cache block is cleared (step 16).
- If the snoop lookup finds the requested cache block in the *Exclusive* state, it is transferred to *Pi* along with its trace bit. By design, the trace bit for a cache block in *Pi* and *Px* is 1, because this cache block is already reported first time when it is read from memory. The new state of the *Pi* cache block is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The trace bit for the invalidated cache block is cleared (step 18).
- If the snoop lookup finds the requested cache block in the *Owned* state, it is transferred to *Pi* along with its trace bit. The new state for cache block in *Pi* cache is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The trace bit for *Pi* and *Px* is 1 because the only way to transition to *Owned* state is from the *Modified* state by having a coherent read request from another processor to that cache block. The trace bit for the invalidated cache block is cleared (step 20).
- If the snoop lookup finds the requested cache block in the *Shared* state, it is transferred to *Pi* along with its trace bit if the current pro-

cessor is responsible to transfer. By design, a cache block can be in the *Shared* state only after a Coherent Read Transaction, therefore the cache block is already reported by another processor, the trace bit for a cache block in *Pi* and *Px* is 1. The cache block of the processor *Px* is invalidated and the corresponding trace bit is cleared (step 23 and step 24).

- If the requested cache block is not found in any processor cache, then it is retrieved from main memory and corresponding trace bit is cleared. The new state of the cache block is set to *Modified* (step 13 and step 14).

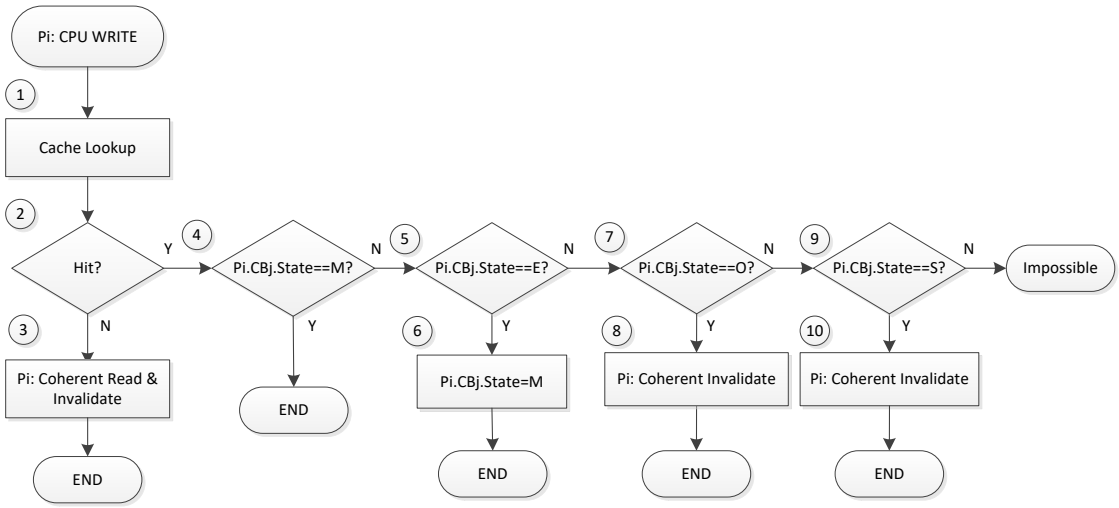

Figure 3.9 *mc²RT* operation on the target core *i* for memory writes

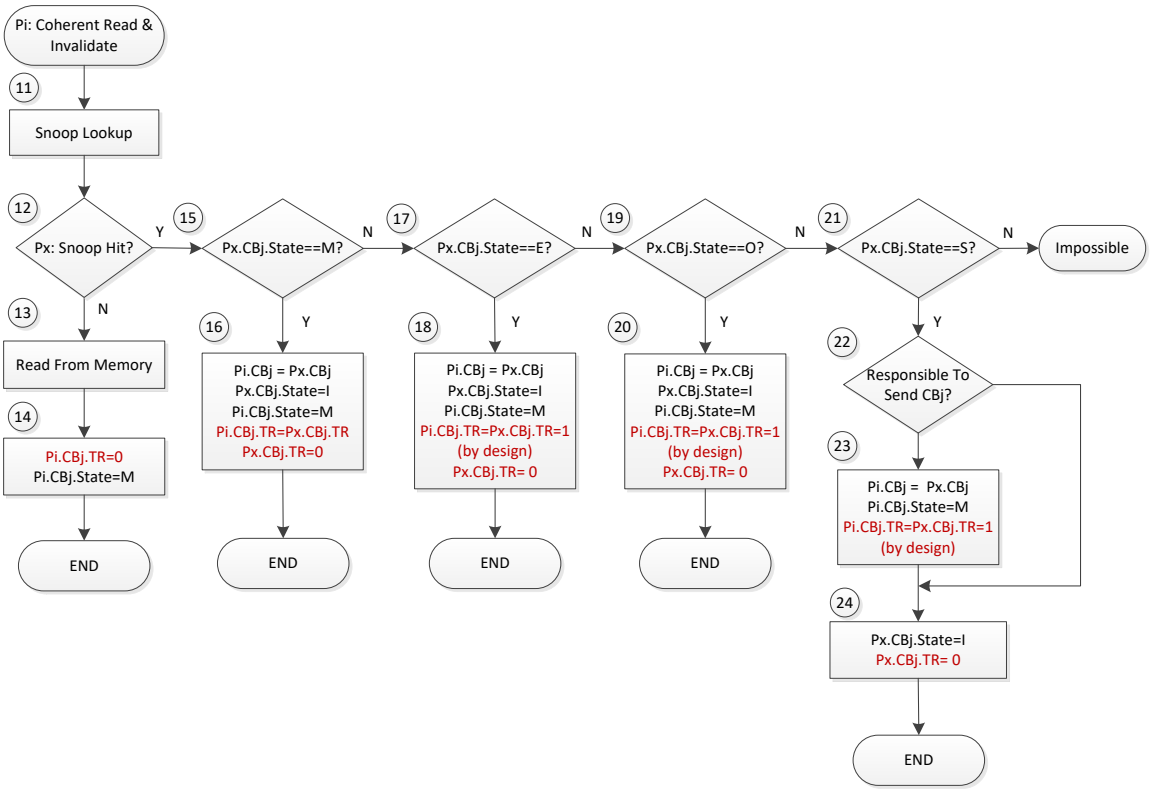

Figure 3.10 Coherent Read and Invalidate in *mc²RT* on target core *i*

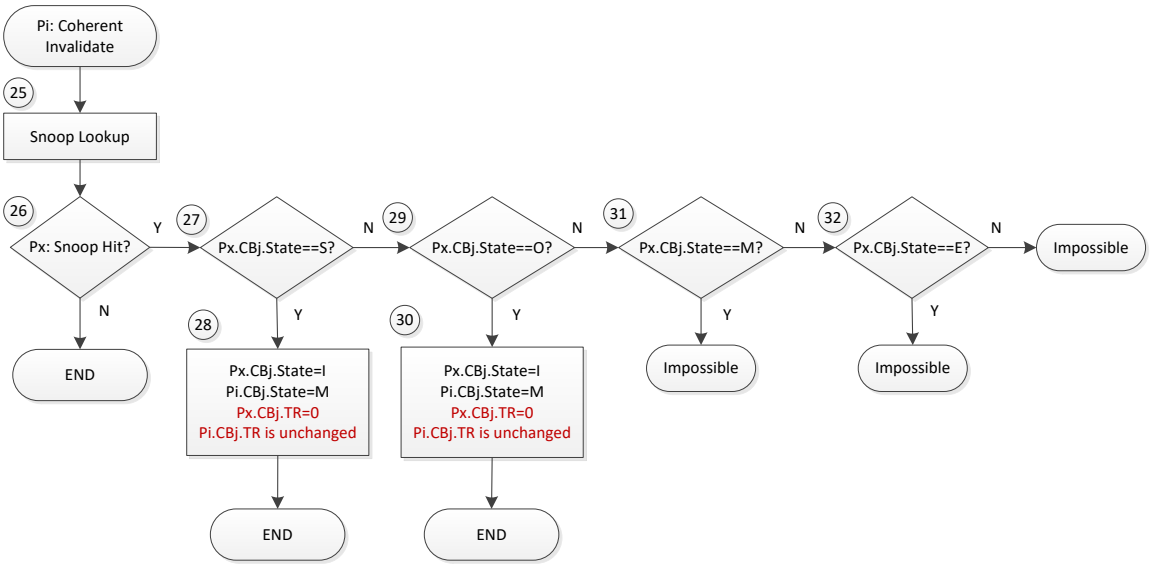

Figure 3.11 Coherent Invalidate in *mc²RT* on target core *i*

Figure 3.12 describes the steps carried out by the software debugger in response to memory reads and memory writes in a steady state. To replay the program offline, the software debugger relies on an instruction set simulator (ISS) of the target platform that uses the software models of data caches, trace hit counters (*Pi.THCnt),* the program binary, exception traces, and the *mc²RT* trace messages received from the target platform. The software debugger reads and decodes trace messages while replaying the program. The formats of the trace messages and lengths of the fields are known to the software debugger. The software copies of data caches and *Pi.THCnt* are updated during the program replay using the same updating policies employed on the target platform. The debugger replays the instructions for each core using the corresponding ISS. For each memory read operation, *Pi.THCnt* is decremented by 1. If *Pi.THCnt*>0 and if the data is found in the software copy of the data cache of processor *Pi* with the corresponding trace bit set, the debugger retrieves the data values from the software copy of the data cache and moves to replay the next instruction. If the corresponding trace bit is not set, it implies that there is an error in tracing (step 7 in Figure 3.12a). If *Pi.THCnt*>0 and data is not found in the software copy of the data cache of processor *Pi*, then the software debugger gets data from another processors data cache. If *Pi.THCnt*=0, we have a trace miss event: the load data value is retrieved from the current trace message, the software copy of the data cache is updated and the corresponding trace bit is set. Then, a new trace message from the trace buffer is read for that core and *Pi.THCnt* is updated with a new value extracted from the trace message (step 3 in Figure 3.12a). For each memory write operation, if the data is found in the software cache and the cache block is shared, the current processor acquires the ownership by

invalidating copies of the cache block in other caches. If the data is a hit in other caches then, the trace bit is inherited.

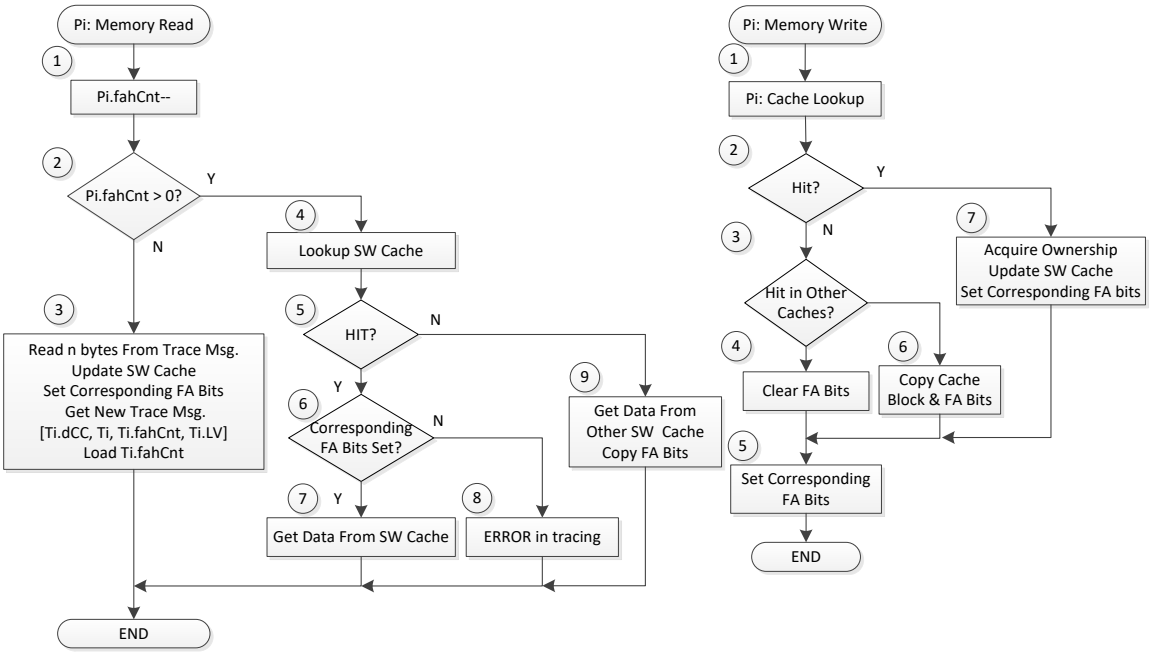

Figure 3.12 *mc²RT* operation in the software debugger on core *i* for a) memory reads b) memory writes

3.3 mc²RFiat

mc²RFiat is a hardware-based mechanism that reduces load data value traces by collecting a minimal set of trace messages by exploiting the MOESI cache coherence protocol. This technique is a combination of *mlvCFiat* and *mc²RT*; it requires additional support to copy first-access tracking bits from another cache. This can be implemented by either having additional data lines on the bus to carry FA bits or an extra bus transaction.

Figure 3.13 shows the block diagram of a multicore system-on-chip (SoC) with trace and debug infrastructure; light blue boxes represents additional *mc²RT*[2] hardware and software modules. *mc²RFiat* requires hardware changes on the L1 data caches to capture and compress load data value traces. It also requires the software debugger to maintain exact models of the data caches with the same organization and updating policies as in the target platform. The data cache models in the software debugger are updated while replaying the program offline in the same way the data caches are updated on the target platform. *mc²RFiat* ensures that the hardware platform emits only trace messages with load data values that cannot be inferred by the software debugger from the software copy of the data caches. This approach significantly reduces the number of trace messages that need to be emitted from the target platform, thus reducing the pressure on the trace port, which in turn reduces on-chip resources and the system cost.

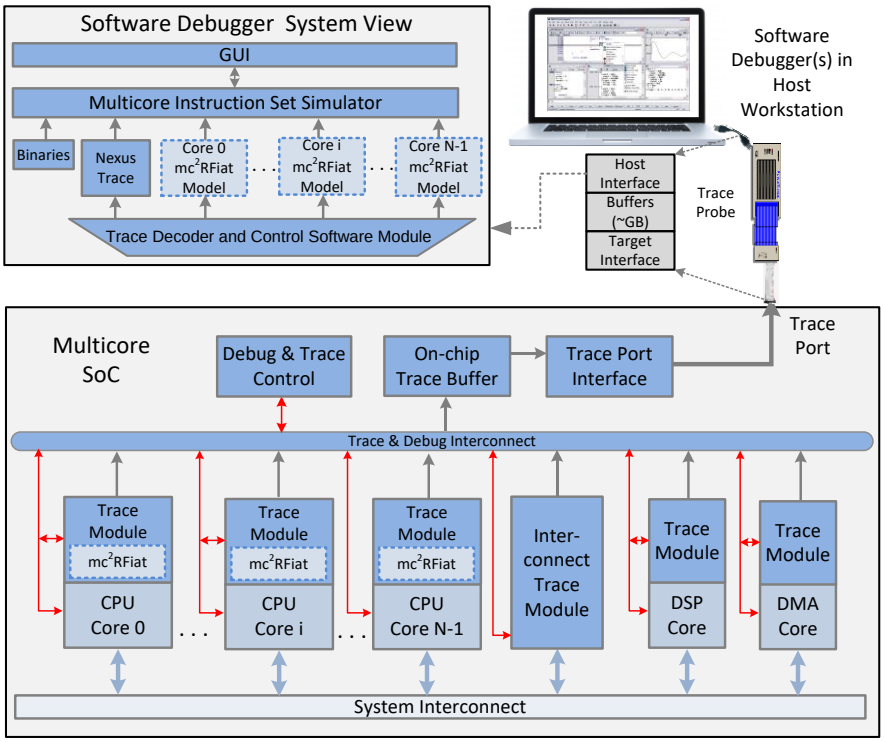

Figure 3.13 A system view of *mc²RFiat*

Figure 3.14 gives a detailed description of hardware changes for a single core on the target platform (core *i)* required to support *mc²RFiat*. Each cache block is augmented with first-access (*FA*) tracking bits. These bits keep track of sub-blocks that need to be reported to the software debugger. For example, if a single first-access tracking bit can protect a sub-block of 4 bytes, then a 32-byte cache block requires 8 first-access tracking bits. However, the size of the sub-block protected by a first-access bit is a design parameter. When a sub-block is read for the first time, the sub-block is traced out and the corresponding *FA* bit(s) is set. 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 temporal and spatial locality of data accesses to significantly reduce the number of trace messages that need to be reported.

Each module includes a local first access hit counter (*Pi.fahCnt*) that counts the number of successive first-access hits on processor core *i*. The value of this counter is reported together in a trace message on a first-access miss. The *Pi.PCC* register records the time stamp of a previously reported first-access miss. This register is used to determine differentially encoded time stamp for the current trace miss event that occurs at clock cycle *Pi.CC* (*Pi.dCC* = *Pi.CC* – *Pi.PCC*).

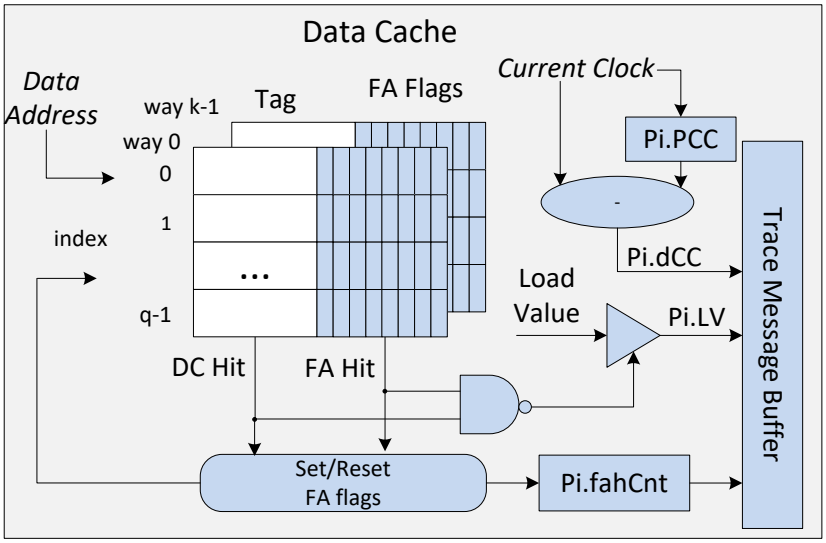

Figure 3.14 *mc²RFiat* structures for core *i*

Figure 3.15 and Figure 3.16 describes the operation of *mc²RFiat* on the target platform for memory reads by core *i*. Each memory read results in an L1 data cache lookup. If the requested data item is found in an L1 data cache (*cache hit event*) and the corresponding first-access bit(s) is set, we have an *FA hit event*. In this case, we do not need to emit a trace message because the software debugger can retrieve data from its software copy of the data cache. To synchronize *mc²RFiat* on the target plat-

form with the software debugger, the first-access hit counter (*Pi.fahCnt*) is incremented (step 7). If the corresponding *FA* bit(s) is not set (*FA miss event*), a trace message is emitted. The trace message includes a differentially encoded time stamp for that core (*Pi.dCC*), core id (*Pi*), the current value of first-access hit counter (*Pi.fahCnt*) and the corresponding data cache sub-block that include the load value (*Pi.LV*) (step 4). Once the trace message is emitted, the corresponding *FA* bit(s) is set, and the *Pi.fahCnt* counter is cleared (step 5). For a *cache read miss event,* a *Coherent Read Transaction* is issued (step 6). The requested cache block is supplied to the core *i* (*Pi*) by another processor cache (*Px*) or by main memory. In *Coherent Read Transaction*, a snoop lookup is performed by all the caches, as follows:

- If the snoop lookup finds the requested cache block in the *Modified (M)* state, it is transferred to *Pi* along with its *FA* bits. The state of the cache block in *Px* transitions to *Owned (O)* and the new state of the *Pi* cache block is *Shared (S)* (step 13).
- If the snoop lookup finds the requested cache block in the *Exclusive (E)* state, it is transferred to *Pi* along with its *FA* bits. The states of the cache block in *Pi* and *Px* are updated to *Shared* state (step 15).
- If the snoop lookup finds the requested cache block in the O*wned* state, it is transferred to *Pi* along with its *FA* bits. The new state of *Pi* cache block is updated to *Shared* (step 19) and the *Px* state is unchanged (step 17).
- If the snoop lookup finds the requested cache block in the *Shared* state, it is transferred along with its *FA* bits to *Pi* only if the current processor is responsible to transfer. The new state of the *Pi* cache

block is updated to *Shared* and the *Px* state is unchanged (steps 20 and 21).

- If the requested cache block is not found in any processor cache, then it is retrieved from main memory and the corresponding *FA* bits are cleared. The new state of the cache block is *Exclusive* (step 11).

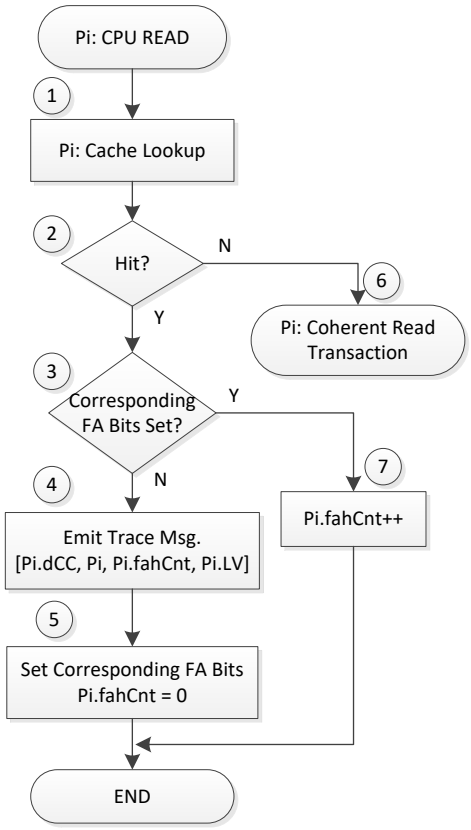

Figure 3.15 *mc²RFiat* operation on the target core *i* for memory reads

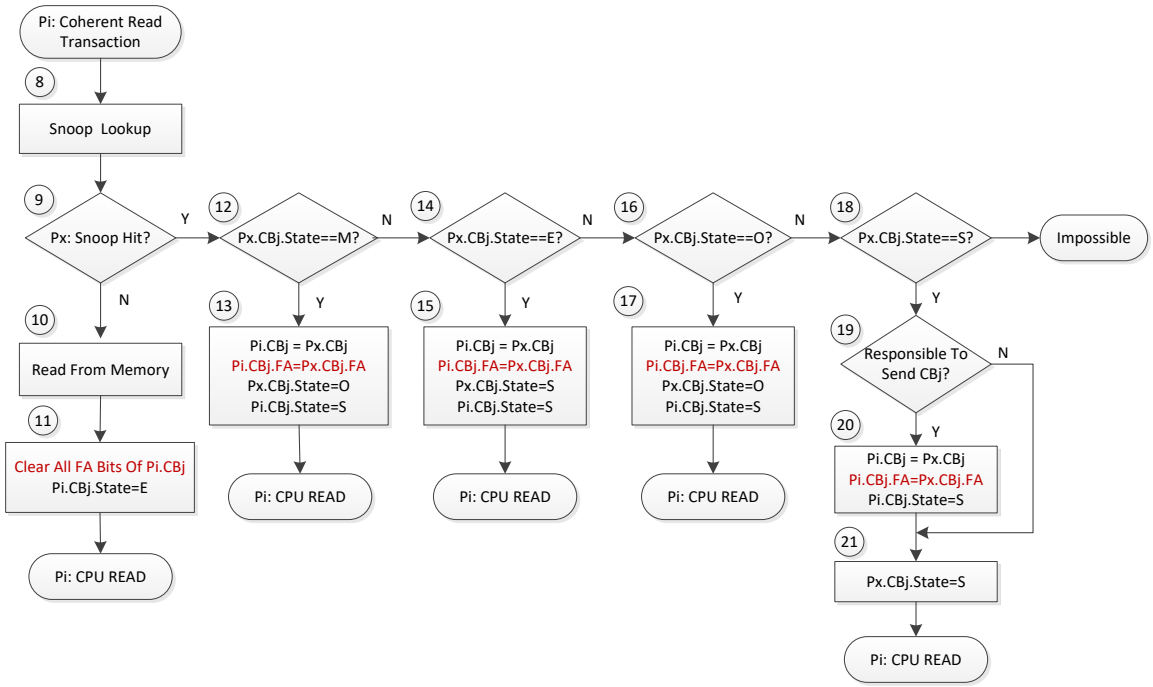

Figure 3.16 Coherent Read Transaction in *mc²RFiat* on the target core *i*

Figure 3.17, Figure 3.18, and Figure 3.19 describes the operation of *mc²RFiat* for memory writes on the target platform by core *i*. A data cache lookup performed for the requested cache block. In case of a cache hit in the *Exclusive* state, the state of the cache block is upgraded to the *Modified* state (step 7). If the requested cache block is in the *Owned* or *Shared* state, a *Coherent Invalidate* transaction (Figure 3.19) is initiated. A snoop lookup is performed by all other caches. If the cache block is hit in the *Shared* or *Owned* state, then the cache block is invalidated and the *FA* bits are cleared. The state of the cache block in *Pi* is updated to *Modified* (state 29 and step 31). If the requested cache block is not found in processor *Pi*, then a *Coherent Read and Invalidate* transaction (Figure 3.18) is initiated. The sequence of

events performed by other caches in response to a *Coherent Read and Invalidate* transaction is described as follows:

- If the snoop request finds the requested block in the *Modified* state, it is transferred to *Pi* along with its *FA* bits. The new state for the *Pi* cache block is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The *FA* bits for the invalidated cache block are cleared (step 17).
- If the snoop request finds the requested block in the *Exclusive* state, it is transferred to *Pi* along with its *FA* bits. The new state for the *Pi* cache block is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The *FA* bits for the invalidated cache block are cleared (step 19).
- If the snoop request finds the requested block in the *Owned* state, it is transferred to *Pi* along with its *FA* bits. The new state for a cache block in *Pi* cache is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The *FA* bits for the invalidated cache block are cleared (step 21).
- If the snoop request finds the requested block in the *Shared* state, it is transferred to *Pi* along with its *FA* bits. The new state for the cache block in *Pi* cache is *Modified* and the state of the cache block in *Px* is updated to *Invalid*. The *FA* bits for the invalidated cache block are cleared (steps 24 and 25).

- If the requested cache block is not found in any processor cache, then it is retrieved from main memory and the corresponding *FA* bits are cleared. The new state of the cache block is set to *Modified* (step 16).

If the current memory write operation writes the entire sub-block protected by FA corresponding FA bits are set.

