HARDWARE DATA VALUE TRACING IN MULTICORES

by

MOUNIKA PONUGOTI

A DISSERTATION

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

HUNTSVILLE, ALABAMA

2019

In presenting this dissertation in partial fulfillment of the requirements for a Doctor of Philosophy 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 dissertation.

_____ _____

(student signature) (date)

DISSERTATION APPROVAL FORM

Submitted by Mounika Ponugoti in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation 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 dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering.

(Dr. Aleksandar Milenkovic) (date)	Committee Chair
(Dr. Rhonda Gaede) (date)	Committee Member
(Dr. Mohammad Haider) (date)	Committee Member
(Dr. Tauhidur Rahman) (date)	Committee Member
(Dr. Earl Wells) (date)	Committee Member
(Dr. Ravi Gorur) (date)	Department Chair
(Dr. Shankar Mahalingam) (date)	College Dean
(Dr. David Berkowitz) (date)	Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Doctor of Philosophy College/Dept. Engineering/Electrical & Computer Engineering

Name of Candidate Mounika Ponugoti
Title Hardware Data Value Tracing in Multicores

Embedded computing systems powered by modern systems-on-a-chip (SoCs) running sophisticated software applications are indispensable in modern communication, transportation, infrastructure, medicine, military, and entertainment. Semiconductor technology trends have enabled the design of SoCs that often combine multiple processor cores, hardware accelerators, direct memory controllers, and peripheral interfaces, all connected through an on-chip interconnect. Faster, cheaper, and smaller SoCs have in turn enabled new applications that may have tens of millions of lines of code. These trends have led to rising software engineering costs in modern embedded systems that currently account for over 80% of the total engineering costs. Recent studies found that software developers spend over 50% of their time finding and fixing software bugs. Thus, software debugging is one of the most challenging aspects of embedded system development due to growing hardware and software complexity, limited visibility of system components, and tightening time-to-market. Providing powerful debugging tools to software developers is thus critical to expedite software development and improve software reliability.

To find software bugs faster, developers often rely on on-chip trace modules with large buffers to capture program execution traces with minimum interference with program execution. However, high volumes of trace data and the high cost of

trace modules limit visibility into the system operation to only short program segments that are often insufficient to locate software bugs. This dissertation introduces a new hardware/software technique for capturing and filtering read data value traces in multicores that enables a complete reconstruction of parallel program execution. The proposed technique called *mcFiltrate* (***m****ulticore* ***c****ache* ***filt****ered* ***r****ead d****a****ta* ***t****rac****e***) utilizes tracking of data reads in data caches and cache coherence protocol states to minimize the number of trace messages that needs to be streamed out of the target platform to the software debugger. The effectiveness of the proposed technique is determined by analyzing the required trace port bandwidth and trace buffer sizes as a function of the data cache size and the number of processor cores. The experimental environment utilizes architectural execution-driven simulator running benchmarks from two suites, Splash2 and Parsec. The results of the experimental evaluation show that the proposed technique significantly reduces the required trace port bandwidth, from 12.2 to 59.6 for a single core processor and from 13.4 to 73.9 times for an octa core processor, when compared to the state-of-the-art Nexus-like read data value tracing. The proposed technique reduces the requirements for the on-chip trace buffers by several orders of magnitude and the number of required trace port pins by up to 16 times when compared to the state-of-the-art Nexus-like tracing. Consequently, *mcFiltrate* enables continuous on-the-fly data tracing while requiring modest changes on the hardware platform and the software debugger.

Abstract Approval: Committee Chair

(Dr. Aleksandar Milenkovic)

Department Chair

(Dr. Ravi Gorur)

Graduate Dean

(Dr. David Berkowitz)

This dissertation is dedicated to the love and energy that made it possible

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Aleksandar Milenkovic. I would not be in this position without his continuous support, guidance, and encouragement. He always inspired me with his patience even in the very odd times. He has shared profound knowledge with me since I started working with him. I feel lucky to work with a professor like him who can understand the educational background and personality of the student from different countries and guide them accordingly. I sincerely hope to be like him professionally and personally in my future and continue to collaborate with him.

I would like to thank Dr. Rhonda Gaede, Dr. Mohammad Haider, Dr. Tauhidur Rahman, and Dr. Earl Wells for serving on my committee. I would like to thank Dr. Ravi Gorur, Chair of the Electrical and Computer Engineering Department, for supporting me financially with a teaching assistantship. I would like to thank teaching and non-teaching staff members for teaching me valuable skills and helping whenever required during my time at the University of Alabama in Huntsville.

Also, I would like to thank Mr. Igor Semenov, Mr. Prawar Poudel, and Mr. Ranjan Hebbar for sharing their knowledge and answering questions whenever I ask them without judging the level of the question. I would like to thank Mr. Amrish K. Tewar who always inspires and teaches me new things whenever I approach him.

Finally, I would like to thank Mr. Srinivas R. Mynampally and his family for taking care of me since my arrival to the USA. I would like to express my deepest gratitude to my parents, Bhagavanth R. Ponugoti and Vimala Ponugoti, and my husband, Vamshi K. Vanapally, for their unconditional love and support.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Embedded systems are computer systems designed to meet the computational requirements of a specific task. They are typically a part of a bigger cyber-physical system, embedded in the environment they operate in, sensing the environment, processing the information, storing information, communicating with other systems, and acting upon the environment. Energy efficiency, low cost, and small form factors are often primary design constraints for embedded systems. They range from low-end to high-end and are found in many application domains, such as medical, transportation, military, industrial, and consumer applications.

Every product has a time window to launch in the market. If the product misses that crucial time window, the sales targets may be missed, resulting in financial losses and missed opportunities. Thus, it is extremely important to reduce the cost and time-to-market, while meeting or exceeding user expectations. To meet the growing performance requirements while maintaining the energy efficiency, hardware designers are incorporating complex hardware structures with multiple processor cores, on-chip interconnect, hardware accelerators, memory controllers, and a range of input/output interfaces on a single chip. As a result, the complexity of modern systems-on-a-chip is increasing, which in turn results in diminished visibility of the system internals.

At the same time, software complexity is also growing rapidly to support higher levels of functionality to meet user expectations and win market share. For

example, today's high-end cars include many functions which were not possible a decade ago. To achieve this, they may run over 100 million lines of code [1]. As we move toward autonomous cars, the code size will continue to increase. Due to shorter development cycles, companies are failing to test the software rigorously to provide reliable software. A recent study shows that recalls in medical devices [2] [3] and automotive devices [4] [5] [6] are mainly due to software bugs. Frequent recalls of products to fix software bugs often hurt companies' reputations, customer trust, and stocks revenue in the long term. Software bugs in cyber-physical systems are even more challenging as they may lead to significant damage and even loss of life [7].

A study from the Judge Business School at the University of Cambridge [8] shows that software developers spend 50% of their development time finding and fixing software bugs, incurring costs of around $312 billion per year. These costs are likely to increase further due to growing system complexity. In addition, more than 80% of development costs in modern embedded systems come from software engineering [9]. With the increased complexity of hardware and software, challenges and costs incurred by software debugging are continually increasing. The current market conditions and trends make it impossible to redesign the development cycles to accommodate additional time for software testing to deliver bug-free software. Hence, it is imperative to provide better tools for software developers to use to locate bugs in the software quickly and cheaply.

The most common debugging techniques include printf debugging, software tracing, and run-control debugging. Printf debugging adds temporary printf statements to track the flow and values used in sections of code under debug. This technique is effective at finding software bugs related to algorithms. However, it is a time-consuming process and that is not feasible when the source code reaches mil-

lions of instructions. Moreover, it is inefficient for finding problems related to time sensitivity, memory allocation, and interrupts. Additionally, outputting the data to a host computer over any communication port can be slow and may impact the behavior of the program. All these reasons make printf debugging not suitable for real-time embedded systems.

Tracing is a method of recording the details of the program as it executes. In software tracing, static and/or dynamic binary and/or source code instrumentation is used to monitor events of interest at the application and/or kernel level. The recorded trace data is either stored in the system memory or streamed through a communication port to the host. The traces are later used for debugging, validation, performance analysis, and performance optimization. Popular examples of software tracing tools include Ftrace [10], Dtrace [11], LTTng [12], eBPF [13], Systemtap [14], WPP [15], Pin [16], among others. Ideally, software tracing can be used in any system as it does not require an external trace debugger. However, instrumentation may affect the performance and/or execution flow of the system/program under inspection. Moreover, these methods are unavailable or ineffective in helping the developers to pinpoint hard-to-find software bugs on resource constrained platforms.

Run-control debugging gives better control over processor execution and is widely used by embedded software developers. In this type of debugging, developers can get control over the program to run step by step, to set watch points or break points, to change, or to observe the content of registers and memory locations when the processor is halted. However, run-control debugging is not practical in many real-time applications – e.g., engine controls, hard disks, robotics, avionics, and automotive, or any application where the processor must run continuously to maintain mechanical stability. First, the sequence of events on the target platform may

change due to run-control debugging, and in the case of multicores, it may not be practical to halt all the processors at the same time. Second, the problems related to external events which are caused when the system is running at full speed cannot be identified. Finally, there is no support to examine the history of the program and thus visibility into the system is either limited or absent when the application is executing. Because of the dependence on external peripherals and the lack of bug reproducibility, run-control debugging is insufficient for real-time embedded systems.

To address this problem, modern processors and systems-on-a-chip (SoCs) include dedicated hardware resources to support hardware tracing and debugging. In hardware tracing, a processor discloses the detailed and accurate internal information at the instruction level granularity with minimal or no performance degradation.

1.1 Motivation

The most common types of hardware traces that can be collected from a processor are control-flow and data-flow. A control-flow trace records the execution path of the program and helps developers understand how the program reached a certain point in execution. It captures information related to branches, subroutine calls, returns, and exceptions. A data-flow trace captures the address and/or value of every memory read and write instruction. Data-flow traces enable developers to replay the program offline and locate bugs in the program under test.

Existing trace modules produce several MB of trace data in a second and they require an average trace port bandwidth of ~0.3 bits per instruction executed for control-flow tracing [17], and 8-16 bits per instruction executed for data-flow tracing for a single core [18]. Since the transfer rate of trace data off-chip is slower than the

rate at which the trace data is produced by the processor, dedicated on-chip trace buffers are used to temporarily hold the trace data. A 2 KB dedicated on-chip trace buffer can hold control-flow traces for about 54 kilo instructions and data-flow traces for about 2 kilo instructions on a single core. In the case of multicores, the buffer space is shared among cores, plus additional bits are required to report the core id. Thus, existing commercial trace modules allow the user to set filters to capture data-flow traces for only a limited range of addresses or instructions. Unfortunately, the traces captured in trace buffers on limited program segments may not be sufficient to locate bugs as a software bug origin and its manifestation may span millions of instructions. To capture data-flow traces for an entire program, it is required either to halt the program when the trace buffer is full or use deep on-chip trace buffers or wider trace ports (i.e. dedicated physical pins) to empty the trace data faster. However, these options are not attractive since deep on-chip trace buffers or wider trace ports increase the system cost for the end user and halting the processor in real-time systems is not feasible.

1.2 *mcFiltrate*: Multicore Cache-Filtered Read Data Trace

This dissertation introduces and evaluates *mcFiltrate,* a new technique for capturing and filtering read data value traces in multicores. *mcFiltrate* stands for ***m****ulti****c****ore* ***filt****ered memory* ***r****ead d****a****ta* ***t****rac****e***. This technique captures and filters the memory read data traces by using L1 data caches and first-access tracking bits on the target processor core. In addition, it requires the software debugger to maintain L1 data cache structures in software identical to those in the target core. The main goal of this technique is to filter out redundant trace messages that can be inferred by the software debugger. The first-access tracking bits associated with sub-blocks of

an L1 data cache block (or line) are used to determine whether the trace data can be inferred by the software debugger. In addition, *mcFiltrate* can exploit cache coherence protocols to minimize the number of redundant trace messages for cache blocks that are actively shared by multiple processor cores.

This dissertation experimentally evaluates the effectiveness of *mcFiltrate* relative to the state-of-the-art Nexus-like data tracing as a function of the number of cores and cache configurations. As metrics of interest, the average and dynamic required trace port bandwidth measured in bits per instruction executed (bpi) and bits per processor clock cycle (bpc) are used. In addition, the worst-case analysis concerning the maximum depth of the trace buffer as a function of trace port emptying rate is performed. The results show that *mcFiltrate* offers a significant reduction in the required trace port bandwidth relative to the existing Nexus-like memory read data value tracing. It reduces the trace port bandwidth in the range of 13.4 to 59.6 times for a single core processor, and from 12.2 to 73.8 times for an octa core processor, depending on the size of data caches.

1.3 Main Contributions

The main contributions of this dissertation are as follows:

- It characterizes trace port bandwidth requirements in bits per instruction and bits per clock cycle in multicores for Nexus-like timestamped read data value traces. The results demonstrate that this type of tracing is practical for only very short program segments and that it becomes cost-prohibitive in both required trace port bandwidth and the number of dedicated trace port pins as program size increases.

- It introduces a hardware/software technique called *mcFiltrate* to capture and compress memory read data value traces in multicores. This technique relies on first-access tracking bits attached to L1 data cache blocks and cache coherence protocol states to determine when memory reads need to be traced out to ensure that the program is replayed offline faithfully. This technique is an extension of previous research [19][20][21].
- It experimentally evaluates the trace port bandwidth required by *mcFiltrate*, while varying the number of processor cores and data cache sizes, when running parallel benchmarks from Splash2 [22] and Parsec suites [23].
- In addition to analyzing the average required trace port bandwidth, dynamic changes in the trace port bandwidth requirements during benchmarks' execution are evaluated. Moreover, a detailed analysis is performed to determine the maximum required size of the trace buffers needed for on-the-fly tracing while varying the actual trace port bandwidth.
- To further reduce the required trace port bandwidth, a dictionary-based compression technique for Nexus-like and *mcFiltrate* traces is proposed. The results of the experimental evaluation show that even though overall trace compression achieved with dictionaries are modest, some individual benchmarks benefit significantly.

1.4 Dissertation Outline

The remaining of the dissertation is organized as follows: CHAPTER 2 briefly discusses various software and hardware tracing techniques. It describes the challenges in data tracing using an analytical model. CHAPTER 3 reviews the literature and state-of-the-art solutions for compressing trace data. CHAPTER 4 introduces *mcFiltrate* and details its operation on a target core and on the software debugger side. CHAPTER 5 discusses the experimental environment used in the evaluation and CHAPTER 6 discusses the results from the experimental evaluation. CHAPTER 7 discusses the dictionary analysis which can be used to further reduce the trace port bandwidth requirements and CHAPTER 8 summarizes this work.

CHAPTER 2

BACKGROUND AND MOTIVATION

Tracing can be treated as a slow-motion video of the real-time program execution. It involves recording program details that can be used for debugging, performance analysis, performance optimization, or security analysis. Depending on how these traces are recorded, two types of tracing exist: (a) software tracing, where software is responsible for capturing and handling traces; and (b) hardware tracing, where dedicated hardware resources are responsible for capturing traces.

In software tracing, the operating system and/or user application programs are instrumented statically and/or dynamically with code to capture and record traces. With instrumentation, any instruction can be trapped to collect the required information. Depending on the configuration, a typical software trace can record various details such as current process identifier (pid), call arguments, stack trace, time, return value of system calls, register values, instruction addresses, and other data. More details about software tracing and the most widely used software tracing facilities at kernel level and application level are discussed in Section 2.1.

In hardware tracing, a processor gives the complete history of the executed instructions with minimal or no overhead. Ideally, the real-time behavior of the program is not affected by the tracing operation. Depending on the level of support and required type of trace, a processor can record various details such as time, processor/thread identifier, instruction addresses, memory values and memory addresses in the case of memory accessing instructions, and other information for executed instructions. Section 2.2 describes hardware tracing and introduces terminology used

throughout the dissertation. Section 2.3 discusses existing commercial hardware tracing solutions. Finally, Section 2.4 discusses how hardware traces can be used in debugging and why it is important to filter and/or compress hardware data traces.

2.1 Software Tracing

In software tracing, the operating system, a program under debug by itself, or an observer program records the trace data. To record the traces, instrumentation of the source code or binary of the operating system and/or user application program is required. Depending on when the instrumentation is performed, it can be classified into: (a) static instrumentation or (b) dynamic instrumentation. With static instrumentation, the kernel or a user application program includes the dedicated code which helps in recording the traces when enabled. With dynamic instrumentation, instrumentation code is inserted or attached dynamically at runtime during the execution of the application.

A probe point is a debug statement that assists in collecting execution characteristics of a program; e.g. the state of the executing program is captured when the probe point is reached [24]. Probe points can be inserted and enabled statically and dynamically. With probe points, the kernel or a user application program is instrumented with code and this instrumentation code is run when the specified probe point is executed by the processor, i.e. when the probe is fired. However, there are other user space dynamic binary instrumentation frameworks where application binary is modified on-the-fly and modified code is executed to capture the details of the execution [16] [25].

Static probe points are inserted by the kernel or application developer at important locations while writing the code. To enable or insert static probe points, e.g.,

inserting *printk* statements, recompilation of the source code is required. However, some static probe points can be enabled or disabled dynamically and a probe handler function can be attached to record the data. Dynamic probe points can be inserted at any address in the kernel and application code without the need for recompilation of the source code. In dynamic probing, a trap instruction is inserted at the desired locations. When the trap instruction is executed, the interrupt is generated and the corresponding handler for that probe is called. Once the handler is finished, the original code continues with normal execution. In dynamic probing the overhead for each instrumented site execution is larger than for static probing because of the trap mechanism. In Linux, tracepoints, kprobes, uprobes, and USDT probes are the main sources of collecting information. Static kernel space probe points are referred to as tracepoints and static user space probe points are referred to as USDT probes (User Statically Defined Tracing points). Kprobe and uprobe are dynamic methods of inserting probe points to any kernel space and user space code, respectively.

There are many open source tools and frameworks available to collect and analyze information from kernel and user application programs. Some of these frameworks support frontend tools to provide an interface (scripting or programming), to insert dynamic probe points, and to enable or disable some of the static probe points. They differ in the level of detail that they can capture, sources of information, and how they handle setting the probes, executing the probe handler, data collection, and frontend tool interface if the probe points are data sources (Section 2.1.1).

2.1.1 Software Tracing Frameworks

Ptrace. *Ptrace*, process trace, is a system call in Unix and Unix-like operating systems that enables one process (controller process, parent) to observe or control the execution of another process (target process) [26]. It can be attached or detached from the process being traced at any time. With *ptrace*, the ability of the control process to change registers and memory of the target process allows it to set breakpoints, run step-by-step, and inject code to a running target process. The controller process can observe and intercept the system calls and signals to and from the target. Thus, *ptrace* is used by debuggers – *gdb* and *dbx*, tracing tools – *strace* and *ltrace*, and by code coverage tools. Since the communication between the control process and the target process requires at least two context switches, the performance overhead of *ptrace* is significant. It is supported by OpenBSD, FreeBSD, NetBSD, IBM AIX, and Linux.

Strace. *Strace,* system call tracer, is a lightweight diagnostic, debugging and instructional user space utility for Linux to trace the interactions of a program with the operating system [27]. *Strace* uses the *ptrace* system call to register itself as a control process (tracer). It is notified of all the system calls and signals and can get the details such as, the type of a system call, arguments passed, stack of the target process, return value, and the time spent in the system call. This is quite useful when debugging the programs where source code is not available.

Ltrace. *Ltrace,* library call tracer, is used to trace calls and returns from a dynamic library function executed by the target process [28]. In addition, it can also trace system calls and signals like *strace*. *Ltrace* uses *ptrace* to place the breakpoints by hooking into the procedure linkable table (PLT). It is supported by Linux.

Ftrace. *Ftrace*, function tracer, is a tracing framework built into the Linux kernel to understand the internals of the kernel [10]. This utility can be used in debugging and to analyze latencies and performance issues which come from kernel space. Although the name of the tracer is function call tracer, the framework includes several tracing utilities. With the hundreds of available tracepoints (static events) that can be enabled via the *tracefs* file system in the kernel, *ftrace* can trace scheduling events, interrupts, virtual guest connections with host, and file systems. More elaborately, it can disclose the information to answer the questions such as what happens when the interrupts are disabled and enabled and when is the task actually scheduled after waking up. The *dynamic ftrace* can trace any functions of the kernel and it also allows the filtering of functions by using *globs*. It can generate call graphs and provide stack usage reports. It can consume tracepoints, kprobes, and uprobes trace sources.

LTTng. *LTTng*, Linux Trace Toolkit Next Generation, is a set of LTTng-tools, LTTng-UST, and LTTng-modules tools [12]. These tools can be used to instrument the Linux kernel (LTTng-modules, includes the modules to instrument and trace the Linux kernel), user applications (LTTng-UST), and to trace control e.g., starting and stopping of the tracing and enabling and disabling of event rules. The optimized events in *LTTng* have low overhead compared to existing trace solutions and can be used to analyze the scheduling decisions, context switches, execution of various background tasks, actual execution time of the process and blocked time. It is supported by Linux and FreeBSD.

Dtrace. *DTrace*, Dynamic tracing, was developed by Sun Microsystems to examine the behavior of applications and the kernel on production systems in real-

time [11]. DTrace can be configured to record the additional data like function arguments, stack trace, and other data whenever the probe of interest is fired. The probes used by *DTrace* are not defined by the tool itself, rather, they come from the kernel modules called providers. Each provider is independent and reports the list of data points (set of probes) that it can instrument. All the instrumentation in *DTrace* is dynamic, i.e. the probes are enabled only when they are utilized – there is no overhead when they are disabled, and no instrumentation code is added for the inactive probes. D language (inspired by C and awk) is used to write the scripts which include the list of probes to enable and the associated action of the probe, probe handler. The scripts are converted to a simplified instruction set, Dynamic Intermediate Format (DIF), by using the compiler which is in *libdtrace* library and they are interpreted by the virtual machine at the kernel level when the probe fires. *DTrace* is supported on Solaris, Illumos, MacOS, FreeBSD, NetBSD, Oracle Linux, and Microsoft Windows.

eBPF. *eBPF*, extended Berkeley Packet Filter, is an in-kernel virtual machine native to Linux [13]. In the original BPF, network packets are captured and filtered by attaching a filter program which runs on a virtual machine to any socket. *eBPF* is enhanced over BPF to give better performance through an expanded set of registers, instructions, helper functions that can be called inside the programs. Additionally, global storage called *eBPF* maps allow sharing of data between *eBPF* kernel and user space programs and between *eBPF* kernel programs. The *eBPF* programs written in C are converted to bytecode and attached to the path of the execution, kprobes, uprobes, and tracepoints as in *DTrace*. The *eBPF* virtual machine per-

forms a sanity check to ensure security before executing the bytecode. With *eBPF*, applications can be traced with low overhead.

Sysdig. *Sysdig* is a high-performance system call tracer which also supports tracing of containerized processes [29]. Instrumentation is done by using a device driver as an external kernel module, sysdig_probe. *Sysdig* uses static tracepoints to intercept system calls and does not support kprobes, uprobes, and USDT. However, a newer version of *sysdig* incorporates support for using *eBPF* for event collection as a backend instead of a kernel module. In addition to tracing, *Sysdig* includes many diagnostic tools such as tcpdump, strace, fuser, lsof, iostat, htop, lspci, ethool, and netstat.

SystemTap. SystemTap is a tool for dynamically instrumenting a Linux kernel based operating systems to monitor the kernel [14]. The scripts are written in the *SystemTap* language; however, they are translated to C language to create a kernel module from it. *SystemTap* does not have in-kernel virtual machine; instead, kernel modules are loaded dynamically. When the event specified in the script occurs, the Linux kernel runs the corresponding handler as a quick sub-routine. Probes in the script are disabled when the *SystemTap* session expires. It can consume, kprobes, uprobes, kernel tracepoints, and USDT.

ETW. ETW, Event Tracing for Windows, is a tracing facility on the Windows operating system which allows logging the events of user applications and kernel device drivers [15]. The application which is to be traced should contain event tracing instrumentation. Even though *ETW* allows tracing to turn on or off dynamically, it does not have the ability to insert trace points at runtime. The events – metadata, localizable message strings, and schematized data payloads are logged to a separate

buffer for each processor stored on disk and/or delivered to the consumer in real-time.

Pin. Pin is a dynamic binary instrumentation framework that allows analysis of user space applications on Linux, Windows, and OS X [16]. It provides APIs to write instrumentation tools called *pintools* in C/C++ to profile or trace an application [25]. *Pin* APIs allows *pintools* to access architecture specific details and support IA-32, x86-64, and MIC instruction-set architectures. The injector loads the *pin* binary into the address space of an application by using the Unix *Ptrace* API and starts it running. *Pin* intercepts the execution of the application at the very first instruction and loads the *pintool*. Once the *pintool* initializes itself, it requests *pin* to start the application. Depending on the instrumentation APIs used in *pintool*, *pin* generates new instrumented code for one trace at a time using just-in-time (JIT) compiler by feeding actual executable as input. A trace is defined as a set of consecutive instructions which terminates at one of the conditions: (a) an unconditional control transfer (branch, call, or return), (b) a pre-defined number of conditional control transfers, or (c) a pre-defined number of instructions have been fetched in the trace. After generating new code, *pin* transfers control to the generated sequence and it make sure to regain the control when the branch exists the sequence. After regaining control, *pin* generates more code for the branch target and continues execution. To speed up the instrumentation, *pin* employs various optimization techniques such as code cache, trace linking, inlining, register re-allocation, liveness analysis, and instruction scheduling.

In software tracing, event tracing is most widely used since collecting traces for every instruction gives the greater performance overhead. Even though software

tracing provides high flexibility to analyze and debug the applications and operating system, these methods are not suitable for real-time embedded system where very intrusive software modules for tracing interfere with normal program behavior. Adding instrumentation code may increase the footprint of the executable in memory, may affect cache access patterns, and is intrusive. Moreover, most of these frameworks cannot be used to debug the firmware and boot time issues because the necessary libraries are not initialized.

2.2 Hardware Tracing

Hardware traces provide a complete history of the instructions executed on a processor and are collected by dedicated hardware trace modules in modern processors. Thus, tracing a processor gives better visibility of the system at any given point than does software. The beauty of hardware tracing is that information is captured with zero or very low overhead as it uses dedicated hardware circuitry. However, hardware tracing produces vast amounts of data in a short period of time.

2.2.1 Types of Hardware Tracing

Depending on the type of information collected by the trace module, hardware tracing can be classified into two types, control-flow tracing and data-flow tracing. A control-flow trace records information related to control-flow instructions and a data-flow trace records information related to memory read and write instructions executed by a processor.

Control-Flow Tracing. A basic block is a portion of the program with a single entry and a single exit. Branch instructions transfer control from the exit of one basic block to the entry of another basic block. These are usually used to analyze

program behavior, for example, to find hot regions [30], in security (control-flow integrity), and by compilers (profile guided optimization).

Tracing each and every executed instruction to reconstruct the flow of the program is expensive as it produces a lot of trace data. In most cases, it is sufficient to record changes in the path of the program (basic block entries) to determine and reconstruct the actual execution. Hence, a control-flow trace involves recording traces for jumps, calls, returns, interrupts, and exceptions, while executing on the target processor. State-of-the-art commercial tracing solutions support capturing the control-flow traces of a program and compress the trace data by filtering the statically available data from the program binary, for example, direct unconditional branches.

Control-flow traces can be used in debugging a program to understand how it reached any given execution point or what path it took to arrive to the given execution point [31] [32]. These traces can also be useful to profile the instructions, detect latencies [33], perform path based optimizations, path sensitive prediction techniques such as branch prediction, to understand the behavior of the malware, and in security [34] [35] [36] [37] [38]. Additionally, these traces can be used to identify the isolating bugs while varying the input since the program behavior changes.

Data-Flow Tracing. Control-flow traces are helpful for uncovering various types of bugs. However, control-flow traces alone cannot help to uncover the problems related to memory accesses, such as data race conditions or memory access patterns in the case of multicores.

A data-flow trace records information about memory read and memory write instructions. A full data-flow trace records instruction address, addresses of memory read and memory write operation, size of operand, and value of memory read and write operation. However, depending on the intended use and tracing requirements,

a subset of these fields can be traced out. For example, the addresses of memory reads and writes are useful (a) to analyze cache performance, internal memory, and data transfer operations, (b) to identify data locality, which can help to find cache conscious data layouts, (c) to develop data prefetching techniques, and (d) to find data race conditions in multicores. The data values are significant in enabling faithful replay of the program offline. Replay of the program enables software developers to find bugs with minimal effort as they can move forward and backward in time. In addition, it is really important to know what data value a processor is reading at a given point of time in security (data integrity) [39] and to solve problems with external inputs in real-time systems. For example, in the case of a sensor malfunctioning, it is not possible to determine the cause for the crash without actually knowing the value it read [40].

2.2.2 Trace and Debug Infrastructure

Figure 2.1 shows a typical multicore system-on-a-chip (SoC) with trace and debug infrastructure. It has *N* processor cores, a DSP core, and a DMA core. All the cores are connected through a system interconnect. Each processor core is connected to a private trace module for on-chip tracing and debugging support. This trace module is responsible for capturing and possibly filtering the trace data. As processors run at very high speeds, megabytes of trace data are generated within a fraction of a second making it difficult to emit the trace data off-chip.

To continuously emit trace data for longer periods of time, a dedicated high-speed trace port (parallel or serial) is used. However, dedicated trace ports require dedicated physical pins which increases the system cost. An external trace probe that reads traces from the target system includes large buffers, typically in Giga-

bytes and it is connected to the development workstation via standard interfaces such as USB or Ethernet [41] [42] [43] (see Figure 2.1). Most commonly available commercial trace probes are HSSTP (High Speed Serial Trace Probe) from ARM [44], SuperTrace Probe from GreenHills [41], PowerTrace Serial from Lauterbach [42], and Ultra-XD from Ashling [45].

To reduce the packaging size and/or cost introduced by the number of dedicated physical pins to stream the generated trace data off-chip, on-chip trace buffers are used to store the trace data temporarily. In that case, data from the trace buffer is either streamed through a low speed trace port like JTAG or a low pin-count trace port or copied to the target's system memory with the help of the operating system. The latter approach burdens the memory bus and system memory, affecting system performance. Hence, it is not well suited for embedded systems where performance degradation cannot be tolerated, and system memory is limited. In our work, we focus on data tracing through trace ports, though our proposed technique can help reduce the obtrusiveness of tracing in system memory.

The software debugger on the host workstation can use this information to understand the interactions between different software components and answer questions such as

- When are interrupts and context switches occurring?
- What is the execution path of the program?
- What events drive the program to a crash?
- Which parts of the program are taking most of the time?
- What memory address and values are accessed for memory operations?

In addition, replay of program execution allows software developers to move forward and backward in time while investigating bugs.

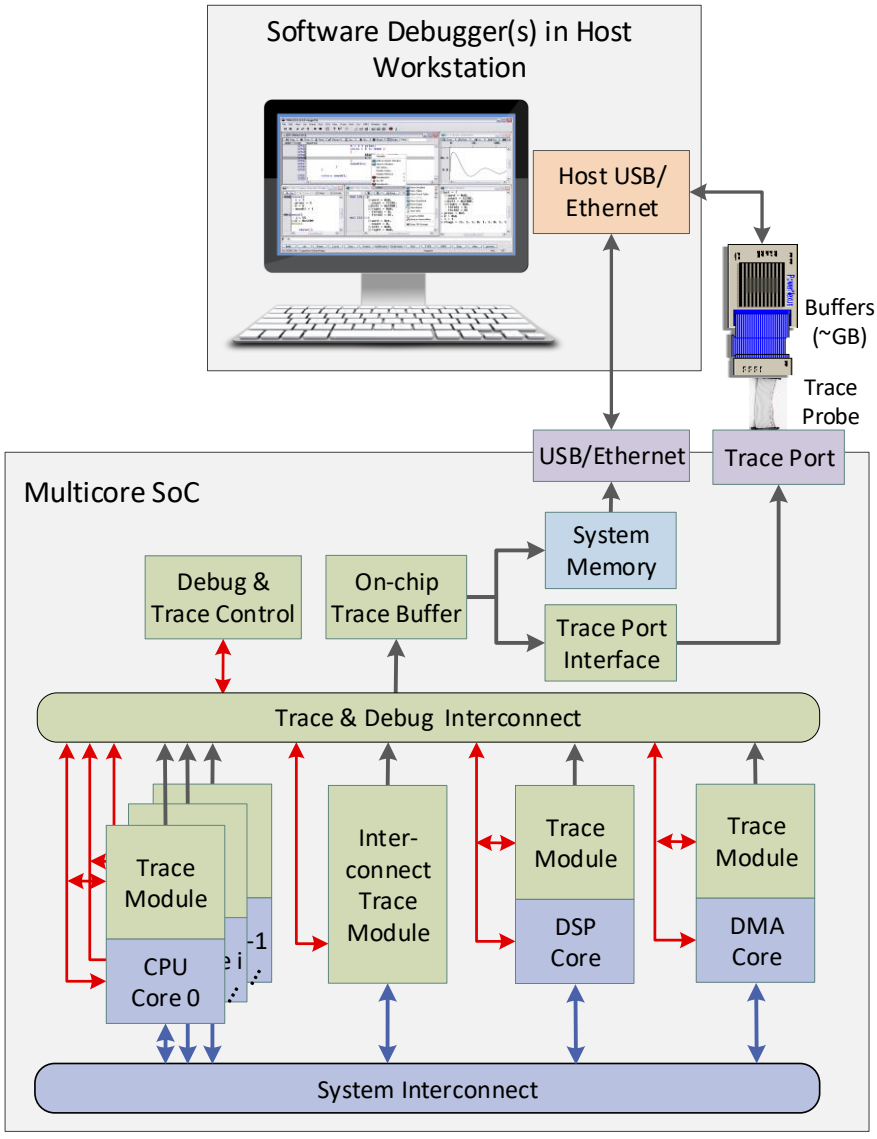


Figure 2.1 Multicore SoC with Trace and Debug Infrastructure

Recognizing the importance of hardware tracing in debugging, hardware vendors are adding on-chip support to capture and emit program traces for offline analysis. State-of-the-art hardware trace capturing technologies include ARM's

CoreSight [46], MIPS's PDtrace [47], Intel's Processor Trace [48], Infineon's Multi-core Debug Solution (MCDS) [49], Synopsys's Real-Time Trace (RTT) [45], Altera [50], NXP [51], and others.

The IEEE Nexus 5001 standard [52] provides a standard interface to trace and debug embedded systems. It is a packet-based protocol and can use either a JTAG port or a high-speed dedicated Aurora or Auxiliary trace port to emit trace messages. It defines four classes of debugging capabilities depending on the level of details traced and exposed. Class 1 supports traditional run-control debugging – single stepping, analyzing and changing the memory values, and setting breakpoints through JTAG. Class 2 supports capturing control-flow traces and Ownership traces (task identifier). Class 3 includes support for capturing memory read and memory write traces. Class 4 adds an advanced capability for emulating memory and I/O accesses through a trace port. Each level progressively supports the features of lower classes. Thus, higher levels require more on-chip resources and wider trace ports, both of which increase system cost.

2.3 State-of-the-art Commercial Hardware Trace Solutions

ARM CoreSight. CoreSight is an advanced trace and debug solution for complex SOC based on the ARM architecture [46]. It is a collection of hardware components for tracing and debugging. Depending on the level of required debugging capabilities, a chip designer can choose individual modules and integrate them in the system. It includes tracing modules such as embedded trace macrocell (ETM), program trace macrocell (PTM), and system trace macrocell (STM). The collected trace data is either stored to a circular on-chip trace buffer, embedded trace buffer

(ETB), or emptied through a high-speed dedicated trace port interface or debug port (serial debug port or JTAG).

ARM STM. The STM module is designed to collect traces of system activity generated by both software and hardware events. A unique pair of master and channel are assigned to each hardware and software trace source. The number of available masters and channels is limited. In auto-instrumentation mode, STM can utilize existing static tracepoints in the kernel and built-in Linux tracing frameworks. By enabling the STM device driver and existing tracing framework [53], all the collected trace information is sent to STM hardware. To instrument applications, an application developer should define the instrumentation statement templates called tracers (like printf) to include in the source code. Tracers can be grouped into subsystems and each subsystem can be assigned to a single STM channel [53]. Every instrumentation statement includes the name, subsystem, and template details. The traces collected by STM can be interpreted and visualized by existing tracing frameworks such as LTTng on the host.

ARM PTM. The PTM module collects control-flow traces of a program [54]. To reduce the amount of trace data while enabling off-line reconstruction of the program flow, PTM avoids outputting the static information available in the program binary, program counter for example, and traces the flow changes such as indirect branches (with address and condition code), direct branches (condition code), context ID changes, exceptions, instruction barriers, changes in the processor instruction and security state, and entry and exit to the debug state. The PTM can collect differential timestamps (the number of clock cycles spent between the current and the previous trace), global system timestamps, and the destination of direct branches, if

required. The trace streams are stored in a FIFO buffer that when full sets an overflow signal that prevents further tracing.

ARM ETM. The ETM module can collect instruction and possibly data traces, memory address and value, for instructions executed by the processor. The ETM can be configured to generate only control-flow traces as in PTM instead of generating the traces for every instruction. The data-flow trace in ETM can be either memory address or memory address and value for load and/or store instructions. To enable efficient use of ETB and increase the trace coverage, ETM incorporates filtering and compression capabilities. For example, while collecting control-flow traces continuously, data-flow traces can be triggered for ranges of address or for a certain address or when the address bus sees a certain data value. In addition, ETM employs multiple trace compression techniques such as encoding multiple traces to a single stream when possible, excluding the program address for all the traces and target of direct branches, and avoiding the traces for some branches, when possible by including the return stack in the trace unit and in the trace analyzer.

