A BINARY INSTRUMENTATION TOOL SUITE FOR CAPTURING AND COMPRESSING TRACES FOR MULTITHREADED SOFTWARE

by

Albert R. Myers

A THESIS

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

HUNTSVILLE, ALABAMA

2014

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

____________________________ ____________

(student signature) (date)

THESIS APPROVAL FORM

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

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

_______________________________________ Committee Chair
(Date)

____

____

____

_______________________________________ Department Chair

_______________________________________ College Dean

_______________________________________ Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

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

Name of Candidate Albert R. Myers

Title A Binary Instrumentation Tool Suite For Capturing and Compressing Traces For Multithreaded Software

Program execution traces are widely used in program debugging, workload characterization, performance analysis, and trace-driven architecture simulation. A number of research efforts have been dedicated to tracing in single-threaded software. Multi-cores that integrate a number of processor cores on a single chip and execute multithreaded software have become the standard in embedded, desktop, and server computer systems. In this research we develop and evaluate a suite of software tools for capturing and compressing traces for multithreaded software called mTrace, which we believe is the first set available. mTrace incorporates the following tools: (i) *mcfTrace* that captures and compresses control-flow traces, (ii) *mlsTrace* that captures and compresses memory referencing traces, (iii) *mcfTRaptor* that captures control-flow traces and compresses them using our TRaptor branch prediction mechanism, and (iv) *mlvCFiat* that captures load value traces and compresses them using our CFiat cache mechanism. The thesis describes the tools' functionality and verification and evaluates their effectiveness by considering trace sizes, execution times, and prediction rates of cache and branch prediction structures for a selected set of benchmarks.

Abstract Approval: Committee Chair ____________________

Department Chair ____________________

Graduate Dean ____________________

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

mlvCFiat ASCII descriptors also include *Thread ID*, *First Access Hit Count*, and *Value*. In Figure 4.36, the ASCII descriptor states thread zero had two first

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This chapter is organized as follows. Section 1.1 gives background and motivation for this thesis. Section 1.2 gives a short overview of the mTrace tool suite developed to enable capturing and storing of program execution traces in multithreaded software. Section 1.3 describes main results of the experimental evaluation of the mTrace tool suite. Section Section 1.4 lists the main contributions of the thesis and Section 1.5 gives an outline of the thesis.

1.1 Background and Motivation

Increasing software complexity and time-to-market contraints have created challenges for system testing and verification. According to the National Institute of Standards and Technology [1], between $22.2 and $59.5 billion are spent nationally because of inadequate software testing infrastructure. One half of the costs are incurred by end users of software through error avoidance and error mitigation activities, and the other half is incured by software developers, reflecting the resources consumed due to inadequate testing methods and tools. The same study found that developers spend an increasing portion of time in software testing and debugging - between 50% and 75% of total development time. Given the ever-increasing sophistication and complexity of software and a market shift toward mutli-core systems, the cost of testing and debugging of software is likely to increase further. These trends underscore a need for better better debugging tools to aide in the software engineering process.

Traditional software debugging is inadequate for real-time systems in avionics, autmotive, or military applications because software instrumentation imposes contraints on the timing requirements of the system. Software bugs that manifest in real-time systems are not easily reproducible, and softare instrumentation itself may affect the dynamic properties of the sofware being analyzed. Multithreaded software can also create difficult to debug race conditions, where execution is nondeterministic. Hardware based debugging techniques do not suffer from these problems, and allow developers to debug software without the need to modify source code or rebuild the executable. Hardware debugging usually traces out the relevant information from the processor chip to a remote system using a software interface. Hardware debugging, or tracing, often requires large on chip buffers and wide trace ports to effectively trace out large quantites of data in real-time. These hardware requirements are the motivation of this research, which seeks to reduce traces to a minimal size while still allowing full program replaybility, and similarly reduce trace port bandwidth. IEEE provides a standard that defines different classes of hardware debugging for embedded systems [2]. This standard, Nexus 5001, specifies four classes of debugging, with each subsequent class requiring more hardware complexity. Class 1 provides basic run-control, including break points and a mechanism for reading register and memory values. Class 2 includes unobtrusive collection of execution traces in real-time, which provides enough information to recreate the entire execution path of the program. Class 3 includes, in addition to the execution traces of class 2, the collection of memory referencing traces to provide complete replayability of the values and addresses written to and read from memory. Class 4 allows the remote system interrogating the processor core to emulate memory accesses.

This research seeks to create a software tool suite for capturing and compressing program execution traces (classes 2 and 3 of Nexus 5001) for multithreaded software. Whereas a number of software tools exist for capturing program execution traces for single-threaded software, no such tools are readily available for multithreaded software. The main goal of this research is the development and verification of a tool suite to support capturing traces in multithreaded programs.

1.2 mTrace Tool Suite

The mTrace tool suite is a collection of Intel Pin tools that provide a means for collecting execution traces (also called control-flow traces) and memory referencing traces with varying degrees of flexibility. The following four Pin tools are included in mTrace:

- *mcfTrace* – Collects and reports control flow traces consisting of branch instruction trace descriptors for multithreaded software. The address of the branch instruction, target address, and type of branch instruction are reported each time a thread retires a branch instruction.
- *mlsTrace* – Collects and reports memory reference traces for multithreaded software. Each trace descriptor includes the load/store instruction's address, operand address, operand size, and operand value.
- *mcfTRaptor* – Collects and reports a minimal control flow trace for multithreaded software using the *TRaptor* [3] branch prediction structure. Trace descritpors are collected for incorrectly predicted branch instruc-

tions, reducing the total trace size needed for complete program replayability.

- *mlvCFiat* – Collects and reports a minimal load value trace for multithreaded software by utilizing the *CFiat* [4] cache access mechanism to reduce the total trace size needed for program replayability. Trace descriptors are collected whenever a cache block is evicted or an operand in a cache block is referenced for the first time.

Each of these four tools uses a variety of parameters that modify the scope of the trace, how tracing occurs, and how the trace is saved. The first two tools, *mcfTrace* and *mlsTrace*, were motivated by a need to inspect general properties of control-flow and memory reference traces for multithreaded software, while the last two, *mcfTRaptor* and *mlvCFiat*, were motivated by the need for hardware tracing techniques to reduce trace sizes and trace port bandwidths. Each tool generates a trace file for a target binary (and any shared libraries it uses), and a statistics file that characterizes the trace execution.

1.3 Results

The mTrace tools are fully tested and verified on a standard set of parallel benchmark programs. We evaluate the effectiveness of the mTrace tools by considering trace file size and the time needed to capture and store traces as a function of the number of software threads. Each trace tool supports an optional general-purpose compression of captured traces before they are written to the secondary storage. To evaluate compressability of individual traces, we measure compression ratio achieved by general-purpose compressors.

For *mcfTRaptor* and *mlvCFiat* tools, we analyze the effectiveness of predictor and cache structures employed by measuring misprediction and cache miss rates. In addition, we analyze two different organizations of *TRaptor* and *CFiat* structures: private in which each software thread owns a prediction structure and shared in which multiple software threads share one structure. Our experimental evaluation indicates that a private organization of branch prediction and cache structures results in smaller control-flow and load value traces when compared to the she shared organization.

1.4 Contributions

This thesis makes the following contributions to the field of software binary instrumentation and tools for trace capture and compression:

- Developed and tested tools for capturing and storing program execution traces of multithreaded software, specifically:
    - *mcfTrace*: a tool for capturing and compressing control-flow traces;
    - *mlsTrace*: a tool for capturing and compressing data traces;
    - *mcfTRaptor*: a tool for capturing and compressing control-flow traces using our TRaptor mechanism;
    - *mlvCFiat*: a tool for capturing and compressing data traces using our CFiat mechanism.
- Performed experimental evaluation of the mTrace tools using SPLASH-2 benchmark suite while varying the number of threads.
- Created a public repository of the mTRace tools and traces available at: http://lacasa.uah.edu/portal/index.php/software-data/32-mtrace-tools-and-traces.

1.5 Outline

The outline of this thesis is as follows: Chapter 2 introduces software tracing, tracing techniques, and future challanges and opportunities. Chapter 3 summarizes the related work and the current state-of-the-art in the field of software and hardware tracing. Chapter 4 describes the mTrace tool suite, summarizes their implementation, and lists the steps taken to verify their behavior. Chapter 5 explains the experimental methodology used to evalute the mTrace tools for a set of benchmarks. Chapter 6 gives the results of the experimental evaluation and Chapter 7 gives concluding remarks.

CHAPTER 2

BACKGROUND

Software tracing provides software developers with detailed information on the dynamic run-time behavior of software at the image, sub-routine, basic block, or instruction level. Because tracing occurs at a lower level of abstraction and can generate billions of records per second, tracing imposes performance constraints during collection and requires large amounts of storage. This chapter covers the background of several aspects of this research. Sections 2.1 and 2.2 describe control flow and memory reference traces and their applications, respectively. Section 2.3 relates the problems of debugging embedded and real-time systems to tracing. Lastly, Section 2.4 explores the challenges faced in this research and opportunities to pursue in the future.

2.1 Control Flow Traces

Control-flow traces are widely used in software debugging, trace-driven architectural simulation (e.g., branch predictor studies), performance optimization and tunning, and workload characterization [5]. Control flow traces of a program running on a processor are created by recording the addresses of the instructions in the order they are executed. Each instruciton executed results in a single record in the control-flow trace. Modern processors may execute billions of instructions per second, generating a vast amount of information that needs to be captured, communicated, and stored. In modern multi-cores, that include a dozen processor cores, the amount of information captured in control-flow traces is even larger. The performance and storage overheads associated with trace capture make such tracing feasible only on small program segments and impractical and cost-prohibitive for the entire program.

Depending on the intended trace use, control flow traces can be modified to include fewer but sufficient number of records. For example, in software debugging the goal is to faithfully replay a program's execution offline in software debugger. By analyzing the actual control-flow captured on a host machine and comparing it with the expected one, software developers can quickly locate sources of software bugs. However, to recreate a program's flow, one does not need to record the address of every single instruction executed. Providing that the software debugger has access to program's executable, we need to record only changes in the program flow. These changes are caused by either control-flow instructions or exceptions. When a change in the program flow occurs, we need to record the program counter (PC) of the currently executing instruction and the branch target address (BTA) in the case of a control-flow instruction or the exception-handler target address (ETA) in the case of an exception. The format of trace records can be further modified to require fewer bits for encoding. For example, the number of instructions executed in dynamic basic blocks may replace the program counters, or the target addresses of direct branches can be omitted from the trace because they can be inferred by the software debugger from the program executable.

Other types of control-flow traces may require more trace records or fewer trace records. For example, control-flow traces intended to be used in branch predictor studies require one trace record per control-flow instruction, regardless of its outcome. In multithreaded software, we may need to include additional information such as thread identification that further qualifies each trace record. In some cases, the time stamp or the processor core identification may be included in the trace.

2.2 Memory Reference Traces

Memory reference or data traces contain information recorded from instructions that read from memory or write to memory in the order in which they occur during program execution. Typically, one trace record contains relevant information on a single memory-referencing instruction, such as the program counter and information about memory operands. For each memory operand, we may record (i) the type of memory operation (read or write), (ii) operand address in memory, (iii) size of the operand in bytes, and (iv) a data value read from memory or written to memory. Other information may be included as well, including thread identification in case of multithreaded software, timestamps for the read or write operation, or processor core identification. The format of trace records depends on trace uses and they may contain all or a subset of the fields described above. Regardless of the exact format of trace records, capturing memory reference traces incurs very high performance and storage overheads.

Similar to control flow traces, memory reference traces can be used for software debugging, performance optimization and tunning, workload characterization, and architectural simulations targeting memory subsystem and cache hierarchies. For example, load value traces, traces that contain data values read from memory, can be used in software debugging. Whereas control-flow traces support reconstruction of the program's control flow only, load value traces enable under certain conditions a complete replay of the executed program. These conditions assume that the software debugger includes an instruction set simulator, has access to the program binary, can access the control-flow traces containing exception records, and can access to the load value traces [4]. Data address traces captured in real-time are of special interest in multi-core systems as they offer valuable information about shared memory access patterns and possible data race conditions.

2.3 Tracing in Embedded and Multi-core Systems

Software developers for server and desktop applications often rely on binary instrumentation tools, software development environments, and software debuggers to debug and trace program execution. For example, software developers may set breakpoints, examine the content of registers and memory at breakpoitns, or step through the program one instruction at a time. Setting breakpoints and examining the processor state to locate difficult and intermittent bugs in large software projects is demanding and time-consuming. Alternatively, developers can collect program execution traces that are analyzed to diagnose program segments where bugs arise faster. These software development environments may require minimal or no hardware support. However, common to all these methods are that they are obtrusive – the program execution in the debug mode differs from the “native” program execution when no debugging is involved. Whereas this interference may not pose challenges during software development for desktop and server applications, it is often signicficant problem in embedded systems, especially real-time systems.

Embedded software developers face a unique set of challenges. These challenges are driven by both technology and market forces and include: (i) a growing level of sophistication of embedded software with multi-layered software stacks, (ii) increased levels of on-chip integration that limit the visibility of internal modules, (iii) high operating frequencies, (iv) limited input/output bandwidths to and from systems-on-a-chip, and (v) shrinking time-to-markets. Setting a breakpoint is often not practical in debugging real-time embedded systems; e.g., it may be harmful for hard drives or engine controllers. In addition, debugging through breakpoints interferes with program execution. The order of events during debugging may deviate from the order native execution; this deviation can cause original bugs to disappear in the debug run.

To meet these challenges and get reliable and high-performance products to market on time, embedded software developers increasingly rely upon on-chip resources for debugging and program tracing. However, even limited hardware support for debugging and tracing is associated with extra cost in chip area for capturing and buffering traces, for integrating these modules into the rest of the system, and for sending out the information through dedicated trace ports. These costs often make system-on-a-chip (SOC) designers reluctant to invest in additional chip area solely devoted to debugging and tracing.

The IEEE's Industry Standard and Technology Organization has proposed a standard for a global embedded processor debug interface (Nexus 5001) [2]. This standard specifies four classes of operation – higher numbered classes progressively support more complex debug operations but require more on-chip resources. Class 1 provides basic debug features for run-control debugging, including single-stepping, breakpoints, and access to processor registers and memory while the processor is not running. Class 1 is traditionally implemented through a JTAG interface. However, this approach is time-consuming and obtrusive; it interferes with the dynamic runtime behavior of the program and can cause original bugs to disappear. More importantly, it is not applicable to debugging real-time embedded systems where setting breakpoints is simply not an option. Class 2 provides debug support for nearly unobtrusive capturing and tracing program execution (control-flow) in real-time. Class 3 provides support for memory and I/O read/write tracing in real-time, while Class 4 provides resources for direct processor control through the trace port.

Many embedded processor vendors have developed modules with advanced tracing and debugging capabilities and integrated them into their embedded platforms, e.g., ARM's Embedded Trace Macrocell [6], MIPS's PDTrace [7], and OCDS from Infineon [8]. The trace and debug infrastructure on a chip typically includes logic that captures address, data, and control signals, logic to filter and compress the trace information, buffers to store the traces, and logic that emits the content of the trace buffer through a trace port to an external trace unit or host machine. In this paper we focus on data traces (Class 3 operation in Nexus).

Existing commercially available trace modules rely either on hefty on-chip buffers to store execution traces of sufficiently large program segments, or on wide trace ports that can transfer a large amount of trace data in real-time. However, large trace buffers and/or wide trace ports significantly increase the system complexity and cost. Moreover, the number and speed of I/O pins dedicated to tracing cannot keep pace with the increase in the speed and the number of processor cores and their speed. These challenges are even more important in multi-core systems.

The mTrace project [9] involves developing the next generation of trace compression methods and infrastructure to make continuous, real-time, unobtrusive, and cost-effective program, data, and bus tracing possible in embedded systems. The approach relies on on-chip hardware to record the processor state and corresponding software modules in the debugger.

The goal of this thesis is to develop of a set of tools for collecting execution traces (also called control-flow traces) and memory referencing traces with varying degrees of flexibility and enable further research in the next generation of hardware-supporting tracing and debugging in embedded systems.

2.4 Challenges and Opportunities

Descriptor orderings in a trace file may differ from run to run for multi-threaded programs because the order in which trace descriptors are serialized to a trace file is not enforced. Each control-flow or memory reference trace collected by an mTrace tool can be used to reconstruct a thread's execution path. However, the relative timing between each thread is not recorded, and a reconstruction of the execution path from the trace does not accurately describe the order of execution between each thread. Certain aspects of dynamic program behavior may change for a single-thread program as well. The operating system may choose different virtual addresses for the stack, heap, and code sections of a program. A shared library may be loaded into a different address and operating system signals may not occur at the same point between execution runs. Furthermore, the behavior of a system call is often a function of the operating systems current state, which can vary. mTrace does not guarantee that control trace and memory reference descriptor orderings will reflect the actual execution and memory refence orderings that occurred at run time.

PinPlay [10] is a set of Pin tools that track thread execution and saves execution instances for *deterministic record-replay*, where the dynamic run time behavior of a program is exactly reproduced in subsequent executions. PinPlay is composed of a logger which records execution of a program to a file called a *pinball*, and a replayer that uses the *pinball* to repeat the captured execution. Other Pin tools can be integrated with PinPlay to correctly capture the dynamic program behavior of multi-threaded software. PinPlay could be integrated with the mTrace tool suite to enforce correct descriptor orderings for multithreaded programs. PinPlay can also solve a performance issue in mTrace. Currently, instructions that write to memory must be protected with a lock, as the act of executing the store instruction and inspecting the memory address that it wrote to is not atomic – a different thread could write to that address before it is inspected. PinPlay removes the need for this lock by redirecting the store value before the instruction is executed.

CHAPTER 3

RELATED WORK

This chapter describes related work in the area of unobtrusive program tracing schemes and software-based trace compression (Sections 3.1) and hardware-based trace compression (Section 3.2).

3.1 Software Trace Compression

A number of software-based trace compression algorithms have been proposed, including PDATS [11] [12], WPP [13], N-tuple [14], and more recently VPC [15], and SBC [5]. The VPC trace compression algorithms [15] are a set of value prediction based algorithms. Each algorithm builds on the success of the previous alogorithm, with VPC1 compressing raw traces with value predictors and VPC2 adding a second compression stage. Most VPC algorithms use value predictors to convert traces into more compressable streams. VPC3 converts raw traces into streams, allowing for a higher compression ratio and faster compression time. VPC4 is the result of optimizations performed on VPC3's predictor table replacement policy and hash function. VPC4 compresses 36 times better, and compresses 53 times faster than *bzip2*.

A single-pass stream-based compression (SBC) technique [5] was designed and shown to have a compression ratio between 18 and 308 for a subset of the CPU2000 benchmark suite. SBC maintains a relation between instruction addresses and unique instruction streams to they they belong. An instruction stream is a block of consecutively executing instructions, and the compressed instruction trace contains a list of indentifiers for each of these streams. Data traces are captured by recording the data address and number of accesses in each stream. SBC can be implemented in hardware for minor resource and compression ratio trade off.

TCgen [16] is a tool that generates high-performance trace compressors. The user provides a description of the trace format and TCgen translates the specification to an optimized compressor using a selection of value predictors. TCgen is able to use last-value predictors, finite-context-method predictors, and differential-finite-context-method predictors. In addition to a value predictor configuration, TCgen requires a description of the program traces in extended Backus-Naur form. TCgen was tested on a subset of the SPECcpu2000 benchmark suite and was found to outperform VPC3 between 6% to 13%.

3.2 Hardware Trace Compression

Hardware trace compression methods usually include architectural extension to the CPU to filter out redundant or unnecessary trace descriptors, before emitting the trace descriptors to a remote system for debugging and replayability. A similar extension is usually maintained in software to keep the state of the debugger consistent with the hardware enhancements. In this section, we summarize proposed hardware techniques for compressing program traces.

Program stream caches and last stream predictors [17] have been proposed as hardware enhancements to filter trace descritpors by exploiting program characteristics. Each basic block is uniquely identified by its starting address (SA) and starting length (SL). In this case a trace descriptor is the pair (SA, SL). A stream detector interrogates the processor's control signals to check when a new program stream is encountered or an exception occurs. A stream descriptor buffer then serializes access to a stream descriptor cache (SDC), which is indexed by the XOR of the stream address and string length. In the event of a cache hit, the set index and way index for the descriptor are sent to the last stream predictor (LSP), which is a simple last event predictor. In the event of a miss a block is evicted in accordance with the replacement policy and the entry is updated. In the event that the LSP makes an incorrect prediction, the set and way indexes are sent to an encoder, which emits a descriptor to the remote debugger. The MiBench [18] benchmark was the target of performance analysis and showed that for a 32 entry SDC, the bits per instruction can vary between .001 (*adpcm_c*) and 1.377 (*ghostscript*).

The Double-Move-To-Front method (DMTF) [19] is a hardware method that uses basic block properties (such as basic block length) to reduce trace sizes. As the name suggests, DMTF makes use of two Move-To-Front [20] transformations, which is used in the popular compression software *bzip2*. DMTF is designed with two history tables containing basic block length and sizes. When a stream is encountered the first table, *mtf1*, is searched for a matching stream address and length. If it is not found, the entries are shifted up, the basic block address and length are inserted into the last entry, and a descriptor is traced out. When a basic block is found in the first table the second table is searched in a similar manner. When a miss occurs in this second table, *mtf2*, the table entry number that the basic block resides in *mtf1* is traced out and saved to *mtf2*. When the correct index is found in *mtf2*, the *mtf2* index is traced out. Decompression is a reversed compression process and occurs in software. Performance analysis for the DMTF method on the MiBench [18] benchmark showed that compression ratios were between 45 (*fft*) and 1738 (*adpcm_c*) for a 128 entry *mtf1* and a 4 entry *mtf2*. In addition, a last value predictor was used for the upper 12 bits of the address and a zero hit counter for *mtf2* hit events to decrease descriptor lengths.

TRaptor [3] is a hardware mechanism that reduces the number of trace records required for program replayability through a remote software debugger. TRaptor reduces the number of traces collected a sufficient amount by utilizing a branch outcome predictor, *gshare*, and a branch target predictor implemented with an indirect branch target buffer and a return address stack. The *gshare* outcome predictor is organized as an array of two bit adaptive predictors, where each entry is accessed using a function of the branch instruction address and a path information register (PIR) which records the outcomes of previous branches. The return address stack stores the return target address for instructions that return from a subprocedure. The indirect branch target buffer saves the target address for branch instructions whose target address is not inferrable from the branch instruction. Instead of emitting a control flow descriptor for each branch instruction, TRaptor records the number of correctly predicted branchs with the parameter *bCnt*, and emits a control flow trace descriptor only for incorrectly predicted branches or exceptions. Exceptions require a separate parameter, *iCnt*, which is incremented for each instruction and is reset if an exception or a branch misprediction occurs. The TRaptor structure is organized to incercept the instruction type, branch instruction address, and branch target address from the target CPU and encode control flow descriptors, when necessary, and send them to a remote host, where an equivalent TRaptor structure in software enables debugging of the target binary. Figure 3.1 contains the algorithm used by TRaptor when presented with a branch instruction. *iCnt* is incremented for every instruction (line 2) and *bCnt* is incremented for all branches (line 3). If a prediction is incorrect, a trace descriptor is emitted (lines 6-7), and both *iCnt* and *bCnt* are reset. If an exception occurs, a trace is emitted (line 13-14) and both parameters are reset. For multithreaded software, a TRaptor structure can be allocated privately to each thread or shared globally amongst all threads.

```
1.  // For each committed instruction in Thread with index i
2.  i.iCnt++;  // increment iCnt
3.  if ((i.iType==IndBr) || (i.iType==DirCB)) {
4.    i.bCnt++;  // increment bCnt
5.    if (TRaptor mispredicts) {
6.      Encode mispredicton event;
7.      Place record into the Trace Buffer;
8.      i.iCnt = 0;
9.      i.bCnt = 0;
10.   }
11. }
12. if (Exception event) {
13.   Encode an exception event;
14.   Place record into the Trace Buffer;
15.   i.iCnt = 0;
16.   i.bCnt = 0;
17. }
```

Figure 3.1 TRaptor Operation for One Thread (Private/Shared)

While originally not concerned with multithreaded software, Figure 3.2 depicts how TRaptor strutures can be allocated to each thread privately. Each thread accesses its TRaptor mechanism through its thread ID and presents, depending on the branch type, the instruction address and branch target address. Each thread can accesss a private gshare, return address stack, and indirect branch target buffer. The *bCnt* and *iCnt* parameters are also private to each thread.

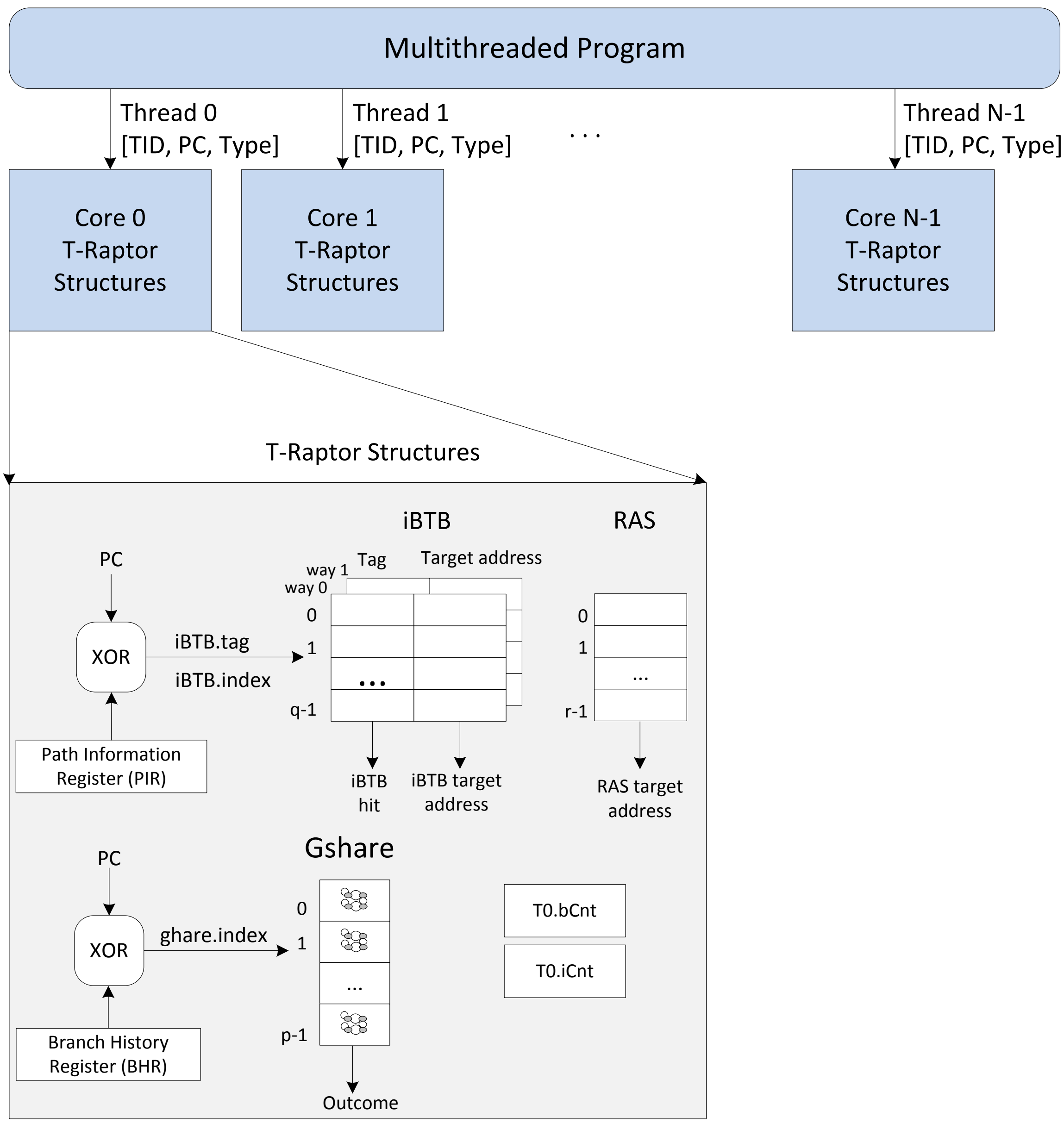

Figure 3.2 *mcfTRaptor* with Private Predictor Structures

Figure 3.3 depicts TRaptor saring among threads in a multithreaded program. Each access is sequential, with each thread sending the instruction address and branch target to the shared TRaptor structure. The *gshare*, return address stack, and indirect branch target buffer are shared among all threads, but the *bCnt* and *iCnt* parameters are private to each thread, allowing off-line program replayability for each thread.