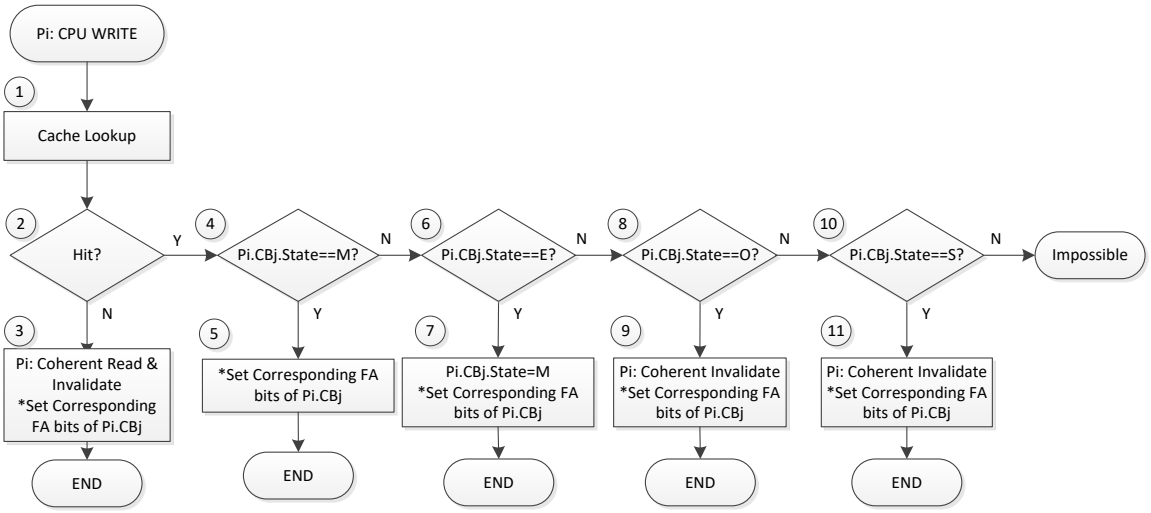

Figure 3.17 *mc²RFiat* operation on the target core *i* for memory writes

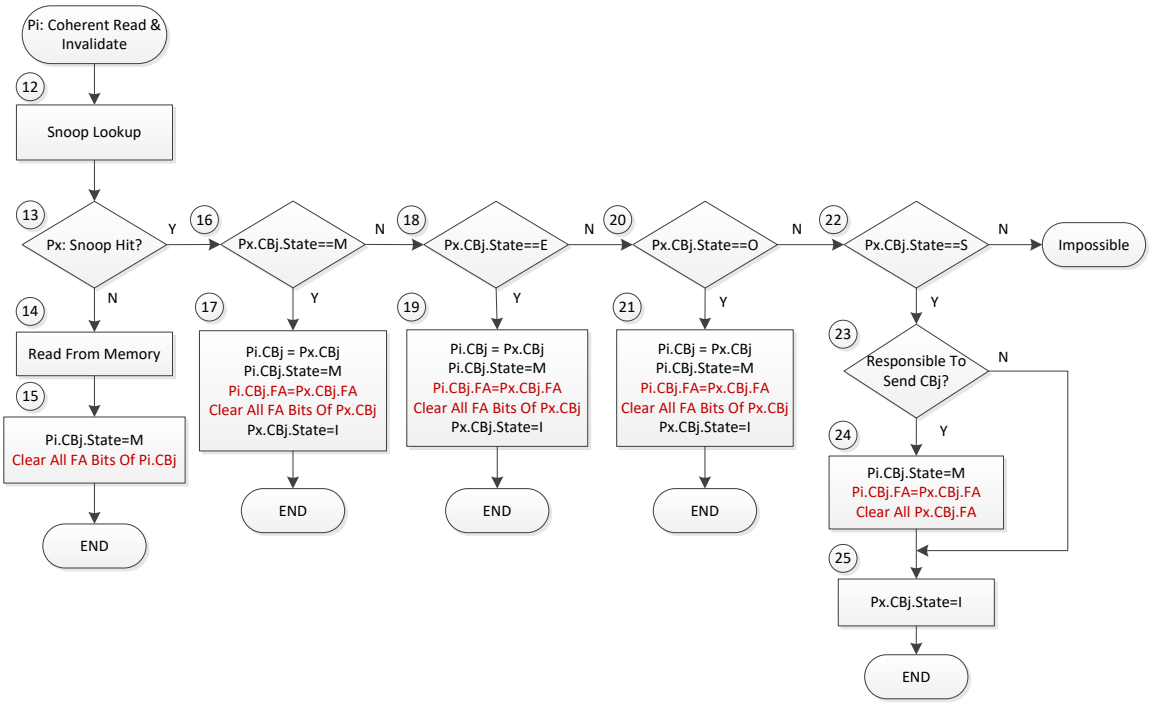

Figure 3.18 Coherent Read and Invalidate in *mc²RFiat* on the target core *i*

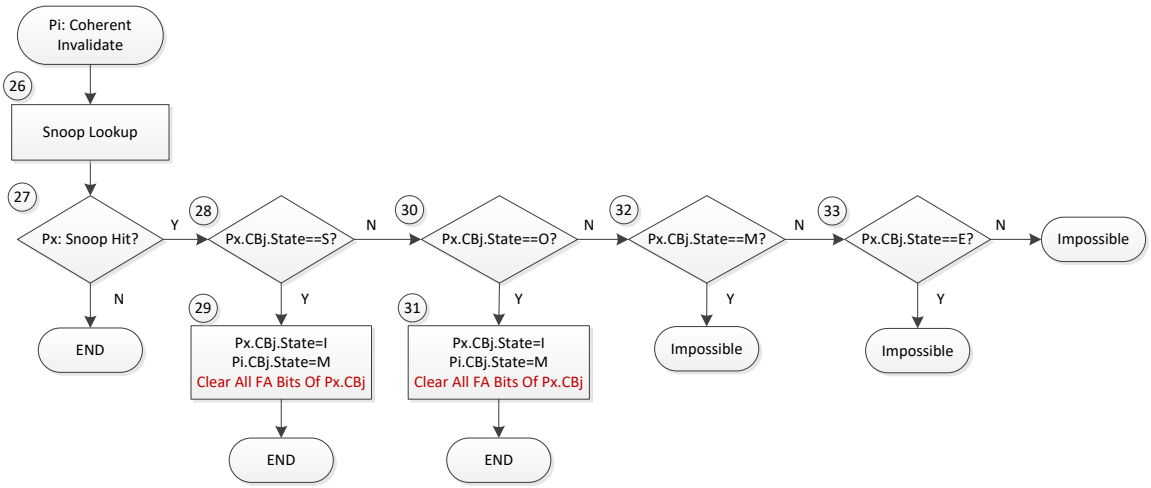

Figure 3.19 Coherent Invalidate in *mc²RFiat* on the target core *i*

Figure 3.20 describes steps carried out by the software debugger in response to memory reads and memory writes in a steady state. To replay the program offline, the software debugger relies on an instruction set simulator (ISS) of the target platform that uses software models of data caches, first access hit counters (*Pi.fahCnt),* the program binary, the exception traces, and the *mc²RFiat* trace messages received from the target platform. The software debugger reads and decodes trace messages while replaying the program. The formats of the trace messages and the lengths of the fields are known to the software debugger. The software copies of data caches and *Pi.fahCnt* are updated during the program replay using the same updating policies employed on the target platform. The debugger replays the instructions for each core using the corresponding ISS. For each memory read operation, *Pi.fahCnt* value is decremented by 1. If *Pi.fahCnt*>0 and if the data is found in the software copy of the data cache of the processor *Pi* with the corresponding *FA* bit(s) set, the debugger retrieves the data values from the software copy of the data cache for that processor and moves to replay the next instruction. If the corresponding *FA* bits are not set, it implies that there is an error in tracing (step 8 in Figure 3.20a). If *Pi.fahCnt*>0 but data is not found in the software copy of the data cache of processor *Pi*, then the software debugger gets data from another processors data cache. If *Pi.fahCnt*=0, we have a first read miss event: the load data value is retrieved from the current trace message, the software copy of the data cache is updated and the corresponding *FA* bits are set. Then, a new trace message from the trace buffer is read for that core and *Pi.fahCnt* is updated with a new value extracted from the trace message (step 3 in Figure 3.20a). For each memory write operation, if the data is found in the software copy of the data cache and the cache block is shared, the current cache acquires

the ownership by invalidating copies of the cache block in other caches and then software copy of cache block is updated. If the data is a hit in another cache, the *FA* bits are also copied along with the cache block. In the case of a cache miss event, *FA* bits for the corresponding block are cleared. If the current memory write operation writes the entire sub-block protected by a single FA bit, corresponding FA bits are set (step 5 in Figure 3.20b).

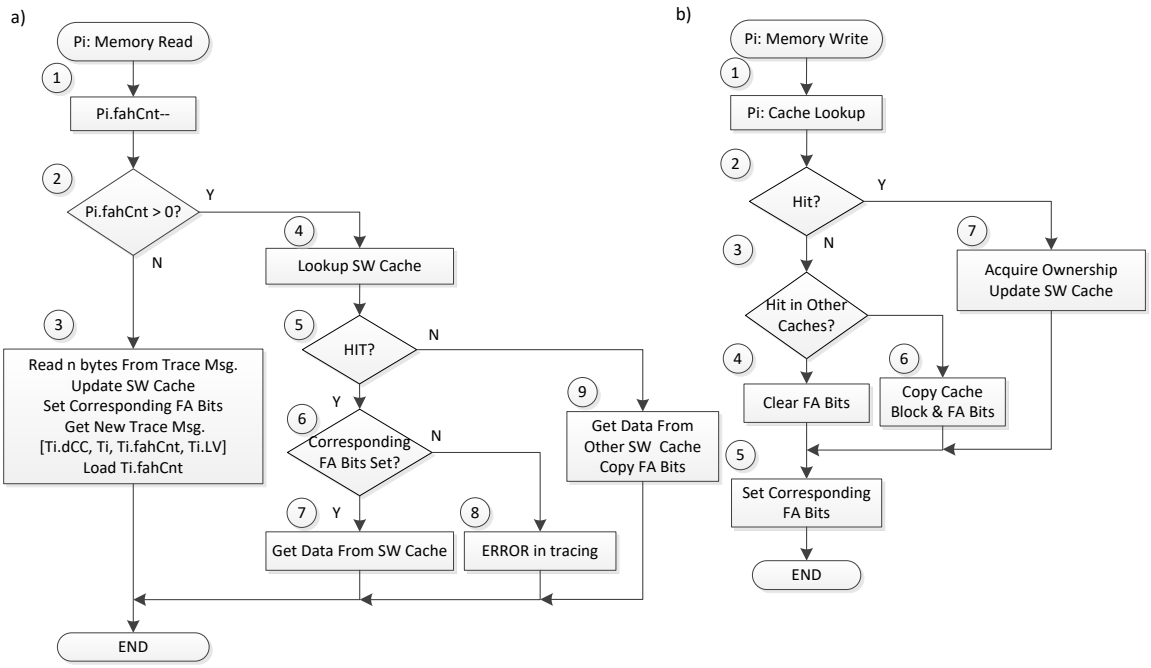

Figure 3.20 *mc²RFiat* operation in the software debugger on core *i* for a) memory reads, and b) memory writes

CHAPTER 4

EXPERIMENTAL EVALUATION

This chapter discusses the experimental setup used for the evaluation of the proposed techniques. Figure 4.1 describes the experimental flow used to create hardware traces and to analyze the trace port bandwidth requirements. The experimental flow include three major components (i) software timed trace generator, (ii) *mlvCFiat*, *mc²RT*, and *mc²RFiat* simulators to filter the load data value traces, and (iii) software to hardware trace converters and encoders. Section 4.1 discusses the TmTrace tool used to create timed software traces and their output format. Section 4.2 discusses the implementation and verification details of the *mlvCFiat* simulator. Section 4.3 discusses the implementation and verification details of the *mc²RT* simulator. Section 4.4 discusses the implementation and verification details of the *mc²RFiat* simulator. Section 4.5 discusses the conversion of software traces to hardware traces. Finally, Section 4.6 discusses the workload used for the trace port bandwidth evaluation and experimental analysis to select a good granularity size and variable encoding parameters.

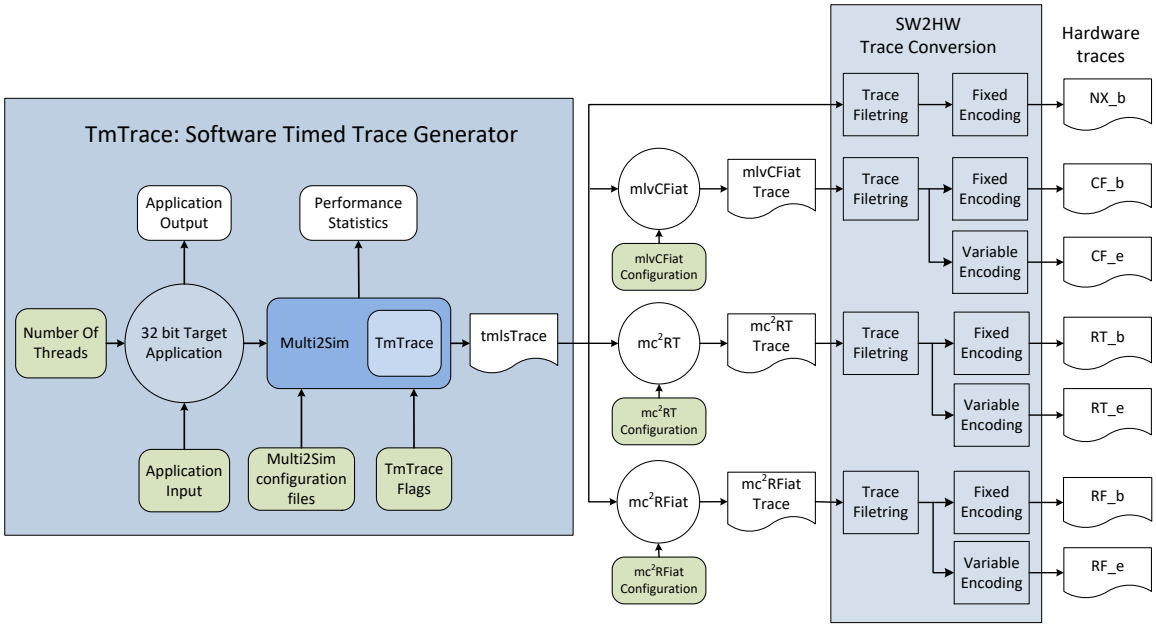

Figure 4.1 Experiment flow to create hardware traces

4.1 Software Timed Trace Generator

TmTrace [28] (Timed Multithreaded Trace) is a software module developed as an extension of the Multi2Sim simulator [37] that was used to create timed software traces. This tool provides different options for collecting traces of interest. For example, we can collect control flow traces for all committed instructions, data traces for memory reads and/or memory writes. In this case, we use this tool to collect traces for all memory read and memory write operations. We refer to these traces as *tmlsTraces*. The format of these traces is shown in Figure 4.2. Each trace message includes the following fields:

- CC: Clock cycle in which the memory instruction is committed;
- Pi: Thread id or processor core id;

- L/S: Memory read (L=0) or write (S=1) instruction;
- PC: Value of the program counter or address of the instruction;
- OA: Address of the operand;
- OS: Size of the operand;
- LV: Value of the operand read from memory. This field is empty for a memory write operation.

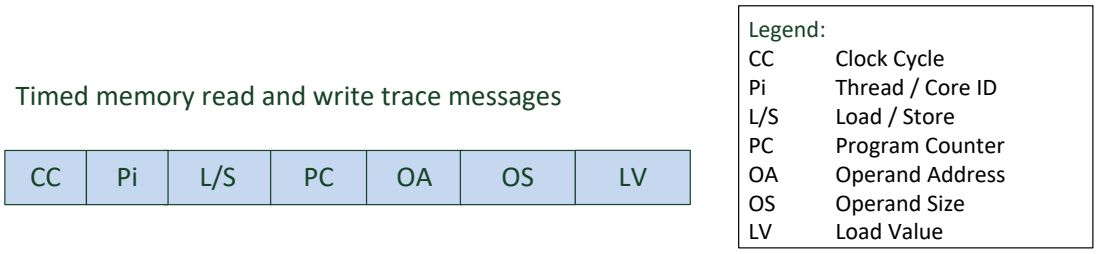

Figure 4.2 Trace messages generated for memory reads and writes

4.2 *mlvCFiat* Simulator

mlvCFiat simulator takes *tmlsTraces* as an input, implements the *mlvCFiat* trace filtering as described in Section 3.1, and outputs the compressed load data value traces. The *mlvCFiat* simulator maintains the *mlvCFiat* structures (Figure 3.6) private to each processor core. Section 4.2.1 discusses the main implementation details of the *mlvCFiat* simulator, and Section 4.2.2 describes the test cases used to confirm its correctness.

Table 4.1 describes the parameters that can be used to control the behavior of the *mlvCFiat* simulator. With the help of these parameters, we can control output file size, size of the data caches, the size of the sub-block protected by a single first-

access bit, and output file name. Figure 4.3 shows the output format of the *mlvCFiat* trace message. It includes a time stamp (*CC*), the thread id (*Pi*), the first access hit counter (*fahCnt*), the operand size (*OS*), the corresponding data cache sub-block(s) that includes the load value (*LV*), and the length of the *LV* field (*LLV*). Note: the length of the *LV* filed depends on the granularity size and it can be greater than or equal to the length of the actual operand. For example, if the granularity size is 4-bytes and the operand size is 1-byte, the minimum length of the *LV* field is 4-bytes. The *mlvCFiat* simulator also outputs the program statistics. An example of a statistics file is given in Figure 4.4. It includes information about the number of memory read and memory write operations, cache hits, cache misses, and first-access hits etc.

Table 4.1 *mlvCFiat* 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 a sub-block 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 Note: *.txt = descriptors, *.Statistics = Statistics of *tmlvCFiat*

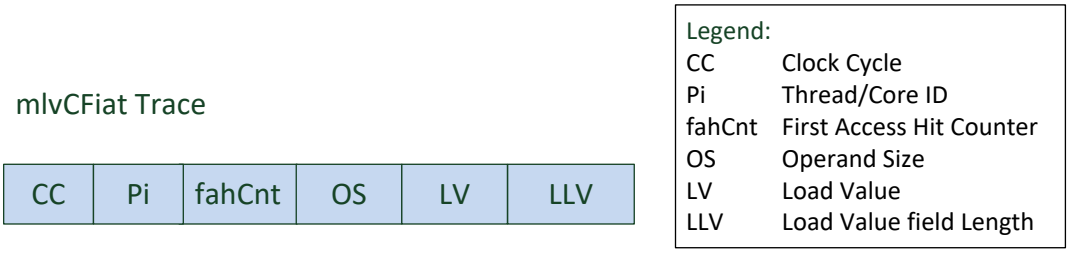

Figure 4.3 *mlvCFiat* trace descriptor format

```
; mlvCFiat: Instrumentation Time 180.608 ms
; timed memory read stats
Recorded 2197 memory read instructions.
    211 ( %9.60 ) Byte Operands
    11 ( %0.50 )  Word Operands
    1971 ( %89.71 ) Doubleword Operands
    2 ( %0.09 ) Quadword Operands
    2 ( %0.09 ) Extended Precision Operands
    0 ( %0.00 ) Octaword Operands
    0 ( %0.00 ) Others Operands
; timed memory write stats
Recorded 2213 memory write instructions.
    108 ( %4.88 ) Byte Operands
    166 ( %7.50 )  Word Operands
    1937 ( %87.53 ) Doubleword Operands
    1 ( %0.05 ) Quadword Operands
    1 ( %0.05 ) Extended Precision Operands
    0 ( %0.00 ) Octaword Operands
    0 ( %0.00 ) Others Operands
; timed memory read flow with CFiat stats
Cache Size (KB): 16
Cache Associativity: 4
Cache Line Size (B): 32
First Access Flag Granularity (B): 4

-- Cache Read References Hits:Misses (Hit Rate)
    Total 2080:117(94%)
    Byte Operands 177:34(83%)
    Word Operands 9:2(81%)
    Doubleword Operands 1891:80(95%)
    Quadword Operands 1:1(50%)
    Extended Precision Operands 2:0(100%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
-- Cache References Hits:Misses (Hit Rate)
    Total 4172:238(94%)
    Byte Operands 282:37(88%)
    Word Operands 175:2(98%)
    Doubleword Operands 3710:198(94%)
    Quadword Operands 2:1(66%)
    Extended Precision Operands 3:0(100%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
-- First Access Flag References Hits:Misses (Hit Rate)
    Total 1829:368(83%)
    Byte Operands 143:68(67%)
    Word Operands 8:3(72%)
    Doubleword Operands 1676:295(85%)
    Quadword Operands 1:1(50%)
    Extended Precision Operands 1:1(50%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
; File size in Binaries
; Type, TotalSizeofTime, TotalSizeofLine, TotalSize
Input Load, 17576, 35152, 52728
Input Store, 17704, 35408, 53112
Input, 35280, 70560, 105840
Output, 2944, 3312, 6256
```

Figure 4.4 *mlvCFiat* simulator statistics example

4.2.1 Implementation Details

Figure 4.5 shows the functional flow of the *mlvCFiat* simulator. It starts by reading the trace messages from the input file and by extracting the individual fields of the trace message. If the trace message is from a new thread, it creates a new private *mlvCFiat* structure as in Figure 3.2 for the corresponding thread. By using the operand address and the operand size, it decides whether the operand spans single cache block or multiple cache blocks. Depending on the operation and the number of cache blocks occupied by the operand, the following functions are invoked: memory read operation with single cache block invokes the *SingleCacheBlockLoadAnalysis()* function, memory write operation with single cache block invokes the *SingleCacheBlockStoreAnalysis()* function, memory read operation with multiple cache blocks invokes the *MultiCacheBlockLoadAnalysis()* function, and memory write operation with multiple cache blocks invokes the *MultiCacheBlockStoreAnalysis()* function.

SingleCacheBlockLoadAnalysis() checks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache hit event*, if all the corresponding *FA* bits are set, *fahCnt* is incremented. Even if a single *FA* bit is not set, it is treated as an *FA miss event*. For a *cache miss event* or *FA miss event*, a trace message for that thread is written to the output file, *fahCnt* is cleared and the corresponding *FA* bits are set (Figure 3.3a). To avoid duplication of sub-blocks protected by the *FA* bit reported by the target platform, *mlvCFiat* emits the trace message with the missing sub-blocks instead of all the sub-blocks corresponding to the requested data.

*SingleCacheBlockStoreAnalysis()*checks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache miss event*, a new

cache block is fetched and the *FA* bits for newly fetched cache block are cleared. For a *cache hit event*, the current processor acquires ownership by invalidating data in other processor caches. If the current write operation writes the entire sub-block protected by an *FA* bit, the corresponding *FA* bit is set (Figure 3.3b).

If the operand spans multiple cache blocks, each cache block is treated as an independent single cache block and operations are carried out as in *SingleCache-BlockStoreAnalysis()* and *SingleCacheBlockLoadAnalysis()*. If the data is found partially in the cache (some cache blocks are hit and some cache blocks are miss), it is treated as a *cache miss event*. Statistics related to caches (cache hit or cache miss, first access bits are hit or miss etc.), instructions (number of memory reads and memory writes), and operands (byte, word, double word etc.) are incremented. These steps are repeated until the output file exceeds the allocated size or the input file reaches its end.

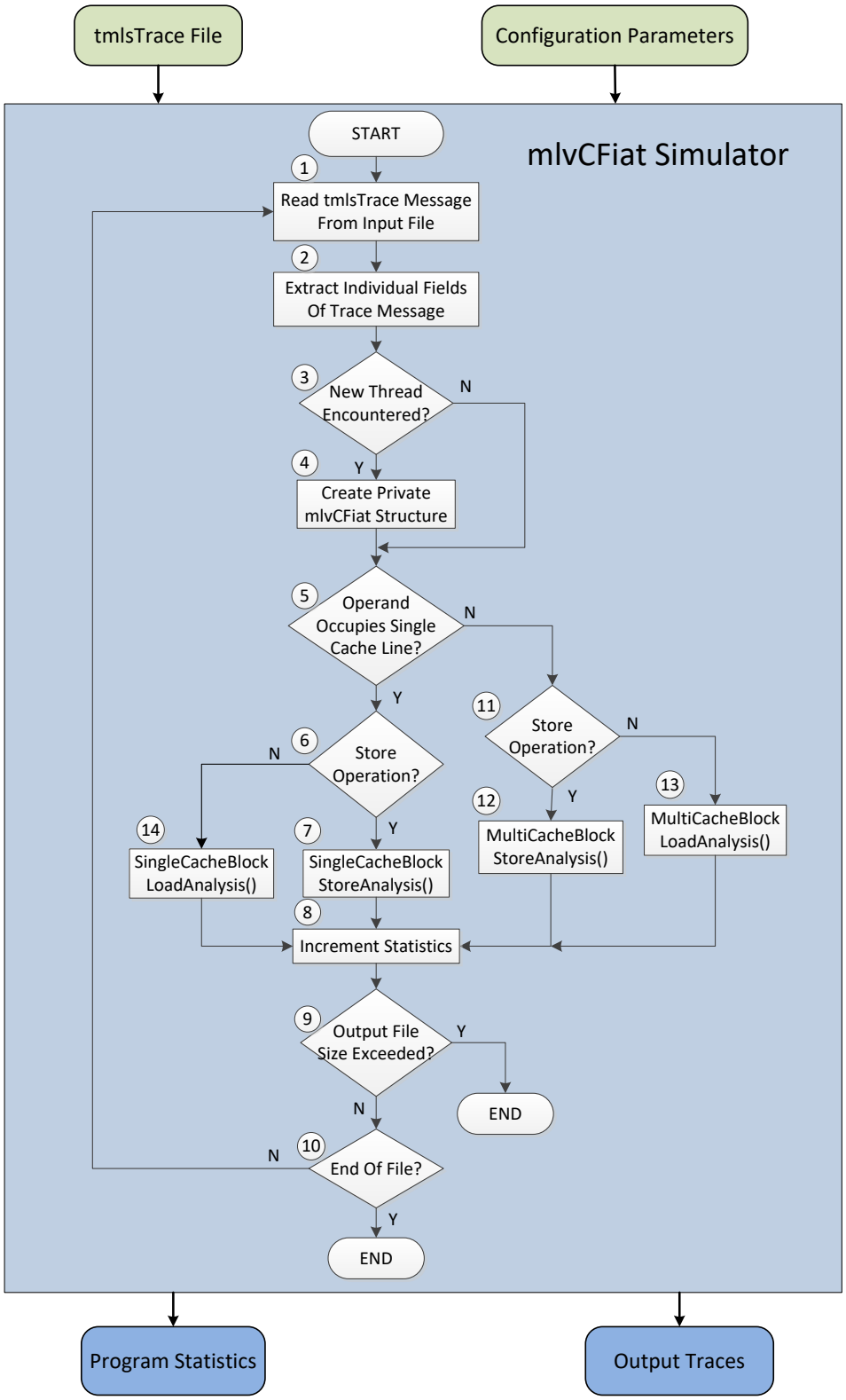

Figure 4.5 *mlvCFiat* simulator functional flow

4.2.2 Verification Details

To verify the correctness of the *mlvCFiat* simulator, we use test cases that cover functional flows of the simulator in the presence of memory reads, memory writes, and invalidations. Because the Intel32 ISA supports variable size operands in memory, our tests consider memory operations on operands residing in a single cache block as well as on operands that span multiple cache blocks. The tests also cover situations such as emitting only missed sub-blocks instead of all relevant sub-blocks, and setting an *FA* bit if the current write operation writes the whole sub-block. The output of the *mlvCFiat* simulator shown in this section includes additional details such as cache hit or miss, *FA* hit or miss, the location of the operand in the data cache, and possibly a trace message emitted for better understanding.

To ensure controlled conditions during verification, the input trace is not captured from a program, but rather it is manually crafted. The data cache configuration used for testing is as follows: 16KB data cache, 32-byte cache block, 4-way set associativity, first-access granularity size of 4-bytes, and LRU replacement policy. Thus, 5 LSB bits (0 to 4) of the operand address are used to identify the index of the operand in the corresponding cache block, 7 bits (5 to 11) are used to get the location of the cache block (set number) and the rest of the bits are used as a tag to differentiate blocks that map to the same set in the data cache. The *mlvCFiat* simulator does not model the data storage portion of the data cache. Thus, the *mlvCFiat* trace messages report the load value read from the *tmlsTraces*, rather than from the sub-blocks corresponding to the load value in the data cache.

Figure 4.6 shows a test input trace with reads and writes on operands occupying a single cache block. Each line of the input and expected output is described below:

- Line 1: The data requested by a memory read operation is not found in the cache (*cache read miss event*). Thus, the corresponding trace message is emitted as shown in line 17. As expected due to the granularity of the FA bits, the length of the *LV* field is 4-bytes instead of 1-byte.
- Lines 2 to 6: The data requested by memory read operations are not found in the cache (*cache read miss events*). The corresponding trace messages are emitted. The data blocks requested in these lines map to the same set of the data cache. Lines 2 to 5 fill all the ways of the set in the data cache. In line 6, a cache miss event from processor 0 is observed; the data block read from memory replaces the least recently used data block (the cache block fetched in line 2) in this set.
- Line 7: This is a *cache miss event* because the cache block corresponding to the requested data is evicted from the cache in line 6. Thus, a trace message corresponding to the load value in line 7 should be emitted (line 47).
- Line 8: The data requested by a memory read operation is found in the data cache (*cache read hit event*) and the corresponding *FA* bit is also set (*FA hit event*). Thus, no trace message is emitted.
- Line 9: The cache block requested by a memory write operation is not found in the cache (*cache write miss event*). Processor 1 replaces the cache block and invalidates the shared data in the processor 0 data

cache to acquire ownership. The current write operation writes the whole sub-block protected by the corresponding *FA* bit.