Altera Nios II. Like the ARM ETM tracing module, the Nios II trace module supports tracing instructions and data [50]. To reduce the volume of trace data, Nios II provides filtering capabilities similar to ARM ETM. For example, it supports triggers to start and stop capturing control-flow and data-flow traces. The on-chip trace buffer size can be set to any size from 28 to 64 K trace frames, using OCI (on-chip instrumentation) On-chip Trace. A trace frame can store the execution trace for more than one instruction and a frame is defined as a unit of allocated memory for trace data and does not represent the absolute trace depth. The larger trace buffer consumes more on-chip M4K RAM blocks and every M4K RAM block can store up to 128 trace frames.

Infineon MCDS. Multicore Debug Solution (MCDS) is a trace and debug solution for Infineon chips [49]. It complies with the Nexus-5001 standard and supports program and data-flow tracing. Similar to other tracing solutions, MCDS employs trace compression and filtering (called trace qualification) to reduce trace data size. Instead of reporting full addresses, it only reports the changes in the address from last address in case of indirect branches, single bit for direct branches, and suppresses repeated zeros and ones in the MSB for data traces. An exception is synchronization traces which occur every few hundreds of clock cycles. Since the data traces produce large amounts of data, MCDS supports context aware compression techniques, for example, enabling capturing of data-traces when required.

Intel LBR. Last Branch Record (LBR) logs taken branches, interrupts, and exceptions to an LBR stack which is generally a set of model specific registers (MSRs) [55]. For every branch, LBR stores the branch-from address, branch-to address, and some additional metadata depending on the format address stored on the LBR stack. These registers act as a ring buffer and can store up to 32 branches depending on the CPU generation. There is no performance overhead for recording the branches, however, there is some overhead when reading trace records from these registers.

Intel BTS. The Branch Trace Store (BTS) provides additional capability to store branch trace messages (BTMs) to some monitoring device or the system memory allocated by the user. The format of the BTM record is similar to LBR: it includes the branch-from address, branch-to address, and a control word in which bit 4 is used to indicate whether the taken branch was predicted or not predicted. The size and location of the BTS buffer can be specified by the user and it can be configured as circular buffer (similar to an LBR stack) or to trigger an interrupt

when it is nearly full. Thus, BTS can store a larger number of branch records compared to LBR. However, this facility comes with overhead as the processor needs to enter special debugging mode in which processor speed drops by 25-30 times [56].

Intel PT. Intel Processor Trace (Intel PTrace) is an advanced version of Intel BTS which records control-flow traces with low runtime overhead (<5%) [57]. To improve the performance compared to BTS, it stores the encoded and more compact trace packets temporarily in the internal buffer before copying these to the memory subsystem. Intel PT supports various types of packets and only records the minimal information required to reconstruct the program flow. For example, it records only 1-bit to indicate whether the conditional branch was taken or not taken and it excludes the unconditional branches, as this information can be extracted from the program binary. With the support of hardware instruction pointer filtering, it can be configured to record traces for the instructions which are in or out of a range of addresses. Along with control flow traces, it also records certain processor mode changes (32/64-bit execution mode, VMCS pointer, TSX transaction state) and periodic synchronization points (TSC, mode frequency, SW context). It can be configured to report the cycle count either for every instruction in the trace or periodically.

MIPS PDtrace. PDtrace provides instruction and data trace capability for the MIPS architecture [47]. The trace logic on the processor outputs traces to a trace control block (TCB) which is responsible for writing these traces to either on on-chip or an off-chip trace buffer. When the trace buffer is full, it stalls the pipeline. To reduce the amount of trace data, PDtrace only captures the control-flow traces that are further compressed with synchronization trace except at the beginning of the process and after buffer overflow. For unpredictable branches, it reports changes in the PC (difference between the instruction address just before the branch and the branch

target). For conditional predictable branches, it either reports a single bit (taken or not taken). Nothing is reported for direct predictable branches. The synchronization trace addresses are reported periodically. Data address traces are compressed by reporting the delta difference and data values can be compressed with bit-block compression. Moreover, data-flow traces (address and value) can be collected only for the addresses which match with hardware breakpoint addresses.

Synopsys RTT. RTT, Real-Time Trace, is an optional hardware tracing module for Synopsys' ARC based processor cores. RTT generates trace messages which are compliant with Nexus 5001 Class 3. Based on power, area, and performance constraints, the RTT module can be configured at design time to capture different traces necessary for debugging. Trace modules are classified as small trace, medium trace, and full trace depending on their capabilities. Small trace modules record only the changes in the program-flow and use the dedicated on-chip trace buffer to store the traces. A medium trace module records program-flow changes and memory reads and writes and uses either dedicated on-chip memory or system shared memory or external storage on trace probes to save the traces. Full trace modules record auxiliary register reads and writes and CPU register writes in addition to the capabilities of medium trace modules. Full trace module can store trace messages on shared system memory or external storage. To store traces on external trace probes, support for the high-speed dedicated trace port, auxiliary Nexus port, is required. Just like any other trace module, RTT supports address range, data, and address trigger filters to reduce the amount of trace data.

2.4 Motivation

Hardware data traces are often necessary to faithfully replay a program under test offline. Generally, hardware data traces (a.k.a., data-flow traces) are generated by capturing relevant information about memory-referencing instructions, e.g., instruction addresses, data addresses, data values, and data sizes and types. However, not all of this information is necessary to replay a program offline. For example, a software debugger can infer memory write values under certain conditions from the program binary and memory read operations. Thus, to faithfully replay a parallel program offline, a software debugger requires only memory read data value traces along with (a) an instruction set simulator for the target platform; (b) the executable of the parallel program; (c) the initial state of the target's general- and special-purpose registers; and (d) exception traces. The exception traces and the memory read data value traces thus need to be captured on the target platform during program execution and streamed out through the target trace port. The following section illustrates the use of memory read data value traces in software debugging, specifically in the detection of race conditions in parallel programs.

2.4.1 Memory Read Data Value Tracing to Detect Race Conditions

Figure 2.2 illustrates memory read data value tracing with a simple multi-threaded OpenMP C program. The program shown in Figure 2.2a accumulates the scaled elements of an array, *a[]*, to a shared variable *sum*, by using four threads running on four cores (*C0-C3*). Each thread in the program accumulates 13 elements of the array. The shared variable *sum* is intentionally not guarded by any locks to illustrate how memory read data value traces can be useful to detect data race conditions. A data race condition is observed on a shared variable when a memory read

followed by a write operation in one thread is interspersed by a memory read or a memory write operation in another thread on the same variable (e.g., *sum* in this example).

(a) OpenMP C program

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(){
5    int a[52];
6    int sum = 0;
7    const int k = 10;
8    for (int i = 0; i < 52; i++) a[i] = i;
9
10 #pragma omp parallel for num_threads(4)
11   for (int i = 0; i < 52; i++)
12      sum += k*a[i];
13
14   return 0;
15 }
```

(b) x86 assembly code for parallel loop