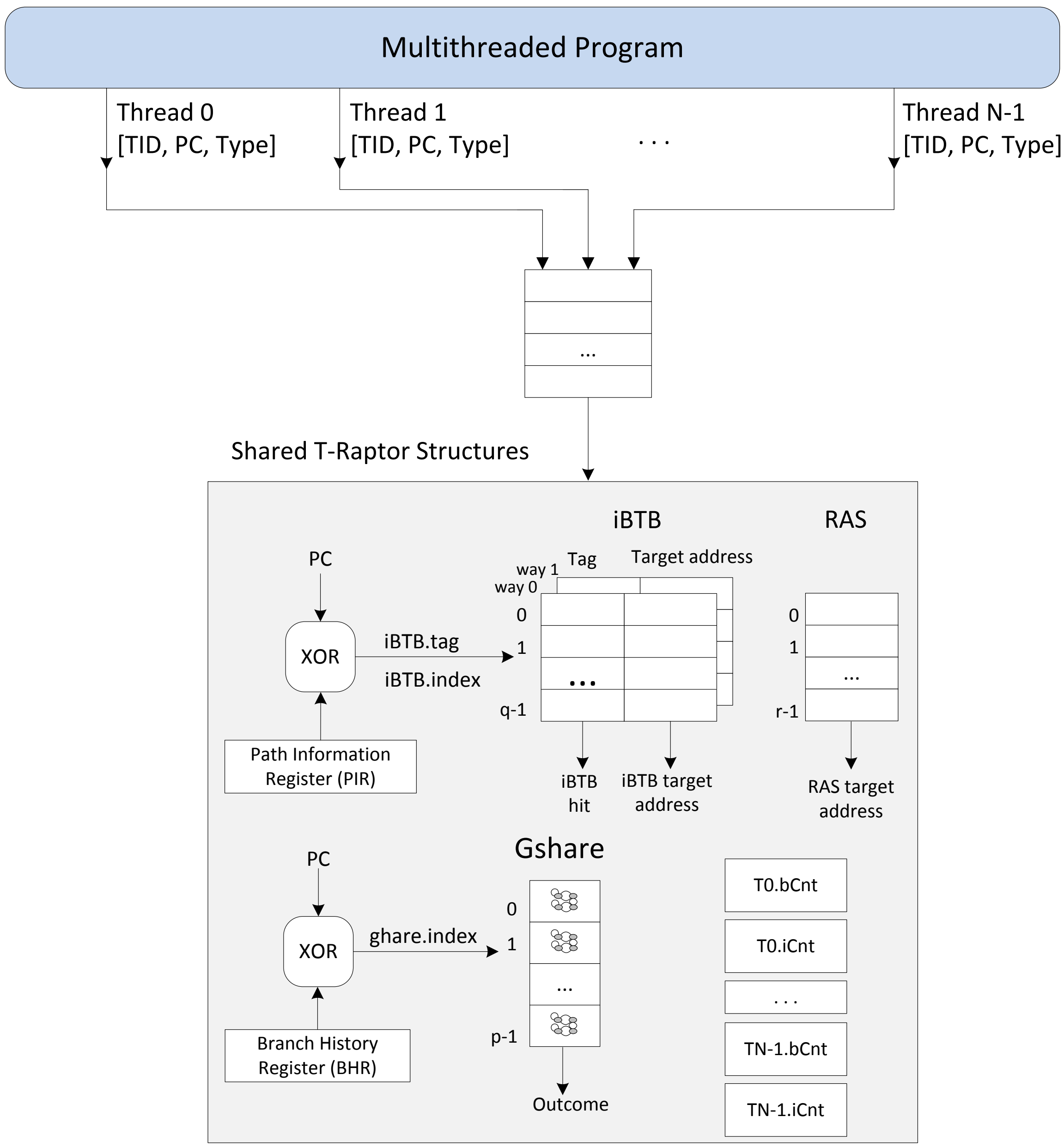

Figure 3.3 *mcfTRaptor* with Shared Predictor Structures

CFiat [4] is a hardware-based mechanism that reduces load value traces by collecting a minimal set of load value trace descriptors through the use of a cache first access mechanism. The CFiat, or cache first access, mechanism emits load value descriptors on the first hit or the eviction of a cache block. The CFiat mechanism extends an already existing data cache with first access flags that protect the operands in each cache block. An operand's first access flag is set to one whenever a trace descriptor is emitted for the operand or when the operand is written to memory. Whenever a cache block is evicted, all flags associated with that cache block are set to zero. Whenever a cache hit occurs and the flags associated with the operand are found to be set to one, the *fahCnt* parameter is incremented. This parameter allows for accurate replaying of traces in an off-line debugger. The size of the operand that a flag can protect is refered to as the flag granularity and is a design parameter.

Figure 3.4 lists the cache first access algorithm. Each operand passes through the cache first access mechanism, and if it results in a cache hit, the flags associated with the operand are checked (line 3). If the flags are set, *fahCnt* is incremented. If the flags are not set, a trace descriptor is eimitted, the flags corresponding to that operand are set, and *fahCnt* is reset. In the event of a cache miss (line 10), all of the flags associated with cache block are reset, a trace descriptor is emitted, the flags associated just with that operand in the newly retrieved cache block are set, and *fahCnt* is reset.

```
// For each retired load that reads n bytes in thread i
if (CacheHit) {
 if (corresponding n FA flags are set)
        i.fahCnt++;
 else {
        Emit trace record into Trace Buffer (tid, fahCnt, loadValue);
        Set corresponding n FA flags;
        i.fahCnt = 0;
 }
} else { // cache miss event
        Clear FA bits for newly fetched cache block;
        Perform steps 5-7;
}
// For each retired store that writes n bytes
Set corresponding n FA bits;
// For external invalidation/update request
Clear FA bits for entire cache block
```

Figure 3.4 CFiat Operation for One Thread (Private/Shared)

Much like TRaptor, CFiat is organized as a hardware extension, in this case to a data cache. The mechanism emits the encoded load value descriptors to on-chip buffers and trace ports were transmitted to trace probe and host machine, where a software copy of the CFiat mechanism is located. This host machine can replay the program of the target binary. Figure 3.5 depicts the organization of the cache mechanism, with each thread allocated with a private data cache and set of first access flags. Each thread accesses its data cache and first access flags independently and emits trace descriptors when the conditions are met. The threads present the memory referencing instruction's address (PC), the operand address (DA), the operand size (DS), type (read or write), and data value (DV).

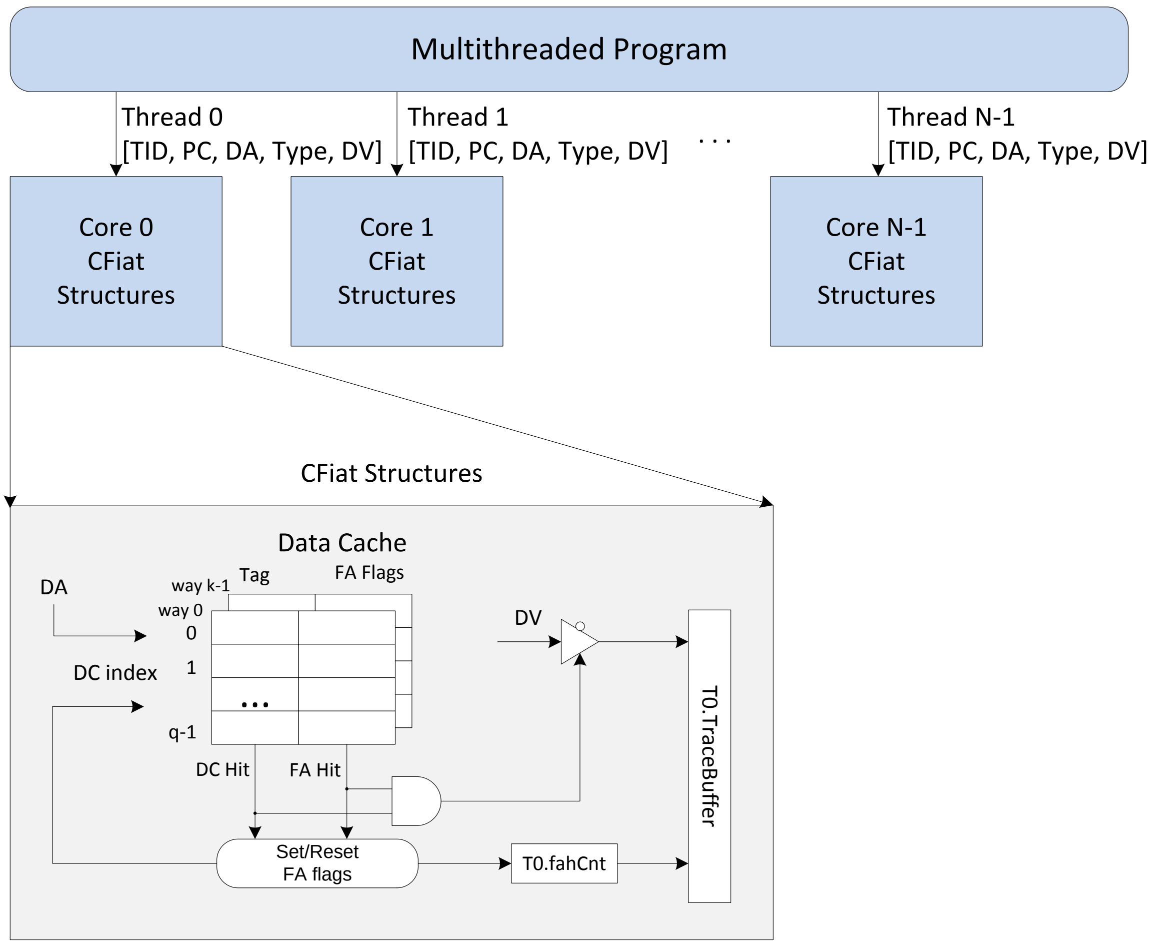

Figure 3.5 *mlvCFiat* with Private Cache Structures

Figure 3.6 depicts sharing of data cache and cache first-access structures among threads in a multithreaded program. Each access is sequential, with each thread sending the instruction address and branch target to the shared data cache. The data cache and first-access bits are shared among all threads, but the *fahCnt* is private to each thread, allowing offline program replayability for each thread.

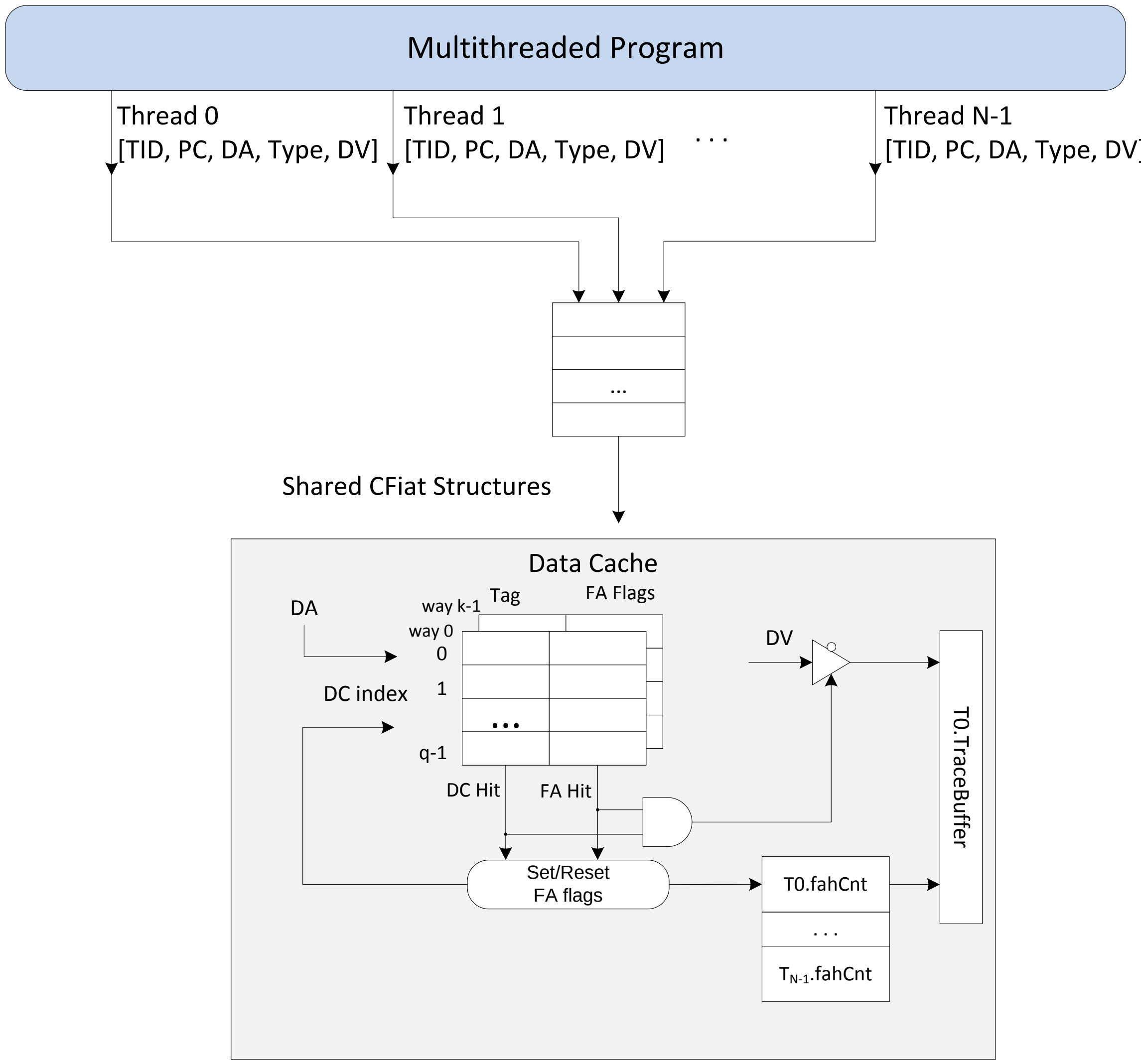

Figure 3.6 *mlvCFiat* with Shared Cache Structures

CHAPTER 4

MTRACE TOOL SUITE

This chapter introduces a set of software tools for capturing and compressing program traces of multithreaded programs, including both control flow and data traces. The mTrace tool suite runs on systems that use the Intel-64/x86 instruction set architectures and relies on Intel's Pin binary instrumentation tool to capture traces. The mTrace suite encompasses the following tools

- *mcfTrace*: a tool for capturing and compressing control-flow traces (Section 4.1);
- *mlsTrace*: a tool for capturing data traces, specifically memory referencing load and store instructions (Section 4.2);
- *mcfTRaptor*: a tool for capturing control-flow traces and compressing them using our *T-Raptor* mechanism that exploits branch predictor structures (Section 4.3);
- *mlvCFiat*: a tool for capturing load value data traces and compressing them using our *C-fiat* mechanism that relies on caches and first-access bits (Section 4.4).

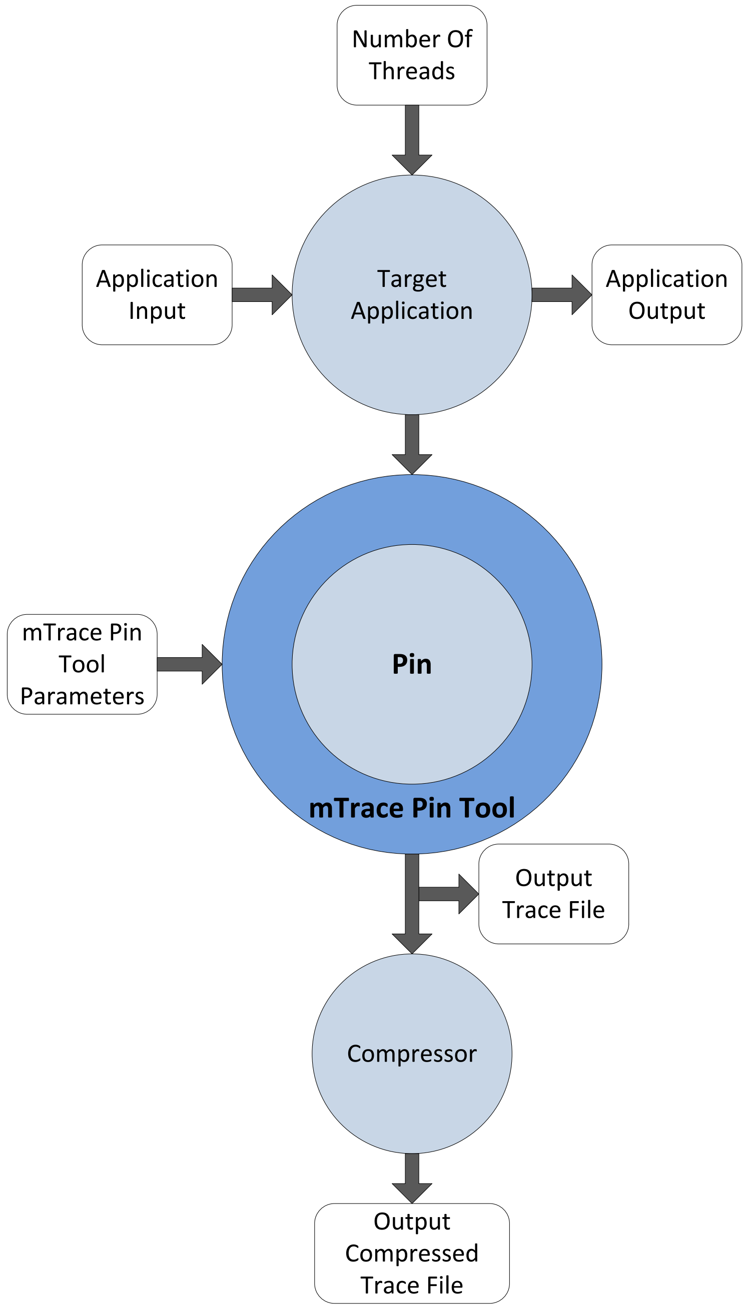

Figure 4.1 *mcfTrace* organization

Figure 4.1 shows the software organization that is shared by all mTrace tools. Starting from the top, the target application is specified (e.g., a multithreaded MatrixMultiply program) with its input and output parameters, including the number of threads (e.g., in MatrixMultiply we specify the matrix size and the number of threads). We designed the mTrace tools to support a number of parameters for controlling program tracing (mTrace Pin Tool Parameters). To accommodate a wide range of trace uses, we allow users to specify which segment of the target application to trace. This is achieved by specifying the number of instructions executed by the target application before the tracing is turned on. The length of the traced segment is controlled by specifying the number of instructions to be traced. In addition, the user can select the format of trace descriptors to be either binary or ASCII text. Other optional parameters allow the user to specify whether the trace descriptors are written directly to an output trace file or go to a general-purpose compressor to be compressed before writing into a compressed trace file. The subsections below describe individual trace tools. For each trace tool, we first give its functional description, then describe high-level implementation details, and finally discuss test steps taken to verify the correctness of our implementation.

4.1 mcfTrace

mcfTrace is a Pin tool designed to collect and save control-flow traces of multithreaded programs to a file. For each control-flow instruction, *mcfTrace* captures a trace descriptor that consists of the following: a logical thread ID of the issuing thread, the address of the instruction, the branch target address, the type of the control-flow instruction, and its outcome. The trace descriptors can be saved to a binary file or text file, or piped to a general purpose compressor. Section 4.1.1 gives a functional description of the *mcfTrace* tool. Section 4.1.2 gives a brief description of tool implementation, and Section 4.1.3 describes verification process and test programs used.

4.1.1 Functional Description

Table 4.1 lists the *mcfTrace* tool parameters that allow a user to control instrumentation and tracing. These parameters are used to control the following: (a) the trace file type (binary or ASCII), (b) the code segment and trace scope at the instruction and sub-procedure level, (c) optional compression (d) the maximum trace size, and (e) others.

Table 4.1 *mcfTrace* Parameters

Parameter	**Description**
-a	Saves trace descriptors in an ASCII file (default is binary)
-c <COMPRESSOR>	Trace descriptors are piped to a general-purpose compressor before saving. <COMPRESSOR> = {bzip2, pbzip2, gzip, pigz}
-d	Each descriptor includes a corresponding assembly code
-f	Trace file size limit in Megabytes. Instrumentation and trace collecting stops after reaching this limit (default limit is 50 GBytes).
-filter_no_shared_libs	Traces only target binary, shared libraries are not traced.
-filter_rtn <routine>	Tracing only occurs in a specified routine(s).
-[h | help]	Displays help message with all parameters and their description.
-l <NIST>	Specifies NIST, the number of instructions that will be instrumented in the target.
-o <FNAME>	Specify trace file name, FNAME.
-s <NIST>	Specifies NIST, the number of instructions to be skipped before instrumentation begins.

Figure 4.2 illustrates the format of the descriptors collected by *mcfTrace*. A *mcfTrace* binary trace descriptor includes the following fields:

- *Thread ID* field is 1 byte long and encodes threads from 0 to 255;
- *Instruction Address* and *Target Address* fields that are 8 bytes long on 64-bit architectures include the instruction address and the branch target address, respectively; and
- *Type & Outcome* field encodes the type of the control-flow instruction and its outcome (taken or not taken). The Intel-64 ISA supports the following branch types: unconditional indirect (Type & Outcome=0), unconditional direct (Type&Outcome = 1), conditional direct taken (Type&Outcome = 2), and conditional direct not taken branches (Type&Outcome = 3).

Except for address sizes (which depend on the system's address size), binary descriptors do not have any variable fields and a binary file can be easily decoded by applying this descriptor format. For Intel-64 architectures, a *mcfTrace* binary descriptor uses exactly 18 bytes.

mcfTrace ASCII descriptors also include *Thread ID, Instruction Address*, *Target Address*, *Type&Outcome* fields, as well as optional assembly code. Individual fields in a descriptor are separated by a comma and individual descriptors are separated by a new line character. Figure 4.2 gives an example of an ASCII descriptor, which specifies that thread 0 issued an instruction at address 0x0000003f_83200b03, and that the instruction is an unconditional direct branch (U, D, T) with the target address 0x0000003f_83201130. In this case we opted to print out the assembly instruction for the descriptor, which is a call instruction.

mcfTrace Descriptor: Binary Format

Thread ID (1 Byte)	Instruction Address (8 Bytes)	Target Address (8 Bytes)	Type&Outcome (1 Byte)

mcfTrace Descriptor: ASCII Format

Thread ID (up to 4 Bytes)	Instruction Address (20 Bytes)	Target Address (20 Bytes)	Type&Outcome (8 Bytes)	Assembly Code (Variable)

Example: 0, 0x0000003f83200b03, 0x0000003f83201130, U, D, T call 0x3f83201130

Figure 4.2 *mcfTrace* Descriptor Formats: Binary (top) and ASCII (bottom).

Figure 4.3 contains an example output from *mcfTrace*. In this example, *mcfTrace* creates the trace file, mcfTrace.out2013_8_31_15.4.1.txt, as well as a text file, mcfTrace.out2013_8_31_15.4.1.Statistics, which contains statistics relating to the branch trace descriptors that are captured. The user can specify an output trace file name or the file name is generated automatically using a time stamp. A selected segment of the output trace file is shown in lines 9-19. The statistics file contains information about the number and types of individual branch instructions as shown in lines 2-8.

```
[myersar@EB245-mhealth3 ManualExamples]$ head mcfTrace.out2013_8_31_15.4.1.Statistics
mcfTrace: Traced 1000000 instructions
mcfTrace: Skipped 3000000 instructions
mcfTrace: Recorded 269334 control transfer instructions.
    4968 ( %1.84 ) Unconditional Direct
    129517 ( %48.09 ) Conditional Direct Taken
    131116 ( %48.68 ) Conditional Direct Not Taken
    3733 ( %1.39 ) Unconditional Indirect
[myersar@EB245-mhealth3 ManualExamples]$ head mcfTrace.out2013_8_31_15.4.1.txt
1, 0x00007f5a40996bbe, 0x00007f5a40996be4, C, D, NT
1, 0x00007f5a40996bc9, 0x00007f5a40996bb8, C, D, T
1, 0x00007f5a40996bbe, 0x00007f5a40996be4, C, D, NT
2, 0x00007f5a40996bc9, 0x00007f5a40996bb8, C, D, T
1, 0x00007f5a40996bc9, 0x00007f5a40996bb8, C, D, T
1, 0x00007f5a40996bbe, 0x00007f5a40996be4, C, D, NT
1, 0x00007f5a40996bc9, 0x00007f5a40996bb8, C, D, T
3, 0x00007f5a40996bc9, 0x00007f5a40996bb8, C, D, T
1, 0x00007f5a40996bbe, 0x00007f5a40996be4, C, D, NT
2, 0x00007f5a40996bbe, 0x00007f5a40996be4, C, D, NT
```

Figure 4.3 *mcfTrace* Example Output

4.1.2 Implementation Details

mcfTrace instruments applications at the instruction level by recompiling basic blocks on a just in time basis with analysis routines that are inserted before branch instructions. *mcfTrace* collects branch instruction data by passing the logical thread ID of the executing thread, the address of the branch instruction, its target (whether static or indirect), the type of branch instruction, and branch outcome as arguments to these analysis routines.

The Intel 64 and IA-32 instruction set [21] control transfer instructions include conditional and unconditional jump instructions, a subroutine call instruction, and a subroutine return instruction. Table 4.2 depicts the three classifications used by *mcfTrace* when collecting descriptors.

Table 4.2 Intel 64 and IA-32 Control Transfer Instruction Classification

	Operand	**Instruction Mnemonics**
Conditional Direct	Memory	jnbe, jnb, jb, jz,, etc. loop, loope, loopne, etc.
Unconditional Direct	Memory	jmp, call,
Unconditional Indirect	Register Indirect, Memory	jmp, call, rtn

The j* and loop* instructions use labels which reference addresses that are generated by a linker and are considered static since they do not change during run-time. These two groups of instructions are also conditional and use condition codes kept in the status registers. Both the *jump* and *call* instructions can either use labels or registers to specify the target address, thus can be classified as either unconditional direct or unconditional indirect control instructions. The *rtn* instruction uses a target referenced by a stack register and is considered indirect.