- Line 10: This is *a cache read miss event* because of the invalidation of the shared data in line 9. The corresponding trace message is emitted.
- Line 11: The data requested by a memory read operation is found in the data cache (*cache read hit event*) and the corresponding *FA* bit is also set in line 9 (*FA hit event*). Thus, no trace message is emitted.
- Line 12: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 0 reads the data block from memory and replaces the cache block. The current write operation writes to 3 sub-blocks. In these only 2 sub-blocks are written completely, thus the corresponding two *FA* bits are set.
- Line 13: This is a cache hit event. The data written in line 12 is read here. This is an *FA miss event* since all *FA* bits are not set. Thus, a trace message (line 69) corresponding to one sub-block is emitted.

```
a) Input to mlvCFiat simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1.    244, 0, 0,  8ba8123, b03ae0b2, 1, 9
2.    250, 0, 0,  8ba8132, b7fa00b8, 4, 97
3.    252, 0, 0,  8ba8134, a7fc00a4, 4, a3
4.    259, 0, 0,  8ba8135, b7fd00a4, 4, b90
5.    260, 0, 0,  8ba8136, 80ad0b4, 4, b60fa008
6.    270, 0, 0,  8ba8138, b0fd00a4, 4, ba
7.    279, 0, 0,  8ba8139, b7fa00b8, 4, 97
8.    299, 0, 0,  8ba813a,  80ad0b4, 4, b60fa008
9.    300, 1, 1,  8ba813b, b7fa00bc, 4
10.   310, 0, 0,  8ba813c, b7fa00bc, 4, 6013
11.   311, 1, 0,  8ba8140, b7fa00bc, 4, 6013
12.   319, 0, 1,  8ba8146, b06ad0aa, 10
13.   336, 0, 0,  8ba8147, b06ad0aa, 10, b7

b) Output of mlvCFiat simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
14.   Tag: b03ae  SetIndex: 5  LineIndex: 12
15.   CacheHit: 0 New wayIndex: 3
16.   Emit ? 1
17.   Emitted trace message: 244, 0, 0, 1, 9, 4
```

```
18.
19.   Tag: b7fa0  SetIndex: 5  LineIndex: 18
20.   CacheHit: 0 New wayIndex: 2
21.   Emit ? 1
22.   Emitted trace message: 250, 0, 0, 4, 97, 4
23.
24.   Tag: a7fc0  SetIndex: 5  LineIndex: 4
25.   CacheHit: 0 New wayIndex: 1
26.   Emit ? 1
27.   Emitted trace message: 252, 0, 0, 4, a3, 4
28.
29.   Tag: b7fd0  SetIndex: 5  LineIndex: 4
30.   CacheHit: 0 New wayIndex: 0
31.   Emit ? 1
32.   Emitted trace message: 259, 0, 0, 4, b90, 4
33.
34.   Tag: 80ad  SetIndex: 5  LineIndex: 14
35.   CacheHit: 0 New wayIndex: 3
36.   Emit ? 1
37.   Emitted trace message: 260, 0, 0, 4, b60fa008, 4
38.
39.   Tag: b0fd0  SetIndex: 5  LineIndex: 4
40.   CacheHit: 0 New wayIndex: 2
41.   Emit ? 1
42.   Emitted trace message: 270, 0, 0, 4, ba, 4
43.
44.   Tag: b7fa0  SetIndex: 5  LineIndex: 18
45.   CacheHit: 0 New wayIndex: 1
46.   Emit ? 1
47.   Emitted trace message: 279, 0, 0, 4, 97, 4
48.
49.   Tag: 80ad  SetIndex: 5  LineIndex: 14
50.   CacheHit: 1 Found in  wayIndex: 3  FAHit: 1
51.   Emit ? 0
52.
53.   Tag: b7fa0  SetIndex: 5  LineIndex: 1c
54.   CacheHit: 0 New wayIndex: 3
55.   Evicted cache block from processor 0 cache
56.
57.   Tag: b7fa0  SetIndex: 5  LineIndex 1c
58.   CacheHit: 0 New wayIndex: 1
59.   Emit ? 1
60.   Emitted trace message: 310, 0, 1, 4, 6013, 4
61.
62.   Tag: b7fa0  SetIndex: 5  LineIndex: 1c
63.   CacheHit: 1 Found in  wayIndex: 3  FAHit: 1
64.   Emit ? 0
65.
66.   Tag: b06ad  SetIndex: 5  LineIndex: a
67.   CacheHit: 0 New wayIndex: 0
68.
69.   Tag: b06ad  SetIndex: 5  LineIndex: a
70.   CacheHit: 1 Found in  wayIndex: 0  FAHit: 0
71.   Emit ? 1
72.   Emitted trace message: 336, 0, 0, 10, b7, 4
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.6 Testing *mlvCFiat*: single cache block access

Figure 4.7 shows the test input used to verify operation of the *mlvCFiat* simulator and its output when memory operands span multiple cache blocks. Each line of the input and expected outputs is described below:

- Line 1: The data requested by a memory read operation is not found in the cache (*cache read miss event*). Thus, the trace message corresponding to load value in line 1 is emitted as shown in line 18. As the operand spans two sub-blocks in two cache blocks, the length of the *LV* field in the trace message is 8-bytes instead of 4-bytes.
- Line 2: The data requested by a memory write operation is found in the cache (*cache write hit event*). Since the current write operation writes the whole sub-block, the corresponding *FA* bit is set.
- Line 3: The data requested by a memory read operation spans two cache blocks and it is partially hit, i.e. the cache block-1 is hit and the cache block-2 is miss. This event is treated as a *cache read miss event*. Therefore, a trace message for a sub-block in the cache block-2 is emitted, as shown in line 30.
- Lines 4 and 5: The data requested by memory read operations are not found in the cache (*cache read miss events*). Thus, trace messages corresponding to the reads are emitted, as shown in lines 39 and 48, respectively.
- Line 6: The data requested by a memory write operation is found in the cache (a *cache write hit event*). Processor 2 acquires the ownership by invalidating the cache blocks in processor 0 and processor 1. The

current memory write operation writes 2-bytes in cache block-1 and 6-bytes in cache block-2. Thus, it sets one *FA* bit in cache block-2.

- Line 7: Because of the invalidation in line 6, this is a *cache read miss event*. Thus, a trace message is emitted, as shown in line 68.
- Line 8: This is a *cache hit event*, but all corresponding *FA* bits are not set (*FA miss event*). The write operation in line 6 sets only a few of the *FA* bits, hence a trace message for the corresponding sub-blocks whose *FA* bit is not set is emitted.
- Line 9: The data requested by a memory write operation is not found in the cache (a *cache write miss event*). The current write operation writes to 3 sub-blocks, but only one sub-block is written completely. Hence, the *FA* bit corresponding to that sub-block is set.
- Line 10: This is a *cache read hit event*. A memory read operation reads three sub-blocks. Two sub-blocks have their FA bits set by the operation in line 9, thus a trace message for one sub-block is emitted.

```
a) Input to mlvCFiat simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1.    296, 0, 0,  8048132, bfff001e, 4, 4
2.    299, 0, 1,  8048133, bfff003c, 4
3.    307, 0, 0,  8048134, bfff003c, 8, 100
4.    308, 1, 0,  8048137, bfff003c, 8, 100
5.    319, 2, 0,  8048140, bfff003c, 8, 100
6.    328, 2, 1,  8048142, bfff003e, 8
7.    329, 1, 0,  8048143, bfff003d, 4, 5
8.    331, 2, 0,  8048145, bfff003e, 8, 5
9.    338, 0, 1,  8048159, b23f003a, 8
10.   340, 0, 0,  8048170, b23f003a, 10, c902

b) Output of mlvCFiat simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11.   Cache block: 1
12.   Tag: bfff0  SetIndex: 0  LineIndex: 1e
13.   LocalCacheHit: 0  New wayIndex: 3
14.   Cache block: 2
15.   Tag: bfff0  SetIndex: 1  LineIndex: 0
16.   LocalCacheHit: 0  New wayIndex: 3
17.   Emit ? 1
```

```
18.   Emitted trace message: 296, 0, 0, 4, 4, 8
19.
20.   Tag: bfff0  SetIndex: 1  LineIndex: 1c
21.   CacheHit: 1 Found in wayIndex: 3
22.
23.   Cache block: 1
24.   Tag: bfff0  SetIndex: 1  LineIndex: 1c
25.   LocalCacheHit: 1 Found in  wayIndex: 3 LocalFAHit: 1
26.   Cache block: 2
27.   Tag: bfff0  SetIndex: 2  LineIndex: 0
28.   LocalCacheHit: 0  New wayIndex: 3
29.   Emit ? 1
30.   Emitted trace message: 307, 0, 0, 8, 100, 4
31.
32.   Cache block: 1
33.   Tag: bfff0  SetIndex: 1  LineIndex: 1c
34.   LocalCacheHit: 0  New wayIndex: 3
35.   Cache block: 2
36.   Tag: bfff0  SetIndex: 2  LineIndex: 0
37.   LocalCacheHit: 0  New wayIndex: 3
38.   Emit ? 1
39.   Emitted trace message: 308, 1, 0, 8, 100, 8
40.
41.   Cache block: 1
42.   Tag: bfff0  SetIndex: 1  LineIndex: 1c
43.   LocalCacheHit: 0  New wayIndex: 3
44.   Cache block: 2
45.   Tag: bfff0  SetIndex: 2  LineIndex: 0
46.   LocalCacheHit: 0  New wayIndex: 3
47.   Emit ? 1
48.   Emitted trace message: 319, 2, 0, 8, 100, 8
49.
50.   Cache block: 1
51.   Tag: bfff0  SetIndex: 1  LineIndex: 1e
52.   LocalCacheHit: 1 Found in  wayIndex: 3
53.   Cache block: 2
54.   Tag: bfff0  SetIndex: 2  LineIndex: 0
55.   LocalCacheHit: 1 Found in  wayIndex: 3
56.   Cache block: 1 evicted from processor 0 cache
57.   Cache block: 2 evicted from processor 0 cache
58.   Cache block: 1 evicted from processor 1 cache
59.   Cache block: 2 evicted from processor 1 cache
60.
61.   Cache block: 1
62.   Tag: bfff0  SetIndex: 1  LineIndex: 1d
63.   LocalCacheHit: 0  New wayIndex: 3
64.   Cache block: 2
65.   Tag: bfff0  SetIndex: 2  LineIndex: 0
66.   LocalCacheHit: 0  New wayIndex: 3
67.   Emit ? 1
68.   Emitted trace message: 329, 1, 0, 4, 5, 8
69.
70.   Cache block: 1
71.   Tag: bfff0  SetIndex: 1  LineIndex: 1e
72.   LocalCacheHit: 1 Found in  wayIndex: 3 LocalFAHit: 1
73.   Cache block: 2
74.   Tag: bfff0  SetIndex: 2  LineIndex: 0
75.   LocalCacheHit: 1 Found in  wayIndex: 3 LocalFAHit: 0
76.   Emit ? 1
77.   Emitted trace message: 331, 2, 0, 8, 5, 4
78.
79.   Cache block: 1
80.   Tag: b23f0  SetIndex: 1  LineIndex: 1a
81.   LocalCacheHit: 0 New wayIndex: 3
82.   Cache block: 2
83.   Tag: b23f0  SetIndex: 2  LineIndex: 0
84.   LocalCacheHit: 0 New wayIndex: 3
85.
86.   Cache block: 1
87.   Tag: b23f0  SetIndex: 1  LineIndex: 1a
```

```
88.   LocalCacheHit: 1 Found in  wayIndex: 3 LocalFAHit: 0
89.   Cache block: 2
90.   Tag: b23f0  SetIndex: 2  LineIndex: 0
91.   LocalCacheHit: 1 Found in  wayIndex: 3 LocalFAHit: 0
92.   Emit ? 1
93.   Emitted trace message: 340, 0, 0, 10, c902, 8
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.7 Testing *mlvCFiat*: multi cache block access

4.3 *mc²RT* Simulator

The *mc²RT* simulator takes *tmlsTraces* as an input, implements the *mc²RT* trace filtering as described in Section 3.2, and outputs the compressed load data value traces. Section 4.3.1 discusses the main implementation details of the *mc²RT* simulator, and Section 4.3.2 describes the test cases used to confirm its correctness.

Table 4.1 (excluding --cfg flag) describes the parameters that are used to control the behavior of *mc²RT* simulator. With the help of these parameters, we can control the output file size, the size of the data caches, and the output file name. Figure 4.8 shows the output format of the *mc²RT* trace messages. A message, includes the time stamp (*CC*), the thread id (*Pi*), the trace hit counter (*THCnt*), the operand size (*OS*), the content of the corresponding data cache blocks that includes the load value (*CB*), and length of the *CB* field (*LCB*). Note: The length of the cache block field (*CB*) can be greater than the actual cache block size for example, for a 32-byte cache block, if the operand spans two cache blocks and misses in both of the cache blocks, then the length of *CB* will be 64-bytes. The *mc²RT* simulator also outputs program statistics as shown in Figure 4.9, including information about the number of memory read and memory write operations, cache hits, cache misses, trace bit hits, read hits in other caches, and invalidations etc.

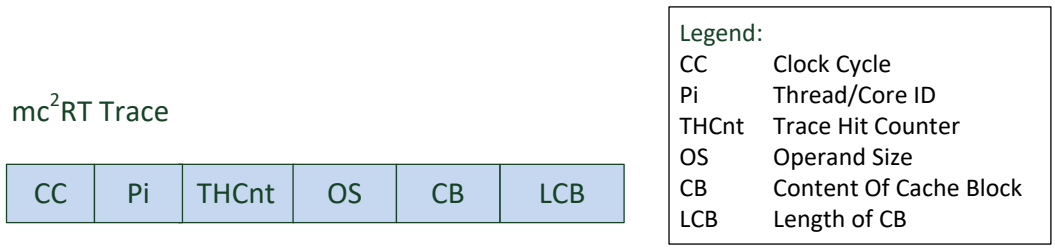

Figure 4.8 *mc²RT* trace descriptor format

```
; mc2RT: Instrumentation Time 840.872 ms
; timed memory read stats
Recorded 7902 memory read instructions.
    2279 ( %28.84 ) Byte Operands
    23 ( %0.29 )  Word Operands
    5600 ( %70.87 ) Doubleword Operands
    0 ( %0.00 ) Quadword Operands
    0 ( %0.00 ) Extended Precision Operands
    0 ( %0.00 ) Octaword Operands
    0 ( %0.00 ) Others Operands
; timed memory write stats
Recorded 30691 memory write instructions.
    125 ( %0.41 ) Byte Operands
    681 ( %2.22 )  Word Operands
    29885 ( %97.37 ) Doubleword Operands
    0 ( %0.00 ) Quadword Operands
    0 ( %0.00 ) Extended Precision Operands
    0 ( %0.00 ) Octaword Operands
    0 ( %0.00 ) Others Operands
; timed memory read flow with mc2RT stats
Cache Size (KB): 16
Cache Associativity: 4
Cache Line Size (B): 32

-- Cache Read References Hits:Misses (Hit Rate)
    Total 7698:204(97%)
    Byte Operands 2259:20(99%)
    Word Operands 22:1(95%)
    Doubleword Operands 5417:183(96%)
    Quadword Operands 0:0(0%)
    Extended Precision Operands 0:0(0%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
-- Cache References Hits:Misses (Hit Rate)
    Total 15039:23554(38%)
    Byte Operands 2375:29(98%)
    Word Operands 673:31(95%)
    Doubleword Operands 11991:23494(33%)
    Quadword Operands 0:0(0%)
    Extended Precision Operands 0:0(0%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
-- Trace bit References Hits:Misses (Hit Rate)
    Total 7588:314(96%)
    Byte Operands 2215:64(97%)
    Word Operands 22:1(95%)
    Doubleword Operands 5351:249(95%)
    Quadword Operands 0:0(0%)
```

```
    Extended Precision Operands 0:0(0%)
    Octaword Operands 0:0(0%)
    Hexaword Operands 0:0(0%)
    Other Sized Operands 0:0(0%)
-- Invalidation References :  Invalidations
    Total : 37
    Byte Operands : 0
    Word Operands : 0
    Doubleword Operands : 37
    Quadword Operands : 0
    Extended Precision Operands : 0
    Octaword Operands : 0
    Hexaword Operands : 0
    Other Sized Operands : 0
-- Read Hit in other Caches :  Hit
    Total : 83
    Byte Operands : 3
    Word Operands : 0
    Doubleword Operands : 80
    Quadword Operands : 0
    Extended Precision Operands : 0
    Octaword Operands : 0
    Hexaword Operands : 0
    Other Sized Operands : 0
-- Hit in other Caches :  Hit (read & write)
    Total : 23137
    Byte Operands : 3
    Word Operands : 0
    Doubleword Operands : 23134
    Quadword Operands : 0
    Extended Precision Operands : 0
    Octaword Operands : 0
    Hexaword Operands : 0
    Other Sized Operands : 0
; File size in Binaries
; Type, TotalSizeofTime, TotalSizeofLine, TotalSize
Input Load, 63216, 126432, 189648
Input Store, 245528, 491056, 736584
Input, 308744, 617488, 926232
Output, 2512, 2826, 5338
```

Figure 4.9 *mc²RT* simulator statistics example

4.3.1 Implementation Details

The *mc²RT* simulator has the same functional flow as the *mlvCFiat* simulator (Figure 4.5). It starts by reading the trace messages from the input file and by extracting the individual fields of the trace message. If the trace message is from a new thread, it creates a new private *mc²RT* structure as in Figure 3.6 for the corresponding thread. By using the operand address and size, it decides whether the operand spans single or multiple cache blocks. Depending on the operation and the

number of cache blocks occupied by the operand, the following functions are invoked: memory read operation with single cache block invokes the *SingleCache-BlockLoadAnalysis()* function, memory write operation with single cache block invokes the *SingleCacheBlockStoreAnalysis()* function, memory read operation with multiple cache blocks invokes the *MultiCacheBlockLoadAnalysis()* function, and memory write operation with multiple cache blocks invokes the *MultiCacheBlock-StoreAnalysis()* function.

SingleCacheBlockLoadAnalysis() checks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache miss event*, the coherent read transaction is issued, if the cache block hits in other caches the trace bit is also transferred along with the cache block. If a trace bit is not set for the new cache block or if the cache block is retrieved from main memory, a trace message for that thread is written to the output file, the trace bit is set and *THCnt* is cleared. For a *cache hit event*, if the trace bit is set *THCnt* is incremented (Figure 3.8).

SingleCacheBlockStoreAnalysis() checks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache miss event*, a new cache block is fetched either from another processors cache or from main memory. If the cache block is fetched from another cache, the corresponding trace bit is also inherited; if the cache is retrieved from main memory, the trace bit for the newly fetched cache block is cleared. For a *cache hit event*, the current processor acquires the ownership by invalidating the cache block in the other processor's cache (Figure 3.11).

If the operand spans multiple cache blocks, each cache block is treated as an independent single cache block and operations are performed as in *SingleCache-*

BlockStoreAnalysis() and *SingleCacheBlockLoadAnalysis()*. If the data is found partially in the cache (some cache blocks are hit and some cache blocks are miss), it is treated as a *cache miss event*. Statistics related to caches (cache hit or cache miss, trace bit hit or trace bit miss etc.), instructions (number of memory reads and memory writes), and operands (byte, word, double word etc.) are incremented. These steps are repeated until the output file exceeds the allocated size or the input file reaches its end.

4.3.2 Verification Details

To verify the correctness of the *mc²RT* simulator, we use two test cases that cover functional flows of the simulator in the presence of memory reads, memory writes, and state transitions. The tests employ memory accesses that use different data types, span single or multiple cache blocks, and exercise different state transitions in the cache-coherence protocol. The output of the *mc²RT* simulator shown in this section includes additional details along with trace messages such as cache hit or miss, trace bit hit or miss, the location of the operand in the data cache, data is hit in other caches or not, and trace message status and content. In the *mc²RT* output, snoop signal *X* indicates the coherent read and invalidate (*CRI*), snoop signal *R* indicates the coherent read (*CR*), snoop signal *I* indicates coherent invalidate (*CI*), and snoop signal *N* indicates no action.

To ensure controlled conditions during verification, the input trace is not captured from a program, but rather it is manually crafted. The data cache configurations used in these tests match those used in the *mlvCFiat* simulator verification. The *mc²RT* simulator does not model the data storage portion of the data cache.

Thus, the *mc²RT* trace messages report the load value read from the *tmlsTraces*, rather than entire cache blocks corresponding to the load value in the data cache.

Figure 4.10 shows the test input used to verify the operation of the *mc²RT* simulator and its output when the operand occupies a single cache block. Each line of the input and expected output is described below.

- Line 1: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 1 generates snoop signal *X* to get the data. Since data does not hit in any other processor, the cache block corresponding to the requested data is loaded from memory and the state of the cache block is set to *Modified*.
- Line 2: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 2 generates snoop signal *X* to get the data. The data hits in the processor 1 cache, thus it transfers the requested cache block and corresponding *TR* bit attached to the cache block to processor 2. The cache block corresponding to the load value is not reported before, thus trace message is emitted as shown in line 20 and the trace bit in both the caches is set.
- Line 3: This operation results a *cache write miss event*. Processor 3 generates the snoop signal *X* to get the data. The requested data hits in processors1 and 2. Since processor 1 is in *Owned* state, it is responsible for sending the requested data along with the trace bit to processor 3.
- Line 4 and 5: The memory read operation in these lines results in a *cache read miss event*, but the cache block can be found in processor 3.

The *TR* bit of the corresponding load values is set (t*race hit event*). Hence, a trace message corresponding to this cache block in processor 0 and processor 2 is not emitted.

- Line 6: This line illustrates *cache read hit* and *trace hit* events. Thus, a trace message is not emitted corresponding to this cache block.
- Line 7: This is a *cache write hit event*. Since this is a shared block, processor 2 generates snoop signal *I* to invalidate the cache block in other caches to acquire the ownership.
- Line 8: The data requested by a memory write operation is not found in the data cache because this cache block is invalidated by processor 2 in line 7. Processor 3 generates snoop signal *X* to get the data. The data can be found in processor 2, thus it transfers the cache block and its *TR* bit to processor 1. Since the *TR* bit is already set, a trace message corresponding to this cache block is not emitted.
- Line 9: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 0 generates snoop signal *R* to get the data. Since this data is not found in any other processor, it is fetched from memory and the state of the cache block is set as *Exclusive*. The trace message corresponding to this cache block is emitted as shown in line 54.
- Line 10: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 2 generates snoop signal *R* to get the data. The data can found in processor 0, it sends the requested cache block along with its *TR* bit. Processor 0 already

emitted a trace message corresponding to this cache block in line 9, thus it is not emitted. The states of the two processors are updated to *Shared*.

- Line 11: The data hits in processors 0 and 2 in the *Shared* state. Even though the data hits in two processors only, one processor is responsible for sending the requested data to processor 3. In this design, the lower numbered processor is responsible to send the requested data.

```
a) Input to mc²RT simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1.    296, 1, 1,  8048132, bfff0000, 4
2.    297, 2, 0,  8048134, bfff0000, 8, 1b
3.    299, 3, 1,  8048135, bfff0000, 4
4.    315, 0, 0,  8048138, bfff0000, 2, 1f9
5.    319, 2, 0,  804813a, bfff0000, 4, 1f9
6.    321, 3, 0,  804813b, bfff0000, 4, 1f9
7.    351, 2, 1,  8048142, bfff0000, 1
8.    352, 1, 0,  8048145, bfff0000, 4, 98c
9.    358, 0, 0,  8048144, a0d0ac00, 4, 7f
10.   377, 2, 0,  8048149, a0d0ac00, 4, 7f
11.   383, 3, 0,  8048150, a0d0ac00, 4, 7f

b) Output of mc²RT simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12.   Tag: bfff0  SetIndex: 0  LineIndex: 0
13.   CacheHit: 0 Snoop Signal: X
14.   Found in other cache: 0
15.
16.   Tag: bfff0  SetIndex: 0  LineIndex: 0
17.   CacheHit: 0 Snoop Signal: R
18.   Found in other cache: 1 TRHit in other cache: 0
19.   Emit: 1
20.   Emitted trace message: 297, 2, 0, 8, 1b, 32
21.
22.   Tag: bfff0  SetIndex: 0  LineIndex: 0
23.   CacheHit: 0 Snoop Signal: X
24.   Found in other cache: 1
25.
26.   Tag: bfff0  SetIndex: 0  LineIndex: 0
27.   CacheHit: 0 Snoop Signal: R
28.   Found in other cache: 1 TRHit in other cache: 1
29.   Emit: 0
30.
31.   Tag: bfff0  SetIndex: 0  LineIndex: 0
32.   CacheHit: 0 Snoop Signal: R
33.   Found in other cache: 1 TRHit in other cache: 1
34.   Emit: 0
35.
36.   Tag: bfff0  SetIndex: 0  LineIndex: 0
37.   CacheHit: 1 Found in wayIndex: 3
38.   State: o TRHit: 1
39.   Emit: 0
40.
41.   Tag: bfff0  SetIndex: 0  LineIndex: 0
```

```
42.   CacheHit: 1 Found in wayIndex: 3
43.   State: s Snoop Signal: I
44.
45.   Tag: bfff0  SetIndex: 0  LineIndex: 0
46.   CacheHit: 0 Snoop Signal: R
47.   Found in other cache: 1 TRHit in other cache: 1
48.   Emit: 0
49.
50.   Tag: a0d0a  SetIndex: 60  LineIndex: 0
51.   CacheHit: 0 Snoop Signal: R
52.   Found in other cache: 0
53.   Emit: 1
54.   Emitted trace message: 358, 0, 1, 4, 7f, 32
55.
56.   Tag: a0d0a  SetIndex: 60  LineIndex: 0
57.   CacheHit: 0 Snoop Signal: R
58.   Found in other cache: 1 TRHit in other cache: 1
59.   Emit: 0
60.
61.   Tag: a0d0a  SetIndex: 60  LineIndex: 0
62.   CacheHit: 0 Snoop Signal: R
63.   Found in other cache: 1 TRHit in other cache: 1
64.   Emit: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.10 Testing *mc²RT*: single cache block access

Figure 4.11 shows the test input used to verify operation of the *mc²RT* simulator and filtered output traces when memory operands span multiple cache blocks. Each line of the input and expected outputs is described below:

- Line 1: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 0 generates a snoop signal *R* to get the data. None of the processors has the requested data, thus it is read from memory and the new state of the cache block is set as *Exclusive*. This cache block is not reported by any other processor, thus a corresponding trace message is emitted.
- Line 2: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 0 generates a snoop signal *X* for two cache blocks separately. Since the cache block

does not hit in any other processor, it is read from memory and the state of the cache block is updated to *Modified* for two cache blocks.

- Line 3: This is a *cache read hit event,* processor 0 reads the same cache blocks accessed in line 2. The *TR* bit for two cache blocks is not hit (*trace miss event*), thus the trace message corresponding to these cache blocks is emitted with the size of *CB* field as 64 bytes as shown in line 34.
- Line 4: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 1 generates a snoop signal *R* separately for two cache blocks. This data hits in processor 0, thus it transfers the data along with its trace bit. The *TR* bit is hit for two cache lines (reported in line 3), thus no trace message is emitted.
- Line 5: The data requested by a memory write operation is found in the data cache (*cache write miss event*). Processor 0 acquires ownership by invalidating the cache block in other caches. It generates the snoop signal *I* for two cache blocks separately to invalidate the shared data.
- Line 6: The cache block is invalidated by processor 0 in line 5; thus it is a *cache read miss event*. Processor 1 generates a snoop signal *R* for two cache blocks separately. Processor 0 transfers the requested data along with its trace bit to processor 1. The *TR* bit for both the cache blocks is set (*trace hit event*), hence a trace message is not emitted.

- Line 7: The data requested by a memory read operation is partially found in the data cache (*cache read miss event*). Cache block-1 is hit and the corresponding *TR* bit is also set and cache block-2 is miss. Processor 0 generates a snoop signal *R* corresponding to cache block-2 to get the data. The data does not hit in any other processor; hence it is read from memory and a trace message with the content of the cache block-2 is emitted. Thus, the length of the *CB* field is 32-bytes instead of 64-bytes (line 80).
- Lines 8: This is a *cache write miss event*. Processor 2 generates a snoop signal *X* for two cache blocks separately to get the data. Cache block-1 can be found in processor 0 but cache block-2 is not found in any other processor.