```
80483b0: mov   DWORD PTR [ebp-0xc],edx
80483b3: mov   ecx,DWORD PTR [ebp-0xc]
80483b6: mov   edx,DWORD PTR [ebp+0x8]
80483b9: mov   edx,DWORD PTR [edx]
80483bb: mov   ecx,DWORD PTR [edx+ecx*4] ; read a[i]
80483be: mov   edx,DWORD PTR [ebp+0x8]
80483c1: mov   edx,DWORD PTR [edx+0x4]
80483c4: mov   edx,DWORD PTR [edx]        ; read k
80483c6: imul  ecx,edx
80483c9: mov   edx,DWORD PTR [ebp+0x8]
80483cc: mov   edx,DWORD PTR [edx+0x8]
80483cf: mov   edx,DWORD PTR [edx]        ; read sum
80483d1: add   ecx,edx
80483d3: mov   edx,DWORD PTR [ebp+0x8]
80483d6: mov   edx,DWORD PTR [edx+0x8]
80483d9: mov   DWORD PTR [edx],ecx        ; write sum
80483db: add   DWORD PTR [ebp-0xc],0x1
80483df: cmp   DWORD PTR [ebp-0xc],eax
80483e2: jl    80483b3
```

(c) Memory read data trace for a snippet of execution

Time	Core 0		...	Core 3		Memory Read Data Traces
[cycles]	read/write	value		read/write	value	[dts, cid, mrv]
t_s+0	read a[0]	{0}				[0, 0, 0]
t_s+1	read k	{10}				[1, 0, 10]
t_s+3	read sum	{530}				[2, 0, 530]
t_s+4	write sum	{530}				-
t_s+72				read a[39]	{39}	[72 , 3, 39]
t_s+73				read k	{10}	[1, 3, 10]
t_s+75				read sum	{530}	[2, 3, 530]
t_s+77				write sum	{920}	-
...						
t_s+1258	read a[7]	{7}				[329, 0, 7]
t_s+1271	read k	{10}		read a[47]	{47}	[13, 0, 10]; [190, 3, 47]
t_s+1272	read sum	{7980}		read k	{10}	[1, 0, 7980]; [1, 3, 10]
t_s+1273				read sum	{7980}	[1, 3, 7980]
t_s+1274	write sum	{8050}				-
t_s+1275				write sum	{8450}	-
t_s+1311				read a[48]	{48}	[38, 3, 48]
t_s+1313				read k	{10}	[2, 3, 10]
t_s+1314	read a[8]	{8}		read sum	{8450}	[42, 0, 8]; [1, 3, 8450]
t_s+1315	read k	{10}				[1, 0, 10]
t_s+1316	read sum	{8450}		write sum	{8930}	[1, 0, 8450]
t_s+1383	write sum	{8530}				-

Figure 2.2 Memory Read Data Value Tracing Example

Figure 2.2b shows the assembly instructions corresponding to the parallel *for* loop. In one iteration of the parallel *for* loop the following memory reads occur: (a) loop counter *i*, (b) element of the array *a[i]*, (c) constant *k*, and (d) variable *sum*. The

memory writes are as follows: (a) updating loop counter *i* and (b) updating *sum*. Figure 2.2c shows the snippet of parallel execution of the *for* loop on cores *C0* and *C3* that execute threads 0 and 3, respectively, starting from the time *ts*. Threads 0 and 3 sum up elements *a[0]-a[12]* and *a[39]-a[51]*, respectively. In Figure 2.2c, the first column shows the timestamp in which the corresponding instruction is retired, and the last column shows the hardware trace messages corresponding to memory reads. Each memory read results in a trace message composed of timestamp (*dts*), thread/core identifier (*cid*), and memory read value (*mrv*) similar to the Nexus 5001 standard [52]. The timestamp, *dts*, reports the number of clock cycles expired since the last trace message is reported from the same core/thread.

In the initial iteration of the *for* loop for *C0*, *a[0]*, *k*, and *sum* are read and *sum* is updated at *ts*+4. The trace messages corresponding to memory read operations are shown in the last column of the first 3 rows. The memory write operation does not result a trace message as it can be inferred by the software debugger while replaying. Similarly, *C3* reads array element *a[39]*, as well as *k* and *sum* and updates the *sum* with a new value at *ts*+77. As we proceed in time, *C0* reads *a[7]*, *k*, and *sum* and writes a new value of the sum at *ts*+1274. However, *C3* reads the *sum* at *ts*+1273 before it is updated by *C0*. Thus, *C0* and *C3* read the same value of *sum* and calculate its new values that are written into *sum* at *ts*+1274 by *C0* and at *ts*+1275 by *C1*. This is a data race condition. While replaying the program in the software debugger, the situation where *C1* reads the value of *sum* before it is updated by C0 is detected when read data traces of the program are supplied. One may argue that, by using data address access patterns we could also detect the race con-

dition. That can be true for this example however it is invaluable when the data is fed from an external source such as a sensor.

2.4.2 Memory Read Data Value Tracing Challenges

As discussed in Section 2.3, some commercial products support hardware tracing up to Class 3 [45] [46] [47] [49] [50] [51]. The trace port bandwidth required for these products is in the range of ~0.3 bits per instruction [17] for control-flow tracing and 8-16 bits per instruction for data-flow tracing [18] for a single core. In multicores, trace port bandwidth increases even further due to the need for reporting the processor identification number and timestamp in which the instruction is retired. With the limited size of on-chip-trace buffers, it is possible to capture traces only for short program segments. Usually the commercial products support collecting control-flow traces continuously and enables data-flow traces only for a range of addresses or for a memory address of interest or when a particular value from memory is read. In modern complex systems, the origin of a bug and its manifestation may be millions of instructions apart. Thus, it is important to capture traces for longer program segments or ideally for the entire program execution.

As a metric for quantitative evaluation of demands placed on designers of debug infrastructure, the required average trace port bandwidth (TPB) measured in bits per instruction (bpi) and bits per clock cycle (bpc) is used. The analytical model given in Eq.(2.1) and Eq.(2.2) can be used to estimate the average required TPB for read data value traces.

$$Average\ NX\_TPB\ [bpi] = mrFreq * (dtsSize + \lceil log_2 N \rceil + mrvSize) \quad (2.1)$$

$$Average\ NX\_TPB\ [bpc] = Average\ NX\_TPB[bpi] * IPC \quad (2.2)$$

where

- *mrFreq* is the frequency of instructions that read data from memory
- *dtsSize* is the average size of the differential timestamp field in bits
- *mrvSize* is the average size of the data read from memory in bits
- *N* is the number of processor cores
- *IPC* is the number of instructions executed per clock cycle when all the cores are considered together

Figure 2.3 shows the estimated TPB in bpi while varying the frequency of instructions that read data from memory (10% to 50%), the number of cores (N = 1, 2, 4, and 8), and the average size of data items read from memory (1-byte to 8-bytes). Please note that, for simplicity, the average required TPB is often referred to just TPB. The TPB is calculated by using the Eq. (2.1); the number of bits to encode the timestamp, *dts*, is set to 8 bits. Figure 2.3 shows how the TPB increases with respect to the frequency of memory reads and the average size of the data read. As the average data size read from memory varies from 1 byte to 8 bytes, the TPB ranges from 1.6 bpi to 7.2 bpi when N=1 and from 1.9 bpi to 7.5 bpi when N=8, when 10% of the instructions are memory reads. For any given configuration, as the number of cores increases, the required TPB increases slightly due to the additional bits used to report core id. However, when the frequency of memory reads increases, the TPB in-

creases gradually as shown by the different rows in the Figure 2.3. When the frequency of memory reads reaches 50% (which is possible in memory intensive applications), the TPB is as high as 37.5 bpi for 8-byte data. As the number of cores increases, the pressure on the trace port increases due to the reduced execution time and multiple trace producers. However, this trend is not captured well using TPB measured in bits per instruction executed.

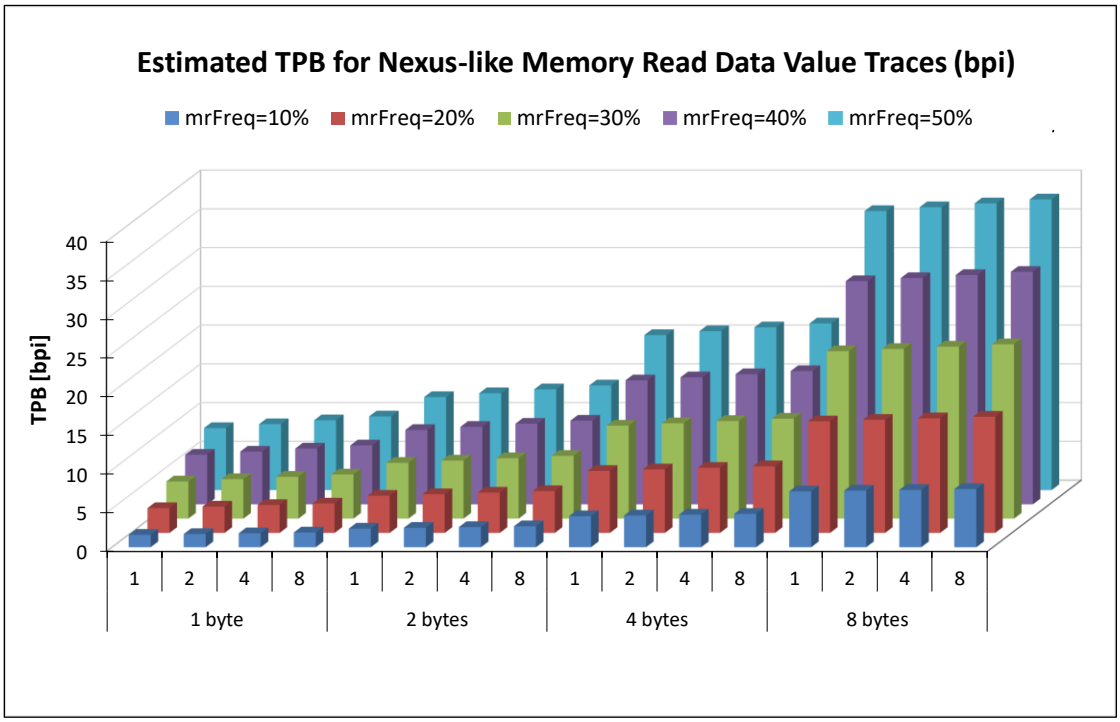


Figure 2.3 Estimation of Required Average TPB in bpi for Nexus-like Memory Read Data Value Traces

Figure 2.4 shows the estimated TPB in bpc while varying the frequency of instructions that read data from memory (10% to 50%), the number of cores (N = 1, 2, 4, and 8), and the average data size read from memory (1-byte to 8-bytes). The TPB shown in Figure 2.4 is calculated by using Eq. (2.2) with IPC set to N/4. It ranges

from 0.4 bpc (N=1) to 3.8 bpc (N=8) when *mrvSize* is 1-byte and from 3.6 bpc (N=1) to 30 bpc (N=8) when *mrvSize* is 8-byte when 10% of the instructions are memory reads. When the frequency of memory reads is 50%, the required TPB is as high as 75 bpc. For TPB estimation, we assumed that memory reads are evenly distributed over the execution of the program. However, there is a possibility of having skewed data reads. In that case, the pressure on the trace port will be even higher.

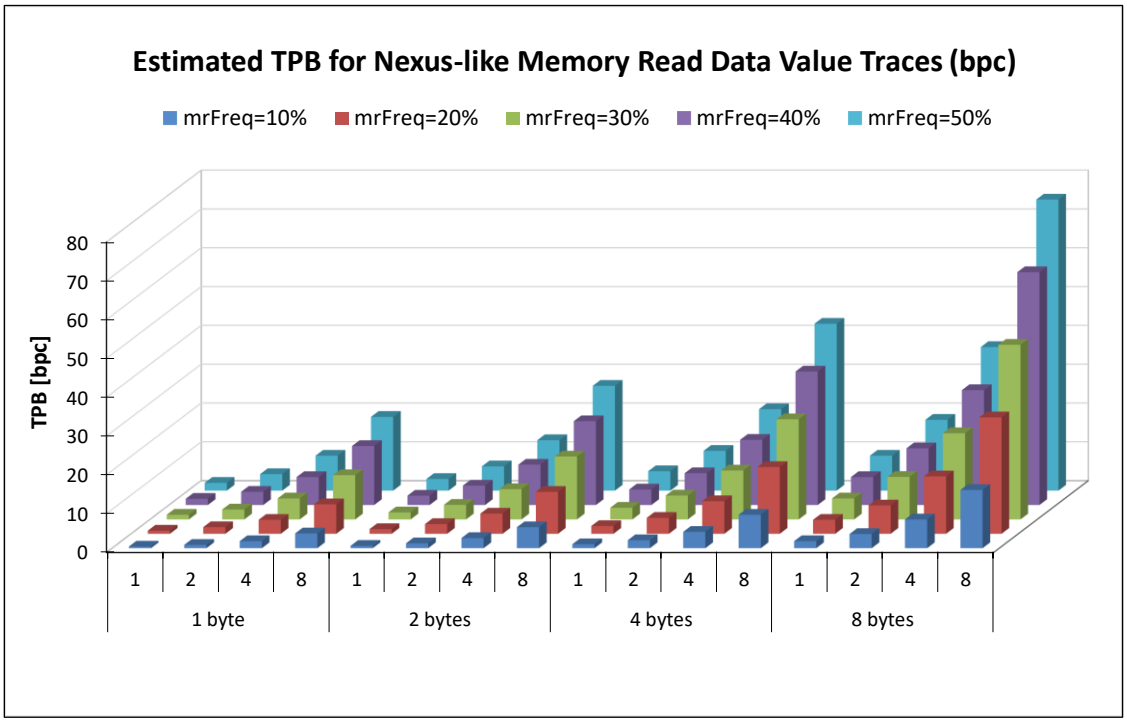


Figure 2.4 Estimation of Required Average TPB in bpc for Nexus like Memory Read Data Value Traces

To capture Nexus trace data (NX) without halting the program execution or without losing trace data requires either deep on-chip trace buffers or wide trace ports or both. The speed of emptying trace data through the trace port pins cannot keep pace with fast generating trace data, because the clock rate at I/O pins is typi-

cally lower than the clock rate of the processor [49]. Thus, a better solution is required to filter and compresses the trace data with minimal hardware resources to provide better debugging tools. Therefore, our work focuses on filtering data value traces as they are important to replay the program.

CHAPTER 3

RELATED WORK

Debugging is a critical part of the software development process in which developers are required to find and fix the bugs quickly. A study from Cambridge University [8] shows that software developers spend about 50% of their time debugging the programs. The challenges in debugging are due to required deeper understanding of the code as well as bugs. In addition, fixing mis-interpreted bugs may introduce new bugs and some bugs are not easy to reproduce in production systems.

Deterministic replay of a program allows software developers to reproduce bugs. For this, details of the program are collected while it is running on the actual target. These details are used for partial or full replay of the program execution offline. Details of program execution can be collected either using software (most of the times with instrumentation) or hardware support. To record traces using software, instrumentation of the program is required. However, instrumentation introduces overhead which cannot be tolerated in real-time embedded systems as the sequence of events in the instrumented runs may differ from native program execution. Section 3.1 discusses record and replay frameworks which allow deterministic replay of the program.

As modern processors generate a vast amount of trace data, capturing and streaming it out of the chip in a cost-effective way is a challenging task. Section 3.2 reviews prior work addressing trace compression solutions in both software as well as hardware.

3.1 Record and Replay

Pin-play [58] and Microsoft's [59] framework allow deterministic replay of user level programs by logging some details while it is running. These frameworks depend on the dynamic binary instrumentation of the program. Thus, the overhead of these methods cannot be tolerated in real-time embedded systems.

To deterministically replay program execution in software debuggers to identify causes of buggy states and debug multithreaded programs in high-end computing platforms, hardware-assisted techniques are available. One popular approach is to log how memory accesses interleave during program execution. Several groups have introduced approaches for deterministic replay in high-performance multiprocessors, including FDR [60], BugNet [61], RTR [62], DeLorean [63], and LReplay [64]. These schemes are designed to run on production systems where software instrumentation can be tolerated. They rely on a certain level of hardware support to capture production-run bugs with minimal overhead in performance and logging requirements. The logs are typically kept in a dedicated portion of main memory and the operating system is responsible for flushing logs from on-chip buffers to main memory, usually during periods when the memory bus is idle, thus minimizing the performance overhead. However, these methods rely on software instrumentation and system memory and thus interfere with program execution, which cannot be tolerated in real-time systems [17]. In addition, the performance degradation and increased energy consumption due to logging may not be tolerable in real-time embedded systems. Thus, hardware support for logging the data which can enable deterministic replay is required.

One interesting solution for debugging multicore SoCs called hidICE is proposed by Hochberger and Weiss [65]. 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 of the target.

3.2 Trace Compression

3.2.1 Software Trace Compression

Software program and data tracing has been widely used for trace driven simulations, architectural evaluation, optimization, and profiling programs. It is popular since instrumentation techniques have been employed to capture software traces. As software traces require a vast amount of disk space to store them, compression of these traces offline is widely explored in academia. One approach to compress the traces is using general-purpose compression algorithms, such as the Lempel-Ziv (LZ) algorithm [66] used in *gzip* or the Burrows-Wheeler transform [67] used in *bzip2*. However, compression ratios achieved with general purpose compressors are not optimal as it is hard to find repeating data patterns. Higher compression ratios can be achieved with trace-specific compressors.

One class of compressors exploits the spatial locality in progams. The basic idea is to take advantage of two successive memory accessing operations that might be consecutive in memory. Mache [68] is a one pass trace compression algorithm.

This algorithm separates instruction, memory read, and memory write traces and reports the differences in nearby addresses instead of full addresses. The transformed traces are then compressed using the LZ compression algorithm. Pleszkun [69] presented a two pass algorithm for compressing traces with instruction and data addresses by replacing repeating patterns of data with a code which requires a lower number of bits. The first pass is a pre-processing step to identify the dynamic basic blocks and procedure calls. The trace is encoded by specifying the basic block and its successor. PDATS (Packed Differential Address and Time Stamp) [70] and PDATS II [71] improves the compression ratio by reporting the variable length address offset calculated from the same type of trace (instruction, data read, and write) and optional repetition count.

Dictionary based compression is used in general purpose compressors [66] [72]. Usually, a dictionary is filled either statically or dynamically. The basic idea of a dictionary is to replace incoming data found in the dictionary with a corresponding index in the dictionary. PDI [73] instruction words are compressed by using the 256 most frequently used instruction words and are stored at the beginning of a trace file. PDI addresses are compressed using the PDATS method. N-Tuple [74] compresses branch instruction traces by replacing the traces with the index of an N-Tuple record table (N-TRT). Traces are divided into N-tuples and the N-TRT table includes unique N-tuples encountered in the program. Stream-Based Compression (SBC) [75] is a one-pass instruction and data address trace compression algorithm. A stream table keeps the details such as starting address, stream length, and instruction words and type of instruction stream (basic block). SBC performs compression by replacing each instruction from a stream by its index in the stream table. Data

addresses are written separately to a different file with offset and the number of repetitions corresponding to the stream table.

Value predictors are cache like structures used to predict the next value depending on the history of streamed values [76] [77] [78]. This method is employed to predict the register value, next data address for pre-fetching [78] [79], and memory load operation [76] to speed up microprocessors. Data address and data value predictors can also be used to compress software traces. The VPC [80] compression algorithm is a set of value prediction-based algorithms – last value predictor, stride predictor, etc., that works on memory addresses or data values. If one of the predictors is correct, the index of that predictor is reported; otherwise, a special code followed by the original value is reported. TCgen [81] generates VPC like high performance trace compressors for a user defined trace format. It translates the user specified trace format to an optimized compressor by using selection of value predictors – last-value predictors, finite-context-method predictors, and differential-finite context-method predictors.

Caches are widely used in software and hardware to speed up program execution by exploiting temporal and spatial locality of instructions and data. Locality Based Trace Compression (LBTC) [82] employs offset encoding of the memory references (as in Mache and PDATS) and static property of attributes and temporal locality of memory references. It assumes that most of the attributes are static and do not change frequently from one dynamic access to another. A small direct mapped cache is emulated to store the memory references. When a memory reference is a hit in the cache, all the static attributes can be extracted without storing them in the trace file. Microsoft's framework uses an iDNA component [59] to record and retrieve instruction level traces. The execution traces include executed code bytes and the state

of the memory location which is accessed to enable faithful replay of the user-mode program. iDNA maintains a tag-less direct cache which is indexed with accessed address. When the instruction accesses memory, the trace is not logged if the value in the cache matches value read from memory. If not, the trace is logged, and the value is updated in the cache.

Another class of compression algorithms depends on Whole Program Paths (WPP) [83] and loops in the program [84]. WPP captures the control traces of a program and uses modified *Sequitur* to compress acyclic paths. These traces can be used to guide compiler optimizations and identify hot spots in the program. However, WPP is a two-pass algorithm. Data addresses can be compressed by linking them to the program loops [84]. In the first pass, loops in the trace are detected by using control flow analysis. In the second pass, data addresses in the loop are classified as constant, loop varying, and chaotic. Constant and loop varying addresses are encoded only once in the compressed trace and chaotic addresses are stored separately in a file.

3.2.2 Hardware Trace Compression

Commercial trace modules usually support run-control debugging, often control-flow tracing, and less often data-flow tracing. For control-flow and data-flow traces, existing commercial trace modules apply on-chip trace filtering (capture the traces for interested instructions and/or memory locations or for specific code segments) and compression (differential encoding or ignoring the most significant zero's) to reduce trace port bandwidth. Yet, the required trace port bandwidths are still in the range from 0.5-4 bits per instruction executed per core for control-flow traces [17] and 8-16 bits for data-flow traces [18]. Thus, even with compression, the

required trace port bandwidth for control-flow and data-flow traces is still in the range of dozens of Gbits/sec, which is not practical. To support control-flow and data tracing, commercial trace modules rely on hefty on-chip buffers to store traces captured. The traces are then read out through the trace port in near real-time. However, large trace buffers and wide trace ports significantly increase the system complexity and cost, making embedded processor vendors reluctant to support higher classes of Nexus 5001 operation. Alternatively, these traces are stored in system memory and they are emptied through system software in the case of self-hosted debugging.

To compress hardware traces, general purpose compression algorithms can be used. However, general-purpose compressors do not yield high compression ratios, as traces combine multiple diverse fields that limit the redundancy visible to compressors. In addition, compression algorithms rely on large search buffers, so their hardware implementations are very expensive. Moreover, general-purpose compression algorithms suffer from long latencies, which are not desirable in program tracing. To partially alleviate these constraints, several research efforts have proposed trace-specific implementations of general-purpose compression algorithms targeting control-flow traces, such as LZ-based trace compressors [85][86].

The Double-move-to-front compressor [87] (DMTF) uses basic block properties and Move-To-Front transformation, which is used in *bzip2*. DMTF uses two history tables storing a stream, basic block length and size. When a stream is found in the first table, the second table is also searched. If it is a hit in the second table, the index of the second table is reported; otherwise, the index of the first table is reported and it is stored in the second table. If it is a miss in the first table, the trace corresponding to the stream is reported and it is inserted in the first table.

Stream cache and predictor [88] is a hardware trace compression mechanism for control flow traces. In this mechanism, stream descriptors describe instruction streams (address and length) that are stored in a stream descriptor cache (SDC). If it is a hit, the stream index (result of concatenation of set and way index of SDC) is reported to the last stream predictor (LSP). If it is a hit in the LSP, a single bit reported, otherwise stream index is reported. If it is a miss in the SDC, the original stream is reported.

In computer architecture, branch predictors are widely used to predict the output of control-flow instruction depending on the history to avoid flushing of the pipeline. However, branch predictors are also explored in compressing control-flow traces [89] [90] [91]. In these techniques, a trace message is reported only when the branch is mis-predicted by the branch predictor in the trace module instead of emitting every trace message. These techniques significantly reduce the number of trace events, and thus reduce the required trace port bandwidth. However, these studies focus on control-flow traces that are easier to compress rather than data traces.

Another class of trace compressor produces the more compact traces [92]. However, the compression ratio achieved with this method is only 8.4% for program traces and 24% for data traces. Unfortunately, data value traces exhibit little redundancy and are typically streamed out uncompressed [92].

Whereas a number of recent papers focuses on capturing, compressing, and filtering control-flow traces, to the best of our knowledge, very few studies focus on compressing the real-time hardware memory read data value traces that are crucial in debugging parallel programs and hard to find errors. This work builds on the previous efforts from the LaCASA laboratory [93] [19] [20], and expands them to multicores and makes the data tracing scalable for parallel programs.

CHAPTER 4

PROPOSED TECHNIQUE: mcFiltrate

This chapter discusses the details of the proposed technique, *mcFiltrate*, and its operation on the target core and in the software debugger. The system view of *mcFiltrate* and the terminology used is discussed in Section 4.1. The operation of *mcFiltrate* for memory read, memory write, and invalidation operations on the target core and in the software debugger are discussed in Section 4.2 and Section 4.3, respectively. The format of the trace message emitted by the hardware is discussed in Section 4.4. Section 4.5 illustrates the operation of *mcFiltrate* in both the target core as well as in the software debugger with an example. Section 4.6 discusses the analytical model which allows estimation of the required average trace port bandwidth.

4.1 System View of *mcFiltrate*

mcFiltrate – ***m****ulti****c****ore* ***filt****ered memory* ***r****ead d****a****ta* ***t****rac****e*** – is a hardware/ software technique for filtering memory read data value traces. The basic idea of *mcFiltrate* is to avoid sending unnecessary trace messages for data items that have already been reported and/or can be inferred by the software debugger. Thus, this technique relies on L1 data caches and first-access (FA) tracking bits to filter the memory read traces.

Figure 4.1 shows a block diagram of a system that uses *mcFiltrate* hardware and software modules. Items colored in purple color in Figure 4.1 are additional hardware required for the operation of *mcFiltrate*. As shown in a detailed view of

mcFiltrate, the L1 data cache on each processor core is expanded to include FA tracking bits. The size of the sub-block being protected by a single FA bit is called the granularity size (G). The main purpose of the FA bits is to indicate whether the sub-block is accessed for the first time or not. When the sub-block(s) is accessed for the first-time, it is traced out and the corresponding FA bit is set to prevent redundant trace messages for the same sub-block in case of future read accesses. This way we can exploit the temporal and spatial locality of data accesses to reduce the number of trace messages that need to be reported to the software debugger. To synchronize an FA hit event on the target platform and on the software debugger while replaying the program, a first-access hit counter (*fct*) is used. The value of this counter is incremented by one for every FA hit event and it is reported along with the next trace message i.e. emitted when an FA miss event occurs.

mcFiltrate requires the software debugger to maintain a software model of the L1 data cache which is identical to the actual L1 data cache on the target platform, i.e., it uses the same cache organization and updating policy. In addition, the software debugger also maintains a software copy of the first-access hit counter.

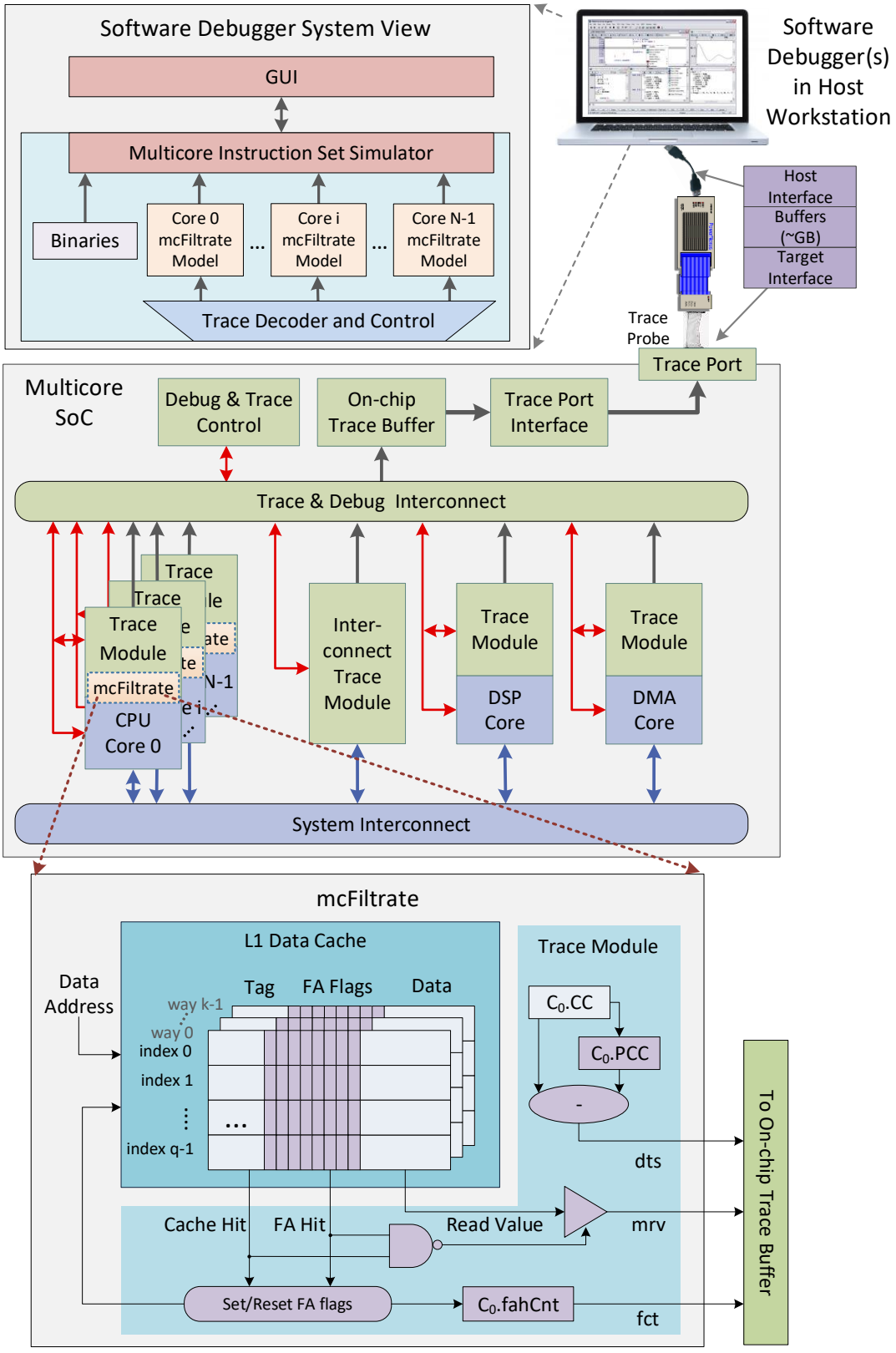


Figure 4.1 System View of *mcFiltrate*

4.2 *mcFiltrate* Operation on Target Platform