Figure 4.4 shows the code segment in mcfTrace.cpp that instruments a target to capture control-flow traces and write trace descriptors to an ASCII file. mcfTrace.cpp contains routines that instrument the target and perform other housekeeping roles such as initializing Pin and detaching Pin from the target. Similar instrumentation code is used when writing to a binary file. The Pin instruments over basic blocks (line 1) and then iterates over individual instructions within the basic block (line 3). Line 5 of the code inserts the `SetFastForwardAndLength` analysis procedure that counts the number of instructions executed in the target. This procedure allows us to implement fast forwarding and trace length control functions. If we are fast forwarding, this analysis function simply counts the number of instructions left to skip until tracing begins. If we are tracing, `SetFastForwardAndLength` counts the number of instructions executed while tracing. Lines 8-23 use Pin calls to filter the different branch instruction classes as described in Table 4.2. The `HasFallThrough` Pin function is true for instructions that potentially do not change control flow and can be used to decide between conditional and unconditional branches. `IARG_THREAD_ID` passes the logical ID of calling thread, `IARG_THREAD_PTR` passes the address of branch instruction, `IARG_BRANCH_TARGET_ADDR` passes the target of the branch instruction, and `IARG_BRANCH_OUTCOME` passes whether or not the branch was taken (only used for conditional branches).

```
for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl) )
{
   for(INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins) )
   {
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)SetFastForwardAndLength,
                         IARG_THREAD_ID, IARG_END);

      if( INS_IsDirectBranchOrCall(ins) && !INS_HasFallThrough(ins) )
         INS_InsertCall(ins, IPOINT_BEFORE,
                   (AFUNPTR)Emit_UnconditionalDirect_ASCII,
                   //Args: Thread ID, Instruction Address, Target Address
                   IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                   IARG_END);

      // Is Conditional and Direct
      else if( INS_IsDirectBranchOrCall(ins) && INS_HasFallThrough(ins) )
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Emit_ConditionalDirect_ASCII,
                   //Args: Thread ID, Instruction Address, Target Address, Taken?
                   IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                   IARG_BRANCH_TAKEN, IARG_END);

      else if( INS_IsIndirectBranchOrCall(ins) || INS_IsRet(ins) )
         INS_InsertCall(ins, IPOINT_BEFORE,
                   (AFUPTR)Emit_UnconditionalIndirect_ASCII,
                   //Args: Thread ID, Instruction Address, Target Address
                   IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                   IARG_END);
}
```

Figure 4.4 *mcfTrace* Instrumentation Implementation from mcfTrace.cpp

Figure 4.5 contains an example of an analysis routine found in mcfTrace.h, which only contains analysis routines injected with mcfTrace.cpp. This routine passes pertinent branch instruction data to a buffer which is later written to a binary file. The `CanEmit` (line 4) function returns early if tracing is not enabled and will detach *mcfTrace* from the target process if the tracing is finished or the file size limit is reached. Lines 6-21 create the binary trace descriptor from the information passed during instrumentation, and lines 24-28 push the descriptor on an STL container which will be written to file at a later point. The STL container is shared between the target's threads and must be protected with a lock.

```
1.  VOID Emit_ConditionalDirect_Bin(const THREADID threadid, const ADDRINT address,
2.                                  const ADDRINT target, const BOOL taken)
3.  {
4.     if( !CanEmit(threadid) ) return;
5.
6.     //setup descriptor
7.     BinaryDescriptorTableEntry binDescriptor;
8.     binDescriptor.tid = *static_cast<UINT8*>(Pin_GetThreadData(tls_key, threadid));
9.     binDescriptor.branchAddress = address;
10.    binDescriptor.targetAddress = target;
11.    //If taken paramater will be non-zero
12.    if(taken == 0)
13.    {
14.       IncrementBranchStatistics(ConditionalDirectNotTaken);
15.       binDescriptor.branchType = ConditionalDirectNotTaken;
16.    }
17.    else
18.    {
19.       IncrementBranchStatistics(ConditionalDirectTaken);
20.       binDescriptor.branchType = ConditionalDirectTaken;
21.    }
22.
23.    //critical section
24.    GetLock(&table_lock, threadid+1);
25.    binDescriptorTable.push_back(binDescriptor);
26.    //increment file counter
27.    IncrementFileCount(BinDescriptorTableEntrySize);
28.    ReleaseLock(&table_lock);
29. }
```

Figure 4.5 Analysis Routine from *mcfTrace*

Figure 4.6 includes the section of *mcfTrace* that writes trace descriptors to file. Because *mcfTrace* can create arbitrarily large control-flow traces, it creates a thread to empty the STL container whenever possible. `ThreadWriteBin` is this thread's function and is launched before the target is instrumented. Lines 12-22 and 32-42 write the descriptor to file or pipe it to a compressor. The `Pin_IsProcessExiting` (Lines 10 and 30) call is used to kill the thread whenever *mcfTrace* detaches from the target process. `ThreadWriteBin` is used in every mTrace tool.

```
VOID ThreadWriteBin(VOID *arg)
{
   THREADID threadid = Pin_ThreadId();
   if(usingCompression)
   {
      while(1)
      {
         //if process is closing (entered fini()) kill thread
         if( Pin_IsProcessExiting() )
            Pin_ExitThread(1);

         GetLock(&table_lock, threadid+1);
         while( !binDescriptorTable.empty() )
         {
            BinaryDescriptorTableEntry temp = binDescriptorTable.front();
            fwrite(&temp.tid, sizeof(temp.tid), 1,outPipe);
            fwrite(&temp.branchAddress, sizeof(temp.branchAddress), 1, outPipe);
            fwrite(&temp.targetAddress, sizeof(temp.targetAddress), 1, outPipe);
            fwrite(&temp.branchType, sizeof(temp.branchType), 1, outPipe);
            binDescriptorTable.pop_front();
         }
         ReleaseLock(&table_lock);
       }
   }
   else
   {
      while(1)
      {
         if( Pin_IsProcessExiting() )
            Pin_ExitThread(1);

         GetLock(&table_lock, threadid+1);
         while( !binDescriptorTable.empty() )
         {
            BinaryDescriptorTableEntry temp = binDescriptorTable.front();
            OutFile.write((char *)&temp.tid, sizeof(temp.tid));
            OutFile.write((char *)&temp.branchAddress, sizeof(temp.branchAddress));
            OutFile.write((char *)&temp.targetAddress, sizeof(temp.targetAddress));
            OutFile.write((char *)&temp.branchType, sizeof(temp.branchType));
            binDescriptorTable.pop_front();
         }
         ReleaseLock(&table_lock);
      }
   }
}
```

Figure 4.6 *mcfTrace* Write Routine

4.1.3 Verification/Test

mcfTrace was tested using two assembly code programs, BranchEnumeration.s and BranchTest.s. BranchEnumeration contains all of the x86_64 branch instructions to ensure that *mcfTrace* collects the correct branch instruction information for each branch. The Intel-64 and x86 instruction sets [21] list branch instructions not shown in BranchEnumeration.s, but they are really mnemonics for the instructions already provided, e.g. the *ja* instruction is really a *jnbe* instruction.

Figure 4.7 contains a small selection of conditional branch instructions from BranchEnumeration.s. Part one of this test program lists branch instructions belonging to the j* conditional jump family and part two contains conditional branch instructions belonging to the loop* branch family. These four branches are not taken and will be reported consecutively by *mcfTrace*. Figure 4.8 contains the branch descriptors from *mcfTrace* for this section of code. All four branch instruction descriptors are shown correctly as conditional direct branches that are not taken. Figure 4.9 contains a section of unconditional branches from BranchEnumeration.s. In this case, the *jmp* and *c*all instructions are unconditional direct branches, while the *rtn* instruction is an unconditional indirect branch.

```
#Part 1
#unsigned conditional direct branches
#all branches will not be taken
#branch if strictly above
#Taken when CF and ZF are both zero
 mov     rax, 1
 cmp     rax, 2
 jnbe    exit1
#branch if above or equal
# Taken when CF is 0
 jnb     exit1
#Part 2
#More conditional branch instructions
#Loop family
 mov rcx, 1
loop1:
 loop loop1
 mov rcx, 1
loop2:
 loope loop2
    mov rcx, 1
```

Figure 4.7 Selection from BranchEnumeration.s

```
mcfTrace ASCII Output, with disassembly:
0, 0x00000000004004d3, 0x000000000040059e, C, D, NT      jnbe 0x40059e
0, 0x00000000004004d9, 0x000000000040059e, C, D, NT      jnb 0x40059e
0, 0x0000000000400572, 0x0000000000400572, C, D, NT      loop 0x400572
0, 0x000000000040057b, 0x000000000040057b, C, D, NT      loope 0x40057b
```

Figure 4.8 *mcfTrace* output for BranchEnumeration.s selection

```
#Unconditional Direct jump
 jmp     Label1
#Not Executed
 test    rax, rax

Label1:
#setup puts
    mov     edi, OFFSET FLAT:.LC0
#unconditional direct branch - call
    call    puts
exit1:
    leave
    .cfi_def_cfa 7, 8
    Ret
```

Figure 4.9 Unconditional branches from BranchEnumeration.s

Figure 4.10 contains the relevant *mcfTrace* output for this section of BranchEnumeration.s. The targets for the *jmp* and *call* instructions match the addresses shown in their assembly mnemonic. The *ret* instruction is indirect and the target shown in the descriptor was taken from the stack.

Next, *mcfTrace* was tested with BranchTest.s, which creates more sophisticated branch contexts. Figure 4.12 contains a sample of BranchTest.s. In Lines 1-12 we take several successful branches before falling through to an indirect call instruction at line 22.

```
mcfTrace ASCII Output, with disassembly:
0, 0x000000000040058f, 0x0000000000400594, U, D, T       jmp 0x400594
0, 0x0000000000400599, 0x00000000004003b8, U, D, T       call 0x4003b8
0, 0x000000000040059f, 0x00007f688ca5ce5d, U, I, T       ret
```

Figure 4.10 *mcfTrace* output for BranchEnumeration.s section

```
1.  Label1:
2.          mov     rax, 2
3.          cmp     rax, 1
4.          jnbe    Label2
5.          test rax, rax
6.  Label2:
7.          cmp rax, 2
8.          jz Label3
9.  Label3:
10.         mov rcx, 5
11. Label4:
12.         loop Label4
13.
14.         #setup puts
15.         mov     edi, OFFSET FLAT:.LC0
16.
17.         #unconditional direct branch - call
18.         call    puts
19.
20.         #unconditional indirect branch - call
21.         mov     rax, OFFSET FLAT:test
22.         call    rax
23.
24.         leave
25.         .cfi_def_cfa 7,8
26.
27.         #indirect taken branch
28.         ret
29.         .cfi_endproc
```

Figure 4.11 Selection from BranchTest.s and *mcfTrace* output

Figure 4.12 lists the correct descriptors for the code shown in Figure 4.11. The first six descriptors map to the branches taken in lines 1-12 in Figure 4.11. The seventh descriptor is a branch that exits the loop in lines 11-12 and the last descriptor is an indirect call shown that is set up and executed in lines 21 and 22.

```
1.  0, 0x00000000004004ec, 0x00000000004004f1 C, D, T    jnbe 0x4004f1
2.  0, 0x00000000004004f5, 0x00000000004004f7 C, D, T    jz 0x4004f7
3.  0, 0x00000000004004fe, 0x00000000004004fe C, D, T    loop 0x4004fe
4.  0, 0x00000000004004fe, 0x00000000004004fe C, D, T    loop 0x4004fe
5.  0, 0x00000000004004fe, 0x00000000004004fe C, D, T    loop 0x4004fe
6.  0, 0x00000000004004fe, 0x00000000004004fe C, D, T    loop 0x4004fe
7.  0, 0x00000000004004fe, 0x00000000004004fe C, D, NT   loop 0x4004fe
8.  0, 0x0000000000400511, 0x00000000004004c4 U, I, T    call rax
```

Figure 4.12 Selection from BranchTest.s and *mcfTrace* output

4.2 mlsTrace

mlsTrace is a *Pin* tool designed to collect and save traces of memory referencing instructions for multithreaded programs into a file. For each read and/or write reference in the program, *mlsTrace* captures a trace descriptor that contains the following fields: the logical ID of the thread that executed the instruction that initiate the reference, the instruction address, the read/write operand virtual address, and the read/write value. When *mlsTrace* is capturing both load and store instructions, an additional field is included in the trace descriptor to differentiate between the two. Trace descriptors are collected in the order in which they are executed. Traces are saved to a text or binary file, or piped to a general purpose compressor. When writing to a binary file, *mlsTrace* trace descriptors also contain the operand sizes for decoding purposes. Section 4.2.1 gives a functional description of the *mlsTrace* tool. Section 4.2.2 gives a brief description of tool implementation, and Section 4.2.3 describes verification process and test programs used.

4.2.1 Functional Description

Table 4.3 lists the *mlsTrace* parameters that allow a user to control tracing of memory referencing instructions. These parameters are used to control the following: (a) the trace file type (binary or ASCII), (b) the code segment and trace scope at the instruction and sub-procedure level, (c) load and/or store value tracing, and (d) optional compression.

Table 4.3 *mlsTrace* Parameters

Parameter	**Description**
-a	Saves trace descriptors in an ASCII file
-c <COMPRESSOR>	Trace descriptors are piped to a general-purpose compressor before saving. <COMPRESSOR> = {bzip2, pbzip2, gzip, pigz}
-d	Each descriptor includes a corresponding assembly code
-f	Trace file size limit in Megabytes. Instrumentation and trace collecting stops after reaching this limit.
-filter_no_shared_libs	Only traces target binary, shared libraries are not traced.
-filter_rtn <routine>	Tracing only occurs in a specified routine(s).
-[h | help]	Displays help message with all parameters and their description.
-l <NIST>	Specifies NIST, the number of instructions that will be instrumented in the target.
-o <FNAME>	Specify trace file name, FNAME.
-s <NIST>	Specify NIST, the number of instructions to be skipped before instrumentation begins.
-store	Instrument store instructions. Trace includes trace descriptors for instructions that write to memory. When this option is enabled, a new field is inserted in every descriptor to distinguish between load and store descriptors.

Figure 4.13 shows the format of load and store value descriptors collected by *mlsTrace*. An *mlsTrace* binary trace descriptor includes the following fields:

- *Thread ID* field is 1 byte long and encodes threads from 0 to 255;
- *Load/Store (optional)* field is a byte that distinguishes between load and store descriptors;
- *Instruction Address* and *Operand Address* are fields that are 8 bytes long on 64-bit architectures and include the instruction address and the operand address, respectively;
- *Operand Size* which is 1 byte long and gives the size of the referenced data in bytes; and
- *Value* of the data stored in memory or loaded from memory, whose size depends on *Operand Size*.

Similar to the *mcfTrace* trace format, the address fields can be either four or eight bytes depending on the system's addressing size. Since descriptors can be collected for load and store instructions, we need the *Load/Store* field to encode the descriptor type. This field only appears when load and store value tracing is enabled.

mlsTrace ASCII descriptors also include *Thread ID*, *Load/Store, Instruction Address*, *Operand Address*, *Value fields*, as well as optional assembly code. Individual fields in a descriptor are separated by a comma and individual descriptors are separated by a new line character. Figure 4.13 gives an example of an ASCII descriptor, which specifies that thread 0 issued a load instruction with address 0x0000003f_83200f08, and that the instruction loaded a quadword at the address 0x0036ff3f_84001130 with the value 0x00000000_64320011. In this case we opted to print out the assembly instruction for the descriptor, which is a mov instruction.

mlsTrace Descriptor: Binary Format

Thread ID (1 Byte)	Load/ Store (1 Byte)	Instruction Address (8 Bytes)	Operand Address (8 Bytes)	Operand Size (1 Byte)	Value (Operand Size Bytes(s))

mlsTrace Descriptor: ASCII Format

Thread ID (up to 4 Bytes)	Instruction Address (20 Bytes)	Operand Address (20 Bytes)	Value (Variable)

Example: 0, 0x0000003f83200f08, 0x0036ff3f84001130, 0x0000000064320011 mov rax, qword ptr [rdx]

Figure 4.13 *mlsTrace* descriptor formats: binary (top) and ASCII (bottom)

Figure 4.14 gives an example run of *mlsTrace*. Where both load and store traces are captured for the MatrixMultiplication_OpenMP program that multiplies two randomly generated squared matrices with 16x16 elements. *mlsTrace* creates the trace file, mlsTrace.out2013_12_18_20.16.52.txt, as well as a text file, mlsTrace.out2013_12_18_20.16.52.Statistics, which contains statistics relating to the load and store instructions and their operand sizes. If the user does not supply his or her own output trace file name, *mlsTrace* will create a file using a time stamp as the name. Lines 2-11 show messages from *mlsTrace* sent to standard output indicating that 8 software threads are created. Line 12 is the target's output. Lines 14-23 are descriptors from the top of the trace file and lines 25-46 are from the statistics file.

```
[myersar@EB245-mhealth3 ManualExamples]$ pin -t obj-intel64/mlsTrace.so -a -store
          -- /Matrix_Multiplication_OpenMP 16
mlsTrace: Writing to text file: mlsTrace.out2013_12_18_20.16.52.txt
mlsTrace descriptor: ThreadID, Load/Store,  Instruction Address,
          Operand Address, Operand Size, Value
mlsTrace: thread begin 0 14635
mlsTrace: thread begin 1 14643
mlsTrace: thread begin 2 14644
mlsTrace: thread begin 3 14645
mlsTrace: thread begin 4 14646
mlsTrace: thread begin 5 14647
mlsTrace: thread begin 6 14648
mlsTrace: thread begin 7 14649
200
[myersar@EB245-mhealth3 ManualExamples]$ head mlsTrace.out2013_12_18_20.16.52.txt
0, S, 0x0000003f83200b03, 0x00007ffffafd9180, 8, 0x0000000000000000
0, S, 0x0000003f83201130, 0x00007ffffafd9178, 8, 0x0000000000000000
0, S, 0x0000003f83201134, 0x00007ffffafd9170, 8, 0x0000000000000000
0, S, 0x0000003f83201136, 0x00007ffffafd9168, 8, 0x0000000000000000
0, S, 0x0000003f83201138, 0x00007ffffafd9160, 8, 0x0000000000000000
0, S, 0x0000003f8320113a, 0x00007ffffafd9158, 8, 0x0000000000000000
0, S, 0x0000003f8320113c, 0x00007ffffafd9150, 8, 0x0000000000000000
0, L, 0x0000003f83201156, 0x0000003f8341fb80, 8, 0x0000003f83201130
0, S, 0x0000003f8320115d, 0x0000003f8341fd48, 8, 0x00059d5cfe8a2ad6
0, L, 0x0000003f83201167, 0x0000003f8341ffc8, 8, 0x0000003f8341fdf0
[myersar@mhealth3 ManualExamples]$cat mlsTrace.out2013_12_18_20.16.52.Statistics
Instrumentation Time: 3838.15 ms
Number of Threads: 8
Traced 11636618 instructions
Skipped 0 instructions
Total Load Operands: 1846653
        125272 ( %6.78 ) Byte Operands
        8284 ( %0.45 ) Word Operands
        1467034 ( %79.44 ) Doubleword Operands
        245226 ( %13.28 ) Quadword Operands
        0 ( %0.00 ) Extended Precision Operands
        837 ( %0.05 ) Octaword Operands
        0 ( %0.00 ) Hexaword Operands
        0 ( %0.00 ) Operands of other size
Total Store Operands: 194098
        6890 ( %3.55 ) Byte Operands
        41 ( %0.02 ) Word Operands
        38830 ( %20.01 ) Doubleword Operands
        147948 ( %76.22 ) Quadword Operands
        0 ( %0.00 ) Extended Precision Operands
```

```
44.        389 ( %0.20 ) Octaword Operands
45.        0 ( %0.00 ) Hexaword Operands
46.        0 ( %0.00 ) Operands of other size
```

Figure 4.14 *mlsTrace* example output

4.2.2 Implementation Details

mlsTrace instrumentation occurs at the instruction and instruction operand level. We iterate over basic blocks and the instructions in each basic block, but because the x86 and Intel 64 instruction sets [21] include instructions that contain more than one read or operand, *mlsTrace* may insert more than one analysis routine for each load/store instruction. *mlsTrace* collects the frequency of load and store operands and their sizes and saves them to a statistics file.

Intel uses five fundamental data types: byte, word (two bytes), doubleword, quadword, and octaword (or double quadword). In addition, Intel has included larger types to supplement their floating point unit and SIMD extensions. The operand sizes that *mlsTrace* tracks are included in Table 4.4. Operands with a size not listed in this table are listed as “other” in the statistics file.

Table 4.4 *mlsTrace* Data Types

Data Type	Size In Bytes
Word	2
Doubleword	4
Quadword	8
Octaword	16
Hexaword	32
Extended Precision Floating Point	10

Figure 4.15 shows an instrumentation routine found in mlsTrace.cpp. mlsTrace.cpp contains instrumentation and other housekeeping functions for thread creation and Pin call backs. *mlsTrace* inserts (in line 5) the analysis function `Set-FastForwardAndLength` which counts the number of instructions executed in the target. This procedure allows us to implement fast forwarding and trace length control functions. If we are fast forwarding, this analysis function simply counts the number of instructions left to skip until tracing begins. If we are tracing, `Set-FastForwardAndLength` counts the number of instructions executed while tracing. Line 8 uses Pin routines to tell us if an instruction reads memory, writes to memory, or both. In line 14 we iterate over each memory referencing operand and insert analysis routines that emit load value descriptors and store value descriptors at lines 16-20 and 32-35, respectively. There are some instructions, such as *inc*, that have operands that read and write to memory, so we must allow for that case.

Instructions that write to memory present analysis-atomicity problems [10] for accurate tracing. The process of executing an instruction that writes to memory and inspecting that memory address with instrumented code is not atomic and different threads could write to the same memory address before inspection. We solve this problem by using a lock to protect store instructions during analysis. In lines 24-26 we insert an analysis function to lock the tool from writing to the same address and in lines 32-35 we capture the descriptor for the store instruction and unlock the lock. Certain control flow instructions, notably the call instruction, write to memory so we need to inspect memory after the branch is taken, otherwise we insert the analysis routine after the instruction. This context is managed by lines 28-30 in Figure 4.15. The logical thread ID of the thread issuing the memory referencing instruction, the instruction address, the operand virtual address, and the operand size are passed to each analysis routines. `InsertPredicatedCall` only injects analysis routines if a load or store instruction has a true predicate, as Intel-64 and x86 architectures can use predicated `mov` instructions.

```
for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl) )
{
   for(INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins) )
   {
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)SetFastForwardAndLength,
                       IARG_THREAD_ID, IARG_END);

      if(INS_IsMemoryRead(ins) || INS_IsMemoryWrite(ins) )
      {
         UINT32 memOperands = INS_MemoryOperandCount(ins);

         for(UINT32 memOp = 0; memOp < memOperands; memOp++)
         {
            if( INS_MemoryOperandIsRead(ins, memOp) && TraceLoad.Value() )
            {
               INS_InsertPredicatedCall(ins,IPOINT_BEFORE,
                        (AFUNPTR)Emit_LoadValueDescriptor_ASCII,
                        IARG_THREAD_ID, IARG_INST_PTR,
                        IARG_MEMORYOP_EA, memOp,
                        IARG_MEMORYREAD_SIZE, IARG_END);
            }
            if(INS_MemoryOperandIsWritten(ins, memOp) && TraceStore.Value() )
            {
               INS_InsertPredicatedCall( ins, IPOINT_BEFORE,
                       (AFUNPTR)lock_WriteLocation, IARG_FAST_ANALYSIS_CALL,
                        IARG_THREAD_ID, IARG_MEMORYOP_EA, memOp, IARG_END);

               IPOINT where = IPOINT_AFTER;
               if (!INS_HasFallThrough(ins))
                    where = IPOINT_TAKEN_BRANCH;

               INS_InsertPredicatedCall( ins, where,
                        (AFUNPTR)Emit_StoreValueDescriptor_ASCII,
                        IARG_THREAD_ID, IARG_INST_PTR,
                        IARG_MEMORYWRITE_SIZE, IARG_END);
            }
         }
      }
   }
}
```

Figure 4.15 *mlsTrace* instrumentation from mlsTrace.cpp

Figure 4.16 lists the analysis functions used by *mlsTrace* to emit a store descriptor to a binary file. These are located in mlsTrace.h, which only contains the analysis routines inserted by mlsTrace.cpp. `lock_WriteLocation` prevents other application threads from writing to a memory location before *mlsTrace* inspects memory for the value written for an instrumented instruction. `Emit_StoreValueDescriptor_Bin` is an analysis function that captures binary trace descriptors for instructions that write to memory and pushes them on an STL container. Similar to *mcfTrace*, *mlsTrace* spawns a thread that continuously writes the contents of this container to a file or compressor. In lines 12 and 13 we copy the value written to memory and release the lock that was protecting the address. The `CanEmit` routine is used to check if tracing is enabled or to detach *mlsTrace* from the process if the trace file limit size is reached. In lines 20-40 we save the trace descriptor fields to a container, which is protected by a lock. Because Intel uses a little endian byte ordering, the load and store values are converted to big endian when writing to an ASCII file.

```
VOID Pin_FAST_ANALYSIS_CALL lock_WriteLocation( const THREADID threadid,
                                                      const ADDRINT * ea)
{
   GetLock(&mem_lock, threadid+1);
   lockedOperandAddress =  ea;
}
VOID Emit_StoreValueDescriptor_Bin( const THREADID threadid, const ADDRINT address,
                                    const UINT32 opSize)
{
   //copy value
   UINT8 valBuf[opSize];
   Pin_SafeCopy(valBuf, lockedOperandAddress, opSize);
   ReleaseLock(&mem_lock);

   //if we can't record yet, return
   if( !CanEmit(threadid) ) return;
   //increment load statistics
   IncrementStoreStatistics(opSize);

   BinaryDescriptorTableEntry BinDescriptor;
   BinDescriptor.type = store;
   BinDescriptor.tid = *static_cast<UINT8*>(Pin_GetThreadData(tls_key, threadid));
   BinDescriptor.insAddr = address;
   BinDescriptor.operandEffAddr = reinterpret_cast<intptr_t>(lockedOperandAddress);
   BinDescriptor.operandSize = opSize;

   //reverse endianess
   ConvertToBigEndian(valBuf, opSize);

   //Allocate memory for value
   BinDescriptor.data = new UINT8[opSize];
   //copy to struct entry
   std::copy(valBuf, valBuf+opSize, BinDescriptor.data);

   //critical section
   GetLock(&table_lock, threadid+1);
   //Push back, will write on close
   binDescriptorTable.push_back(BinDescriptor);
   IncrementFileCount(BinaryDescriptorSize+opSize);
   ReleaseLock(&table_lock);
}
```