```
a) Input to mc²RT simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1.    209, 0, 0, 8048192, 818a0d4, 4, b7ef0088
2.    242, 0, 1, 8048195, b7ef00b8, 10
3.    249, 0, 0, 8048199, b7ef00b8, 10, 1
4.    256, 1, 0, 8048232, b7ef00ba, 8, 1
5.    263, 0, 1, 8048239, b7ef00ba, 8
6.    270, 1, 0, 8048240, b7ef00ba, 8, b7e3008
7.    271, 0, 0, 8048242, b7ef00de, 8, b7e3008
8.    272, 2, 1, 8048242, b7ef00fe, 8

b) Output of mc²RT simulator
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
9.    Tag: 818a  SetIndex: 6  LineIndex: 14
10.   CacheHit: 0 Snoop Signal: R
11.   Found in other cache: 0
12.   Emit: 1
13.   Emitted trace message: 209, 0, 0, 4, b7ef0088, 32
14.
15.   Cache Block: 1
16.   Tag: b7ef0  SetIndex: 5  LineIndex: 18
17.   LocalCacheHit: 0 Snoop Signal: X
18.   Found in other cache: 0
19.   Cache Block: 2
20.   Tag: b7ef0  SetIndex: 6  LineIndex: 0
21.   LocalCacheHit: 0 Snoop Signal: X
22.   Found in other cache: 0
23.
24.   Cache Block: 1
25.   Tag: b7ef0  SetIndex: 5  LineIndex: 18
26.   LocalCacheHit: 1 Found in wayIndex: 3
```

```
27.   State: m LocalTRHit: 0
28.   Cache Block: 2
29.   Tag: b7ef0  SetIndex: 6  LineIndex: 0
30.   LocalCacheHit: 1 Found in wayIndex: 2
31.   State: m LocalTRHit: 0
32.   Emit: 1
33.   Emitted trace message: 249, 0, 0, 10, 1, 64
34.
35.   Cache Block: 1
36.   Tag: b7ef0  SetIndex: 5  LineIndex: 1a
37.   LocalCacheHit: 0
38.   Cache Block: 2
39.   Tag: b7ef0  SetIndex: 6  LineIndex: 0
40.   LocalCacheHit: 0
41.   Cache Block: 1 Snoop Signal: R
42.   Found in other cache: 1 TRHit in other cache: 1
43.   Cache Block: 2 Snoop Signal: R
44.   Found in other cache: 1 TRHit in other cache: 1
45.   Emit: 0
46.
47.   Cache Block: 1
48.   Tag: b7ef0  SetIndex: 5  LineIndex: 1a
49.   LocalCacheHit: 1 Found in wayIndex: 3
50.   State: o Snoop Signal: I
51.   Cache Block: 2
52.   Tag: b7ef0  SetIndex: 6  LineIndex: 0
53.   LocalCacheHit: 1 Found in wayIndex: 2
54.   State: o Snoop Signal: I
55.
56.   Cache Block: 1
57.   Tag: b7ef0  SetIndex: 5  LineIndex: 1a
58.   LocalCacheHit: 0
59.   Cache Block: 2
60.   Tag: b7ef0  SetIndex: 6  LineIndex: 0
61.   LocalCacheHit: 0
62.   Cache Block: 1 Snoop Signal: R
63.   Found in other cache: 1 TRHit in other cache: 1
64.   Cache Block: 2 Snoop Signal: R
65.   Found in other cache: 1 TRHit in other cache: 1
66.   Emit: 0
67.
68.   Cache Block: 1
69.   Tag: b7ef0  SetIndex: 6  LineIndex: 1c
70.   LocalCacheHit: 1 Found in wayIndex: 2
71.   State: o LocalTRHit: 1
72.   Cache Block: 2
73.   Tag: b7ef0  SetIndex: 7  LineIndex: 0
74.   LocalCacheHit: 0
75.   Cache Block: 1 Snoop Signal: N
76.   Found in other cache: 1 TRHit in other cache: 1
77.   Cache Block: 2 Snoop Signal: R
78.   Found in other cache: 0
79.   Emit: 1
80.   Emitted trace message: 271, 0, 0, 8, b7e3008, 32
81.
82.   Cache Block: 1
83.   Tag: b7ef0  SetIndex: 7  LineIndex: 1c
84.   LocalCacheHit: 0 Snoop Signal: X
85.   Found in other cache: 1
86.   Cache Block: 2
87.   Tag: b7ef0  SetIndex: 8  LineIndex: 0
88.   LocalCacheHit: 0 Snoop Signal: X
89.   Found in other cache: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.11 Testing *mc²RT*: multi cache block access

4.4 mc²RFiat Simulator

The *mc²RFiat* simulator takes *tmlsTraces* as input, implements the *mc²RFiat* trace filtering as described in Section 3.3, and outputs the compressed load data value traces. Section 4.4.1 discusses some main implementation details of the *mc²RFiat* simulator and Section 4.4.2 describes the test cases used to confirm the correctness of the *mc²RFiat* simulator.

Table 4.1 describes the parameters that can be used to control the behavior of the *mc²RFiat* simulator. With the help of these parameters, we can control output file size, the size of the data caches, and the output file name. The output format of the *mc²RFiat* trace message is the same as the *mlvCFiat* trace message (Figure 4.3) and the format of the output statistics file is the same as the *mc²RT* statistics file (Figure 4.9).

4.4.1 Implementation Details

The *mc²RFiat* simulator has the same functional flow as the *mlvCFiat* simulator (Figure 4.5). It starts by reading the trace messages from the input file and by extracting the individual fields of the trace message. If the trace message is from a new thread, it creates a new private *mc²RFiat* structure as in Figure 3.14 for the corresponding thread. By using the operand address and the operand size, it decides whether the operand spans single or multiple cache blocks. Depending on the operation and the number of cache blocks occupied by the operand, the following functions are invoked: memory read operation with single cache block invokes the *SingleCacheBlockLoadAnalysis()* function, memory write operation with single cache block invokes the *SingleCacheBlockStoreAnalysis()* function, memory read operation with multiple cache blocks invokes the *MultiCacheBlockLoadAnalysis()* function,

and memory write operation with multiple cache blocks invokes the *MultiCache-BlockStoreAnalysis()* function.

SingleCacheBlockLoadAnalysis() checks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache hit event*, if all the corresponding *FA* bit(s) is set, *fahCnt* is incremented. Even if a single *FA* bit is not set, it is treated as an *FA miss event*. For an *FA miss event*, a trace message for that thread is written to the output file, *fahCnt* is cleared and the corresponding *FA* bit(s) is set. For a *cache miss event*, the coherent read transaction is issued; if the cache block hits in other caches, the *FA* bits for the corresponding cache block are also transferred along with the cache block. If the corresponding *FA* bit(s) is not set or if the cache block is retrieved from main memory, a trace message for that thread is written to the output file, the corresponding *FA* bit(s) is set and *fahCnt* is cleared. To avoid duplication of sub-blocks protected by the *FA* bit reported by the target platform, *mlvCFiat* emits the trace message with the missing sub-blocks instead of all the sub-blocks corresponding to the requested data (Figure 3.16).

*SingleCacheBlockStoreAnalysis() c*hecks whether the requested data is found in the cache (*cache hit event*) or not (*cache miss event*). For a *cache miss event*, a new cache block is fetched from either another processor's cache or from main memory. If the cache block is fetched from another cache, the corresponding *FA* bits are also inherited; if the cache is retrieved from main memory, the *FA* bits for the newly fetched cache block are cleared. For a *cache hit event*, the current processor acquires ownership by invalidating the cache block in the other caches. If the current write operation writes the entire sub-block protected by a cache's *FA* bit, the corresponding *FA* bit(s) is set (Figure 3.19).

If the operand spans multiple cache blocks, each cache block is treated as an independent single cache block and operations are performed as in *SingleCache-BlockStoreAnalysis()* and *SingleCacheBlockLoadAnalysis()*.If the data is found partially in the cache (some cache blocks are hit and some cache blocks are miss), it is treated as a *cache miss event*. Statistics related to caches (cache hit or cache miss, invalidations, hits in other caches, first access bits are hit or miss etc.), instructions (number of memory reads and memory writes), and operands (byte, word, double word etc.) are incremented. These steps are repeated until the output file exceeds the allocated size or the input file reaches its end.

4.4.2 Verification Details

To verify the correctness of the *mc²RFiat* simulator, we use test cases that cover functional flows of the simulator in the presence of memory reads, memory writes, and state transitions. The tests employ memory accesses that use different data types, span single or multiple cache blocks, and exercise different state transitions in cache-coherence protocol. The output of the *mc²RFiat* simulator shown in this section includes additional details such as cache hit or miss, *FA* hit or miss, the location of the operand in the cache, data hit or miss in other caches, and trace message status and content. In the *mc²RFiat* output, snoop signal *X* indicates the coherent read and invalidate (*CRI*), snoop signal *R* indicates the coherent read (*CR*) and snoop signal *I* indicates coherent invalidate (*CI*), snoop signal *N* indicates no action.

To ensure controlled conditions during verification, the input trace is not captured from a program, but rather is manually crafted. The data cache configurations used matches ones used in verification of the *mlvCFiat* simulator. The *mlvCFiat* simulator does not model data storage portion of the data cache. Thus, the *mc²RFiat*

trace messages report the load value read from the *tmlsTraces*, rather than from the sub-blocks corresponding to the load value in the data cache.

Figure 4.12 shows the test input used to verify operation of the *mc²RFiat* simulator and its output when the memory operand occupies a single cache block. Each line of the input and expected output is described below.

- Line 1: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 0 generates snoop signal *X* to get the missing data. The data is not found in any processor, thus it is read from memory and the state of the cache block is updated to *Modified*. The current memory write operation does not write the whole sub-block protected by *FA* bit. Thus, no corresponding *FA* bit is set.
- Line 2: This is a *cache hit event* but the *FA* bits for the corresponding sub-blocks are not set (*FA miss event*). Thus, a corresponding trace message is emitted as shown in line 9. The length of the *LV* field is 8-bytes since the operand spans two sub-blocks. This illustrates the point that the *LV* field length can be greater than the actual operand size.
- Line 3: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 1 generates a snoop signal *R* to get the missing data. The data hits in processor 0, hence it sends the requested data and its *FA* bits. The load value in this memory read operation spans three sub-blocks but two sub-blocks are

already reported by processor 0 hence a trace message with one sub-block only is emitted as shown in line15.

- Line 4: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 2 generates a snoop signal *R* to get the missing data. The data hits in processor 0 in the *Owned* state and processor 1 in the *Shared* state. Thus, processor 0 is responsible for sending the requested cache block and its *FA* bits. A trace message for one sub-block is emitted as shown in line 21.
- Line 5: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). Processor 0 generates a snoop signal *R* to get the missing data. Since data does not hit in any other processor, processor 0 reads data from memory and the state of the cache block is updated to *Exclusive*. The corresponding *FA* bits load value are not set (*FA miss event*), thus a trace message is emitted for the corresponding sub-block (line 27).
- Line 6: This is a *cache miss event*. Processor 1 generates a snoop signal *R* to get the missing data. The data hits in processor 0 in the *Exclusive* state. Processor 0 sends the requested data and its *FA* bits. The current operand spans two sub-blocks and one of the sub-block is already reported in line 5, hence the trace message is emitted only for one sub-block. The state of both processors is updated to *Shared*.
- Line 7: This is a *cache miss event*. Processor 2 generates a snoop signal *R* to get the missing data. Data hits in processors 0 and 1 in *Shared* state. The requested data can be transferred by processor 0 or proces-

sor 1, it is a design consideration. In this simulator, lower numbered processor is responsible for sending the cache block i.e. processor 0 sends the requested data and its *FA* bits. The *FA* bit is set only for one sub-block (*FA miss event*), thus a trace message is emitted for another sub-block.

- Line 8: This is a *cache hit event* but the corresponding *FA* bit is not set (*FA miss event*). A trace message corresponding to the load value as shown in line 45 is emitted. The length of the *LV* field in trace message is 4-bytes instead of 1-byte due to the granularity of the FA bits.
- Line 9: This is a *cache hit event* but only a few *FA* flags corresponding to the load value are set (*FA miss event*). Thus, a trace message is emitted only for sub-blocks that miss. The length of the *LV* field in the emitted trace message (line 51) is less than the actual operand size
- Line 10: The data requested by a memory write operation is found in the data cache (*cache write hit event*). Processor 3 generates a snoop signal *I* to acquire ownership. The *FA* bit is set since the current memory write operation writes the entire sub-block protected by the *FA* bit.
- Line 11: This is a *cache miss event* because of the invalidation in line 10. The *FA* bit corresponding to the load value is set in line 10, thus a trace message is not emitted.

a) Input to *mc²RFiat* simulator

```
1.    258, 0, 1,  80492ba, b7ed60e2, 4
2.    317, 0, 0,  80492bc, b7ed60e2, 4, 3f8edcf
3.    319, 1, 0,  80492bd, b7ed60e2, 8, 3f8edcf
4.    320, 2, 0,  80492c4, b7ed60e2, 8, 3f8edcf
5.    333, 0, 0,  80493ca, b0ed60f4, 4, 2
6.    373, 1, 0,  80493cb, b0ed60f6, 4, 2
7.    387, 2, 0,  80493cd, b0ed60f6, 4, 2
8.    393, 0, 0,  80493e0, b7ed60ed, 1, 9
9.    394, 0, 0,  80493e1, b7ed60e2, 10, 2
10.   399, 3, 1,  80493e2, b7ed60fc, 4
11.   400, 0, 0,  80493e4, b7ed60fc, 4, 2
```

b) Output of *mc²RFiat* simulator

```
1.    Tag: b7ed6  SetIndex: 7  LineIndex: 2
2.    CacheHit: 0 Snoop Signal: X
3.    Found in other cache: 0
4.    Tag: b7ed6  SetIndex: 7  LineIndex: 2
5.    CacheHit: 1 Found in wayIndex: 3
6.    State: m FAHit: 0
7.    Emit: 1
8.    Emitted trace message: 317, 0, 0, 4, 3f8edcf, 8
9.
10.   Tag: b7ed6  SetIndex: 7  LineIndex: 2
11.   CacheHit: 0 Snoop Signal: R
12.   Found in other cache: 1 FAHit in other cache: 0
13.   Emit: 1
14.   Emitted trace message: 319, 1, 0, 8, 3f8edcf, 4
15.
16.   Tag: b7ed6  SetIndex: 7  LineIndex: 2
17.   CacheHit: 0 Snoop Signal: R
18.   Found in other cache: 1 FAHit in other cache: 0
19.   Emit: 1
20.   Emitted trace message: 320, 2, 0, 8, 3f8edcf, 4
21.
22.   Tag: b0ed6  SetIndex: 7  LineIndex: 14
23.   CacheHit: 0 Snoop Signal: R
24.   Found in other cache: 0
25.   Emit: 1
26.   Emitted trace message: 333, 0, 0, 4, 2, 4
27.
28.   Tag: b0ed6  SetIndex: 7  LineIndex: 16
29.   CacheHit: 0 Snoop Signal: R
30.   Found in other cache: 1 FAHit in other cache: 0
31.   Emit: 1
32.   Emitted trace message: 373, 1, 0, 4, 2, 4
33.
34.   Tag: b0ed6  SetIndex: 7  LineIndex: 16
35.   CacheHit: 0 Snoop Signal: R
36.   Found in other cache: 1 FAHit in other cache: 0
37.   Emit: 1
38.   Emitted trace message: 387, 2, 0, 4, 2, 4
39.
40.   Tag: b7ed6  SetIndex: 7  LineIndex: d
41.   CacheHit: 1 Found in wayIndex: 3
42.   State: o FAHit: 0
43.   Emit: 1
44.   Emitted trace message: 393, 0, 0, 1, 9, 4
45.
46.   Tag: b7ed6  SetIndex: 7  LineIndex: 2
47.   CacheHit: 1 Found in wayIndex: 3
48.   State: o FAHit: 0
49.   Emit: 1
50.   Emitted trace message: 394, 0, 0, 10, 2, 4
51.
52.   Tag: b7ed6  SetIndex: 7  LineIndex: 1c
53.   CacheHit: 0 Snoop Signal: X
```

```
54.   Found in other cache: 1
55.
56.   Tag: b7ed6  SetIndex: 7  LineIndex: 1c
57.   CacheHit: 0 Snoop Signal: R
58.   Found in other cache: 1 FAHit in other cache: 1
59.   Emit: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.12 Testing *mc²RFiat*: single cache block access

Figure 4.13 shows the test input used to verify the operation of the *mc²RFiat* simulator and filtered output traces when memory operands span multiple cache blocks. The verification will be the same as that for single cache lines except for a few cases. Each line of the input and expected output is described below.

- Line 1: The data requested by a memory read operation is not found in the data cache (*cache read miss event*). The load value spans two cache blocks, processor 0 generates a snoop signal *R* for each cache block separately to get the missing data. These cache blocks do not hit in any other processor, hence it reads from memory, thus the *FA* bits are also not set (*FA miss event*). A trace message corresponding to the load value which includes sub-blocks from two cache blocks is emitted. The state of the two cache blocks is updated to *Exclusive*.
- Line 2: This is a *cache read miss event*. Processor 1 generates a snoop signal *R* to get the missing data for two cache blocks separately. The data hits in processor 0, thus it transfers the cache block and its *FA* bits. The *FA* bit for one of the sub-blocks in cache block-1 is not set, thus a trace message is emitted for the corresponding sub-block. The state of the two cache blocks in processors 0 and 1 are updated to *Shared*.

- Line 3: This is a *cache read miss event*. Processor 2 generates a snoop signal *R* to get the missing data for two cache blocks separately. The data hits in processors 0 and 1, but processor 0 sends the requested data and its *FA* bits to processor 2. The *FA* bit for one of the sub-block in cache block-1 is not set, thus a trace message is emitted for the corresponding sub-block.
- Line 4: The data requested by a memory write operation is not found in the data cache (*cache write miss event*). Processor 3 generates a snoop signal *X* for each cache block separately. Cache block-1 hits in processor 0 in *Shared* state, thus processor 0 transfers cache block-1 and its *FA* bits. Cache block 2 does not hit in any processor, thus it is read from memory. The state of the both cache blocks is updated to *Modified*.
- Line 5: This is a *cache read miss event*. Processor 0 generates snoop signal *R* for each cache block separately to get the data. The requested data and its *FA* bits are read from processor 3. All the *FA* bits corresponding to the load value are not set (*FA miss event*), hence a trace message for the sub-blocks which are miss is emitted.
- Line 6: This is a *cache read miss event*. Processor 1 generates a snoop signal *R* for each cache block separately to get the data. The requested data hits in processors 3 and 0. Since processor 3 is in *Owned* state, it is responsible for transfering the requested data and its *FA* bits to processor 1. All the *FA* bits corresponding to the load value are not set (*FA miss event*), hence a trace message is emitted.

- Line 7: This is a *cache read hit* and all the corresponding *FA* bits are set (*FA hit event*), thus no trace message emitted.
- Line 8: The operand is partially hit i.e. cache block-1 is a miss and cache block-2 is a hit. This event is considered as a *cache write miss event*. Processor 3 generates a snoop signal *X* for cache block-1 and a snoop signal *I* to cache block-2. The *FA* bits for the few sub-blocks which are written completely are set.
- Line 9: The operand is partially hit i.e. cache block-1 is a miss (invalidated by processor 3 in line 8) and cache block-2 is a hit. Processor 1 generates a snoop signal *R* for cache block-1. The requested data hits in processor 3 in *Modified* state; hence it transfers the requested data along with its *FA* bit to processor 1. All the *FA* bits are set (*FA hit event*), thus a trace message is not emitted.

a) Input to *mc²RFiat* simulator

```
1.    218, 0, 0,  80663bd, b7fb00bc, 8, 40
2.    219, 1, 0,  80663be, b7fb00b8, 10, 140
3.    220, 2, 0,  80663c0, b7fb00b8, 10, 140
4.    239, 3, 1,  80663c2, b7fb00d2, 20
5.    254, 1, 0,  80663c5, b7fb00d4, 20, 755
6.    255, 0, 0,  80663c6, b7fb00cc, 20, 30
7.    258, 1, 0,  80663c9, b7fb00d4, 20, 755
8.    260, 3, 1,  80663cc, b7fb00bd, 8
9.    262, 1, 0,  80663d0, b7fb00d4, 20, 755
```

b) Output of *mc²RFiat* simulator

```
1.    Cache Block: 1
2.    Tag: b7fb0  SetIndex: 5  LineIndex: 1c
3.    LocalCacheHit: 0
4.    Cache Block: 2
5.    Tag: b7fb0  SetIndex: 6  LineIndex: 0
6.    LocalCacheHit: 0
7.    Cache Block: 1 Snoop Signal: R
8.    Found in other cache: 0
9.    Cache Block: 2 Snoop Signal: R
10.   Found in other cache: 0
11.   Emit: 1
12.   Emitted trace message: 218, 0, 0, 8, 40, 8
13.
14.   Cache Block: 1
```

```
15.   Tag: b7fb0  SetIndex: 5  LineIndex: 18
16.   LocalCacheHit: 0
17.   Cache Block: 2
18.   Tag: b7fb0  SetIndex: 6  LineIndex: 0
19.   LocalCacheHit: 0
20.   Cache Block: 1 Snoop Signal: R
21.   Found in other cache: 1 FAHit in other cache: 0
22.   Cache Block: 2 Snoop Signal: R
23.   Found in other cache: 1 FAHit in other cache: 1
24.   Emit: 1
25.   Emitted trace message: 219, 1, 0, 10, 140, 4
26.
27.   Cache Block: 1
28.   Tag: b7fb0  SetIndex: 5  LineIndex: 18
29.   LocalCacheHit: 0
30.   Cache Block: 2
31.   Tag: b7fb0  SetIndex: 6  LineIndex: 0
32.   LocalCacheHit: 0
33.   Cache Block: 1 Snoop Signal: R
34.   Found in other cache: 1 FAHit in other cache: 0
35.   Cache Block: 2 Snoop Signal: R
36.   Found in other cache: 1 FAHit in other cache: 1
37.   Emit: 1
38.   Emitted trace message: 220, 2, 0, 10, 140, 4
39.
40.   Cache Block: 1
41.   Tag: b7fb0  SetIndex: 6  LineIndex: 12
42.   LocalCacheHit: 0 Snoop Signal: X
43.   Found in other cache: 1
44.   Cache Block: 2
45.   Tag: b7fb0  SetIndex: 7  LineIndex: 0
46.   LocalCacheHit: 0 Snoop Signal: X
47.   Found in other cache: 0
48.
49.   Cache Block: 1
50.   Tag: b7fb0  SetIndex: 6  LineIndex: 14
51.   LocalCacheHit: 0
52.   Cache Block: 2
53.   Tag: b7fb0  SetIndex: 7  LineIndex: 0
54.   LocalCacheHit: 0
55.   Cache Block: 1 Snoop Signal: R
56.   Found in other cache: 1 FAHit in other cache: 1
57.   Cache Block: 2 Snoop Signal: R
58.   Found in other cache: 1 FAHit in other cache: 0
59.   Emit: 1
60.   Emitted trace message: 254, 1, 0, 20, 755, 4
61.
62.   Tag: b7fb0  SetIndex: 6  LineIndex: c
63.   CacheHit: 0 Snoop Signal: R
64.   Found in other cache: 1 FAHit in other cache: 0
65.   Emit: 1
66.   Emitted trace message: 255, 0, 0, 20, 30, 8
67.
68.   Cache Block: 1
69.   Tag: b7fb0  SetIndex: 6  LineIndex: 14
70.   LocalCacheHit: 1 Found in wayIndex: 3
71.   State: s LocalFAHit: 1
72.   Cache Block: 2
73.   Tag: b7fb0  SetIndex: 7  LineIndex: 0
74.   LocalCacheHit: 1 Found in wayIndex: 3
75.   State: s LocalFAHit: 1
76.   Emit: 0
77.
78.   Cache Block: 1
79.   Tag: b7fb0  SetIndex: 5  LineIndex: 1d
80.   LocalCacheHit: 0 Snoop Signal: X
81.   Found in other cache: 1
82.   Cache Block: 2
83.   Tag: b7fb0  SetIndex: 6  LineIndex: 0
84.   LocalCacheHit: 1 Found in wayIndex: 3
```

```
85.   State: o Snoop Signal: I
86.
87.   Cache Block: 1
88.   Tag: b7fb0  SetIndex: 6  LineIndex: 14
89.   LocalCacheHit: 0
90.   Cache Block: 2
91.   Tag: b7fb0  SetIndex: 7  LineIndex: 0
92.   LocalCacheHit: 1 Found in wayIndex: 3
93.   State: s LocalFAHit: 1
94.   Cache Block: 1 Snoop Signal: R
95.   Found in other cache: 1 FAHit in other cache: 1
96.   Cache Block: 2 Snoop Signal: N
97.   Found in other cache: 1 FAHit in other cache: 0
98.   Emit: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 4.13 Testing *mc²RFiat*: multi-cache block access

4.5 Software to Hardware Trace Translation

We developed custom tools that can translate software traces to hardware traces and evaluate the trace port bandwidth in bits per executed instruction and bits per clock cycle. The software debugger does not require all the fields of *tmlsTrace*, *mlvCFiat*, *mc²RT,* and *mc²RFiat* trace messages. Some of the fields can be inferred by the software debugger from the program binary with the help of an instruction set simulator. Thus, the software traces generated by the simulator are translated to hardware traces by eliminating redundant trace messages and redundant fields of the trace messages that can be inferred by the software debugger. Figure 4.1 shows the experimental environment used to create software traces and translation of software traces to hardware traces.

Every memory read results in a trace message in Nexus-like load data value traces. Thus, the custom tool filters *tmlsTraces* and eliminates the trace messages for memory writes and some of the redundant fields of *tmlsTraces* such as *OA*, *PC*, and *OS* as they can be inferred by the instruction set simulator from the program

binary. This custom tool also encodes the filter traces and writes to a binary file to analyze the trace port bandwidth. We refer to these traces as the Nexus-like load data value traces (*NX_b*).

mlvCFiat and *mc²RFiat* share the same format for output trace messages as shown in Figure 4.3. The *mc²RT* trace message format is shown in Figure 4.8. A custom tool filters the output of these simulators to discard the fields *OS* and *LLV/LCB* as they can be inferred by the software debugger and encodes the trace messages using fixed chunk sizes and variable chunk sizes. Encoded trace messages are written to a binary file for the trace port bandwidth analysis.

mlvCFiat traces are encoded using fixed chunk sizes (referred to as *CF_b*) and using variable chunk sizes (referred to as *CF_e*). *mc²RT* traces are encoded using fixed chunk sizes (referred to as *RT_b*) and using variable chunk sizes (referred to as *RT_e*). *mc²RFiat* traces are encoded using fixed chunk sizes (referred to as *RF_b*) and variable chunk sizes (referred to as *RF_e*).

Figure 4.14 shows the format of the trace messages for *NX_b, CF_b, CF_e, RT_b, RT_e, RF_b,* and *RF_e*. The time field (*Pi.dCC*), carries the information about the clock cycle in which the current trace generating instruction is retired. It reports a number of clock cycles expired since the last trace message is reported on core *i* instead of an absolute clock cycle from the beginning of the program, i.e. $Pi.dCC = Pi.CC - Pi.PCC$, $Pi.PCC = Pi.CC$. The length of the load value field in *NX_b* depends on the size of the operand read from memory. For the IA32 instruction set architecture it may vary from 1 to 120 bytes. The length of the *LV* field in *CF_b*, *CF_e*, *RF_b*, and *RF_e* depends on the first-access granularity size. For example, if the first-

access granularity size is 4-bytes and the operand size is 1 byte, the length of the *LV* field will be at least 4-bytes instead of 1 byte.