Memory reads. The operation of *mcFiltrate* for memory reads on the target core *i* is shown in Figure 4.2. For every memory read operation, a data cache lookup is performed (step 1). If the requested data is found in the data cache, it is referred to as a *cache hit event;* otherwise it is a *cache miss event* (step 2). If all the FA bit(s) corresponding to the requested data item are set, it is referred to as an *FA hit event*; otherwise it is an *FA miss event* (step 3). The blue boxes in Figure 4.2 indicate the additional steps required to support *mcFiltrate* with respect to a regular memory read operation in a processor core.

In the case of an *FA hit event*, the software debugger can find the data item in the software version of the data cache, thus, no trace message is emitted. Instead, *fct* is incremented by 1 (step 4) to indicate the *FA hit event*. If the corresponding FA bit(s) is not set, a trace message composed of timestamp (*dts*), core id (*cid*), first-access hit counter (*fct*), and data value (*mrv*) corresponding to the cache sub-block(s) is emitted. Additionally, *fahCnt* is reset to zero and the corresponding FA bit(s) is set (step 6). Please note that FA bits for sub-blocks are verified independently and the trace message includes sub-blocks for which FA bit are not set.

In the case of a *cache miss event*, a *Coherent Read Transaction* is initiated (step 5). Without loss of generality we assume that the MOESI [94] cache coherence protocol is used. The five letters of MOESI represent the possible states of a cache block at any given time. The possible states are *M* – Modified, *O* – Owned, *E* – Exclusive, *S* – Shared, and *I* – Invalid. The MOESI cache coherence protocol allows the cache-to-cache transfers of the dirty data without updating the main memory. A brief description of MOESI states is given below:

M – This is the only valid copy of the cache block and main memory is not up to date. From this state, the valid state transitions are *O* (with coherent read) and *I* (with coherent read and invalidate or coherent invalidate).

O – This state allows the cache-to-cache transfer of a dirty cache block without updating the main memory. Multiple caches may have the valid cache block in state *S*. However, only one cache can be in state *O*. It can transition to state *M* (with memory write) or *I* (with coherent read and invalidate or coherent invalidate) or *S* (by writing cache block to memory).

E – A single cache is the exclusive owner of the cache block and its content is the same as in main memory. It can transition to state *M* (with memory write) or *S* (with coherent read) or *I* (with coherent read and invalidate or coherent invalidate).

S – Several caches may have a copy of the cache block in state *S* or *O* (only one can be in the owned state). The main memory does not necessarily have to be up to date. It can transition to state *M* (with memory write) or *E* (by issuing coherent invalidate) or *I* ((with coherent read and invalidate or coherent invalidate).

I – This cache block is not valid. It can transition to *M* (with memory write) or *E* (with memory read), and *S* (with coherent read and when multiple copies exist in other caches).

In *Coherent Read Transaction*, all other processors perform a snoop lookup to determine whether the requested data is available in their cache or not (step 7). In Figure 4.2, *Cx* represents any processor core other than the requesting processor core *Ci*. If the data is available in any other processor data cache (a remote cache) and if that processor is responsible for cache-to-cache (C2C) data transfer, the cache block is transferred to the requesting processor, *Ci* (steps 8-10). The state of the

cache block in both the requesting and the responding processor is updated according to the cache coherence protocol and FA bits corresponding to the cache block are cleared in processor *Ci* (step 11). The state of the cache block is updated to *S* in *Ci* (*Ci.CBj.state*) and to either *S* or *O* in *Cx* (*Cx.CBj.state*), depending on the initial state - new state is *O* if the initial state is *M* or *O*, otherwise it is *S*.

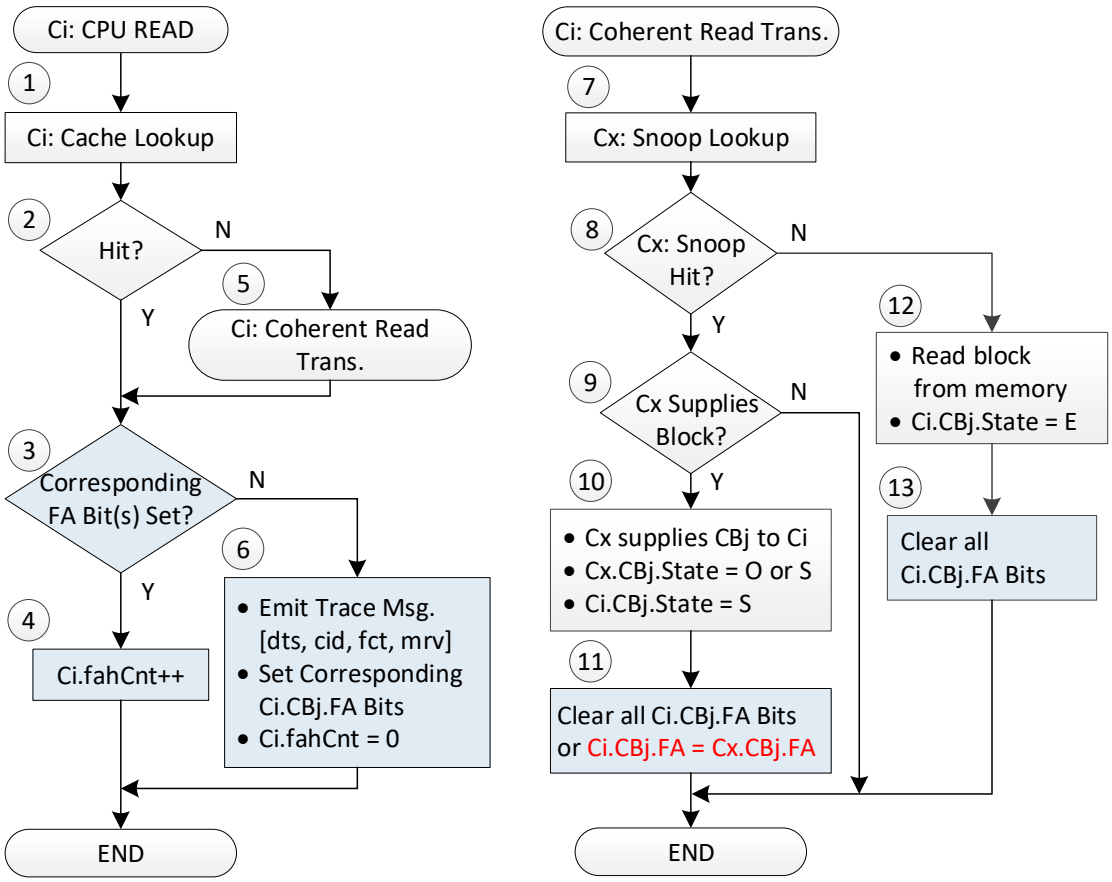


Figure 4.2 *mcFiltrate* Operation on Target Platform Core *i* for a Memory Read

In multithreaded programs data is usually shared among multiple threads while the program is executing in parallel. As we have seen in *Coherent Read Transaction*, when the data is found in a remote cache (*snoop hit*), the cache block is transferred to the requesting processor. In this case there is a possibility that the

cache block might have already been reported to the software debugger by the responding processor. Since the FA bits are cleared for newly fetched cache block in *Ci*, the data will be reported again. Thus, to reduce redundant trace messages generated for shared data by multiple threads/cores, we consider inheriting the FA bits from the responding processor, *Cx* (step 11). However, to keep track of which core has actually modified the data and when it is modified, inheriting FA bits cannot be allowed when *Cx* has the cache block in the *Modified* (M) state. This exception allows the software debugger to properly serialize reads and writes while replaying a program without assuming a cycle-accurate target simulator. Since the FA bits are cleared, the trace message *Ci* will emit a trace message with a timestamp and data content that helps the software debugger to infer the order of reads and writes on shared variables as it occurred on the hardware platform.

Please note that, to inherit the FA bits, *mcFiltrate* requires additional hardware support. Two possible solutions for supporting inheriting FA bits are: (a) issuing an extra bus transaction and (b) adding extra data lines to transfer FA bits between caches. The former approach adds additional bus traffic, thus increasing the latency, whereas the latter approach requires additional hardware support that is proportional to the number of FA bits per cache block. The rest of this dissertation assumes the latter approach.

In the case of a *snoop miss event*, the requested cache block is retrieved from the main memory. Since the software debugger does not have access to the contents of the main memory, the FA bits are cleared, and the state of the cache block is updated to *E* (step 13).

Memory writes. Figure 4.3 shows the sequence of steps carried out in *mcFiltrate* for memory writes on the target processor core *Ci*. In the case of a *cache hit event*, a *Coherent Invalidate Transaction* (step 7) is initiated by the processor core *Ci* to acquire ownership if the cache block is in the state *S* or *O*. In *Coherent Invalidate Transaction* all the other processors perform the snoop lookup. In the case of a *snoop hit event*, the cache block is invalidated (steps 10-12) and the corresponding FA bits are cleared (step 13).

In the case of a *cache miss event*, a *Coherent Read and Invalidate Transaction* is issued by the processor core *Ci* (step 3). If the requested data block is found in any other processor's data cache and if that processor is responsible for transferring the cache block, then the cache block is transferred to the processor core *Ci* and the corresponding FA bits are cleared for processor *Ci* (steps 19-21). If inheriting the FA bits is supported, then the FA bits are also copied along with the cache block (step 21). The state of the cache block in any other processor's cache is updated to *I* (step 22) and the corresponding FA bits are cleared (step 23). Once the processor *Ci* acquires the ownership (the state of the cache block is *M*) (step 8), its data cache is updated with the current write operation and the FA bits are set for all sub-blocks which are written completely (step 5). If the data is retrieved from the main memory, the state of the cache block is updated to *M* (step 17), and all the FA bits for the corresponding cache block are cleared (step 18).

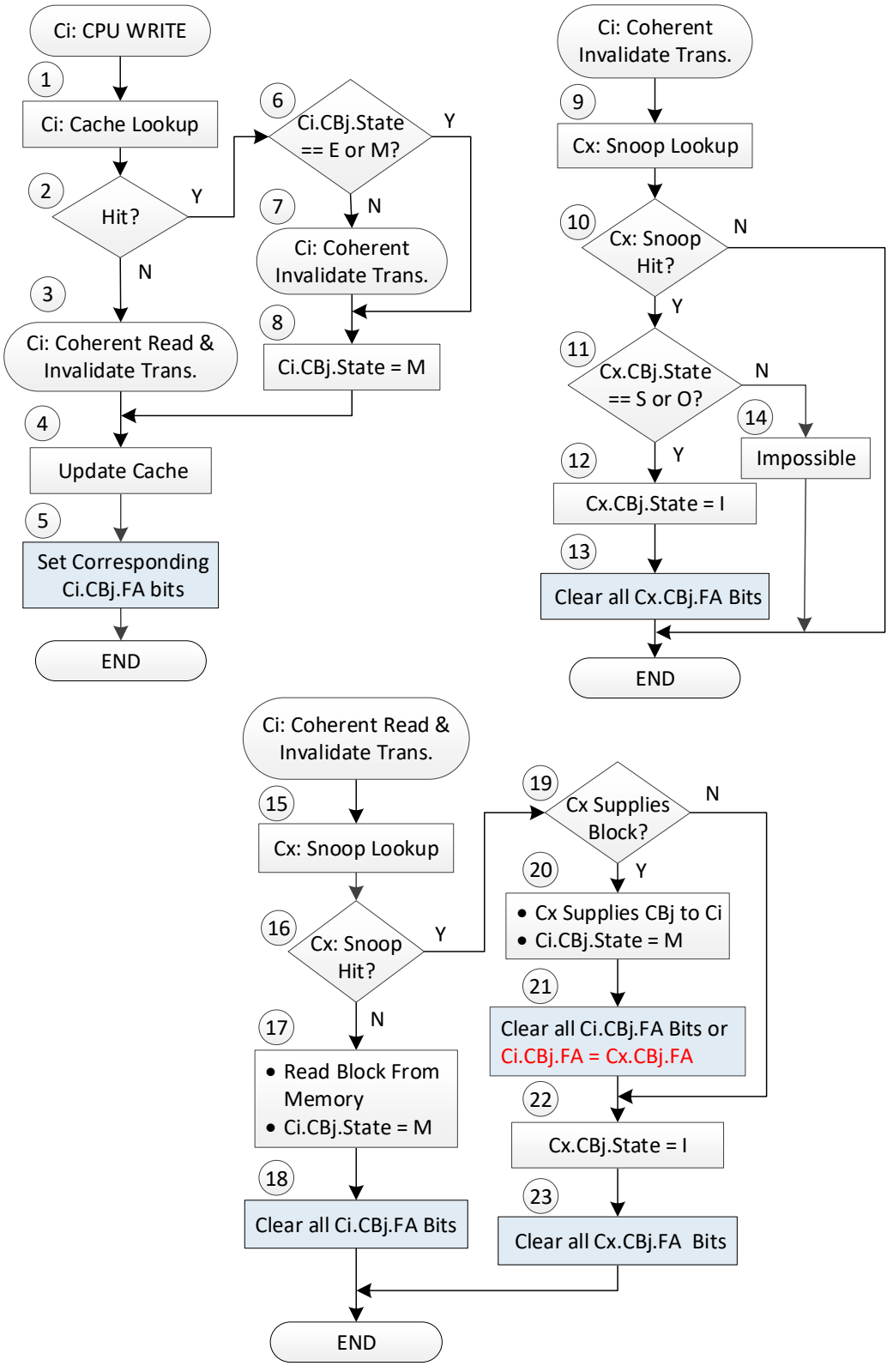


Figure 4.3 *mcFiltrate* Operation on Target Platform Core *i* for a Memory Write

4.3 *mcFiltrate* Operation on the Software Debugger

The software debugger replays the program using an instruction set simulator (ISS) and the program binary as well as reading and decoding the trace messages received from the target. The format of the trace messages and length of the encoding parameters is already known to the software debugger. The software debugger maintains software copies of data caches with the same cache size and organization and applies updating policies that match those on the target platform. It also maintains a software copy of the first-access hit counter.

The operation of *mcFiltrate* on the software debugger side for memory reads, memory writes, and external invalidations is shown in Figure 4.4. Timestamps and core identifiers from trace messages allow all software threads to synchronize and establish the ordering of events that matches the one on the target platform for shared variables. The first-access hit counter specifies the number of memory read operations executed in the debugger before reading the value of *mrv* from the current trace message.

Memory reads. For every memory read (Figure 4.4(a)), *Ci.fahCnt* is decremented by 1 if its value is greater than 0. *Ci.fahCnt>0* indicates the *FA hit event* on the target core, hence the software debugger reads the data from the processor core *Ci* software cache or other processor's software cache, if inheriting the FA bits is supported on the target platform (step 6). If the data is read from a remote cache, the FA bits are also copied (step 12). If *Ci.fahCnt*=0, the software debugger extracts the data from the previously decoded trace message and updates the software cache (step 2). In addition, it reads a new trace message originated on core *Ci* from the trace buffer and decodes it (step 3). *Ci.fahCnt* is updated with the new value ex-

tracted from the decoded trace message. Please note that the value of *Ci.fahCnt* is set to zero before the start of program execution.

Memory writes and invalidations. For every memory write (Figure 4.4 (b)), the processor core *Ci* acquires the ownership (step 23) mirroring steps taken on the target platform if the data is found in the software cache (a *cache hit event*). For a *cache miss event*, the processor core *Ci* gets the cache block along with FA bits from a remote cache (step 27) if the FA bit inheritance is supported. If the inheriting of the FA bits is not supported, the FA bits corresponding to the cache block are cleared (step 29). If the data is not available in any of the remote processor software caches, meaning that the cache block was supplied by the main memory on the target platform, the FA bits are cleared (step 28). Regardless of cache hit or miss event, the software cache is updated with the current write operation and the FA bits for the sub-blocks which are written completely are set (step 24). For external invalidations (Figure 4.4(c)), the FA bits corresponding to the invalidated cache block are cleared.

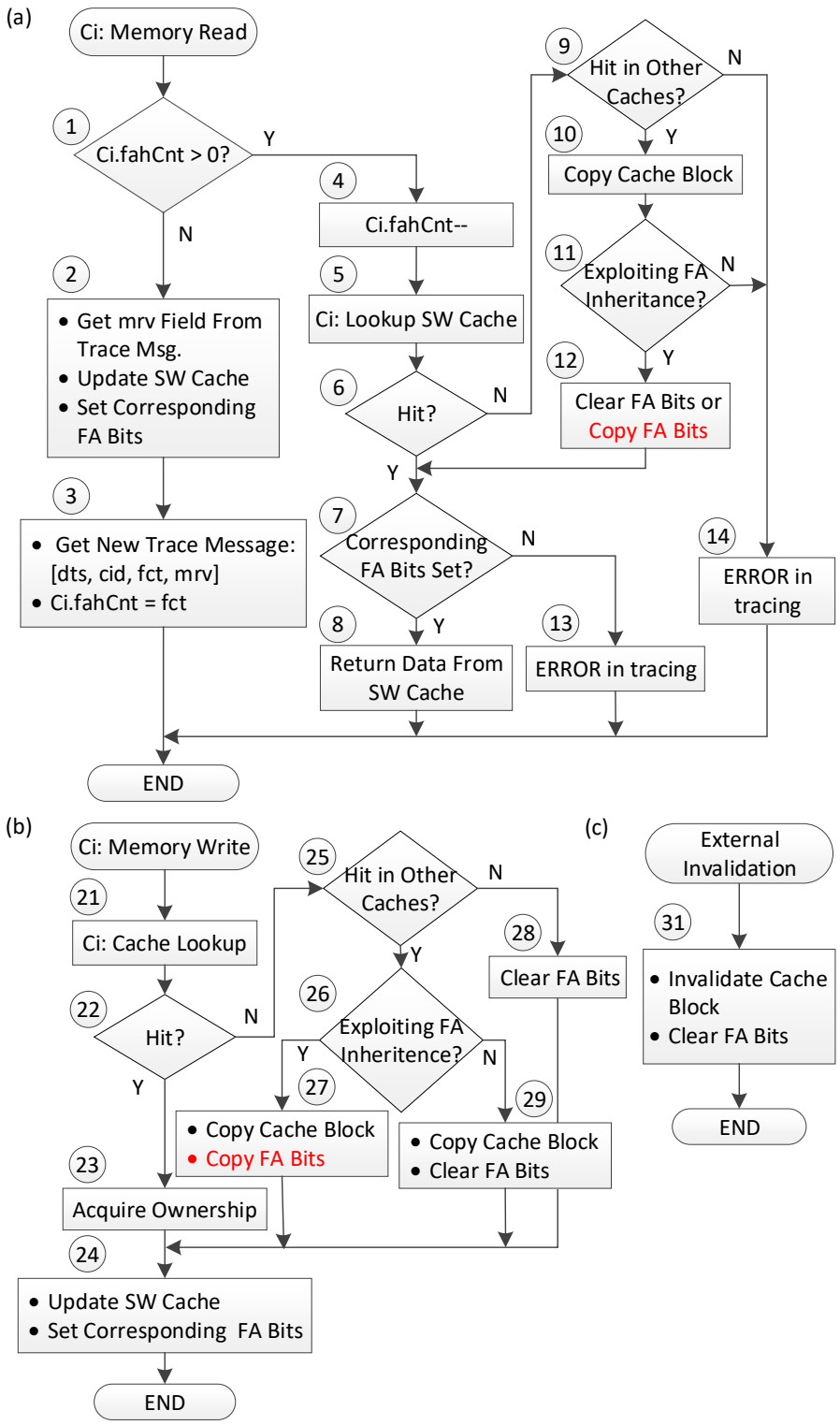


Figure 4.4 *mcFiltrate* Operation on Software Debugger (a) a Memory Read (b) a Memory Write (c) an External Invalidation

4.4 Encoding of Trace Messages

This section describes the encoding of Nexus-like and *mcFiltrate* trace messages. Each Nexus-like (NX) trace message is composed of *dts*, c*id*, and *mrv* fields as discussed in 2.3 (Figure 4.5). The timestamp, *dts*, carries information about the clock cycle in which the current trace-generating instruction has retired. To reduce the number of bits required to represent the timestamp, we report a differential timestamp, i.e, rather than reporting absolute clock cycle, we report the number of clock cycles expired since the last trace message was emitted by the core *i*, *Ci.dts=Ci.CC–Ci.PCC*; *Ci.PCC=Ci.CC* (CC – Current Clock Cycle, PCC – Previous Clock Cycle). Note: the first trace message contains the time from the beginning of the program. For simplicity, we assume all cores share a global clock. The number of bits needed to represent the *dts* field changes from one program to other and within the program during execution. Thus, we use 8-bit chunks to encode *dts*. The connect bit (C) determines whether more 8-bit chunks are needed to fully encode the *dts* value (C=1) or not (C=0). For example, a *dts* value of 260 is encoded as 00100000**1**10000000 (bits are aligned from the least to the most significant). The length of the *cid* field is fixed and is a function of the number of cores. For N cores, the length of *cid* is $\lceil \log_2 N \rceil$ bits. The length of the *mrv* field depends on the size of the operand read from memory (for IA32 ISA it ranges from 1 to 120 bytes) and is thus $8 \cdot$ *sizeof(operand type)* bits.

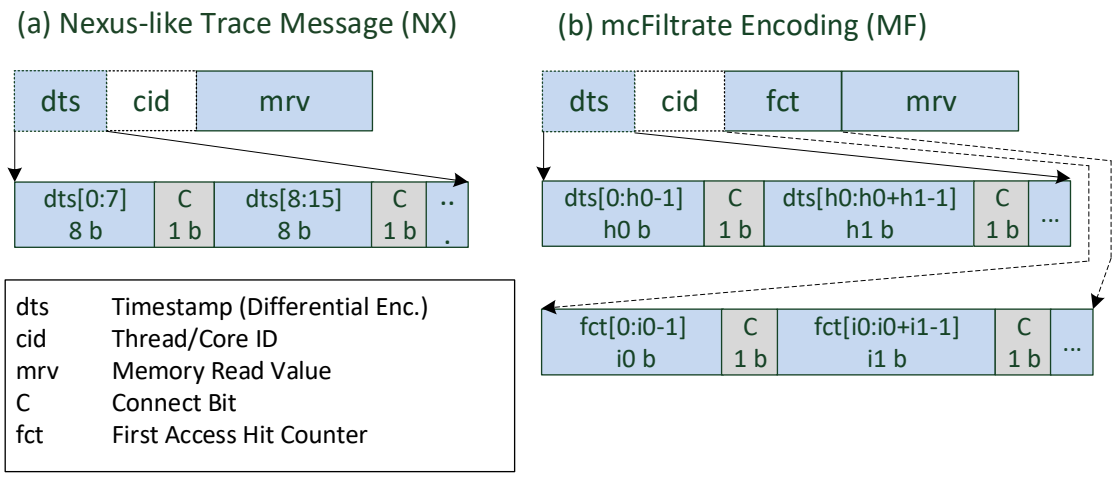


Figure 4.5 Encoding of Trace Messages

mcFiltrate trace messages (MF) include the first-access hit counter field, *fct*, in addition to the fields which are in NX (Figure 4.5). The number of bits required to encode *fct* varies with the distribution of FA miss rate while the program is executing. The length of the *mrv* field depends on the granularity size and can be calculated as follows: $8 \cdot G \cdot \lceil sizeof(type)/G \rceil$ bits. For example, if the operand size is 2-bytes and G=4, the length of the *mrv* field is 4 bytes or 32 bits. Note: if the data address is not perfectly aligned with sub-block then there is the possibility that the size of the *mrv* field is more than $8 \cdot G \cdot \lceil sizeof(type)/G \rceil$ bits. For example, consider the same example as above except that now the operand spans two sub-blocks, then the length of the *mrv* field is 8 bytes or 64 bits. Unlike in NX, to further squeeze the trace port bandwidth we employ a variable encoding mechanism. We allow chunks for encoding *fct* (i0, i1, i2, …) and chunks for encoding *dts* (*h0, h1, h2*, …) to be variable in size as shown in Figure 4.5 [21]. Chapter 5.4 gives the results of an evaluation that explores different encoding arrangements in order to select good values that minimize the number of bits needed to encode these fields.

4.5 An Illustrative Example

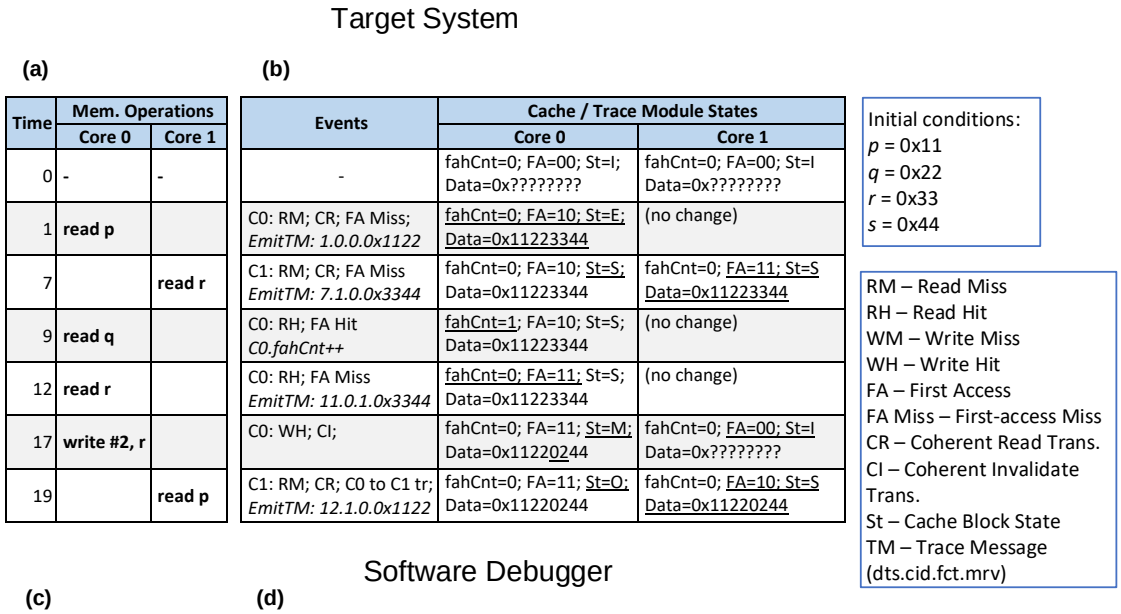
To illustrate trace operations in the hardware module and the software debugger with *mcFiltrate*, an example shown in Figure 4.6 is considered. We will assume two cores (*C0* and *C1*) that have a single-block direct-mapped cache with a size of 4 bytes. Four one-byte variables, *p*, *q*, *r*, and *s* are mapped into a single cache block. The granularity size is 2 bytes; thus, a cache block has two first-access bits, one guarding *p* and *q* and the other guarding *r* and *s*.

Let us consider a sequence of memory operations executed on the target platform in the order as described in Figure 4.6(a). Figure 4.6(b) shows the state of the caches and tracing structures on the hardware platform. Initially, the caches are empty, the first-access bits are cleared, as well as *fahCnt* counters. At time 1, *C0* reads *p* – this operation results in a cache read miss (RM) and a coherent read transaction (CR) is issued. The cache block is retrieved from main memory and its state is *Exclusive (E)*. This is a first-access miss (FA miss), thus a trace message is emitted containing the differential *timestamp* (1), *core id* (0), *fahCnt* (0), and data value (0x1122). At time T=7, *C1* reads *r*. This operation also results in a cache read miss. A coherent read transaction is issued and *C0* will supply the requested cache block together with its FA bits; the fetched cache block is in the *Shared (S)* state. As the FA bit guarding the variable *r* is not set, a new trace message is emitted as shown in the Figure 4.6 (b). At time *T=9*, *C0* reads *q*. This operation results in a cache read hit and first-access hit; *fahCnt* is incremented and no other actions are taken. Please note that though *q* is read for the first time in this sequence, it has been reported previously as it shares the FA bit with the variable p. At time *T=12*, *C0* reads *r*. This operation results in a cache read hit and an FA miss, causing a new

trace message to be emitted. At time *T=17*, *C0* performs a write operation on *r*. A coherent invalidate transaction (CI) is issued causing invalidation in the C1 data cache. At time *T=19*, *C1* reads *p*. This operation results in a read miss and a coherent read transaction is issued. *C0* will supply the requested block. Though *C0* contains the requested cache block with both FA bits set, we cannot allow *C1* to inherit FA bits from *C0* and thus a new trace message is emitted. The reason for this seemingly unnecessary trace operation is to ensure that software debugger can infer the order of memory operations on shared variables without requiring a cycle-accurate simulator.

Now, let us describe the steps that will take place on the software debugger side (Figure 4.6(c)-(f)). The software debugger replays instructions in the ISS as shown in Figure 4.6(e) and parses the trace messages retrieved from the hardware platform shown in Figure 4.6 (f). We assume identical starting conditions: the current time *T=0*, software copies of the data caches are empty with cleared FA bits, and the software copies of *fahCnt* counters are cleared. For simplicity we only look at memory operations presented by the ISS from the program binary. The *fahCnt* counters are cleared and thus the first two messages for *C0* and *C1* are decoded as shown in row 2 of Figure 4.6(d). The first events occur at time *T=1* on *C0* and at time *T=7* on *C1*. Thus, the debugger will replay the "*read p*" operation on *C0* first. The cache is empty, so the data is retrieved from *C0.mrv*, the FA bit is set, and the software cache updated accordingly. Because *fahCnt*=0, the following trace message (TM1) for *C0* will be processed. The *fahCnt* counter is updated from the content of that message (*C0.fahCnt=1*). The timestamp of the next memory operation that resulted in a miss at *C0* is set to *C0.T*=*C0.T*+*C0.TM1.dts*=1+11=12. As the *C0.fahCnt*=1, that

means "*read q*" can be replayed using information from the software data cache only. The next operation to be replayed is "*read r*" at *T=7* on *C1*. By replaying "*read r*" on *C1*, the software copy of the data cache is updated, and the following trace message (TM1) is parsed (*C1.fahCnt=0*, *C1.T*=*C1.T*+*C1.TM1.dts*=7+12=19). The next event scheduled at *C0* is "*read r*" at *T=12* (please note that the previous operation on *C0* is replayed before). The software copy of the data cache is updated accordingly. The next event scheduled to replay is "*read p*" on *C1* at *T=19*. However, this trace message is resulted on the shared data block because of the write after read operation. This means, "*write #2, r*" on *C0* is executed before "*read p*" on *C1*. Thus, the software debugger replays "*write #2, r*" on *C0* before replaying "*read p*" on C1.



Time	Mem. Operations	
	Core 0	Core 1
0	-	-
1	**read p**	
7		**read r**
9	**read q**	
12	**read r**	
17	**write #2, r**	
19		**read p**

Events	Cache / Trace Module States	
	Core 0	Core 1
-	fahCnt=0; FA=00; St=I; Data=0x????????	fahCnt=0; FA=00; St=I Data=0x????????
C0: RM; CR; FA Miss; *EmitTM: 1.0.0.0x1122*	fahCnt=0; FA=10; St=E; Data=0x11223344	(no change)
C1: RM; CR; FA Miss *EmitTM: 7.1.0.0x3344*	fahCnt=0; FA=10; St=S; Data=0x11223344	fahCnt=0; FA=11; St=S Data=0x11223344
C0: RH; FA Hit *C0.fahCnt++*	fahCnt=1; FA=10; St=S; Data=0x11223344	(no change)
C0: RH; FA Miss *EmitTM: 11.0.1.0x3344*	fahCnt=0; FA=11; St=S; Data=0x11223344	(no change)
C0: WH; CI;	fahCnt=0; FA=11; St=M; Data=0x11220244	fahCnt=0; FA=00; St=I Data=0x????????
C1: RM; CR; C0 to C1 tr; *EmitTM: 12.1.0.0x1122*	fahCnt=0; FA=11; St=O; Data=0x11220244	fahCnt=0; FA=10; St=S Data=0x11220244

Time	Mem. Operations	
	Core 0	Core 1
0	-	-
1	**read p**	
7		**read r**
?	**read q**	
12	**read r**	
	write #2,r	
19		**read p**

Trace Decoding	Core 0 Software Cache	Core 1 Software Cache
	fahCnt=0; FA=00; St=I; Data=0x????????	fahCnt=0; FA=00; St=I; Data=0x????????
Read TM0 from Core 0 and Core 1 C0.T=1, C0.fahCnt=0; C0.mrv=0x1122 C1.T=7, C1.fahCnt=0; C1.mrv=0x3344	(no change)	(no change)
Read TM1 from Core 0 C0.T=12, C0.fahCnt=1; C0.mrv=0x3344	fahCnt=1; FA=10; St=E; Data=0x1122????	(no change)
Read TM1 from Core 1 C1.T=19, C1.fahCnt=0; C1.mrv=0x1122	fahCnt=1; FA=10; St=S; Data=0x1122????	fahCnt=0; FA=11; St=S; Data=0x11223344
	fahCnt=0; FA=10; St=S; Data=0x1122????	(no change)
	fahCnt=0; FA=11; St=S; Data=0x11223344	(no change)
	fahCnt=0; FA=11; St=M; Data=0x11220244	fahCnt=0; FA=00; St=I; Data=0x????????
	fahCnt=0; FA=11; St=O; Data=0x11220244	fahCnt=0; FA=10; St=S; Data=0x11220244

Mem. Operations from ISS	
Core 0	Core 1
read p	read r
read q	read p
read r	
write #2, r	

Trace Messages		
ID	Core 0	Core 1
TM0	1.0.0.0x1122	7.1.0.0x3344
TM1	11.0.1.0x3344	12.1.0.0x1122

Figure 4.6 An Illustrative Example of Data Tracing with *mcFiltrate*

4.6 *mcFiltrate* Analytical Model

It is useful to derive an analytical model for assessing the required TPB for *mcFiltrate*. The required average TPB for *mcFiltrate* can be estimated as a function of the FA miss rate as shown in Equations (4.1) and (4.2).

$$Average\ MF\_TPB\ [bpi] = mrFreq * faMissRate * \Big(dtsSize + fctSize + \lceil log_2 N \rceil + \big(G * \lceil {}^{mrvSize}/_{G} \rceil * 8\big)\Big) \tag{4.1}$$

$$Average\ MF\_TPB[bpc] = Average\ MF\_TPB\ [bpi] * IPC \tag{4.2}$$

where

- *mrFreq* is the frequency of instructions that read data from memory
- *faMissRate* is the first-access miss rate; depends on the temporal and spatial locality of the data
- *dtsSize* is the average size of the differential timestamp field in bits
- *fctSize* is the average size of the first-access hit counter field in bits
- *N* is the number of processor cores
- *G* is the granularity size; size of the sub block protected by a FA bit
- *mrvSize* is the average size of the data read from memory in bytes
- *IPC* is the number of instructions executed per clock cycle when all the cores are considered together

Figure 4.7 shows the estimated average TPB in bpi while varying the frequency of instructions that read data from memory (10% to 50%) and the FA miss rate (5% to 55%) when N=8. While calculating the TPB using Eq. (4.1), the number

of bits to encode timestamp (*dtsSize*) and first-access hit counter (*fctSize*) is set to 8 bits each and the granularity is set to 4 for simplicity. We assume that the dominant data type is 64-bit long, memory accesses are aligned, and every trace message includes the data value from two sub-blocks. However, in reality, a trace message may include data values from one or more sub-blocks depending on the size of the memory read operation, memory alignment, and temporal and special locality of the data in the data cache.

The estimated TPB with *mcFiltrate* ranges from 0.42 bpi with 10% of memory reads to 2.08 bpi with 50% of memory reads when the FA miss rate is 5%. As the FA miss rate increases, the average TPB increases (see Eq. (4.1)). Thus, with a 55% FA miss rate, the TPB ranges from 4.57 bpi to 22.83 bpi. However, the equivalent TPB for NX from Section 2.4.2 with N=8 ranges from 7.5 bpi to 37.5 bpi when the frequency of memory reads ranges from 10% to 50% and the dominant data type is 64-bit long. The results clearly show that *mcFiltrate* has a potential to reduce the TPB requirements.

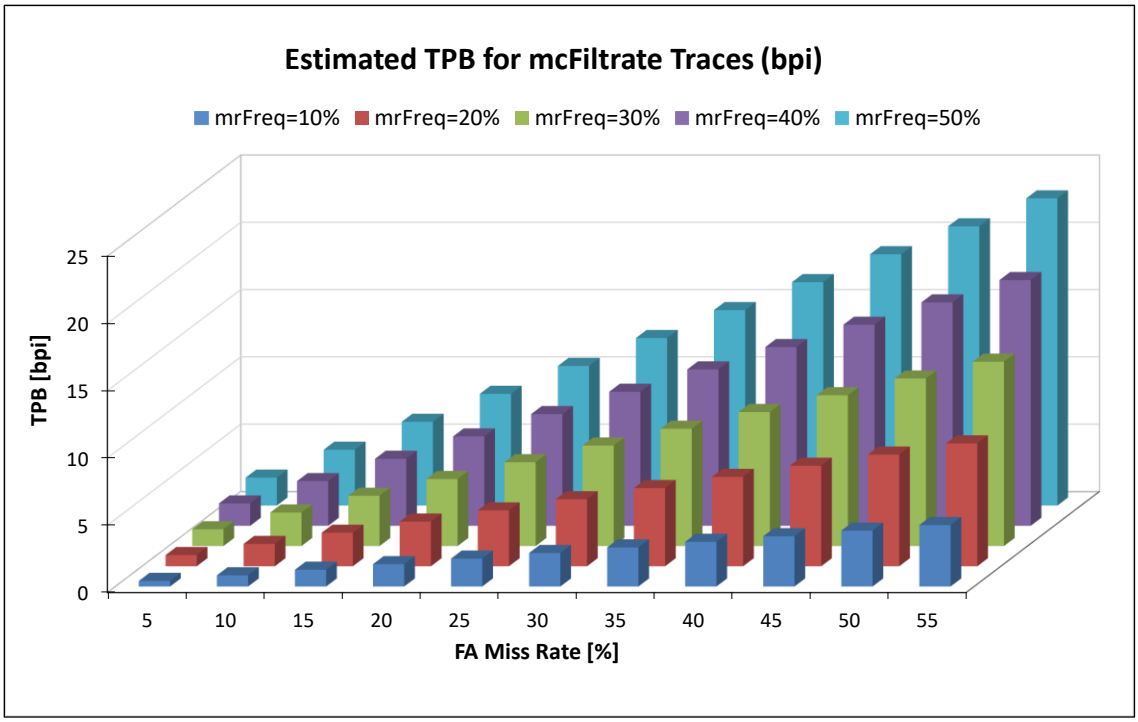


Figure 4.7 Estimation of Required Average TPB in bpi for *mcFiltrate* when N=8

Figure 4.8 shows the estimated TPB in bpc calculated using Eq. (4.2) with IPC = N/4 when N=8. With 10% of memory reads, the required TPB ranges from 0.83 bpc when the FA miss rate is 5% to 9.13 bpc when the FA miss rate is 55%. With 50% of memory reads, the TPB ranges from 4.15 bpc when the FA miss rate is 5% to 45.65 bpc when FA miss rate is 55%. However, the equivalent TPB for NX ranges from 15 bpc to 75 bpc. Thus, *mcFiltrate* reduces the required TPB relative to NX by ~1.6 times (when the FA miss rate is high) to ~17.9 times (when the FA miss rate is low). The effectiveness of *mcFiltrate* in reducing the required TPB for actual workloads is discussed in CHAPTER 6.

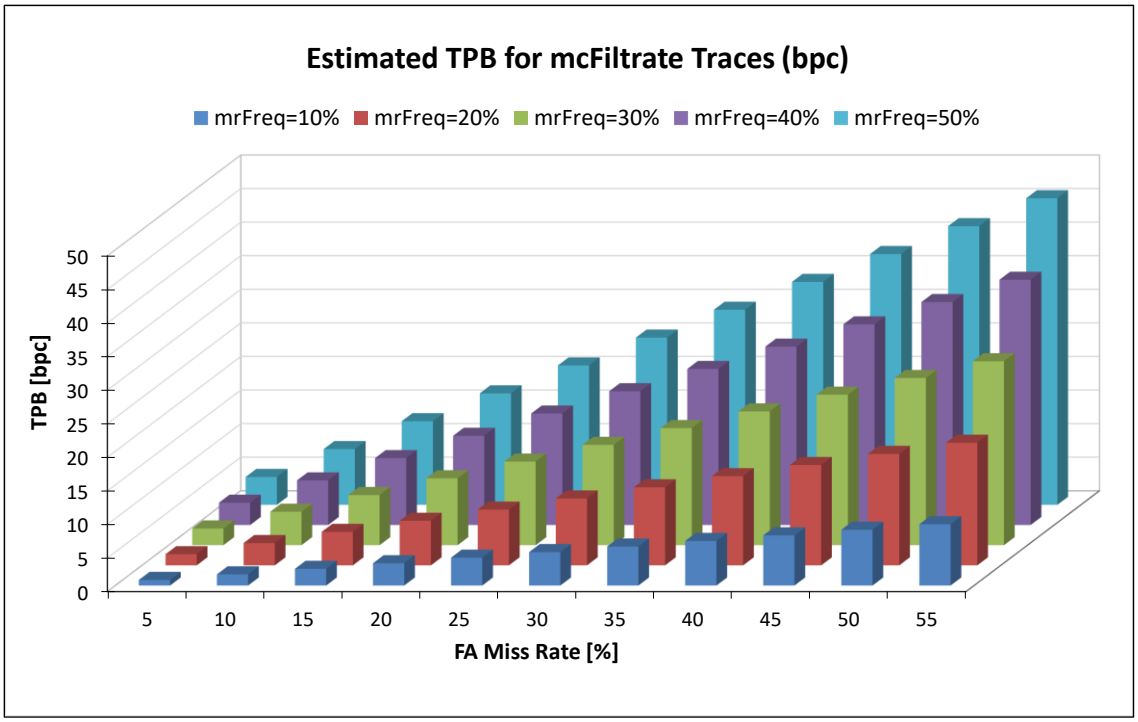


Figure 4.8 Estimation of Required Average TPB in bpc for *mcFiltrate* when N=8

CHAPTER 5

EXPERIMENTAL ENVIRONMENT

This chapter discusses the experimental environment used to evaluate the proposed technique, *mcFiltrate*. We define the metrics for quantitively evaluating the baseline and *mcFiltrate* in Section 5.1. The experimental flow used to create NX and *mcFiltrate* filtered hardware traces is discussed in Section 5.2. Section 5.3 discusses the benchmarks used for the evaluation and Section 5.4 discusses the analysis of experimental parameters like granularity size and encoding parameters.

5.1 Metrics

To evaluate the effectiveness of NX and *mcFiltrate,* we use the average TPB measured in bits per instruction (bpi) and bits per clock cycle (bpc). Each benchmark within a suite exhibits a unique behavior. Benchmarks differ in execution time, number of instructions, frequency of memory reads, size of data reads and others. To summarize the effectiveness of *mcFiltrate* using a single number for an entire benchmark suite, we use the total average TPB. Calculating the arithmetic mean is not a good option since it gives equal weight to all the benchmarks regardless of their trace size. Hence, we calculate the total average TPB that considers read data traces generated by all benchmarks in the suite as shown below:

$$The\ total\ average\ TPB\ [bpi] = \frac{\sum_{all\ bench.} Total\ trace\ size\ in\ bits}{\sum_{all\ bench.} Number\ of\ executed\ instructions} \quad (5.1)$$

$$The\ total\ average\ TPB\ [bpc] = \frac{\sum_{all\ bench.} Total\ trace\ size\ in\ bits}{\sum_{all\ bench.} Execution\ time\ measured\ in\ clock\ cycle} \quad (5.2)$$

We also evaluate the maximum size of on-chip trace buffers required to collect memory read traces without stalling the program to empty the trace buffer when it is full and without losing trace data, while varying the number of bits that can be emitted off-chip per clock cycle (actual emptying trace port bandwidth).

5.2 Experimental Flow