Figure 4.16 *mlsTrace* analysis example from mlsTrace.h

4.2.3 Verification/Test

mlsTrace was tested with a program that executes a number of load and store instructions with varying operand sizes. Two examples are given in this section from the test program, mlsTest.c, along with the load and store value descriptors captured by *mlsTrace*. Figure 4.17 depicts the first section of mlsTest.c, which references some unsigned and signed byte operands, and an unsigned doubleword that is used as a loop counter. In line 5-7 of Figure 4.17 we print the addresses of these variables to standard output. Both loops, starting at lines 9 and 14, generate load and store instructions for the loop counter. Lines 11 and 12 execute store instructions for the two byte arrays, while lines 16 and 17 load array elements to registers.

```
1.  int i;
2.  //bytes
3.  volatile uint8_t uint8[17];
4.  volatile int8_t sint8[17];
5.  printf("i address: %p\n", &i);
6.  printf("uint8 address: %p\n",uint8);
7.  printf("sint8 address: %p\n",sint8);
8.
9.  for(i=0;i<17;i++)
10. {
11.    uint8[i] = i;
12.    sint8[i] = -i;
13. }
14. for(i=0;i<17;i++)
15. {
16.    uint8[i];
17.    sint8[i];
18. }
19.
```

Figure 4.17 Example 1 from mlsTest.c

Figure 4.18 lists the program output from Example 1 and a selection of the load and store value descriptors captured by *mlsTrace* along the assembly instructions associated with each descriptor. Lines 1-3 are from standard output and show the addresses of the three variables used in this example. Lines 4-5 contain the descriptors for the loop counter, i, which is initialized and moved to the eax register (which will be used in a cmp instruction). Line 6 loads the loop counter value into the edx register which is written to uint8 at line 7, negated and written to sint8 at line 9 (negation instruction not shown). These two descriptors come from the source code lines 11 and 12 from Figure 4.17. There are two more store descriptors associated with this loop at lines 12 and 14, with values 0x01 and 0xff, respectively. Lines 15-20 correspond to the second and third iteration of the loop at line 14 in Figure 4.17. The first two byte descriptors, at lines 16 and 17, show that the values 0x01 and 0xff were loaded from memory, while the last two byte descriptors have the values 0x02 and 0xfe. Throughout this example and the following example, the eax register is used to check the for loop condition.

```
1.  i address: 0x7fff19df619c
2.  uint8 address: 0x7fff19df6180
3.  sint8 address: 0x7fff19df6160
4.  0, S, 0x000000000040056a, 0x00007fff19df619c, 4, 0x00000000
    mov dword ptr [rsp+0x1dc], 0x0
5.  0, L, 0x00000000004005b8, 0x00007fff19df619c, 4, 0x00000000
    mov eax, dword ptr [rsp+0x1dc]
6.  0, L, 0x000000000040057e, 0x00007fff19df619c, 4, 0x00000000
    mov edx, dword ptr [rsp+0x1dc]
7.  0, S, 0x0000000000400587, 0x00007fff19df6180, 1, 0x00
    mov byte ptr [rsp+rax*1+0x1c0], dl
8.  0, L, 0x0000000000400595, 0x00007fff19df619c, 4, 0x00000000
    mov edx, dword ptr [rsp+0x1dc]
9.  0, S, 0x00000000004005a0, 0x00007fff19df6160, 1, 0x00
    mov byte ptr [rsp+rax*1+0x1a0], dl
10. 0, L, 0x00000000004005a7, 0x00007fff19df619c, 4, 0x00000001
    mov eax, dword ptr [rsp+0x1dc]
11. 0, L, 0x000000000040057e, 0x00007fff19df619c, 4, 0x00000001
    mov edx, dword ptr [rsp+0x1dc]
12. 0, S, 0x0000000000400587, 0x00007fff19df6181, 1, 0x01
    mov byte ptr [rsp+rax*1+0x1c0], dl
13. 0, L, 0x0000000000400595, 0x00007fff19df619c, 4, 0x00000001
    mov edx, dword ptr [rsp+0x1dc]
14. 0, S, 0x00000000004005a0, 0x00007fff19df6161, 1, 0xff
    mov byte ptr [rsp+rax*1+0x1a0], dl
15. 0, L, 0x00000000004005d1, 0x00007fff19df619c, 4, 0x00000001
    mov eax, dword ptr [rsp+0x1dc]
16. 0, L, 0x00000000004005da, 0x00007fff19df6181, 1, 0x01
    movzx eax, byte ptr [rsp+rax*1+0x1c0]
17. 0, L, 0x00000000004005eb, 0x00007fff19df6161, 1, 0xff
    movzx eax, byte ptr [rsp+rax*1+0x1a0]
18. 0, L, 0x0000000000400604, 0x00007fff19df619c, 4, 0x00000002
    mov eax, dword ptr [rsp+0x1dc]
19. 0, L, 0x00000000004005da, 0x00007fff19df6182, 1, 0x02
    movzx eax, byte ptr [rsp+rax*1+0x1c0]
20. 0, L, 0x00000000004005eb, 0x00007fff19df6162, 1, 0xfe
    movzx eax, byte ptr [rsp+rax*1+0x1a0]
```

Figure 4.18 mlsTest.c output and *mlsTrace* descriptors for Example 1

Example 2, shown in Figure 4.19, is similar to the first example and executes load and store instructions with 10 and 32 byte operands. Intel processors can utilize 10 byte double extended precision floating point data types to reduce the precision loss that occurs in many floating point algorithms. Line 2 prints the address of a double extended precision array and lines 3-8 contain code that reads and writes these array elements to memory. This test program also executes instructions from Intel's advanced vector extensions instruction set (AVX), which are single instruction multiple data instructions that act on 256 bit wide operands. In this case, mlsTest.c uses a compiler intrinsic to perform the addition of two 256 bit operands organized as a vector of eight single precision floating point numbers. Lines 12-14 map to a sequence of AVX load and store instructions that will be captured by *mlsTrace*.

```
1.  volatile long double extendedPre80[8];
2.  printf("extendedPre80 address: %p\n",extendedPre80);
3.  for(i=0;i<8;i++)
4.  {
5.     extendedPre80[i]=i;
6.  }
7.  for(i=0;i<8;i++)
8.  {
9.     extendedPre80[i] = extendedPre80[i] + i;
10. }
11.
12. volatile __m256 A = _mm256_set_ps(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 );
13. volatile __m256 B = _mm256_set_ps(2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 );
14. volatile __m256 R = _mm256_hadd_ps(A,B);
```

Figure 4.19 Example 2 from mlsTest.c

Figure 4.20 contains the output from Example 2 in Figure 4.19 and some of the load and store value trace descriptors captured by *mlsTrace*. Line 1 is from the standard output and contains the address of the double extended precision array. Lines 2-11 come from the first loop in Figure 4.19. Lines 2 and 3 initialize the loop counter while line 4 writes the same integer value to the stack. The descriptor at line 5 shows that this four byte signed integer is loaded from the stack and is converted to a 10 byte double extended precision number and saved to a register. At line 6 the byte operand is saved to the array extendedPre80. The second iteration of the first loop is depicted in lines 7-11, where the floating point value 1.0 is written to extendedPre80. Lines 12-18 are captured during the first iteration of the second loop in Figure 4.19, with Lines 12, 13, 15, and 16 concerning the loop variable. At line 14 a previously saved extendedPre80 element is loaded to a register and another signed doubleword is converted at line 17. The addition instruction is not captured by *mlsTrace*, but the store instruction is at line 18.

```
1.  extendedPre80 address: 0x7fff1a03b980
2.  0, S, 0x0000000000400711, 0x00007fff1a03badc, 4, 0x00000000
    mov dword ptr [rsp+0x1dc], 0x0
3.  0, L, 0x0000000000400760, 0x00007fff1a03badc, 4, 0x00000000
    mov eax, dword ptr [rsp+0x1dc]
4.  0, S, 0x000000000040072c, 0x00007fff1a03b91c, 4, 0x00000000
    mov dword ptr [rsp+0x1c], eax
5.  0, L, 0x0000000000400730, 0x00007fff1a03b91c, 4, 0x00000000
    fild st0, dword ptr [rsp+0x1c]
6.  0, S, 0x000000000040074d, 0x00007fff1a03b980, 10, 0x00000000000000000000
    fstp ptr [rsp+0x80], st0
7.  0, S, 0x0000000000400759, 0x00007fff1a03badc, 4, 0x00000001
    mov dword ptr [rsp+0x1dc], eax
8.  0, L, 0x0000000000400760, 0x00007fff1a03badc, 4, 0x00000001
    mov eax, dword ptr [rsp+0x1dc]
9.  0, S, 0x000000000040072c, 0x00007fff1a03b91c, 4, 0x00000001
    mov dword ptr [rsp+090], eax
10. 0, L, 0x0000000000400730, 0x00007fff1a03b91c, 4, 0x00000001
    fild st0, dword ptr [rsp+0x1c]
11. 0, S, 0x000000000040074d, 0x00007fff1a03b990, 10, 0x3fff8000000000000000
    fstp ptr [rsp+0x90], st0
12. 0, S, 0x000000000040076c, 0x00007fff1a03badc, 4, 0x00000000
    mov dword ptr [rsp+0x1dc], 0x0
13. 0, L, 0x00000000004007de, 0x00007fff1a03badc, 4, 0x00000000
    mov eax, dword ptr [rsp+0x1dc]
14. 0, L, 0x000000000040079f, 0x00007fff1a03b980, 10, 0x00000000000000000000
    fld st0, ptr [rsp+0x80]
15. 0, L, 0x00000000004007a1, 0x00007fff1a03badc, 4, 0x00000000
    mov eax, dword ptr [rsp+0x1dc]
16. 0, S, 0x00000000004007a8, 0x00007fff1a03b91c, 4, 0x00000000
    mov dword ptr [rsp+0x1c], eax
17. 0, L, 0x00000000004007ac, 0x00007fff1a03b91c, 4, 0x00000000
    fild st0, dword ptr [rsp+0x1c]
18. 0, S, 0x00000000004007cb, 0x00007fff1a03b980, 10, 0x00000000000000000000
    fstp ptr [rsp+0x80], st0
```

Figure 4.20 mlsTest.c output and *mlsTrace* descriptors for Example 2

Lastly, Figure 4.21 contains the remaining descriptors for the AVX intrinsics in Figure 4.19. Lines 1 and 2 write the single precision floating point vectors to memory. The trace file includes descriptors for the many doubleword load instructions and vector packing and unpacking instructions, but they are not included here for succinctness. At line three the two vectors are added (with one operand reading from memory) and the result stored at line 4.

```
1.  0, S, 0x00000000004008ba, 0x00007fff1a03b960, 32,
    0x000000410000e0400000c0400000a0400000804000004040000000400000803f
    vmovaps ymmword ptr [rsp+0x60], ymm0
2.  0, S, 0x0000000000400986, 0x00007fff30833c00, 32,
    0x0000104100000041000 0e0400000c0400000a040000080400000404000000040
    vmovaps ymmword ptr [rsp+0x40], ymm0
3.  0, L, 0x00000000004009b3, 0x00007fff30833de0, 32,
    0x000010410000004100 00e0400000c0400000a040000080400000404000000040
    vhaddps ymm0, ymm0, ymmword ptr [rsp+0x60]
4.  0, S, 0x00000000004009bc, 0x00007fff30833be0, 32,
    0x000070410000304100008841000050410000e040000040400000104100 00a040
     vmovaps ymmword ptr [rsp+0x20], ymm0
```

Figure 4.21 *mlsTrace* descriptors for SIMD instructions in Example 2

4.3 mcfTRaptor

Similar to *mcfTrace*, *mcfTRaptor* is a Pin tool designed to collect control flow traces from multithreaded software and save the trace descriptors to a file. However, *mcfTRaptor* seeks to reduce the number of descriptors collected by using the *TRaptor* branch prediction structure to correctly predict branch outcomes and branch targets. Branch instruction trace descriptors are collected whenever a *TRaptor* structure incorrectly predicts branch outcomes or branch targets, and when an exception occurs. *mcfTRaptor* is designed to trace multithreaded software, with a *TRaptor* branch predictor allocated privately to a thread or shared amongst all threads in the process. Descriptors collected by *mcfTRaptor* can be saved to a text or binary file, or be piped to a general purpose compressor. Section 4.3.1 gives a functional description of the *mcfTRaptor*, section 4.3.2 describes some of the implementation details of *mcfTRaptor*, and section 4.2.3 lists steps taken to verify the output of *mcfTRaptor*.

4.3.1 Functional Description

Table 4.5 contains the parameters for controlling control flow tracing with *mcfTRaptor*. These parameters are used to control the following: (a) the trace file type (binary or ASCII)), (b) the *TRaptor* branch prediction structure parameters (c) the segment of the target to trace at the subroutine level, and (d) optional compression. The *TRaptor* predictor contains a *gshare* branch outcome predictor, a return address stack (RAS), and an indirect branch target buffer (iBTB). A user may specify the size and configuration of these predictor structures, such as the number of entries in the *gshare* outcome predictor (ranging from 0 to 4096), the number of entries in the RAS (0, 8, 16, and 32) and the number of entries in the iBTB. In addition, a user may specify whether these structures are thread private or shared by all threads.

Table 4.5 *mcfTRaptor* parameters

Parameter	Description
-gshare <ENTRIES>	gshare outcome predictor size for TRaptor trace module. <ENTRIES> = {0, 256, 512, or 1024, 2048, 4096}.
-RAS <ENTRIES>	Size of return address stack for TRaptor. <ENTRIES> = {0, 8, 16, 32}
-iBTB<ENTRIES>	Size of 2-way set associative indirect branch target buffer for TRaptor. <ENTRIES> = { 0, 16, 32, 64 }
-TRaptorShare	TRaptor structures are shared between threads. This includes the gshare outcome predictor, return address stack, and indirect branch target buffer.
-a	Saves trace descriptors in an ASCII file
-c <COMPRESSOR>	Trace descriptors are piped to a general-purpose compressor before saving. <COMPRESSOR> = {bzip2, pbzip2, gzip, pigz}
-d	Each descriptor includes a corresponding assembly code
-f	Trace file size limit in Megabytes. Instrumentation and trace collecting stops after reaching this limit.
-filter_no_shared_libs	Only traces target binary, shared libraries are not traced.
-filter_rtn <routine>	Tracing only occurs in a specified routine(s).
-[h | help]	Displays help message with all parameters and their description.
-l <NIST>	Specifies NIST, the number of instructions that will be instrumented in the target.
-o <FNAME>	Specify trace file name, FNAME.
-s <NIST>	Specify NIST, the number of instructions to be skipped before instrumentation begins.

Figure 4.22 categorizes the format control flow descriptors collected by *mcfTRaptor*. *mcfTRaptor* uses three distinct trace descriptors for mispredicted out-

comes, mispredicted targets, and exceptions. The binary trace descriptor fields for the three categories are described below.

- *Mispredicted Outcome*
    - The *Thread ID* field is 1 byte long and encodes threads from 0 to 255.
    - The *bCnt* field is 4 bytes long and holds the number of correctly predicted branch outcomes and targets before an incorrect prediction occurs. Whenever a trace descriptor is captured with *mcfTRaptor*, this value is reset to one.
- *Mispredicted Target*
    - The *Thread ID* field is 1 byte long and encodes threads from 0 to 255.
    - The *bCnt* field is 4 bytes long and holds the number of correctly predicted branch outcomes and targets before an incorrect prediction occurs. Whenever a trace descriptor is captured by *mcfTRaptor*, this value is reset to one.
    - The *Taken* field is a 1 byte field used to distinguish between mispredicted target and mispredicted outcome descriptors.
    - *Target Address* is 8 bytes long an contains the correct branch instruction target.
- *Exception*
    - The *Thread ID* field is 1 byte long and encodes threads from 0 to 255.

- The *Exception* field is a 4 byte long field categorizes the trace descriptor as an exception descriptor. This field will always have the value zero, and is used to distinguish an *Exception* descriptor from a *Mispredicted Target* descriptor.
- *iCnt* is 4 byte long field that holds the number of instructions executed before the exception occurred. Like *bCnt*, this field is reset whenever a descriptor is captured by *mcfTRaptor*.
- The *Target address* field is an 8 byte field that holds the exception handler address.

The use of three types of control flow descriptors with varying sizes necessitates the use of fields that differentiate each descriptor for decoding purposes. The *bCnt* field will never take on the value zero, which is reserved for the *exception* field in the exception descriptor. The address fields can be either four or eight bytes depending on the system's addressing size. When tracing in binary mode, mispredicted outcome descriptors are 5 bytes long, mispredicted target descriptors are 14 bytes long, and exception descriptors are 17 bytes long. ASCII descriptors include the same fields as binary descriptors, but can augmented with the assembly instruction that corresponds to the descriptor. Figure 4.22 gives examples for all three ASCII descriptor types that *mcfTRaptor* can capture.

mcfTRaptor descriptor: Binary Format

Mispredicted Outcome

Descriptor Type (1 Byte)	Thread ID (1 Byte)	bCnt (4 Bytes)

Mispredicted Target

Descriptor Type (1 Byte)	Thread ID (1 Byte)	bCnt (4 Bytes)	Taken (1 Byte)	Target Address (8 Bytes)

Exception

Descriptor Type (1 Byte)	Thread ID (1 Byte)	Exception(4 Bytes)	iCnt (4 Bytes)	Target Address (8 Bytes)

mcfTRaptor descriptor: ASCII Format

Mispredicted Outcome

Thread ID (up to 4 Bytes)	bCnt (up to 12 Bytes)

0, 1   jz 0x7f428fe34618

Mispredicted Target

Thread ID (up to 4 Byte)	bCnt (up to 12 Bytes)	Taken (3 Bytes)	Target Address (20 Bytes)

1, 1, T, 0x0000003f83e07780   call rax

Exception

Thread ID (up to 4 Byte)	Exception (4 Bytes)	iCnt (up to 12 Bytes)	Target Address (20 Bytes)

0, 0, 78, div dword ptr [rbp-36]

Figure 4.22 *mcfTRaptor* descriptor formats: binary (top) and ASCII (bottom)

Figure 4.23 contains an example *mcfTRaptor* run for a simple multithreaded matrix multiplication program. In this example, the number of entries for the gshare outcome predictor is set to 4096, the size of the return address stack is 32 entries, and the number of entries for the two-way set-associative indirect branch target buffer is 64. The target is traced in ASCII mode with assembly instructions appended to each descriptor. Lines 2-14 contain output written to standard out from *mcfTRaptor* and line 15 is standard output from the target. Lines 19-25 contain the contents of the statistics file, mcfTRaptor.out2014_1_3_13.16.20.Statistics, generated for this run. Here, we can see the outcome prediction rates for direct conditional branches and target prediction rates for indirect unconditional branches. Lastly, lines 28-37 contain the beginning of the trace file saved by *mcfTRaptor*, mcfTRaptor.out2014_1_3_13.16.20.txt. Each descriptor shown in Figure 4.23 represents a mispredicted outcome for a conditional branch, with all but two descriptors coming from the same instruction, jbe 0x3f832011b0.

```
[mhealth3 ManualExamples]$ pin -t obj-intel64/mcfTRaptor.so -a -d -
          gshare 4096 -RAS 32 -iBTB 64 -- ./Matrix_Multiplication_OpenMP 32
mcfTRaptor: Writing to ASCII file: mcfTRaptor.out2014_1_3_13.16.20.txt
mcfTRaptor: mcfTRaptor descriptors -
          Mispredicted outcomes for direct conditional branches: Thread ID, bCnt
          Mispredicted targets for indirect unconditional branches: Thread ID, bCnt,
                T, Target Address
          Exceptions: Thread ID, bCnt, iCnt, Target Address
mcfTRaptor: Private TRaptor thread begin 0 30326
mcfTRaptor: Private TRaptor thread begin 1 30334
mcfTRaptor: Private TRaptor thread begin 2 30335
mcfTRaptor: Private TRaptor thread begin 3 30336
mcfTRaptor: Private TRaptor thread begin 4 30337
mcfTRaptor: Private TRaptor thread begin 5 30338
mcfTRaptor: Private TRaptor thread begin 6 30339
mcfTRaptor: Private TRaptor thread begin 7 30340
224
[EB245-mhealth3 ManualExamples]$ head mcfTRaptor.out2014_1_3_13.16.20.Statistics
mcfTRaptor: Instrumentation Time 6886.79 ms
mcfTRaptor: Skipped 0 instructions
mcfTRaptor: Recorded 5123700 direct conditional branches, indirect unconditional
            branches, and exceptions
5064797 conditional direct branches
          5040574 ( %99.52 ) outcomes predicted
          24223 ( %0.48 ) outcomes mispredicted
58903 unconditional indirect branches
          56288 ( %95.56 ) targets predicted
          2615 ( %4.44 ) targets mispredicted
0 exceptions
[mhealth3 ManualExamples]$ head mcfTRaptor.out2014_1_3_13.16.20.txt
0, 2    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
0, 3    jnbe 0x3f832014d0
0, 1    jbe 0x3f832014f0
0, 4    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
0, 2    jbe 0x3f832011b0
```

Figure 4.23 *mcfTRaptor* example output

4.3.2 Implementation Details

For *mcfTRaptor*, instrumentation occurs at the instruction level. We iterate over basic blocks and the instructions in each basic block, and insert analysis routines for each branch instruction encountered. The instrumentation code for tracing in ASCII mode is given in Figure 4.24. As with *mcfTrace*, *mcfTRaptor* iterates over newly encountered basic blocks and inserts analysis code for conditional branch, jump, call, and return instructions. Because we can see the outcome and target of a branch instruction before the instruction is executed, the analysis code is always inserted before the branch instruction, using the `IPOINT_BEFORE` context provided by Pin. The analysis routines are organized for efficiency, as some control flow instructions may only use one component of *TRaptor*. For example, lines 8-11 are used to instrument conditional branch instructions, which will only use the gshare outcome predictor in *TRaptor*. Lines 13-15 instrument return instructions, which only utilizes the return address stack when making a target prediction. Subprocedure calls that utilize indirect addressing are instrumented in lines 17-20. This analysis routine consults the indirect branch target buffer for a target prediction and pushes the return address for the call instruction onto the return address stack. Lines 22-25 instrument non-call indirect branch instructions, with the analysis code checking the *iBTB* for target predictions, and lines 27-30 instrument direct calls, where the analysis code only pushes the return address onto the RAS. Lastly, the exception descriptor requires that *mcfTRaptor* count the number of instructions that execute prior to the exception occurring. Lines 32-34 instrument every instruction to increment *iCnt* in the event an exception occurs.

```
for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl) )
{
   for(INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins) )
   {
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)SetFastForwardAndLength,
         IARG_THREAD_ID, IARG_END);
      if(INS_IsDirectBranchOrCall(ins) && INS_HasFallThrough(ins))
         INS_InsertCall(ins, IPOINT_BEFORE,
            (AFUNPTR)Private_DirectConditional_ASCII, IARG_THREAD_ID,
            IARG_INST_PTR, IARG_BRANCH_TAKEN,IARG_END);
      else if(INS_IsRet(ins))
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Private_Ret_ASCII,
            IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR, IARG_END);
      else if(INS_IsIndirectBranchOrCall(ins) && INS_IsCall(ins))
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Private_IndirectCall_ASCII,
            IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR, IARG_ADDRINT,
            INS_NextAddress(ins), IARG_END);
      else if(INS_IsIndirectBranchOrCall(ins))
         INS_InsertCall(ins, IPOINT_BEFORE,
            (AFUNPTR)Private_OtherUnconditionalIndirect_ASCII,
            IARG_THREAD_ID, IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR, IARG_END);
      else if(INS_IsCall(ins))
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Private_Direct_Call,
            IARG_THREAD_ID, IARG_INST_PTR, IARG_ADDRINT, INS_NextAddress(ins),
            IARG_END);
      else
         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Private_iCnt_Increment,
            IARG_THREAD_ID, IARG_END);
   }
}
```

Figure 4.24 *mcfTRaptor* instrumentation

In the rest of this section, we explore the implementation of two analysis routines and the *TRaptor* components they utilize. Figure 4.25 shows the analysis code from mcfTRaptorPrivateAnalysis.h for indirect sub procedure calls when tracing in binary mode. Intel's Pin provides thread local storage, and the *TRaptor* object associated with the thread that executed the branch instruction is retried with the thread ID at line 6. *mcfTRaptor* can also use a shared *TRaptor* object that is protected by a lock. In every *mcfTRaptor* analysis routine the *bCnt* and *iCnt* counters are incremented (lines 8 and 9). The *iBTB* index is generated with the instruction address in the *TRaptor* method `GetIBTBindex` which is listed in Figure 4.26. After retrieving the index the *iBTB* entry needs to check for target prediction. `IBTBisHit`, which is listed in Figure 4.27, takes the *iBTB* index and target and returns true if the prediction is correct. This sub procedure also updates the *iBTB* if necessary. If the target prediction is incorrect, *mcfTRaptor* writes a descriptor to file and resets *bCnt* and *iCnt*. Lastly, the return address of the call instruction is pushed on to the return address stack.

```
VOID Private_IndirectCall_Bin(const THREADID threadid, const ADDRINT addr, const
                              ADDRINT target, const ADDRINT ret)
{
   if(!CanEmit(threadid)) return;

   TRaptor* traptor = get_tls(threadid);

   traptor->IncrementiCnt();
   traptor->IncrementbCnt();

   ADDRINT index = traptor->GetIBTBindex(addr);
   BOOL correct = traptor->IBTBisHit(addr, index, target);

   IncrementBranchStatistics(Target, correct);

   if ( !correct )
   {
      BinaryDescriptorTableEntry mispredicted_outcome;
      mispredicted_outcome.tid = traptor->tid;
      mispredicted_outcome.bCnt = traptor->bCnt;
      mispredicted_outcome.targetAddress = target;
      mispredicted_outcome.type = Target;
      PushBinDescriptor(threadid, mispredicted_outcome);
      traptor->ResetCounters();
   }
   traptor->PushRAS(ret);
}
```

Figure 4.25 *mcfTRaptor* – indirect call analysis code

Figure 4.26 contains the procedure that generates the iBTB index for branch target prediction. The iBTB index is a hash index generated using the upper bits of the branch instruction address and the upper bits of the *path information register* (*PIR*), which records iBTB hits and misses. Line 5 calculates this index and returns it to the analysis routine.

```
inline ADDRINT TRaptor::GetIBTBindex(const ADDRINT address) const
{
   // iBTB.index = PIR[8+index size-1: 8] xor PC[4+index size-1:4];
   if( this->hasIBTB )
      return ( ( PIR >> 8 ) & indexMask ) ^ ( ( address >> 4 ) & indexMask );
   else
      return 0;
}
```

Figure 4.26 *mcfTRaptor* – iBTB index

Figure 4.27 lists the *TRaptor* function to check iBTB predictions and update iBTB entries. The conditions in lines 16-25 are met when the predicted target saved in the iBTB matches the actual branch target of the instruction. These entries are referenced by the index generated by `GetIBTBIndex`, and each way is accessed serially. When a correct prediction is made, the least recently used way is noted for the next access and the path information register (PIR) is updated in the *TRaptor* structure.

```
inline BOOL TRaptor::IBTBisHit(const ADDRINT address, const ADDRINT index,
                               const ADDRINT target)
{
   if( !this->hasIBTB )
      return false;

   BOOL hit;
   if( this->way0[index] == target )
   {
      this->LastUsedWay[index] = WAY0;
      hit = true;
   }
   else if( way1[index] == target )
   {
      this->LastUsedWay[index] = WAY1;
      hit = true;
   }
   else
   {
      hit = false;
      if( this->LastUsedWay == WAY0 )
      {
         this->way1[index] = target;
         this->LastUsedWay[index] = WAY1;
      }
      else
      {
         this->way0[index] = target;
         this->LastUsedWay[index] = WAY0;
      }
   }
   //update PIR
   //PIR[12:0] = ((PIR[12:0]<<2) xor PC[16:4]) | Outcome
   this->PIR = (  (this->PIR << 2) ^  ( ( address >> 4 ) & 0x1fff ) ) | 1;
   return hit;
}
```