The number of bits needed to represent values of the *dCC*, *fahCnt,* and *THCnt* fields in Figure 4.14 is a function of benchmarks behavior and it may vary from benchmark to benchmark and within a single benchmark as we move to the different phases of the program execution. In base encoding, these fields are divided into 8-bit chunks and a connect bit (*C*). If the value of *dCC* or *fahCnt* or *THCnt* can fit in 8-bits then we will have one 8-bit chunk and a connect bit of zero (*C*=0) to indicate the end of that field. To reduce the redundant bits when using 8-bit chunks, we employ variable encoding of these three fields where the size of individual chunks may vary. Finding good chunk sizes for variable encoding is a part of our experimental evaluation. With our experimental evaluation, we found that the chunk sizes listed in Table 4.5 work well for all the benchmarks.

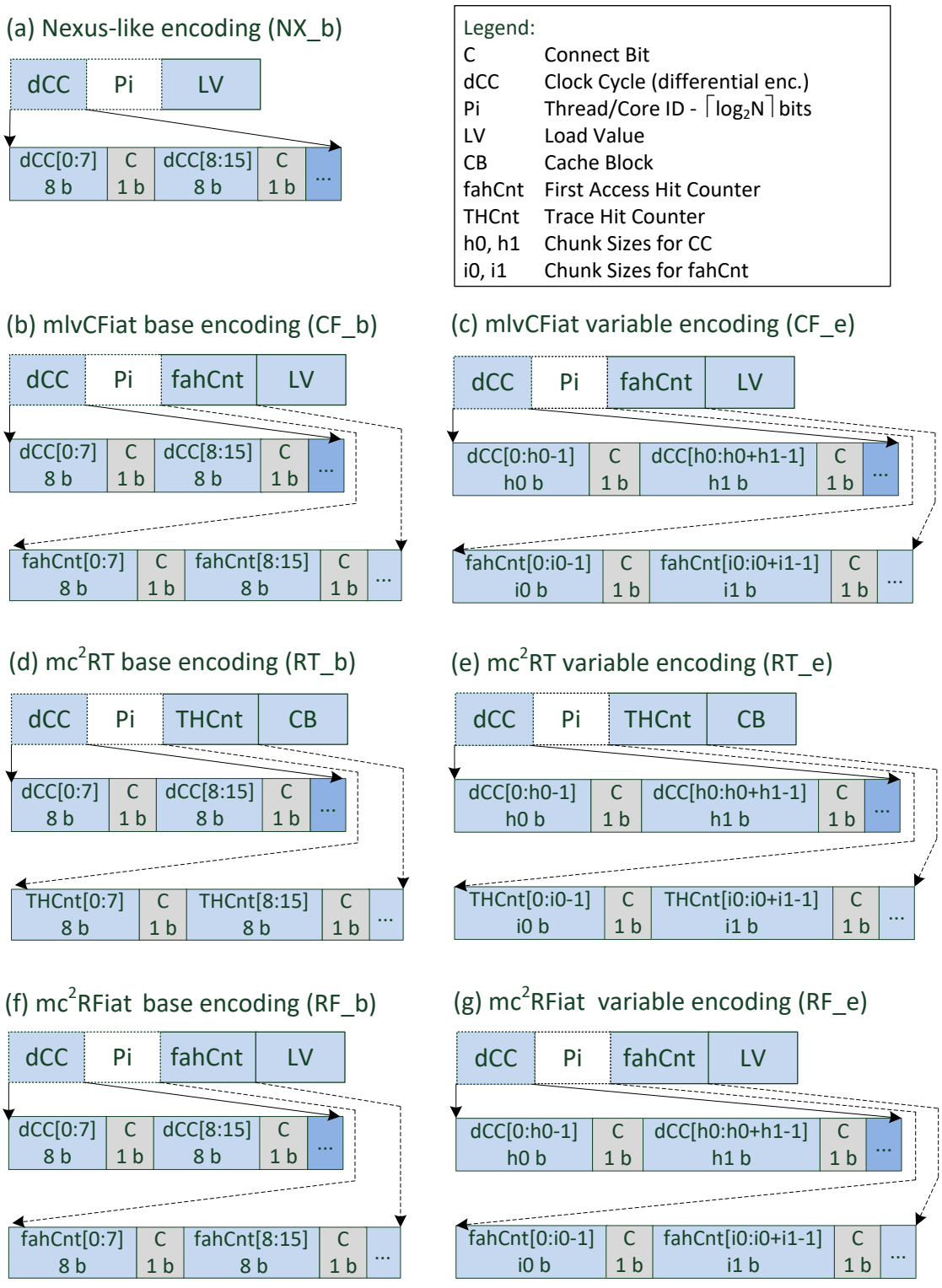

Figure 4.14 Formats of trace messages for *NX_b*, *CF_b*, CF*_e, RT_b, RT_e, RF_b and RF_e*

4.6 Experimental Environment

The main goal of the experimental evaluation is to determine the effectiveness of the newly proposed techniques *mlvCFiat*, *mc²RT*, and *mc²RFiat* for filtering load data value traces relative to the baseline Nexus-like load data value traces (*NX_b*) as a function of a number of cores. In addition, the goal is to quantitatively assess the impact of configuration parameters (cache sizes, granularity sizes) and encoding parameters (baseline and variable) on its performance. As a measure of effectiveness, we use average trace port bandwidth measured in bits per instruction and bits per clock cycle. As a workload, we use 10 benchmarks from the Splash2 [38] benchmark suite. More details about the benchmarks are given in Section 4.6.2. As a part of the experimental evaluation, we also determine good granularity size (Section 4.6.4) and encoding parameters (Section 4.6.5) that work well across all benchmarks.

4.6.1 Experimental Setup

The Multi2Sim [37] simulator supports building a cycle-accurate model for a multicore processor including processor and memory hierarchy. The multicore model we used is shown in Figure 4.15 with up to 8 single-threaded x86 processor cores. Each core has private level 1 instruction (*L1I*) and data caches (*L1D*). To evaluate the effectiveness of our proposed techniques as a function of cache size, we consider three cache configurations: *CS16* with 16KB *L1D*, *CS32* with 32KB *L1D*, and *CS64* with 64KB *L1D*. *L1I* cache sizes match the *L1D* cache size. The hit latency for a level 1 cache is 4 clock cycles. The unified level 2 (*L2*) cache is shared by all cores and has a hit latency of 12 clock cycles. The size of the *L2* cache depends on the number of cores, *N*, and it is set to *N*•64KB for the *CS16* configuration, *N*•128KB for the *CS32*

configuration, and N•256KB for the *CS64* configuration. The cache block size is set to 32 bytes for all cache configurations and for all cache levels. The *L1D* and *L1I* caches are 4-way set associative and the *L2* cache is 16-way set associative with the least-recently-used (*LRU*) replacement policy. The latency of main memory is 100 clock cycles. The interconnection networks between the *L1* and *L2* caches and the *L2* and main memory are identical in buffer size and bandwidth. The experiments are conducted on a Dell PowerEdge T620 server. It has two sockets, each with octa-core Intel Xeon CPU E5-2650 v2 processors having a total of 64GB physical memory and capable of running two threads.

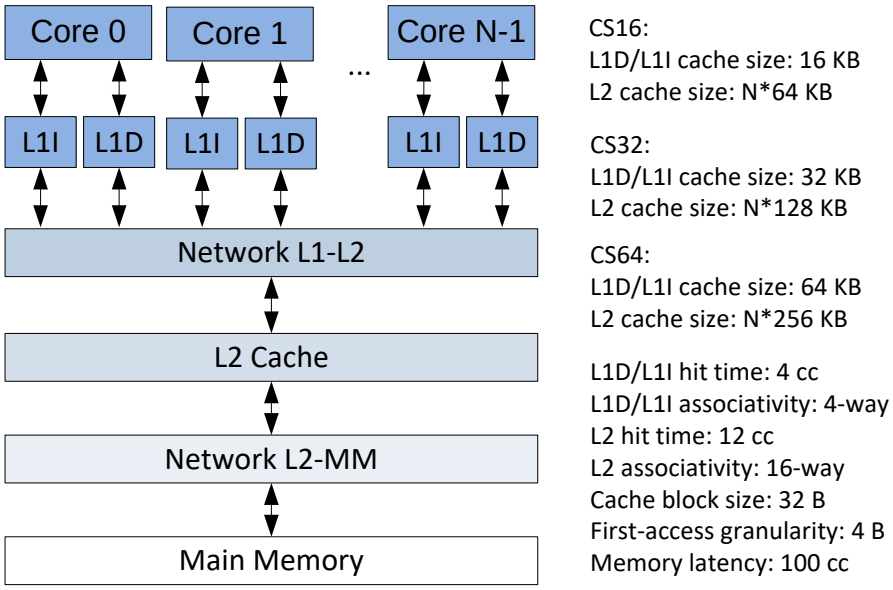

Figure 4.15 Multicore model in Mult2Sim

4.6.2 Benchmarks

To measure the effectiveness of the proposed techniques we use 10 benchmarks from the Splash2 benchmark suite [38] [39]. The benchmarks are precompiled for the IA32 instruction set architecture and run on a cycle-accurate Multi2Sim simulator that models multicores with N=1, 2, 4, and 8 cores.

This suite provides six different sets of input files to run a given benchmark namely *native*, *simdev*, *simlarge*, *simmedium*, *simsmall* and *test*. The *test* and *simdev* inputs are very small input sets and cannot be used for performance measurements. The *native* input is a very large input set intended for large-scale experiments on real machines. Since we are using a cycle accurate simulator for architectural simulations, the *simsmall* input set is used.

Table 4.2 shows the number of instructions executed by a benchmark (*IC*) in billions, instructions per clock cycle (*IPC*), and frequency of memory reads. The instruction count remains the same or slightly increases as the number of cores increases. The average number of instructions executed per clock cycle depends on the type of benchmarks, multicore models, and the number of cores. Thus, when N=1, the IPC ranges from 0.19 for *cholesky* to 0.66 for *water-sp*. The total *IPC* for the entire benchmark suite is calculated by dividing the sum of all instructions executed by all benchmarks with the sum of all execution times measured in clock cycles for all benchmarks. It ranges from 0.4 for N=1 to 1.95 for N=8. *IPC* as a function of the number of cores indicates how well performance scales. The frequency of instructions reading data from memory varies from 13.02% for *fmm* to 35.09% for *radix* when N=1 and from 13.48% for *fmm* to 35.09% for *radix* when N=8. The average frequency of instructions reading data from memory for the entire benchmark suite is

calculated by dividing the sum of all instructions reading data from memory for all benchmarks with the sum of all executed instructions from all benchmarks. It ranges from 22.77% when *N*=1 to 23.67% when *N*=8. It increases slightly with an increase in the number of cores because of the increased memory reads due to synchronization in multicores.

Table 4.2 Splash2 benchmark suite characterization

Benchmarks	*Instruction Count [x10^9]*				*Instructions Per Cycle [IPC]*				*% Loads*			
No. of Cores	*N=1*	*N=2*	*N=4*	*N=8*	*N=1*	*N=2*	*N=4*	*N=8*	*N=1*	*N=2*	*N=4*	*N=8*
barnes	2.13	2.13	2.13	2.14	0.37	0.54	0.96	1.69	28.78	28.78	28.78	28.79
cholesky	1.27	1.43	1.95	3.07	0.19	0.41	0.92	2.12	27.78	29.54	30.32	31.30
fft	0.92	0.92	0.92	0.92	0.26	0.44	0.72	1.04	19.20	19.20	19.20	19.21
fmm	2.79	2.80	2.82	2.86	0.41	0.80	1.52	2.70	13.02	13.06	13.27	13.49
lu	0.45	0.45	0.45	0.45	0.39	0.74	1.27	1.95	20.20	20.22	20.25	20.31
radiosity	2.23	2.33	2.29	2.32	0.48	0.87	1.65	2.99	27.51	27.45	27.38	26.79
radix	1.59	1.59	1.59	1.60	0.23	0.36	0.54	0.65	35.09	35.09	35.09	35.09
raytrace	2.47	2.46	2.47	2.47	0.50	0.93	1.68	2.67	28.49	28.48	28.48	28.47
water-ns	0.74	0.74	0.74	0.75	0.61	1.17	2.22	3.90	16.31	16.33	16.36	16.42
water-sp	5.03	5.03	5.03	5.03	0.66	1.07	1.73	2.73	17.38	17.38	17.38	17.38
Total	19.61	19.87	20.39	21.60	0.40	0.69	1.21	1.95	22.77	22.96	23.21	23.67

The average trace port bandwidth depends on the frequency of memory reads and also on the size of the operand accessed from memory. Table 4.3 shows a frequency of memory reads for different sizes of the operand. Very few benchmarks (*radix*, *fft*) read operands less than 4-bytes. For *barnes*, 60% of the memory reads are 4-bytes, ~37% are 8-bytes long, and the frequency of memory read instructions are also high. Thus, the average trace port bandwidth is also high.

Table 4.3 Characterization of memory reads in Splash2

Bench-marks	*Total Memory reads*	*% Byte operand*	*% Word operand*	*% Double-word operand*	*% Quard-word operand*	*% Extended precision operand*	*% Octa-word operand*	*% Others*
barnes	613093875	0	3.26	60.1	36.65	0	0	0
cholesky	352542470	1.33	0	54.09	44.59	0	0	0
fft	176252532	0.01	9.52	41.16	49.31	0	0	0
fmm	362804834	0	0.15	16.3	83.55	0	0	0
lu	90032066	0	2.04	41.77	56.19	0	0	0
radiosity	613309897	0	0	90.61	9.39	0	0	0
radix	558023567	4.51	29.31	57.16	9.02	0	0	0
raytrace	702412760	0.8	0.96	58.93	39.3	0	0	0
water-ns	120913006	0.6	0.01	23.2	76.19	0	0	0
water-sp	874383440	0.55	0.02	22.63	76.8	0	0	0
Total	4463768447	0.92	4.70	50.25	44.14	0	0	0

4.6.3 Experiments

Table 4.4 list the experiments conducted using different techniques with different cache configurations. The effectiveness of *mlvCFiat*, *mc²RT*, *mc²RFiat* is measured relative to Nexus-like load data value traces (*NX_b*) while varying the number of cores *N*=1, 2, 4 and 8. To analyze the effectiveness of *mlvCFiat*, *mc²RT*, and *mc²RFiat,* we considered following the cache configurations:

- CS16: 16KB data cache size
- CS32: 32KB data cache size
- CS64: 64KB data cache size

All cache configurations are 4-way set associative, use least recently used (LRU) replacement policy to update cache blocks and use 32 byte cache blocks.

Table 4.4 Experiments conducted

Technique	Method	CS16	CS32	CS64
Nexus-like (N = 1,2,4 & 8)	*NX_b*		✓	
mlvCFiat (N = 1,2,4 & 8)	*CF_b*	✓	✓	✓
	CF_e	✓	✓	✓
mc²RT (N = 1,2,4 & 8)	*RT_b*	✓	✓	✓
	RT_e	✓	✓	✓
mc²RFiat (N = 1,2,4 & 8)	*RF_b*	✓	✓	✓
	RF_e	✓	✓	✓

4.6.4 Granularity Study

The effectiveness of *mlvCFiat* and *mc²RFiat* depends on the first-access granularity size. Figure 4.16 shows the total average trace port bandwidth normalized to GS(4) as a function of the number of cores and first-access granularity for the *CS64* configuration for *mlvCFia*t. We consider granularity sizes of 1-byte GS(1), 4-bytes GS(4), 8-bytes GS(8), 16-bytes GS(16), and 32-bytes GS(32) for our evaluation. GS(1) slightly reduces (1 to 2%) the total average trace port relative to GS(4) for *mlvCFiat*. However, GS(1) quadruples the number of fist-access bits relative to GS(4): GS(1) requires 32 bits, whereas GS(4) requires 8 bits. As the granularity size increases, the trace port bandwidth increases, but the hardware overhead due to fist-access bits decreases. A similar analysis with all cache configurations (*CS16*, *CS32*, and *CS64*) shows that a granularity of size 4-bytes works well for all cache configurations. When we analyze the benchmarks individually, some benchmarks benefit from lower granularity and some benchmarks benefit from higher granulari-

ty. However, our main goal is to find the granularity size that works well for all benchmarks.

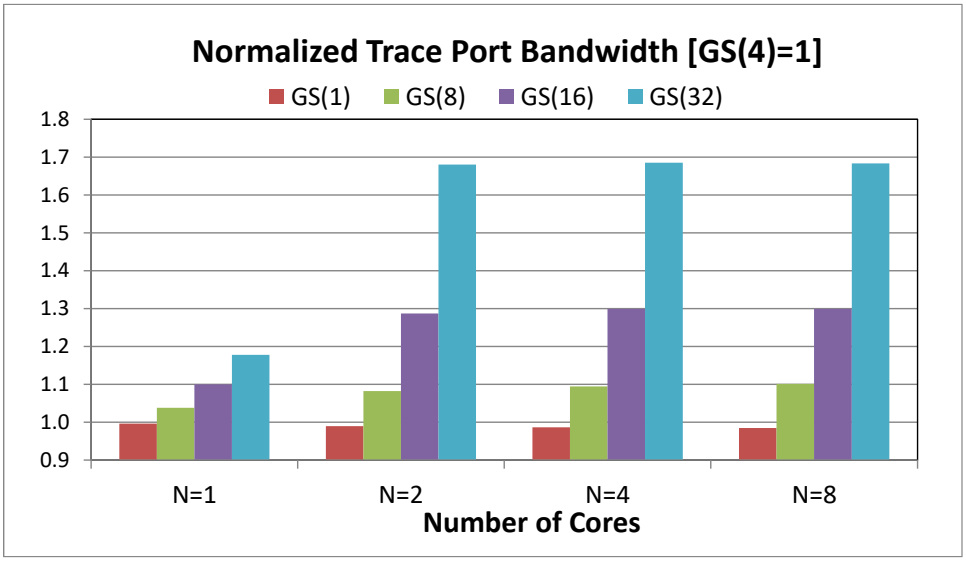

Figure 4.16 Normalized trace port bandwidth as a function of first-access granularity for *mlvCFiat*

Figure 4.17 shows the total average trace port bandwidth normalized with GS(4) as a function of the number of cores and first-access granularity for the *CS64* configuration for *mc²RFiat*. Our analysis shows that a granularity of size 4-bytes works well for all cache configurations.

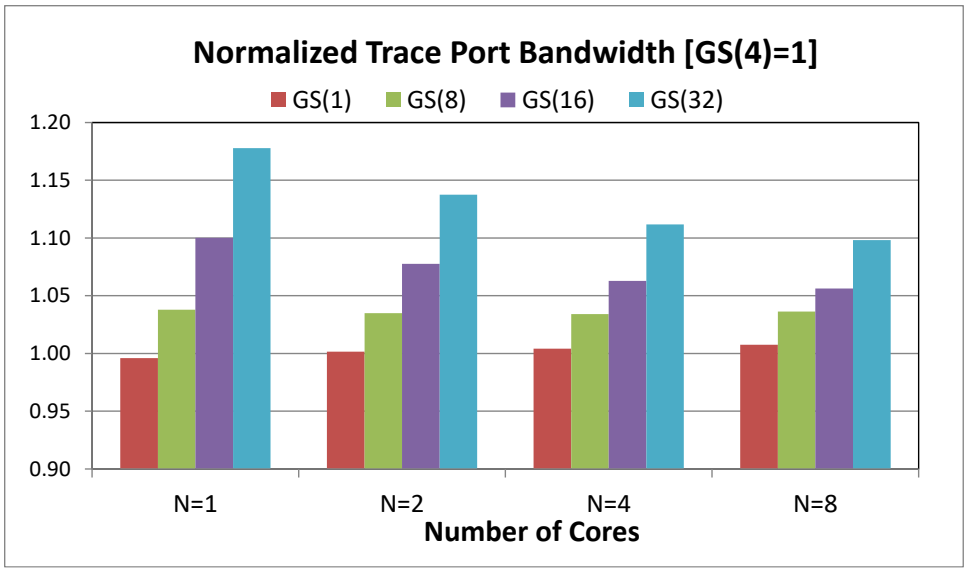

Figure 4.17 Normalized trace port bandwidth as a function of first-access granularity for *mc²RFiat*

4.6.5 Variable Encoding

To evaluate the significance of encoding mechanisms, we analyze the trace port bandwidth for both base and variable encoding mechanisms. To select the best encoding parameters, we profiled the Splash2 benchmarks to determine the minimum required bit lengths for the *dCC* and *fahCnt* fields. Figure 4.18 shows the cumulative distribution function of the *fahCnt* when *N*=1 for the *CS16* configuration for *mlvCFiat*. Most of the benchmarks require less than 3 bits for encoding the *fahCnt* field 60% of the time and require less than 7 bits more than 90% of the time. Figure 4.19 shows the cumulative distribution function of *dCC* length when *N*=1 for the *CS16* configuration for the *mlvCFiat* technique. As the graph shows, most of the benchmarks require less than 6 bits for encoding *dCC* field 60% of the time and re-

quire less than 10 bits more than 90% of the time. The profiles of CDF functions indicate that variable encoding could be beneficial in further reducing the size of trace fields.

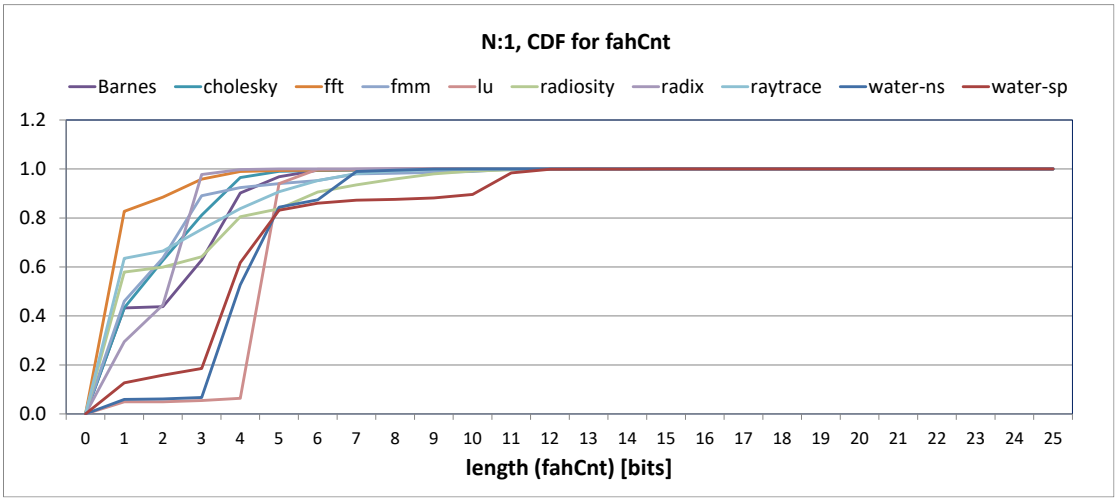

Figure 4.18 CDF of the minimum length for *fahCnt* for *mlvCFiat*

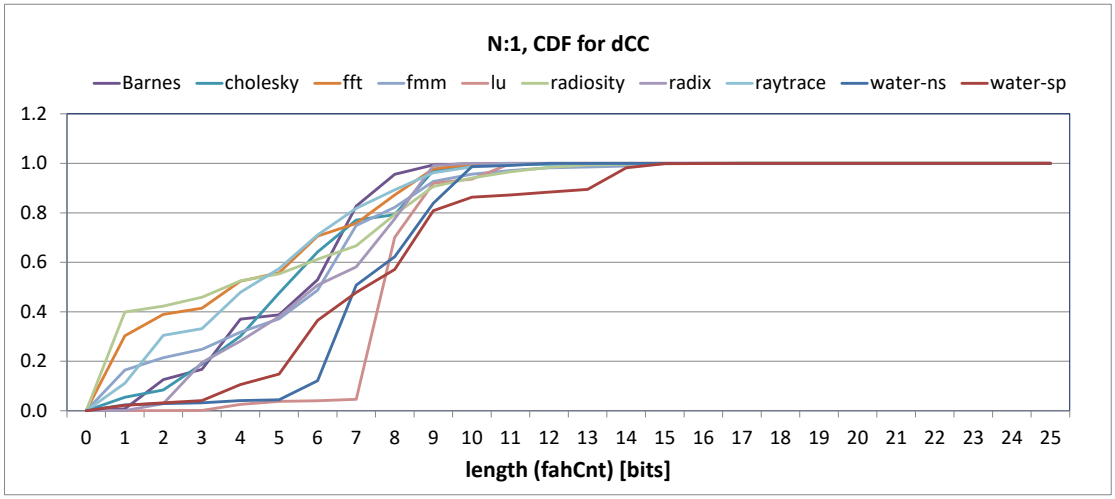

Figure 4.19 CDF of the minimum length for *dCC* for *mlvCFiat*

We could consider finding good chunk sizes for each benchmark separately, but here we seek chunk sizes that work well across all benchmarks. We limit the search space by setting *i1=i2=....=ik,* and *h1=h2=....=hk.* Figure 4.20 shows the average *fahCnt* and *dCC* field sizes as a function of chunk sizes (h0, h1) and (i0, i1) = {(2,1),(2,2)... (6,6)} when all the benchmarks are considered together for *mlvCFiat* with *CS16* configuration. The results show that chunk sizes (i0, i1)=(2, 2) and (h0, h1)=(4,2) perform well for the *fahCnt* and *dCC* fields, respectively. We perform similar analysis for different techniques with different cache configurations. The chunk sizes as listed in Table 4.5 work well for the *dCC* and *fahCnt* fields across all benchmarks for a given cache configuration.

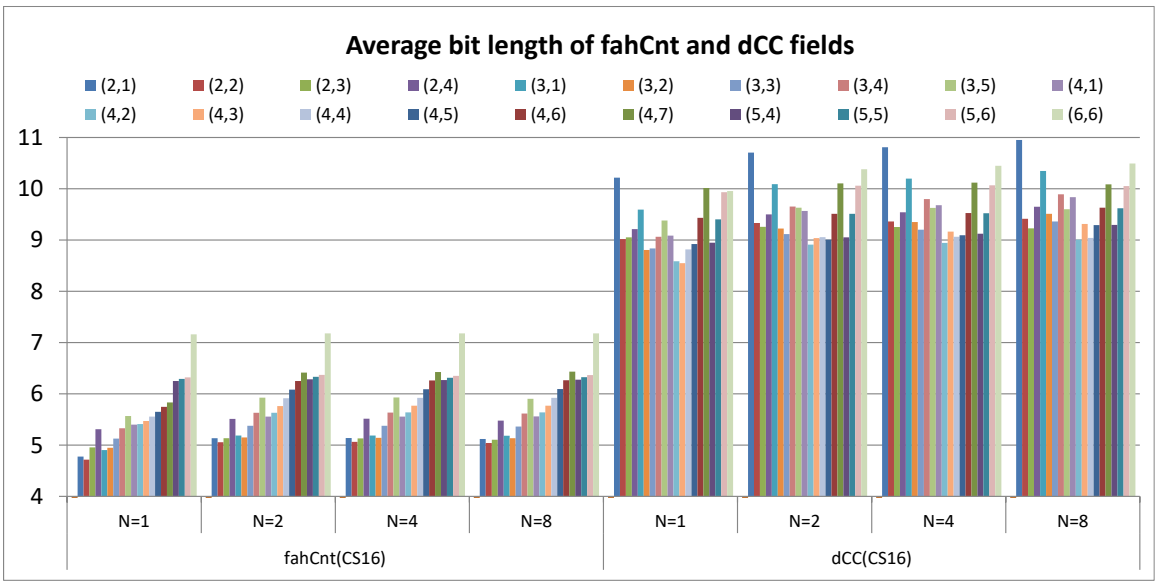

Figure 4.20 Average *fahCnt* and *dCC* fields as a function of chunk sizes for *mlvCFiat*

Table 4.5 Summary of variable encoding parameters for different fields

Variable Encoding Parameters				
Mechanism	*fields*	*CS16*	*CS32*	*CS64*
mlvCFiat (N= 1, 2, 4, 8)	*dCC*	4_2	4_2	5_4
	fahCnt	2_2	3_2	3_2
mc^2RT (N= 1, 2, 4, 8)	*dCC*	5_4	4_2	5_5
	fahCnt	3_2	4_2	3_3
mc^2RFiat (N= 1, 2, 4, 8)	*dCC*	4_2	4_2	5_4
	fahCnt	2_2	2_2	3_2

CHAPTER 5

TRACE PORT BANDWIDTH ANALYSIS

This chapter discusses the main results from an experimental evaluation of the proposed techniques for data tracing in multicores. We measure the average trace port bandwidth as a function of the number of processor cores in a multicore, an encoding mechanism, and different data cache configurations. To quantify the pressure on the trace port, the average trace port bandwidth is measured in bits per instruction [bpi] and bits per clock cycle [bpc]. The average trace port bandwidth in bits per clock cycle for a given benchmark is calculated by dividing the total trace size in bits with the total number of instructions executed by the benchmark. The average trace port bandwidth in bits per clock cycle is calculated by dividing the total trace size in bits with the execution time measured in clock cycles. Section 5.1 discusses the average trace port bandwidth requirements for the Nexus-like load data value traces (*NX_b*), the *mlvCFiat* technique with fixed (*CF_b*) and variable encoding mechanisms (*CF_e*), the *mc²RT* technique with fixed (*RT_b*) and variable encoding mechanisms (*RT_e*), the *mc²RFiat* technique with fixed (*RF_b*) and variable encoding mechanisms (*RF_e*).