To evaluate the effectiveness of the *mcFiltrate* mechanism in filtering memory read traces, the first step is to determine the impact of configuration parameters such as granularity size and encoding parameters. While evaluating performance, we use the total average TPB measured in bits per instruction (bpi) and bits per clock cycle (bpc) as a yardstick. Although the total average TPB provides an opportunity to quantify the trace port bandwidth requirements, it does not fully capture dynamic changes of the required trace port bandwidth during various phases of a program execution. Thus, we use dynamic trace port bandwidth analysis to determine the peak bandwidth requirements while the benchmark is executing.

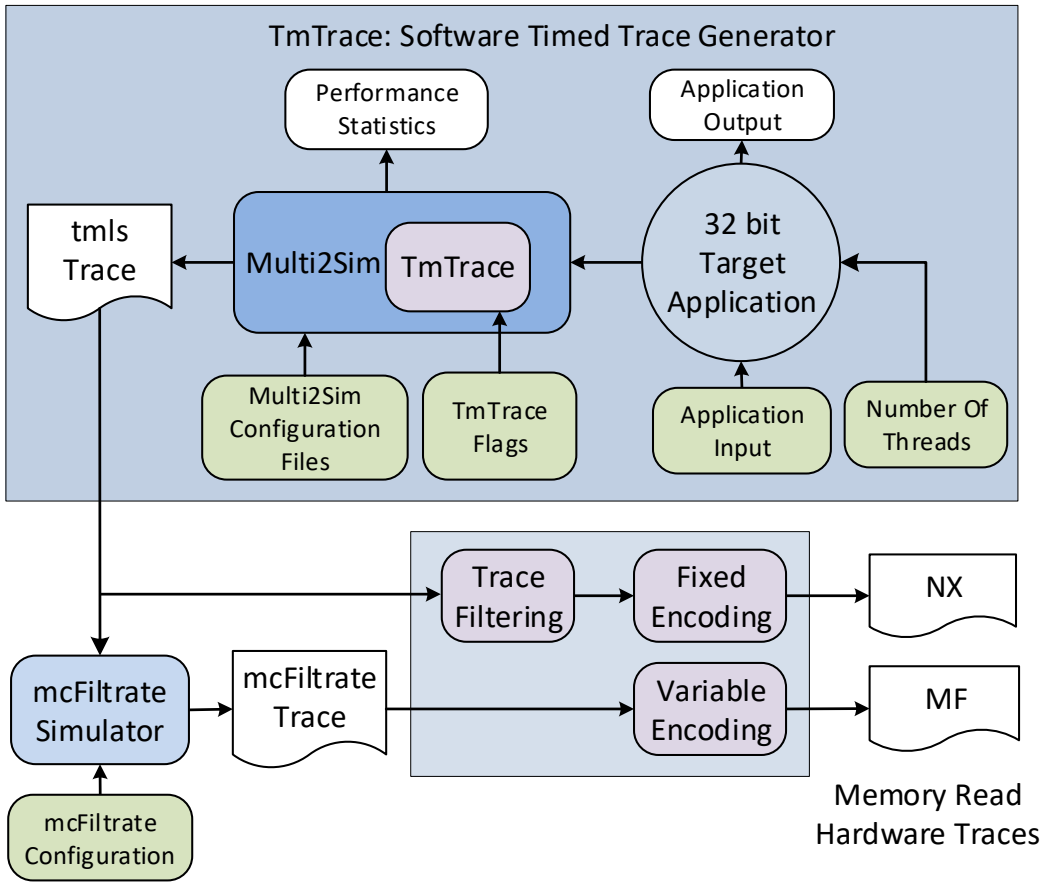


Figure 5.1 Experimental Environment

The experimental flow used to create hardware traces is shown in Figure 5.1. To evaluate *mcFiltrate*, Multi2Sim [95] – a cycle-accurate architectural simulator for Intel32 processors is used. Multi2Sim is extended with a custom *TmTrace* [96] module to facilitate collecting time-stamped memory reads and memory writes (*tmrwTrace*). The timestamp represents the clock cycle in which the trace generating instruction is committed. The traces can be collected by using either a global clock or separate local clocks for each processor. We use global timestamps in the experiments. As a baseline we generate Nexus like (NX) traces which comply with the Nexus 5001 standard [52]. NX traces are produced by filtering the memory reads

from *tmrwTraces* and encoding them using the format of trace messages shown in Figure 4.5. The *tmrwTraces* are also read by the *mcFiltrate* simulator to generate filtered memory read traces as discussed in CHAPTER 4. The filtered traces are processed by variable encoder with the parameters discussed in Table 5.7 to generate the final MF hardware traces. These final hardware traces, NX and MF, are used to determine the trace port bandwidth. To differentiate *mcFiltrate* hardware traces with and without inheriting FA bits from remote caches, we name these traces as MF.I and MF.B, respectively. The *mcFiltrate* simulator implements *mcFiltrate* operations using L1D cache with the MOESI cache coherence protocol.

The Multi2Sim simulator supports building a cycle-accurate model for a multicore processor including processor and memory hierarchy. We use a multicore with up to 8 single-threaded x86 processor cores as shown in Figure 5.2. Each core has its private level 1 instruction (L1I) and data (L1D) caches with a hit latency of 4 clock cycles. To evaluate the effectiveness of *mcFiltrate* as a function of the cache size, we consider three configurations of caches: CS16 with 16 KB L1D/L1I, CS32 with 32 KB L1D/L1I, and CS64 with 64 KB L1D/L1I. The FA tracking bits are added to the L1 data cache tags. The L1 data caches are 4-way set-associative with the Least-Recently Used (LRU) replacement policy, and cache block sizes are set to 32 bytes. The unified L2 cache memory is shared by all cores and has a hit latency of 12 clock cycles. The L2 cache size varies with the number of cores, N, and it is set to N•64 KB for the CS16 configuration, N•128 KB for the CS32 configuration, and N•256 KB for the CS64 configuration. The main memory latency is set to 100 clock cycles.

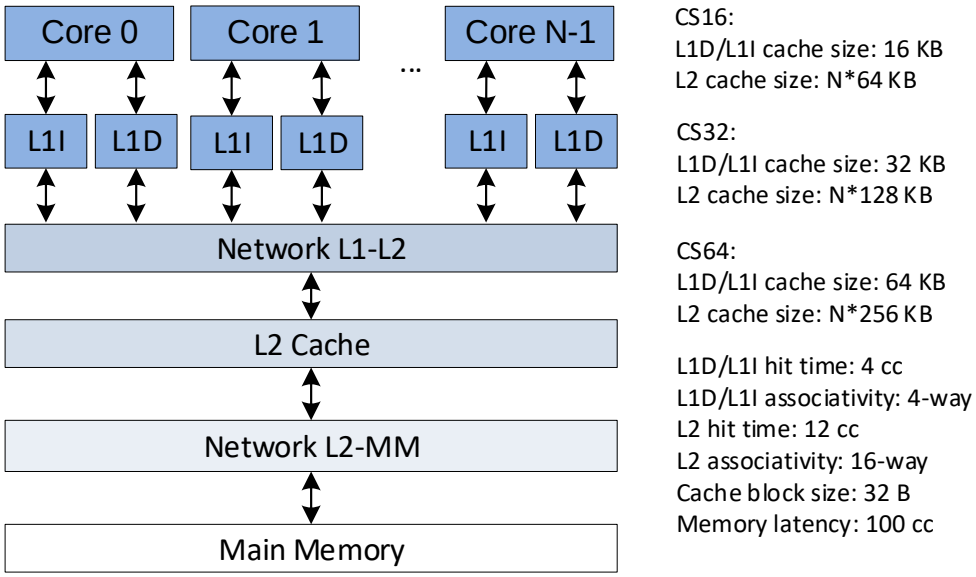


Figure 5.2 Multicore Model

5.3 Benchmarks

To evaluate *mcFiltrate*, we use the Splash2 [22] [97] and Parsec [23] benchmark suites. These suites are composed of multithreaded programs and each benchmark can run multiple input sets differing in size – *test*, *simdev*, *simsamll*, *simmedium*, *simlarge*, and *native*. 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. In our experiments, we use the *simsmall* input set. The benchmarks are precompiled for Intel IA32 instruction set architecture to run on the Multi2Sim simulator that models processors N=1, 2, 4, and 8 cores as shown in Figure 5.2.

Table 5.1 and Table 5.2 show the benchmark characteristics such as the instruction count (IC), the number of instructions executed per clock cycle (IPC), and the frequency of instructions that read from memory for each benchmark from

Splash2 and Parsec, respectively. In Splash2, the smallest benchmark, *lu*, executes about 0.45 billion instructions and the largest benchmark, *water-sp*, executes about 5 billion instructions on a single core (N=1). In Parsec, the smallest benchmark is *blackscholes* with 0.23 billion instructions and the largest benchmark is *x264* with 6.37 billion instructions. Except for *cholesky*, the total number of executed instructions remains constant or increases slightly as the number of cores increases due to the added instructions for synchronization. The total number of instructions executed from both suites is comparable.

Table 5.1 Benchmark Characteristics for Splash2

Benchmarks	Instruction Count [x10^9]				Instructions Per Cycle [IPC]				% Memory Reads			
# 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	22.95	23.20	23.67

Table 5.2 Benchmark Characteristics for Parsec

Benchmarks	Instruction Count [x10^9]				Instructions Per Cycle [IPC]				% Memory Reads			
# 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
blackscholes	0.23	0.23	0.23	0.23	0.52	0.84	1.18	1.50	25.47	25.47	25.47	25.47
bodytrack	1.41	1.41	1.41	-	0.25	0.41	0.62	-	27.51	27.51	27.51	-
canneal	1.58	1.58	1.58	1.58	0.19	0.23	0.26	0.27	32.61	32.61	32.61	32.61
dedup	1.97	1.97	1.98	1.98	0.21	0.38	0.68	1.03	36.97	36.97	36.97	36.97
ferret	1.98	1.99	1.98	1.98	0.25	0.52	0.84	1.07	26.35	26.34	26.35	26.36
fluidanimate	1.77	1.82	1.87	2.02	0.50	0.82	1.10	1.33	27.08	27.25	27.40	27.84
swaptions	0.76	0.76	0.77	0.77	0.43	0.79	1.35	2.02	27.41	27.41	27.33	27.42
vips	3.74	3.74	3.74	3.74	0.26	0.48	0.97	1.70	25.15	25.16	25.16	25.16
x264	6.37	6.35	6.35	6.32	0.64	0.86	1.39	1.90	28.70	28.68	28.66	28.62
Total	19.83	19.86	19.90	18.61	0.32	0.52	0.82	1.09	28.61	28.62	28.62	28.73

The total IPC for the entire benchmark suite is calculated as the sum of all instructions executed by all benchmarks divided by the sum of all execution times in clock cycles as shown in Eq.(5.3). The IPC as a function of the number of cores indicates how well performance scales. Thus, Splash2's *radix* scales poorly because its 8-core speedup is S(8)=IPC(8)/IPC(1)=2.8, but *water-ns* scales well because its 8-core speedup is S(8)=6.4. In Parsec, *canneal* scales poorly with S(8)=1.5 and *vips* scales well with S(8)=6.7.

$$The\ total\ IPC = \frac{\sum_{all\ bench.} Number\ of\ executed\ instructions}{\sum_{all\ bench.} Execution\ time\ measured\ in\ clock\ cycle} \tag{5.3}$$

An important parameter in read data tracing is the size of operands. Table 5.3 and Table 5.4 show the distribution of operand sizes of memory reads in single-threaded benchmarks, for Splash2 and Parsec, respectively. The frequency of in-

structions reading data from memory increases slightly with an increase in the number of cores; hence the distribution is still valid for all other core configurations. In tables Table 5.3 and Table 5.4, a byte operand is 1-byte long, a word operand is 2-bytes long, a double word operand is 4-bytes long, etc. The row named *total* shows the total memory read instructions and also indicates the percentage of memory reads when all the benchmarks are considered together for a given data type. It is calculated by dividing sum of memory reads for corresponding operand size from all the benchmarks with the sum of all memory reads. In Splash2, memory read operations are dominated by double word (4-bytes) and quad word (8-byte) operands. However, very few benchmarks (*radix*) have a significant number of memory reads that are words (29.3%). On the other hand, Parsec is dominated by memory reads which are byte (27.66%) and double word (61.35%) operands. A few benchmarks have a significant percentage of memory reads which are words (*bodytrack, vips*) and quad words (*swaptions, blackscholes*). With these results we could hypothesize that a granularity of size 4 bytes may work better for both benchmark suites. However, before proceeding to evaluate *mcFiltrate*, we need to do the experimental evaluation to determine the granularity size.

Table 5.3 Distribution of Memory Read Operands in Splash2

Benchmarks	Total Memory Reads	% of Memory Reads						
		Byte	Word	Double Word	Quad Word	Extended Precision	Octa Word	Others
barnes	613093875	0	3.26	60.10	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	0.80	0.96	58.93	39.30	0	0	0
water-ns	120913006	0.60	0.01	23.20	76.19	0	0	0
water-sp	874383440	0.55	0.02	22.63	76.80	0	0	0
Total	4463768447	0.92	4.70	50.25	44.14	0	0	0

Table 5.4 Distribution of Memory Read Operands in Parsec

Benchmarks	Total Memory reads	% of Memory Reads						
		Byte	Word	Double Word	Quad Word	Extended Precision	Octa Word	Others
blackscholes	58642406	4.31	0.04	71.16	24.49	0	0	0
bodytrack	388711884	12.95	17.24	69.21	0.60	0	0	0
canneal	513684647	17.85	0	81.47	0.68	0	0	0
dedup	730041560	13.00	12.40	74.60	0	0	0	0
ferret	522918974	11.31	3.92	79.21	5.56	0	0	0
fluidanimate	480078577	2.44	0.31	97.25	0	0	0	0
swaptions	209573296	0.03	0	75.09	24.88	0	0	0
vips	940633141	8.62	14.97	70.57	5.84	0	0	0
x264	1829814170	64.37	8.03	27.60	0	0	0	0
Total	5674098655	27.66	8.24	61.35	2.76	0	0	0

5.4 Experimental Parameters

In this section we evaluate the experimental parameters, granularity size and encoding parameters which are used in the experiments.

5.4.1 Impact of Granularity Size on Trace Port Bandwidth

The effectiveness of the *mcFiltrate* mechanism depends on the size of sub-blocks protected by a single FA bit. Hence, a part of the experimental evaluation is to select granularity size that works well across all benchmarks. As we have seen in Table 5.3 and Table 5.4, the number of memory reads and their distribution among different operand sizes varies from one benchmark to another. Thus, each benchmark may require different granularity size to reduce the trace port bandwidth requirements. Whereas implementing *mcFiltrate* with configurable granularity size is possible, in this dissertation we chose the granularity size which works well among all the benchmarks for all the core configurations. Table 5.5 and Table 5.6 show the total average trace port bandwidth in bpi for Splash2 and Parsec, respectively, while varying the granularity size from 1-byte, G(1), to 32-byte, G(32), with MF.I and CS64 configuration when N=8. The green boxes mark the best-performing granularity size for the given benchmark. As we can observe, overall, the granularity of size 4 (G(4)) works the best for Splash2 and Parsec. The same observation holds for different core configurations (N= 1, 2, 4, and 8), cache configurations (CS16, CS32, and CS64), and *mcFiltrate* configurations (MF.B and MF.I) for both Splash2 and Parsec.

However, *mcFiltrate* performs better with G(32) for some of the Splash2 benchmarks, e.g., *cholesky*, *fft, radix,* reducing the TPB from 6.2% to 25.8% compared to G(4) with CS64 and MF.I (Table 5.5). This finding is the result of strong spatial locality in these benchmarks, where reporting multiple trace messages with

smaller granularity size increases the overhead due to *dts* and *cid* fields. Similarly, several Parsec benchmarks, e.g., *vips*, *fluidanimate*, *x264*, achieve lower TPB for granularity sizes other than G(4), ranging from 0.4% to 14.8% (Table 5.6).

Table 5.5 TPB for MF.I with CS64, N=8 as a Function of Granularity Size for Splash2

Benchmark	Granularity Size				
	G(1)	G(4)	G(8)	G(16)	G(32)
barnes	0.2126	0.1975	0.2558	0.2620	0.3196
cholesky	0.3128	0.3128	0.2958	0.3007	0.2932
Fft	1.2765	1.2765	1.2766	1.2190	1.1353
fmm	0.1373	0.1372	0.1801	0.2093	0.2453
lu	0.0523	0.0523	0.0526	0.0841	0.0718
radiosity	0.0379	0.0379	0.0502	0.0553	0.0636
radix	0.5375	0.5375	0.4717	0.4241	0.3987
Raytace	0.0679	0.0684	0.0783	0.0903	0.1024
Swater-ns	0.0128	0.0127	0.0156	0.0239	0.0304
water-sp	0.0230	0.0230	0.0240	0.0318	0.0399
Total	0.1960	0.1945	0.2016	0.2054	0.2136

Table 5.6 TPB for MF.I with CS64, N=8 as a Function of Granularity Size for Parsec

Benchmark	Granularity Size				
	G(1)	G(4)	G(8)	G(16)	G(32)
blackscholes	0.5718	0.5720	0.8490	1.4126	2.5463
bodytrack	-	-	-	-	-
canneal	1.4369	1.2993	1.7351	2.2709	3.1809
dedup	0.9781	0.8806	1.0285	1.1223	1.2894
ferret	0.5638	0.5399	0.5858	0.6573	0.7650
fluidanimate	0.2150	0.2158	0.2134	0.2131	0.2144
swaptions	0.0008	0.0008	0.0013	0.0149	0.0139
vips	0.9927	1.1653	1.0725	1.0290	1.0248
x264	0.2822	0.1611	0.1524	0.1544	0.1785
Total	0.6113	0.5805	0.6195	0.6819	0.8096

5.4.2 Impact of Encoding Parameters on Trace Port Bandwidth

The format of the encoded trace messages is shown in Figure 4.5. The number of bits required to encode *dts* and *fct* varies during different phases of program execution and depends on the frequency and distribution of FA misses, which in turn depend on the number of cores and cache configurations. Chunk sizes can be tailored to each benchmark to get the best possible compression ratio. However, in our experiments we seek chunk sizes that perform well across all the benchmarks to reduce the complexity of encoding hardware. We can utilize multiple chunks with different lengths *h0, h1, h2... hk* and *i0, i1, i2... ik* to encode *dts* and *fct*, respectively, as shown in Figure 4.5. However, to reduce the complexity we limit the search space by setting *h1=h2=…=hk* and *i1=i2=…=ik*.

Figure 5.3 shows the average bit length for *dts* and *fct* when all the benchmarks are considered together while varying the chunk size with CS16 for Splash2.

The chunk sizes are varied from (2,1) to (6,6) for (*h0,h1*) and (*i0,i1*). As we can see, the chunk size (*h0,h1*)=(4,2) works the best for *dts* and (*i0,i1*)=(2,2) works the best for *fct*. Similar experimental analysis is performed with different cache configurations and *mcFiltrate* configurations. We find that chunk sizes (*h0,h1*)=(4,2) and (*i0,i1*)=(2,2) work the best for the Parsec benchmark suite, regardless of the flavor of *mcFiltrate* and the cache configurations (Table 5.7). However, Splash2 requires different encoding parameters (*h0,h1*)=(4,2) or (5,4) and (*i0,i1*)=(4,2) or (2,2) depending on the configuration as summarized in (Table 5.7). Surprisingly we find that the chunk sizes given in (Table 5.7) exhibit good performance regardless of the number of cores.

Table 5.7 Encoding Parameters

Mechanism	Encoding Parameters	Splash2			Parsec		
		CS16	CS32	CS64	CS16	CS32	CS64
MF.B	h0,h1	4,2	4,2	5,4	4,2	4,2	4,2
	i0,i1	2,2	3,2	3,2	2,2	2,2	2,2
MF.I	h0,h1	4,2	4,2	5,4	4,2	4,2	4,2
	i0,i1	2,2	2,2	3,2	2,2	2,2	2,2

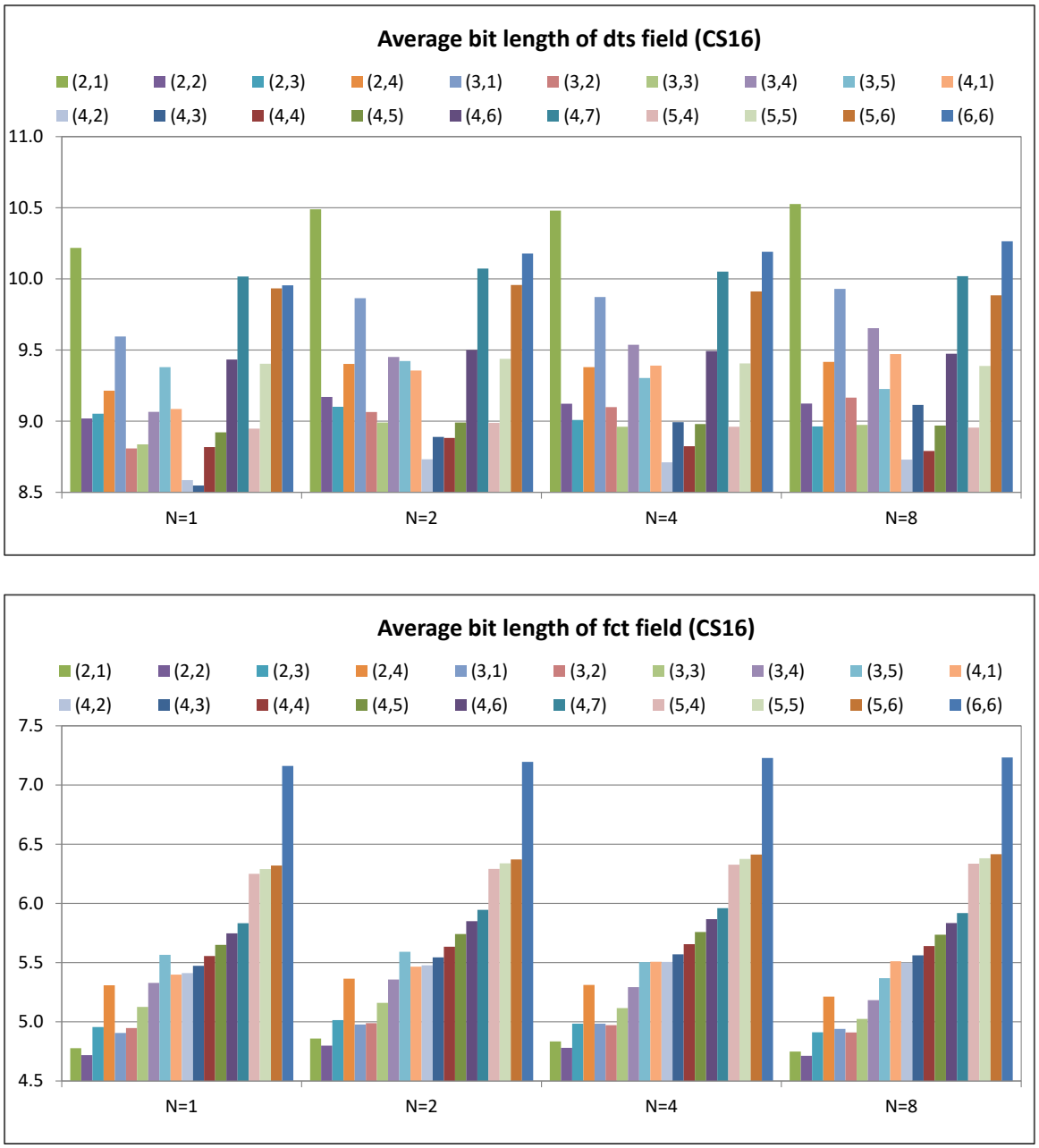


Figure 5.3 Encoding Parameter Selection for Splash2 with MF.I and CS16

CHAPTER 6

RESULTS

This chapter compares the results from the experimental evaluation of *mcFiltrate* and the existing and the proposed techniques for read data tracing. The effectiveness of *mcFiltrate* depends on the FA miss rate. Hence, we first evaluate the FA miss rate (Section 6.1) and then the average trace port bandwidth (TPB) measured in bits per instruction (bpi) (Section 6.2) and bits per clock cycle (bpc) (Section 6.3). Since the average TPB does not capture changes in the required trace port during program execution, we also evaluate the dynamic required TPB in Section 6.4. Section 6.5 describes the results of trace buffer analysis under different emptying rates on the trace port. Finally, Section 6.6 discusses the hardware complexity of *mcFiltrate*.

6.1 First-access Miss Rate

The effectiveness of *mcFiltrate* depends on the FA miss rate as described in an analytical model given in Eq.(4.1). The FA miss rate is, in turn, a function of the L1 data cache read miss rate. So, first we will discuss the L1 data cache read miss rates as a function of the data cache size. From now on the L1 data cache read miss rate is simply referred to as cache read miss rate. The total cache read miss rate per benchmark suite is calculated by dividing the sum of cache read misses from all the benchmarks by the sum of executed instructions from all the benchmarks as shown below:

$$Total\ Cache\ Read\ Miss\ Rate\ = \frac{\sum_{all\ bench.} Number\ of\ cache\ read\ misses}{\sum_{all\ bench.} Number\ of\ executed\ instructions} \quad (6.1)$$

The cache read miss rate depends on the size of the data cache, the temporal and spatial locality of data accesses, coherence invalidations in multicores, and the cache replacement policy. Figure 6.1 shows the total cache read miss rates for Splash2 and Parsec benchmark suites with different cache configurations (CS16, CS32, CS64), while varying the number of cores (N=1, 2, 4, 8). The error bars represent the minimum and maximum cache read miss rates for a given configuration within the given benchmark suite. In Splash2 with CS16, the total cache read miss rate is 1.7% and it ranges from 0.19% (*water-sp*) to 4.6% (*fft*) when N=1. As the cache size increases, the total cache read miss rate decreases due to a lower number of capacity misses. Thus, in Splash2 with CS64, the cache read miss rate is 0.36% and it ranges from 0.05% (*water-sp*) to 2.1% (*fft*). As the number of cores increases, the capacity misses still dominate in configurations with smaller caches, whereas the cold and coherence misses dominate in configurations with larger caches. As a result, the cache read miss rate increases as the number of cores increases. Thus, when N=8 the total cache read miss rate ranges from 1.2% (CS64) to 2% (CS16) and it reaches a maximum of 4.6% for CS16 (*fft*) and 2.87% for CS64 (*fft*).

Parsec follows similar trends and has cache read miss rates comparable to those in Splash2. The total cache read miss rate ranges from 0.93% (CS64) to 1.62% (CS16) when N=1 and 1.34% (CS64) to 1.98% (CS16) when N=8. It reaches as high as 5.87% when N=8 with CS64 for *blackscholes*.

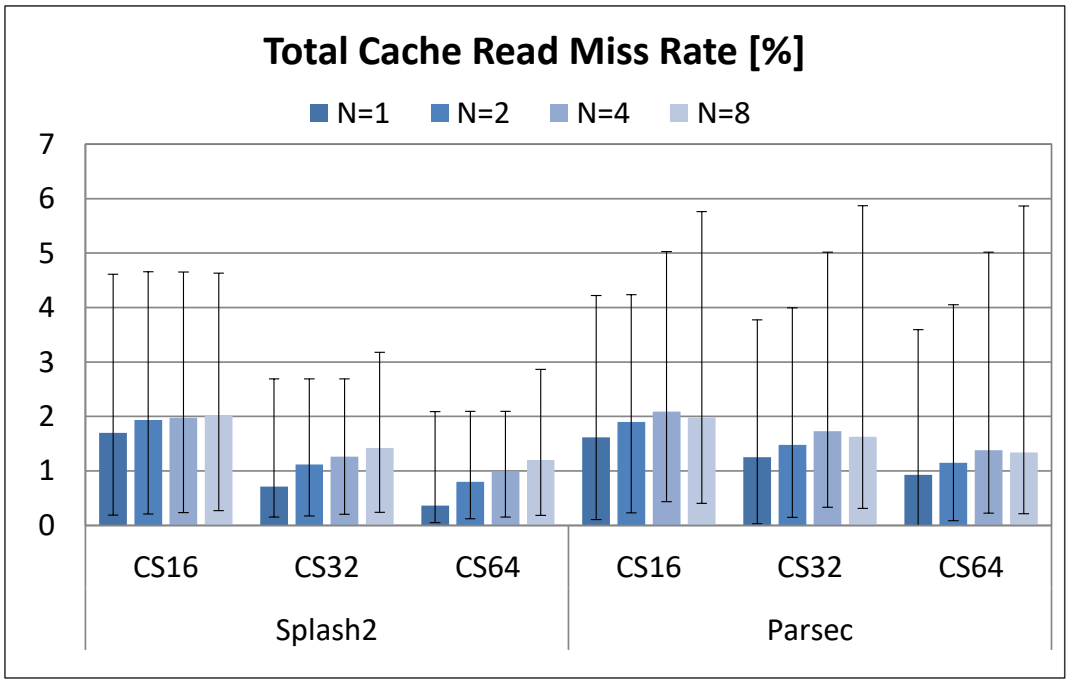


Figure 6.1 Total L1 Data Cache Read Miss Rate

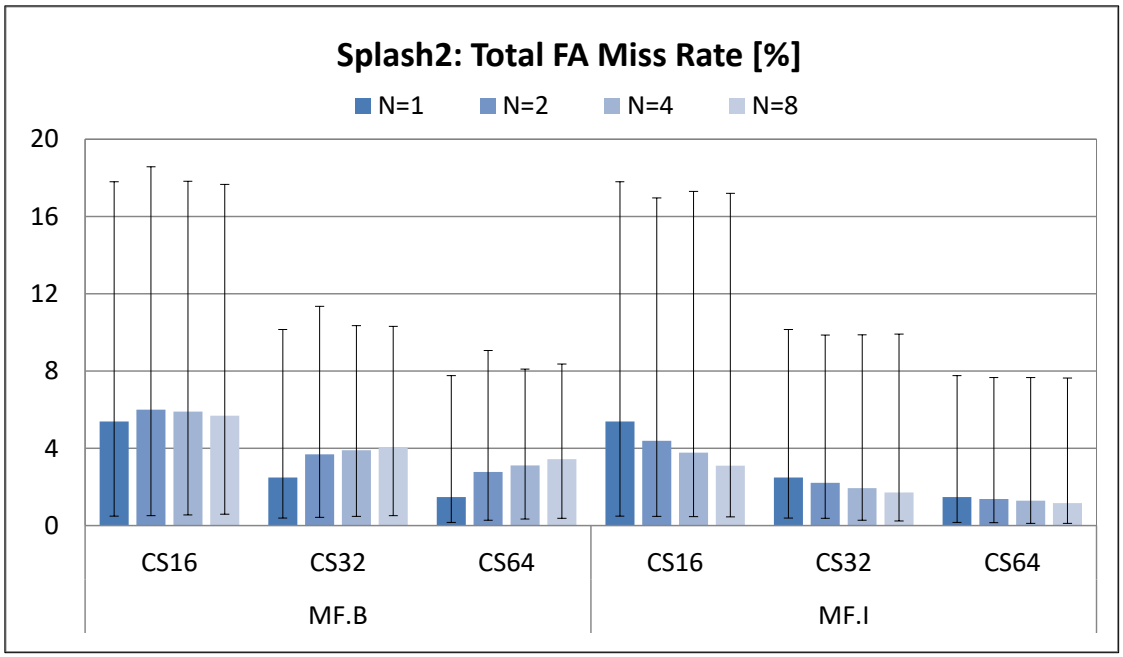
The FA miss rates are higher than the cache read miss rates. A cache block fetched into the data cache (a single read miss) may contain multiple data items that are read consecutively, each resulting in a single FA miss. Thus, a 32-byte cache block that contain 8 4-byte data items that are read consecutively, results in 8 FA misses. If a data item is guarded by multiple FA bits, an FA hit requires that all FA bits are set; if at least one FA bit is not set, we consider this event as a FA miss. The total FA miss rate of a benchmark suite is calculated as shown in Eq.(6.2)

$$Total\ FA\ Miss\ Rate\ = \frac{\sum_{all\ bench.} Number\ of\ FA\ misses}{\sum_{all\ bench.} Number\ of\ executed\ instructions} \tag{6.2}$$

Figure 6.2 shows the total first-access (FA) miss rate for Splash2 (top) and Parsec (bottom) while varying the number of cores and cache configurations. In Splash2, the FA miss rate ranges from 1.47% (CS64) to 5.39% (CS16) when N=1. As

the number of cores increases, the FA miss rate increases for MF.B because of an increase in the cache read miss rate. Thus, when N=8, the FA miss rate ranges from 3.44% (CS64) to 5.68% (CS16). However, opposite trends occur with MF.I when inheriting the FA bits in cache-to-cache transfers. With MF.I, the FA miss rate ranges from 1.16% (CS64) to 3.10% (CS16) when N=8. The maximum FA miss rate reaches as high as ~18% for *fft* with CS16 and ~7.8% with CS64, regardless of the number of cores. One interesting observation is that some benchmarks, e.g., *fft* and *cholesky*, do not benefit from inheriting FA bits and some benchmarks benefit greatly, e.g., *barnes* (28% to 89%), *radiosity* (76% to 93%), and *water-ns* (6% to 96%). In these benchmarks, the benefit of MF.I increases as the number of cores increases.

In Parsec, the total FA miss rate ranges from 5.36% when N=1 to 5.94% when N=8 with CS16 and 3.82% when N=1 to 4.45% when N=8 with CS64. Similar trends as in Splash2 are observed in Parsec, except that the FA miss rate does not decrease significantly as we increase the data cache size. This is because the Parsec data set does not fit completely in the data caches explored in our experiments. Overall, regardless of the cache configurations and the number of cores, the FA miss rate is less than 11.3%. Whereas MF.I has no or little impact in reducing the FA misses in some benchmarks – *canneal*, *dedup*, and *ferret*, it greatly helps *blackscholes*, *bodytrack*, *fluidanimate*, *and swaptions*. Overall, MF.I reduces the FA miss rate by 8% to 18% depending on the benchmark and cache configuration. These results confirm that *mcFiltrate* has a great potential to reduce the required TPB.



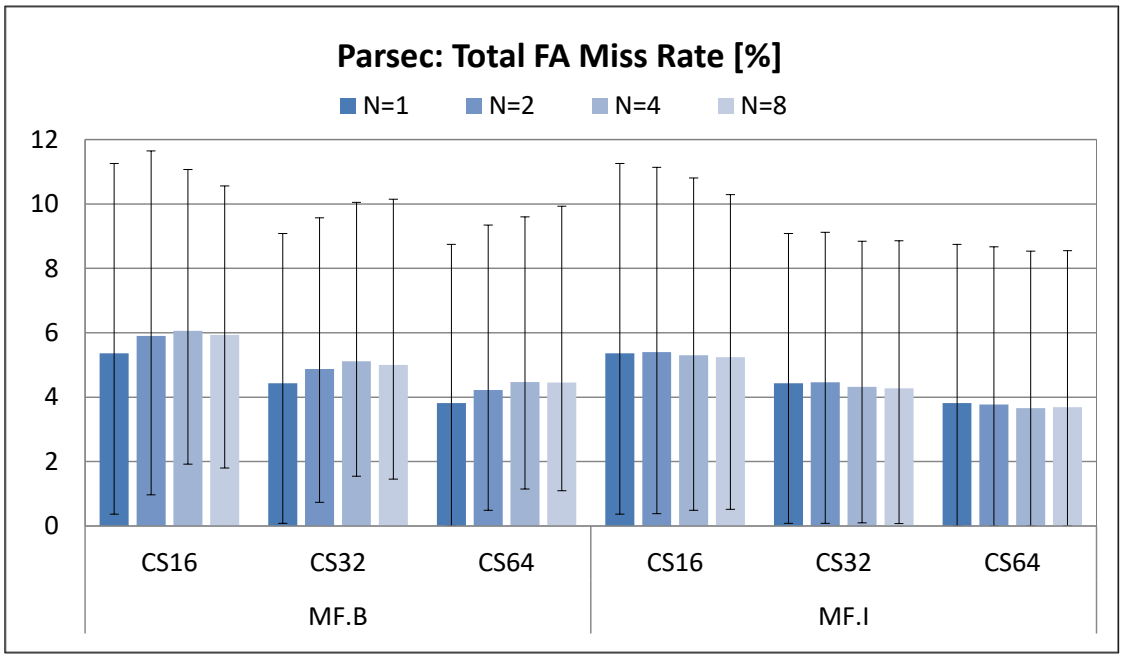


Figure 6.2 Total First-access Miss Rate of Splash2 (top) and Parsec (bottom)

6.2 Trace Port Bandwidth in BPI

The average TPB in bpi for NX and MF tracing for Splash2 with CS16 and CS64 is shown in Table 6.1 and Table 6.2, respectively. The results for CS32 are shown in Appendix A.1. The light green and orange boxes mark the benchmarks that have the lowest and highest TPB for a given column and thick borders mark the cases where inheriting FA bits has no advantage. In the case of NX tracing for Splash2, the total average TPB is 12.34 bpi, and ranges from 8.82 bpi (*fmm*) to 15.35 bpi (*cholesky*) when N=1. As the cache size increases, the average TPB remains constant. It increases slightly as the number of cores increases because of an increase in the number of memory reads due to synchronizations in multicores and the additional bits needed to encode the core/thread identifier (*cid*). The total average TPB when N=8 is 13.17 bpi and it ranges from 9.33 bpi (*fmm*) to 15.86 bpi (*barnes*). When we consider individual benchmarks, *barnes*, *cholesky*, *radix*, and *raytrace* require high TPB because of higher frequency of memory reads and larger average size of operands read from memory.

MF.B with CS16 requires an average TPB of 0.77 bpi (ranges from 0.07 bpi to 2.57 bpi) when N=1 and 0.88 bpi (ranges from 0.08 bpi to 2.67 bpi) when N=8. As the cache size increases, the average TPB decreases because of a decrease in FA miss rates. Thus, with CS64 the average TPB ranges from 0.21 bpi when N=1 to 0.53 bpi when N=8. In MF.B, the average TPB increases due to the additional bits emitted to report the core/thread identifier and due to an increase in the number of coherence misses. However, MF.I takes advantage of cache-to-cache transfers to reduce the TPB by inheriting the FA bits. Thus, with MF.I the average TPB goes down up to ~97% depending on the benchmark, number of cores, and cache configurations.

When we look at individual benchmarks with MF.I *fft* does not benefit at all, *cholesky* and *fmm* benefit 9-15%, and all other benchmarks benefit at least ~40% when N=8, regardless of the cache configuration. Thus, the total average TPB decreases to 0.48 bpi with CS16 and 0.18 bpi with CS64 when N=8.

Table 6.1 Average TPB in bpi for Splash2 with CS16

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.03	2.17	15.31	2.26	1.63	15.59	2.35	1.19	15.86	2.40	0.89
cholesky	15.35	1.76	16.21	1.21	1.20	15.87	0.88	0.83	15.59	0.56	0.50
fft	10.65	2.57	10.84	2.62	2.62	11.02	2.65	2.65	11.19	2.67	2.67
fmm	8.82	0.33	8.96	0.34	0.30	9.14	0.34	0.28	9.33	0.35	0.28
lu	11.88	0.54	12.07	0.57	0.54	12.27	0.58	0.45	12.47	0.61	0.43
radiosity	12.11	0.22	12.36	0.51	0.12	12.59	0.51	0.12	12.58	0.59	0.09
radix	13.41	0.75	13.75	1.64	0.77	14.09	1.73	0.79	14.54	1.78	0.82
raytrace	15.17	0.93	15.45	1.10	0.78	15.73	1.18	0.72	16.01	1.34	0.64
water-ns	10.64	0.47	10.81	0.50	0.48	10.98	0.54	0.37	11.15	0.54	0.08
water-sp	11.38	0.07	11.55	0.07	0.06	11.73	0.08	0.06	11.90	0.08	0.06
Total	12.34	0.77	12.63	0.88	0.64	12.89	0.88	0.56	13.17	0.88	0.48

Table 6.2 Average TPB in bpi for Splash2 with CS64

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.02	0.19	15.31	0.67	0.16	15.59	1.04	0.16	15.86	1.50	0.18
cholesky	15.29	0.62	16.23	0.65	0.62	15.87	0.51	0.45	15.62	0.36	0.29
fft	10.63	1.15	10.82	1.17	1.17	11.00	1.18	1.18	11.19	1.19	1.19
fmm	8.82	0.14	8.96	0.14	0.14	9.14	0.15	0.13	9.33	0.15	0.13
lu	11.86	0.47	12.06	0.30	0.28	12.26	0.32	0.19	12.47	0.16	0.05
radiosity	12.11	0.04	12.32	0.39	0.03	12.61	0.40	0.04	12.61	0.50	0.03
radix	13.38	0.43	13.74	1.29	0.44	14.09	1.36	0.46	14.53	1.42	0.47
raytrace	15.16	0.11	15.45	0.34	0.09	15.73	0.44	0.07	16.01	0.62	0.06
water-ns	10.64	0.02	10.81	0.05	0.02	10.97	0.31	0.01	11.15	0.39	0.01
water-sp	11.38	0.03	11.55	0.04	0.03	11.73	0.05	0.03	11.90	0.05	0.02
Total	12.33	0.21	12.62	0.40	0.20	12.88	0.47	0.19	13.17	0.53	0.18

Table 6.3 and Table 6.4 show the average TPB in bpi for NX and MF for Parsec with CS16 and CS64, respectively. In the case of NX tracing, the total average TPB ranges from 9.71 bpi to 10.62 bpi when N=8. Even though the frequency of memory reads in Parsec is comparable to Splash2, the total TPB in Parsec is less than that in Splash2. This is because the average size of the operands read from memory is different (Table 5.3 and Table 5.4). Whereas in Splash2 ~94% of the memory reads are either 4-bytes or 8-bytes, in Parsec only ~61.35% of the memory reads are 4-bytes and 27.66% are 1-byte.

Table 6.3 Average TPB in bpi for Parsec with CS16

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.18	0.49	12.43	0.97	0.49	12.68	1.13	0.50	12.94	1.18	0.51
bodytrack	9.77	0.62	10.02	0.97	0.59	10.29	1.11	0.55	-	-	-
canneal	12.12	1.32	12.41	1.34	1.33	12.72	1.36	1.36	13.04	1.39	1.39
dedup	13.32	1.15	13.69	1.17	1.17	14.03	1.19	1.18	14.39	1.21	1.20
ferret	10.42	1.49	10.67	1.50	1.50	10.93	1.51	1.51	11.19	1.47	1.46
fluidanimate	10.94	0.24	11.28	0.42	0.24	11.63	0.59	0.25	12.14	0.65	0.24
swaptions	13.42	0.06	13.69	0.20	0.07	13.96	0.45	0.09	14.24	0.73	0.10
vips	9.71	1.13	9.95	1.23	1.14	10.19	1.26	1.16	10.43	1.33	1.17
x264	6.98	0.22	7.22	0.28	0.28	7.49	0.27	0.26	7.76	0.26	0.25
Total	9.74	0.73	10.01	0.82	0.75	10.30	0.86	0.75	10.62	0.87	0.76

With MF.B, the total average TPB ranges from 0.73 bpi (N=1) to 0.87 bpi (N=8) with CS16 and 0.51 bpi (N=1) to 0.65 bpi (N=8) with CS64. MF.I reduces the total TPB by 9% to 20%. However, when we look at individual benchmarks, only three benchmarks *blackscholes*, *fluidanimate*, and *swaptions*, benefit from inheriting FA bits (~43% to ~100%) regardless of the cache configuration, whereas other benchmarks see little or no benefit at all. Thus, the total average TPB ranges from 0.73 bpi (N=1) to 0.76 bpi (N=8) with CS16 and 0.51 bpi (N=1) to 0.52 bpi (N=8) with CS64.

Table 6.4 Average TPB in bpi for Parsec with CS64

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.17	0.47	12.43	0.96	0.47	12.68	1.13	0.48	12.94	1.17	0.49
bodytrack	9.72	0.42	10.00	0.79	0.34	10.29	0.93	0.28	-	-	-
canneal	12.06	1.10	12.39	1.12	1.12	12.71	1.15	1.14	13.04	1.18	1.16