Figure 4.27 *mcfTRaptor* – iBTB lookup

Figure 4.28 contains the analysis code inserted for conditional branches, which references the gshare branch outcome predictor structure in *TRaptor*. The gshare index is generated at line 13, and *mcfTRaptor* retrieves the prediction for the given instruction and compares it to the actual outcome at line 16. If the prediction is correct, a binary descriptor associated with this instruction is created and written to file. Lastly, the *gshare* entry is updated with the branch outcome. Because this analysis routine is used when tracing with shared branch outcome and target structures, the *TRaptor* structure is protected with a lock at line 6 and 17.

```
VOID Shared_DirectConditional_Bin(const THREADID threadid, const ADDRINT addr,
                                  const BOOL taken)
{
   //see if we can emit
   if(!CanEmit(threadid)) return;
   GetLock(&shared_lock, threadid+1);


   //increment bCnt and iCnt
   SharedTRaptor->IncrementiCnt();
   SharedTRaptor->IncrementbCnt();

   ADDRINT index = SharedTRaptor->GetGSHAREindex(addr);

   //see if prediction is correct
   BOOL correct = SharedTRaptor->OutcomePredictionIsCorrect(index, taken);

   //update statistics for DirCB
   IncrementBranchStatistics(Outcome, correct);

   //Emit descriptor if prediction is wrong or doesn't use gshare
   if( !correct )
   {
      BinaryDescriptorTableEntry mispredicted_outcome;
      mispredicted_outcome.tid = SharedTRaptor->tid;
      mispredicted_outcome.bCnt = SharedTRaptor->bCnt;
      mispredicted_outcome.type = Outcome;
      PushBinDescriptor(threadid, mispredicted_outcome);
      //reset iCnt, bCnt
      SharedTRaptor->ResetCounters();
   }
   //update TRaptor's GSHARE
   SharedTRaptor->UpdateGSHARE(index, taken);
   ReleaseLock(&shared_lock);
}
```

Figure 4.28 *mcfTRaptor* – conditional branch analysis

The last code section for *TRaptor*, Figure 4.29, contains the functions that generate the *gshare* index and update *gshare*. The *gshare* index is created using the branch history register and address of the branch instruction. The `PCmask` and and

`BHRmask` variables are used to mask off the correct bits and depend on the number of gshare entries. Line 10 begins the procedure that updates the *TRaptor gshare* and uses the previously generated index. The *BHR* is updated with the outcome at line 16 and the two bit state machine updated at lines 21-27.

```
1.  inline ADDRINT TRaptor::GetGSHAREindex(const ADDRINT address) const
2.  {
3.     //create gshare index
4.     //gshare.index = BHR[log2(p):0] xor PC[4+log2(p):4]
5.     if( hasgshare )
6.        return ( (this->PCmask & address) >> 4 ) ^ (this->BHR & this->BHRmask);
7.     return 0;
8.  }
9.
10. inline void TRaptor::UpdateGSHARE(const ADDRINT index, const BOOL taken)
11. {
12.    if( !this->hasgshare )
13.       return;
14.
15.    //update BHR
16.    this->BHR = (this->BHR << 1) | taken;
17.
18.    UINT16 temp = (1 << log2p) - 1;
19.    this->BHR &= temp;
20.
21.    UINT8 prev = this->GSHARE[index];
22.    UINT8 B1 =  prev >> 1;
23.    UINT8 B0 =  prev & 0x1;
24.
25.    UINT8 F1 = ((B1 & B0) | (taken & B1) | (taken &B0)) << 1;
26.    UINT8 F0 = (B1 & ~B0) | (taken & ~B0) | (taken & B1);
27.    this->GSHARE[index] = F1 | F0;
28. }
```

Figure 4.29 *mcfTRaptor* – gshsare index and update

4.3.3 Verification/Test

mcfTRaptor was tested with a number of different situations to verify that the *TRaptor* branch predictor correctly references the gshare outcome predictor, the return address stack, and the indirect branch target buffer for a variety of different instructions. Three sections of an assembly program, traptortest.s, which verifies each component of the *TRaptor* branch prediction mechanism implemented by *mcfTRaptor* are presented here. The *gshare* outcome predictor is tested by using a series of branch instructions seen in Figure 4.30. In this section of the program, the branch instruction at line 9 is executed ten times, followed by two more branch instructions at line 11. The jmp instruction at line 2 does not reference the *TRaptor* mechanism or generate a branch instruction because it is a direct unconditional and can be inferred from the binary. The branch instruction at line 9 is incorrectly predicted by gshare multiple times before a correct prediction is made, as the *branch history register* (*BHR*) needs to shift in the results before the index of *gshare* entry is stabilized.

```
1.          mov     DWORD PTR [rbp-4], 0
2.          jmp     .L2
3.  .L3:
4.          mov     eax, DWORD PTR [rbp-4]
5.          mov     DWORD PTR [rbp-8], eax
6.          add     DWORD PTR [rbp-4], 1
7.  .L2:
8.          cmp     DWORD PTR [rbp-4], 9
9.          jbe     .L3
10.         jbe     .L3
11.         jbe     .L3
```

Figure 4.30 gshare Example

Figure 4.31 contains the resulting test output for this section of the program. For each conditional branch instruction, the *gshare* index, prediction, result, and updated prediction are shown. The trace descriptor is also given. The first five entries correspond the to 6th through 10th iterations of the loop from lines 3-8 in Figure 4.30, and the last two correspond the last two branch instructions in Figure 4.30. The *gshare* index changes every iteration until the *BHR* is saturated (line 24-28). The *gshare* branch predictor uses a two-bit counter to encode prediction states and is initialized with the “weak” not taken state. The first prediction made with index 183 is incorrect and the predictor is updated to the “weak” taken state. The next loop iteration is correctly predicted, but the last iteration is mispredicted (lines 36-40) as the recent outcome results have changed the gshare index. The 6th entry is for an instruction located at after the loop body but still uses the same index. The last entry references a different index and is correctly predicted because the entry was initialized to “weak” not taken.

```
bCnt: 1
GSHARE[119]: 1(NT)
Actual Result: T
*Mispredicted outcome: 0, 2     jbe 0x400481
Next Prediction for GSHARE[119]: 2

bCnt: 1
GSHARE[55]: 1(NT)
Actual Result: T
*Mispredicted outcome: 0, 2     jbe 0x400481
Next Prediction for GSHARE[55]: 2

bCnt: 1
GSHARE[183]: 1(NT)
Actual Result: T
*Mispredicted outcome: 0, 2     jbe 0x400481
Next Prediction for GSHARE[183]: 2

bCnt: 1
GSHARE[183]: 2(T)
Actual Result: T
*Correct outcome prediction
Next Prediction for GSHARE[183]: 3

bCnt: 2
GSHARE[183]: 3(T)
Actual Result: NT
*Mispredicted outcome: 0, 3     jbe 0x400481
Next Prediction for GSHARE[183]: 2

bCnt: 1
GSHARE[183]: 2(T)
Actual Result: NT
*Mispredicted outcome: 0, 2     jbe 0x400481
Next Prediction for GSHARE[183]: 1

bCnt: 1
GSHARE[181]: 1(NT)
Actual Result: NT
*Correct outcome prediction
Next Prediction for GSHARE[181]: 0
```

Figure 4.31 gshare Entries Test Output

In the next example we want to verify that the *RAS* correctly records the target of function return targets. Figure 4.32 contains the assembly code for testing the *RAS*. `main()` starts at line 21, and calls the subroutine `funct()` with an integer parameter set to 5. `funct()` is called recursively five times, totaling seven return instructions for the entire program. Tracing was limited solely to these two functions. The ret instruction returning control to the operating system loader causes a misprediction because the target address was not saved upon program entry.

```
funct:
.LFB0:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        je      .L5
.L2:
        sub     DWORD PTR [rbp-4], 1
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    funct
        jmp     .L4
.L5:
        nop
.L4:
        leave
        Ret

main:
.LFB1:
        push    rbp
        mov     rbp, rsp
        mov     edi, 5
        call    funct
        leave
        ret
```

Figure 4.32 Return Address Stack Example

Figure 4.33 contains the test output for the assembly program shown in Figure 4.32. For each *ret* instruction the *bCnt*, target address, and *RAS* prediction are printed. The descriptor is also printed for mispredictions. The first five entries are the returns inside `funct()`, while the sixth entry is a return to `main()` from funct after the first `funct()` returns. The remaining entry is a return to the operating system ELF loader in the kernel. Except for the last entry, every prediction was correct.

```
bCnt: 1
Target = 400493
RAS[6] = 400493
*Correct target prediction

bCnt: 2
Target = 400493
RAS[5] = 400493
*Correct target prediction

bCnt: 3
Target = 400493
RAS[4] = 400493
*Correct target prediction

bCnt: 4
Target = 400493
RAS[3] = 400493
*Correct target prediction

bCnt: 5
Target = 400493
RAS[2] = 400493
*Correct target prediction

bCnt: 6
Target = 4004a6
RAS[1] = 4004a6
*Correct target prediction

bCnt: 7
Target = 7fd2349a7cdd
RAS[0] = 0
*Mispredicted target: 0, 8, T, 0x00007fd2349a7cdd       ret
```

Figure 4.33 Return Address Stack Example Results

Lastly, we test the *iBTB* for indirect function calls. Figure 4.34 contains an assembly program that executed an indirect call instruction to `fpoint1()` ten times at line 15. The address of the function is loaded into the rdx register at line 14. The descriptors and test output for the conditional branch at line 20 is suppressed. Each time this indirect call is executed we inspect the *iBTB* index and its entry.

```
1.  fpoint1:
2.          push    rbp
3.          mov     rbp, rsp
4.          leave
5.          ret
6.  main:
7.          push    rbp
8.          mov     rbp, rsp
9.          sub     rsp, 16
10.         mov     QWORD PTR [rbp-16], OFFSET FLAT:fpoint1
11.         mov     DWORD PTR [rbp-4], 0
12.         jmp     .L4
13. .L5:
14.         mov     rdx, QWORD PTR [rbp-16]
15.         mov     eax, 0
16.         call    rdx
17.         add     DWORD PTR [rbp-4], 1
18. .L4:
19.         cmp     DWORD PTR [rbp-4], 9
20.         jle     .L5
21.         leave
22.         Ret
```

Figure 4.34 iBTB Example

Figure 4.35 contains the iBTB test output for five iterations of the loop containing the indirect call, the first two iterations and the last three iterations. The pathway information register (PIR), set index, target, predictions from both ways are printed for each execution. If the prediction is incorrect, the descriptor is given as well. The first iteration results in a compulsory miss while the second iteration results in a hit. The last three entries show that the iBTB correctly predicted the targets for the last three iterations of the loop. Intermediate iterations that are not shown here incurred more compulsory misses as different PIR values generated different set indexes.

```
PIR = 0x0, address = 0x40049c
set index = 9
target = 400474
way0[9] = 0
way1[9] = 0
Miss! way0 set to last used way.
new PIR = 0x49
*Mispredicted target: 0, 2, T, 0x0000000000400474       call rdx

PIR = 0x49, address = 0x40049c
set index = 9
target = 400474
way0[9] = 400474
way1[9] = 0
Found in way0. way0 set as last used.
new PIR = 0x6d
*Correct target prediction

PIR = 0x16d, address = 0x40049c
set index = 8
target = 400474
way0[8] = 0
way1[8] = 0
Miss! way0 set to last used way.
new PIR = 0xfd
*Mispredicted target: 0, 2, T, 0x0000000000400474       call rdx

PIR = 0xaabd, address = 0x40049c
set index = 3
target = 400474
way0[3] = 400474
way1[3] = 0
Found in way0. way0 set as last used.
new PIR = 0xbd
*Correct target prediction

PIR = 0xaabd, address = 0x40049c
set index = 3
target = 400474
way0[3] = 400474
way1[3] = 0
Found in way0. way0 set as last used.
new PIR = 0xbd
*Correct target prediction
```

Figure 4.35 iBTB Results

4.4 mlvCFiat

mlvCFiat is a Pin tool for load value tracing that reduces the number of descriptors needed to replay the execution path for multithreaded software. *mlvCFiat* collects a reduced set of load value descriptors by using a cache first access mechanism to track cache block evictions. First access flags provide the status of cache blocks. A trace descriptor is collected whenever the access flags are reset, either on a cache miss or on the first hit for a cache block. After the first hit, the register *fahCnt* is incremented for each following cache hit, which is captured with the next trace descriptor. A hardware implementation of cache first access would augment the existing cache structure with first access flag bits, but the *mlvCFiat* Pin tool simulates a cache in software. *mlvCFiat* can trace multithreaded software by allocating a cache first access structure privately to each thread or utilize a shared global first access structure. Like the preceding tools, trace descriptors can be saved to a binary or text file, or piped to a general purpose compressor. Section 4.4.1 provides a functional description of *mlvCFiat* and section 4.4.2 some of the implementation details found in *mlvCFiat*. Section 4.4.3 describes the steps taken to verify the output of *mlvCFiat*.

4.4.1 Functional Description

Table 4.6 includes a description of the parameters that can be used with *mlvCFiat* to control the following: (a) the trace file type (binary or ASCII), (b) the *mlvCFiat* cache and first access mechanism parameters, (c) the segment of the target to trace at the subroutine level, and (d) optional compression.

Table 4.6 *mlvCFiat* parameters

Parameter	Description
-a	Saves trace descriptors in an ASCII file
-c <COMPRESSOR>	Trace descriptors are piped to a general-purpose compressor before saving. <COMPRESSOR> = {bzip2, pbzip2, gzip, pigz}
-ca <ASSOCIATIVITY>	Sets the associativity of the cache first access structure, with one being direct mapped. By default, the associativity is four. ASSOCIATIVITY = { 1, 2, 4, …}
-cfg <GRANULARITY>	Sets the first access flag granularity, with each flag protecting an operand of size GRANULARITY in a cache block. By default, the granularity is set to four (word). GRANULARITY = { 1, 2, 4, 8, ..}
-cls <LINE SIZE>	Sets the cache block size for the cache utilizing the cache first access flags. By default, the line size is 32 bytes. LINE SIZE = { 1, 2, 4, 8, … }
-cs <KILOBYTES>	The size of the cache utilizing the first access flags in kilobytes. By default, the cache is 32 KB. KILOBYTES = { 1, 2, 4, 8, …}
-cshare	Shares a global cache and first access mechanism between each thread. This is turned off by default, and each thread is allocated a cache and first access mechanism.
-d	Each descriptor includes a corresponding assembly code
-f	Trace file size limit in Megabytes. Instrumentation and trace collecting stops after reaching this limit.
-filter_no_shared_libs	Only traces target binary, shared libraries are not traced.
-filter_rtn <routine>	Tracing only occurs in a specified routine(s).
-[h | help]	Displays help message with all parameters and their description.
-l <NIST>	Specifies NIST, the number of instructions that will be instrumented in the target.
-o <FNAME>	Specify trace file name, FNAME.
-s <NIST>	Specify NIST, the number of instructions to be skipped before instrumentation begins.

Figure 4.36 illustrates the format of the binary and ASCII descriptors collected with *mlvCFiat*. A *mlvCFiat* descriptor collected in binary mode includes the following fields:

- *Thread ID* is a byte long field that encodes the logical ID for the thread that executed the load instruction;
- *First Access Hit Count* is four bytes long and holds the number of cache hits following the last cache eviction or new cache block entry;
- *Operand Size* is one byte long and is the size of the operand contained in *Operand Value*; and
- the *Value* of the operand associated with the load instruction, which is *Operand Size* bytes long.

The *operand size* field is only used for decoding purposes and is not found in the ASCII descriptor, and the *value* field's endianess is corrected.

mlvCFiat descriptor: Binary Format

Thread ID (1 Byte)	First Access Hit Count (4 Byte)	Operand Size (1 Byte)	Operand Value (Operand Size)

mlvCFiat descriptor: ASCII Format

Thread ID (Up To 4 Bytes)	First Access Hit Count (Up To 12 Bytes)	Operand Value (Operand Size)

0, 2, 0x00000004 mov r8d, dword ptr [rax]

Figure 4.36 *mlvCFiat* Descriptor Format *mlvCFiat* ASCII descriptors also include *Thread ID*, *First Access Hit Count*, and *Value*. In Figure 4.37, the ASCII descriptor states thread zero had two first access flag hits before *mlvCFiat* had a first access flag miss for a four byte load operand with a value of 0x00000004.

Figure 4.38 contains an example of *mlvCFiat*'s output. A simple multithreaded matrix multiplication is traced with *mlvCFiat* using a 64 KB cache with 32 byte long cache blocks and an associativity of four. The lines 1 through 12 are output from *mlvCFiat* and in line 13 we inspect the contents of the statistics file that contains cache and first access flag hit statistics. The name of the statistics and trace files were generated with a timestamp because we did not supply *mlvCFiat* with a filename. In line 38 we inspect the beginning of the trace file created by *mlvCFiat*, with several descriptors associated with eight byte mov instructions shown in lines 38-48.

```
pin -t obj-intel64/mlvCFiat.so -a -d -- ./Matrix_Multiplication_OpenMP 32
mlvCFiat: Writing to text file: mlvCFiat.out2014_1_22_16.13.24.txt
mlvCFiat descriptor: ThreadID, fahCnt, Load Value
mlvCFiat: thread begin 0 13711
mlvCFiat: thread begin 1 13719
mlvCFiat: thread begin 2 13720
mlvCFiat: thread begin 3 13721
mlvCFiat: thread begin 4 13722
mlvCFiat: thread begin 5 13723
mlvCFiat: thread begin 6 13724
mlvCFiat: thread begin 7 13725
c563
[myersar@EB245]$ cat mlvCFiat.out2014_1_22_16.13.24.Statistics
Instrumentation Time (ms): 5297.710000
Instructions Traced: 27329641
Skipped Instructions: 0
```

```

-- Cache References Hits:Misses (Hit Rate)
        Total 4769372:23463(99%)
        Byte Operands 177687:4534(97%)
        Word Operands 7195:1900(79%)
        Doubleword Operands 3672973:3584(99%)
        Quadword Operands 907307:13357(98%)
        Extended Precision Operands 0:0(0%)
        Octaword Operands 4210:88(97%)
        Hexaword Operands 0:0(0%)
        Other Sized Operands 0:0(0%)
-- First Access Flag References Hits:Misses (Hit Rate)
        Total 4265739:37172(99%)
        Byte Operands 136310:13283(91%)
        Word Operands 5329:1832(74%)
        Doubleword Operands 3564260:5817(99%)
        Quadword Operands 556745:16219(97%)
        Extended Precision Operands 0:0(0%)
        Octaword Operands 3095:21(99%)
        Hexaword Operands 0:0(0%)
        Other Sized Operands 0:0(0%)
[myersar@EB245]$ head mlvCFiat.out2014_1_22_16.13.24.txt
0, 0, 0x0000003f83201130        sub r13, qword ptr [rip+0x21ea23]
0, 0, 0x0000003f8341fdf0        add rdx, qword ptr [rip+0x21ee5a]
0, 0, 0x000000000000000e        mov rax, qword ptr [rdx]
0, 0, 0x0000000000000004        mov rax, qword ptr [rdx]
0, 0, 0x000000006ffffef5        mov rax, qword ptr [rdx]
0, 0, 0x0000000000000005        mov rax, qword ptr [rdx]
0, 0, 0x0000000000000006        mov rax, qword ptr [rdx]
0, 0, 0x000000000000000a        mov rax, qword ptr [rdx]
0, 0, 0x000000000000000b        mov rax, qword ptr [rdx]
0, 0, 0x0000000000000003        mov rax, qword ptr [rdx]
```

Figure 4.38 *mlvCFiat* Example

4.4.2 Implementation Details

This section describes some of *mlvCFiat*'s implementation details, including the two instrumentation and analysis routines, and handling multiline cache references. *mlvCFiat* instruments a target at the instruction and operand level by inserting analysis code before newly encountered load and store instructions. Because of *mlvCFiat*'s cache first access mechanism, every load and store operand must be investigated for cache and first access flag accesses. Load instructions necessitate that *CFiat* parameters be updated on cache hits, cache misses, and cache block admittance. Whenever a cache miss or hit occurs and the first access flags are not set, *mlvCFiat* retrieves the necessary information and save the descriptor to file. Trace descriptors are not collected for store instructions that cause a cache miss, as the value stored can be inferred from the instruction. However, the corresponding first access flags must be set by store instructions if it is the first hit.

Figure 4.39 contains a section of code from *mlvCFiat*.cpp used to instrument a target. As with the other tools in mTrace, *mlvCFiat* iterates over newly encountered basic blocks at run-time and inserts analysis routines, shown in lines 18, 24, 32, and 38. *mlvCFiat* uses the thread ID of the issuing thread, the address of the operand, and the size of the operand to create a descriptor. These parameters are passed to the analysis routines, which are located in mlvCFiat.h. Operands can be longer than the cache block size used by *mlvCFiat* and a separate analysis routine needs to iterate over subsequent cache sets before acknowledging the cache reference as a hit or miss. This condition is checked in line 12. Load and store instructions may contain more than one operand, so an analysis routine may be inserted more than once before a load or store instruction.

```
for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl) )
{
   for(INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins) )
   {
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)SetFastForwardAndLength,
                     IARG_THREAD_ID, IARG_END);

      UINT32 memOperands = INS_MemoryOperandCount(ins);
      for(UINT32 memOp = 0; memOp < memOperands; memOp++)
      {
         const UINT32 size = INS_MemoryOperandSize(ins, memOp);
         const BOOL single = ( size <= 4 );

         if(INS_MemoryOperandIsRead(ins, memOp))
         {
            if( single )
            {
               INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
               (AFUNPTR)Load_SingleCacheLine_ASCII_Private, IARG_THREAD_ID,
               IARG_MEMORYOP_EA, memOp, IARG_MEMORYREAD_SIZE, IARG_END);
            }
            else
            {
               INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
               (AFUNPTR)Load_MultiCacheLines_ASCII_Private, IARG_THREAD_ID,
               IARG_MEMORYOP_EA, memOp, IARG_MEMORYREAD_SIZE, IARG_END);
            }
            if(INS_MemoryOperandIsWritten(ins, memOp))
            {
               if( single )
               {
                  INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
                  (AFUNPTR)Store_SingleCacheLine_Private, IARG_THREAD_ID,
                  IARG_MEMORYOP_EA, memOp, IARG_MEMORYWRITE_SIZE, IARG_END);
                              }
               else
               {
                  INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
                  (AFUNPTR)Store_MultiCacheLines_Private, IARG_THREAD_ID,
                  IARG_MEMORYOP_EA, memOp, IARG_MEMORYWRITE_SIZE, IARG_END);
               }
            }
         }
      }
   }
```

```
}
```

Figure 4.39 *mlvCFiat* Instrumentation

Figure 4.40 contains an analysis routine inserted before operands that may extend over multiple cache blocks. In this case, a thread will utilize a private data cache and associated cache first access flags privately. These structures are given to a thread when it initially spawns and is referenced by its thread ID. In line 5 the thread's private storage is retrieved. The thread local storage contains the data cache and first access mechanism, along with the *fahCnt* parameter. When tracing with a shared data cache, the *fahCnt* value is still private to each thread. In line 9 the cache is referenced to see if the operand causes a miss or a hit on a new cache block. This procedure is shown in Figure 4.41. The return value dictates whether a trace descriptor associated with operand is collected and pushed on to a data structure to be written to file at a later point. *fahCnt* is reset when a descriptor is emitted and incremented otherwise.

```
VOID Load_MultiCacheLines_Bin_Private(const THREADID threadid,
                                      const ADDRINT * addr, const UINT32 size)
{
   if( !CanEmit(threadid) ) return;
   tls *localStorage = static_cast<tls*>(Pin_GetThreadData(tls_key, threadid));

   uintptr_t address = reinterpret_cast<uintptr_t>(addr);

   bool emit = localStorage->localCache->LoadMultiLine(address, size);

   if(emit)
   {
      BinaryDescriptorTableEntry BinDescriptor;

      UINT8 valBuf[size];
      Pin_SafeCopy(valBuf, addr, size);
      ConvertToBigEndian(valBuf, size);

      BinDescriptor.tid = localStorage->tid;
      BinDescriptor.fahCnt = localStorage->fahCnt;
      BinDescriptor.operandSize = size;
      BinDescriptor.data = new UINT8[size];
      std::copy(valBuf, valBuf+size, BinDescriptor.data);

      GetLock(&table_lock, threadid+1);
      binDescriptorTable.push_back(BinDescriptor);
      IncrementFileCount(BinaryDescriptorSize+size);
      ReleaseLock(&table_lock);

      localStorage->fahCnt = 0;
   }
   else
      localStorage->fahCnt++;
}
```

Figure 4.40 *mlvCFiat* Multiline Cache Load Analysis

Lastly, Figure 4.41 lists the function that handles load operands that potentialy span more than one cache block. In the event that a descriptor needs to be collected, this subroutine will return true. It takes the address and size of the operand in question, and returns false on either of the two conditions:

- At least one of the cache blocks associated with operand is evicted because of a cache miss;
- At least one first access flag protecting the operand is not set.