5.1 Trace Port Bandwidth for Load Data Value Traces

5.1.1 NX_b

Table 5.1 shows the trace port bandwidth in bpi and bpc for the Nexus-like load data value traces as a function of a number of cores/threads (N=1, 2, 4 and 8)

for all benchmarks from the Splash-2 benchmark suite. The last row in the table named *Total* is the total average trace port bandwidth when all the benchmarks are considered together. The total average trace port bandwidth in bits per instruction is calculated by dividing the sum of trace sizes in bits for all the benchmarks with the sum of executed instructions from all the benchmarks. Similarly, the total average trace port bandwidth in bits per clock cycle is calculated by dividing the sum of trace sizes in bits from all the benchmarks with the sum of execution times measured in clock cycles for all the benchmarks.

For a single core (N=1), the total average trace port bandwidth is 12.34 bpi and it ranges from 8.82 bpi for *fmm* to 15.35 bpi for *cholesky*. The trace port bandwidth requirement is highly correlated with the frequency of memory reads and the size of typical operands read from memory. For example, the frequency of memory reads in c*holesky* is higher than in *fmm* (Table 4.2); in addition, ~45% of the time memory reads in c*holesky* are 8 bytes long (Table 4.3). Thus, *cholesky* requires a higher trace port bandwidth than *fmm*. The trace port bandwidth slightly increases with an increase in the number of cores because of a higher overhead in reporting core ID (*Pi*) and a higher frequency of memory reads due to synchronization in multicores. Thus, the average trace port bandwidth for octa core (N=8), ranges from 9.33 bpi for *fmm* to 16.01 bpi for *raytrace*. The average trace port bandwidth for the octa-core (N=8) is 13.17 bpi.

The average trace port bandwidth in bits per instruction does not fully capture the pressure on the trace port in multicores. To further illustrate the bandwidth requirement challenges in multicores we consider the trace port bandwidth measured in bits per clock cycle. The trace port bandwidth in bpc depends not only on fre-

quency and typical operand sizes but also on the multicore processor model (pipeline, out-of-order execution, caches, and other), which can be characterized by the number of instructions committed per clock cycle (IPC). For the processor model described in Section 4.6.1, the average trace port bandwidth ranges from 2.79 bpc for *fft* to 12.68 bpc for *water-ns* in the single core system (*N*=1). *Raytrace* and *water-ns* require more than 42 bpc for an octa-core system (*N*=8).

Table 5.1 Trace port bandwidth for *NX_b* for Splash2 benchmarks

Benchmarks	*Trace Port Bandwidth [bpi]*				*Trace Port Bandwidth [bpc]*			
No. of Cores	*N=1*	*N=2*	*N=4*	*N=8*	*N=1*	*N=2*	*N=4*	*N=8*
barnes	15.03	15.31	15.59	15.86	5.50	8.24	14.96	26.79
cholesky	15.35	16.21	15.87	15.59	2.93	6.58	14.67	32.99
fft	10.65	10.84	11.02	11.19	2.79	4.78	7.93	11.59
fmm	8.82	8.96	9.14	9.33	3.59	7.17	13.92	25.16
lu	11.88	12.07	12.27	12.47	4.67	8.97	15.56	24.38
radiosity	12.11	12.36	12.59	12.58	5.87	10.79	20.81	37.60
radix	13.41	13.75	14.09	14.54	3.14	5.01	7.67	9.41
raytrace	15.17	15.45	15.73	16.01	7.53	14.35	26.40	42.70
water-ns	10.64	10.81	10.98	11.15	6.49	12.68	24.34	43.51
water-sp	11.38	11.55	11.73	11.90	7.50	12.40	20.29	32.48
Total	12.34	12.63	12.89	13.17	4.92	8.76	15.61	25.64

Figure 5.1 shows the average trace port bandwidth for the Nexus-like load value trace, broken down into different fields of the trace messages: *LV*, *Pi*, *dCC*. The majority of the bandwidth is consumed tracing out the load data values (*LV*). The *LV* portion ranges from 83% for *N*=1 to 78% for *N*=8. The timestamp field (*dCC*) requires16-17% of the total trace port bandwidth. Thus, even if we order trace messages in the trace buffer and stream them out without the time field we can only

save 16-17% of the total trace port bandwidth. Further, this approach requires additional hardware resources for buffering and sorting the trace messages.

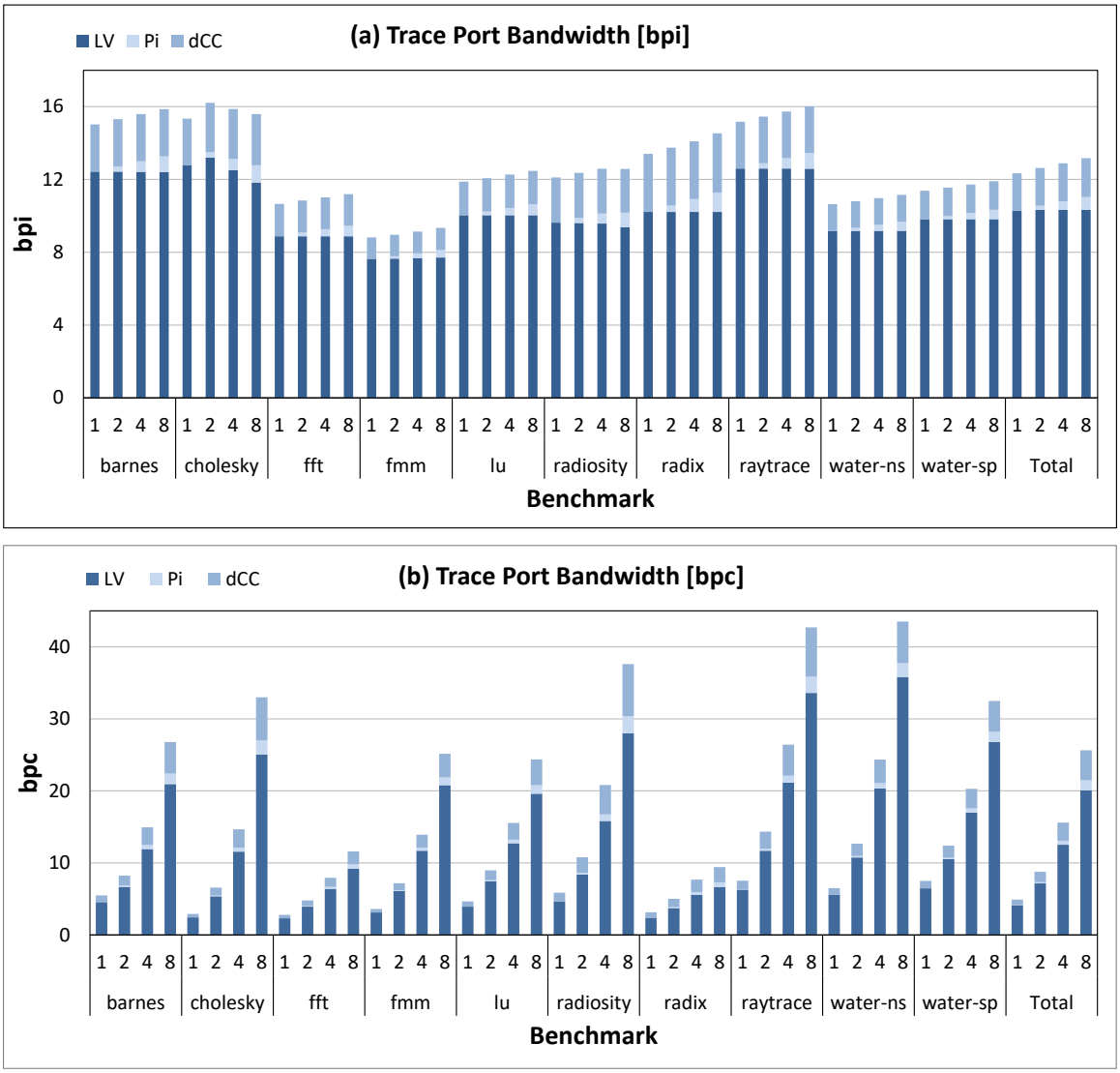

Figure 5.1 Breakdown of trace port bandwidth for *NX_b* for Splash2 benchmarks

We compress the *NX_b* traces in software in order to estimate how much we can reduce the average trace port bandwidth if the software compressor is imple-

mented in hardware. We should note that implementing a software compressor in hardware would require a significant amount of hardware resources. Table 5.2 shows compression ratios for the *NX_b* traces compressed using the *gzip* utility with level 1 compression. Two types of *NX_b* data traces are considered: when the *NX_b* traces are fed into the compressor as is (*Unified)* and when the trace messages are split into the header and load data value fields and then fed into separate compressors (*Split*). The compression ratio for the *Unified* method is calculated by dividing the total *NX_b* trace size with the total trace size after compression. With this method, we are able to achieve a compression ratio of 1.3 to1.4 times. To exploit redundancy present in different fields of trace messages, we divide trace message into two streams (a) *LV* (b) *dCC*, *Pi*. These two streams are compressed separately using gzip with compression level 1 (*Split* field in Table 5.2). The compression ratio for the *Split* method is calculated by dividing the sum of raw *NX_b* traces with the sum of separately compressed files for two streams. Whereas the compression ratio for compressed Split *NX_b* traces exceeds the compression ratio for Unified trace (2.18-3.33 times), the compression ratios remain relatively modest. It should be noted that implementing a gzip -1 compressor entirely in hardware would be cost prohibitive.

Table 5.2 Compression ratios achieved by gzip

# Cores	N=1		N=2		N=4		N=8	
Config	Split	Unified	Split	Unified	Split	Unified	Split	Unified
barnes	2.10	1.38	1.78	1.30	1.66	1.24	1.58	1.27
cholesky	6.73	1.74	3.85	1.67	3.53	1.85	4.13	2.50
fft	1.93	1.39	1.79	1.36	1.68	1.30	1.66	1.37
fmm	4.95	1.95	3.73	1.85	3.06	1.58	2.75	1.59
lu	5.93	1.56	3.58	1.54	3.14	1.42	3.06	1.77
radiosity	3.86	1.63	2.50	1.54	2.10	1.37	1.95	1.48
radix	4.23	2.02	3.05	1.81	2.08	1.46	1.96	1.42
raytrace	3.88	1.51	2.61	1.47	2.26	1.32	2.08	1.38
water-ns	2.69	1.41	2.08	1.38	1.94	1.27	1.87	1.35
water-sp	3.03	1.37	2.40	1.36	2.11	1.26	2.02	1.39
Total	3.33	1.54	2.52	1.49	2.21	1.38	2.18	1.52

All these results indicate that capturing load data value traces requires deeper trace buffers and much wider trace ports. To solve this problem, we developed hardware techniques that reduce the size of the trace data that need to be emitted. These techniques are shown in the following sections.

5.1.2 mlvCFiat

The effectiveness of *mlvCFiat* directly depends on (a) benchmark characteristics such as the type, frequency, and distribution of memory read operations, (b) data cache miss rates and first-access flag miss rates, and (c) encoding parameters. The first-access flag miss rate is a good indicator of *mlvCFiat* effectiveness – the lower it is, the fewer trace messages need to be streamed out through the trace port. Figure 5.2 shows the total first-access miss rate for the entire benchmark suite as a function of the number of cores and the data cache configurations (*CS16*, *CS32*, and *CS64*). The total first-access miss rate is calculated by dividing the total number of

first-access misses with the total number of data reads when all the benchmarks are considered together.

The total first-access miss rate ranges from 5.39% for *N*=1 to 5.68% for *N*=8 for the *CS16* configuration, 2.49% for *N*=1 to 4.06% for *N*=8 for the *CS32* configuration and 1.47% for *N*=1 to 3.44% for *N*=8 for the *CS64* configuration. As the cache size increases, the first-access miss rate decreases because larger caches result in a smaller number of read misses and consequently a smaller number of first-access miss events. Figure 5.2 also shows the minimum and the maximum first-access miss rates. The first-access miss rate reaches as high as 17.96% for *fft* when *N*=2 with *CS16* configuration and as low as 0.17% for *water-ns* when *N*=1 with *CS64* configuration.

The results confirm our expectations that *mlvCFiat* can indeed significantly reduce the number of trace messages that need to be streamed out through the trace port. We find that the total first-access miss rate does not increase significantly as we increase the number of cores. With an increase in the number of cores, we may decrease the number of conflict-induced misses, but we may also increase the number of misses caused by invalidations.

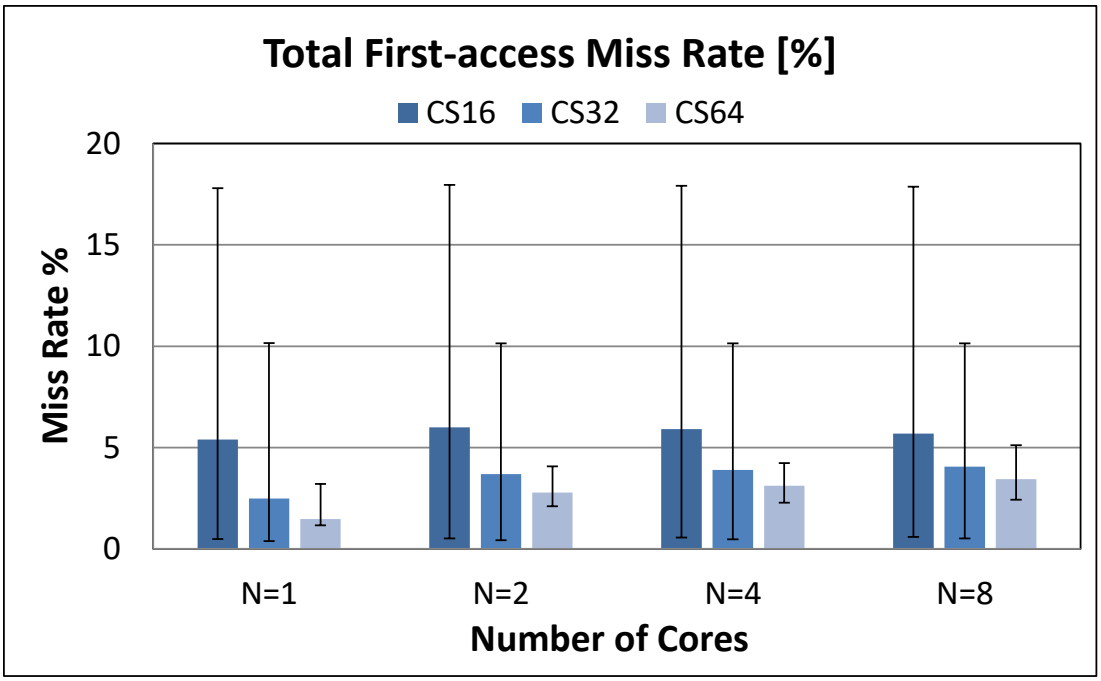

Figure 5.2 First Access Miss Rate for *mlvCFiat* for Splash2 benchmarks

Figure 5.3 shows the trace port bandwidth for *mlvCFiat* with base encoding (*CF_b*), *mlvCFiat* with variable encoding (*CF_e*) as a function of a number of cores (N=1, 2,4 and 8) and data cache configurations (*CS16*, *CS32*, and *CS64*).

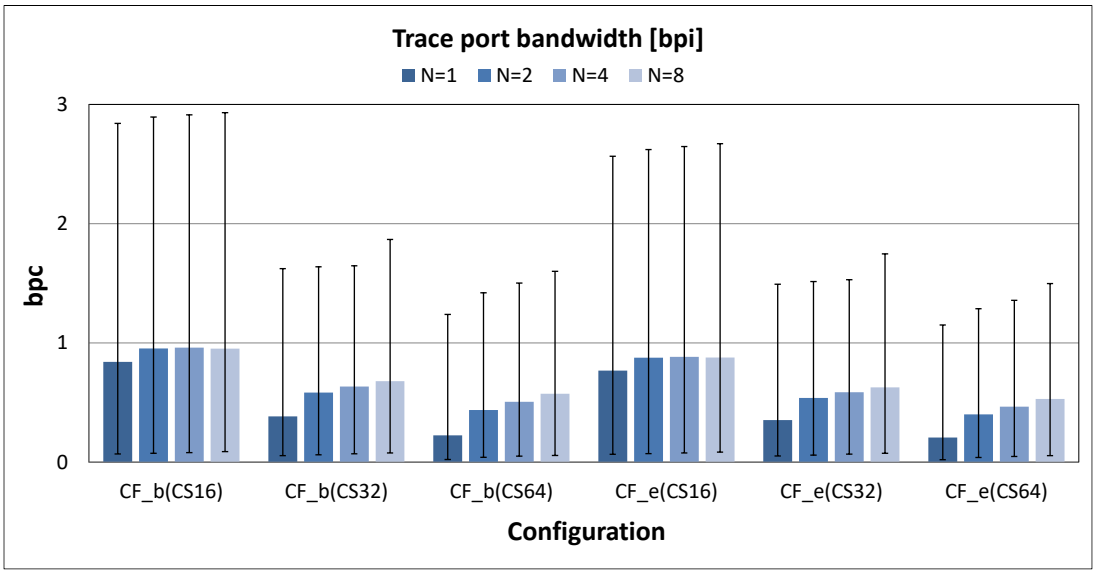

Figure 5.3 Total average trace port bandwidth in bpi for *CF_b* and *CF_e*

Table 5.3 and Table 5.4 shows the trace port bandwidth for base encoding (*CF_b*) and variable encoding (*CF_e*) mechanisms for each benchmark separately. The total average trace port bandwidth is less than 1 bit per instruction regardless of the number of cores. *mlvCFiat* with base encoding (*CF_b*) reduces the trace port bandwidth as follows:

- CS16 configuration: The average trace port bandwidth ranges from 0.07 bpi for *water-sp* to 2.84 bpi for *fft* when *N*=1 and 0.09 bpi for *water-sp* to 2.93 bpi for *fft* when *N*=8. The total average trace port bandwidth is 0.84 bpi for *N*=1 and 0.95 bpi when *N*=8.
- CS32 configuration: The average trace port bandwidth ranges from 0.05 bpi *water-sp* to 1.62 bpi for *fft* when *N*=1 and 0.08 bpi for *water-sp* to 1.87 bpi for *barnes* when *N*=8. The total average trace port bandwidth is 0.38 bpi for *N*=1 and 0.68 bpi when *N*=8.

- CS64 configuration: The average trace port bandwidth ranges from 0.02 bpi for *water-ns* to 1.24 bpi for *fft* when *N*=1 and 0.06 bpi for *water-sp* to 1.6 bpi for *barnes* when *N*=8. The total average trace port bandwidth is 0.23 bpi for *N*=1 and 0.57 bpi when *N*=8.

Table 5.3 Trace port bandwidth bpi for *CF_b*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	2.34	0.82	0.20	2.43	1.23	0.72	2.52	1.55	1.12	2.57	1.87	1.60
cholesky	1.94	0.74	0.68	1.32	0.79	0.71	0.96	0.62	0.56	0.61	0.44	0.39
fft	2.84	1.62	1.24	2.89	1.64	1.26	2.91	1.65	1.26	2.93	1.67	1.28
fmm	0.36	0.23	0.15	0.37	0.23	0.15	0.37	0.24	0.16	0.37	0.24	0.16
lu	0.54	0.53	0.49	0.57	0.53	0.31	0.58	0.54	0.33	0.62	0.42	0.18
radiosity	0.25	0.10	0.05	0.58	0.48	0.44	0.58	0.48	0.45	0.67	0.58	0.56
radix	0.85	0.60	0.49	1.76	1.52	1.42	1.86	1.61	1.50	1.90	1.65	1.53
raytrace	1.04	0.35	0.12	1.23	0.59	0.37	1.31	0.69	0.48	1.49	0.89	0.68
water-ns	0.49	0.23	0.02	0.52	0.26	0.05	0.56	0.40	0.33	0.57	0.43	0.41
water-sp	0.07	0.05	0.03	0.07	0.06	0.04	0.08	0.07	0.05	0.09	0.08	0.06
Total	0.84	0.38	0.23	0.95	0.58	0.44	0.96	0.63	0.51	0.95	0.68	0.57

The variable encoding mechanism (*CF_e*) further reduces the average trace port bandwidth compared to base encoding (*CF_b*).

- CS16 configuration: The total average trace port bandwidth is 0.77 bpi for *N*=1 to 0.88 bpi when *N*=8. *CF_e* reduces the trace port bandwidth relative to *CF_b* by 1.10 times when *N*=1 and 1.09 times when *N*=8.
- CS32 configuration: The total average trace port bandwidth is 0.35 bpi for *N*=1 to 0.63 bpi when *N*=8. *CF_e* reduces the trace port bandwidth relative to *CF_b* by 1.09 times when *N*=1 and 1.08 times when *N*=8.

- CS64 configuration: The total average trace port bandwidth is 0.21 bpi for *N*=1 to 0.53 bpi when *N*=8. *CF_e* reduces the trace port bandwidth relative to *CF_b* by 1.09 times when *N*=1 and 1.08 times when *N*=8.

Table 5.4 Trace port bandwidth bpi for *CF_e*

# Cores	*N=1*			*N=2*			*N=4*			*N=8*		
Config	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*
barnes	2.17	0.77	0.19	2.26	1.15	0.67	2.35	1.45	1.04	2.40	1.75	1.50
cholesky	1.76	0.67	0.62	1.21	0.71	0.65	0.88	0.56	0.51	0.56	0.40	0.36
fft	2.57	1.49	1.15	2.62	1.51	1.17	2.65	1.53	1.18	2.67	1.55	1.19
fmm	0.33	0.21	0.14	0.34	0.22	0.14	0.34	0.22	0.15	0.35	0.23	0.15
lu	0.54	0.52	0.47	0.57	0.52	0.30	0.58	0.53	0.32	0.61	0.40	0.16
radiosity	0.22	0.08	0.04	0.51	0.43	0.39	0.51	0.42	0.40	0.59	0.52	0.50
radix	0.75	0.53	0.43	1.64	1.41	1.29	1.73	1.48	1.36	1.78	1.53	1.42
raytrace	0.93	0.32	0.11	1.10	0.53	0.34	1.18	0.62	0.44	1.34	0.81	0.62
water-ns	0.47	0.22	0.02	0.50	0.25	0.05	0.54	0.39	0.31	0.54	0.41	0.39
water-sp	0.07	0.05	0.03	0.07	0.06	0.04	0.08	0.07	0.05	0.08	0.07	0.05
Total	0.77	0.35	0.21	0.88	0.54	0.40	0.88	0.59	0.47	0.88	0.63	0.53

Table 5.5 shows the compression ratio or speedups achieved by *mlvCFiat* relative to *NX_b* for all benchmarks. The compression ratio for a given benchmark is calculated by dividing the trace port bandwidth required for Nexus-like load data value traces with the trace port bandwidth required for *CF_e*. Compression ratios range from as low as 4.2 for *fft* to 521.5 for *water-ns* when *N*=1 and 4.2 for *fft* to 218.3 for *water-sp* when *N*=8. The total compression ratio when all the benchmarks are considered together ranges from 16.1 when *N*=1 to 15.0 when *N*=8 with CS16 configuration, from 35.0 when *N*=1 to 21.0 when *N*=8 with *CS32* configuration and from 59.6 when N=1 to 24.8 when *N*=8 with *CS64* configuration. The effectiveness of

mlvCFiat decreases as we increase the number of cores. This can be explained by an increase in the cache coherent traffic which results in the number of trace messages because *mlvCFiat* does not support coherent traffic.

Table 5.5 Compression ratio of *CF_e* relative to *NX_b*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	6.9	19.5	79.5	6.8	13.4	22.9	6.6	10.7	15.0	6.6	9.1	10.6
cholesky	8.7	22.9	24.7	13.4	22.8	25.1	18.0	28.1	31.1	27.8	38.7	43.9
fft	4.2	7.1	9.2	4.1	7.1	9.3	4.2	7.2	9.3	4.2	7.2	9.4
fmm	26.4	42.2	64.8	26.3	41.6	63.0	26.5	41.4	61.8	26.8	41.3	60.7
lu	22.0	22.8	25.0	21.3	23.4	39.8	21.3	23.3	38.0	20.6	30.9	75.6
radiosity	55.5	144.5	288.3	24.1	29.1	31.6	24.5	29.9	31.4	21.3	24.4	25.3
radix	17.9	25.3	30.9	8.4	9.8	10.7	8.2	9.5	10.4	8.2	9.5	10.3
raytrace	16.3	48.0	140.5	14.0	29.1	45.6	13.3	25.2	36.0	12.0	19.8	25.6
water-ns	22.5	47.9	521.5	21.6	43.4	232.9	20.4	28.3	35.2	20.5	27.2	28.8
water-sp	173.7	216.7	354.4	162.7	194.1	295.4	150.9	175.3	244.8	140.3	160.4	218.3
Total	16.1	35.0	59.6	14.4	23.5	31.5	14.6	22.0	27.7	15.0	21.0	24.8

Figure 5.4 shows the total average trace port bandwidth with minimum-maximum ranges in bits per clock cycle for *mlvCFiat* with base and variable encoding mechanisms. *mlvCFiat* provides significant reductions in the trace port bandwidth. Thus, *mlvCFiat* (*CS16*) with variable encoding requires 0.31 bpc when *N*=1 and 1.71 bpc when *N*=8. *CS64* configuration with variable encoding requires 0.09 bpc when *N*=1 and 1.05 bpc when *N*=8. The benchmarks *raytrace* and *water-ns* which require more than 42 bpc when N=8 now requires only ~2 bpc with *CS64* configuration.