dedup	13.28	0.79	13.66	0.80	0.79	14.02	0.81	0.80	14.39	0.83	0.81
ferret	10.39	0.50	10.66	0.48	0.48	10.93	0.48	0.48	11.19	0.48	0.47
fluidanimate	10.93	0.21	11.28	0.38	0.20	11.63	0.56	0.20	12.13	0.61	0.19
swaptions	13.42	0.00	13.69	0.11	0.00	13.97	0.42	0.00	14.24	0.64	0.00
vips	9.67	1.00	9.93	1.06	1.03	10.17	1.13	1.04	10.43	1.24	1.05
x264	6.97	0.14	7.21	0.17	0.17	7.49	0.16	0.16	7.76	0.16	0.15
Total	9.71	0.51	10.00	0.58	0.51	10.29	0.63	0.51	10.62	0.65	0.52

To underscore the effectiveness of *mcFiltrate*, we compare it to a software compressor. We feed the NX traces to the *gzip* compressor with level-1 compression to determine the trace port bandwidth requirements, if such a utility is implemented in hardware. The reason to use compression level-1 is that it requires smaller buffers, offering compression ratios that closer match those that could be achieved in hardware compressors. Please note that implementing a full general-purpose compressor requires a significant amount of hardware resources.

We consider two variants of software compression of NX traces. The first one, NX_u.gz, feeds encoded NX trace messages directly to the *gzip* compressor. However, the NX messages combine header information with data values, thus limiting the compression ratio. To exploit redundancies in data values (*mrv* field), the trace messages are divided into two streams that are compressed separately, one with memory read values (*mrv*) and the other with timestamp and core identifier (*dts*,

cid). This approach is referred to as NX_s.gz. Figure 6.3 shows the total average compression ratios achieved by the *gzip* compressors and *mcFiltrate* for Splash2 and Parsec, while varying the number of cores and cache configurations. Table 6.5 and Table 6.6 also show the compression ratios for individual benchmark from Splash and Parsec respectively for CS64. The compression ratios given in Table 6.5 and Table 6.6 are calculated using Eq (6.3):

$$Compression\ Ratio\ = \frac{Required\ TPB\ for\ NX}{Required\ TBP\ for\ NX\_u.gz\ or\ NX\_s.gz\ or\ mcFiltate} \qquad (6.3)$$

The compression ratios achieved with NX_u.gz are relatively low, ranging from 1.5 to 1.7 regardless of the number of cores. The maximum possible compression ratio is 2.5 for *cholesky* from Splash2 and 2.0 for *fluidanimate* from Parsec. Even with NX_s.gz, the compression ratio is still relatively moderate and it is in the range of 2.1 to 3.4 for Splash2 and Parsec. The maximum compression ratio that can be achieved is 7.29 for *cholesky* from Splash2 and 5.45 for *blackscholes* from Parsec.

mcFiltrate reduces the total TPB regardless of the configurations and the number of cores. For Splash2 with CS16, the overall compression ratio ranges from 16.1 (N=1) to 15 (N=8) with MF.B. The compression ratio achieved with CS64 is even higher because of reduced FA miss rates. Thus, the overall compression ratio ranges from 59.6 (N=1) to 24.8 (N=8). However, with MF.I the overall compression ratio reaches to 27.59 with CS16 and to 73.8 with CS64 when N=8. If we consider individual benchmarks, all the benchmarks except *fft* achieve good compression ratios regardless of the cache configuration. The maximum achieved compression ratio is 944.9 for *water-ns* with CS64 when N=8. For Parsec with MF.I, the overall compression ratio ranges from 13.38 (N=1) to 14.05 (N=8) with CS16 and from 19.22

(N=1) to 20.3 (N=8) with CS64. However, the maximum possible compression ratio is 139,655 with CS64 when N=4 for *swaptions*.

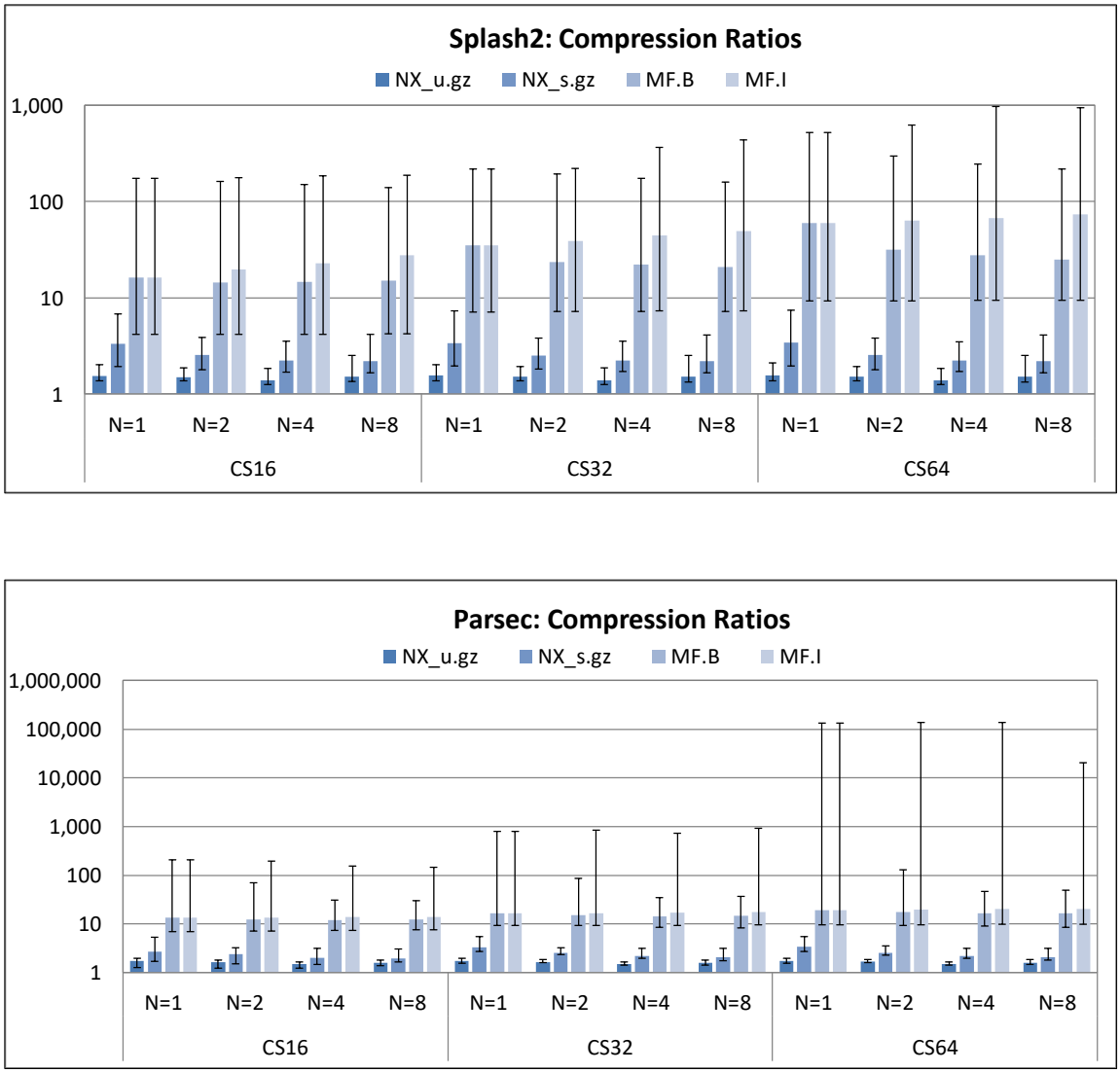


Figure 6.3 Compression Ratios of Splash2 (top) and Parsec (bottom)

Table 6.5 Compression Ratios for Splash2 with CS64

# Cores	N=1			N=2				N=4				N=8			
Mechanism	NX_s .gz	NX_u .gz	MF.B|I	NX_s .gz	NX_u .gz	MF.B	MF.I	NX_s .gz	NX_u .gz	MF.B	MF.I	NX_s .gz	NX_u .gz	MF.B	MF.I
barnes	2.21	1.40	79.49	1.79	1.31	22.89	94.57	1.66	1.24	14.95	98.66	1.58	1.27	10.59	85.98
cholesky	7.36	1.79	24.73	3.78	1.67	25.12	25.99	3.50	1.83	31.14	35.07	4.09	2.50	43.95	54.36
fft	1.94	1.39	9.24	1.80	1.37	9.28	9.28	1.70	1.31	9.33	9.33	1.67	1.38	9.39	9.40
fmm	5.05	1.98	64.82	3.82	1.92	62.98	66.34	3.10	1.59	61.78	68.65	2.77	1.60	60.73	72.93
lu	5.95	1.60	25.04	3.58	1.57	39.77	43.75	3.11	1.43	38.04	64.67	3.10	1.80	75.64	250.97
radiosity	3.90	1.64	288.25	2.50	1.55	31.57	355.07	2.10	1.38	31.41	343.56	1.95	1.48	25.27	369.85
radix	4.31	2.11	30.87	2.96	1.91	10.68	30.94	2.09	1.47	10.38	30.88	1.96	1.42	10.25	30.85
raytrace	3.99	1.52	140.54	2.60	1.49	45.56	178.39	2.25	1.32	36.01	211.96	2.08	1.40	25.62	255.68
water-ns	2.72	1.41	521.52	2.09	1.39	232.90	628.28	1.92	1.27	35.21	971.23	1.86	1.34	28.85	944.94
water-sp	3.04	1.36	354.44	2.41	1.39	295.43	391.57	2.11	1.26	244.78	449.24	2.02	1.39	218.33	543.33
Total	3.41	1.56	59.58	2.52	1.51	31.49	63.44	2.21	1.38	27.68	67.63	2.17	1.52	24.85	73.80

Table 6.6 Compression Ratios for Parsec with CS64

# Cores	N=1			N=2				N=4				N=8			
Mechanism	NX_s .gz	NX_u .gz	MF.B|I	NX_s .gz	NX_u .gz	MF.B	MF.I	NX_s .gz	NX_u .gz	MF.B	MF.I	NX_s .gz	NX_u .gz	MF.B	MF.I
blackscholes	5.45	1.68	25.94	3.51	1.69	12.97	26.21	2.47	1.43	11.25	26.28	2.24	1.45	11.11	26.44
bodytrack	3.22	1.69	22.92	2.66	1.67	12.65	29.02	2.26	1.49	11.06	36.49	-	-	-	-
canneal	3.48	1.66	11.00	3.32	1.74	11.02	11.06	3.13	1.67	11.06	11.13	3.14	1.87	11.10	11.22
dedup	2.86	1.78	16.74	2.31	1.70	17.08	17.25	2.04	1.54	17.30	17.63	1.94	1.56	17.33	17.85
ferret	3.75	1.73	20.70	2.53	1.66	22.07	22.10	2.21	1.53	22.87	22.93	2.15	1.63	23.53	23.67
fluidanimate	4.81	2.00	53.30	3.05	1.86	29.39	56.73	2.58	1.64	20.90	56.85	2.40	1.65	19.74	63.50
swaptions	3.94	1.83	134168	2.71	1.73	128.67	136909	2.30	1.49	33.37	139655	2.15	1.49	22.08	20343
vips	4.05	1.82	9.63	2.34	1.62	9.41	9.68	1.95	1.47	8.97	9.77	1.80	1.45	8.41	9.93
x264	2.75	1.56	49.00	2.42	1.60	41.95	42.69	1.98	1.45	46.18	48.26	1.97	1.58	49.31	52.38
Total	3.40	1.73	19.22	2.56	1.67	17.32	19.56	2.18	1.52	16.31	20.316	2.09	1.58	16.37	20.319

Understanding how the required trace port bandwidth is utilized across different fields of the trace messages is helpful to design and attain a better compression ratio if required. The distribution of trace port bandwidth among different fields of trace message, *dts*, *cid*, *fct*, *mrv*, for Splash2 and Parsec for CS64 is shown in Figure 6.4. Expectedly, much of the required trace port bandwidth is occupied by the memory read values (*mrv*). For NX, the *mrv* portion ranges from 83% for N=1 to 78% for N=8. The time field is responsible for ~17% and ~27% of the total required TPB regardless of the number of cores for Splash2 and Parsec, respectively. Thus, ordering the trace messages coming from different cores in the trace buffer and streaming them out without the time field reduces the required TPB only by ~17% and ~27% for Splash2 and Parsec, respectively. However, buffering and ordering the trace messages from multiple cores requires additional hardware support. With *mcFiltrate*, the *mrv* field accounts for 68%-80% of the trace port bandwidth depending on the number of cores, the *fct* field for ~10%, and the *dts* field for ~19% regardless of the cache configuration for both benchmark suites.

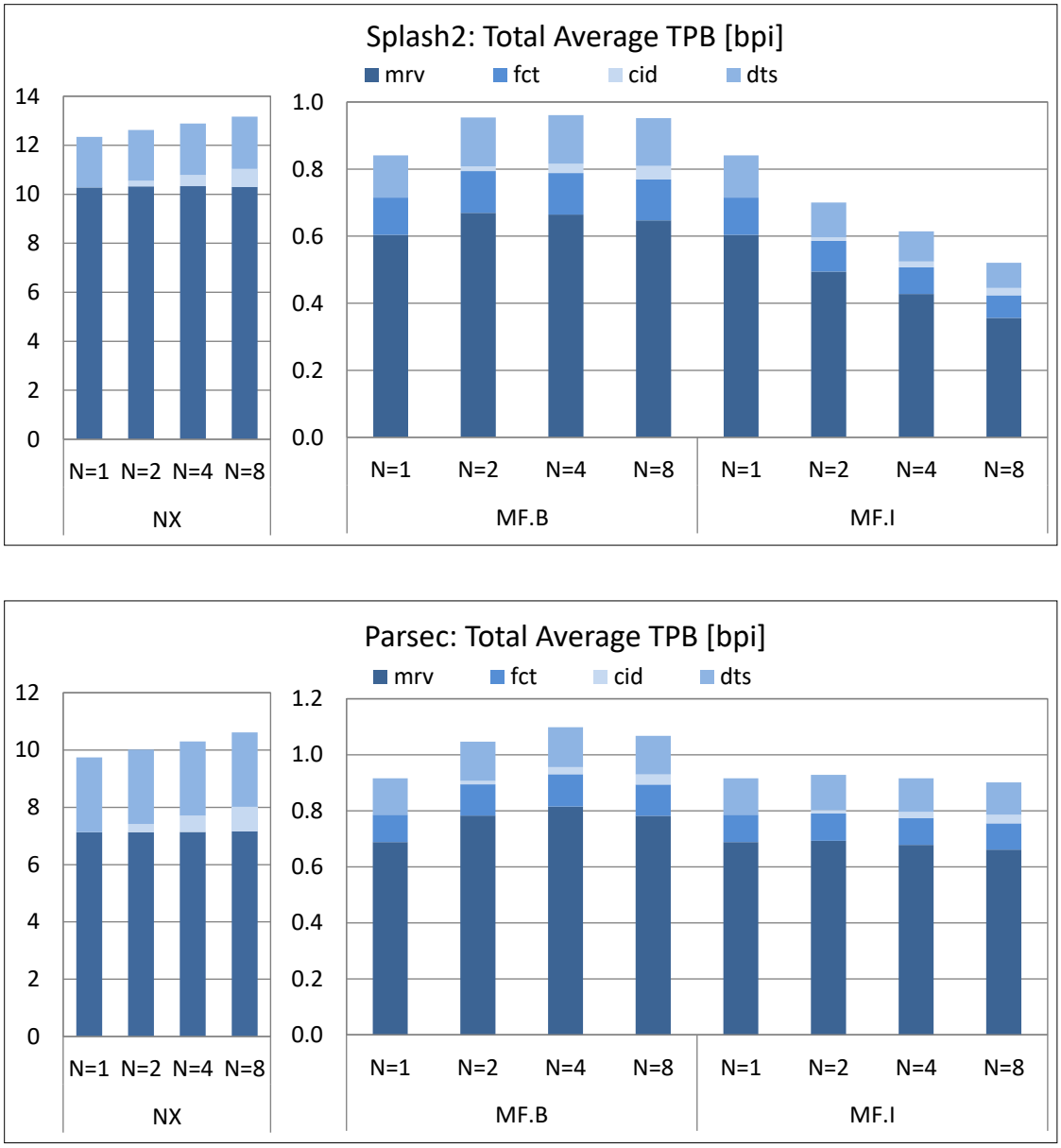


Figure 6.4 Break down of TPB in bpi for Splash2 (top) and Parsec (bottom) for CS64

6.3 Trace Port Bandwidth in BPC

Figure 6.5 shows the total average TPB in bpc for both Splash2 and Parsec. For Splash2, NX with CS16 requires a total average TPB of 4.92 bpc and it ranges from 2.79 bpc (*fft*) to 7.53 bpc (*raytrace*) when N=1. The TPB in bpc for individual benchmarks and Splash2 and Parsec for different cache configurations are available in Appendix A.2. As

the cache size increases the trace port bandwidth increases because of reduced execution time and increased pressure on the trace port (the same amount of trace data needs to be taken off of the chip in a shorter period of time). Thus, with CS64 the average TPB increases to 5.65 bpc. As the number of cores increases, the average TPB increases due to increased overall IPC and reduced execution time. When N=8, the average TPB ranges from 25.64 bpc (CS16) to 26.14 bpc (CS64). When we consider individual benchmarks, more than half of the benchmarks from Splash2 require more than the total average TPB and *raytrace* reaches as high as 46.47 bpc when N=8 with CS64. With MF.B, the average TPB ranges from 0.09 bpc (CS64) to 0.31 bpc (CS16) when N=1 and 1.05 bpc (CS64) to 1.71 bpc (CS16). However, with MF.I the average TPB is reduced due to inheriting FA bits and it ranges from 0.35 bpc (CS64) to 0.93 bpc (CS16). With *mcFiltrate*, the average TPB decreases as cache size increases because of the reduced FA miss rates. Even the worst-case benchmark requires an average TPB that is less than ~4 bpc for MF.B/I.

For Parsec, *mcFiltrate* with CS16 reduces the total average TPB from 3.12 bpc in NX to 0.23 bpc when N=1 and from 11.56 bpc in NX to 0.95 bpc (MF.B) and to 0.82 bpc (MF.I) when N=8. MF.I with CS64 requires a total average TPB of 0.59 bpc (N=8). These results confirm that the *mcFiltrate* reduces the pressure on the trace port and scales well with increasing the number of processor cores.

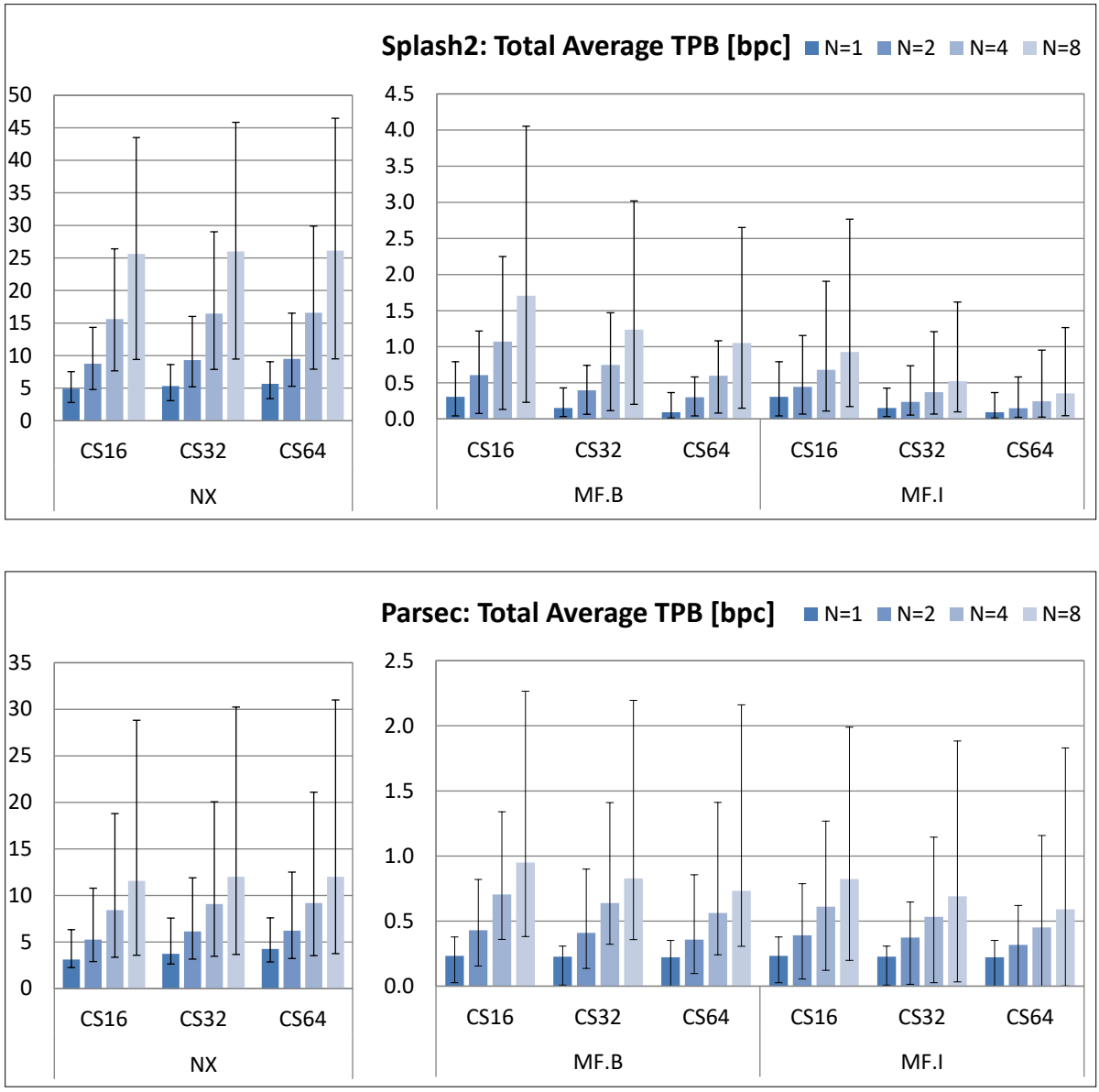


Figure 6.5 Trace Port Bandwidth in bpc for Splash2 (top) and Parsec (bottom)

6.4 Dynamic Trace Port Bandwidth Analysis

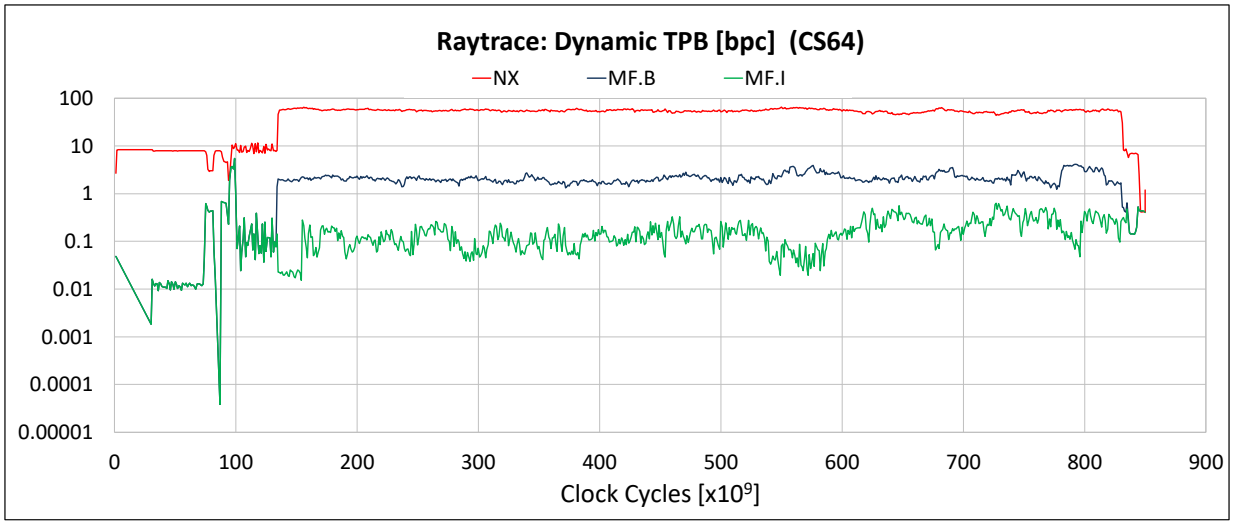
The average TPB allows us to quantify the overall effectiveness of *mcFiltrate*. However, it does not fully capture the peak TPB requirements that occur in individual benchmarks during their execution. The TPB depends on the frequency and distribution of memory reads and FA misses. Thus, the bandwidth requirements may exceed the average TPB at any given moment during program execution.

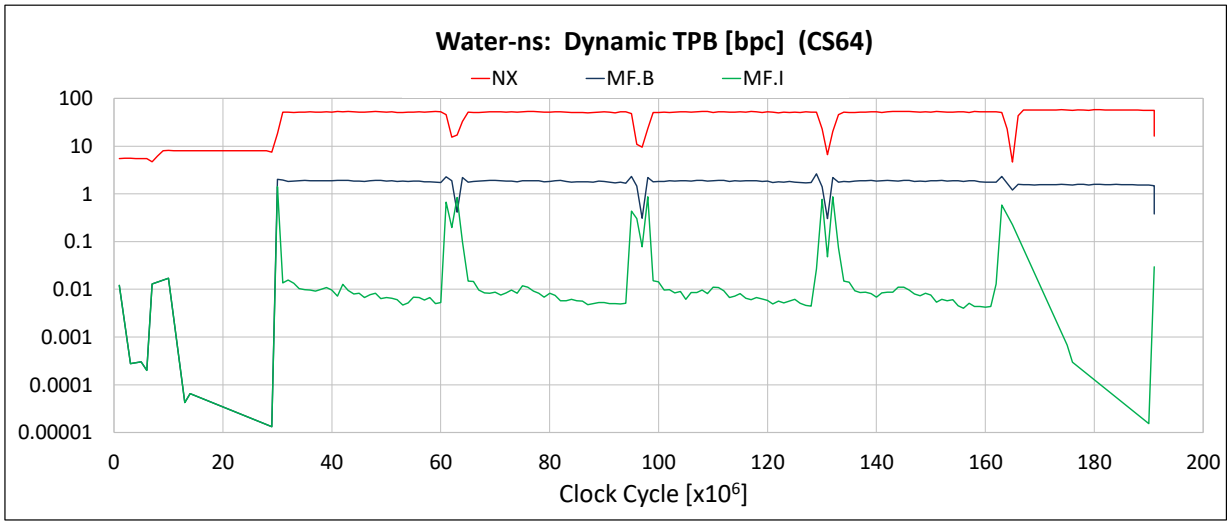
We analyze the required TPB for NX, MF.B, and MF.I with all the cache configurations (CS16, CS32, and CS64) and core configurations (N=1, 2, 4, and 8) for all the individual benchmarks from Splash2 and Parsec. For this analysis, the average TPB in bpc is logged every 1 million clock cycles. Figure 6.6 shows the dynamic TPB requirements for several characteristic benchmarks when N=8 with the CS64 configuration. The selected benchmarks, *raytrace*, *water-ns*, and *swaptions*, require the highest average TPB for the NX traces among all the benchmarks from both suites. In addition, the benchmarks *cholesky* and *fft* are considered because of their unique dynamic behavior. The average TPB required for NX with CS64 for the worst-case benchmarks, *raytrace* and *water-ns* from Splash2 and *swaptions* from Parsec is 46.47 bpc, 43.43 bpc, and 30.99 bpc, respectively. However, the peak TPB for *raytrace*, *water-ns*, and *swaptions* reaches as high as 64.8 bpc, 57.84 bpc, and 34.34 bpc respectively. Thus, the on-chip trace buffer requirements to capture the traces with no loss exceeds those implied by the average total TPB.

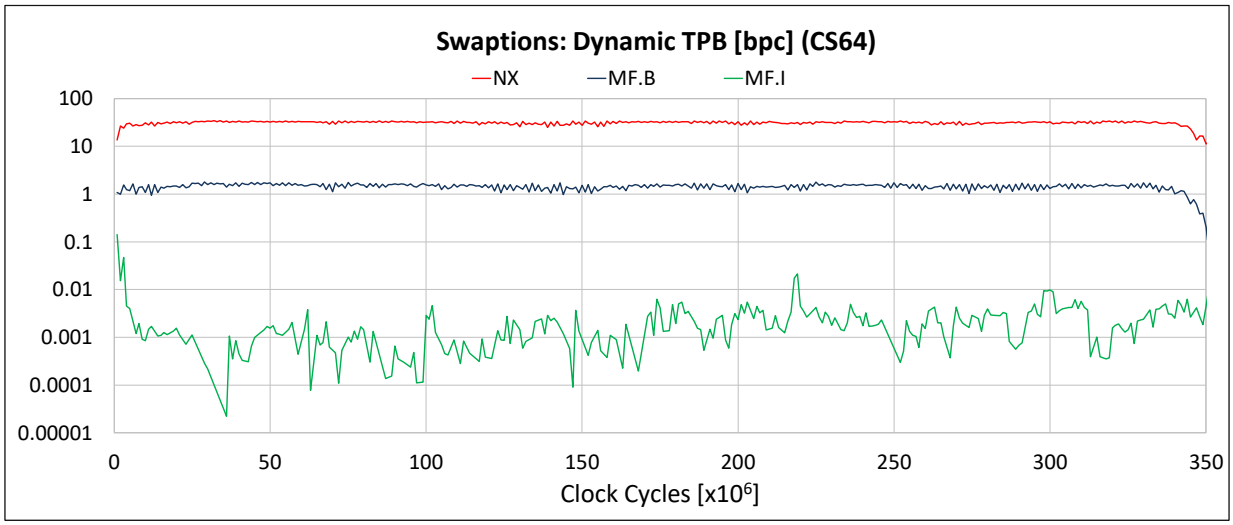
For *raytrace*, MF.B and MF.I require average TPBs of 1.81 bpc and 0.18 bpc, respectively. The peak TPB requirements reach 5.97 bpc (MF.B) and 5.45 bpc (MF.I). For *water-ns* and *swaptions* with MF.I the peak TPB reaches 1.41 bpc and 0.16 bpc, respectively. The results confirm that *mcFiltrate* not only reduces the average TPB, but it also reduces the peak TPB requirements.

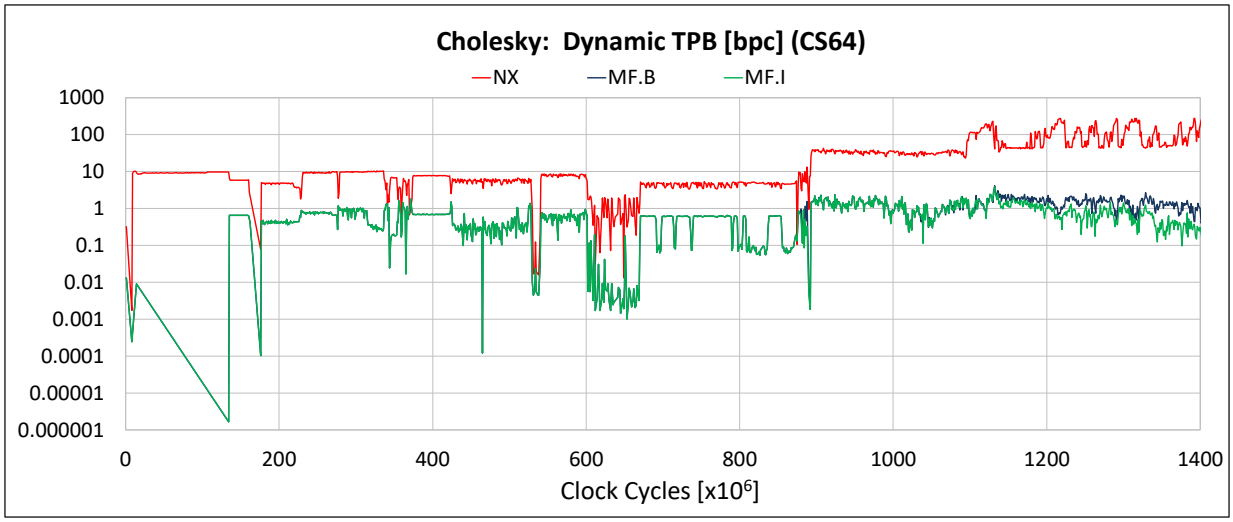
The analysis of benchmark execution reveals its behavior and helps make decisions on size of on-chip-trace buffers that can hold the trace data in the worst case. However, the deep trace buffer requirements do not have to come from the benchmarks that have the highest average TPB. Any benchmark that has a burst of data reads in a short period of time may present the worst-case scenario for data tracing. For example, *cholesky* requires an average TPB of 33.06 bpc. However, the peak bandwidth reaches 278.8 bpc with CS64 (Figure 6.6) because of the accumulation of memory reads towards the end of

the program. *mcFiltrate* effectively works in this case and reduces the required peak TPB to 4.19 bpc and 3.9 bpc with MF.B and MF.I respectively. *fft* shows a unique behavior among all the benchmarks (Figure 6.6). Even though the required TPB is moderate in the beginning of the program execution it reaches the peak of 45.9 bpc, due to the burst of memory reads towards the end of the program. This effect is not observed by analyzing the average TPB because it distributes the trace data evenly over the entire execution of the program. *mcFiltrate* works effectively and no trace message is emitted in the beginning of the program but due to the skewed memory reads in the end, the peak TPB reaches 6.05 bpc with MF.I.


Raytrace: Dynamic TPB [bpc] (CS64)
NX
MF.B
MF.I
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0
100
200
300
400
500
600
700
800
900
Clock Cycles [x10^9]


Water-ns: Dynamic TPB [bpc] (CS64)
NX
MF.B
MF.I
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0
20
40
60
80
100
120
140
160
180
200
Clock Cycle [x10^6]


Swaptions: Dynamic TPB [bpc] (CS64)
NX
MF.B
MF.I
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0
50
100
150
200
250
300
350
Clock Cycles [x10^6]



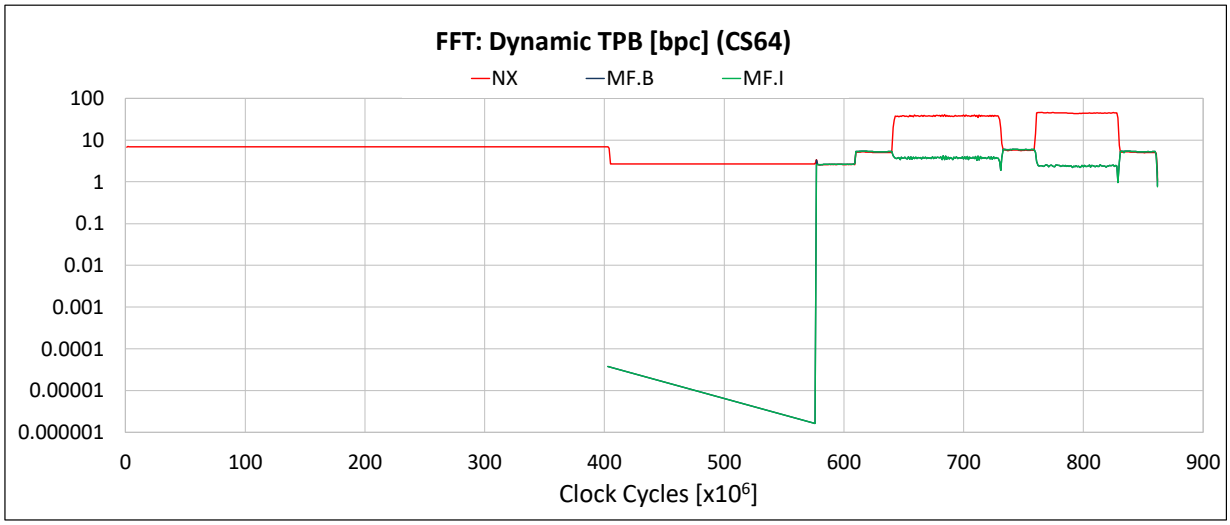


Figure 6.6 Dynamic TPB in bpc for Characteristic Benchmarks

6.5 Trace Buffer Size Analysis

Trace messages are temporarily stored in on-chip trace buffers before they are emitted off-chip through the trace port over dedicated physical pins. Even though dynamic TPB analysis gives deeper insights in trace port bandwidth requirements, it does not fully reveal the information about the required size of on-chip-trace buffers and their emptying rate. Assume that a program is generating 10 bits per clock cycle, but the trace port can trace out only 2 bits per clock cycle. The internal trace buffers will get filled

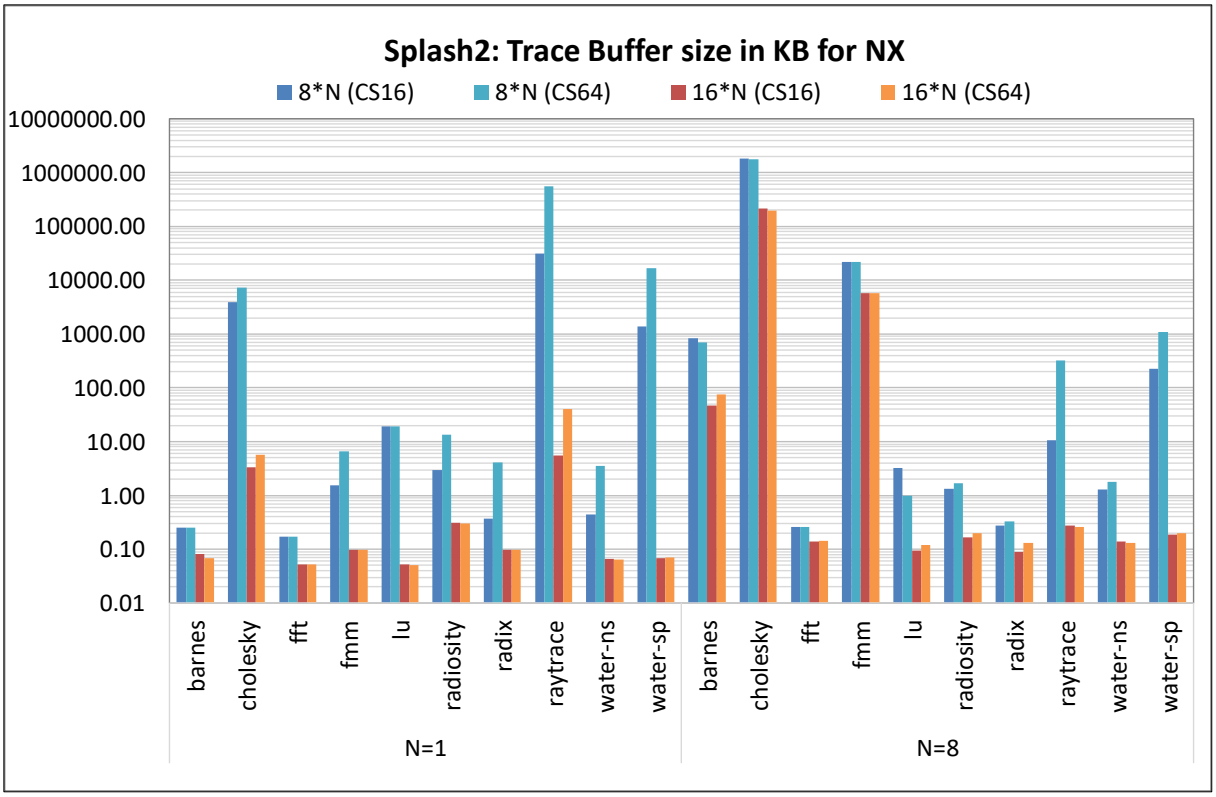
quickly resulting in lost trace messages. However, if we empty the trace data at the rate of 10 bpc, the required on-chip trace buffer size is moderate. Thus, to evaluate the requirements of on-chip trace buffer, we use the metric actual TPB which represents the ability of the trace module to send the trace data off the chip through the trace port. By changing the actual TPB we evaluate the on-chip trace buffer sizes needed to capture the traces for the entire program without stalling the processor to empty the trace data or without losing the trace data.

Figure 6.7 shows the maximum on-chip trace buffer size required for Splash2 and Parsec for NX tracing, while varying the actual TPB as N•8 and N•16, where N is the number of processor cores. Although allocating this many port pins to the trace port is cost-prohibitive, the purpose of this exercise is to emphasize challenges in on-the-fly tracing using the state-of-the-art.

With the actual TPB of 8 bpc when N=1, most of the benchmarks from Splash2 require less than 20 KB of on-chip trace buffer. However, the worst-case benchmark, *raytrace,* requires 543.9 MB with CS64, which is cost-prohibitive. In most of the benchmarks, as the cache size increases, the required trace-buffer size increases because of the reduced execution time of the benchmarks. When the actual TPB is increased to 16 bpc, the required trace buffer size is 0.35 KB for most of the benchmarks except *raytrace* which requires ~41 KB. When N=8 with the actual TPB of 64 bpc, *cholesky* requires 1.75 GB with CS16 and the other benchmarks from Splash2 require about 0.3 KB to 21 MB. When the actual TPB is increased to 128 bpc, *cholesky* requires 226 MB and all other benchmarks require about 0.3 KB. Similar trends are observed for Parsec. With the actual TPB of N•16 bpc, the required trace buffer is ~1.7 KB when N=1 and about 0.5 KB when N=8. The worst-case *fluidanimate* requires ~191 KB with N•16 bpc when N=8. Except for a few outliers – *cholesky*, *fmm*, *fluidanimate*, all the other benchmarks require a few KBs of on-

chip-trace buffer with the actual TPB of either N•8 or N•16. However, clocks to transmit the data off-chip runs way slower than the processor clock. Thus, it requires more physical pins than the actual TPB which further increases the system cost. This analysis shows that even with almost unlimited actual trace port bandwidth, the on-chip trace buffers may still be relatively large.

Figure 6.8 and Figure 6.9 show the on-chip trace buffer requirements while varying the actual TPB from N to 8•N with MF.B and MF.I for Splash2 and Parsec, respectively. With an actual TPB of 1 bpc when N=1, all the benchmarks except *fft* with MF.B require less than 1.4 MB (*fft* with CS16 requires 33.3 MB). As the actual TPB increases to 8 bpc, the required trace buffer is less than 11 KB and most of the benchmarks require less than 1 KB in Splash2 and less than 2 KB in Parsec. As the number of cores increases, the required trace buffer size decreases because of the wider trace port (except for *fft*). When N=8, with the actual TPB of 8 bpc, all benchmarks from Splash2 and Parsec except *fft* require the trace buffer sizes from 0.1 KB to 89 KB, whereas *fft* with CS16 requires 45.44 MB. However, when the actual TPB is increased to 64 bpc, the required trace buffer size is less than ~ 0.1 KB. Overall, when we allocate the actual TPB of 4 bpc for N=1 and N=2, 8 bpc for N=4 and N=8, all the benchmarks except *cholesky* with CS16 (45.45 MB) require a trace buffer less than ~120 KB. As we pointed out before, the benefits of MF.I over MF.B vary across benchmarks. Thus, for some benchmarks, MF.I helps to reduce the trace buffer size moderately (1% to 20%) and for some other benchmarks it reduces the maximum trace buffer size from 20% to 95% depending on the number of cores, cache configuration, and actual TPB.



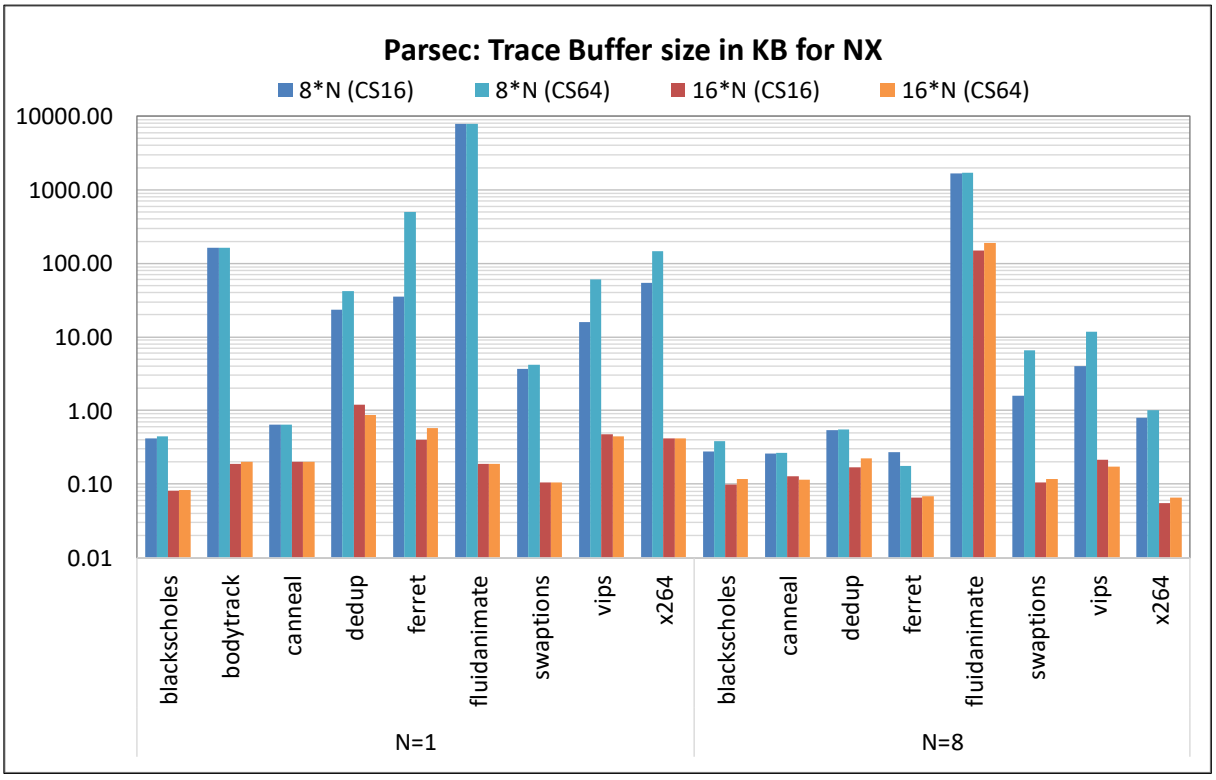


Figure 6.7 On-chip Trace Buffer Size for Splash2 (top) and Parsec (bottom) for NX

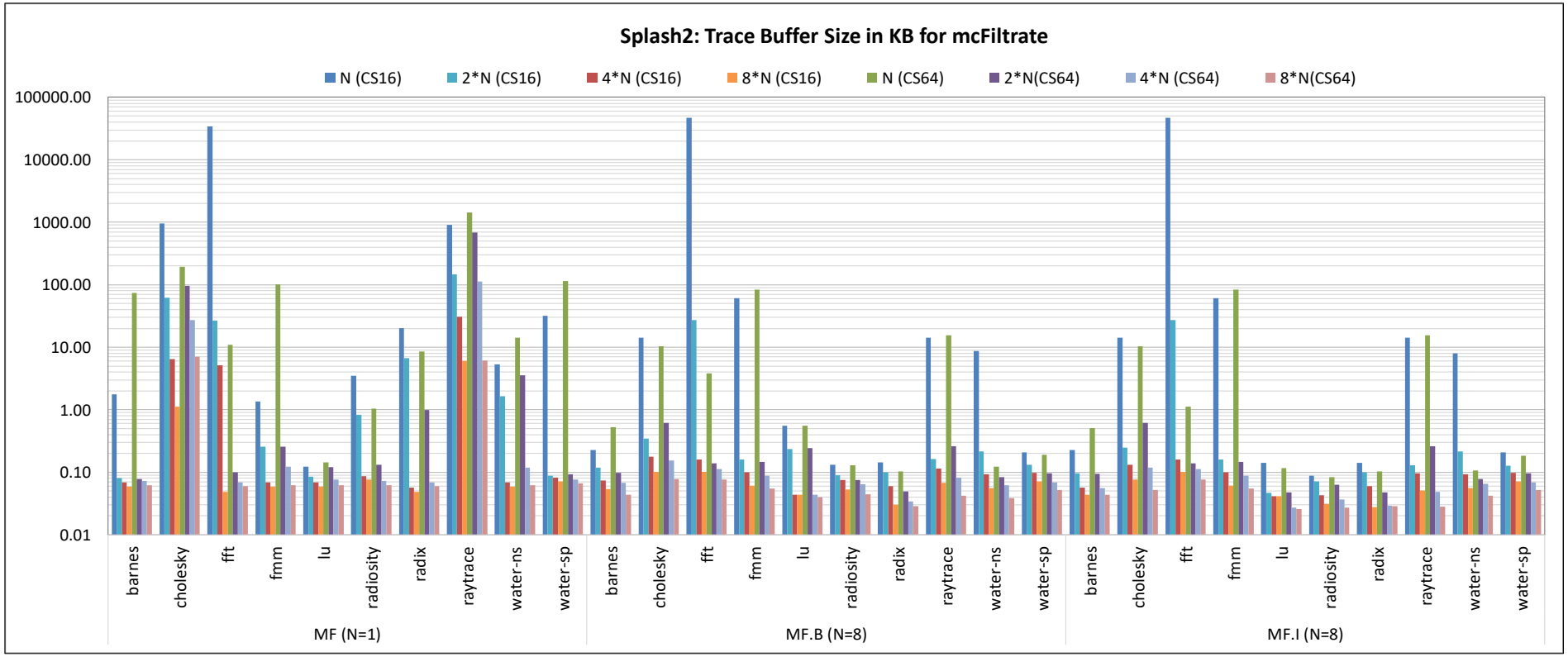


Figure 6.8 On-chip Trace Buffer Size in KB for Splash2

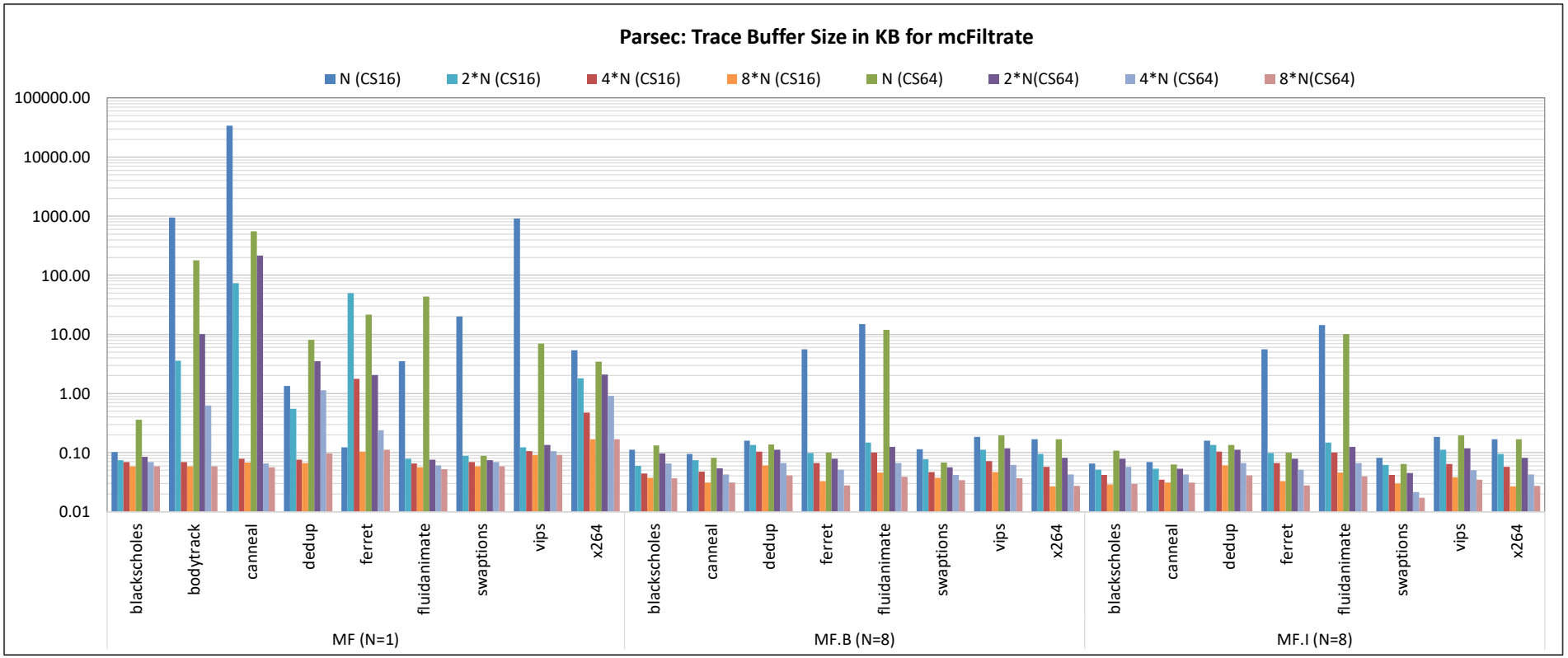


Figure 6.9 On-chip Trace Buffer Size in KB for Parsec

6.6 Hardware Complexity Analysis

mcFiltrate requires hardware extensions to support the first-access tracking in L1 data caches for all processor cores. The majority of hardware overhead is due to the FA bits. In the case of inheriting FA bits to further filter the shared data in multicores, additional hardware support is required. The overall overhead depends on the granularity of the FA bits and their location, data cache size, and block size. The overhead due to the control logic for the FA bits and trace encoding is negligible.

The size of the sub-block which is being protected by a single FA bit is called the granularity size (G). The hardware complexity of *mcFiltrate* primarily depends on granularity size as shown in Eq. (6.4). For example, consider a 16 KB L1 data cache with 32-byte cache block size. For the granularity of size 4 (i.e. 4-bytes), a single cache block requires 8 FA bits and all the cache blocks require 512 bytes which is 1/32 of the data cache size. However, for the granularity of 1 (1-byte), it requires 32 FA bits and overall overhead is 1/8 of the cache size, i.e. 2 KB.

$$FA\ Storage\ Size = \frac{Cache\ size\ (CS)\ in\ bytes}{Granularity\ Size\ (G)\ in\ bytes * 8}\ bytes \tag{6.4}$$

If the data cache physical design can be changed then, the first-access tracking can be attached to the data cache blocks and control logic is added to maintain them (Figure 4.1). If we assume a processor core with 32 KB data cache, 32-byte cache blocks, and FA bit granularity of 4 bytes, then the overhead is 1/32nd of the data cache capacity, or 1 KB of additional storage. With finer granularity, i.e. when each byte is protected with FA bit, the size of trace messages can be reduced if the byte sized memory reads dominate. However, the required on-chip area increases. On the other hand, with a coarse-grain granularity, every FA miss event results in

reporting the entire sub-block regardless of the size of the memory read. In cases of poor spatial locality, coarse-grain granularity may have negative effects on the total size of trace messages. However, it can also contribute to reducing the number of trace messages in cases of strong spatial locality. For example, when short operands are accessed sequentially, the number of bits needed to report the *dts*, *cid*, and *fnt* is reduced with coarse-grain granularity.

If the physical design of the data cache cannot be changed, alternatively, the FA bits can be implemented outside of processor cores in trace modules and connected to processor cores through a well-defined interface. In this case, *mcFiltrate* would need to include cache tags and an address decoding unit, which introduces additional hardware overhead. However, this approach may offer higher modularity and flexibility because trace modules do not have to mirror actual processor data caches.

The hardware complexity is estimated using Cacti tools [98] that report the area occupied by the tag and data portions of cache structures. Table 6.7 shows the area required by each original L1 data caches (columns 2-4), the area required by the data cache when the FA bits are attached to the cache block (columns 5-7), and the area required by the data cache when the FA bits are external (column 9). When the FA bits are attached to the cache block, the total overhead ranges from ~0.5 to 4.2% (column 8) of the regular L1 data cache area. However, duplicating cache tags results in an increased overhead that is 12.1% to 13.5% (column 10) of the total L1 data cache area. In our analysis, we assume that the first-access bits are tied to the L1 data cache and cache to cache transferring of FA bits. Please note that hardware support is required to inherit FA bits from a source cache if the cache coherence protocol states are utilized to reduce the trace port bandwidth. One way to copy FA bits is, issuing an extra bus transaction. However, this approach may add additional bus

traffic. Another way is to add extra data lines connecting each cache for cache-to-cache transfer of FA bits.

Table 6.7 Hardware Complexity Estimation

Cache Size	Base Cache Area (μm²)			FA Bits Tied to the Data Cache (μm²)				External FA bits (μm²)	
	Tag Area	Data Area	Total Area	Tag + FA Bits Area	Data Area	Total Area	Over-head	Tag + FA Bits Area	Over-head
16KB	16,693	154,916	171,609	23,179	154,916	178,094	1.038	23,179	1.135
32KB	22,105	215,620	237,725	32,130	215,620	247,750	1.042	32,130	1.135
64KB	44,130	336,040	380,170	45,998	336,040	382,038	1.005	45,998	1.121

CHAPTER 7

DICTIONARY ANALYSIS

This chapter considers further improvements to data tracing by exploiting redundancy in data values read from memory. It applies to both NX and mcFiltrate tracing techniques. As shown in CHAPTER 6 (Figure 6.4), the most significant portion of the required trace port bandwidth, ~60%-80% of the total, is used for reporting data values read from memory (the *mrv* field). Thus, eliminating redundant data values in trace messages is the most beneficial approach in a quest to further reduce the required TPB. Towards this end, this chapter explores how to augment tracing techniques with statically selected dictionaries or dynamic dictionaries or a combination of both (hybrid dictionaries).

Section 7.1 discusses the details of how dictionaries can be used to filter the trace messages and Section 7.2 discusses the operation of *mcFiltrate* when dictionaries are used. Section 7.3 discusses the experimental details and Section 7.4 discusses the results from the experimental evaluation.

7.1 Preliminaries

Dictionaries keep the most frequent data values read from memory. A dictionary contains a certain number of entries (DS – dictionary size) of given length (DES – dictionary entry size) as shown in Figure 7.1. When a trace message with a data value is ready, a dictionary lookup is performed instead of sending it to the trace port. If the dictionary contains a entry matching the data field in the trace message (a *dictionary hit*), a shorter trace message is going to be created as follows.

The data field in the trace message is replaced by an index of the dictionary entry (*dIn*) that matches the data value from the trace message. This way long data fields encoding an operand value are replaced by a few bits encoding an entry in the dictionary. If both the target platform trace module and the software debugger maintain synchronized dictionaries, the software debugger can recreate the operand values, even though they are not sent through the trace port. If the data value is not found in the dictionary, i.e., a *dictionary miss* occurs, the original data value is emitted.

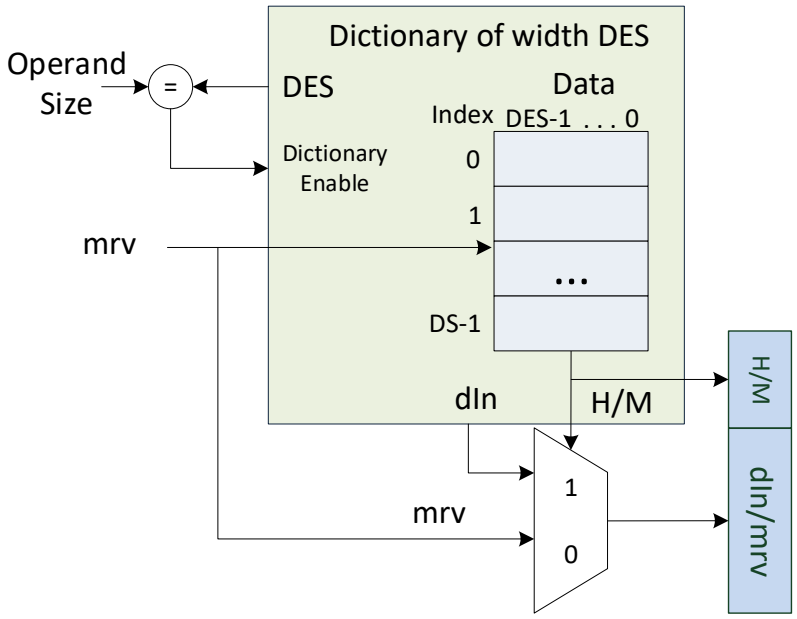


Figure 7.1 System View of a Dictionary-Based Trace Compressor

Dictionary compression can be applied to both Nexus-like and *mcFiltrate* trace messages. Figure 7.2 shows the format of the corresponding trace messages that support dictionary compression. The *dts*, *cid*, *fct* fields are encoded as described in Figure 4.5. To indicate whether the modified data field in the trace message con-

tains an index in the dictionary or the data value, a data header bit called H/M is needed (1 indicates a dictionary hit, 0 indicates a dictionary miss). Thus, the size of the data field is $1+\lceil log_2(DS) \rceil$ in the case of a dictionary hit, or 1+*sizeof(operand)* in the case of a dictionary miss.

A dictionary is implemented in hardware using content-addressable memories (CAM). The CAM memories are somewhat costly, so dictionaries should have a limited number of entries to reduce the cost of their implementation and the length of the trace messages. Yet, they should be large enough so that the number of dictionary hits is sufficiently large to outweigh overhead due to the additional data header bit in the trace message.

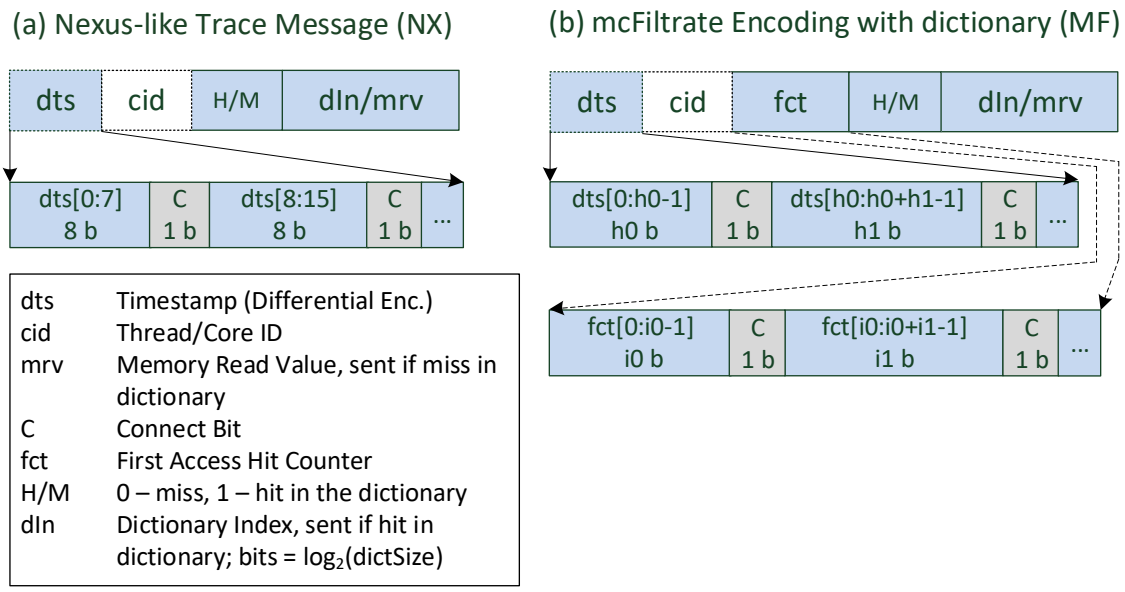


Figure 7.2 Format of Trace Messages Supporting Dictionaries

A dictionary can be initialized statically or it can be updated dynamically. A statically initialized directory contains predefined most frequently used operand values of a given size (DES). These values can be either known constants or known

contents of memory locations in the program or can be extracted by profiling the actual program. In the case of a static dictionary, the trace module controller should enable initialization of the directory entries for a given program before the tracing starts.