The data cache is referenced by an index, tag, and line index. The line index is used to reference each byte along a cache block, and `SplitAddress()` routine at line 18 generates the set index and tag from the operand address. An operand that extends beyond the end of a cache block will reside in the following cache sets, and possibly in a different way. This problem is solved using the ending address of an operand and looping through each set that the operand resides in by generating a new starting address for the operand (line 25-32). In line 22 we check if the reference is a hit or a miss. When the cache reference hits (line 25) the flags protecting the operand in that cache block are examined. If at least one flag is zero the return condition is set to false, the flags associated with the operand are set, and the next address is generated if the operand extends beyond the current cache block. In the event of a miss (line 45) we set the return condition to false, clear all of the flags protecting the cache block, set the flags associated with the operand, and continue on to the next cache block if necessary. At the end of the routine, cache and first access statistics are incremented. First access flag hits and misses are only noted when there were no cache misses. Lastly, the condition to collect a descriptor is returned in line 70.

```
LoadMultiLine(ADDRINT addr, const UINT32 size)
{
    bool emit = false;
    const ADDRINT highAddr = addr + size;
    bool cacheAllHit = true;
    bool flagsAllHit = true;

    const ADDRINT lineSize = LineSize();
    const ADDRINT notLineMask = ~(lineSize - 1);
    UINT32 globalSize = size;

    do
    {
        CACHE_TAG tag;
        UINT32 setIndex;
        UINT32 wayIndex;
        UINT32 lineIndex;
        SplitAddress(addr, tag, setIndex, lineIndex);

        SET & set = _sets[setIndex];

        bool localCacheHit = set.Find(tag, wayIndex);
        cacheAllHit &= localCacheHit;

        if( localCacheHit )
        {
            UINT32 localSize;
            if( globalSize + lineIndex > lineSize )
                localSize = lineSize - lineIndex;
            else
                localSize = globalSize;
            globalSize -= localSize;

            bool localFlagsHit = set.AreFlagsSet(wayIndex, lineIndex, localSize);
            //if at least one flag is not set
            if( ! localFlagsHit )
            {
                flagsAllHit = false;
                emit = true;
                //set FA flags
                set.SetFlags(wayIndex, lineIndex, localSize);
            }
        }
        else
        {
```

```
            wayIndex = set.Replace(tag);
            set.ClearFlags(wayIndex);
            emit = true;

            UINT32 localSize;
            if( globalSize + lineIndex > lineSize )
                localSize = lineSize - lineIndex;
            else
                localSize = globalSize;

            globalSize -= localSize;

            set.SetFlags( wayIndex, lineIndex, localSize); // Set FA Flags
        }
        addr = (addr & notLineMask) + lineSize; // start of next cache block
    }
    while (addr < highAddr);

    OPERAND_TYPE opType = getOperandType(size);
    //increment cache statistics
    _cache_accesses[opType][ACCESS_TYPE_LOAD][cacheAllHit]++;
    //increment flag statistics if cache was hit
    if(cacheAllHit)
        _flag_accesses[opType][flagsAllHit]++;
    return emit;
}
```

Figure 4.41 Multiline Cache Load Operation

4.4.3 Verification/Test

This section describes some of the steps taken to test the *mlvCFiat* tool. Two assembly programs, evict.s and multiblock.s, test the cache simulator and first access flag mechanism utilized by *mlvCFiat*. evict.s ensures that the cache simulator handles cache evictions correctly and that the first access mechanism clears and sets the appropriate flags associated with an operand. multiblock.s ensures that the cache simulator and first access mechanism handle operands larger than the line size of the cache. Figure 4.42 contains the assembly program evict.s, which simply writes to a contiguous array of four byte operands. The cache is set to an unrealistic size, 32 bytes, to test way evictions. The associativity is set to two and the line size to 16 bytes. Lines 2 and 3 set up the stack for `main()` and line 4 intializes the loop variable in the ecx register for loop in lines 6-13. Lastly, the leave instruction, which is really two explicit instructions that roll back the state of the stack, is seen at line 14, and the return instruction is at line 15. This loop will execute nine times causing three total line evictions. Since each way is 16 bytes there will be two compulsory misses followed by capacity/compulsory miss which will evict the cache block. In total, there will be one store from the push instruction, 9 stores from the loop, and two loads from the leave instruction and return instruction.

```
1.  main:
2.          push    rbp
3.          mov     rbp, rsp
4.          mov     ecx, 0
5.          jmp     .L2
6.  .L3:
7.          mov     eax, ecx
8.          cdqe
9.          mov     DWORD PTR [rbp-48+rax*4], eax
10.         add     ecx, 1
11. .L2:
12.         cmp     ecx, 8
13.         jle     .L3
14.         leave
15.         ret
```

Figure 4.42 Evict.s

Figure 4.43 contains the result of tracing the evict.s test program. Each cache reference is given an an entry that contains the instruction, operand address, operand size, set index, tag, and line index. Operands larger than 8 bytes may span more than one cache block and some parameters are given for each block belonging to the cache, such as set index and line index. The first entry, for the push instruction is shown in lines 1-6 and potentially spans more than one cache block. In this case, it does not and registers a compulsory miss. The next four entries belong to the first four iterations of the loop in evict.s. The first entry for mov instruction starts at line 8 and causes a cache miss. The next three iterations are hits while the fifth iteration causes a miss in the second way (way 0 here). The sixth, seventh, and eigth iterations are clearly hits in this second way. The ninth iteration, starting at line 56 is a miss and replaces the cache block used for the first four iterations of the loop. The last two instructions are load instructions. The leave instruction causes a cache miss, and a trace descriptor is collected. The return instruction is a hit because the leave instruction brought the return address into the cache. However, when this block was brought into the cache the first access flags were only set for the operand associated with the leave instruction, which is located eight bytes above the return address. Therefore, the first access flags protecting this operand were not set and a trace descriptor is collected.

```
1.  push rbp
2.  Operand Address = 0x7fffa51e1680
3.  Operand Size = 8
4.  Block: 1
5.  Set Index = 0, Tag = 8795997725032, Line Index = 0
6.  Local Miss! Replacing way 1
```

```

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1650
Operand Size = 4
Set Index = 0, Tag = 8795997725029, Line Index = 0
Cache Miss! Replacing way 0

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1654
Operand Size = 4
Set Index = 0, Tag = 8795997725029, Line Index = 4
Cache Hit! Found at way 0

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1658
Operand Size = 4
Set Index = 0, Tag = 8795997725029, Line Index = 8
Cache Hit! Found at way 0

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e165c
Operand Size = 4
Set Index = 0, Tag = 8795997725029, Line Index = 12
Cache Hit! Found at way 0

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1660
Operand Size = 4
Set Index = 0, Tag = 8795997725030, Line Index = 0
Cache Miss! Replacing way 1

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1664
Operand Size = 4
Set Index = 0, Tag = 8795997725030, Line Index = 4
Cache Hit! Found at way 1

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e1668
Operand Size = 4
Set Index = 0, Tag = 8795997725030, Line Index = 8
Cache Hit! Found at way 1

mov dword ptr [rbp+rax*4-0x30], eax
Operand Address = 0x7fffa51e166c
```

```
52. Operand Size = 4
53. Set Index = 0, Tag = 8795997725030, Line Index = 12
54. Cache Hit! Found at way 1
55.
56. mov dword ptr [rbp+rax*4-0x30], eax
57. Operand Address = 0x7fffa51e1670
58. Operand Size = 4
59. Set Index = 0, Tag = 8795997725031, Line Index = 0
60. Cache Miss! Replacing way 0
61.
62. Leave
63. Operand Address = 0x7fffa51e1680
64. Operand Size = 8
65. Block: 1
66. Set Index = 0, Tag = 8795997725032, Line Index = 0
67. Local Cache Miss! Replacing way 1
68.
69. ret
70. Operand Address = 0x7fffa51e1688
71. Operand Size = 8
72. Block: 1
73. Set Index = 0, Tag = 8795997725032, Line Index = 8
74. Local Cache Hit! Found in way 1
75. Local Flags Miss
```

Figure 4.43 evict.s Results

Figure 4.44 contains the assembly program multiblock.s, which uses Intel's AVX SIMD instructions to cache 32 byte operands. The cache parameters are the same from previous testing: the cache size is 32 bytes, the line size is 16 bytes, and the associativity is two. The first push instruction and ending leave and ret instructions are ignored in the testing output, as their entries are equal to entries shown in Figure 4.43. Lines 7-14 set four operands to zero one the stack. These 32 bytes will be packed in the AVX ymm0 register as four, 8-byte doubles in lines 16-22. They are then stored as a single 32 byte operand on the stack in line 24. The next vmovapd instruction at line 25 loads the operand to a different register, ymm1. At this point the program brings an unrelated cache block into one of the ways at line 27. Lastly, line 29 loads the 32 byte operand to the ymm1 register for a second time.

```
1.  main:
2.
3.          push    rbp
4.          mov     rbp, rsp
5.          and     rsp, -32
6.
7.          movabs  rax, 0
8.          mov     QWORD PTR [rsp-8], rax
9.          movabs  rax, 0
10.         mov     QWORD PTR [rsp-16], rax
11.         movabs  rax, 0
12.         mov     QWORD PTR [rsp-24], rax
13.         movabs  rax, 0
14.         mov     QWORD PTR [rsp-32], 0
15.
16.         vmovsd  xmm0, QWORD PTR [rsp-8]
17.         vmovsd  xmm1, QWORD PTR [rsp-16]
18.         vunpcklpd       xmm1, xmm1, xmm0
19.         vmovsd  xmm0, QWORD PTR [rsp-24]
20.         vmovsd  xmm2, QWORD PTR [rsp-32]
21.         vunpcklpd       xmm0, xmm2, xmm0
22.         vinsertf128     ymm0, ymm0, xmm1, 0x1
23.
24.         vmovapd YMMWORD PTR [rsp-64], ymm0
25.         vmovapd ymm1, YMMWORD PTR [rsp-64]
26.
27.         mov QWORD PTR [rsp-96], 1
28.
29.         vmovapd ymm1, YMMWORD PTR [rsp-64]
30.
31.         leave
32.         ret
```

Figure 4.44 multiblock.s

Figure 4.45 contains the results of testing multiblock.s. Only the entries associated with the middle half of the program are shown. The operands referenced by the first four entries are already found in the cache as they were brought in earlier in the program (lines 7-14 in Figure 4.44). These operands are all 8 bytes long and possibly span more than one cache block. The next entry, starting at line 33, is for the 32 byte store instruction from line 24 in Figure 4.44. This instruction causes a miss and brings the operand into the cache, with the first half residing in way 0 and the second half resisiding in way 1. The following 32 byte load instruction traverses both ways and are shown as hits. The entry for the 8 byte mov instruction at line 55 shows that it was a miss. At this point, the cache contains the second half of the operand from the instruction at line 25 of Figure 4.44 in way 1 and the entire operand at line 27 of Figure 4.44 in way 0. When the last instruction executed, a miss occurs and way 1 is replaced before way 0. The last half of the operand would have hit if way 0 was replaced first. Regardless, the cache reference is a miss, and a trace descriptor is collected.

```
1.  vmovsd xmm0, qword ptr [rsp-0x8]
2.  Load Instruction (potentially multiblock)
3.  Operand Address = 0x7fffb79aa778
4.  Operand Size = 8
5.  Block: 1
6.  Set Index = 0, Tag = 8796017109623, Line Index = 8
7.  Local Cache Hit! Found in way 0
8.
9.  vmovsd xmm1, qword ptr [rsp-0x10]
10. Load Instruction (potentially multiblock)
11. Operand Address = 0x7fffb79aa770
12. Operand Size = 8
13. Block: 1
14. Set Index = 0, Tag = 8796017109623, Line Index = 0
15. Local Cache Hit! Found in way 0
```

```
vmovsd xmm0, qword ptr [rsp-0x18]
Load Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa768
Operand Size = 8
Block: 1
Set Index = 0, Tag = 8796017109622, Line Index = 8
Local Cache Hit! Found in way 1

vmovsd xmm2, qword ptr [rsp-0x20]
Load Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa760
Operand Size = 8
Block: 1
Set Index = 0, Tag = 8796017109622, Line Index = 0
Local Cache Hit! Found in way 1

vmovapd ymmword ptr [rsp-0x40], ymm0
Store Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa740
Operand Size = 32
Block: 1
Set Index = 0, Tag = 8796017109620, Line Index = 0
Local Miss! Replacing way 0
Block: 2
Set Index = 0, Tag = 8796017109621, Line Index = 0
Local Miss! Replacing way 1

vmovapd ymm1, ymmword ptr [rsp-0x40]
Load Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa740
Operand Size = 32
Block: 1
Set Index = 0, Tag = 8796017109620, Line Index = 0
Local Cache Hit! Found in way 0
Block: 2
Set Index = 0, Tag = 8796017109621, Line Index = 0
Local Cache Hit! Found in way 1

mov qword ptr [rsp-0x60], 0x1
Store Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa720
Operand Size = 8
Block: 1
Set Index = 0, Tag = 8796017109618, Line Index = 0
```

```
Local Miss! Replacing way 0

vmovapd ymm1, ymmword ptr [rsp-0x40]
Load Instruction (potentially multiblock)
Operand Address = 0x7fffb79aa740
Operand Size = 32
Block: 1
Set Index = 0, Tag = 8796017109620, Line Index = 0
Local Cache Miss! Replacing way 1
Block: 2
Set Index = 0, Tag = 8796017109621, Line Index = 0
Local Cache Miss! Replacing way 0
```

Figure 4.45 multiblock.s Results

CHAPTER 5

EXPERIMENTAL METHODOLOGY

This chapter describes our experimental setup used for the demonstration and evaluation of the mTrace trace tools. Section 5.1 describes hardware and software setup used for running our experiments. Section 5.2 defines evaluation metrics and methods used to acquire them. Section 5.3 gives a short description of a selected set of SPLASH-2 benchmarks that are used as a workload. Finally, Section 5.4 describes the way experiments are run and controlled.

5.1 Environment

Our experimental setup includes a Dell PowerEdge T110 II server with a single Intel Xeon E3-1240 v2 processor and 16 Gbytes of memory. The Xeon E3-1240 v2 processor consists of a single monolithic die with four 2-way threaded physical processor cores for a total of 8 logical processor cores, a shared 8 Mbytes L3/LLC cache memory, an integrated memory controller, PCI and DMI interfaces, a graphics processor, and a system agent. A block diagram is shown in Figure 5.1.

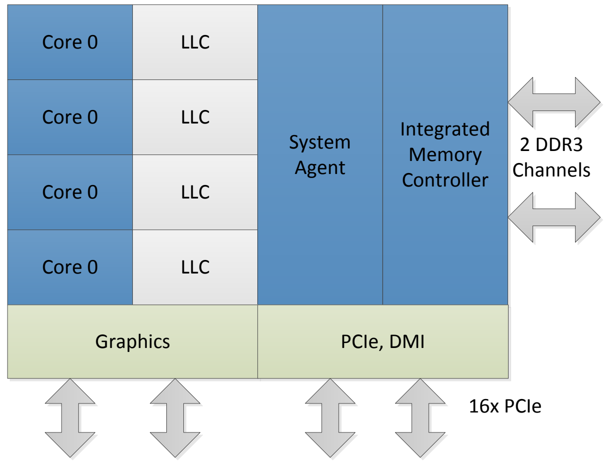

Figure 5.1 Block Diagram of the Xeon E3-1240 v2 processor

The server runs the CentOS 6.3 operating system with 2.6.32 Linux kernel. The mTrace tools are developed and tested under Pin versions 2.12 and 2.13 [22]. The mTrace tools and the target benchmark programs are compiled using the GNU C/C++ compiler gcc-4.7.7. When running experiments mTrace tools can optionally invoke general-purpose trace compressors. The compressors included are gzip 1.3.12 [23] based on a deflate algorithm, a block sorting file compressor bzip2 1.0.5 [24], and their parallel implementations pigz and pbzip2, respectively.

5.2 Metrics

The mTrace tools are designed to capture program traces of multithreaded programs and are primarily evaluated for their functionality. In addition, we conduct a set of experiments to determine their effectiveness by measuring the trace file size and time to capture the trace files (wall-clock time). For each generated raw trace file we determine and report its size. The mTrace trace tools are also combined with bzip2 and gzip general-purpose compressors and the file sizes of compressed traces are reported. To illustrate compressability of individual types of traces, we report the compression ratio, defined as the ratio between a raw trace file size and its compressed trace file size.

The mTrace tools introduce significant overhead to benchmark execution under Pin. The overhead stems from operations perfomed to capture program traces from individual program threads, to write them into a buffer (this operation has to be serialized), and to empty buffers into trace files. Additional overhead comes from combining general-purpose compressors with the tracing tools. To quantify this overhead for a tracing tool, we report benchmark execution time under Pin with the corresponding tool capturing and storing program traces.

5.3 Benchmarks

We use a subset of the SPLASH-2 benchmark suite to evaluate the mTrace tools. SPLASH-2 is a well-established [25] set of parallel programs designed to characterize a wide range of scientific and engineering applications for purposes of exploring architectural properties and interactions of shared-memory multi-core processors and distributed-memory multiprocessors. Each benchmark was executed with one, four, and eight threads. The SPLASH-2 benchmark suite supports multiple benchmark inputs – from a simtest input set that results in relatively short program runs with several hundred millions of instructions executed in a benchmark run, to a simnative input set that involves many billions of instructions executed in a single benchmark run. In our experiments, we use a simsmall input set resulting in benchmark runs with up to several billions of instructions executed.

We report the results of tracing for the following benchmarks: *cholesky*, *fft*, *radiocity*, *radix*, and *raytrace*.

cholesky is a cache block optimized Cholesky decomposition for solving systems of linear equations for sparse matrices. It factors a sparce matrix into the product of a lower triangular matrix and its transpose. c*holesky* is run in a default mode which is optimized for a 16 KB cache.

fft is a one dimensional complex fast Fourier transform algorithm [26] that is optimized to reduce interprocessor communication and cache blocked to maximize data cache reuse. The data set consists of contiguous arrays of data points to be transformed and complex roots of unity, each organized as a set of matrices to be assigned to a processor. 2^{20} total data points are transformed. The data points are organized for 2^{16} cache blocks with a length of 16 bytes.

radix is an iterative radix sort that uses local and global histograms to permute the radix sort keys. The radix used is 4,096 with 4,194,304 keys to sort.

radiosity uses the iterative hierarchical diffuse radiosity method to find the equilibrium distribution of light in a graphical scene. Each scene is initially modeled with a number of polygons. In each iteration of the kernel, light transport interactions are computed among the polygons in a scene, and each polygon is placed in a hierarchy to improve accuracy over the lifetime of the kernel. At the end of each iteration, the overall radiosity is checked for convergence. Data structure accesses are highly irregular and no effort is made to partition data between threads.

raytrace renders a three-dimensional scene using ray tracing. A scene is represented using a hierarchical uniform grid and a ray is traced through each pixel in the image plane. The ray reflects off the scene in an unpredictable way. The partitioning is done by dividing an image plane into contiguous blocks of pixels that are queued to execute on individual processors.

Characterizing the benchmark programs helps analyze the effectiveness of the tracing tools. Control flow and data trace files sizes depend on benchmark characteristics. For example, the perecentage and the type of control flow instructions directly impact the number and size of trace descriptors emitted by the *mcfTrace* tool. Similarly, the number and type of memory referencing instructions directly impact the number and size of trace descriptors emitted by the *mlsTrace* tool. In addition to these parameters, the number of emitted descriptors depends on the type, size, and accuracy of predictor and cache structures used in the *mcfTRaptor* and *mlvCFiat* tools.

Table 5.1 shows relevant information about control flow instructions in the benchmarks of interest as a function of the number of threads (N = 1, 4, and 8). The number of instructions (IC – instruction count) varies between ~698 million (*radix*) and ~1,541 million (*raytrace*). The percentage of branch instructions varies between as low as ~ 1% for *radix* to ~14.5% for *raytrace*. An increase in the number of threads results in a slight increase in the number of instructions and the percentage of the control-flow instructions. This can be explained by an increase in synchronization overhead and data partitioning of the input between each thread as the number of threads increases. The dominant type of branches is conditional direct – ranging from ~1% of all instructions for *radix* to ~10.9% for *raytrace*. A smaller percentage of branches are unconditional direct and unconditional indirect.

Table 5.1 Benchmark Characterization for Control-flow Instructions

Bench-mark	**N**	**IC**	**Bra-nches**	**Condi-tional Direct**	**Uncondi-tional Di-rect**	**Uncondi-tional In-direct**
			[%]	**[%]**	**[%]**	**[%]**
fft	1	960,254,393	8.80	5.84	1.65	1.31
fft	4	960,821,401	8.80	5.85	1.64	1.31
fft	8	961,572,523	8.81	5.85	1.64	1.31
radix	1	694,668,483	1.07	1.07	0.00	0.00
radix	4	696,138,350	1.10	1.10	0.00	0.00
radix	8	698,381,703	1.14	1.14	0.00	0.00
cholesky	1	1,081,038,955	5.62	5.09	0.35	0.18
cholesky	4	1,096,051,872	6.70	6.12	0.40	0.18
cholesky	8	1,136,640,946	7.57	6.98	0.41	0.18
radiosity	1	1,215,384,053	13.74	9.45	2.88	1.41
radiosity	4	1,241,890,515	13.89	9.62	2.85	1.41
radiosity	8	1,242,543,809	13.92	9.64	2.87	1.41
raytrace	1	1,537,614,427	14.58	10.94	1.99	1.66
raytrace	4	1,540,131,445	14.59	10.94	1.99	1.66
raytrace	8	1,541,210,870	14.59	10.95	1.99	1.66

Table 5.2 shows the total number of instructions (IC), the number of operands read from memory (MReads), and the number of operands written to memory (MWrites). The last two columns show the number of operands read from memory and the number of operands written to memory per an exectued instruction (MReads/Ins and MWrites/Ins, respectively). The number of operads read from memory per instruction executed varies between 0.11 for *radix* and 0.31 for *raytrace*. The number of operands written to memory per instruction executed varies between 0.06 for radix to 0.12 for *radiosity*.

Table 5.2 Benchmark Characterization for Memory Reads and Writes

Bench-mark	N	IC	MReads	MWrites		MReads /Ins	MWrites/Ins
fft	1	960,254,510	199,615,611	107,609,072		0.21	0.11
fft	4	960,820,067	199,755,754	107,657,802		0.21	0.11
fft	8	961,583,506	199,947,706	107,726,156		0.21	0.11
radix	1	694,669,006	74,618,267	42,024,250		0.11	0.06
radix	4	696,137,588	75,197,210	42,307,947		0.11	0.06
radix	8	698,379,415	76,088,233	42,728,340		0.11	0.06
cholesky	1	1,081,038,992	275,046,475	98,308,268		0.25	0.09
cholesky	4	1,123,237,461	304,718,317	86,620,208		0.27	0.08
cholesky	8	1,248,542,661	352,888,899	86,830,251		0.28	0.07
radiosity	1	1,215,384,100	392,047,261	140,962,559		0.32	0.12
radiosity	4	1,244,627,545	398,452,881	142,537,302		0.32	0.11
radiosity	8	1,265,132,243	399,635,959	142,946,527		0.32	0.11
raytrace	1	1,537,614,410	481,039,746	166,102,517		0.31	0.11
raytrace	4	1,539,518,453	481,481,500	166,190,940		0.31	0.11
raytrace	8	1,541,626,207	482,135,505	166,258,371		0.31	0.11

Memory reads can be further characterized based on the size of the operand read from memory. Table 5.3 shows a breakdown of the number of operands read from memory per an instruction executed depending on the size of the operand. The Intel64 instruction set supports a number of operand sizes, including 8-bit Byte, 16-bit word, 32-bit double word (DWord), 64-bit quad word (Qword), 80-bit extended precision (EWord), 128-bit octa word (OWord), 256-bit hexa word (HWord), and Others. Expectedly, 64-bit (QWord) are read from memory more often than operands of other sizes, though some benchmarks read 256-bit operands (e.g., *radiosity*). Similarly, Table 5.4 shows a breakdown of the number of operands written to memory per an instruction executed depending on the size of the operand.

Table 5.3 Benchmark Characterization of Memory Reads

Bench-mark	**N**	**Read Bytes**	**Read Words**	**Read Dwords**	**Read QWords**	**Read EWords**	**Read OWords**	**Read HWords**	**Read Oth-ers**
fft	1	0.00	0.02	0.01	0.17	0.00	0.00	0.00	0.01
fft	4	0.00	0.02	0.01	0.17	0.00	0.00	0.00	0.01
fft	8	0.00	0.02	0.01	0.17	0.00	0.00	0.00	0.01
radix	1	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00
radix	4	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00
radix	8	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00
cholesky	1	0.00	0.00	0.00	0.22	0.00	0.00	0.00	0.03
cholesky	4	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.02
cholesky	8	0.00	0.00	0.00	0.26	0.00	0.00	0.00	0.02
radiosity	1	0.00	0.00	0.22	0.09	0.00	0.01	0.00	0.00
radiosity	4	0.00	0.00	0.22	0.09	0.00	0.01	0.00	0.00
radiosity	8	0.00	0.00	0.22	0.09	0.00	0.01	0.00	0.00
raytrace	1	0.00	0.00	0.00	0.30	0.00	0.00	0.00	0.01
raytrace	4	0.00	0.00	0.00	0.30	0.00	0.00	0.00	0.01
raytrace	8	0.00	0.00	0.00	0.30	0.00	0.00	0.00	0.01

Table 5.4 Benchmark Characterization of Memory Writes

Bench-mark	N	Write Byte	Write Word	Write Dword	Write QWord	Write EWord	Write OWord	Write HWord	Write Others
fft	1	0.00	0.01	0.01	0.10	0.00	0.00	0.00	0.00
fft	4	0.00	0.01	0.01	0.10	0.00	0.00	0.00	0.00
fft	8	0.00	0.01	0.01	0.10	0.00	0.00	0.00	0.00
radix	1	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00
radix	4	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00
radix	8	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00
cholesky	1	0.00	0.00	0.00	0.08	0.00	0.01	0.00	0.00
cholesky	4	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00
cholesky	8	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00
radiosity	1	0.00	0.00	0.05	0.06	0.00	0.01	0.00	0.00
radiosity	4	0.00	0.00	0.05	0.06	0.00	0.01	0.00	0.00
radiosity	8	0.00	0.00	0.05	0.06	0.00	0.01	0.00	0.00
raytrace	1	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00
raytrace	4	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00
raytrace	8	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00

5.4 Running Experiments

To evaluate the effectiveness of the mTrace tools, we conducted a number of trace collection runs for each mTrace tool. Table 5.5 illustrates the trace collection runs performed on each tool (*mcfTrace*, *mlsTrace*, *mcfTRaptor*, and *mlvCFiat*). For each benchmark, we collected traces when the number of threads is N=1, N=4, and N=8. For each (benchmark, N) pair, we ran an mTrace tool while collecting original traces (Raw), original traces streamed to the gzip compressor (gzip), and original traces streamed to the bzip2 compressor. Thus, we performed 45 trace collection runs for each tool, or 180 trace collection runs for all mTrace tools combined.