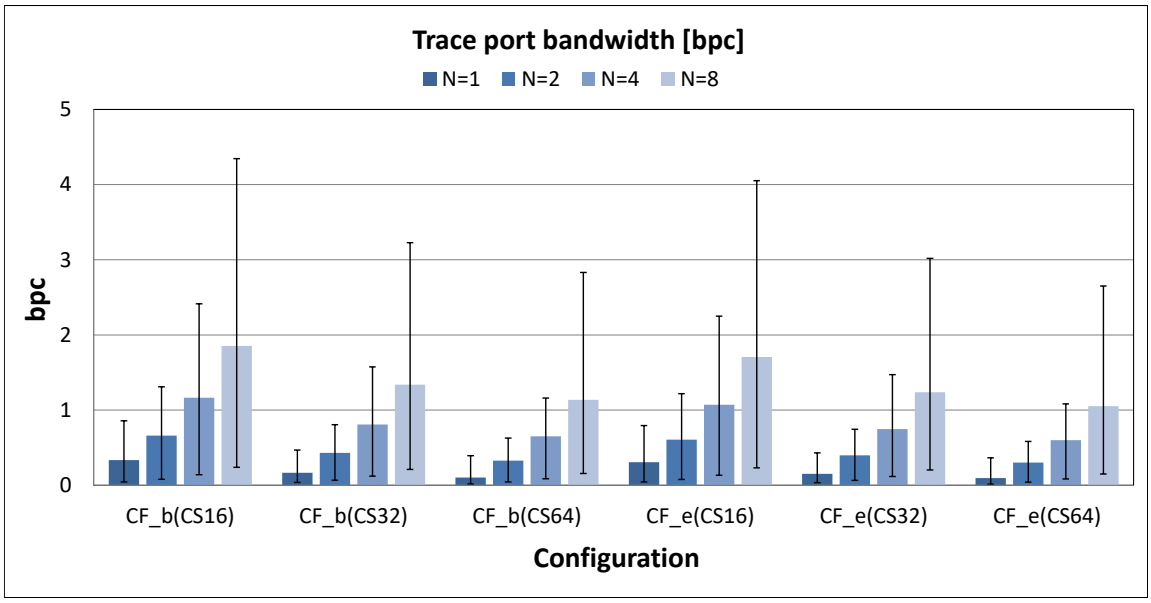

Figure 5.4 Total average trace port bandwidth in bpc for *CF_b* and *CF_e*

5.1.3 mc²RT

The effectiveness of *mc²RT* directly depends on (a) benchmark characteristics such as the type, frequency, and distribution of memory read operations, (b) data cache miss rates and first-access flag miss rates, and (c) encoding parameters. Trace miss rate is a good indicator of *mc²RT* effectiveness – the lower it is, the fewer trace messages need to be streamed out through the trace port.

Figure 5.5 shows the total trace miss rate for the entire benchmark suite as a function of the number of cores and the data cache configurations (*CS16*, *CS32*, and *CS64*). The total trace miss rate is calculated by dividing the total number of trace misses with the total number of data reads when all the benchmarks considered together. The total trace miss rate ranges from 1.91% for N=1 to 1.05% for N=8 with *CS16* configuration, 0.81% for N=1 to 0.52% for N=8 with *CS32* configuration

and 0.41% for *N*=1 to 0.32% for *N*=8 with *CS64* configuration. As the cache size increases, the trace miss rate decreases because larger caches result in a smaller number of read misses and consequently a smaller number of trace miss events. Figure 5.5 also shows the minimum and the maximum trace miss rates. The trace miss rate reaches as high as 4.70% for *barnes* when *N*=1 with *CS16* configuration and as low as 0.05% for *water-ns* when *N*=4 *with CS64* configuration.

The results confirm our expectations that *mc²RT* can indeed significantly reduce the number of trace messages that needs to be streamed out through the trace port. With an increase in the number of processor cores, the portion of truly shared data is growing. Since our mechanism inherits tracing bits during cache coherent transactions, we will avoid tracing cache blocks that have been previously reported by other processors.

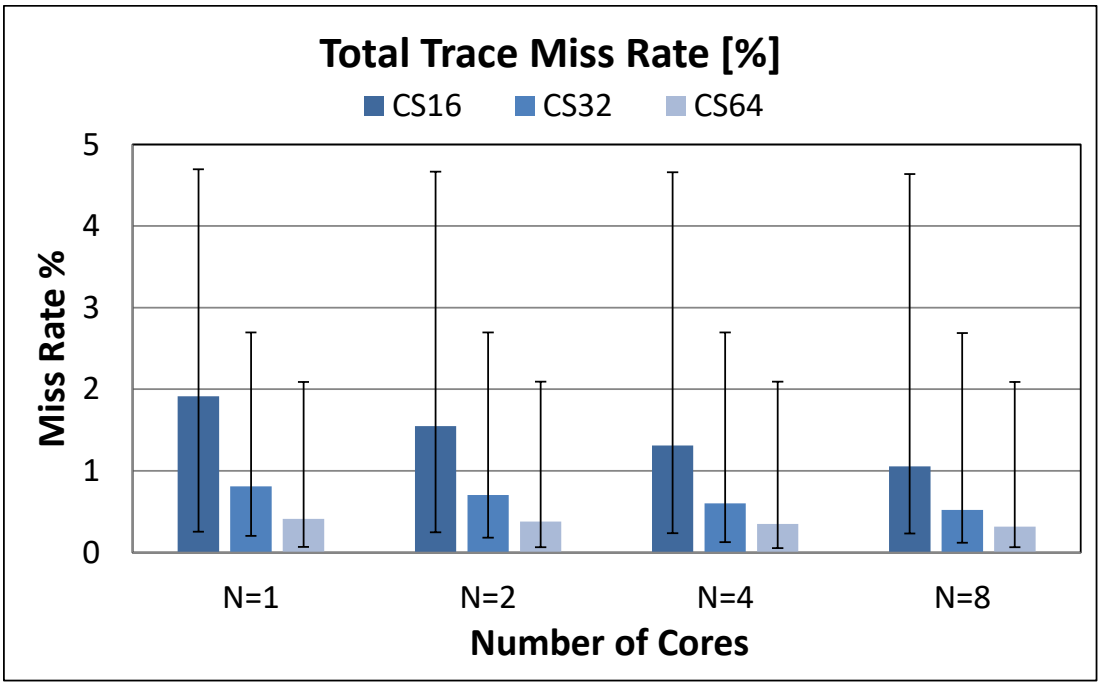

Figure 5.5 Trace Miss Rate for *mc²RT* for Splash2 benchmarks

Figure 5.6 shows the trace port bandwidth for *mc²RT* with base encoding (*RT_b*), *mc²RT* with variable encoding (*RT_e*) as a function of a number of cores (*N*=1, 2, 4 and 8) and data cache configurations (*CS16*, *CS32*, and *CS64*).

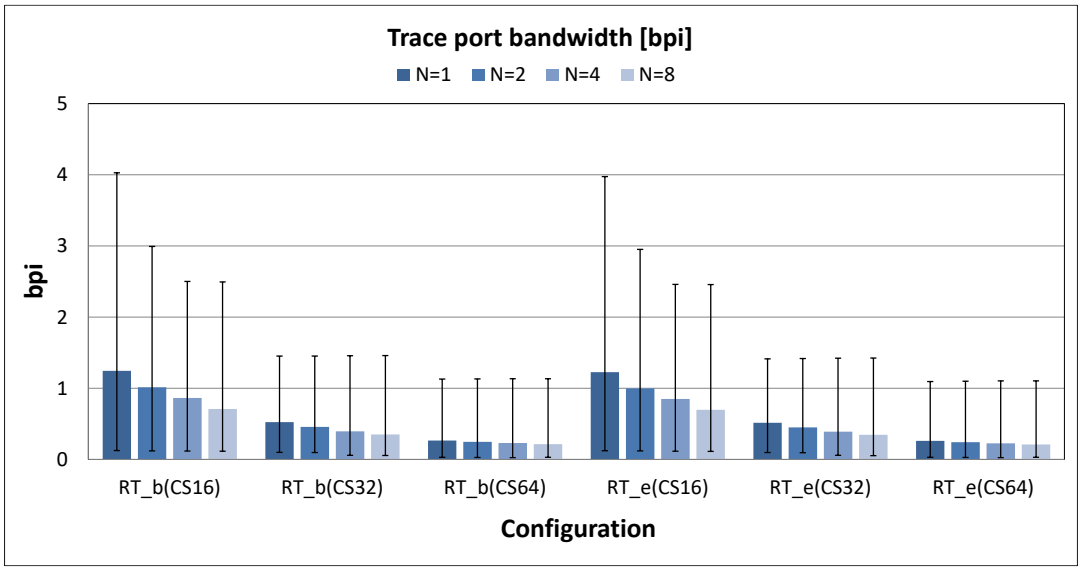

Figure 5.6 Total average trace port bandwidth in bpi for *RT_b* and *RT_e*

Table 5.6 and Table 5.7 shows the trace port bandwidth for base encoding (*RT_b*) and variable encoding (*RT_e*) mechanisms for each benchmark separately. The total average trace port bandwidth is less than 1.5 bit per instruction regardless of the number of cores. *mc²RT* with base encoding (*RT_b*) reduces the trace port bandwidth as follows:

- CS16 configuration: The average trace port bandwidth ranges from 0.12 bpi for *water-sp* to 2.47 bpi for *fft* when *N*=1 and 0.12 bpi for *wa-*

ter-sp to 2.49 bpi for *fft* when *N*=8. The total average trace port bandwidth is 1.24 bpi for *N*=1 and 0.71 bpi when *N*=8.

- CS32 configuration: The average trace port bandwidth ranges from 0.10 bpi *water-sp* to 1.45 bpi for *fft* when *N*=1 and 0.09 bpi for *water-sp* to 1.46 bpi for *fft* when *N*=8. The total average trace port bandwidth is 0.52 bpi for *N*=1 and 0.35 bpi when *N*=8.
- CS64 configuration: The average trace port bandwidth ranges from 0.03 bpi for *water-ns* to 1.13 bpi for *fft* when *N*=1 and 0.04 bpi for *water-sp* to 1.14 bpi for *fft* when *N*=8. The total average trace port bandwidth is 0.27 bpi for *N*=1 and 0.21 bpi when *N*=8.

Table 5.6 Trace port bandwidth bpi for *RT_b*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	4.03	1.41	0.37	2.99	1.05	0.31	2.16	0.80	0.29	1.60	0.67	0.33
cholesky	1.76	0.70	0.63	1.23	0.72	0.64	0.86	0.53	0.46	0.52	0.36	0.29
fft	2.47	1.45	1.13	2.50	1.45	1.13	2.50	1.46	1.13	2.49	1.46	1.14
fmm	0.54	0.37	0.27	0.50	0.36	0.26	0.48	0.35	0.26	0.47	0.34	0.25
lu	0.50	0.49	0.45	0.50	0.46	0.28	0.28	0.25	0.15	0.27	0.17	0.07
radiosity	0.55	0.19	0.09	0.31	0.14	0.07	0.29	0.12	0.08	0.21	0.10	0.06
radix	1.42	0.68	0.38	1.43	0.69	0.38	1.44	0.69	0.39	1.46	0.70	0.40
raytrace	1.97	0.61	0.20	1.58	0.46	0.15	1.42	0.41	0.13	1.24	0.35	0.10
water-ns	0.82	0.37	0.03	0.83	0.38	0.03	0.64	0.06	0.03	0.15	0.06	0.03
water-sp	0.12	0.10	0.06	0.12	0.10	0.06	0.12	0.09	0.05	0.12	0.09	0.04
Total	1.24	0.52	0.27	1.01	0.46	0.25	0.86	0.40	0.23	0.71	0.35	0.21

The variable encoding mechanism (*RT_e*) further reduces the average trace port bandwidth compared to base encoding (*RT_b*).

- CS16 configuration: The total average trace port bandwidth is 1.23 bpi for *N*=1 to 0.70 bpi when *N*=8. *RT_e* reduces the trace port bandwidth relative to *RT_b* by 1.02 times.
- CS32 configuration: The total average trace port bandwidth is 0.52 bpi for *N*=1 to 0.35 bpi when *N*=8. *RT_e* reduces the trace port bandwidth relative to *RT_b* by 1.02 times.
- CS64 configuration: The total average trace port bandwidth is 0.26 bpi for *N*=1 to 0.21 bpi when *N*=8. *RT_e* reduces the trace port bandwidth relative to *RT_b* by 1.02 times.

Table 5.7 Trace port bandwidth bpi for *RT_e*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	3.97	1.39	0.37	2.95	1.04	0.31	2.13	0.79	0.28	1.58	0.66	0.32
cholesky	1.73	0.68	0.62	1.21	0.71	0.63	0.85	0.52	0.45	0.51	0.35	0.29
fft	2.42	1.41	1.09	2.46	1.42	1.10	2.46	1.42	1.10	2.46	1.43	1.11
fmm	0.53	0.36	0.26	0.49	0.35	0.26	0.47	0.35	0.25	0.46	0.33	0.24
lu	0.49	0.48	0.45	0.49	0.45	0.27	0.28	0.25	0.15	0.26	0.17	0.07
radiosity	0.54	0.19	0.09	0.31	0.14	0.07	0.29	0.12	0.08	0.21	0.10	0.06
radix	1.39	0.67	0.38	1.40	0.67	0.38	1.41	0.68	0.38	1.42	0.69	0.39
raytrace	1.95	0.61	0.19	1.56	0.46	0.15	1.40	0.41	0.12	1.22	0.35	0.10
water-ns	0.82	0.37	0.03	0.82	0.37	0.03	0.63	0.06	0.03	0.15	0.05	0.03
water-sp	0.12	0.10	0.06	0.12	0.10	0.06	0.12	0.09	0.05	0.11	0.08	0.04
Total	1.23	0.52	0.26	1.00	0.45	0.24	0.85	0.39	0.23	0.70	0.35	0.21

Table 5.8 shows the compression ratio or speedups achieved by *mc²RT* relative to *NX_b* for all benchmarks. The compression ratio for a given benchmark is calculated by dividing the trace port bandwidth required for Nexus-like load data value

traces with the trace port bandwidth required for *RT_e*. Compression ratios range from as low as 4.4 for *fft* to 337.7 for *water-ns* when *N*=1 and 4.6 for *fft* to 372.9 for *water-ns* when *N*=8. The total compression ratio when all the benchmarks are considered together ranges from 10.1 when N=1 to 18.9 when *N*=8 with *CS16* configuration, from 23.9.0 when *N*=1 to 38.0 when *N*=8 with *CS32* configuration and from 47.3 when N=1 to 62.5 when *N*=8 with *CS64* configuration.

Table 5.8 Compression ratio of *RT_e* relative to *NX_b*

# Cores	*N=1*			*N=2*			*N=4*			*N=8*		
Config	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*
barnes	3.8	10.8	40.8	5.2	14.8	50.0	7.3	19.8	54.9	10.1	24.0	49.0
cholesky	8.9	22.5	24.6	13.4	22.9	25.9	18.7	30.2	35.0	30.4	44.6	54.1
fft	4.4	7.5	9.7	4.4	7.6	9.8	4.5	7.7	10.0	4.6	7.8	10.1
fmm	16.5	24.2	33.7	18.3	25.5	34.7	19.5	26.5	36.2	20.4	27.9	38.5
lu	24.4	24.5	26.6	24.8	26.6	44.0	44.5	49.9	81.0	47.7	74.2	174.9
radiosity	22.4	63.0	130.3	40.5	90.0	170.2	43.8	102.5	167.7	60.6	126.6	202.1
radix	9.7	20.1	35.7	9.8	20.4	36.3	10.0	20.8	36.7	10.2	21.1	37.1
raytrace	7.8	25.0	78.4	9.9	33.8	104.0	11.2	38.6	126.7	13.1	45.9	158.0
water-ns	13.0	28.7	337.7	13.1	28.9	371.4	17.3	187.3	437.2	75.3	203.1	372.9
water-sp	92.5	115.7	187.1	96.4	120.5	207.8	100.8	130.0	242.8	104.1	140.5	302.0
Total	10.1	23.9	47.3	12.7	28.0	52.0	15.2	33.1	56.8	18.9	38.0	62.5

Figure 5.7 shows the total average trace port bandwidth with minimum-maximum ranges in bits per clock cycle for *mc²RT* with base and variable encoding mechanisms. *mc²RT* provides significant reductions in the trace port bandwidth. Thus, *mc²RT* (*CS16*) with variable encoding requires 0.49 bpc when *N*=1 and 1.36 bpc when *N*=8. *CS64* configuration with variable encoding requires 0.12 bpc when

N=1 and 0.42 bpc when *N*=8. The benchmarks *raytrace* and *water-ns* which required more than 42 bpc when N=8 now require only ~0.3 bpc with*CS64* configuration.

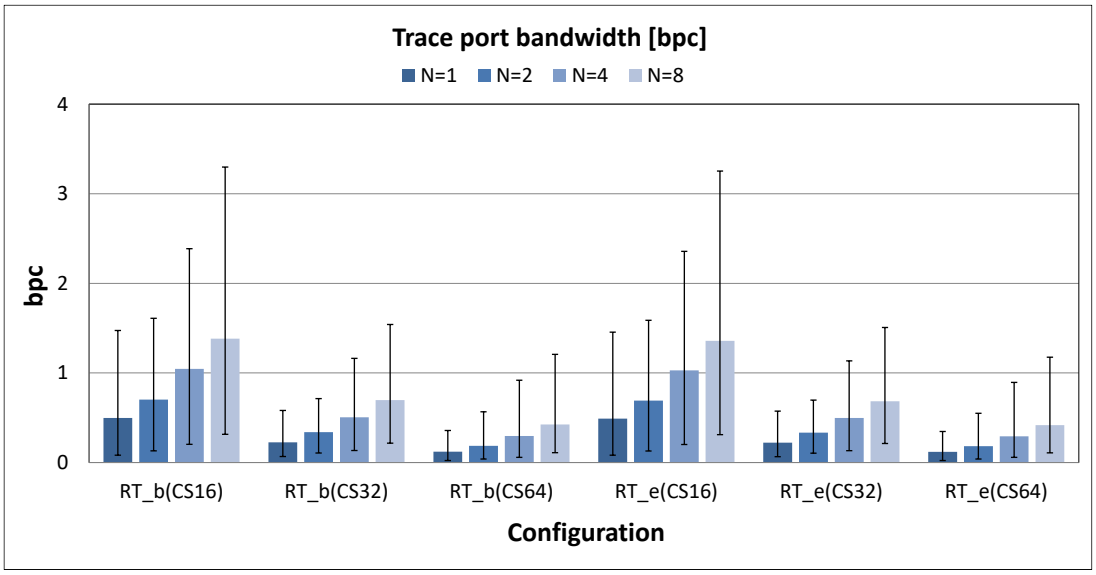

Figure 5.7 Total average trace port bandwidth in bpc for *RT_b* and *RT_e*

5.1.4 *mc²RFiat*

The effectiveness of *mc²RFiat* directly depends on (a) benchmark characteristics such as the type, frequency, and distribution of memory read operations, (b) data cache miss rates and first-access flag miss rates, and (c) encoding parameters. The first-access flag miss rate is a good indicator of the *mc²RFiat* effectiveness – the lower it is, the fewer trace messages need to be streamed out through the trace port.

Figure 5.8 shows the total first-access miss rate for the entire benchmark suite as a function of a number of cores and the three data cache configurations (*CS16*, *CS32*, and *CS64*). The total first-access miss rate is calculated by dividing

the total number of first-access misses with the total number of data reads when all the benchmarks are considered together. The total first-access miss rate ranges from 5.39% for *N*=1 to 3.10% for *N*=8 with *CS16* configuration, 2.49% for *N*=1 to 1.71% for *N*=8 with *CS32* configuration and 1.47% for *N*=1 to 1.16% for *N*=8 with *CS64* configuration. As the cache size increases, the first-access miss rate decreases because larger caches result in lower number of read misses and thus smaller number of first-access miss events. Figure 5.8 also shows the minimum and the maximum first-access miss rates. The first-access miss rate reaches as high as 17.96% for *fft* when *N*=2 with *CS16* configuration and as low as 0.11% for *water-ns* when *N*=4 *with CS64* configuration.

The results confirm our expectations that *mc²RFiat* can indeed significantly reduce the number of trace messages that needs to be streamed out through the trace port. With an increase in the number of processor cores, the portion of truly shared data is growing. Since our mechanism inherits tracing bits during cache coherent transactions, we will avoid tracing cache blocks that have been previously reported by other processors.

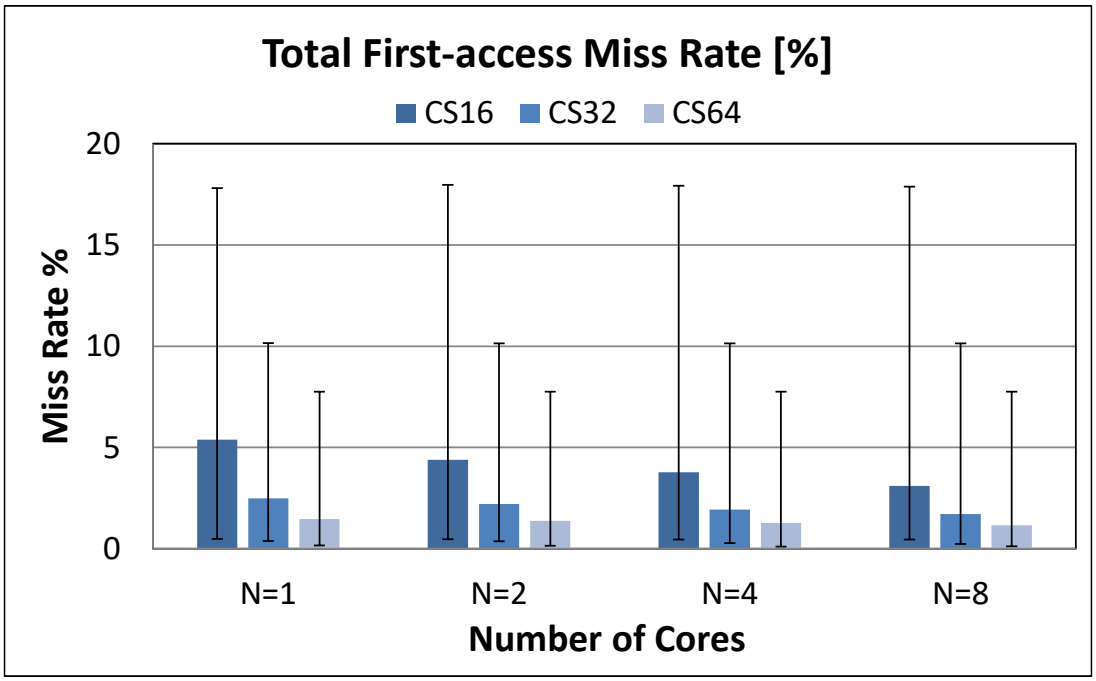

Figure 5.8 First Access Miss Rate for *mc²RT* for Splash2 benchmarks

Figure 5.9 shows the trace port bandwidth for *mc²RFiat* with base encoding (*RF_b*), *mc²RFiat* with variable encoding (*RF_e*) as a function of a number of cores (*N*=1, 2, 4 and 8) and data cache configurations (*CS16*, *CS32*, and *CS64*).

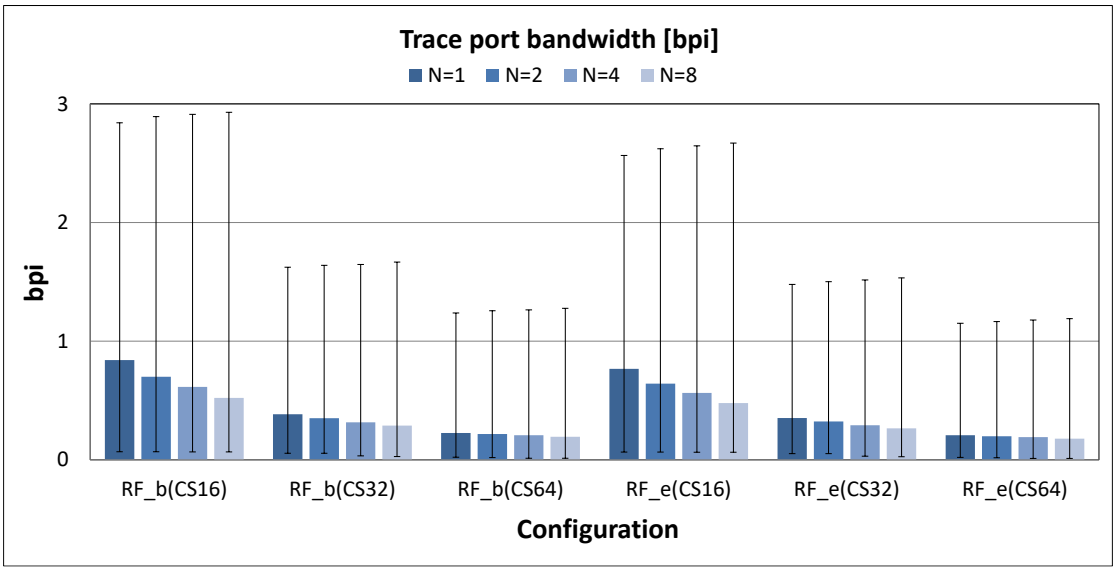

Figure 5.9 Total average trace port bandwidth in bpi for *RF_b* and *RF*_e

Table 5.10 and Table 5.9 shows the trace port bandwidth for base encoding (*RF_b*) and variable encoding (*RF_e*) mechanisms for each benchmark separately. The total average trace port bandwidth is less than 1 bit per instruction regardless of the number of cores. *mc²RFiat* with base encoding (*RF_b*) reduces the trace port bandwidth significantly as follows:

- CS16 configuration: The average trace port bandwidth ranges from 0.07 bpi for *water-sp* to 2.84 bpi for *fft* when *N*=1 and 0.07 bpi for *water-sp* to 2.93 bpi for *fft* when *N*=8. The total average trace port bandwidth is 0.84 bpi for *N*=1 and 0.52 bpi when *N*=8.
- CS32 configuration: The average trace port bandwidth ranges from 0.05 bpi *water-sp* to 1.62 bpi for *fft* when *N*=1 and 0.03 bpi for *water-ns* to 1.67 bpi for *fft* when *N*=8. The total average trace port bandwidth is 0.38 bpi for *N*=1 and 0.29 bpi when *N*=8.

- CS64 configuration: The average trace port bandwidth ranges from 0.02 bpi for *water-ns* to 1.24 bpi for *fft* when *N*=1 and 0.06 bpi for *water-sp* to 1.24 bpi for *barnes* when *N*=8. The total average trace port bandwidth is 0.23 bpi for *N*=1 and 0.19 bpi when *N*=8.

Table 5.9 Trace port bandwidth bpi for *RF_b*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	2.34	0.82	0.20	1.75	0.61	0.17	1.28	0.47	0.17	0.95	0.41	0.20
cholesky	1.94	0.74	0.68	1.31	0.77	0.69	0.91	0.57	0.49	0.54	0.38	0.31
fft	2.84	1.62	1.24	2.89	1.64	1.26	2.91	1.65	1.26	2.93	1.67	1.28
fmm	0.36	0.23	0.15	0.32	0.22	0.15	0.30	0.21	0.14	0.30	0.21	0.14
lu	0.54	0.53	0.49	0.54	0.50	0.28	0.45	0.37	0.19	0.43	0.23	0.05
radiosity	0.25	0.10	0.05	0.14	0.07	0.04	0.13	0.06	0.04	0.10	0.05	0.04
radix	0.85	0.60	0.49	0.86	0.61	0.51	0.88	0.63	0.52	0.91	0.65	0.54
raytrace	1.04	0.35	0.12	0.87	0.28	0.10	0.80	0.25	0.08	0.71	0.22	0.07
water-ns	0.49	0.23	0.02	0.49	0.23	0.02	0.38	0.03	0.01	0.08	0.03	0.01
water-sp	0.07	0.05	0.03	0.07	0.05	0.03	0.07	0.05	0.03	0.07	0.05	0.02
Total	0.84	0.38	0.23	0.70	0.35	0.22	0.61	0.32	0.21	0.52	0.29	0.19

The variable encoding mechanism (*RF_e*) further reduces the average trace port bandwidth compared to base encoding (*RF_b*).

- CS16 configuration: The total average trace port bandwidth is 0.77 bpi for *N*=1 to 0.48 bpi when *N*=8. *RF_e* reduces the trace port bandwidth relative to *RF_b* by 1.10 times when *N*=1 and 1.09 times when *N*=8.
- CS32 configuration: The total average trace port bandwidth is 0.35 bpi for *N*=1 to 0.27 bpi when *N*=8. *RF_e* reduces the trace port bandwidth relative to *RF_b* by 1.09 times.

- CS64 configuration: The total average trace port bandwidth is 0.21 bpi for *N*=1 to 0.18 bpi when *N*=8. *RF_e* reduces the trace port bandwidth relative to *RF_b* by 1.09 times.