A dynamic dictionary is not initialized in advance, rather it is updated continuously based on an update policy. Initially, the dictionary is empty and is filled with data values from trace messages. Both the hardware component in the trace module of the target platform and its software counterpart in the software debugger should use identical policies when updating the dictionary for both hit and miss events (e.g., round-robin or LRU replacement policy).

Finally, a combination of a static and a dynamic dictionary can also be used. In this case, the most frequently used values, e.g., constants like π, 0, or 1 are stored in the static dictionary and other values are updated in the dynamic dictionary. Such a scheme requires more than one data header bit to indicate whether the data is a hit in the static or dynamic dictionary. By design, hybrid dictionaries can be of two types. One method uses separate static and dynamic dictionaries and the 2-bit H/M indicates whether data is a hit in static dictionary or dynamic dictionary or missed in both as described in Table 7.1. The other method is to incorporate the static dictionary in the H/M field as described in Table 7.2. In the latter method, the size of the data header field is adjusted depending on the number of entries in the static dictionary.

Table 7.1 Hybrid Dictionary Data Header Encoding (Method 1): An Example

H/M Value	Event
00	Miss followed by actual memory read data value
01	Hit in the static dictionary followed by index
10	Hit in the dynamic dictionary followed by index
11	Unused

Table 7.2 Hybrid Dictionary Data Header Encoding (Method 2): An Example

H/M Value	Event
00	Miss followed by actual memory read data value
01	Constant 0
10	Constant 1
11	Hit in dynamic dictionary followed by index

7.2 Operation of *mcFiltrate* with Dictionaries

Figure 7.3 shows the memory read operation of *mcFiltrate* on target core *i* when dictionaries are enabled. Light purple colored boxes show the additional steps required for dictionaries compared to the original memory read operation with baseline *mcFiltrate*. In the case of an *FA miss event*, a lookup in the dictionary is performed if the dictionary is available for current operand size (steps 6 and 9) before emitting the trace message. If the dictionary is not available or the data value is not found in the dictionary, a *dictionary miss event* occurs and a trace message is reported. The trace message includes the time stamp (*dts*), the core identifier (*cid*), the first-access hit counter (*fct*), *H/M* set to 0, and the memory read value (*mrv*) (step 10). If the value is found in the dictionary, a *dictionary hit event* occurs, the index of the matching dictionary entry (*dIn*) is reported and the H/M is set to 1 (step 11). FA

bits are not set in the case of dictionary hit events because the value of full sub-block(s) may not be reported when the memory reads are not aligned and when the operand size is less than the granularity size. Please note that the dictionary holds the actual values of the operands instead of values of the sub-blocks. FA bits for sub-blocks corresponding to a memory read operation are verified independently. In the case of an *FA miss event*, the *mrv* field may include the data from a single sub-block or multiple sub-blocks for which FA bits are not set. Thus, reported sub-blocks may not be contiguous. If the dictionary is enabled to store values of sub-blocks instead of original memory read data value, multiple dictionary indices may need to be reported in a single trace message. Hence, dictionaries are used for actual memory read values with the sizes matching the original data sizes. *mcFiltrate* with dictionaries does not require any operation changes for memory writes and the flowchart in Figure 4.3 is used.

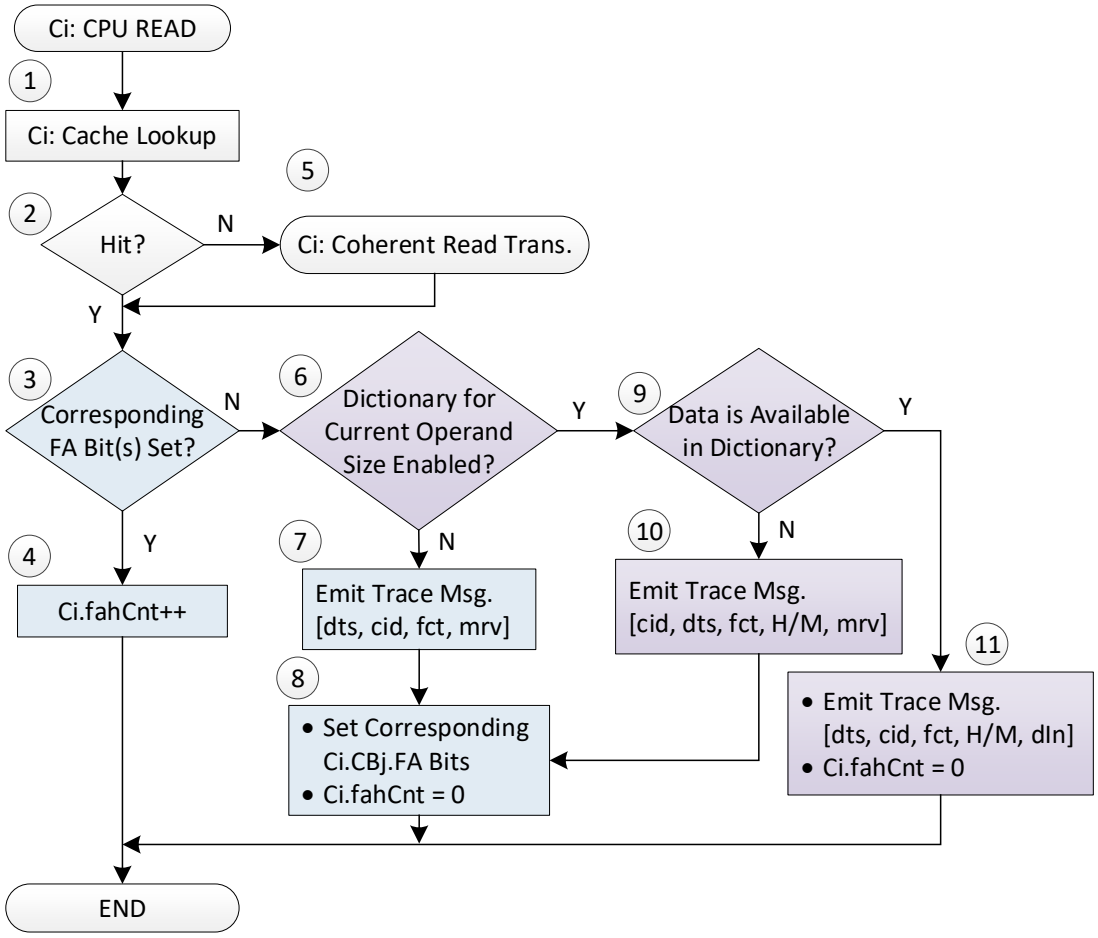


Figure 7.3 *mcFiltrate* Operation with Dictionary on Target Platform Core *i* for Memory Read

Figure 7.4 shows the operation of *mcFiltrate* with dictionaries on the software debugger side for memory reads. These steps are similar to the steps described in Figure 4.4 except when the first-access hit counter is zero (*Ci.fahCnt=0*). If the dictionary is enabled for current memory read operation operand size, the H/M bit of the trace message is checked to determine whether the data following the H/M field is a dictionary index or the actual data value (step 2). In the case of a dictionary hit event (H/M=1), the index (*dIn*) is read from the trace message and the actual value of memory read is fetched from the dictionary (steps 3 and 5). Otherwise (H/M=0),

the actual data value (*mrv*) is read from the trace message (step 4). The software cache is updated with the new value and the next trace message is read. The memory writes and invalidation operations are the same as in Figure 4.4.

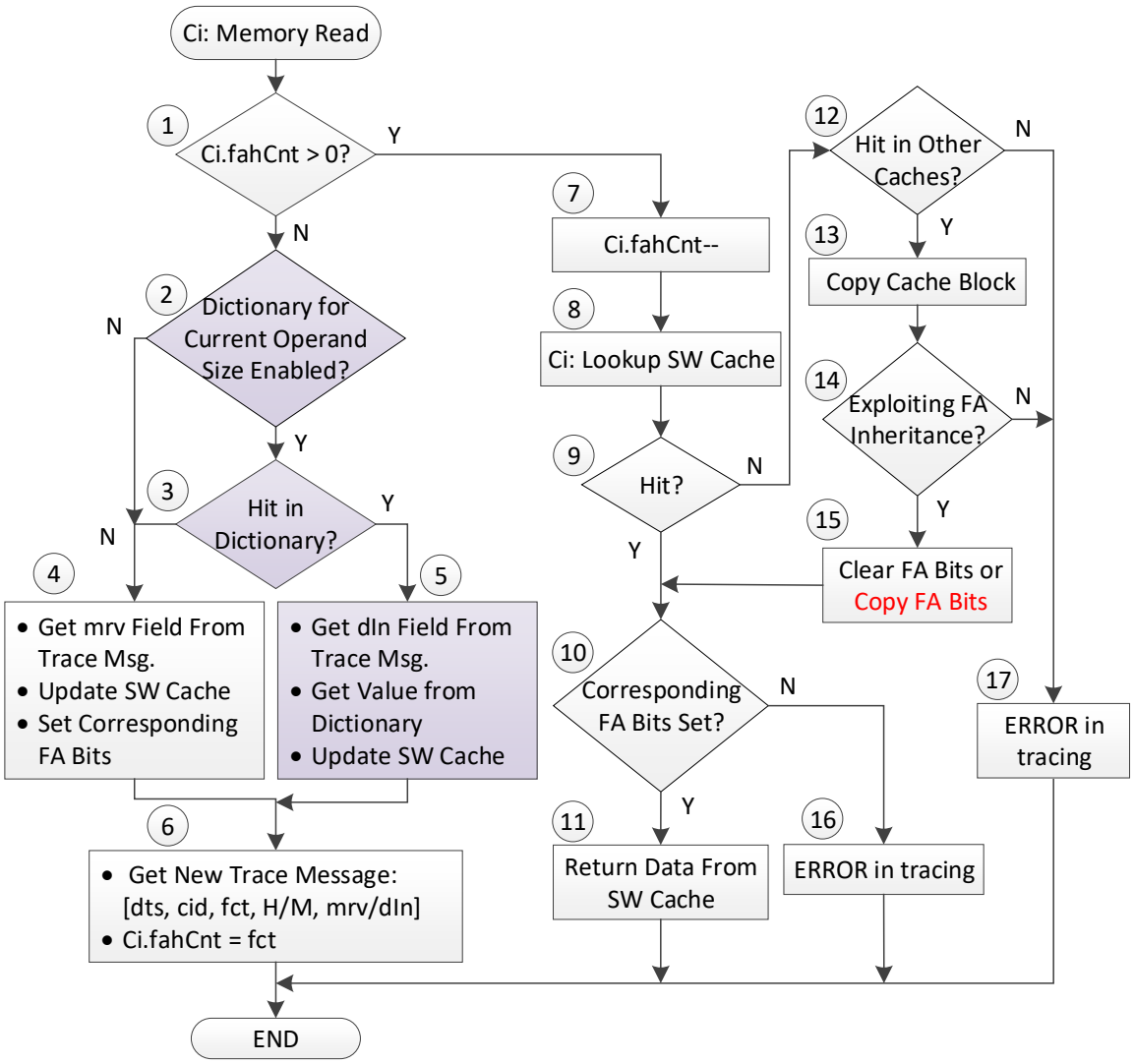


Figure 7.4 *mcFiltrate* Operation with Dictionary on Software Debugger for Memory Reads

7.3 Experimental Evaluation

We evaluate the ability of static and dynamic dictionaries to compress the data value field in trace messages, while varying the cache configurations, *mcFiltrate* configurations, and the size of the dictionary. In static dictionary analysis, dictionaries are filled with the most frequently used data extracted from the trace files produced by *tmTrace*. In dynamic dictionary analysis, the LRU replacement policy is used to replace the data in the case of a dictionary miss.

For simplicity, a dictionary lookup or update is performed only when the size of the memory read operand matches the size of the dictionary entry (DES). Thus, even if a 4-byte (e.g., type integer) and an 8-byte (e.g., type long integer) operand data values are the same, they are not stored in the same dictionary. The number of entries in a dictionary (DS) is varied and we explore 4, 8, 16, 32, 64, 128, and 256 entries. At any given time, only one dictionary is active and thus, each data type is analyzed independently. When the dictionary is enabled for the given operand size, it uses the trace format as shown in Figure 7.2, otherwise the trace format given in Figure 4.5 is used to avoid reporting the H/M bit when not required. Please note that a dictionary is global to all processor cores. Depending on the available resources, the sizes of the dictionary and the number of dictionaries which support different size of data can be adjusted.

The effectiveness of dictionaries is quantitatively measured by reporting the compression ratio as shown in Eq.(7.1). The compression ratio achieved with dictionary for NX and *mcFiltrate* is calculated by dividing the required TPB when the dictionaries are not enabled with the TPB when the dictionaries are enabled.

$$Compression\ Ratio\ = \frac{TPB\ without\ dictionaries}{TBP\ with\ either\ static\ or\ dynamic\ dictionary} \quad (7.1)$$

7.4 Results

This section discusses the compression ratios of NX and *mcFiltrate* traces when static and dynamic dictionaries are enabled.

7.4.1 Nexus-like (NX)

The frequency of memory reads of 1-byte and 2-byte operands is relatively low in Splash2 benchmarks. Thus, even if the dictionaries are used for these operand sizes, savings due to reporting the index instead of data value will be reduced because of the additional H/M bit. Thus, this section mainly focusses on evaluating the effectiveness of dictionaries in compressing data values when the size of a dictionary entry is 4-bytes (DES=4) and 8-bytes (DES=8).

Table 7.3 shows the compression ratios for Splash2 benchmarks (CS64) achieved by a 256-entry static dictionary (DS=256), with DES=4 and DES=8. Please note that the boxes shaded in green and orange color in the table represents the best-case and worst-case benchmarks respectively for a given column. However, sometimes only best-case benchmarks are marked when they are few to avoid the clutter created by marking worst-case benchmarks. The total average TPB is reduced by ~8% with DES=4 and ~12% with DES=8 when N=1. When considering individual benchmarks, the compression ratios when DES=4 are significant for *radiosity*, *radix*, *barnes*, and *raytrace* since they have a high frequency of 4-byte memory reads (see Table 5.3). With DES=8, *fmm*, *water-ns*, *water-sp*, and *raytrace* see relatively high compression ratios because they have a high frequency of 8-byte reads.

As the number of cores increases, the compression ratio may increase or decrease depending on the benchmark. For example, as the number of cores increases the compression ratio for *cholesky* increases, however, for *radiosity* it decreases. The reason for this could be the behavior of the benchmark. In the case of multithreaded programs, dictionaries may give good compression ratio when the amount of shared work is high but may affect the efficiency of the dictionaries when all the threads work independently. Please note that for simplicity results are shown for DS=256 though dictionary with 4 entries (DS=4) give reasonable compression ratios for most of the benchmarks. As the dictionary size increases, the compression ratios for some benchmarks decreases due to the additional index bits. In these cases, the level of redundancy that can be exploited by static dictionaries is not sufficient to compensate for longer index fields. On the other hand, the compression ratio increases for some benchmarks since the dictionary can hold more values. Similar trends are observed for NX with CS16 and CS32 cache configurations. When multiple dictionaries are used, the compression ratio will be higher (Table. A.21). However, to reduce the design and simulation space, results are shown only for a single dictionary.

Table 7.3 Compression Ratio for Splash2 with Static Dictionary (DS=256) for NX (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
barnes	1.13	1.13	1.13	1.12	1.09	1.09	1.09	1.09
cholesky	1.03	1.06	1.20	1.39	1.01	1.01	1.01	1.00
fft	1.08	1.08	1.08	1.08	1.04	1.04	1.04	1.04
fmm	1.01	1.01	1.02	1.03	1.43	1.42	1.40	1.38
lu	1.08	1.08	1.08	1.08	0.99	0.99	0.99	0.99
radiosity	1.22	1.21	1.20	1.18	1.05	1.04	1.04	1.04
radix	1.15	1.15	1.15	1.14	1.08	1.08	1.08	1.07
raytrace	1.10	1.09	1.09	1.09	1.12	1.11	1.11	1.11
water-ns	1.04	1.04	1.04	1.04	1.16	1.15	1.15	1.15
water-sp	1.03	1.03	1.03	1.03	1.16	1.16	1.16	1.15
Total	1.08	1.09	1.10	1.13	1.12	1.12	1.11	1.10

Table 7.4 shows the compression ratios achieved with the dynamic dictionary of size 256 entries (DS=256) for Splash2 benchmarks with CS64 when DES=4 and DES=8. For a single core, the total average TPB is reduced by 22% with DES=4 and 40% with DES=8. With DES=4 except *fmm*, *water-ns*, and *water-sp* and with DES=8, except *fft*, *radiosity*, and *raytrace* all other benchmarks achieve good compression ratio. When N=8, the average TPB is reduced by ~22% with DES=4 and ~24% with DES=8. However, when the dictionaries are enabled for both DES=4 and DES=8, compression ratio reaches as high as 12.93 for *swapations* when N=1 (Table. A.21). Overall, the compression ratios with dynamic dictionaries outperform the correspondingly ones with static dictionaries.

Table 7.4 Compression Ratio of Splash2 with Dynamic Dictionary (DS=256) for NX (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
barnes	1.31	1.30	1.29	1.28	1.15	1.15	1.15	1.15
cholesky	1.25	1.24	1.36	1.51	1.50	1.45	1.30	1.14
fft	1.17	1.17	1.16	1.16	1.07	1.06	1.06	1.06
fmm	1.05	1.04	1.04	1.04	2.10	2.04	1.90	1.48
lu	1.15	1.15	1.15	1.15	1.76	1.74	1.75	1.71
radiosity	1.27	1.25	1.25	1.25	1.07	1.07	1.07	1.07
radix	1.56	1.52	1.47	1.37	2.30	2.16	1.92	1.61
raytrace	1.44	1.45	1.40	1.35	1.07	1.08	1.07	1.06
water-ns	1.07	1.05	1.05	1.04	1.70	1.67	1.49	1.40
water-sp	1.07	1.05	1.05	1.04	1.69	1.65	1.55	1.44
Total	1.22	1.21	1.21	1.22	1.40	1.38	1.33	1.24

The compression ratios achieved with static and dynamic dictionaries for Parsec with DES=4 and DES=8 are showed in Table 7.5 and Table 7.6, respectively. The average TPB is reduced by 8% to 22% with static a dictionary and 44% to 52% with a dynamic dictionary when DES=4. Even though 1-byte operand size memory reads are dominating in most of the benchmarks (Table 5.4), the benefits with using the dictionary with DES=1 is not significant. The reason for this is the size of *mrv* field is 8-bits when the operand size is 1-byte. When dictionaries are enabled, an additional H/M bit is reported in every trace message and 2 index bits when DS=4 and 8 bits when DS=256 are reported in case of a hit in the dictionary. Hence, the number of bits saved with reporting dictionary index are not significant. On the other

hand, few benchmarks have memory reads dominated by 8-bytes, thus, few benchmarks benefit from an 8-byte data type dictionary (DES=8).

Table 7.5 Compression Ratio of Parsec with Static Dictionary (DS=128) for NX (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
blackscholes	1.26	1.25	1.24	1.24	1.12	1.12	1.12	1.12
bodytrack	1.24	1.23	1.22	-	1.00	1.00	1.00	-
Canneal	1.30	1.29	1.28	1.27	1.00	1.00	1.00	1.00
Dedup	1.16	1.13	1.12	1.11	1.00	1.00	1.00	1.00
Ferret	1.16	1.16	1.15	1.15	1.02	1.02	1.02	1.02
fluidanimate	1.30	1.29	1.28	1.28	1.00	1.00	1.00	1.00
swaptions	1.28	1.23	1.20	0.99	1.16	1.16	1.15	1.15
Vips	1.27	1.25	1.24	0.98	1.09	1.09	1.09	1.08
x264	1.17	1.16	0.99	0.99	1.00	1.00	1.00	1.00
Total	1.22	1.21	1.15	1.08	1.03	1.03	1.03	1.03

Table 7.6 Compression Ratio of Parsec with Dynamic Dictionary (DS=256) for NX (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
blackscholes	1.32	1.38	1.30	1.28	1.32	1.32	1.30	1.28
bodytrack	1.52	1.52	1.52	-	1.01	1.01	1.01	-
canneal	1.70	1.67	1.63	1.59	1.01	1.01	1.01	1.01
dedup	1.67	1.64	1.60	1.56	1.00	1.00	1.00	1.00
ferret	1.62	1.59	1.56	1.53	1.03	1.03	1.02	1.02
fluidanimate	2.13	2.04	1.97	1.90	1.00	1.00	1.00	1.00
swaptions	1.44	1.43	1.41	1.38	1.28	1.25	1.24	1.23
vips	1.46	1.44	1.42	1.39	1.08	1.07	1.07	1.07
x264	1.27	1.25	1.24	1.22	1.00	1.00	1.00	1.00
Total	1.52	1.50	1.47	1.44	1.03	1.03	1.03	1.03

7.4.2 *mcFiltrate*

Table 7.7 and Table 7.8 show the compression ratios for *mcFiltrate* with MF.I when using static and dynamic dictionary for Splash2, respectively. The total TPB is reduced from 1% to 6% with the static dictionary and 5% to 9% with dynamic dictionary. However, when we consider individual benchmarks, *fmm* TPB is reduced by ~25% with static dictionaries. With dynamic dictionaries, *radix* (DES=4) and *cholesky* (DES=8) achieve a better compression ratio (~19% to ~69%) compared to the other benchmarks. The compression ratios are in the same range for CS16 and CS32 with MF.I and for all cache configurations for *mcFiltrate* with MF.B.

Table 7.7 Compression Ratio of Splash2 with Static Dictionary (DS=256) for MF.I (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
barnes	1.02	1.02	1.02	1.03	1.07	1.02	1.00	1.01
cholesky	1.02	1.02	1.02	1.02	1.14	1.12	1.12	1.12
fft	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
fmm	1.02	1.02	1.02	1.02	1.25	1.24	1.24	1.23
lu	1.00	1.00	1.00	1.01	0.99	0.99	0.99	1.01
radiosity	0.86	0.84	0.85	0.86	1.00	1.00	1.00	1.00
radix	1.06	1.04	1.04	1.03	1.03	1.02	1.01	1.01
raytrace	1.07	1.07	1.09	1.10	1.00	1.00	1.00	1.00
water-ns	1.02	1.03	1.04	1.03	1.16	1.13	1.10	1.09
water-sp	1.01	1.01	1.01	1.01	1.10	1.10	1.10	1.10
Total	1.02	1.02	1.01	1.01	1.06	1.05	1.05	1.05

Table 7.9 and Table 7.10 show the compression ratio for Parsec benchmarks for CS64 with static and dynamic dictionary, respectively. The average TPB is reduced by 3% to 14% depending on the dictionary type, dictionary entry size (DES), and number of cores. When individual benchmarks are considered, *x264* and *bodytrack* with DES=1, *dedup* with DES=2, and *blackscholes* with DES=4 benefit the most.

Table 7.8 Compression Ratio of Splash2 with Dynamic Dictionary (DS=256) for MF.I (CS64)

Dictionary Entry Size in Bytes	DES=4				DES=8			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
barnes	1.00	1.00	1.00	1.01	0.99	0.99	0.99	1.00
cholesky	1.07	1.09	1.10	1.11	1.69	1.61	1.54	1.47
fft	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
fmm	1.02	1.02	1.02	1.02	1.31	1.28	1.27	1.25
lu	1.00	1.00	1.00	1.00	0.99	0.99	1.29	1.29
radiosity	1.30	1.66	1.79	2.63	1.00	1.00	1.00	1.00
radix	1.46	1.31	1.25	1.19	1.12	1.07	1.05	1.04
raytrace	1.09	1.05	1.04	1.03	1.00	1.00	1.00	1.00
water-ns	1.00	1.00	1.00	1.00	1.13	1.13	1.13	1.10
water-sp	1.01	1.00	1.01	1.01	1.13	1.12	1.12	1.12
Total	1.06	1.05	1.06	1.06	1.09	1.08	1.08	1.08

Table 7.9 Compression Ratio of Parsec with Static Dictionary (DS=256) for MF.I (CS64)

Dictionary Entry Size in Bytes	DES=1				DES=2				DES=4			
Benchmark/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
blackscholes	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.12	2.08	2.03	1.97
bodytrack	1.55	1.49	1.43	-	1.00	1.00	1.00	-	1.04	1.03	1.03	-
canneal	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.20	1.19	1.19	1.18
dedup	1.17	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.12	1.13	1.13	1.13
ferret	1.04	1.04	1.04	1.03	1.01	1.01	1.01	1.01	1.09	1.10	1.09	1.09
fluidanimate	1.00	1.01	1.01	1.01	1.00	1.00	1.00	1.00	1.18	1.18	1.18	1.19
swaptions	1.20	1.28	1.33	0.95	1.14	1.21	1.26	0.94	1.45	1.57	1.67	1.04
vips	1.12	1.12	1.12	1.12	1.02	1.02	1.02	1.02	1.01	1.01	1.01	1.01
x264	1.66	1.59	1.61	1.60	1.02	1.03	1.02	1.02	1.03	1.04	1.03	1.03
Total	1.14	1.14	1.13	1.12	1.03	1.03	1.03	1.04	1.08	1.08	1.08	1.09

Table 7.10 Compression Ratio of Parsec with Dynamic Dictionary (DS=256) for MF.I (CS64)

Dictionary Entry Size in Bytes	DES=1				DES=2				DES=4			
Benchmark /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
blackscholes	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.12	2.08	2.03	1.97
bodytrack	1.45	1.42	1.39	-	1.00	1.00	1.00	-	1.05	1.06	1.07	-
canneal	1.08	1.07	1.07	1.07	1.00	1.00	1.00	1.00	1.18	1.17	1.17	1.17
dedup	1.10	1.09	1.08	1.07	1.16	1.16	1.16	1.15	1.16	1.15	1.14	1.14
ferret	1.04	1.03	1.03	1.02	1.01	1.01	1.01	1.01	1.21	1.21	1.20	1.20
fluidanimate	1.00	1.01	1.01	1.01	1.00	1.00	1.00	1.00	1.17	1.18	1.18	1.18
swaptions	1.18	1.26	1.31	0.94	1.14	1.21	1.26	0.94	1.27	1.37	1.46	1.01
vips	1.12	1.11	1.09	1.08	1.02	1.02	1.01	1.01	1.03	1.02	1.02	1.02
x264	1.62	1.54	1.57	1.57	1.02	1.02	1.02	1.01	1.04	1.05	1.04	1.04
Total	1.14	1.13	1.12	1.10	1.03	1.03	1.03	1.03	1.10	1.10	1.09	1.10

In conclusion, a dictionary helps reduce the number of reported trace bits. However, the effectiveness of a dictionary depends on the behavior of the benchmarks. In embedded systems, most of the time applications run by a processor are fixed and known ahead of the time. Thus, designers can use these dictionaries depending on the application.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The growing complexity of hardware with a shift to multicores and systems-on-a-chip, the growing complexity and sophistication of the software stack, and tightening time-to-market requirements make software testing and debugging one of the most critical and time-consuming aspects of embedded systems development. Thus, providing better tools to locate and fix software bugs faster helps increase the software reliability and reduce development cost.

Many vendors of embedded platforms are allocating on-chip resources dedicated to hardware tracing and debugging. Hardware control-flow and data-flow traces can be used to faithfully replay programs offline and thus locate bugs faster, and determine the history of events that led to a system crash. However, state-of-the-art hardware trace solutions typically do not support the on-the-fly data tracing that is crucial to debug and replay parallel programs because it is cost-prohibitive, requiring both deep on-chip trace buffers and wide trace ports.

This dissertation introduces a new hardware/software technique for on-the-fly capturing and filtering read data value traces in multicore systems called *mcFiltrate*. *mcFiltrate* uses first-access tracking bits attached to the L1 data caches of each processor core to keep track of data items that have already been emitted. In addition, *mcFiltrate* exploits cache coherence protocol states to further reduce the number of trace messages by eliminating the need to report a single shared data item by multiple processor cores. The software debugger with instruction set simula-

tor is extended to model the behavior of tracing hardware structures and to decode incoming trace messages to ensure faithful program replay on its side.

mcFiltrate effectiveness is explored using a simulation-based experimental environment built on top of the Multi2Sim architectural simulator running parallel programs from the Splash2 and Parsec benchmark suites. As a measure of effectiveness, the trace port bandwidth expressed in the number of bits streamed on the trace port per instruction executed and the number of bits per processor clock cycle is used. In addition, the size of on-chip trace buffers and the number of trace pins required for on-the-fly tracing is determined. The results show that *mcFiltrate* significantly reduces the required trace port bandwidth when compared to the original Nexus-like or even compressed Nexus-like read data tracing. The effectiveness of *mcFiltrate* is higher in systems with larger private caches and improves as the number of processors increase if inheriting FA bits is used in cache-to-cache transfers. For a single core processor, *mcFiltrate* reduces the average required trace port bandwidth relative to Nexus-like tracing from 13.4 times for Parsec running on the small cache configuration to 59.6 times for Splash2 running on the large cache configuration. For an octa-core processor, *mcFiltrate* reduces the average trace port bandwidth from 14.1 times for Parsec running on the small cache configuration to 73.8 times for Splash2 running on the large cache configuration. In addition, *mcFiltrate* makes real-time data tracing feasible as it reduces the requirements for on-chip trace buffers by the several orders of magnitude, and the number of required trace port pins is reduced up to 16 times. This dissertation also explores the effectiveness of additional dictionary-based trace compression and finds its overall effectiveness to be modest, although several benchmarks experienced significant benefits.

Although this dissertation focuses on unobtrusive data tracing through trace debug ports, *mcFiltrate* is applicable to hardware supported tracing scenarios where collected traces are written into the system memory. Future research can explore the obtrusiveness of *mcFiltrate* when traces are exposed to the system memory. Another possible venue for future research is to explore scaling to a larger number of processor cores and how hidden processor cores solely dedicated to handling trace data can be utilized for achieving even higher effectiveness.

APENDIX A

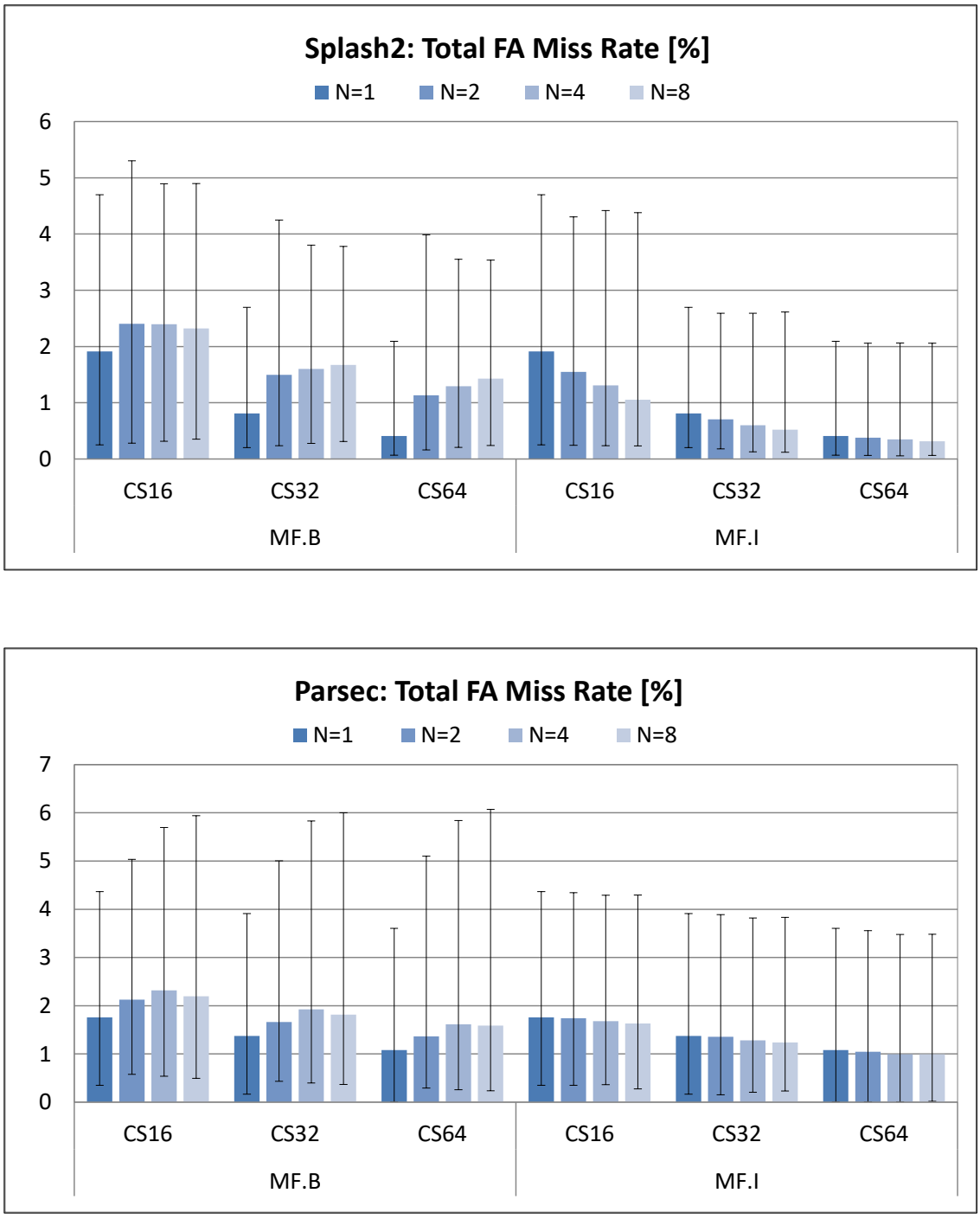


Figure. A.1 Total First-access Miss Rate of Splash2 (top) and Parsec (bottom) with G=32

A.1 Trace Port Bandwidth in BPI

A.1.1 Granularity Size is 4 (G=4)

Table. A.1 Average TPB in bpi for Splash2 with CS32 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.03	0.77	15.31	1.15	0.57	15.59	1.45	0.44	15.87	1.75	0.38
cholesky	15.33	0.67	16.27	0.71	0.70	15.85	0.56	0.52	15.61	0.40	0.34
fft	10.64	1.49	10.82	1.51	1.50	11.00	1.53	1.52	11.19	1.55	1.53
fmm	8.82	0.21	8.96	0.22	0.20	9.14	0.22	0.20	9.34	0.23	0.19
lu	11.87	0.52	12.06	0.52	0.50	12.26	0.53	0.37	12.47	0.40	0.23
radiosity	12.11	0.08	12.36	0.43	0.06	12.61	0.42	0.05	12.61	0.52	0.05
radix	13.40	0.53	13.74	1.41	0.56	14.09	1.48	0.57	14.53	1.53	0.60
raytrace	15.17	0.32	15.45	0.53	0.25	15.73	0.62	0.23	16.01	0.81	0.20
water-ns	10.64	0.22	10.81	0.25	0.22	10.98	0.39	0.03	11.15	0.41	0.03
water-sp	11.38	0.05	11.55	0.06	0.05	11.73	0.07	0.05	11.90	0.07	0.05
Total	12.34	0.35	12.62	0.54	0.32	12.88	0.59	0.29	13.17	0.63	0.27

Table. A.2 Average TPB in bpi for Parsec with CS32 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.17	0.47	12.43	0.99	0.48	12.68	1.13	0.48	12.94	1.17	0.49
bodytrack	9.74	0.57	10.01	0.93	0.52	10.29	1.06	0.47	-	-	-
canneal	12.08	1.19	12.39	1.22	1.21	12.71	1.24	1.24	13.04	1.27	1.26
dedup	13.30	0.99	13.66	1.02	1.01	14.02	1.01	1.00	14.39	1.04	1.02
ferret	10.41	0.77	10.67	0.77	0.77	10.92	0.75	0.75	11.19	0.73	0.73
fluidanimate	10.94	0.22	11.28	0.40	0.22	11.63	0.57	0.22	12.14	0.63	0.21
swaptions	13.42	0.02	13.69	0.16	0.02	13.96	0.41	0.02	14.24	0.64	0.02
vips	9.70	1.05	9.93	1.07	1.07	10.17	1.19	1.08	10.43	1.28	1.10
x264	6.97	0.18	7.21	0.22	0.22	7.49	0.22	0.21	7.76	0.21	0.20
Total	9.73	0.59	10.00	0.67	0.61	10.29	0.72	0.60	10.62	0.73	0.61

A.1.2 Granularity Size is 32 (G=32)

Table. A.3 Average TPB in bpi for Splash2 with CS16 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.03	3.97	15.31	4.07	2.95	15.59	4.14	2.13	15.86	4.20	1.58
cholesky	15.35	1.73	16.21	1.23	1.21	15.87	0.91	0.85	15.59	0.59	0.51
fft	10.65	2.42	10.84	2.46	2.46	11.02	2.46	2.46	11.19	2.46	2.46
fmm	8.82	0.53	8.96	0.55	0.49	9.14	0.56	0.47	9.33	0.57	0.46
lu	11.88	0.49	12.07	0.52	0.49	12.27	0.52	0.28	12.47	0.56	0.26
radiosity	12.11	0.54	12.36	1.19	0.31	12.59	1.21	0.29	12.58	1.38	0.21
radix	13.41	1.39	13.75	4.15	1.40	14.09	4.33	1.41	14.54	4.39	1.42
raytrace	15.17	1.95	15.45	2.27	1.56	15.73	2.43	1.40	16.01	2.70	1.22
water-ns	10.64	0.82	10.81	0.92	0.82	10.98	1.04	0.63	11.15	1.15	0.15
water-sp	11.38	0.12	11.55	0.14	0.12	11.73	0.16	0.12	11.90	0.17	0.11
Total	12.34	1.23	12.63	1.55	1.00	12.89	1.56	0.85	13.17	1.55	0.70

Table. A.4 Average TPB in bpi for Splash2 with CS32 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.03	1.39	15.31	2.01	1.04	15.59	2.46	0.79	15.87	2.92	0.66
cholesky	15.33	0.68	16.27	0.73	0.71	15.85	0.58	0.52	15.61	0.41	0.35
fft	10.64	1.41	10.82	1.42	1.42	11.00	1.42	1.42	11.19	1.43	1.43
fmm	8.82	0.36	8.96	0.38	0.35	9.14	0.40	0.35	9.34	0.41	0.33
lu	11.87	0.48	12.06	0.48	0.45	12.26	0.49	0.25	12.47	0.40	0.17
radiosity	12.11	0.19	12.36	0.95	0.14	12.61	0.95	0.12	12.61	1.16	0.10
radix	13.40	0.67	13.74	3.45	0.67	14.09	3.60	0.68	14.53	3.64	0.69
raytrace	15.17	0.61	15.45	1.01	0.46	15.73	1.21	0.41	16.01	1.52	0.35
water-ns	10.64	0.37	10.81	0.47	0.37	10.98	0.82	0.06	11.15	1.06	0.05
water-sp	11.38	0.10	11.55	0.12	0.10	11.73	0.14	0.09	11.90	0.15	0.08
Total	12.34	0.52	12.62	0.96	0.45	12.88	1.04	0.39	13.17	1.11	0.35

Table. A.5 Average TPB in bpi for Splash2 with CS64 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	15.02	0.37	15.31	1.13	0.31	15.59	1.70	0.28	15.86	2.44	0.32
cholesky	15.29	0.62	16.23	0.65	0.63	15.87	0.51	0.45	15.62	0.36	0.29
fft	10.63	1.09	10.82	1.10	1.10	11.00	1.10	1.10	11.19	1.11	1.11
fmm	8.82	0.26	8.96	0.28	0.26	9.14	0.30	0.25	9.33	0.31	0.24
lu	11.86	0.45	12.06	0.30	0.27	12.26	0.32	0.15	12.47	0.20	0.07