Table 5.5 Trace Collection Runs

Bench-mark	**N = 1**			**N = 4**			**N = 1**		
	Raw	**gzip**	**bzip2**	**Raw**	**gzip**	**bzip2**	**Raw**	**gzip**	**bzip2**
fft	√	√	√	√	√	√	√	√	√
radix	√	√	√	√	√	√	√	√	√
cholesky	√	√	√	√	√	√	√	√	√
radiosity	√	√	√	√	√	√	√	√	√
raytrace	√	√	√	√	√	√	√	√	√

Figure 5.2 shows an excerpt of a script file that controls collection of control flow traces using *mcfTrace* on the fft benchmark. Line 1 of the script specifies the path to the pin executable and Line 2 specifies the path to the *mcfTrace* tool. Line 16 specifies the path to the fft executable with its command line parameters. Lines 18-21 describe the commands for trace collection runs for Raw, gzip, and bzip2 experiments when the number of threads is N=1, and Lines 30-33 describe the commands when the number of threads is N=4. Similar commands are prepared for other benchmarks and other mTrace tools. The commands specify the name of the output trace files and the output Statistics file. In addition, we measure the execution time of the trace collection runs using the Unix *time* command.

```
Pin="/home/myersar/mtrace/pin-2.12-58423-gcc.4.4.7-linux/pin -t"
TOOL=/home/myersar/mtrace/pin-2.12-58423-gcc.4.4.7-linux/source/tools/ManualExamples/obj-intel64/mcfTrace.so

mkdir ./mcfTraceResults
cd ./mcfTraceResults
rm *

#run, collect run time, collect trace size
echo 'mcfTrace Time overhead in seconds' > mcfTraceTime.csv
echo 'benchmark,num threads,raw real,raw user,raw sys,gzip real,gzip user,gzip sys,bzip2 real,bzip2 user,bzip2 sys' >> mcfTraceTime.csv

echo 'mcfTrace trace size in bytes' > mcfTraceSize.csv
echo 'benchmark,num threads,raw,gzip,bzip2' >> mcfTraceSize.csv

#FFT
BENCHMARK="/opt/parsec-3.0/ext/splash2x/kernels/fft/inst/amd64-linux.gcc/bin/fft -m20"
#1 thread
time -p ($Pin $TOOL -o fft_mcfTrace_p1_raw -- $BENCHMARK -p1 > mcfTraceLog.txt) 2> fftTimeP1raw.txt
time -p ($Pin $TOOL -o fft_mcfTrace_p1_gzip -c gzip -- $BENCHMARK -p1 >> mcfTraceLog.txt) 2> fftTimeP1gzip.txt
time -p ($Pin $TOOL -o fft_mcfTrace_p1_bzip2 -c bzip2 -- $BENCHMARK -p1 >> mcfTraceLog.txt) 2> fftTimeP1bzip2.txt
fileSize "fft" "1" "fft_mcfTrace_p1_raw.bin" "fft_mcfTrace_p1_gzip.bin.gz" "fft_mcfTrace_p1_bzip2.bin.bz2"
#timeParse "fft" "1"
#4 threads
time -p ($Pin $TOOL -o fft_mcfTrace_p4_raw -- $BENCHMARK -p4 >> mcfTraceLog.txt) 2> fftTimeP4raw.txt
time -p ($Pin $TOOL -o fft_mcfTrace_p4_gzip -c gzip -- $BENCHMARK -p4 >> mcfTraceLog.txt) 2> fftTimeP4gzip.txt
time -p ($Pin $TOOL -o fft_mcfTrace_p4_bzip2 -c bzip2 -- $BENCHMARK -p4 >> mcfTraceLog.txt) 2> fftTimeP4bzip2.txt
fileSize "fft" "4" "fft_mcfTrace_p4_raw.bin" "fft_mcfTrace_p4_gzip.bin.gz" "fft_mcfTrace_p4_bzip2.bin.bz2"
#timeParse "fft" "4"
#4 threads
time -p ($Pin $TOOL -o fft_mcfTrace_p8_raw -- $BENCHMARK -p8 >> mcfTraceLog.txt) 2> fftTimeP8raw.txt
time -p ($Pin $TOOL -o fft_mcfTrace_p8_gzip -c gzip -- $BENCHMARK -p8 >> mcfTraceLog.txt) 2> fftTimeP8gzip.txt
```

```
32. time -p ($Pin $TOOL -o fft_mcfTrace_p8_bzip2 -c bzip2 -- $BENCHMARK -p8 >>
    mcfTraceLog.txt) 2> fftTimeP8bzip2.txt
33. fileSize "fft" "8" "fft_mcfTrace_p8_raw.bin" "fft_mcfTrace_p8_gzip.bin.gz"
    "fft_mcfTrace_p8_bzip2.bin.bz2"
```

Figure 5.2 An Excerpt of a Script File that Runs *mcfTrace* on the fft Benchmark.

CHAPTER 6

RESULTS

This chapter describes the result of our experimental evaluation. Section 6.1 shows the evaluation results of *mcfTrace*. Section 6.2 shows the evalaution results of *mlsTrace*. Section 6.3 shows the results of *mcfTRaptor* and Section 6.4 shows the results of *mlvCFiat*.

6.1 mcfTrace

Table 6.1 shows the sizes of control-flow traces in bytes generated by *mcfTrace*. For each benchmark, *mcfTrace* is run to generate files containing the control-flow traces in the original binary format (raw) or the compressed binary streams using the gzip or bzip2 general-purpose compressors (gzip and bzip2). The last two columns show the compression ratio achieved by *mcfTrace* when run in combination with the compression utilities. The size of the raw trace files depends on the number of instructions and the frequency of control-flow instructions in a benchmark and ranges from as low as ~133 MB for *radix* to over 4 GB for *raytrace*. We can observe an increase in the raw trace size with an increase in the number of threads.

Table 6.1 *mcfTrace* Output Trace Files Sizes and Compression Ratio

		Trace File Size [Bytes]				Compression Ratio	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	1,520,637,084	13,677,731	2,633,546		111.18	577.41
fft	4	1,522,295,568	38,990,553	21,508,820		39.04	70.78
fft	8	1,524,474,126	47,736,778	26,809,040		31.94	56.86
radix	1	133,813,926	353,890	40,769		378.12	3282.25
radix	4	137,653,992	3,627,161	1,954,396		37.95	70.43
radix	8	143,367,768	5,856,226	3,105,096		24.48	46.17
cholesky	1	1,093,147,254	9,473,216	2,342,543		115.39	466.65
cholesky	4	1,322,821,530	32,300,945	17,279,494		40.95	76.55
cholesky	8	1,549,532,592	56,277,295	31,236,020		27.53	49.61
radiosity	1	3,006,352,998	55,618,847	12,206,438		54.05	246.29
radiosity	4	3,104,184,060	313,693,918	186,335,060		9.90	16.66
radiosity	8	3,113,836,020	330,900,955	204,813,100		9.41	15.20
raytrace	1	4,035,907,728	117,315,720	25,216,232		34.40	160.05
raytrace	4	4,043,987,280	451,195,014	276,887,323		8.96	14.61
raytrace	8	4,048,774,776	469,470,186	303,907,747		8.62	13.32

The compression ratio achieved with gzip varies with benchmarks and the number of threads in a benchmark run. For single-threaded benchmark runs, gzip achieves compression ratios between 34 for *raytrace* and 378 for *radix*. bzip2 achieves even better compression ratios, ranging from 160 for *raytrace* to 3282 for *radix*. The compression ratio is significantly lower in benchmark runs with N = 4 and N = 8 threads. Streaming trace descriptors that come from multiple threads in a single trace file significantly limit the ability of gzip and bzip2 compressors to find and exploit redundancy that exists in each execution thread separately. In benchmark runs with N = 4 threads, gzip achieves compression ratios between 9 for *ray-*

trace and 41 for *cholesky*, whereas bzip2 achieves compression ratio between 14.6 for *raytrace* and 76.6 for *cholesky*. The compression ratios for N = 8 are slightly lower than those observed N = 4.

Table 6.2 shows executions times of benchmarks run under Pin with the *mcfTrace* tool capturing raw or compressed control-flow traces in a file. Overhhead due to capturing traces depends on many factors, from benchmark characteristics, number of threads, and specification of the host machine. In general, that overhead ranges between 20 and 100 times relative to the simplest Pin tool that captures the number of instructions (inscount_tls). The last two columns show the slowdown caused by streaming the trace descriptors into general-purpose compressors, gzip and bzip2, respectively. Interestingly, gzip does not increase the overhead caused by *mcfTrace* when capturing raw traces. This can be explained by the relatively small computational ovehead of gzip that is overlapped with writes to the hard disk. Smaller size of trace files written to the hard disk also reduces the overhead. On the other hand bzip2 significantly increases the tracing overhead, for 8 - 10 times for single-threaded benchmark runs, and for 2.8 – 5.3 times for multithreaded benchmark runs.

Considering both trace file sizes and instrumentation time we recommend *mcfTrace* to be used in combination with gzip because it generates smaller trace files sizes relative to the uncompressed trace file sizes with no additional overhead. If trace file size minimization is a must, then bzip2 should be used – it will produce smaller trace files for multithreaded benchmark runs and significantly smaller trace files for single-threaded benchmark runs when compared to gzip.

Table 6.2 *mcfTrace* Running Times and Slowdown Due to Compression

		Execution Time [sec]				Compression Slowdown	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	53.3	49.7	525.5		0.9	9.9
fft	4	77.4	75.9	407.4		1.0	5.3
fft	8	78.3	82.1	415.7		1.0	5.3
radix	1	19.1	19.1	52.9		1.0	2.8
radix	4	54.0	53.4	87.6		1.0	1.6
radix	8	57.0	56.2	81.8		1.0	1.4
cholesky	1	46.9	46.1	391.0		1.0	8.3
cholesky	4	95.9	92.5	458.8		1.0	4.8
cholesky	8	103.7	109.6	507.2		1.1	4.9
radiosity	1	83.6	81.2	895.3		1.0	10.7
radiosity	4	140.7	136.5	431.5		1.0	3.1
radiosity	8	144.4	142.6	402.5		1.0	2.8
raytrace	1	115.5	107.6	1018.2		0.9	8.8
raytrace	4	175.6	172.2	529.3		1.0	3.0
raytrace	8	182.6	185.0	505.0		1.0	2.8

6.2 mlsTrace

Table 6.3 shows the sizes of memory traces captured by *mlsTrace*. The *mlsTrace* tool captures raw memory read and write traces as well as raw traces compressed using gzip and bzip2. The last two columns show the compression ratio achieved when *mlsTrace* captured data traces streams into gzip and bzip2. Raw trace file sizes are a function of benchmark characteristics such as the instruction count, frequency of memory reads and writes, and operand sizes. Data traces exhibit limited redundancy. The compression ratio achieved by gzip ranges between 4.3 for *radix* and ~9.7 for *radiosity* in signle-threaded benchmark runs and between ~3.7 for *radix* and ~5 for *cholesky* in multithreaded benchmark runs. The compression ratio achieved by bzip2 ranges between 5.7 for *fft* and 24.9 for *radiosity* in single-threaded benchmark runs (N = 1), and between 4.9 for *radix* and 9 for *radiosity* in multi-threaded benchmark runs (N = 4 and N = 8).

Table 6.3 *mlsTrace* Output Trace Files Sizes and Compression Ratio

		Trace File Size [Bytes]				**Compression Ratio**	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
Fft	1	5,596,485,289	1,020,079,791	972,506,097		5.49	5.75
Fft	4	5,599,133,141	1,186,968,787	1,063,716,714		4.72	5.26
Fft	8	5,602,725,000	1,234,018,537	1,105,283,032		4.54	5.07
radix	1	2,116,357,785	485,556,506	366,216,968		4.36	5.78
radix	4	2,133,325,668	552,612,832	426,307,857		3.86	5.00
radix	8	2,159,411,996	581,422,985	443,246,734		3.71	4.87
cholesky	1	8,663,432,796	1,471,125,246	851,148,565		5.89	10.18
cholesky	4	9,145,447,183	1,819,089,301	1,139,067,412		5.03	8.03
cholesky	8	10,172,979,510	2,020,187,096	1,373,024,647		5.04	7.41
radiosity	1	9,177,533,244	940,365,491	368,705,920		9.76	24.89
radiosity	4	9,332,676,058	1,743,056,093	1,035,667,173		5.35	9.01
radiosity	8	9,357,975,123	1,940,078,805	1,185,528,817		4.82	7.89
raytrace	1	12,953,707,377	1,849,508,083	807,429,052		7.00	16.04
raytrace	4	12,966,140,811	2,865,055,779	1,886,428,420		4.53	6.87
raytrace	8	12,977,316,764	3,126,812,569	2,135,364,871		4.15	6.08

Table 6.4 shows execution times of benchmarks run under Pin with *mlsTrace* capturing memory read and write descriptors and streaming them into trace files on

the hard disk directly or through the general-purpose compressors, gzip and bzip2. The last two columns show compression slowdown when trace descriptors are streamed to the compressors before they are written into a trace file. Similarly to *mcfTrace*, *mlsTrace* too minimally increases execution time when combined with gzip. However, the slowdown in cases when *mlsTrace* employs bzip2 is significant, runing between 2.1 and 5.1 times relative to the uncompressed trace capturing.

Table 6.4 *mlsTrace* Execution Times and Compression Slowdowns

		Execution Time [sec]				**Compression Slowdown**	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	158.7	178.5	808.4		1.1	5.1
fft	4	174.8	190.8	868.6		1.1	5.0
fft	8	171.2	194.8	810.0		1.1	4.7
radix	1	92.7	120.5	267.7		1.3	2.9
radix	4	128.2	179.7	295.5		1.4	2.3
radix	8	131.6	178.4	281.5		1.4	2.1
cholesky	1	205.3	226.8	900.2		1.1	4.4
cholesky	4	233.3	281.5	990.4		1.2	4.2
cholesky	8	280.7	309.1	1120.5		1.1	4.0
radiosity	1	280.4	282.4	1319.1		1.0	4.7
radiosity	4	302.5	329.0	1001.7		1.1	3.3
radiosity	8	321.8	313.7	961.1		1.0	3.0
raytrace	1	336.0	340.6	1639.2		1.0	4.9
raytrace	4	360.3	415.3	1326.6		1.2	3.7
raytrace	8	385.7	420.7	1311.2		1.1	3.4

Considering both data trace files sizes and execution overhead we can observe that bzip2 achieves slightly higher compression ratio than gzip, but the difference is likely too small to justify the significantly higher overhead of bzip2 relative to gzip.

6.3 mcfTRaptor

In evaluating *mcfTRaptor* we consider two approaches: private *TRaptor* and shared *TRaptor*. The private *TRaptor* relies on a private predictor structure, exclusively maintained by a single program thread. The shared *TRaptor* assumes a predictor structure shared by all program threads. Regardless of configuration, the predictor structures include a 4096-entry gshare outcome predictor, a 64-entry indirect branch target buffer (iBTB), and a 32-entry return address stack (RAS).

Predictor structures have proved to be very effective in filtering the number of trace descriptors that need to be emitted to a trace file in single-threaded benchmarks [3], and we expect them to work well in multithreaded benchmarks. Misprediction rates in the outcome predictor and the target address predictors (iBTB and RAS) serve as good indicators of the *TRaptor* effectiveness. Lower misprediction rates mean fewer trace descriptors that need to be recorded in a trace file.

Table 6.5 shows the number of conditional direct branches and outcome misprediciton rates, as well as the number of indirect unconditional branches and target address misprediction rates for our benchmark runs. The outcome mispredicition rate ranges from as low as ~0.07% for *radix* to ~8.6% for *raytrace*. An increase in the number of threads does not result in a significant increase in the outcome misprediction rates as each thread has its own predictor structures. A slight increase is still possible due to the time needed to warm-up predictor structures. Similar observations can be made for target address misprediction rates. *Radix* exhibits a relatively high percentage of mispredictions ~13-15%, but the actual number of indirect branches is negligible. Based on these misprediction rates, we can expect *TRaptor* to generate dramatically smaller trace files than *mcfTrace*.

Table 6.5. Private TRaptor Misprediction Rates

		Conditional Direct	**Outcome Misprediction**		**Unconditional Indirect**	**Target Misprediction**
Benchmark	**N**		**[%]**			**[%]**
fft	1	56,075,154	2.367		12,608,375	0.003
fft	4	56,164,674	2.419		12,609,525	0.005
fft	8	56,283,270	2.437		12,610,998	0.007
radix	1	7,427,127	0.073		2,355	13.503
radix	4	7,637,388	0.095		3,750	15.253
radix	8	7,950,663	0.116		5,672	13.082
cholesky	1	55,014,795	3.904		1,955,003	0.041
cholesky	4	75,344,569	3.237		1,988,820	0.052
cholesky	8	112,548,429	2.318		2,043,430	0.067
radiosity	1	114,846,298	8.155		17,161,173	0.085
radiosity	4	119,746,055	8.195		17,474,632	0.105
radiosity	8	121,657,810	8.033		17,576,046	0.091
raytrace	1	168,163,801	8.742		25,484,168	2.998
raytrace	4	168,571,698	8.619		25,503,088	3.165
raytrace	8	168,959,087	8.581		25,538,844	3.089

Table 6.6 shows the trace file sizes for raw *TRaptor* traces (raw) and their compressed versions (gzip and bzip2). By comparing the TRaptor generated raw trace file sizes with the corresponding raw control-flow trace fize sizes generated by *mcfTrace*, we can see a significant reduction is size, ranging from ~40 times for *raytrace* to ~3,500 for *radix*. The last two columns illustrate the potential of TRaptor traces to be further compressed by a factor of ~4 for *raytrace* to ~12 for *cholesky* using gzip, and a factor of 5.9 for *raytrace* to 20.7 for *cholesky* using bzip2.

Table 6.6. Private TRaptor Trace File Sizes

		Output Trace Size [Bytes]				**Compression Ratio**	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	7,968,642	682,878	404,422		11.7	19.7
fft	4	8,160,789	1,072,299	638,049		7.6	12.8
fft	8	8,242,686	1,233,471	784,802		6.7	10.5
radix	1	37,500	5,855	4,338		6.4	8.6
radix	4	52,188	8,731	6,633		6.0	7.9
radix	8	66,510	11,802	9,150		5.6	7.3
cholesky	1	12,898,629	1,040,887	623,559		12.4	20.7
cholesky	4	14,647,143	1,796,840	1,217,190		8.2	12.0
cholesky	8	15,670,374	2,235,523	1,556,134		7.0	10.1
radiosity	1	56,410,818	6,105,575	4,247,789		9.2	13.3
radiosity	4	59,154,543	13,002,449	8,969,660		4.5	6.6
radiosity	8	58,872,636	14,771,288	10,325,310		4.0	5.7
raytrace	1	99,661,752	9,338,611	6,100,864		10.7	16.3
raytrace	4	99,282,171	21,553,551	14,585,726		4.6	6.8
raytrace	8	98,828,169	25,024,844	16,796,688		3.9	5.9

Table 6.7 shows the execution times for benchmark runs with the private *TRaptor* as well as slowdown when using general-purpose compressors. When compared to *mcfTrace*, *mcfTRaptor* requires much more time to produce raw files than *mcfTrace*, in spite of having to write smaller files on the hard disk. This can be explained by an additional overhead cuased by lookups in the simulated predictor structures. When combined with gzip and bzip2, *mcfTRaptor* adds very little or no overhead in the execution time as the compression task can be fully overlapped with capturing traces.

Table 6.7 Private *mcfTRaptor* Execution Times and Slowdown Due to Compression

		Execution Times [sec]				**Compression Slowdown**	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	214.58	246	238.75		1.15	1.11
fft	4	278.81	274.15	266.52		0.98	0.96
fft	8	269.43	276.6	269.76		1.03	1.00
radix	1	176.46	170.09	172.21		0.96	0.98
radix	4	219.97	219.73	218.43		1.00	0.99
radix	8	205.08	214.29	214.56		1.04	1.05
cholesky	1	272.8	273.4	274.96		1.00	1.01
cholesky	4	324.26	335.96	329.87		1.04	1.02
cholesky	8	358.38	363.78	363.87		1.02	1.02
radiosity	1	358.38	363.78	363.87		1.02	1.02
radiosity	4	372.17	401.49	384.44		1.08	1.03
radiosity	8	383.49	373.27	386.11		0.97	1.01
raytrace	1	359.98	396.71	370.57		1.10	1.03
raytrace	4	487.56	466.67	478.48		0.96	0.98
raytrace	8	474.76	459.63	473.05		0.97	1.00

Table 6.8 shows the number of conditional direct branches and outcome mispredicition rates, as well as the number of indirect unconditional branches and target address misprediction rates for our benchmark runs for the shared TRaptor. The outcome mispredicition rates significantly increase relative to those observed in the private TRaptor when the number of threads is N = 4 and N = 8 in all benchmarks except *radix*. For example, the outcome misprediction with N = 4 reaches 47.3% for *radiosity*, 12.3% for *cholesky*, and 43.8% for *raytrace*. Similar trends can be observed for the target address misprediction rates, ranging from 0.007% for *fft* with N = 4 to

76% for *radiosity* when N = 8. The dramatic deterioration of predictor structures' performance is expected as it is caused by conflicting requests coming from different program threads to the shared predictor structures. It should be noted that predictor designs could be enchanced to better support multithreaded workloads, but that is out of the scope of this thesis.

Table 6.9 shows the trace file sizes for raw traces generated by the shared *TRaptor* (raw) and their compressed versions (gzip and bzip2). By comparing the raw trace files sizes generated by the shared *TRaptor* to those generated by the private *TRaptor*, we can see a significant increase in the file sizes in the shared *TRaptor* when the number of threads is N = 4 or N = 8 (Figure 6.1). High misprediction rates on the shared predictor structures result in an increased number of trace descriptors that needs to be emitted during tracing. For example, when N = 4 the shared TRaptor generates 7.1 times larger raw trace file size than the private TRaptor for *fft*, 3.7 times for *cholesky*, and 8.8 times for *radiosity*. An exception is *radix* where the increase is only 1.4 times. Similar observations can be made for N = 8 and for the compressed traces. Figure 6.1 illustrates the ratios calculated by dividing the corresponding trace file sizes generated by the shared TRaptor and by the private TRaptor.

Table 6.8. Shared TRaptor Misprediction Rates

Bench-mark	N	Conditional Direct	Outcome Mispre-diction		Uncondi-tional Indi-rect	Target Mispre-diction
			[%]			[%]
fft	1	56,075,136	2.367		12,608,375	0.003
fft	4	56,165,249	17.185		12,609,565	0.007
fft	8	56,281,901	19.322		12,610,841	0.012
radix	1	7,427,133	0.073		2,357	13.534
radix	4	7,637,233	0.125		3,761	25.445
radix	8	7,950,682	0.190		5,634	28.772
cholesky	1	55,014,795	3.795		1,955,003	0.042
cholesky	4	72,565,834	12.358		1,988,809	4.456
cholesky	8	104,323,219	22.041		2,043,279	4.911
radiosity	1	114,846,298	8.155		17,161,173	0.085
radiosity	4	119,536,557	47.367		17,472,879	70.145
radiosity	8	121,466,057	47.419		17,553,933	76.119
raytrace	1	168,163,801	8.524		25,484,168	2.969
raytrace	4	168,258,563	43.826		25,465,355	48.850
raytrace	8	169,040,607	43.875		25,556,188	53.867

Table 6.9. Shared TRaptor Trace File Sizes

		Output Trace Size [Bytes]				Compression Ratio	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	7,968,525	682,677	388,285		11.67	20.52
fft	4	57,925,020	9,357,182	6,796,357		6.19	8.52
fft	8	65,272,347	11,637,459	8,558,896		5.61	7.63
radix	1	37,479	5,855	4,345		6.40	8.63
radix	4	71,535	13,188	10,268		5.42	6.97
radix	8	114,789	23,028	18,198		4.98	6.31
cholesky	1	12,539,103	1,039,380	623,573		12.06	20.11
cholesky	4	55,135,530	8,042,678	5,765,925		6.86	9.56
cholesky	8	139,465,914	23,564,216	16,599,347		5.92	8.40
radiosity	1	56,410,554	6,106,785	4,247,521		9.24	13.28
radiosity	4	523,573,119	83,876,736	54,543,137		6.24	9.60
radiosity	8	546,015,318	100,094,387	65,742,972		5.46	8.31
raytrace	1	97,359,720	9,239,680	6,099,587		10.54	15.96
raytrace	4	629,043,807	106,700,282	70,171,451		5.90	8.96
raytrace	8	651,491,616	125,150,497	83,276,963		5.21	7.82

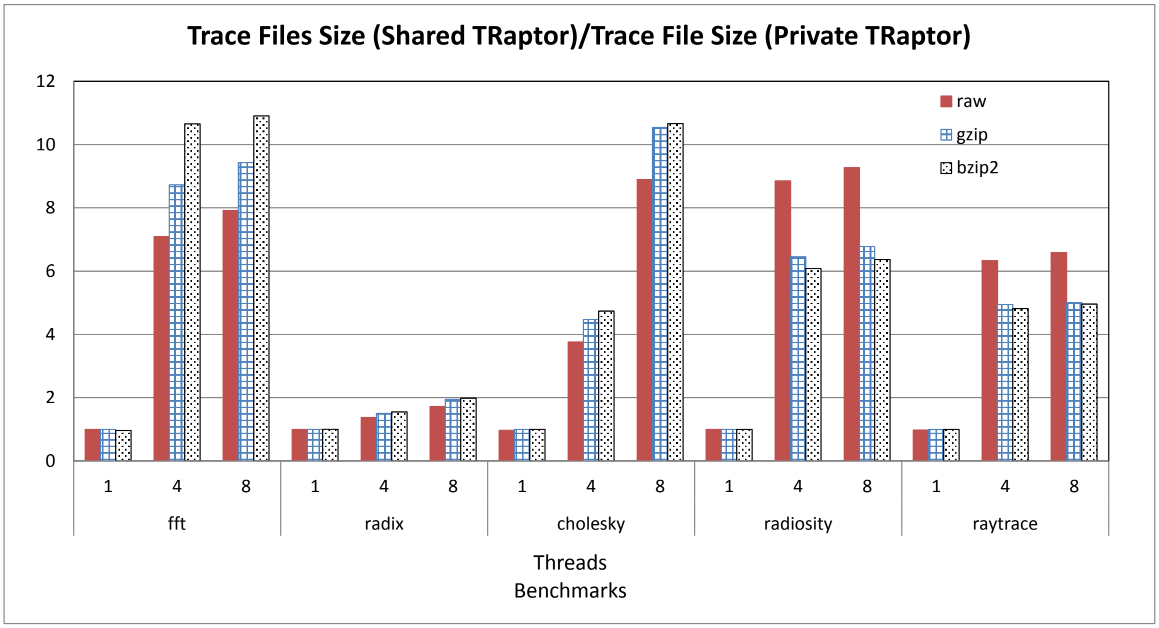

Figure 6.1 Ratio of Trace File Sizes for Shared and Private TRaptor

Table 6.10 shows the execution times for benchmark runs with the shared *TRaptor* as well as slowdown when using general-purpose compressors. When compared to the private *mcfTRaptor*, the executions times slightly increase. Similarly to the private *TRaptor*, the shared *TRaptor* adds a very little or no overhead in the execution time when captured traces are streamed to general-purposed compressors.

Table 6.10 Shared *mcfTRaptor* Execution Times and Slowdown Due to Compression

		Execution Time				Slowdown	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
		[s]	**[s]**	**[s]**			
fft	1	258.96	252.59	257.66		0.98	0.99
fft	4	283.09	272.53	286.55		0.96	1.01
fft	8	279.93	253.37	280.59		0.91	1.00
radix	1	173.14	176.77	164.25		1.02	0.95
radix	4	223.6	226.57	206.54		1.01	0.92
radix	8	204.99	213.37	207.93		1.04	1.01
cholesky	1	284.85	261.42	283.44		0.92	1.00
cholesky	4	345.34	335.91	340.15		0.97	0.98
cholesky	8	369.8	374.25	373.7		1.01	1.01
radiosity	1	279.4	267.66	328.85		0.96	1.18
radiosity	4	405.86	388.68	415.42		0.96	1.02
radiosity	8	412.4	433.37	414.16		1.05	1.00
raytrace	1	359.29	387.29	407.14		1.08	1.13
raytrace	4	479.89	486.57	519.05		1.01	1.08
raytrace	8	501.37	527.28	524.37		1.05	1.05

6.4 mlvCFiat

mlvCFiat implements a version of the *CFiat* technique for filtering load values captured in multithreaded programs using cache first-access bits. We consider two options as follows: (i) a private *CFiat* in which each thread maintains a separate data cache with first-access bits, and (ii) a shared *CFiat* in which all threads share a single data cache with first-access bits. To evaluate the effectiveness of each configuration, we consider direct metrics, such as trace file size and tracing time under *mlvCFiat*. In addition, we consider indirect metrics such as cache hit rates and load first-access hit rates that help us evaluate the impact of various trade-offs faster. The private and shared caches are configured as follows: 64 kB cache size, 4-way set-associativity, and 64 B cache block size.