Table 5.10 Trace port bandwidth bpi for *RF_e*

# Cores	*N=1*			*N=2*			*N=4*			*N=8*		
Config	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*	*CS16*	*CS32*	*CS64*
barnes	2.17	0.77	0.19	1.63	0.57	0.16	1.19	0.44	0.16	0.89	0.38	0.18
cholesky	1.76	0.67	0.62	1.20	0.70	0.62	0.83	0.52	0.45	0.50	0.34	0.29
fft	2.57	1.48	1.15	2.62	1.50	1.17	2.65	1.52	1.18	2.67	1.53	1.19
fmm	0.33	0.21	0.14	0.30	0.20	0.14	0.28	0.20	0.13	0.28	0.19	0.13
lu	0.54	0.53	0.47	0.54	0.50	0.28	0.45	0.37	0.19	0.43	0.23	0.05
radiosity	0.22	0.08	0.04	0.12	0.06	0.03	0.12	0.05	0.04	0.09	0.05	0.03
radix	0.75	0.54	0.43	0.77	0.56	0.44	0.79	0.57	0.46	0.82	0.60	0.47
raytrace	0.93	0.31	0.11	0.78	0.25	0.09	0.72	0.23	0.07	0.64	0.20	0.06
water-ns	0.47	0.22	0.02	0.48	0.22	0.02	0.37	0.03	0.01	0.08	0.03	0.01
water-sp	0.07	0.05	0.03	0.06	0.05	0.03	0.06	0.05	0.03	0.06	0.05	0.02
Total	0.77	0.35	0.21	0.64	0.32	0.20	0.56	0.29	0.19	0.48	0.27	0.18

Table 5.8 shows the compression ratio or speedups achieved by *mc²RFiat* relative to *NX_b* for all benchmarks. The compression ratio for a given benchmark is calculated by dividing the trace port bandwidth required for Nexus-like load data value traces with the trace port bandwidth required for *RF_e*. Compression ratios range from as low as 4.2 for *fft* to 521.5 for *water-ns* when *N*=1 and 4.2 for *fft* to 944.9 for *water-ns* when *N*=8. The total compression ratio when all the benchmarks are considered together ranges from 16.1 when *N*=1 to 27.6 when *N*=8 with *CS16* configuration, from 35.0 when *N*=1 to 49.6 when *N*=8 with *CS32* configuration and from 59.6 when *N*=1 to 73.8 when *N*=8 with *CS64* configuration.

Table 5.11 Compression ratio of *RF_e* relative to *NX_b*

# Cores	N=1			N=2			N=4			N=8		
Config	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64	CS16	CS32	CS64
barnes	6.9	19.6	79.5	9.4	26.9	94.6	13.1	35.4	98.7	17.9	41.9	86.0
cholesky	8.7	22.9	24.7	13.6	23.3	26.0	19.1	30.8	35.1	31.4	45.4	54.4
fft	4.2	7.2	9.2	4.1	7.2	9.3	4.2	7.3	9.3	4.2	7.3	9.4
fmm	26.4	42.2	64.8	30.4	44.9	66.3	32.5	46.4	68.7	33.8	48.6	72.9
lu	22.0	22.3	25.0	22.3	24.2	43.8	27.0	33.2	64.7	29.0	53.9	251.0
radiosity	55.5	146.2	288.3	100.8	209.8	355.1	109.2	237.0	343.6	147.2	271.1	369.9
radix	17.9	24.8	30.9	17.9	24.7	30.9	17.8	24.5	30.9	17.7	24.3	30.8
raytrace	16.3	48.3	140.5	19.9	61.7	178.4	21.9	69.0	212.0	25.2	80.0	255.7
water-ns	22.5	47.8	521.5	22.7	48.2	628.3	29.6	363.4	971.2	142.2	435.6	944.9
water-sp	173.7	216.7	354.4	178.0	222.1	391.6	184.1	237.8	449.2	188.0	252.1	543.3
Total	16.1	35.0	59.6	19.7	39.1	63.4	22.9	44.4	67.6	27.6	49.6	73.8

Figure 5.10 shows the total average trace port bandwidth with minimum-maximum ranges in bits per clock cycle for *mc²RFiat* with base and variable encoding mechanisms. *mc²RFiat* provides significant reductions in the trace port bandwidth. Thus, *mc²RFiat* (*CS16*) with variable encoding requires 0.31 bpc when *N*=1 and 0.93 bpc when *N*=8. *CS64* configuration with variable encoding requires 0.09 bpc when *N*=1 and 0.35 bpc when *N*=8. The benchmarks *raytrace* and *water-ns* which required more than 42 bpc when N=8 now requires only ~0.2 bpc with *CS64* configuration.

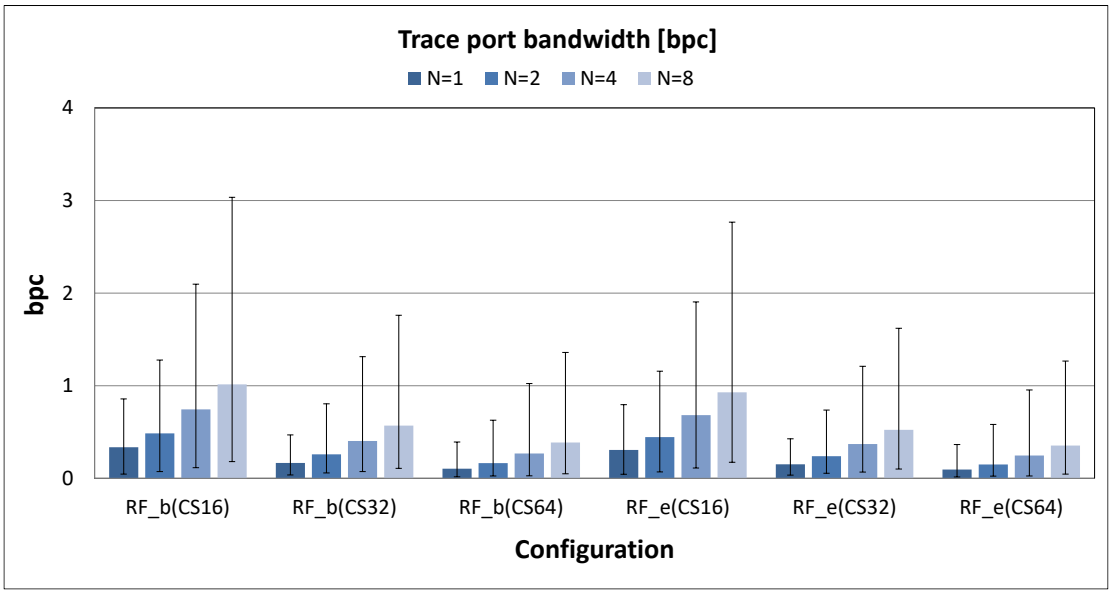

Figure 5.10 Total average trace port bandwidth in bpc for *RF_b* and *RF_e*

5.2 Dynamic Trace Port Bandwidth Analysis for Load Data Value Traces

The average trace port bandwidth allows us to quantify the effectiveness of proposed techniques but it does not fully capture the peak bandwidth requirements that occur in different phases of the program execution. Depending on the frequency and distribution of memory reads and first-access misses/trace bit misses, the peak trace port bandwidth may exceed the average trace port bandwidth. To analyze peak trace port bandwidth requirements, we consider two benchmarks *raytrace* and *water-ns*. These two benchmarks are critical because they require the trace port bandwidth of more than 42 bpc when N=8. We analyze the bandwidth requirements for *NX_b*, *NX_b.gz*, *CF_e*, *RT_e*, and *RF_e* traces.

Figure 5.11 shows the dynamic trace port bandwidth in bpc for *raytrace* for *CS64* configuration. The average trace port bandwidth for *NX_b* with *CS64* configu-

ration for raytrace is 46.47 bpc. However, the peak trace port bandwidth reaches 64.84 bpc, further underscoring the challenges in the load data value tracing. Software compression of *NX_b* traces using the *gzip* utility with level 1 compression (*NX_b.gz*) requires an average trace port bandwidth of 33.21 bpc with a peak rate of 45.89 bpc, *CF_e* requires an average trace port bandwidth of 1.81 bpc with a peak rate of 5.89 bpc, *RT_e* requires an average trace port bandwidth of 0.29 bpc with a peak rate of 5.89 bpc and *RF_e* requires an average trace port bandwidth of 0.18 bpc with peak a rate of 5.89 bpc.

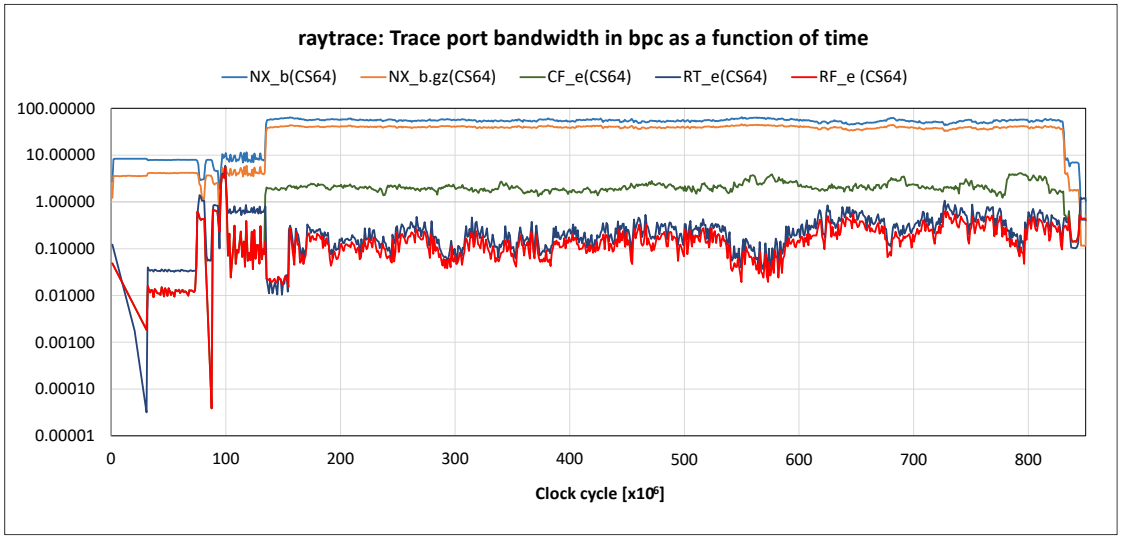

Figure 5.11 Dynamic trace port bandwidth in bpc during execution of *raytrace* for *N*=8

Figure 5.12 shows the dynamic trace port bandwidth in bpc for *water-ns* for *CS64* configuration. The average trace port bandwidth for *NX_b* with *CS64* configuration for raytrace is 46.47 bpc. However, peak trace port bandwidth reaches 57.94

bpc; *NX_b.gz* requires an average trace port bandwidth of 32.5 bpc with a peak rate of 44.9 bpc. *CF_e* requires an average trace port bandwidth of 1.51 bpc with a peak rate of 2.63 bpc. *RT_e* requires an average trace port bandwidth of 0.12 bpc with a peak rate of 2.34 bpc. *RF_e* requires an average trace port bandwidth of 0.05 bpc with a peak rate of 1.41 bpc. These results clearly show that our techniques not only reduce the average trace port but also reduce the peak trace port bandwidth requirements.

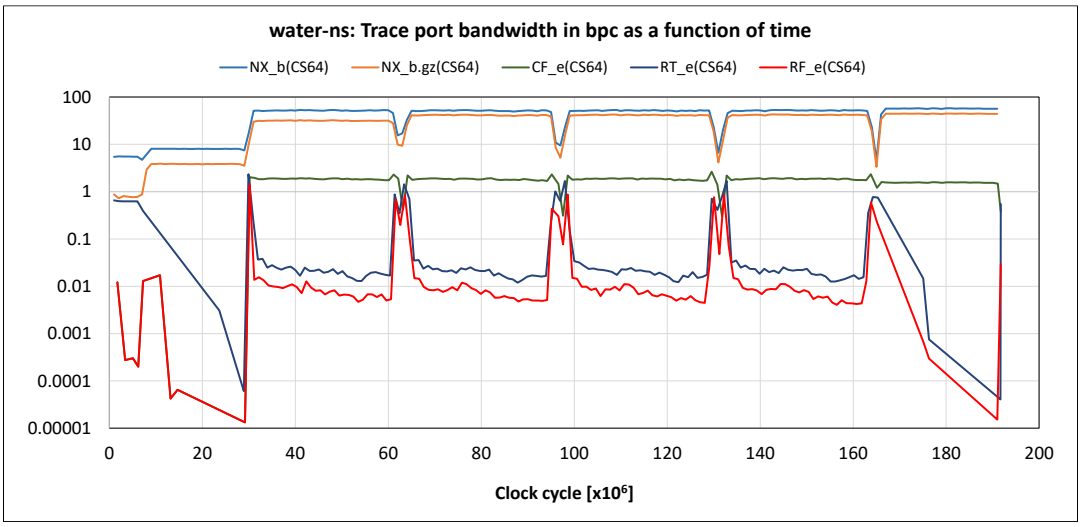

Figure 5.12 Dynamic trace port bandwidth in bpc during execution of *water-ns* for *N*=8

5.3 Putting It All Together

This section compares the improvements achieved with all proposed trace filtering techniques and additional hardware resources required to support them on the target platforms. Figure 5.13 shows the total average trace port bandwidth in

bpi for the trace filtering techniques for the *CS64* configuration. *NX_b* requires total average trace port bandwidth in the range from 12.34 bpi when *N*=1 to 13.17 bpi when *N*=8. *mlvCFiat* with variable encoding mechanism (*CF_e*) requires a total average trace port bandwidth in the range from 0.21 bpi when *N*=1 to 0.53 bpi when *N*=8. *mc²RT* with variable encoding mechanism (*RT_e*) requires a total average trace port bandwidth in the range from 0.26 bpi when *N*=1 to 0.21 bpi when *N*=8. *mc²RFiat* with variable encoding mechanism (*RF_e*) requires a total average trace port bandwidth in the range from 0.21 bpi when *N*=1 to 0.18 bpi when *N*=8.

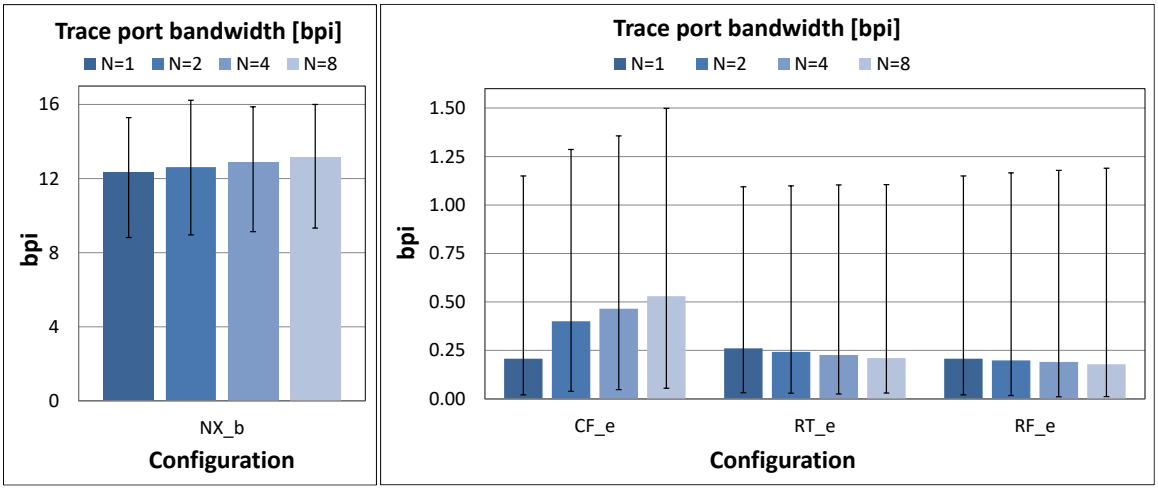

Figure 5.13 Trace port bandwidth in bpi for *CS64* configuration

Figure 5.14 shows the total average trace port bandwidth in bpc for the trace filtering techniques for the *CS64* configuration. *NX_b* requires a total average trace port bandwidth in the range from 5.65 bpc when *N*=1 to 26.14 bpc when *N*=8. *mlvCFiat* with variable encoding mechanism (*CF_e*) requires a total average trace port bandwidth in the range from 0.09 bpc when *N*=1 to 1.05 bpc when *N*=8. *mc²RT*

with variable encoding mechanism (*RT_e*) requires a total average trace port bandwidth in the range from 0.12 bpc when *N*=1 to 0.42 bpc when *N*=8. *mc²RFiat* with variable encoding mechanism (*RF_e*) requires a total average trace port bandwidth in the range from 0.09 bpc when *N*=1 to 0.35 bpc when *N*=8.

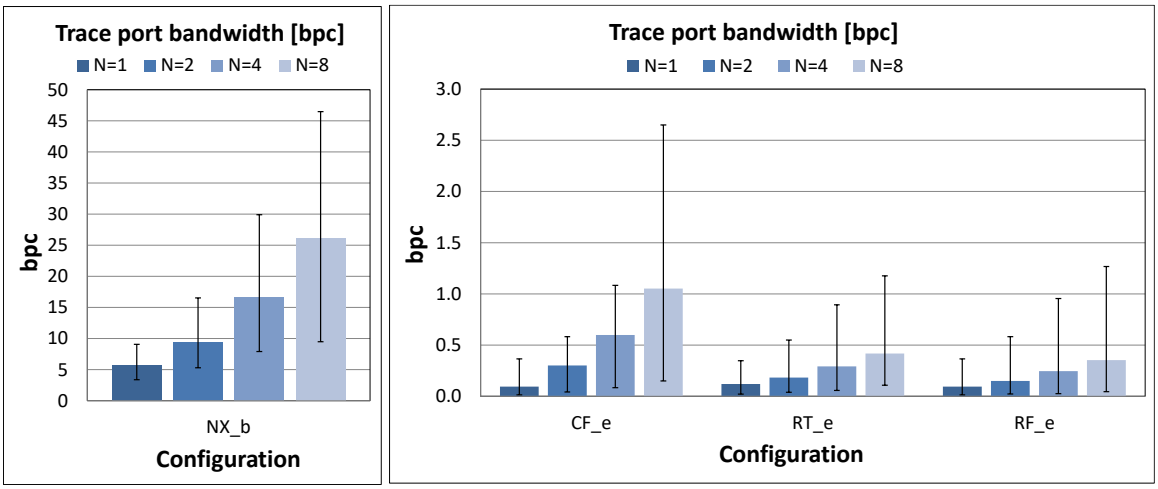

Figure 5.14 Trace port bandwidth in bpc for *CS64* configuration

To support *mlvCFiat*, first-access tracking bits need to be added to each cache block of the L1 data cache. The complexity of *mlvCFiat* depends on the granularity size. If the granularity size is higher, it requires fewer first-access bits but the trace port bandwidth may increase. As we can observe from Figure 5.13 and Figure 5.14, the effectiveness of *mlvCFiat* decreases as we increase the number of cores. This can be explained by an increase in the cache coherence traffic which results in the number of trace messages because *mlvCFiat* does not support coherence traffic. To solve this problem we introduced *mc²RT*.

To support *mc²RT*, a single trace tracking bit is added to each cache block of the L1 data cache. *mc²RT* also makes use of the *MOESI* cache coherence protocol and relies on inheriting trace tracking bits from other caches to reduce the number of trace messages. To support this feature requires hardware support for copying trace bit from other caches. For a single core, the average trace port bandwidth is higher than in *mlvCFiat* (Figure 5.13 and Figure 5.14) because a single trace message includes the content of the entire cache block regardless of the size of the actual operand. However, this feature may work well in the case of strong spatial locality in data accesses.

mc²RFiat combines the strengths of both *mlvCFiat* and *mc²RT*. *mc²RFiat* uses first-access tracking bits added to each cache block of the L1 data cache. It also uses the *MOESI* cache coherence protocol with support for inheriting first-access tracking bits for selected cache-to-cache transfers. This technique requires additional hardware support for copying first-access tracking bits from other caches. This can be done either by extending the width of data lines which support cache to cache transfer or initiating an extra bus transaction.

Depending on the requirements and hardware resources available, we can use any one of the described techniques to capture load data value traces on the target platform in real-time in multicore systems.

CHAPTER 6

CONCLUSIONS

The growing complexity of hardware and software stacks, a recent shift toward multicores, and ever-tightening time-to-market requirements make software testing and debugging one of the most critical aspects of embedded system development. Software developers need better debugging tools to reduce the time and effort it takes to find bugs. A software debugger can replay programs offline under certain conditions; it opens the doors for software developers to find bugs faster.

Load data value traces are essential in program replay but tracing entire programs we require wider trace ports and deeper buffers, which in turn increases the system cost. In this thesis, we introduce techniques, *mlvCFiat*, *mc²RT*, and *mc²RFiat* that capture, filter, and emit the load data value traces unobtrusively in real time. These techniques make use of L1 data caches with minimal hardware changes to filter load data value traces. These techniques also require a software debugger to maintain a software copy of the data caches with organization and updating policies that mirror those employed on the target platform.

To measure the effectiveness of the proposed techniques, we evaluate the trace port bandwidth measured in bits per executed instruction and bits per clock cycle while varying the number of cores $N = 1, 2, 4,$ and 8 and with different cache configurations (*CS*16, *CS*32, and *CS*64). To further reduce the average trace port bandwidth, we also consider variable encoding mechanism. Variable encoding mech-

anism reduces the average trace port bandwidth by 8 to 10% relative to base encoding.

mlvCFiat uses first-access tracking bits to minimize the number of load data value traces emitted by the target platform. The complexity of *mlvCFiat* depends on the granularity size. The total average trace port bandwidth for Nexus like load data value traces ranges from 12.34 bpi (N=1) to 13.17 bpi (N=8). *mlvCFiat* with variable encoding mechanism reduces the average trace port bandwidth by 14.7 to 54.7 times when N=1 and 13.8 to 23.0 times when N=8 relative to Nexus-like load data value traces.

mc²RT exploits cache coherence protocol along with a single tracking bit attached to a cache block of L1 data cache to reduce the number of load data value traces emitted by the target platform. Usage of cache coherence protocol and inheriting tracking bits eliminate redundant trace messages reporting shared data by different processor cores. Thus, *mc²RT* with variable encoding mechanism reduces the average trace port bandwidth by 9.9 to 46.4 times when N=1 and 18.6 to 61.4 times when N=8 relative to Nexus-like load data value traces.

mc²RFiat is a hybrid technique that uses the best qualities from *mlvCFiat* and *mc²RT*. Thus, it uses cache coherence protocol along with first-access tracking bits to reduce the number of load data value traces emitted by the target platform. This technique requires hardware support to inherit first-access tracking bits from other caches. *mc²RFiat* with variable encoding mechanism reduces the average trace port bandwidth by 14.7 to 54.7 times when N=1 and 25.3 to 67.7 times when N=8 relative to Nexus-like load data value traces.

The future work will focus on expanding the existing work in several directions. First, an analysis of hardware overhead for the proposed techniques can be carried out through the development of detailed hardware models. Whereas, all techniques require relatively small overhead due to tracking bits associated with level 1 data caches, *mc²RT* and *mc²RFiat* require extra control lines on the cache-coherent bus that carry tracking bits during cache-to-cache transfers. An additional aspect of the hardware overhead is determining the maximum size of trace buffers that keep trace messages before they are traced out.

Another promising venue for future work is to consider expanding the proposed technique to high-end embedded processors with private second-level data caches. This change may further improve effectiveness of the proposed techniques at the cost of additional hardware complexity.

Third venue of future research may focus on analyzing the impact of various processor models on the required trace port bandwidth in bits per clock cycle. Using more aggressive superscalar processors with higher number of retired instructions per clock cycle will increase the required trace port bandwidth, and thus make the proposed techniques even more valuable.

REFERENCES

[1] "International Technology Roadmap for Semiconductors 2007 Edition." [Online]. Available: https://goo.gl/TdZY52. [Accessed: 08-Apr-2016].

[2] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," 2003. [Online]. Available: http://nexus5001.org/nexus-5001-forum-standard/. [Accessed: 28-Mar-2016].

[3] IEEE, "IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture -Description," 2001. [Online]. Available: http://standards.ieee.org/reading/ieee/std_public/description/testtech/1149.1-1990_desc.html.

[4] W. Orme, "Debug and Trace for Multicore SoCs," 2008. [Online]. Available: https://www.arm.com/files/pdf/CoresightWhitepaper.pdf. [Accessed: 28-Mar-2016].

[5] MIPS, "MIPS PDtrace Specification," 2009. [Online]. Available: http://goo.gl/UwIYGv. [Accessed: 01-Apr-2016].

[6] Infineon, "MCDS - Multi-Core Debug Solution - Infineon Technologies," 07-Dec-2011. [Online]. Available: https://www.ip-extreme.com/IP/mcds.shtml. [Accessed: 01-Apr-2016].

[7] 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.

[8] B. Mihajlović, Ž. Žilić, and W. J. Gross, "Architecture-Aware Real-Time Compression of Execution Traces," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 4, p. 75:1–75:24, Sep. 2015.

[9] N. Stollon, *On-Chip Instrumentation: design and debug for systems on chip*. New York: Springer, 2011.

[10] 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.

[11] E. A. Daoud and N. Nicolici, "Real-Time Lossless Compression for Silicon Debug," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 1387–1400, Sep. 2009.

[12] E. E. Johnson, "PDATS II: improved compression of address traces," in *Proceedings of the IEEE International Performance, Computing and Communications Conference*, Phoenix, AR, 1999, pp. 72–78.

[13] J. R. Larus, "Whole Program Paths," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, New York, NY, USA, 1999, pp. 259–269.

[14] 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.

[15] A. Milenkovic and M. Milenkovic, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Trans. Model. Comput. Simul.*, vol. 17, pp. 1–27, 2007.

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

[17] A. Milenkovic and M. Milenkovic, “Stream-Based Trace Compression,” *IEEE Computer Architecture Letter*, vol. 2, pp. 9–12, 2003.

[18] A. Milenkovic and M. Milenkovic, “Exploiting Streams in Instruction and Data Address Trace Compression,” in *Proceedings of the IEEE International Workshop on Workload Characterization, 2003*, Austin, TX, 2003, pp. 99–107.

[19] M. Burtscher, “VPC3: A Fast and Effective Trace-Compression Algorithm,” *SIGMETRICS Perform. Eval. Rev.*, vol. 32, pp. 167–176, 2004.

[20] 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.

[21] 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 of the great lakes symposium on Great lakes symposium on VLSI*, New York, NY, USA, 2011, pp. 331–336.

[22] 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.

[23] M. Milenkovic, A. Milenkovic, and M. Burtscher, “Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces,” Snowbird, UT, 2007, pp. 55–65.

[24] 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 (ICCD'09)*, 2009, pp. 173–178.

[25] A. Milenković, V. Uzelac, M. Milenković, and B. 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.

[26] 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.

[27] 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 for Embedded Systems (CASES'10)*, Scottsdale, AZ, 2010, pp. 97–106.

[28] A. K. Tewar, "Experimental Evaluation of Techniques for Capturing and Compressing Hardware Traces in Multicores," University of Alabama in Huntsville, Huntsville, AL, USA, 2015.

[29] C. Hochberger and A. Weiss, "Acquiring an exhaustive, continuous and real-time trace from SoCs," in *IEEE International Conference on Computer Design, 2008. ICCD 2008*, Lake Tahoe, CA, 2008, pp. 356–362.

[30] 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.

[31] 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.

[32] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," *IEEE Trans. Comput.*, vol. 55, pp. 174–184, 2006.

[33] M. Ponugoti, A. K. Tewar, and A. Milenkovic, "On-the-fly load data value tracing in multicores," in *Proceedings of the International conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'16)*, Pittsburgh, PA, 2016.

[34] M. Ponugoti and A. Milenkovic, "Exploiting Cache Coherence for Effective On-the-Fly Data Tracing in Multicores," in *Proceedings of the IEEE International Conference on Computer Design (ICCD'16)*, Phoenix, AZ, 2016.

[35] V. Uzelac and A. Milenkovic, "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging," *ACM TECS*, vol. 12, no. 2s, pp. 1–18, May 2013.

[36] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham MA: Morgan Kaufmann/Elsevier, 2012.

[37] 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*, New York, NY, USA, 2012, pp. 335–344.

[38] 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.

[39] "Multi2Sim/m2s-bench-splash2," *GitHub*. [Online]. Available: https://github.com/Multi2Sim/m2s-bench-splash2. [Accessed: 01-Apr-2016].