radiosity	12.11	0.09	12.32	0.87	0.07	12.61	0.90	0.08	12.61	1.12	0.06
radix	13.38	0.38	13.74	3.17	0.38	14.09	3.30	0.38	14.53	3.35	0.39
raytrace	15.16	0.19	15.45	0.63	0.15	15.73	0.83	0.12	16.01	1.15	0.10
water-ns	10.64	0.03	10.81	0.13	0.03	10.97	0.76	0.03	11.15	1.04	0.03
water-sp	11.38	0.06	11.55	0.08	0.06	11.73	0.10	0.05	11.90	0.12	0.04
Total	12.33	0.26	12.62	0.72	0.24	12.88	0.84	0.23	13.17	0.95	0.21

Table. A.6 Average TPB in bpi for Parsec with CS16 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.18	2.53	12.43	3.24	2.52	12.68	3.83	2.53	12.94	4.24	2.53
bodytrack	9.77	2.02	10.02	3.37	1.62	10.29	4.12	1.22	-	-	-
canneal	12.12	3.89	12.41	3.92	3.90	12.72	3.93	3.91	13.04	3.95	3.91
dedup	13.32	2.39	13.69	2.43	2.42	14.03	2.41	2.39	14.39	2.42	2.40
ferret	10.42	2.15	10.67	2.13	2.13	10.93	2.11	2.11	11.19	1.99	1.98
fluidanimate	10.94	0.26	11.28	1.14	0.27	11.63	2.02	0.28	12.14	2.36	0.28
swaptions	13.42	0.76	13.69	1.10	0.77	13.96	1.73	0.79	14.24	2.49	0.79
vips	9.71	1.48	9.95	1.79	1.42	10.19	1.81	1.43	10.43	2.00	1.40
x264	6.98	0.36	7.22	0.46	0.46	7.49	0.43	0.42	7.76	0.40	0.38
Total	9.74	1.38	10.01	1.67	1.37	10.30	1.83	1.33	10.62	1.75	1.30

Table. A.7 Average TPB in bpi for Parsec with CS32 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.18	2.48	12.43	3.25	2.48	12.68	3.86	2.48	12.94	4.25	2.45
bodytrack	9.77	1.85	10.02	3.22	1.42	10.29	3.95	1.01	-	-	-
canneal	12.12	3.48	12.41	3.51	3.49	12.72	3.52	3.49	13.04	3.54	3.48
dedup	13.32	1.85	13.69	1.88	1.88	14.03	1.85	1.83	14.39	1.87	1.84
ferret	10.42	1.20	10.67	1.16	1.16	10.93	1.11	1.11	11.19	1.07	1.07
fluidanimate	10.94	0.25	11.28	1.14	0.24	11.63	2.01	0.25	12.14	2.33	0.24
swaptions	13.42	0.13	13.69	0.47	0.12	13.96	1.18	0.16	14.24	1.88	0.18
vips	9.71	1.21	9.95	1.23	1.21	10.19	1.54	1.17	10.43	1.76	1.14
x264	6.98	0.27	7.22	0.34	0.34	7.49	0.32	0.31	7.76	0.29	0.28
Total	9.74	1.08	10.01	1.31	1.07	10.30	1.52	1.02	10.62	1.45	0.99

Table. A.8 Average TPB in bpi for Parsec with CS64 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	12.18	2.48	12.43	3.33	2.48	12.68	3.88	2.45	12.94	4.25	2.31
bodytrack	9.77	1.10	10.02	2.44	0.66	10.29	3.21	0.41	-	-	-
canneal	12.12	3.17	12.41	3.20	3.17	12.72	3.22	3.16	13.04	3.23	3.14
dedup	13.32	1.26	13.69	1.27	1.26	14.03	1.28	1.26	14.39	1.30	1.27
ferret	10.42	0.84	10.67	0.80	0.80	10.93	0.78	0.77	11.19	0.76	0.75
fluidanimate	10.94	0.22	11.28	1.11	0.21	11.63	1.98	0.22	12.14	2.31	0.21
swaptions	13.42	0.00	13.69	0.27	0.00	13.96	1.08	0.00	14.24	1.85	0.01
vips	9.71	1.08	9.95	1.16	1.07	10.19	1.35	1.05	10.43	1.64	1.01
x264	6.98	0.19	7.22	0.23	0.23	7.49	0.20	0.19	7.76	0.19	0.18
Total	9.74	0.85	10.01	1.08	0.83	10.30	1.28	0.79	10.62	1.27	0.80

A.2 Trace Port Bandwidth in BPC

A.2.1 Granularity Size is 4 (G=4)

Table. A.9 Average TPB in bpc for Splash2 with CS16 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	5.50	0.79	8.24	1.22	0.88	14.96	2.25	1.14	26.79	4.05	1.50
cholesky	2.93	0.34	6.58	0.49	0.49	14.67	0.81	0.77	32.99	1.19	1.05
fft	2.79	0.67	4.78	1.16	1.16	7.93	1.91	1.91	11.59	2.77	2.77
fmm	3.59	0.14	7.17	0.27	0.24	13.92	0.53	0.43	25.16	0.94	0.74
lu	4.67	0.21	8.97	0.42	0.40	15.56	0.73	0.58	24.38	1.19	0.84
radiosity	5.87	0.11	10.79	0.45	0.11	20.81	0.85	0.19	37.60	1.77	0.26
radix	3.14	0.18	5.01	0.60	0.28	7.67	0.94	0.43	9.41	1.15	0.53
raytrace	7.53	0.46	14.35	1.02	0.72	26.40	1.98	1.21	42.70	3.57	1.70
water-ns	6.49	0.29	12.68	0.59	0.56	24.34	1.19	0.82	43.51	2.12	0.31
water-sp	7.50	0.04	12.40	0.08	0.07	20.29	0.13	0.11	32.48	0.23	0.17
Total	4.92	0.31	8.76	0.61	0.44	15.61	1.07	0.68	25.64	1.71	0.93

Table. A.10 Average TPB in bpc for Splash2 with CS32 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	6.20	0.32	8.92	0.67	0.33	15.82	1.47	0.45	27.43	3.02	0.65
cholesky	3.30	0.14	7.44	0.33	0.32	16.75	0.60	0.54	32.91	0.85	0.73
fft	3.07	0.43	5.32	0.74	0.74	8.77	1.22	1.21	11.83	1.63	1.62
fmm	3.70	0.09	7.37	0.18	0.16	14.12	0.34	0.30	25.36	0.61	0.52
lu	5.16	0.23	9.68	0.41	0.40	16.68	0.72	0.50	24.64	0.80	0.46
radiosity	6.11	0.04	11.26	0.39	0.05	21.85	0.73	0.09	38.37	1.57	0.14
radix	3.39	0.13	5.20	0.53	0.21	7.87	0.83	0.32	9.47	0.99	0.39
raytrace	8.62	0.18	16.01	0.55	0.26	29.03	1.15	0.42	45.82	2.31	0.57
water-ns	7.00	0.15	13.50	0.31	0.28	24.71	0.87	0.07	43.29	1.59	0.10
water-sp	7.63	0.04	12.57	0.06	0.06	20.42	0.12	0.09	32.64	0.20	0.13
Total	5.31	0.15	9.32	0.40	0.24	16.44	0.75	0.37	25.99	1.24	0.52

Table. A.11 Average TPB in bpc for Splash2 with CS64 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	6.63	0.08	9.20	0.40	0.10	16.20	1.08	0.16	28.05	2.65	0.33
cholesky	4.10	0.17	7.68	0.31	0.30	16.64	0.53	0.47	33.06	0.75	0.61
fft	3.37	0.36	5.41	0.58	0.58	8.91	0.96	0.95	11.91	1.27	1.27
fmm	3.77	0.06	7.41	0.12	0.11	14.19	0.23	0.21	25.38	0.42	0.35
lu	5.59	0.22	9.82	0.25	0.22	16.87	0.44	0.26	24.82	0.33	0.10
radiosity	6.23	0.02	11.33	0.36	0.03	21.99	0.70	0.06	38.50	1.52	0.10
radix	3.57	0.12	5.28	0.49	0.17	7.91	0.76	0.26	9.49	0.93	0.31
raytrace	9.05	0.06	16.53	0.36	0.09	29.89	0.83	0.14	46.47	1.81	0.18
water-ns	7.50	0.01	14.51	0.06	0.02	25.04	0.71	0.03	43.43	1.51	0.05
water-sp	7.72	0.02	12.61	0.04	0.03	20.47	0.08	0.05	32.68	0.15	0.06
Total	5.65	0.09	9.49	0.30	0.15	16.59	0.60	0.25	26.14	1.05	0.35

Table. A.12 Average TPB in bpc for Parsec with CS16 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	6.31	0.26	10.47	0.82	0.41	15.01	1.34	0.59	19.36	1.77	0.76
bodytrack	2.43	0.16	4.10	0.40	0.24	6.38	0.69	0.34	-	-	-
canneal	2.24	0.24	2.90	0.31	0.31	3.36	0.36	0.36	3.58	0.38	0.38
dedup	2.78	0.24	5.19	0.44	0.44	9.60	0.81	0.81	14.82	1.24	1.23
ferret	2.64	0.38	5.60	0.79	0.79	9.21	1.27	1.27	12.01	1.57	1.57
fluidanimate	5.49	0.12	9.20	0.34	0.19	12.83	0.65	0.27	16.13	0.87	0.31
swaptions	5.71	0.03	10.78	0.15	0.05	18.79	0.61	0.12	28.80	1.47	0.20
vips	2.48	0.29	4.76	0.59	0.54	9.89	1.23	1.13	17.74	2.27	1.99
x264	4.45	0.14	6.18	0.24	0.24	10.42	0.37	0.37	14.72	0.49	0.47
Total	3.12	0.23	5.24	0.43	0.39	8.41	0.70	0.61	11.56	0.95	0.82

Table. A.13 Average TPB in bpc for Parsec with CS32 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	7.57	0.29	11.31	0.90	0.43	15.88	1.41	0.61	19.68	1.78	0.75
bodytrack	2.89	0.17	4.36	0.41	0.23	6.58	0.68	0.30	-	-	-
canneal	2.63	0.26	3.16	0.31	0.31	3.47	0.34	0.34	3.67	0.36	0.36
dedup	3.55	0.27	6.58	0.49	0.49	11.21	0.81	0.80	15.09	1.09	1.07
ferret	3.62	0.27	7.05	0.51	0.51	11.27	0.78	0.78	14.72	0.96	0.96
fluidanimate	5.72	0.12	9.46	0.34	0.18	13.00	0.64	0.25	16.19	0.84	0.28
swaptions	6.43	0.01	11.89	0.14	0.01	20.05	0.59	0.03	30.22	1.36	0.03
vips	2.85	0.31	6.01	0.65	0.65	10.77	1.26	1.14	17.93	2.20	1.88
x264	5.18	0.14	7.07	0.22	0.22	11.11	0.32	0.31	15.01	0.40	0.39
Total	3.73	0.23	6.12	0.41	0.37	9.07	0.64	0.53	12.01	0.83	0.69

Table. A.14 Average TPB in bpc for Parsec with CS64 (G=4)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	7.59	0.29	11.10	0.86	0.42	15.89	1.41	0.60	19.70	1.77	0.75
bodytrack	3.21	0.14	4.43	0.35	0.15	6.61	0.60	0.18	-	-	-
canneal	2.84	0.26	3.22	0.29	0.29	3.53	0.32	0.32	3.74	0.34	0.33
dedup	4.29	0.26	6.88	0.40	0.40	11.50	0.67	0.65	15.31	0.88	0.86
ferret	4.20	0.20	7.06	0.32	0.32	10.61	0.46	0.46	12.86	0.55	0.54
fluidanimate	5.93	0.11	9.51	0.32	0.17	13.04	0.62	0.23	16.20	0.82	0.26
swaptions	6.62	0.00	12.49	0.10	0.00	21.07	0.63	0.00	30.99	1.40	0.00
vips	3.38	0.35	6.00	0.64	0.62	11.31	1.26	1.16	18.18	2.16	1.83
x264	6.00	0.12	7.21	0.17	0.17	11.15	0.24	0.23	15.12	0.31	0.29
Total	4.26	0.22	6.21	0.36	0.32	9.19	0.56	0.45	11.99	0.73	0.59

A.2.2 Granularity Size is 32 (G=32)

Table. A.15 Average TPB in bpc for Splash2 with CS16 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	5.50	6.20	8.24	8.92	1.59	14.96	15.82	27.43	26.79	27.43	2.66
cholesky	2.93	3.30	6.58	7.44	0.49	14.67	16.75	32.91	32.99	32.91	1.09
fft	2.79	3.07	4.78	5.32	1.08	7.93	8.77	11.83	11.59	11.83	2.54
fmm	3.59	3.70	7.17	7.37	0.39	13.92	14.12	25.36	25.16	25.36	1.23
lu	4.67	5.16	8.97	9.68	0.36	15.56	16.68	24.64	24.38	24.64	0.51
radiosity	5.87	6.11	10.79	11.26	0.27	20.81	21.85	38.37	37.60	38.37	0.62
radix	3.14	3.39	5.01	5.20	0.51	7.67	7.87	9.47	9.41	9.47	0.92
raytrace	7.53	8.62	14.35	16.01	1.44	26.40	29.03	45.82	42.70	45.82	3.25
water-ns	6.49	7.00	12.68	13.50	0.96	24.34	24.71	43.29	43.51	43.29	0.58
water-sp	7.50	7.63	12.40	12.57	0.13	20.29	20.42	32.64	32.48	32.64	0.31
Total	4.92	5.31	8.76	9.32	0.69	15.61	16.44	25.99	25.64	25.99	1.36

Table. A.16 Average TPB in bpc for Splash2 with CS32 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	6.20	1.45	8.92	2.19	0.60	15.82	3.97	7.09	27.43	7.09	1.14
cholesky	3.30	0.33	7.44	0.50	0.33	16.75	0.84	1.25	32.91	1.25	0.74
fft	3.07	0.63	5.32	1.08	0.70	8.77	1.77	2.55	11.83	2.55	1.51
fmm	3.70	0.22	7.37	0.44	0.29	14.12	0.86	1.54	25.36	1.54	0.91
lu	5.16	0.19	9.68	0.38	0.36	16.68	0.66	1.10	24.64	1.10	0.33
radiosity	6.11	0.26	11.26	1.04	0.13	21.85	1.99	4.12	38.37	4.12	0.30
radix	3.39	0.33	5.20	1.51	0.25	7.87	2.35	2.84	9.47	2.84	0.45
raytrace	8.62	0.97	16.01	2.11	0.47	29.03	4.08	7.21	45.82	7.21	1.00
water-ns	7.00	0.50	13.50	1.08	0.47	24.71	2.30	4.48	43.29	4.48	0.21
water-sp	7.63	0.08	12.57	0.15	0.10	20.42	0.27	0.47	32.64	0.47	0.23
Total	5.31	0.49	9.32	1.07	0.33	16.44	1.89	3.02	25.99	3.02	0.68

Table. A.17 Average TPB in bpc for Splash2 with CS64 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
barnes	6.63	0.57	9.20	1.17	0.18	16.20	2.50	5.05	28.05	5.05	0.57
cholesky	4.10	0.15	7.68	0.33	0.30	16.64	0.61	0.87	33.06	0.87	0.61
fft	3.37	0.41	5.41	0.70	0.55	8.91	1.14	1.51	11.91	1.51	1.18
fmm	3.77	0.15	7.41	0.31	0.21	14.19	0.61	1.11	25.38	1.11	0.66
lu	5.59	0.21	9.82	0.39	0.22	16.87	0.67	0.79	24.82	0.79	0.14
radiosity	6.23	0.10	11.33	0.87	0.07	21.99	1.65	3.54	38.50	3.54	0.19
radix	3.57	0.17	5.28	1.31	0.15	7.91	2.01	2.37	9.49	2.37	0.26
raytrace	9.05	0.34	16.53	1.05	0.16	29.89	2.22	4.34	46.47	4.34	0.29
water-ns	7.50	0.24	14.51	0.59	0.04	25.04	1.86	4.12	43.43	4.12	0.12
water-sp	7.72	0.07	12.61	0.13	0.06	20.47	0.24	0.42	32.68	0.42	0.11
Total	5.65	0.22	9.49	0.71	0.18	16.59	1.33	2.20	26.14	2.20	0.42

Table. A.18 Average TPB in bpc for Parsec with CS16 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	6.31	1.31	10.47	2.73	2.12	15.01	4.53	2.99	19.36	6.34	3.79
bodytrack	2.43	0.50	4.10	1.38	0.66	6.38	2.55	0.76	-	-	-
canneal	2.24	0.72	2.90	0.92	0.91	3.36	1.04	1.03	3.58	1.08	1.07
dedup	2.78	0.50	5.19	0.92	0.92	9.60	1.65	1.64	14.82	2.49	2.47
ferret	2.64	0.54	5.60	1.12	1.12	9.21	1.78	1.78	12.01	2.13	2.13
fluidanimate	5.49	0.13	9.20	0.93	0.22	12.83	2.23	0.31	16.13	3.14	0.37
swaptions	5.71	0.33	10.78	0.87	0.61	18.79	2.33	1.06	28.80	5.03	1.60
vips	2.48	0.38	4.76	0.85	0.68	9.89	1.76	1.39	17.74	3.39	2.38
x264	4.45	0.23	6.18	0.39	0.39	10.42	0.59	0.58	14.72	0.75	0.72
Total	3.12	0.44	5.24	0.88	0.72	8.41	1.50	1.09	11.56	1.91	1.42

Table. A.19 Average TPB in bpc for Parsec with CS32 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	7.57	1.54	11.31	2.96	2.26	15.88	4.83	3.10	19.68	6.46	3.73
bodytrack	2.89	0.55	4.36	1.41	0.62	6.58	2.53	0.65	-	-	-
canneal	2.63	0.76	3.16	0.89	0.89	3.47	0.96	0.95	3.67	1.00	0.98
dedup	3.55	0.49	6.58	0.91	0.90	11.21	1.48	1.47	15.09	1.96	1.93
ferret	3.62	0.42	7.05	0.77	0.76	11.27	1.15	1.15	14.72	1.41	1.41
fluidanimate	5.72	0.13	9.46	0.95	0.20	13.00	2.24	0.28	16.19	3.10	0.32
swaptions	6.43	0.06	11.89	0.41	0.10	20.05	1.69	0.23	30.22	4.00	0.38
vips	2.85	0.36	6.01	0.74	0.73	10.77	1.63	1.24	17.93	3.03	1.97
x264	5.18	0.20	7.07	0.34	0.33	11.11	0.47	0.46	15.01	0.57	0.54
Total	3.73	0.41	6.12	0.80	0.66	9.07	1.34	0.90	12.01	1.64	1.12

Table. A.20 Average TPB in bpc for Parsec with CS64 (G=32)

# Cores	N=1		N=2			N=4			N=8		
Mechanism	NX	MF.B|I	NX	MF.B	MF.I	NX	MF.B	MF.I	NX	MF.B	MF.I
blackscholes	7.59	1.55	11.10	2.97	2.22	15.89	4.86	3.07	19.70	6.47	3.52
bodytrack	3.21	0.36	4.43	1.08	0.29	6.61	2.06	0.26	-	-	-
canneal	2.84	0.75	3.22	0.83	0.82	3.53	0.89	0.88	3.74	0.93	0.90
dedup	4.29	0.41	6.88	0.64	0.64	11.50	1.05	1.04	15.31	1.38	1.35
ferret	4.20	0.34	7.06	0.53	0.53	10.61	0.76	0.75	12.86	0.87	0.87
fluidanimate	5.93	0.12	9.51	0.94	0.18	13.04	2.22	0.25	16.20	3.08	0.28
swaptions	6.62	0.00	12.49	0.25	0.00	21.07	1.63	0.00	30.99	4.02	0.03
vips	3.38	0.38	6.00	0.70	0.65	11.31	1.50	1.16	18.18	2.86	1.76
x264	6.00	0.16	7.21	0.23	0.23	11.15	0.31	0.29	15.12	0.37	0.34
Total	4.26	0.37	6.21	0.67	0.51	9.19	1.14	0.70	11.99	1.43	0.90

A.3 Compression Ratio with Dictionaries

Table. A.21 Compression Ratio of Splash2 with DS=256 and DES = 4 and 8 for MF.I (CS64)

Dictionary Entry Size in Bytes	Static				Dynamic			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
blackscholes	1.25	1.25	1.24	1.23	1.58	1.56	1.55	1.53
bodytrack	1.04	1.08	1.22	1.41	2.14	2.03	1.97	1.84
canneal	1.13	1.13	1.12	1.12	1.27	1.25	1.24	1.23
dedup	1.44	1.44	1.44	1.43	2.33	2.23	2.05	1.57
ferret	1.08	1.07	1.07	1.07	2.29	2.27	2.28	2.22
fluidanimate	1.28	1.28	1.26	1.24	1.39	1.36	1.36	1.36
swaptions	1.26	1.25	1.25	1.24	12.93	8.38	5.04	2.87
vips	1.24	1.23	1.22	1.22	1.60	1.61	1.54	1.46
x264	1.21	1.21	1.20	1.20	1.91	1.81	1.60	1.49
Total	1.21	1.21	1.20	1.20	1.89	1.80	1.67	1.54

Table. A.22 Compression Ratio of Parsec with DS=256 and DES = 4 and 8 for MF.I (CS64)

Dictionary Entry Size in Bytes	Static				Dynamic			
Benchmark/Cores	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
blackscholes	1.46	1.44	1.43	1.42	1.95	2.07	1.84	1.79
bodytrack	1.24	1.23	1.22	-	1.53	1.54	1.54	-
canneal	1.30	1.29	1.28	1.27	1.72	1.68	1.64	1.61
dedup	1.16	1.13	1.12	1.11	1.67	1.64	1.60	1.56
ferret	1.18	1.18	1.17	1.17	1.70	1.66	1.62	1.58
fluidanimate	1.30	1.29	1.28	1.28	2.13	2.04	1.97	1.90
swaptions	1.55	1.47	1.42	1.13	2.12	2.00	1.92	1.85
vips	1.42	1.39	1.38	1.06	1.63	1.59	1.57	1.52
x264	1.17	1.16	0.99	0.99	1.27	1.25	1.24	1.22
Total	1.26	1.24	1.18	1.11	1.60	1.57	1.54	1.51

REFERENCES

[1] R. N. Charette, "This Car Runs on Code," *IEEE Spectrum: Technology, Engineering, and Science News*, 01-Feb-2009.

[2] E. Sweeney, "Medical device recalls reach historic levels in 2018 with software as leading cause," *FierceHealthcare*, 09-May-2018.

[3] Z. Fu, C. Guo, S. Ren, Y. Jiang, and L. Sha, "Study of Software-Related Causes in the FDA Medical Device Recalls," in *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, Fukuoka, Japan, 2017, pp. 60–69.

[4] "5 Automotive Embedded Software Recalls and Updates we've seen in 2014 | Vector Software." [Online]. Available: https://www.vectorcast.com/blog/2014/06/5-automotive-embedded-software-recalls-and-updates-weve-seen-2014-0. [Accessed: 01-Jul-2018].

[5] J. Wakefield, "Nest thermostat bug leaves users cold," *BBC News*, 14-Jan-2016.

[6] J. Plungis, "Fiat Chrysler Recalls 4.8 Million Vehicles Because Cruise Control May Not Turn Off," *Consumer Reports*. [Online]. Available: https://www.consumerreports.org/car-recalls-defects/fiat-chrysler-recalls-4-8-million-vehicles-because-cruise-control-may-not-turn-off/. [Accessed: 01-Jul-2018].

[7] A. Al-Heeti and D. Kerr, "Uber's fatal self-driving crash reportedly caused by software," *CNET*, 07-May-2018.

[8] "University of Cambridge Reverse Debugging Study." [Online]. Available: https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study. [Accessed: 17-Dec-2017].

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

[10] S. Rostedt, “ftrace - Function Tracer — The Linux Kernel documentation.” [Online]. Available: https://www.kernel.org/doc/html/v4.18/trace/ftrace.html. [Accessed: 19-Apr-2019].

[11] “About DTrace.” [Online]. Available: http://dtrace.org/blogs/about/. [Accessed: 26-Apr-2019].

[12] “LTTng v2.10 — LTTng Documentation,” *LTTng*. [Online]. Available: http://lttng.org/docs/v2.10/. [Accessed: 26-Apr-2019].

[13] M. Fleming, “A thorough introduction to eBPF.” [Online]. Available: https://lwn.net/Articles/740157/. [Accessed: 26-Apr-2019].

[14] “SystemTap.” [Online]. Available: https://sourceware.org/systemtap/. [Accessed: 26-Apr-2019].

[15] windows-sdk-content, “About Event Tracing - Windows applications.” [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/etw/about-event-tracing. [Accessed: 19-Mar-2019].

[16] C.-K. Luk *et al.*, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Chicago, IL, USA, 2005, pp. 190–200.

[17] M. Williams, “ARMV8 debug and trace architectures,” in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Vienna, Austria, 2012, pp. 1–6.

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

[19] M. Ponugoti and A. Milenković, “Exploiting Cache Coherence for Effective On-the-Fly Data Tracing in Multicores,” in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Phoenix, AZ, 2016, pp. 312–319.

[20] 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, pp. 312–319.

[21] 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.

[22] 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.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, 2008, p. 72.

[24] P. Anand, "Dynamic tracing in Linux user and kernel space," *Opensource.com*, 06-Jul-2017. [Online]. Available: https://opensource.com/article/17/7/dynamic-tracing-linux-user-and-kernel-space. [Accessed: 26-May-2019].

[25] 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.

[26] P. Padala, "Playing with ptrace, Part I | Linux Journal." [Online]. Available: https://www.linuxjournal.com/article/6100. [Accessed: 19-Mar-2019].

[27] "Strace." [Online]. Available: https://strace.io/. [Accessed: 28-Apr-2019].

[28] "ltrace." [Online]. Available: https://www.ltrace.org/. [Accessed: 28-Apr-2019].

[29] "Linux system exploration and troubleshooting tool with first class support for containers: draios/sysdig." [Online]. Available: https://github.com/draios/sysdig. [Accessed: 26-Apr-2019].

[30] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001, pp. 3–14.

[31] Free Software Foundation, "Debugging with GDB: Process Record and Replay." [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Process-Record-and-Replay.html. [Accessed: 19-Mar-2019].

[32] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: a technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*, Monterey, California, 2015, pp. 344–360.

[33] S. D. Sharma and M. Dagenais, "Hardware-assisted instruction profiling and latency detection," *The Journal of Engineering*, vol. 2016, no. 10, pp. 367–376, Oct. 2016.

[34] L. Chen, S. Sultana, and R. Sahita, "HeNet: A Deep Learning Approach on Intel Processor Trace for Effective Exploit Detection," *arXiv:1801.02318 [cs]*, Jan. 2018.

[35] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows Using Intel Processor Trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2017, pp. 585–598.

[36] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek, "VM-CFI: Control-Flow Integrity for Virtual Machine Kernel Using Intel PT," in *Computational Science and Its Applications – ICCSA 2018*, 2018, pp. 127–137.

[37] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace," presented at the Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, Arizona, USA, 2017, pp. 173–184.

[38] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and Efficient CFI Enforcement with Intel Processor Trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, USA, 2017, pp. 529–540.

[39] X. Wang, F. Huang, and H. Chen, "DTrace: fine-grained and efficient data integrity checking with hardware instruction tracing," *Cybersecurity*, vol. 2, no. 1, p. 1, Jan. 2019.

[40] J. Glanz, J. Creswell, T. Kaplan, and Z. Wichter, "After a Lion Air 737 Max Crashed in October, Questions About the Plane Arose," *The New York Times*, 10-Mar-2019.

[41] Greenhills, "SuperTrace Probe hardware debugger." [Online]. Available: https://www.ghs.com/products/supertraceprobe.html. [Accessed: 30-Jun-2018].

[42] Lauterbach, "TRACE32 PowerTrace Serial." [Online]. Available: https://www.lauterbach.com/frames.html?home.html. [Accessed: 30-Jun-2018].

[43] ARM, "DSTREAM." [Online]. Available: https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream. [Accessed: 02-Jul-2018].

[44] ARM, "ARM High Speed Serial Trace Probe (HSSTP)," *ARM Developer*. [Online]. Available: https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream-family/dstream/high-speed-serial-trace-probe. [Accessed: 25-Mar-2019].

[45] J. Campbell, V. Kazantsev, and H. O'Keeffe, "Real-Time Trace: A Better Way to Debug Embedded Applications," Ashling Microsystems, White Paper.

[46] ARM, "Arm Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.4," 30-Apr-2018. [Online]. Available: https://static.docs.arm.com/ihi0064/f/etm_v4_4_architecture_specification_IHI0064F.pdf. [Accessed: 07-Jun-2018].

[47] MIPS Technologies, "MIPS PDtrace Specification," 19-Dec-2012. [Online]. Available: http://www.t-es-t.hu/download/mips/md00439g.pdf. [Accessed: 01-Apr-2016].

[48] Intel, "Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3C," Sep-2016. [Online]. Available: https://goo.gl/QLKR85. [Accessed: 11-Jul-2017].

[49] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Boosting Debugging Support for Complex Systems on Chip," *Computer*, vol. 40, no. 4, pp. 76–81, Apr. 2007.

[50] Intel, "Nios II Processor Reference Guide," Intel, Apr. 2018.

[51] Freescale, "Freescale - MPC555 / MPC556 USER'S MANUAL," Nov-2009. [Online]. Available: http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf. [Accessed: 28-Mar-2016].

[52] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," 2012. [Online]. Available: http://nexus5001.org/nexus-5001-forum-standard/. [Accessed: 28-Mar-2016].

[53] R. Mijat, "Better Trace for Better Software," ARM, White Paper, 2010.

[54] ARM, "CoreSight PTM-A9 Technical Reference Manual," 08-Jul-2011. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0401c/DDI0401C_coresight_ptm_a9_r1p0_trm.pdf. [Accessed: 25-Mar-2019].

[55] "Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3." Intel, Sep-2016.

[56] M. L. Soffa, K. R. Walcott, and J. Mars, "Exploiting Hardware Advances for Software Testing and Debugging (NIER Track)," in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 888–891.

[57] "Intel PT Micro Tutorial." [Online]. Available: https://sites.google.com/site/intelptmicrotutorial/home. [Accessed: 21-Mar-2019].

[58] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, New York, NY, USA, 2010, pp. 2–11.

[59] S. Bhansali *et al.*, "Framework for Instruction-level Tracing and Analysis of Program Executions," in *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, New York, NY, USA, 2006, pp. 154–163.

[60] M. Xu, R. Bodik, and M. D. Hill, "A 'Flight Data Recorder' for Enabling Full-system Multiprocessor Deterministic Replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2003, pp. 122–135.

[61] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *SIGARCH Comput. Archit. News*, vol. 33, pp. 284–295, 2005.

[62] M. Xu, M. D. Hill, and R. Bodik, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, California, USA, 2006, pp. 49–60.

[63] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *Proceedings of the 35th International Symposium on Computer Architecture*, Beijing, China, 2008, pp. 289–300.

[64] Y. Chen, W. Hu, T. Chen, and R. Wu, "LReplay: A Pending Period Based Deterministic Replay Scheme," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010, pp. 187–197.

[65] 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.

[66] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transaction on Information Theory*, vol. 23, pp. 337–343, 1977.

[67] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital SRC, 1994.

[68] A. D. Samples, "Mache: No-loss Trace Compaction," in *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 1989, pp. 89–97.

[69] A. R. Pleszkun, "Techniques for compressing program address traces," in *Proceedings of the 27th annual international symposium on Microarchitecture*, San Jose, CA, USA, 1994, pp. 32–39.

[70] E. E. Johnson and Jiheng Ha, "PDATS Lossless Address Trace Compression For Reducing File Size And Access Time," in *Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications*, Phoenix, Arizona, USA, 1994, p. 213.

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

[72] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, Nov. 2000.

[73] E. E. Johnson, J. Ha, and M. B. Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 158–173, 2001.

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

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

[76] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 1996, pp. 138–147.

[77] M. Burtscher and B. G. Zorn, "Exploring Last n Value Prediction," in *International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, CA, USA, 1999.

[78] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, Washington, DC, USA, 1997, pp. 281–290.

[79] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, Washington, DC, USA, 1997, pp. 248–258.

[80] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam, "The VPC Trace-Compression Algorithms," *IEEE Trans. Comput.*, vol. 54, no. 11, pp. 1329–1344, 2005.

[81] M. Burtscher, "TCgen 2.0: a tool to automatically generate lossless trace compressors," *SIGARCH Computer Architecture News*, vol. 34, no. 3, pp. 1–8, 2006.

[82] Y. Luo and L. K. John, "Locality-Based Online Trace Compression," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 723–731, 2004.

[83] 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.

[84] E. N. Elnozahy, "Address Trace Compression Through Loop Detection and Reduction," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 1999, pp. 214–215.

[85] K. Irrgang and T. B. Preußer, "An LZ77-style bit-level compression for trace data compaction," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, London, UK, 2015, pp. 1–4.

[86] 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, no. 3, pp. 530–543, Mar. 2007.

[87] V. Uzelac and A. Milenkovic, "A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method," in *Proceedings of the Design Automation Conference*, San Francisco, CA, 2009, pp. 738–743.

[88] 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.

[89] A. Tewar, A. Myers, and A. Milenković, "mcfTRaptor: Toward unobtrusive on-the-fly control-flow tracing in multicores," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 601–614, Nov. 2015.

[90] 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.

[91] 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, pp. 75:1–75:24, Sep. 2015.

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

[93] V. Uzelac and A. Milenković, "Hardware-based data value and address trace filtering techniques," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (CASES '10)*, New York, USA, 2010, pp. 117–126.

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

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

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

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

[98] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HPL-2008-20, Apr. 2008.