Table 6.11 shows the number of cache accesses and the cache miss rate, as well as the number of memory reads and the first-access miss rate for the private *CFiat*. Smaller cache miss rates and smaller first-access miss rates translate directly into fewer *mlvCFiat* trace descriptors that need to be recorded in trace files. The number of cache accesses vary across benchmarks, from ~116 million for *radix* to ~647 million for *raytrace*. It stays roughly the same as the number of threads increases for *fft*, *radix*, and *raytrace*, and increases slightly for *cholesky*. The cache miss rate is relatively small for all benchmarks except for *radix* and varies little with an increase in the number of threads. The last column shows the first-access miss rate (FA miss rate). It ranges between 0.86% for *radiosity* with N = 1 and 14.83% for *cholesky* with N = 8. This means that fewer than one memory read out of one hundred will result in a *mlvCFiat* trace descriptor emitted to a file in the case of *radiosity* with N = 1, and that one out of seven memory reads will result in a *mlvCFiat* trace descriptor emitted to a trace file in the case of *cholesky* with N = 8. Based on these results, we expect *mlvCFiat* to be highly effective in filtering the number of trace records emitted to a trace file.

Table 6.11. Private *mlvCFiat* Cache and First Access Hit Rates

Bench-mark	N	Number of Cache Acesses	Cache Miss Rate		Number of Load Accesses	FA Miss Rate
			[%]			[%]
fft	1	307,224,668	1.78		197,292,509	2.29
fft	4	307,413,543	1.15		197,406,682	2.39
fft	8	307,668,664	1.06		197,568,023	2.49
radix	1	116,642,428	13.18		67,680,866	11.54
radix	4	117,505,266	13.15		68,199,113	11.71
radix	8	118,815,351	13.10		68,994,011	12.02
cholesky	1	373,354,743	2.26		267,062,119	14.83
cholesky	4	392,929,212	1.64		300,656,363	9.83
cholesky	8	442,409,902	1.34		350,594,170	7.33
radiosity	1	533,009,818	0.53		390,948,586	0.86
radiosity	4	542,874,634	0.58		398,371,769	0.92
radiosity	8	542,276,706	0.57		398,005,075	0.92
raytrace	1	647,142,264	0.58		477,542,067	1.93
raytrace	4	647,450,206	0.66		477,430,176	2.14
raytrace	8	647,853,582	0.65		477,746,593	2.10

Table 6.12 shows the sizes of raw and compressed trace files captured with *mlvCFiat* for all benchmark runs. The last two columns show the compression ratio determined as the size of a raw *mlvCFiat* trace file divided by the size of the corresponding compressed *mlvCFiat* trace file (gzip and bzip2). The frequency of memory reads, the size of operands, the cache hit rate, and the first-access hit rate are parameters that determine the number of trace descriptors that need to be recorded and thus the trace file sizes. Thus, benchmarks with a high percentage of first-access miss rates produce larger trace files sizes (e.g., *cholesky* and *radix*).

Table 6.12. Private *mlvCFiat* Trace File Sizes

		Output Trace Size [Bytes]				Compression Ratio	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	344,461,089	119,049,767	129,072,841		2.89	2.67
fft	4	347,114,036	124,190,432	133,015,225		2.80	2.61
fft	8	349,597,059	126,559,483	135,926,071		2.76	2.57
radix	1	382,845,328	154,293,762	124,421,632		2.48	3.08
radix	4	387,956,660	162,775,319	133,313,801		2.38	2.91
radix	8	396,445,955	166,961,934	137,552,631		2.37	2.88
cholesky	1	1,689,590,188	185,703,785	193,552,377		9.10	8.73
cholesky	4	1,212,625,123	153,181,545	140,051,857		7.92	8.66
cholesky	8	1,017,343,050	131,351,074	118,539,070		7.75	8.58
radiosity	1	57,384,634	12,846,491	7,273,049		4.47	7.89
radiosity	4	64,365,196	16,855,664	11,362,710		3.82	5.66
radiosity	8	64,093,547	17,879,489	12,508,787		3.58	5.12
raytrace	1	234,938,443	50,705,335	24,831,230		4.63	9.46
raytrace	4	257,305,028	63,450,387	35,401,668		4.06	7.27
raytrace	8	252,083,715	75,029,509	40,395,721		3.36	6.24

Figure 6.2 offers an alternative view into the effectiveness of *mlvCFiat* with private caches. It shows the number of bytes in a trace file divided by the total number of executed instructions (Bytes/Ins) as well as the number of bytes in a trace file divided by the total number of read operations (Bytes/Read). These metrics offer more insights than the total sizes as they capture the number of bytes traced per executed instruction or per read operation in a benchmark. We can see that *radix*, despite having a relatively high first-access miss rate, does not have large number of bytes per instruction emitted to a trace file as such instructions occur infrequently.

Conversely, *cholesky* has a relatively high cost of tracing load values regardless of metric used. These results demonstrate that the modified *CFiat* technique extended to multithreaded applications with private caches and fist-access bits promises a dramatic reduction in the number of trace descriptors that need to be emitted from the target platform.

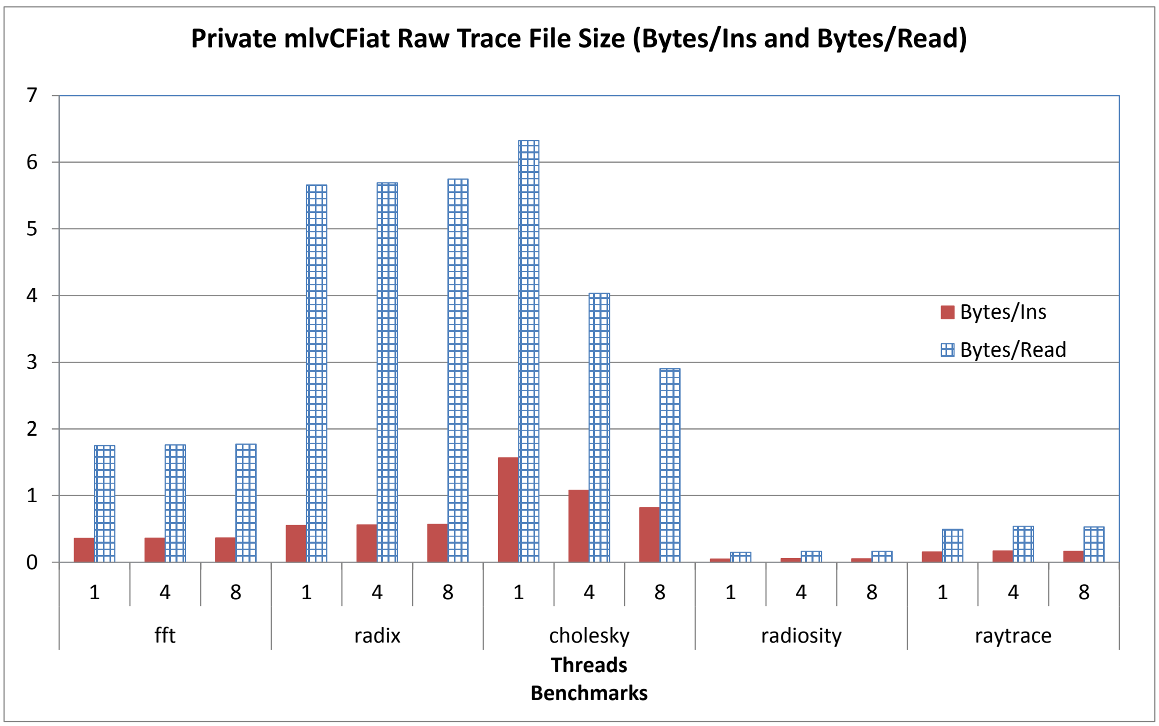

Figure 6.2 Trace File Size in Bytes/Ins and Byte/Read for Private *mlvCFiat*

Table 6.13 shows the execution times of *mlvCFiat* for our benchmark runs. The last two columns show the slowdown caused by general-purpose compressors when *mlvCFiat* streams captured descriptors into them. The execution times of *mlvCFiat* are generally lower than the execution times of *mlsTrace*. For example, *mlvCFiat* requires 289.4 seconds for raytrace with N = 8, compared to 385.7 (Table 6.4) seconds required by *mlsTrace*. Unlike *mlsTrace* that generates large trace files, *mlvCFiat* generates smaller trace files and thus spends less time into trace recording. The slowdown due to compression is negligible in the case of *mlvCFiat* when combined with gzip.

Table 6.13 Private *mlvCFiat* Running Times and Compression Slowdown

		Execution Time [sec]				Compression Slowdown	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	102.3	101.82	122.81		1.00	1.20
fft	4	136.39	137.56	154.61		1.01	1.13
fft	8	140.43	137.34	158.64		0.98	1.13
radix	1	67.19	68.53	88.39		1.02	1.32
radix	4	106.97	107.52	129.86		1.01	1.21
radix	8	110.18	112.55	132.15		1.02	1.20
cholesky	1	131.39	131.74	264.54		1.00	2.01
cholesky	4	184.22	186.28	252.67		1.01	1.37
cholesky	8	214.44	207.05	259.33		0.97	1.21
radiosity	1	135.89	132.59	133.49		0.98	0.98
radiosity	4	235.09	236.67	231.47		1.01	0.98
radiosity	8	243.36	239.67	237.61		0.98	0.98
raytrace	1	176.93	174.17	181.07		0.98	1.02
raytrace	4	281.01	284.52	285.81		1.01	1.02
raytrace	8	289.37	282.13	296.94		0.97	1.03

Table 6.14 shows the number of cache accesses and the cache miss rate as well as the number of memory read operations and the first-access miss rate for the shared *mlvCFiat*. The results show that the data cache miss rates increases for benchmark runs when the number of threads is N = 4 and N = 8 relative to the miss rate observed in the private *mlvCFiat*. This is expected as the references from multiple threads now compete for the limited resources in the cache. Still, the cache miss rate remains relatively small in all benchmark runs except for *radix*, where it reaches 20.7% for N = 4 and 25.3% for N = 8. The shared *mlvCFiat* first-access miss rate also increases relative to the private *mlvCFiat* first-access miss rate. It ranges from 1.6% for *radiosity* with N = 4 to 21% for *cholesky* when N = 8.

Table 6.14. Shared *mlvCFiat* Cache and First Access Hit Rates

Bench-mark	**N**	**Number of CacheAcesses**	**CacheMiss-Rate**		**Number of Load Accesses**	**FA Mis Rate**
			[%]			**[%]**
fft	1	307,224,680	1.782		197,291,797	2.290
fft	4	307,413,525	2.587		194,943,784	6.117
fft	8	307,669,595	3.819		190,981,103	11.634
radix	1	116,642,518	13.182		67,680,804	11.541
radix	4	117,505,524	20.700		60,392,966	9.719
radix	8	118,818,621	25.339		56,320,482	11.409
cholesky	1	373,354,743	2.264		267,063,640	14.833
cholesky	4	391,682,559	2.571		295,844,475	16.556
cholesky	8	460,589,711	3.318		359,646,296	20.234
radiosity	1	533,009,818	0.527		390,946,458	0.867
radiosity	4	543,191,528	0.764		397,715,012	1.556
radiosity	8	542,213,475	1.599		393,746,820	3.999
raytrace	1	647,142,264	0.649		477,183,421	2.107
raytrace	4	647,148,563	2.693		465,870,745	8.280
raytrace	8	648,889,926	4.817		456,687,210	14.122

Table 6.15 shows the raw and compressed trace file sizes captured with the shared *mlvCFiat* for our benchmark runs. The last two columns show the compression ratio determined as the size of the raw shared *mlvCFiat* trace file divided by the size of the corresponding compressed shared *mlvCFiat* trace file (gzip and bzip2). When compared to the private *mlvCFiat* trace file size, the shared *mlvCFiat* generates larger trace file sizes when the number of threads is N = 4 and N = 8. This result is expected as the cache miss rate and the first access hit rate both increased for the shared *mlvCFiat*. Looking at compressability of traces generated by the shared

mlvCFiat for N = 4 and N = 8, we can observe that the compression ratio achieved by gzip and bzip2 in general lags behind the compression ratio achieved by the private *mlvCFiat*.

Figure 6.3 shows the trace file sizes expressed in bytes per executed instruction (Bytes/Ins) and in bytes per memory read operation (Bytes/Read). We can see that the number of bytes per executed instruction does not exceed 2 bytes, with a maximum observed for *cholesky*.

Table 6.15. Shared *mlvCFiat* Trace File Sizes

		OutputTraceSize [Bytes]				**Compression Ratio**	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	344,516,436	119,170,380	129,081,747		2.89	2.67
fft	4	584,411,446	298,416,289	288,514,246		1.96	2.03
fft	8	904,131,002	524,738,825	507,005,862		1.72	1.78
radix	1	382,843,113	154,292,674	124,414,035		2.48	2.39
radix	4	467,715,206	180,305,339	142,886,992		2.59	3.76
radix	8	547,826,231	205,499,558	160,391,829		2.67	3.83
cholesky	1	1,689,931,981	185,552,488	193,770,645		9.11	8.72
cholesky	4	1,649,797,130	274,290,421	217,698,002		6.01	7.58
cholesky	8	1,969,518,113	440,007,097	305,085,785		4.48	6.46
radiosity	1	57,535,418	12,707,221	7,540,207		4.53	7.63
radiosity	4	102,728,392	28,093,318	19,255,596		3.66	5.33
radiosity	8	252,636,840	71,238,996	42,514,386		3.55	5.94
raytrace	1	253,544,922	51,766,218	24,147,799		4.90	10.50
raytrace	4	948,341,506	280,543,654	137,608,864		3.38	6.89
raytrace	8	1,541,792,874	471,699,763	251,114,623		3.27	6.14

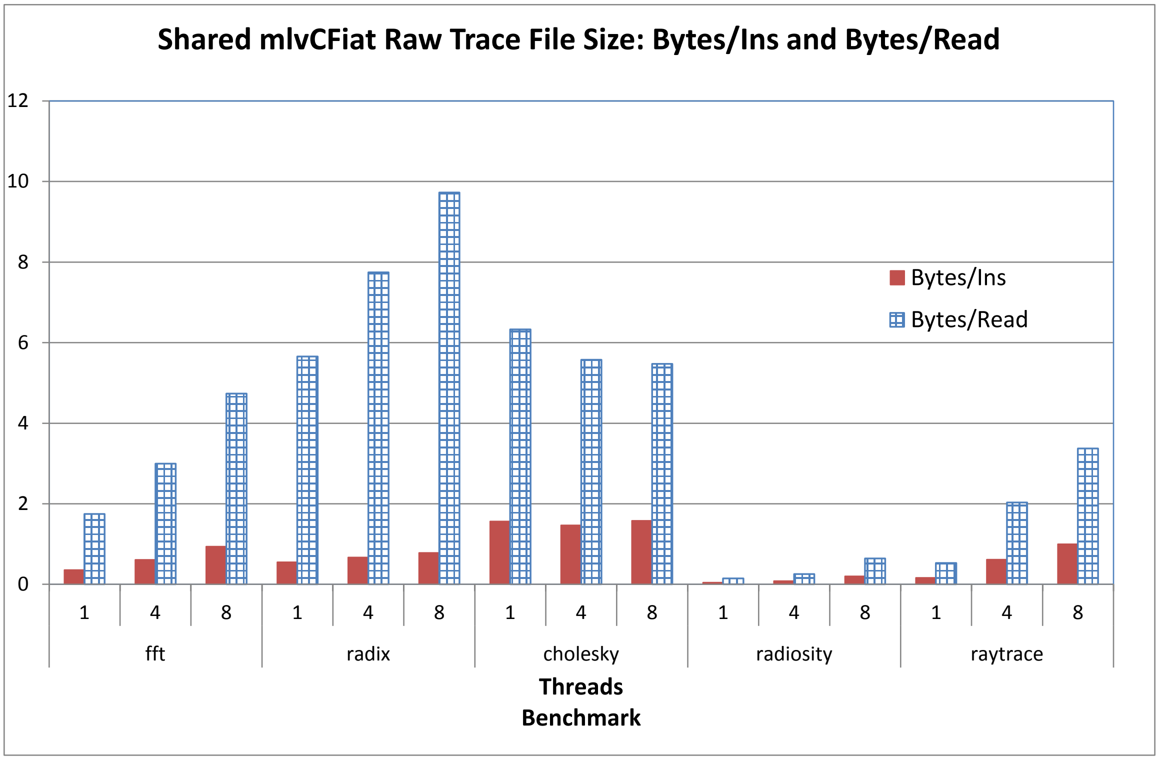

Figure 6.3 Trace File Sizes in Bytes/Ins and Bytes/Read for Shared *mlvCFiat*

Table 6.16 shows the execution times of the shared *mlvCFiat* for our benchmark runs. The last two columns show the slowdown when the shared *mlvCFiat* streams captured descriptors into the general-purpose compressors (gzip and bzip2). The execution times of the shared *mlvCFiat* are generally slightly longer than the execution times of the private *mlvCFiat*, but still shorter than the execution times of *mlsTrace*. The slowdown due to compression is negligible in case of the shared *mlvCFiat* when combined with gzip or bzip2.

Table 6.16 Shared *mlvCFiat* Running Times and Compression Slowdown

		Excution Time [sec]				Compression Slowdown	
Benchmark	**N**	**raw**	**gzip**	**bzip2**		**gzip**	**bzip2**
fft	1	127.66	132.7	153.59		1.04	1.20
fft	4	156.78	157.85	200.43		1.01	1.28
fft	8	158.46	177.37	229.02		1.12	1.45
radix	1	79.25	78.17	99.63		0.99	1.26
radix	4	110.13	111.09	140.38		1.01	1.27
radix	8	114.79	117.7	142.76		1.03	1.24
cholesky	1	162.99	171.83	294.2		1.05	1.81
cholesky	4	206.29	201.55	295.89		0.98	1.43
cholesky	8	231.53	239.78	334.41		1.04	1.44
radiosity	1	187.21	186.66	190.68		1.00	1.02
radiosity	4	247.71	245.41	249.59		0.99	1.01
radiosity	8	254.48	253.04	259.92		0.99	1.02
raytrace	1	242.67	239.49	253.41		0.99	1.04
raytrace	4	300.36	303.09	343.54		1.01	1.14
raytrace	8	313.45	344.41	363.58		1.10	1.16

CHAPTER 7

CONCLUSIONS

This research focuses on the development of a suite of binary instrumentation tools called mTrace. The mTrace tool suite includes four tools that support capturing and compressing control-flow and memory-reference traces of multithreaded software running on x86/Intel 64 computers under Intel's Pin dynamic binary instrumentation framework. The mTrace tools and traces generated by these tools are designed primarily to aide in the design and evaluation of hardware-based tracing mechanisms. However, they can be also used to aide software debugging as well as in trace-driven simulation of multi-core computer systems. Two of the tools, *mcfTrace* and *mlsTrace*, are used to analyze general control-flow and memory reference traces. *mcfTRaptor* seeks to reduce control-flow trace sizes for complete program replayability by exploiting branch prediction schemes for outcome and target prediction. *mlvCFiat* uses a cache extension to reduce load value trace sizes for complete program replayability. All four of these Pin tools can target multithreaded programs, with the option of organizing the predicition structures and cache extensions in *mcfTRaptor* and *mlvCFiat* privately per thread, or globally over all threads.

The mTrace tools are designed to allow a user to control program tracing by specifying the type of output trace file (binary or ASCII), the code segment to be traced (fast-forwarding or subroutine tracing), the optinal compression of captured traces using general-purpose compressors. In addition, *mcfTRaptor* and *mlvCFiat* allow a user to specify the configuration of predictor and cache structures. In addition to trace files, the mTrace tools generate an output file that contains extensive statistics on benchmark execution and efficiency of internal predictors in the *mcfTRTaptor* and *mlvCFiat* tools.

The mTrace tools are fully verified using a set of carefully crafted test programs that exercise various program and tool characteristics. The mTrace tools are used to generate program traces of SPLASH-2 parallel benchmark programs with N = 1, N = 4, and N = 8 threads. We evaluated the efficacy of the mTrace tools by analyzing the size of output trace files, execution times, and other metrics of interest, such as misprediction rates and cache miss rates at predictor structures. We find that additional compression using gzip usually does not impose an additional overhead in execution times for all considered tools. *mcfTRaptor* and *mlvCFiat* with private predictor and cache structures proved to be very effective in reducing the number of trace descriptors that needs to be recorded in the trace file.

Opportunities related to this work include enforcing te dynamic run-time behavior between executions of the target executable. Because the trace descriptors collected by these four tools are not perfect when compared to the native execution of the software, trace descriptor orderings will change between execution runs. Other changes may occur as well, including the location of shared libraries, system calls, the stack, and the heap in the virtual address space. Capturing that information will allow for complete and accurate replayability of the target program.

The mTrace tools and traces generated are available publicly and can be found at: http://lacasa.uah.edu/portal/index.php/software-data/32-mtrace-tools-and-traces.

CHAPTER 8

BIBLIOGRAPHY

[1] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," NIST, Research Triangle Park, 2002.

[2] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," IEEE, 2003.

[3] V. Uzelac, A. Milenkovic, M. Milenkovic and M. Burtscher, "Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing," *IEEE Transactions on Computers,* vol. PP, no. 99, p. 30, 2012.

[4] V. Uzelac and A. Milenkovic, "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging," *ACM Transaction on Embedded Computing Systems,* vol. 12, no. 2, p. 18, 2013.

[5] A. Milenkovic and M. Milenkovic, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Transactions on Modeling and Computer Simulation,* vol. 17, no. 1, January 2007.

[6] ARM, "CoreSight Trace Macrocells," ARM, [Online]. Available: http://arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php. [Accessed 2 March 2014].

[7] MIPS Technologies, "MIPS PDtrace Specification," MIPS Technologies, [Online]. Available: http://files.tomek.cedro.info/electronics/doc/mips/architecture/MD00439-2B-PDTRACETCB-SPC-06.16.pdf. [Accessed 3 March 2014].

[8] Infineon Technologies, "Microcontrollers On chip Debug Support," August 2001. [Online]. Available: http://www.infineon.com/dgdl/C166SV1-OCDS.pdf?folderId=db3a304412b407950112b41f1dd13613&fileId=db3a304412b407950112b41f1e303614. [Accessed 3 March 2014].

[9] A. Milenković, "mTrace," UAH, 4 February 2014. [Online]. Available: http://lacasa.uah.edu/portal/index.php/research/31-mtrace. [Accessed 3 March 2014].

[10] H. Patil, C. Pereira, M. Stallcup, G. Lueck and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," in *IEEE/ACM international symposium on Code generation and optimization*, New York, NY, 2010.

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

[12] E. Johnson, "PDATS II: Improved Compression of Address Traces," in *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, 1999.

[13] J. R. Larus, "Whole Program Paths," in *Proceedings of the ACM SIGPLAN*

1999 conference on Programming language design and implementation, Atlanta, GA, 1999.

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

[15] M. Burtscher, "VPC3: A Fast and Effective Trace-Compression Algorithm," *ACM SIGMETRICS Performance Evaluation Review,* vol. 32, no. 1, pp. 167-176, June 2004.

[16] M. Burtscher, "TCgen 2.0: A Tool to Automatically Generate Lossless Trace Compressors," *ACM SIGARCH Computer Architecture News,* vol. 34, no. 3, pp. 1-8, June 2006.

[17] A. Milenkovic, A. Uzelac, M. Milenkovic and M. Burtscher, "Caches and Predictors for Real-time Unobtrusive, and cost-effective Program Tracing in Embedded Systems," *IEEE Transactions on Computers,* vol. 60, no. 7, pp. 992-1005, 2011.

[18] M. R. Guthaus, T. M. Austin, R. B. Brown, J. D. Ernst, S. J. Ringenberg and N. T. Mudge, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, Austin, 2001.

[19] V. Uzelac and A. Milenkovic, "A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method," in *Proceedings of the 46th Annual Design Automation Conference*, 2009.

[20] J. L. Bentley, "A Locally Adaptive Data Compression Scheme," *Commun. ACM,* vol. 29, no. 4, pp. 320-330, 1986.

[21] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," September 2013. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf. [Accessed 10 December 2013].

[22] Intel, "Pintool: A framework for Dynamic Binary Instrumentation," Dec. 2009. [Online]. Available: http://www.pintool.org.

[23] J.-L. Gailly. [Online]. Available: http://www.gzip.org/. [Accessed 06 02 2014].

[24] J. Seward. [Online]. Available: http://www.bzip.org/. [Accessed 06 02 2014].

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, "The SPLASH-2 Programs: Characerization and Methodological Considerations," in *22nd Annual International Symposium*, 1995.

[26] D. H. Bailey, "FFTs in External or Hierarchical Memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* , 1990.

[27] R. A. Uhlig and T. N. Mudge, "Trace-drivien Memory Simulation: A Survey," *ACM Computing Surveys,* vol. 29, no. 2, pp. 128-170, June 1997.

[28] T. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[29] K. Sayood, Introduction to Data Compression, 3rd ed. ed., Morgan Kauffman,

2005.

[30] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory,* vol. 23, no. 3, pp. 337-343, May 1977.

[31] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the I.R.E.,* vol. 40, no. 9, pp. 1098-1101, September 1952.

[32] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Palo Alto, CA, 1994.

[33] J. Seward, February 2010. [Online]. Available: http://www.bzip.org/.

[34] C. G. Nevill-Manning and I. Witten, "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research,* vol. 7, no. 1, pp. 67-82, September 1997.

[35] A. D. Samples, "Mache: No-Loss Trace Compression," Berkeley, CA, 1988.

[36] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *IEEE Computer Architecture Letters,* vol. 1, no. 1, pp. 9-12, January 2002.

[37] T. Moseley, D. Grunwald and R. Peri, "Seekable Compressed Traces," in *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, 2007.

[38] M. Burtscher, July 2006. [Online]. Available: http://www.csl.cornell.edu/~burtscher/research/TCgen/.

[39] M. Burtscher and N. Sam, "Automatic Generation of High-Performance Trace Compressors," in *Proceedings of the international symposium on Code generation and optimization*, San Jose, CA, 2005.

[40] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 2001.

[41] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, New York, 1990.

[42] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid and H. Meyr, "Fine-grained application source code profiling for ASIP design," in *Proceedings of the 42nd annual Design Automation Conference*, Anaheim, CA, 2005.

[43] A. R. Lebeck and D. A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE Computer,* vol. 27, no. 10, pp. 15-26, October 1994.

[44] New Mexico State University, November 2002. [Online]. Available: http://tracebase.nmsu.edu/tracebase.html.

[45] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News,* vol. 34, no. 4, pp. 1-17, September 2006.

[46] C. Erbas, A. D. Pimentel, M. Thompson and S. Polstra, "A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures," *EURASIP Journal on Embedded Systems,* vol. 2007, no.

1, p. 2, January 2007.

[47] M. Milenkovic, S. T. Jones, F. Levine and E. Pineda, "Performance Inspector Tools with Instruction Tracing and Per-Thread / Function Profiling," in *Proceedings of the Linux Symposium*, Ottawa, 2008.

[48] "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN*, New York, 2005.

[49] F. S. Foundation, "GCC, the GNU Compiler Collection," [Online]. Available: http://gcc.gnu.org/. [Accessed 06 02 2014].

[50] X. Zhang, N. Gupta and R. Gupta, "Whole Execution Traces and Their Use in Debugging," in *The Compiler Design Handbook*, Boca Raton, Florida, CRC Press, 2002, pp. 4-4.

[51] M. L. Soffa, K. Walcott and J. Mars, "Exploiting Hardware Advances for Software Testng and Debugging," in *ICSE*, Honolulu, HI, 2011.